Introduction to GitHub Concepts

Description

Do you ever struggle to keep track of everything going on in a project? Or confuse GitHub with Git? In this course, you'll learn how to leverage the power of GitHub, become a successful collaborator, and recognize the differences between GitHub and Git.

Building on the topics covered in Introduction to Version Control with Git, this conceptual course enables you to navigate the user interface of GitHub effectively.

You will perform everyday tasks, including creating public and private repositories, creating and modifying files, branches, and issues, assigning tasks, tagging users, reviewing pull requests, and merging branches. You will also discover how to clone and fork repositories and generate private access tokens (PAT).

By the end of this course, you'll be able to take these new skills and apply them to any coding or data project, making you feel on track and in control. Everyone will want to collaborate with you on GitHub!

Course Notes

Chapter 1: Introduction to GitHub

1.1 What is GitHub?

- **Main Definition:** GitHub is a **cloud-based hosting service**.
- **Purpose:** It is used for users to upload and **track their work**. Usually, this work is code-based.
- **Version Control:** This "tracking" of work is formally known as **version control**.
- **Extended Understanding of Version Control:** Version control is essentially a time machine for your projects. Imagine being able to see every single change you've made to a document, who made it, when they made it, and why. More importantly, you can revert back to any previous state if something goes wrong. This becomes invaluable when you're working on complex projects where a single mistake could break everything, or when multiple people are editing the same files simultaneously.
- **Meaning of "Cloud-Based":** GitHub provides on-demand resources (like storage space) to its users over the internet. This prevents us from having to keep large project

files on our own computers.

- **The Practical Advantage of Cloud Storage:** Beyond just saving space on your local machine, the cloud-based nature of GitHub means your work is accessible from anywhere in the world. If your laptop crashes, your work is safe. If you need to switch computers mid-project, you simply access GitHub from the new machine. This also means natural disaster recovery is built-in, something that would be much more complicated if you were managing version control entirely on local servers.
- **Popularity:** While alternatives like GitLab and BitBucket exist, **GitHub is the most popular** platform of its kind.
- **Why GitHub's Popularity Matters:** GitHub's dominance in the market means that when you learn GitHub, you're learning the tool that the vast majority of the open-source community uses. This translates to better community support, more tutorials and resources, and a larger network of potential collaborators. Many employers specifically look for GitHub experience because it's become the de facto standard for collaborative software development.

Uses and Benefits of GitHub

- **Storage and Collaboration:** Its main uses are **storing and tracking projects** and, fundamentally, **collaborating** with other people.
- **Social Network:** It also functions as a kind of **social network**, allowing users to connect with each other.
- **Understanding GitHub as a Professional Network:** Think of your GitHub profile as a living portfolio. Unlike a traditional resume that lists what you say you can do, your GitHub profile shows what you have actually built. Recruiters and hiring managers often review candidates' GitHub profiles to assess their coding style, their ability to collaborate, and the types of projects they've worked on. Following other developers, starring repositories you find interesting, and contributing to open-source projects all become part of your professional identity in the tech world.
- **Open-Source:** GitHub hosts a vast number of **open-source projects** (public projects) that anyone can use to learn, practice, or even contribute to by editing them.
- **The Learning Opportunity in Open-Source:** Open-source projects on GitHub represent an unparalleled learning resource. You can read the code of world-class software written by expert developers. You can see how large projects are structured, how teams communicate through issues and pull requests, and how bugs are identified and fixed. For beginners, contributing to open-source projects (even just fixing typos in documentation) is one of the best ways to gain real-world experience and build your reputation in the developer community.
- **Solo Projects:** It is also very beneficial for individual (solo) projects, as it provides a **complete history of every project stage**.
- **The Value of History for Solo Developers:** Even when working alone, having a detailed history of your project is incredibly valuable. You might make a change that seems like a good idea at the time, only to realize a week later that it broke something

subtle. With version control, you can easily identify exactly when the problem was introduced and what changed. This historical record also helps you understand your own evolution as a developer—looking back at code you wrote six months ago often reveals how much you've learned.

The Key Difference: GitHub vs. Git

This is a crucial point for understanding the platform:

- **Git:**
  - It is the **version control software** itself.
  - It can be used entirely **independently** of GitHub (e.g., locally or with other platforms).
  - **Git's Core Functionality:** Git operates entirely on your local computer and doesn't require an internet connection. You can create repositories, make commits, create branches, and manage your entire version control workflow offline. Git was created by Linus Torvalds (the creator of Linux) in 2005 as a distributed version control system, meaning every developer has a complete copy of the entire project history on their local machine. This distributed nature makes it incredibly robust and fast.
- **GitHub:**
  - It is a **platform that enhances and facilitates the use of Git**.
  - It makes managing projects and collaboration much easier than using Git alone.
  - **GitHub's Enhanced Features:** While Git provides the fundamental version control capabilities, GitHub adds a beautiful web interface, issue tracking, project management tools, continuous integration capabilities, code review features, and much more. Think of Git as the engine and GitHub as the entire car with all its comfort features, dashboard, and navigation system. You need the engine to move, but the additional features make the journey much more pleasant and productive.
  - **Dependency:** GitHub is **completely dependent on Git**. You cannot use GitHub without Git.
  - **The Technical Relationship:** Every GitHub repository is fundamentally a Git repository hosted on GitHub's servers. When you interact with GitHub through the web interface (creating files, making commits, merging branches), GitHub is actually executing Git commands on your behalf in the background. Understanding this relationship helps you appreciate that everything you do on GitHub could theoretically be done with Git commands in the terminal, but GitHub makes it much more accessible and user-friendly.

Collaboration and Repositories (Repos)

- **Easing the Workflow:** GitHub allows the entire version control workflow (tracking changes, allowing multiple people to work on the same files) to happen directly on the web platform, instead of having to use Git via the Command Line Interface (CLI).

- **The Accessibility Advantage:** Before platforms like GitHub, using Git required comfort with the command line, which presented a barrier to entry for many people, especially those from non-technical backgrounds. By providing a graphical interface, GitHub democratized access to version control. Data scientists, designers, writers, and other professionals can now leverage the power of version control without needing to master terminal commands first, though learning the command line remains valuable for more advanced workflows.
- **Universal Access:** Since the project is stored in the cloud, **any team member can access it**, making collaboration extremely easy.
- **The Synchronization Benefit:** One of the most powerful aspects of having your repository in the cloud is that everyone is always looking at the most up-to-date version of the truth. Gone are the days of emailing files back and forth with names like "final_version_v3_FINAL_really_final.docx". With GitHub, there's one canonical version of the project, and everyone can see exactly what state it's in at any moment. When someone makes a change, everyone else can pull that change down to their local machine and stay synchronized.
- **Repository (Repo):**
  - A project is stored in a "repository" (or "repo").
  - A repository contains **all the project's files** (code, data, etc.) and also the history of all past versions of those files (stored in a `.git` file).
  - **Understanding the `.git` Directory:** Technically, it's a `.git` directory (a hidden folder), not just a single file. This directory contains the entire history and metadata of your project. It stores every commit you've ever made, every branch you've created, and all the information Git needs to reconstruct any previous state of your project. This is why a Git repository can sometimes be larger than the current project files—it's storing the entire historical record. The beauty of this system is that this complete history travels with your repository, so whether you're working on GitHub's servers or your local machine, you have access to the full project timeline.
- **Remote vs. Local Repository:**
  - **Remote Repo:** This is the repository stored on the internet—in this case, on GitHub.
  - **Local Repo:** This is the copy of the repository that is saved on a user's local computer.
  - **The Synchronization Dance:** Understanding the relationship between remote and local repositories is fundamental to effective Git and GitHub usage. Think of the remote repository as the "source of truth" that everyone agrees upon, while your local repository is your personal workspace. You pull changes from the remote to update your local copy, and you push changes from your local to share them with others via the remote. This separation allows you to work offline, experiment freely in your local environment without affecting others, and only share your work when you're ready. The commands `git pull` (to receive updates) and `git push` (to send updates) become the bridges between these two worlds.

1.2 Setting up a repo

How to Create a New Repository

There are two main ways to start from the GitHub interface:

- **Option 1:** Click the **plus sign (+)** in the top-right corner and select **"New repository"**.
- **Option 2:** Go to your **"Repositories"** tab and click the green **"New"** button.

Both options lead to the "Create a new repository" page. On this page, you can import an existing repo or use a template, but for this course, we are creating one **from scratch**.

Initial Setup Steps

When creating a new repository, you must configure several key items:

- **Name:** The first step is to name your repository (e.g., "soccer-analysis").
- **Naming Best Practices:** Repository names should be descriptive, concise, and use hyphens or underscores instead of spaces. Lowercase is conventional. A good name immediately tells someone what the project is about. For example, "ml-weather-prediction" is better than "project1" or "stuff". Remember that this name becomes part of your repository's URL, so choose something you won't be embarrassed to share professionally.
- **Description:** This is optional but highly recommended, especially for collaborative projects, as it explains what the project is about.
- **Crafting Effective Descriptions:** While optional, skipping the description is a missed opportunity. A good description is one to two sentences that clearly states the project's purpose. For example, "Machine learning model to predict housing prices using Python and scikit-learn" immediately tells visitors exactly what to expect. This description appears in search results and on your profile, making it easier for others to discover and understand your work. Think of it as the elevator pitch for your project.
- **Visibility (Public vs. Private):**
  - **Public:** The project will be **visible to anyone** on the internet.
  - Important: Even if it's public, you (the owner) **still control who can make changes** (commits) to the project.
  - **The Open-Source Default:** Public repositories are the heart of GitHub's open-source mission. When you make a repository public, you're contributing to the global knowledge base and potentially allowing others to learn from, use, or improve your work. This doesn't mean giving up control—you remain the gatekeeper who decides which proposed changes (pull requests) get accepted. Many developers keep their learning projects and portfolio pieces public to showcase their skills, while keeping client work or proprietary code private.

Initializing Essential Files

Before finishing, it is crucial to initialize the repository with fundamental files:

- `README` **file:**
  - It is considered **standard practice** to include one in all repositories.
  - It acts as a **guide for the project**. It is vital for collaboration as it provides details on what the project is about and how it can be used.
  - **Why the README is the First Thing People See:** The README file is automatically rendered and displayed on your repository's main page. This makes it the landing page for your project. A visitor should be able to read your README and understand within 30 seconds what your project does, why it exists, and how to get started with it. A repository without a README is like a book without a cover or table of contents—technically functional, but significantly less accessible and professional.
- `.gitignore` **file:**
  - **Purpose:** It acts like a "block list." It prevents specific files from being saved (committed) into the repository.
  - **Common Use:** It is used to ignore files containing **confidential information** (like passwords or API keys) or unnecessary system files.
  - **Understanding What Should Be Ignored:** Beyond secrets and credentials, you'll want to ignore files that are generated automatically (like compiled code or build artifacts), files specific to your development environment (like IDE configuration files), and operating system files (like MacOS's `.DS_Store` or Windows' `Thumbs.db`). Large data files or binary files that change frequently are also good candidates for `.gitignore`. The key principle is this: your repository should contain the source material needed to recreate your project, not every temporary or derived file that gets created along the way.
  - **Templates:** GitHub offers pre-configured templates for different languages (the **Python** template is selected in the video).
  - **The Value of Language-Specific Templates:** These templates are created based on community best practices and include common patterns for each language. For Python, the template includes patterns for virtual environments (`venv/`, `env/`), compiled Python files (`*.pyc`, `__pycache__/`), and common tool artifacts. Starting with a template saves you from having to remember or research what should be ignored. You can always add more patterns later as you discover files specific to your project that shouldn't be tracked.
- **License:**
  - In this case, **"None"** is selected.
  - This means that **default copyright laws apply**.
  - **The Legal Implications of "None":** Choosing no license is actually quite restrictive. Under default copyright law, you retain all rights, and others technically cannot use, copy, modify, or distribute your code without your explicit permission,

even if it's in a public repository. If you want others to be able to use your work (which is common for open-source projects), you should choose a license. Popular options include MIT (very permissive), Apache 2.0 (includes patent rights), and GPL (requires derivative works to also be open-source). If you're unsure, websites like choosealicense.com can help you understand the differences.

Navigating the Created Repo

Once you hit the "Create repository" button, you are taken to the main view of the new repo, which includes several key tabs:

- **"Code" section:**
  - This is the main view where all the project's **folders and files** are visible (including the newly created `README` and `.gitignore`).
  - **Understanding the Code View:** This is where you'll spend most of your time. The file browser shows your repository's structure, with folders and files organized hierarchically. You can click on any file to view its contents, and folders to navigate deeper into the structure. The most recent commit message for each file is shown next to it, giving you quick context about what was last changed and why. This view also shows the number of commits, branches, and contributors, giving you a quick health check of project activity.
- **"About" section:**
  - Located on the right-hand side, this displays the **description** added during creation.
  - It can be edited by clicking the gear (⚙) icon.
  - **Maximizing the About Section:** This section can include much more than just your description. You can add tags (topics) to help people discover your project, add your project's website URL, and check boxes to indicate if you're seeking contributors. These metadata elements make your repository more discoverable in GitHub's search and help set expectations for visitors.
- **"Issues" tab:**
  - This is the place to **track tasks, bugs, or problems** within the project.
  - It is also used to **communicate** with others about these items.
  - **Issues as Project Management:** Think of the Issues tab as a combination of a to-do list, bug tracker, and discussion forum. Each issue can be assigned to team members, labeled with categories (like "bug", "enhancement", or "documentation"), and tracked through its lifecycle from open to closed. Issues support markdown formatting, can reference other issues and pull requests, and maintain a chronological discussion thread. Many teams use issues to plan features, track known problems, and coordinate work. The ability to close issues with specific commits (by including "fixes #123" in a commit message) creates automatic documentation of how problems were resolved.
- **"Pull Requests" (or PRs) tab:**
  - **Definition:** A **request to make a change** to the project.

    - **Analogy:** Think of it like a "suggestion box."
    - **Function:** A PR shows the suggested changes and **compares them to the project's current version**, allowing the owner to review and decide whether to accept (merge) the changes.
    - **The Pull Request as a Review Process:** Pull requests are one of GitHub's most powerful features for collaboration and code quality. When someone opens a PR, they're not just proposing changes—they're starting a conversation. Reviewers can comment on specific lines of code, suggest improvements, request changes, or approve the PR. The PR interface shows exactly what will change (a "diff"), allows for automated testing to run, and creates a permanent record of why a change was made and what alternatives were considered. This review process catches bugs, improves code quality, facilitates knowledge sharing, and ensures that everyone understands significant changes before they're merged into the main codebase.
- **"Settings" tab:**
    - This allows you to make various changes to your repo, such as **changing the name** or managing **access permissions**.
    - **The Power and Danger of Settings:** The Settings tab is where you have administrative control over your repository. Beyond name changes and access control, you can configure branch protection rules, set up webhooks for automation, manage GitHub Actions for continuous integration, configure GitHub Pages for hosting websites, and much more. The settings area also includes the "Danger Zone" at the bottom, where you can transfer ownership, archive the repository, or delete it permanently. These are powerful tools that should be used thoughtfully, especially on collaborative projects where changes might affect multiple people.

1.3 Creating a README

How to Edit the README File

- **Location:** The `README.md` file is always displayed at the bottom of the main **"Code"** section of your repository, just below the list of files.
- **File Type:** It is a **Markdown file**, which is indicated by the `.md` extension.
- **Why Markdown?** Markdown was chosen because it strikes a perfect balance between human readability and formatting capability. A markdown file is just plain text with simple markup symbols, so it can be read and edited in any text editor, but it can also be rendered into beautifully formatted HTML. This makes it ideal for documentation—the raw file is readable by humans, but it also displays nicely on the web. Markdown has become the standard for README files, documentation, and even note-taking across the tech industry.
- **Editing:** To edit the file, scroll to the rendered README view and click the **pencil icon** located in its top-right corner.

- **Edit vs. Preview:**
  - **Edit:** This tab shows the raw Markdown syntax.
  - **Preview:** This tab shows how the formatted content will look. You can toggle between these to check your syntax as you work.
  - **The Iterative Process of Writing Markdown:** When you're first learning Markdown, the Preview tab becomes your best friend. You'll find yourself switching back and forth frequently as you learn which syntax produces which result. Over time, you'll become comfortable enough to write markdown without previewing, but even experienced users toggle to preview when formatting tables or checking how images will appear. This workflow mirrors how you might work in any WYSIWYG editor, but with the added benefit that your source format remains simple and portable.

Markdown Syntax Fundamentals

Markdown is a lightweight syntax used to format plain text.

- **Headings:**
  - Use hashtags `#` to create headings.
  - `#` (one hashtag) is the largest title.
  - `######` (six hashtags) is the smallest available heading.
  - **Heading Hierarchy and Document Structure:** Headings in Markdown create a structured outline for your document. Use `#` for your main title, `##` for major sections, `###` for subsections, and so on. This hierarchy isn't just visual—it creates a logical structure that can be used to generate a table of contents automatically. A well-structured document with proper heading levels is easier to scan and navigate. Think of headings as the skeleton of your document that helps readers quickly find the information they need.
- **Text Formatting:**
  - **Bold:** Wrap text in two asterisks `**text**`.
  - *Italic:* Wrap text in one asterisk `*text*`.
  - **When to Use Emphasis:** Bold and italic formatting should be used purposefully, not decoratively. Bold is excellent for emphasizing key terms when they're first introduced, highlighting critical warnings or requirements, or making important numbers stand out. Italics work well for technical terms, file names, or for adding subtle emphasis to a phrase. Overuse of either dilutes their impact—if everything is bold, nothing stands out.
- **Hyperlinks (Links):**
  - **Syntax:** `[Visible Text](https://your-url.com)`
  - The text in the square brackets `[]` becomes the clickable link.
  - The URL in the parentheses `()` is the destination.
  - **Writing Meaningful Link Text:** The text you put in the square brackets matters significantly for usability and accessibility. Never write "click here" as your link text.

Instead, make the descriptive text itself the link. For example, instead of "To learn more, click here", write "Learn more about GitHub Actions in the official documentation." This makes your content more accessible to screen readers and helps readers know where the link will take them before they click.

- **Images:**
  - **Syntax:** `![Alternative text](image-url.png)`
  - This is similar to a link but starts with an **exclamation mark** `!`.
  - **Alternative Text:** The text in the `[]` is crucial for **accessibility**. It describes the image for screen readers or if the image fails to load.
  - **Writing Effective Alt Text:** Alternative text is not optional—it's an accessibility requirement. Good alt text describes the content and function of the image. For a chart, you might write "Bar chart showing monthly sales increasing from January to December." For a screenshot, "GitHub interface showing the Pull Requests tab." Be specific and concise. Alt text helps visually impaired users understand your content and improves the experience for everyone when images fail to load due to slow connections.
  - **Drag and Drop:** You can simply drag and drop an image file from your computer directly into the text editor, and GitHub will automatically upload it and generate the correct Markdown syntax.
  - **Understanding GitHub's Image Hosting:** When you drag and drop an image, GitHub uploads it to its own image hosting service and generates a permanent URL for that image. This URL is what gets inserted into your markdown. This means you don't need to manage image hosting separately—GitHub handles it for you. The image becomes part of your repository's assets and will remain available as long as the repository exists.

What Should a README Include?

The README is the **"instruction manual"** for your repository. It's often the first file a person will see, and its purpose is to allow anyone to understand your project, its contents, and how to use it.

- **Essential Items:**
  - Project title
  - A description of the project
  - Technology used (and why you chose it)
  - The process used to answer the project's question (and why)
  - A table of contents (for larger projects)
  - **Building a Narrative:** The best READMEs tell a story. They start with the problem the project solves, explain the approach taken, guide the reader through setup and usage, and acknowledge limitations or future plans. Think of your README as an invitation to explore your work. Each section should answer a question your reader might have: What is this? Why should I care? How do I use it? What are the

results? This narrative structure makes your documentation engaging rather than just informational.

- **Characteristics of a *Good* README (Extras):**
  - The motivation behind the project (how it came about).
  - Any limitations or challenges encountered.
  - A recap of the problem the project intends to solve.
  - The project's intended use.
  - **Credits:** Always include necessary credits if you are sourcing information or code from elsewhere.
  - **The Professional Polish:** These "extra" elements elevate your README from functional to exceptional. Discussing motivation shows your thought process. Acknowledging limitations demonstrates intellectual honesty and helps set appropriate expectations. Recapping the problem provides context for your solution. Explaining intended use helps prevent misunderstandings about your project's scope. Credits and acknowledgments are both ethically important and build community—they show respect for others' work and contribute to the culture of open source collaboration. These elements together paint a complete picture of your project and demonstrate professionalism and maturity as a developer.

Chapter 2: Working with Repos

2.1 Modifying a repo structure

How to Create a New File

- **Action:** Go to the "Code" section of your repo, click the `Add file` button, and select `Create new file`.
- **Naming:** Type the full filename in the top box, making sure to include the **file extension** (e.g., `requirements.txt`).
  - **The Importance of File Extensions:** File extensions aren't just labels—they tell both humans and computers what type of content to expect. The extension determines which program opens the file, how syntax highlighting works in editors, and how GitHub displays the file. For example, `.md` files get markdown rendering, `.py` files get Python syntax highlighting, and `.json` files get JSON formatting. Always use the correct extension for your file type. For Python scripts, use `.py`. For configuration files, use the appropriate extension like `.yaml`, `.json`, or `.config`.
- **Editing:** Add the desired content into the built-in editor.
  - **GitHub's In-Browser Editor:** The web editor includes basic features like syntax highlighting and indentation. While it's convenient for quick changes, it lacks the

advanced features of a dedicated code editor like auto-completion or linting. For simple edits or creating configuration files, the web editor works perfectly. For more complex coding work, you'll likely prefer to clone the repository and use your local development environment.

- **Saving:** See the "Committing the File" section below.

How to Upload an Existing File

- **Action:** If you have a file on your local computer, click the `Add file` button and select `Upload files`.
- **Process:** You can then **drag and drop** the file (or files) directly onto the page.
    - **Batch Upload Capability:** One particularly useful feature is that you can drag multiple files at once, or even entire folders. This makes it efficient to add several related files to your repository in one operation. However, remember that each upload counts as a single commit, so if you're adding many files, consider whether they logically belong together in one commit or should be added separately with distinct commit messages.
- **Saving:** See the "Committing the File" section below.

Committing the File (Saving Changes)

Committing is the act of saving your changes to the repository's history.

- **Commit Message:** At the bottom of the page, add a **brief, descriptive message** in the first text box (e.g., "Add requirements file").
    - **The Art of Good Commit Messages:** A commit message is a note to your future self and your collaborators. It should answer the question "What does this commit do?" Good messages use the imperative mood, as if giving a command: "Add feature" not "Added feature" or "Adding feature." Keep the first line under 50 characters if possible—this is the subject line. For more complex changes, use the extended description box to explain the why behind your change. A commit message like "Fix bug" is almost useless, while "Fix off-by-one error in date calculation for leap years" tells a complete story. Remember, months from now when investigating a bug, these messages will be your only clues about what changes were made and why.
- **Extended Description (Optional):** You can add more details in the second box, but it's often left blank for simple changes.
    - **When to Use Extended Descriptions:** Use the extended description field when the "what" is clear but the "why" needs explanation. For example, if you're changing how a calculation works, the commit summary might be "Refactor revenue calculation method," and the extended description could explain "Previous method didn't account for international tax differences. New approach uses a lookup table for accuracy across regions." This additional context is invaluable during code reviews and when tracing the history of a particular decision.

- **Commit Options:** You have two choices:
  1. **Commit directly to the current branch:** This is the most common approach for small, direct changes to your own repo.
  2. **Create a new branch for this commit and start a pull request:** This is used in collaborative workflows (covered later).
     - **Understanding When to Use Each Option:** Committing directly to a branch is quick and efficient for straightforward changes, especially when working alone or on a development branch. However, creating a new branch and opening a pull request is the safer, more professional approach for several reasons. It allows changes to be reviewed before being merged, keeps the main branch stable, enables automated testing, and creates a discussion thread around the change. In professional settings, directly committing to the main branch is often prohibited by branch protection rules. The rule of thumb is: if the change is trivial and you're certain it's correct, commit directly to your working branch. If the change is significant, affects others, or you want feedback, use a pull request.
- **Result:** After committing, GitHub returns you to the "Code" section, where your new or updated file will be visible.

How to Create a New Directory (Folder)

GitHub has a specific trick for creating directories, as it **does not allow empty directories**.

- **Action:** Click `Add file` > `Create new file`.
- **Process:** In the filename box, type the **name of your directory** followed by a **forward slash** `/`.
  - *Example:* To create a directory named `EDA`, you would type: `EDA/`
  - **Why Git (and GitHub) Won't Track Empty Directories:** This limitation exists because Git tracks file content, not directories themselves. In Git's internal model, directories emerge implicitly from the files they contain. This is actually by design—Git is optimized for tracking changes to code, and empty directories don't contribute to that goal. Other version control systems handle this differently, but Git's approach keeps the repository structure lean and focused on content.
- **Add a File:** The forward slash will move you into the new directory. You must *immediately* create a file inside it.
- **Placeholder:** It's common practice to add a placeholder file, like a `README.md`, which you can add details to later.
  - **The `.gitkeep` Convention:** While not mentioned in the original notes, there's a common convention in the Git community to create a file called `.gitkeep` as a placeholder. This file is typically empty and serves no purpose other than to make Git track the directory. It's not a Git feature (Git doesn't recognize `.gitkeep` specially), but it's a naming convention that signals intent: "I want this directory to exist, even though it's currently empty." You could use any filename, but `.gitkeep`

has become the standard. Alternatively, a `README.md` that explains the directory's intended purpose is even more useful.

- **Commit:** Save this new file (e.g., with the message "Add EDA directory") to create the directory.

How to Modify an Existing File

- **Action:** Navigate to and **click on the file** you want to modify (e.g., `requirements.txt`).
- **Editing:** Click the **pencil icon** (✏️) in the file view to open the editor.
- **Process:** Add or remove content as needed.
- **Commit:** Save your changes with a new commit message (e.g., "Add seaborn to requirements").
    - **Tracking Changes Through Commits:** Every time you commit a modification, GitHub preserves the previous version. This creates a chain of changes over time. You can click on a file's "History" button to see every commit that affected that file, view exactly what changed in each commit (the diff), and even blame-view to see who wrote each line. This historical record is invaluable for understanding how a file evolved, troubleshooting when bugs were introduced, and learning from past decisions.

How to Delete a File

- **Action:** Navigate to and **click on the file** you want to delete (e.g., `.gitignore`).
- **Process:** In the file view, click the **bin/trash can icon** (🗑️) located in the top-right corner (next to the pencil icon).
- **Commit:** Confirm the deletion by saving it as a new commit.
    - **Deletion is Not Permanent in Git:** Here's a crucial concept that separates Git from a regular file system. When you delete a file and commit that deletion, the file disappears from the current state of the repository, but it still exists in the history. If you later realize you need that file, you can retrieve it from an earlier commit. This safety net means you can confidently delete files that seem unnecessary—they're always recoverable if you need them. The file is removed from the working tree but preserved in the repository's history.

2.2 Working with branches

What Are Branches?

- **Purpose:** Branches are used to allow for **concurrent work** on different parts of a project simultaneously.
- **Benefit:** They reduce the risk of creating **conflicting versions** of files.

- **Example:** You might analyze Spanish soccer data in a branch named `la-liga`, while a colleague analyzes English data in a `premier-league` branch.
    - **The Power of Isolation:** Branches create isolated environments where you can experiment, develop features, or fix bugs without affecting the stable main codebase. Imagine branches as parallel universes for your code. In one universe (the `main` branch), your project is stable and working. In another universe (a `feature` branch), you're implementing a risky new feature. If the feature works, you merge that universe into main. If it fails, you simply abandon that branch without ever having compromised the working code in main. This isolation is what makes branches so powerful—they enable fearless experimentation and parallel development.
- **Default Branch:** When a repository is created, it has one default branch, which is named `main`.
    - **The Shift from "Master" to "Main":** If you've used Git before 2020, you might remember the default branch being called "master." In 2020, the tech community moved to rename this to "main" as part of a broader effort to remove unnecessary references to slavery from technical terminology. GitHub and Git both adopted "main" as the new default. Older repositories might still use "master," but all new repositories use "main." These are functionally identical—just different names for the same concept.

Creating and Switching Branches

- **Viewing Branches:**
    - You can see which branch you are currently in by looking at the branch icon in the **"Code"** section.
    - Clicking this icon lists all available branches. The **"Default"** branch is indicated.
- **Creating a New Branch:**
    1. Click the branch icon and then click the **"New branch"** button.
    2. In the pop-up window, enter a descriptive **branch name** (e.g., `la-liga`, `dev`, or `testing`).
        - **Branch Naming Best Practices:** Branch names should be descriptive and follow a consistent convention across your team. Common patterns include feature-based names like `feature/user-authentication` or `add-payment-processing`, bug-fix names like `bugfix/fix-login-error` or `hotfix/security-patch`, and task-based names like `update-documentation` or `refactor-database-queries`. Some teams include ticket numbers from their project management system, like `feature/123-add-search-functionality`. Avoid vague names like `test` or `fix`—future you will have no idea what these branches were for.
    3. Choose the **Branch source** (the branch that the new one will be copied from). Since we only have `main`, it is the source by default.
        - **Understanding Branch Ancestry:** When you create a new branch from a source branch, you're creating a complete copy of that branch's current state. The new

branch starts with identical content to its source. Think of it like making a photocopy of a document before making edits—you now have two versions, and you can modify one without affecting the other. The source branch you choose matters because your new branch will contain all the commits and current state of that source branch at the moment of branching.

4. Select **"Create branch"**.

- **Switching Branches:**
  - Click the current branch name to see the dropdown list of all active branches.
  - Select the name of the branch you want to switch to (e.g., select `main` to switch back to it).
  - **What Happens When You Switch:** Switching branches is one of Git's most magical features. When you switch from one branch to another, GitHub (or Git on your local machine) instantly updates the entire repository to reflect that branch's state. Files that exist in one branch but not another will appear or disappear. Contents of files will change to match the branch's version. It's like stepping between those parallel universes mentioned earlier. This switching happens almost instantaneously because Git is just updating pointers—it's not actually copying all the files.
- **Branch Isolation:**
  - Files added or modified in one branch **only exist in that branch**.
  - *Example:* A plot added to the `la-liga` branch will **not** be visible in the `main` branch until it is merged.
  - **The Independence Principle:** This isolation is the whole point of branches. Each branch maintains its own independent line of development. You can have twenty branches, each working on different features, and none of them interfere with each other. Changes in one branch don't affect any other branch until you explicitly merge them. This allows teams to work in parallel without stepping on each other's toes. It's why one developer can be fixing a critical bug in a hotfix branch while another works on a new feature in a feature branch, all without conflicts.

Branching Strategy

- **Small Solo Projects:** It is often acceptable to work exclusively on the `main` branch.
  - **When Simple is Better:** For small learning projects, experiments, or simple scripts, the overhead of managing branches might not be worth it. If you're the only person working on the code and it's not mission-critical, committing directly to main can be perfectly fine. The key is recognizing when complexity justifies additional process versus when it adds unnecessary overhead. As you grow as a developer, you'll develop intuition for when branches add value and when they're overkill.
- **Collaborative Projects:** Branches are essential. They are used to work on different project tasks simultaneously.
  - The `main` branch is generally treated as the final, stable version of the project.

- Content from other branches is only incorporated (merged) into `main` **after the work is finished and verified** to ensure there are no issues.
- **The Main Branch as Production:** In professional software development, the main branch often represents what's deployed to production—what users are actually seeing and using. This means main should always be in a working state. Breaking main can break the application for all users. This is why in team settings, changes are developed in feature branches, thoroughly tested, reviewed through pull requests, and only then merged to main. This workflow might seem like extra steps, but it's a safety mechanism that prevents bugs from reaching production and maintains a stable codebase that everyone can depend on.

4. Branch Protection Rules

- **Purpose:** GitHub allows you to enforce rules on specific branches (especially `main`) to add a layer of protection against errors.
- **How to Add Rules:**
  1. Go to the repository's **"Settings"** tab.
  2. Select the **"Branches"** section from the side menu.
  3. Click the **"Add rule"** button in the "Branch protection rules" section.
- **Example Rule Setup:**
  1. **Branch name pattern:** (Mandatory) You must specify which branch(es) the rule applies to. (The video example names the rule "protect main" in this field).
     - **Understanding Pattern Matching:** The branch name pattern field supports wildcards. You could enter `main` to protect only that branch, or `release/*` to protect all branches that start with "release/". This pattern-matching capability allows you to protect multiple branches with similar names using a single rule, which is useful for organizations with many active release branches.
  2. **Require a pull request before merging:** Check this box. This is the most common rule.
     - **Why This Rule Matters:** This is perhaps the single most important protection you can add. It prevents anyone (even repository administrators) from pushing commits directly to the protected branch. All changes must go through a pull request, which means they can be reviewed, discussed, and tested before being merged. This requirement creates a mandatory checkpoint that catches bugs, improves code quality through peer review, and ensures that no changes slip into the main branch unnoticed. It's the foundation of professional Git workflows.
  3. **Require approvals:** This option is selected by default when you require a pull request. You can set the number of required approvals (the default is **1**).
     - **The Review Gate:** Requiring approvals means that a pull request cannot be merged until the specified number of team members have reviewed the changes and explicitly approved them. One approval is common for small teams, while larger organizations might require two or even three approvals for critical code. This creates accountability and distributes knowledge across the team—multiple people

become familiar with each change. It also catches bugs that the author might have missed. Code review is widely considered one of the most effective quality assurance practices in software engineering.

4. Click **"Create"** at the bottom to save the rule.

- **Other Rules:** You can also set rules to restrict who can delete a protected branch.
  - **Additional Protection Options:** Beyond pull request requirements, GitHub offers many other branch protection rules. You can require status checks (automated tests) to pass before merging, require branches to be up-to-date before merging, restrict who can push to the branch, require signed commits for additional security, and more. For critical branches in production environments, teams often enable multiple layers of protection. The goal is to make it nearly impossible to accidentally break the main branch through a combination of automated checks and human oversight.

2.3 Repo access

🔒 Public vs. Private: The "Why"

While GitHub is famous for open-source, many projects require restricted access.

- **Public Repo:** Anyone on the internet can see the repository. You (the owner) still control who can **commit** (write changes), but the code and history are fully visible.
- **Private Repo:** The repository is completely hidden from the public. You explicitly control who can **see**, **read**, and **write** to the repository.
  - **The Visibility Spectrum:** Understanding repository visibility is crucial for both security and collaboration. Public repositories embrace the open-source philosophy—transparency, community contribution, and shared learning. They're excellent for portfolio projects, educational content, and tools you want others to use. Private repositories, conversely, are about controlled access. They're necessary when confidentiality matters, whether for legal reasons, competitive advantage, or privacy concerns. The decision between public and private should be intentional and based on the nature of your content, not just a default choice.

When to Use a Private Repo

You must use a private repository if your project contains:

- **Personally Identifiable Information (PII):** Any data that could identify a specific person, such as names, addresses, or phone numbers. The video's "bank marketing" example is a perfect case for this.
  - **The Legal Imperative of PII Protection:** PII is heavily regulated worldwide. Laws like GDPR in Europe, CCPA in California, and many others impose strict requirements and severe penalties for mishandling personal data. Even if your

project isn't subject to these specific laws, exposing PII is ethically wrong and can harm individuals through identity theft, harassment, or discrimination. In academic and professional settings, exposing PII can result in loss of institutional access, legal liability, and reputational damage. If your data contains real names, email addresses, phone numbers, addresses, social security numbers, medical records, or financial information, it absolutely must be in a private repository—and ideally, it should be encrypted and access-logged.

- **Proprietary Code / Intellectual Property (IP):** Commercial code you intend to sell or that gives your business a competitive advantage (e.g., a custom AI model for stock predictions).
    - **Protecting Your Competitive Edge:** For businesses, proprietary code is often a key asset. Publishing your unique algorithms, business logic, or innovative features in a public repository would be like a restaurant publishing its secret recipes. Even if you don't plan to sell the code directly, it might provide strategic advantage that would be lost if competitors could simply copy your approach. Many startups and established companies maintain private repositories for this reason. That said, many successful companies also open-source large portions of their codebase, keeping only the most critical, differentiating features private. The decision requires weighing the benefits of community contribution and transparency against competitive considerations.
- **Sensitive Credentials (Secrets):** Files containing API keys, database passwords, or other secrets.
    - **Pro-Tip:** Even in a public repo, you should *never* commit these files. The standard practice is to use a `.gitignore` file to ignore them and store the actual secrets in a secure vault or as "GitHub Secrets" in the repo settings.
    - **Why Secrets in Git Are Forever:** Here's a critical concept that many developers learn the hard way. Once you commit a secret to Git history, simply deleting the file in a later commit doesn't remove it. The secret still exists in the repository's history and can be retrieved by examining old commits. If you accidentally commit a secret to a public repository, you must assume it's compromised. You need to immediately revoke that credential (change the password, rotate the API key, etc.) and then use tools like `git filter-branch` or `BFG Repo-Cleaner` to remove the secret from the entire Git history. This is complex and can break things, so the better approach is to never commit secrets in the first place. Use environment variables, configuration files that are gitignored, or secret management services like AWS Secrets Manager, Azure Key Vault, or HashiCorp Vault.

How to Create and Manage a Private Repo

Creating a New Private Repo

1. From the `New repository` page, give your project a name (e.g., `bank-marketing`).
2. Select the `Private` option instead of the default `Public`.

How to Verify a Repo is Private

- **The "Private" Label:** You will see a `Private` label next to your repository's name on its main page.
- **The Incognito Test (Hack):** The fastest way to check is to copy your repo's URL, open a **new Incognito or Private browser window** (where you are not logged in), and paste the URL. If it's private, you will get a **404 "Page not found"** error.
    - **Why This Test Works:** When you're logged into GitHub, you have access to all your repositories, both public and private. This makes it easy to forget which repositories are actually visible to the outside world. The incognito window simulates what an unauthenticated user (the general public) would see. Getting a 404 error confirms that the repository is truly private and inaccessible to unauthorized users. This quick verification is especially valuable after changing a repository's visibility settings to confirm the change took effect as expected. It's a simple but effective sanity check that can prevent embarrassing or serious information leaks.

🤝 Collaboration in Private Repos

The main challenge of private repos is giving access to your team.

- **Action:** Go to `Settings` tab > `Access` > `Collaborators`.
- Click the green `Add people` button.
- Search for your colleague.
    - **Pro-Tip:** GitHub searches its entire user base. Searching by "full name" or "email" can be difficult. **Always ask for your colleague's exact GitHub username** to add them correctly.
    - **The Global Search Challenge:** GitHub has hundreds of millions of users, and many people share common names. Searching for "John Smith" will return hundreds or thousands of results. Even email addresses might not be unique if someone has multiple accounts. The GitHub username is the definitive, unique identifier. When starting a new project with a team, one of your first tasks should be collecting everyone's GitHub usernames. Create a shared document listing team members and their usernames to make adding collaborators straightforward. This also helps when reviewing pull requests—you'll know who authored each one.
- **Pending Invite:** The user will receive an email invitation. Their status will show as `Pending Invite` until they accept. Once accepted, they will have access to the repository.
    - **The Invitation Workflow:** Email invitations sometimes end up in spam folders, so if someone says they didn't receive an invitation, that's the first place to check. The pending status is useful for tracking who has and hasn't accepted access yet. Some organizations have policies requiring invitations to be accepted within a certain timeframe for security reasons. Once accepted, the collaborator appears in your

repository's insights and contributor graphs, and they can clone the repository, create branches, and submit pull requests according to their permission level.

🔑 Advanced Concept: Permission Levels (Roles)

When you add a collaborator to a *personal* private repo, you give them `Write` access by default. In a more professional setup (especially within a **GitHub Organization**), you have much finer control over *what* they can do.

Understanding these roles is critical:

Role	Can They See?	Can They Write Code?	Can They Manage Settings?
Read	✅ Yes	❌ No	❌ No
Write	✅ Yes	✅ Yes	❌ No
Admin	✅ Yes	✅ Yes	✅ Yes

- **Admin:** Full control. Can add/remove other collaborators, change settings, and even **delete the repository**. Be *extremely* careful who you give this role to.
    - **The Admin Responsibility:** Admin access should be restricted to project owners and highly trusted individuals. An admin can not only delete the entire repository but also change its visibility (making a private repo public, exposing all your code and history), transfer ownership, disable security features, and more. In professional settings, admin access is typically limited to technical leads, project managers, or the repository owner. Even senior developers might only have Write access to prevent accidental or malicious changes to critical settings. If someone needs temporary elevated access for a specific task, grant it temporarily and revoke it afterward.
- **Write:** The standard "collaborator" role. They can push code, commit, and review pull requests.
    - **The Developer Role:** Write access is what most active developers need. They can create branches, push commits, submit and review pull requests, and participate fully in the development workflow. They cannot, however, change repository settings, modify branch protection rules, or manage other collaborators' access. This strikes a balance between enabling productive work and maintaining necessary security boundaries. In a team setting, most developers will have Write access to the repositories they actively work on.
- **Read:** Good for auditors, stakeholders, or external reviewers who need to *see* the code but should not be allowed to *change* it.
    - **Read-Only Use Cases:** Read access is valuable in several scenarios. External consultants or contractors who need to review code but shouldn't make changes. Compliance officers or auditors who need to verify code for security or regulatory purposes. Stakeholders like product managers or executives who want visibility into

the technical work without directly participating. New team members during onboarding who are learning the codebase before being given Write access. The Read role enables transparency and oversight without the risks associated with allowing changes to the codebase.

⚠️ Pro-Tip: Changing an Existing Repo's Visibility

You can change a repo's visibility at any time.

- **Location:** `Settings` > `General` > scroll to the bottom to the **"Danger Zone"**.
- Click `Change repository visibility`.

CRITICAL WARNING: If you change a `Public` repository to `Private`, you will **permanently lose all stars and watchers** for that repository. This action cannot be undone.

- **Understanding the Stakes:** Stars and watchers represent community engagement with your project. Stars are similar to likes or bookmarks—they indicate that people found your project valuable. Watchers are users who've chosen to receive notifications about your project's activity. Losing these metrics when going from public to private makes sense (since private repositories aren't discoverable), but it's irreversible. If you later make the repository public again, you start from zero. This is one reason to carefully consider a repository's visibility from the start. If you're uncertain, starting private is safer—you can always go public later without losing anything. Going from public to private should be a carefully considered decision, ideally discussed with any active contributors or users of your project.

2.4 Personal Access Tokens (PAT)

🔒 The "Authentication Failed" Problem

- **The Scenario:** You try to use a Git command from your terminal to interact with a GitHub remote repository (e.g., `git clone`, `git push`, `git pull`).
- **The Process:** The terminal correctly prompts for your `Username:`. You enter it. Then, it prompts for `Password:`.
- **The Failure:** You enter your GitHub account password, and it **fails**. The terminal returns an error stating that **password authentication was removed on August 13th, 2021.**
- **The Reason:** This was a major security update by GitHub to improve security for all users.
    - **Why GitHub Deprecated Password Authentication:** Passwords are inherently less secure than tokens for several reasons. First, they're typically used across multiple services, so compromising one service's database could expose your GitHub account. Second, passwords can't be scoped—your password gives full access to everything in your account. Third, if a password is leaked, changing it affects all your applications and scripts that use it. Fourth, passwords are often weak or reused. GitHub's move to require token-based authentication addressed all

these issues and aligned with industry best practices. The August 13, 2021 date was after a long deprecation period where GitHub warned users about the upcoming change, giving everyone time to transition to tokens.

What is a Personal Access Token (PAT)?

A PAT is a secure, computer-generated string of characters that acts as an alternative to your password for authenticating to GitHub from the command line or via APIs.

Why are PATs more secure than a password?

- **Scoped:** A password gives **full access** to your entire account. A PAT only has the permissions (or "scopes") you choose to give it (e.g., only "read" access).
    - **The Principle of Least Privilege:** Scoping is a fundamental security concept called "least privilege"—giving each credential only the minimum permissions it needs to function. If you're writing a script that only needs to read public repository data, you can create a PAT with only public read permissions. If that token is somehow compromised, an attacker can't use it to delete your repositories or access private data. This granular control isn't possible with a password, which provides all-or-nothing access to your entire account.
- **Revocable:** If a PAT is leaked, you can **instantly revoke that one token** from your GitHub settings without needing to change your main account password.
    - **Incident Response Made Easy:** Imagine you accidentally commit a PAT to a public repository (this happens more often than you'd think). With passwords, you'd need to change your account password immediately, which would break all your other services and scripts using that password, causing widespread disruption. With a PAT, you simply go to your GitHub settings, find that specific token in the list, and delete it. All your other tokens continue working. You create a new token to replace the compromised one, and only the specific affected service needs updating. This containment and ease of remediation is a huge security advantage.
- **Temporary:** PATs can be set with an **expiration date**, which limits the window of opportunity for an attacker if the token is lost.
    - **Expiration as a Security Layer:** Expiration dates implement the security principle of "temporal containment." Even if a token is stolen and you don't know it, the token will automatically become useless after its expiration date, limiting the potential damage window. This is especially valuable for tokens used in automated systems or on shared machines. The trade-off is that you need to remember to renew tokens before they expire, but this small inconvenience is far outweighed by the security benefit. GitHub will email you reminders before tokens expire, helping you stay on top of renewal.

> **Key Distinction:** You only need a PAT when interacting with GitHub from an external tool, like your **terminal**. You do *not* need a PAT to log in or use the **GitHub.com website** in your browser.

- **Understanding the Two Authentication Paths:** This distinction confuses many new users. Your GitHub password is for the website—logging into GitHub.com, changing settings, browsing repositories. Personal Access Tokens are for programmatic access—when Git, scripts, APIs, or other tools need to authenticate as you. The website uses session-based authentication with cookies, while command-line tools use token-based authentication. These are separate authentication mechanisms for separate use cases. You'll use your password dozens of times, but you might only create a handful of PATs for specific purposes.

💡 Key Concept: The "Password" Prompt

This is the most confusing part for new users. When the terminal asks you for...

`Password for 'https://YourUsername@github.com':`

...it is **NOT** asking for your GitHub password. It is asking for your **Personal Access Token (PAT)**.

`Prompt: Password` → `Input: Your_PAT_gph_...`

- **Why This Terminology Confusion Exists:** Git's password prompt hasn't been updated to say "Password or Token" because Git is a separate project from GitHub and doesn't know about GitHub-specific authentication methods. From Git's perspective, it's asking for a password to authenticate over HTTPS. GitHub has repurposed this prompt to accept tokens instead. This mismatch in terminology is unfortunate but unavoidable given how Git and GitHub are separate projects that must maintain compatibility. Understanding this distinction—that "password" in the terminal means "token" for GitHub—is critical for successfully using Git with GitHub.

How to Create a PAT

1. In GitHub, click your **profile icon** in the top-right corner.
2. Go to `Settings`.
3. Scroll all the way down and select `Developer settings` from the left-hand menu.
4. Select `Personal access tokens` > `Tokens (classic)`.
5. Click the `Generate new token` button (and select `Generate new token (classic)`).
6. **Configure the token:**
    - **Note:** Give it a descriptive name so you remember what it's for (e.g., "My-Laptop-Terminal" or "bank-marketing-project").
    - **The Importance of Descriptive Names:** Months from now, when looking at your list of tokens, a name like "token1" will be useless. You won't remember what it was for, which system is using it, or whether it's safe to delete. Descriptive names like "MacBook-Pro-Git-Access" or "CI-CD-Pipeline-Production" immediately tell you the token's purpose and where it's used. Include relevant context like the machine name, project name, or purpose. This makes token management much easier and helps

you audit your security regularly by identifying unused or forgotten tokens that should be revoked.

- **Expiration:** Set an expiration date. **30 days is the default and a good practice.** A token for a short-term project can be 7 days.
- **Balancing Security and Convenience:** Shorter expiration periods are more secure (smaller window of vulnerability if compromised) but require more frequent renewal. For development tokens you use daily, 30-90 days is reasonable. For tokens used in automated systems, longer periods like 1 year might be practical, but these require extra care in storage and monitoring. For temporary collaborations or one-off tasks, 7 days works well. The key is setting expiration dates that match the token's intended lifespan. Never choose "no expiration"—tokens should have a defined lifetime to force periodic security reviews.
- **Select scopes:** This is the most important step. It defines *what* the token can do. For general development, you must check the `repo` scope. This grants full read/write access to all your repositories.
- **Understanding Scopes in Detail:** The scope selection is where you define the token's capabilities. Each checkbox represents a specific permission. The `repo` scope is powerful—it grants full control over all your repositories, both public and private. Other useful scopes include `workflow` (for GitHub Actions), `read:packages` (for GitHub Packages), `delete:packages`, `admin:org` (for organization management), and many more. When creating a token, think about exactly what it needs to do and enable only those scopes. If a token is only for reading public repository data, don't check `repo`—use `public_repo` instead. This careful scoping is a critical security practice.

🔐 Pro-Tip: "Tokens (classic)" vs. "Fine-grained tokens"

You will see two options when generating a token. The video describes "Tokens (classic)," which uses broad scopes like `repo`.

- **Classic Tokens:** Powerful and simple to set up (e.g., the `repo` scope gives access to *all* your repos).
- **Fine-grained Tokens (Recommended):** The new, more secure standard. They force you to be more explicit. You can grant a token access to **only specific repositories** and with **specific permissions** (e.g., "Read-only" for Issues, "Read and Write" for Code).
    - **The Evolution Toward Granularity:** Fine-grained tokens represent GitHub's response to security best practices evolving over time. With classic tokens, the `repo` scope gives access to all repositories in your account—if you have 100 repositories, the token works on all of them. This violates least privilege if you only need access to one repository. Fine-grained tokens let you specify exactly which repositories the token can access and what actions it can perform on those repositories. For example, you might create a fine-grained token that has read

access to repository A and write access to repository B, but no access to repository C. This precision significantly reduces the blast radius if a token is compromised.

Best Practice: Whenever possible, use a **Fine-grained token** and give it the *minimum permissions* it needs to do its job.

- **Implementing Least Privilege Properly:** When creating any credential, ask yourself: "What is the absolute minimum this needs to do?" If you're setting up a deployment script, it might only need push access to one specific repository. If you're creating a token for a monitoring tool, it might only need read access to workflow status. Don't default to giving full `repo` scope to everything. Take the time to understand the available permissions and configure them precisely. Your future self will thank you when investigating a security incident, and you'll sleep better knowing that even a compromised token can do limited damage.

How to Use and Store Your PAT

Step 1: Copy the Token

- After you click `Generate token`, GitHub will show you the token **ONE TIME**.
- This is your only chance to see it. **You must copy it immediately.**
- Click the clipboard icon to copy the full string (it will look something like `ghp_...`).
    - **Why GitHub Won't Show It Again:** GitHub doesn't store the token in a reversible format. Once you navigate away from the token creation page, even GitHub cannot retrieve the token's value—they only store a hash of it. This is a security feature, not a limitation. It ensures that even if GitHub's database is compromised, the actual token values are not exposed. If you close the page without copying your token, your only option is to delete that token and generate a new one. This is inconvenient but necessary for security. Make it a habit to immediately copy newly created tokens and store them securely before doing anything else.

Step 2: Use the Token

- Go back to your terminal.
- When prompted for `Password:`, **paste your newly copied PAT** and press Enter. (Note: the terminal will not show any characters as you paste; this is normal.)
    - **Understanding Terminal Security:** The lack of visual feedback when pasting passwords or tokens is a security feature, not a bug. Terminals hide password input to prevent shoulder surfing (someone looking over your shoulder and seeing your credentials). They also don't show asterisks or dots because those can leak information about password length. This can be unsettling at first—you paste your token and see nothing, making you wonder if it worked. Trust that the paste occurred, press Enter, and the command will execute. If the authentication fails, you'll get an error message. If it succeeds, your Git command will proceed as expected.

- Your `git clone` (or other command) will now succeed.

Step 3: Store the Token (The "Hack")

You don't want to do this every time. How do you save it?

- **Best Practice (The Easy Way):** Most modern Git installations (on Mac and Windows) come with a **Git Credential Manager**. The *first time* you successfully use your PAT, the Credential Manager will ask to save it (e.g, in macOS Keychain or Windows Credential Manager). Say yes. From then on, Git will automatically use the saved token.
    - **How Credential Managers Work:** Git Credential Manager acts as an intermediary between Git and your operating system's secure storage. When Git needs credentials, it asks the Credential Manager, which retrieves them from the OS-level secure vault (Keychain on macOS, Credential Manager on Windows, or Secret Service on Linux). The first time you authenticate, the manager saves your token securely, and on subsequent operations, it provides the token automatically without prompting you. This is secure because the tokens are stored with OS-level encryption and are tied to your user account. They're not stored in plain text files where anyone could read them.
- **Best Practice (Manual):** Store your PAT in a **password manager** (like 1Password, Bitwarden, etc.), just like you would any other password. This is the most secure place to keep it.
    - **Why Password Managers Are Essential:** Password managers solve multiple problems simultaneously. They generate strong, unique credentials, store them encrypted with strong encryption, sync them securely across your devices, and protect them with multi-factor authentication. For tokens, password managers provide a backup copy in case your Git Credential Manager storage gets corrupted or you need to set up a new machine. They also make it easy to track when tokens were created and when they'll expire. Modern password managers can even alert you when tokens are about to expire, helping you proactively renew them. Using a password manager is one of the most important security practices you can adopt as a developer.

⚠️ CRITICAL WARNING:

Treat your PAT exactly like a password. Never share it with anyone, and never save it in a public-facing file (like in a script or a README in your repo). If you think it has been leaked, go to your GitHub settings and revoke (delete) it immediately.

- **The Accidental Commit Scenario:** One of the most common security mistakes developers make is accidentally committing tokens to their repositories. You might be testing an API call and hardcode the token temporarily, intending to remove it later. You forget, commit the code, and push to GitHub. If the repository is public, your token is now exposed to the entire internet. There are automated bots that scan GitHub looking for exactly this kind of mistake, and they will find and exploit your token within minutes. If

this happens: immediately revoke the compromised token, review what access it had to understand the potential impact, check your repository history to see if sensitive data was accessed, create a new token with appropriate scopes, and consider whether you need to rotate any other credentials. Then, use tools like BFG Repo-Cleaner to remove the token from your Git history. Prevention is far easier than remediation, so adopt habits that prevent accidental exposure: never hardcode credentials, always use environment variables or configuration files that are gitignored, and regularly audit your `.gitignore` to ensure sensitive files are excluded.

Final Thoughts on These Notes: What you've built here is a comprehensive foundation for understanding GitHub. These notes cover not just the "what" and "how" but also the "why"—the reasoning behind GitHub's features and best practices. As you continue your learning journey, remember that GitHub is a tool that rewards practice. Reading these notes is valuable, but the concepts will truly solidify when you apply them to real projects. Start by creating repositories for your learning projects, practice making branches and pull requests even when working alone to build muscle memory, and gradually incorporate more advanced features like branch protection rules and fine-grained tokens as you become comfortable with the basics. The GitHub community is vast and helpful, so don't hesitate to explore public repositories, read other developers' code, and learn from how successful open-source projects structure their workflows. Your understanding will deepen with every project you work on, and these notes will serve as a valuable reference guide throughout that journey.

Chapter 3: Collaboration with GitHub

3.1 Using Other Repos

Cloning

- **Definition:** Cloning a repo is similar to copy-paste; however, it has a **link to the original repo**.
- **What Cloning Creates:** It creates a copy of the repo in its current state on your local computer and allows you to send updates back and forth.
- **The Bidirectional Connection:** Understanding the link between a cloned repository and its origin is fundamental to collaborative Git workflows. When you clone a repository, Git automatically sets up a "remote" connection called "origin" that points back to the source repository. This connection is what enables the synchronization magic of Git. You can work on the files locally, push changes back to the repo, or pull any new updates to your local version using Git. Think of it as a two-way street where information flows in both directions—you can receive updates that others have made, and you can send your own updates back.
- **Access Requirements:** Anyone can clone a public repo, but a private repo owner needs to grant access.

- **Understanding Access Permissions for Cloning:** The ability to clone a repository doesn't automatically grant you the ability to push changes back to it. For public repositories, anyone can clone and create a local copy to study, learn from, or experiment with. However, only users with write access can push changes back to the original repository. For private repositories, even cloning requires explicit permission—the repository owner must add you as a collaborator first. This distinction between read access (cloning) and write access (pushing) is an important security layer that protects repositories from unauthorized modifications.

How to Clone a Repo

1. Navigate to the repository you want to clone on GitHub (in the example, George Boorman's bank marketing private repo).
2. Click on the green **"Code"** button to reveal a dropdown menu.
3. The **HTTPS** tab should already be selected by default.
    - **HTTPS vs. SSH:** GitHub offers two primary methods for cloning repositories—HTTPS and SSH. HTTPS is the simpler option and works immediately with your Personal Access Token for authentication. SSH requires setting up SSH keys on your computer and adding the public key to your GitHub account, but it's more convenient once configured because you don't need to enter credentials repeatedly. For beginners, HTTPS is recommended. SSH becomes valuable when you're frequently pushing and pulling from multiple repositories. The URL format differs between them—HTTPS URLs start with `https://github.com/`, while SSH URLs start with `git@github.com:`. Both methods clone the exact same repository content; they just use different authentication mechanisms.
4. Copy the URL provided by clicking the **copy icon** next to it.
5. Open your terminal window.
6. Check that you are in the correct directory where you want the repository to be saved.
    - **Choosing the Right Location:** Before cloning, think about where you want the repository folder to live on your computer. Many developers create a dedicated directory for all their Git projects, something like `~/Projects` or `~/GitHub` on Mac/Linux, or `C:\Users\YourName\Projects` on Windows. Navigate to this parent directory before running the clone command. The clone command will create a new folder with the repository's name inside your current directory, so you don't need to create a folder for the repository yourself—Git does this automatically.
7. Type the command `git clone` followed by the URL you copied from GitHub and hit Enter.
    - **What Happens During Cloning:** When you execute `git clone`, Git performs several operations automatically. It creates a new directory with the repository's name, initializes a `.git` directory inside it, fetches all the repository's data (every file, every commit, the entire history), checks out the default branch (usually `main`), and sets up the remote connection to the original repository. You'll see progress messages showing the objects being received and resolved. For large repositories

with extensive histories, this can take a few minutes. For smaller projects, it's nearly instantaneous.

8. You may be prompted to enter your GitHub username and Personal Access Token (PAT). Enter them and hit Enter again to initiate the clone.
    - **Authentication During Cloning:** If the repository is private, or if you want to push changes later, GitHub will need to verify your identity. As discussed in the PAT section, when prompted for your password, you should paste your Personal Access Token, not your GitHub account password. If you've already authenticated and your credentials are saved in Git Credential Manager, you might not be prompted at all—Git will use the stored credentials automatically. If authentication fails, double-check that you've been granted access to the repository and that your token has the necessary permissions (the `repo` scope for private repositories).
9. The repository is now cloned to your local machine in a folder matching the repository name.

Cloning an Empty Repository

- **Special Case:** If the repo you are cloning is empty (has no files yet), the process is similar with one key difference.
- **Where to Find the URL:** The link you need to copy will appear on the repo's main page within a **blue box called "Quick Setup"** instead of requiring you to click the Code button.
- **Understanding Empty Repository Cloning:** When a repository is brand new and contains no files, GitHub presents a streamlined setup interface. This "Quick Setup" page provides the clone URL prominently and also offers helpful commands for initializing a repository from the command line or pushing an existing local repository to GitHub. Cloning an empty repository creates the local folder and sets up the Git connection, but there will be no files inside initially except the hidden `.git` directory. This is a common scenario when starting a new project—you create the repository on GitHub first to establish it as the central source of truth, then clone it to your local machine to begin working.

Forking

- **Definition:** Forking creates an **independent copy** of a repository on your GitHub account, without linking back to the original in terms of synchronized updates.
- **The Fork's Independence:** A fork of a repo creates an independent copy of it on your GitHub account, meaning you can run experiments without the risk of anything reaching the original repo. Think of forking as creating a parallel version of a project that exists alongside the original. While the fork retains a reference to where it came from (GitHub shows "forked from original-repo"), the two repositories are operationally separate. You have complete control over your fork—you can delete files, restructure the project, experiment with risky changes—all without any possibility of affecting the original repository. This safety makes forking ideal for experimentation and learning.

- **Common Use Case:** This option is often used for collaboration in the open-source community.
- **The Open-Source Contribution Workflow:** Forking is the foundation of the open-source contribution model. Here's how it typically works in practice. You discover an open-source project you'd like to improve—maybe you've found a bug or want to add a feature. You fork the repository to your account, which gives you a personal copy you can modify. You make your changes in your fork, test them thoroughly, and then submit a pull request back to the original repository. The original project maintainers review your pull request and, if they approve, merge your changes into the official project. This workflow allows thousands of developers to contribute to projects they don't own without requiring the project maintainers to grant write access to everyone, which would be a security nightmare.
- **Access Requirements:** Anyone with a GitHub account can fork a public repo, but the owner of a private repo needs to allow this feature in the repository settings.
- **Why Fork Permission Matters for Private Repos:** For private repositories, forking has implications for information security. When you fork a private repository, you create a complete copy of its entire history, including all commits, branches, and files. If the repository contains sensitive information, the owner needs to carefully consider who should be allowed to create forks. A malicious actor with fork permission could copy the repository and retain that data even if they're later removed as a collaborator. This is why fork permissions for private repositories require explicit enabling and should be managed thoughtfully.
- **Forking vs. Branching:** Forking differs from creating a new branch in that you don't need to be registered as a project collaborator to fork a public repository.
- **Understanding the Collaboration Models:** Branches and forks represent two different collaboration models. Branches are for internal team members who have write access to the repository—they create branches within the same repository to work on different features. Forks are for external contributors who don't have write access—they create their own copy of the entire repository in their account. In practice, small teams or companies typically use branches for collaboration since everyone has access to the repository. Large open-source projects use forks because they can't grant write access to every potential contributor. Some projects use a hybrid model where core team members use branches and external contributors use forks.

How to Fork a Repo

1. Navigate to the repository you want to fork on GitHub (in the example, the bank marketing repo with forking enabled by the owner).
2. Click the **"Fork"** button in the top-right corner of the page.
3. In the dropdown menu, select **"Create a new fork"**.
4. **Select the Owner:** If you are part of any organization on GitHub, you may see several options in a dropdown list to select the owner. Otherwise, there should only be one option (your personal account).

- **Understanding Fork Ownership:** This dropdown appears because GitHub allows you to fork repositories into different namespaces. If you're a member of GitHub Organizations (like a company or team workspace), you can choose to fork the repository either to your personal account or to an organization you belong to. The choice matters because it determines who has access to the fork and who can manage its settings. Forking to your personal account gives you complete control. Forking to an organization means the organization's administrators also have access and can manage the repository. For learning or personal experimentation, fork to your personal account. For team projects, fork to the appropriate organization.

5. **Repository Name:** The default name will be the same as the repo you are forking from. You should amend this to differentiate your fork from the original if desired (though keeping the same name is also common and acceptable).
    - **Naming Conventions for Forks:** There's no strict rule about whether to rename forks, and the community is divided. Many developers keep the original name because it makes the fork easy to identify and shows clear lineage from the source project. Others add suffixes like `-fork` or prefixes like `my-` to distinguish their copy. If you plan to significantly diverge from the original project and make it your own, renaming makes sense. If you're forking to contribute back to the original project, keeping the same name is cleaner. The "forked from" notation that GitHub adds provides context regardless of your naming choice.
6. **Description:** You have the option to add a description. This will be copied from the original repository by default, but you can modify or leave it blank.
7. **Select Branch:** Choose which branch you want to fork from. If there is only one `main` branch, select that one. The default is to fork only the main branch even if the repository has multiple branches.
    - **Branch Selection Strategy:** By default, GitHub offers to copy only the default branch when forking. This makes sense because it keeps your fork smaller and more focused. The default branch typically represents the stable, current state of the project. However, you have the option to copy all branches if you want the complete picture of the project, including feature branches and development work in progress. For most purposes, forking just the default branch is sufficient. You can always pull additional branches later if needed. If you're contributing to an active project with specific feature branches you want to work with, you might want those branches included in your fork from the start.
8. Click **"Create fork"**.
9. **Result:** You have successfully created an independent copy of the original repository on your GitHub account.
10. **Next Step:** To work on this forked project on your local computer, you need to clone your fork (not the original repository) using the cloning steps described earlier.
    - **The Fork-Clone-Contribute Workflow:** Understanding the complete workflow is essential. First, you fork the original repository to your GitHub account. This creates your copy on GitHub's servers. Second, you clone your fork to your local machine.

This gives you a local working copy. Third, you make changes locally, commit them, and push them to your fork on GitHub. Fourth, you create a pull request from your fork back to the original repository. This workflow might seem complex at first, but it provides powerful benefits—you have complete freedom to experiment in your fork, the original repository remains protected, and your changes can be reviewed before being accepted. This is how millions of contributions are made to open-source projects every day.

Clone vs. Fork: Summary

Cloning:

- Creates a **linked copy** on your local computer
- Requires using **Git** via the command line
- Updates are sent back and forth using Git's `push` and `pull` commands
- You must have access to the repository to push changes back
- Best for: Working on repositories you have write access to, or downloading code you want to run locally

Forking:

- Creates an **independent copy** on your GitHub account (not on your local computer)
- Performed entirely through the GitHub web interface
- Changes to the original don't automatically update your fork
- Changes you make are submitted back to the original through a **pull request**
- Best for: Contributing to projects you don't have write access to, experimenting without risk to the original

Both Are Complementary: In practice, you'll often use both together. The typical workflow for contributing to an open-source project is to fork the repository on GitHub (creating your independent copy), then clone your fork to your local machine (so you can edit files locally). You make changes locally, push them to your fork on GitHub, and then create a pull request from your fork back to the original repository. Both are great for collaboration, but forking allows for safer experimentation and is the standard model for open-source contribution.

3.2 GitHub Issues

What is an Issue?

- **Definition:** GitHub issues are messages used to **track problem fixes, plans, important tasks, and communications** for a project.
- **Issues as Project Management:** Issues are much more than just bug reports—they're a complete project management system built into GitHub. You can use issues to plan

features before writing any code, track bugs that need fixing, document questions and discussions about project direction, create to-do lists for complex features, coordinate work among team members, and maintain a searchable history of decisions and their rationale. Many teams use issues as their primary project management tool, replacing separate systems like Jira or Trello. The advantage is that issues live alongside your code, can reference specific commits and pull requests, and are version-controlled just like your code.

- **Location:** Issues live under the **"Issues"** tab, where you can see all issues for that repo in one centralized place.
- **The Issues Dashboard:** The Issues tab provides a powerful interface for managing project work. You can filter issues by their status (open or closed), by who they're assigned to, by labels, by milestones, and more. The search functionality lets you find specific issues quickly even in projects with thousands of issues. Each repository maintains its own separate issue tracker, which keeps conversations focused and relevant. This centralization means team members don't need to search through email threads or chat histories to find important discussions—everything related to the project lives in the repository.

How to Create a New Issue

1. Navigate to the **"Issues"** tab of the repository.
2. You'll see all existing issues for this specific repo. If there are no issues yet, the page will be empty.
3. Click the **"New issue"** button on the right side to create a new one.
4. **Give your issue a title:** This should be concise and descriptive (e.g., "Fix broken login button" or "Add dark mode toggle").
    - **Writing Effective Issue Titles:** The title is the first thing people see when scanning the issues list, so it needs to be clear and informative. Good titles are specific without being overly long. They often start with a verb describing the action needed: "Fix," "Add," "Update," "Remove," "Refactor." They identify the specific component or feature affected. Compare "Fix bug" (vague, unhelpful) with "Fix date formatting error in reports page" (specific, actionable). The title should let someone quickly understand what the issue is about without needing to read the full description. Think of it as the subject line of an email—it should convince people this issue is worth their attention.
5. **Add details:** Fill in the description box with all relevant information about the issue.
    - **What Makes a Good Issue Description:** A comprehensive issue description anticipates the questions readers will have. For bug reports, include steps to reproduce the problem, what you expected to happen, what actually happened, your environment (browser, operating system, relevant versions), and ideally screenshots or error messages. For feature requests, explain the problem you're trying to solve, why this feature would be valuable, and potentially how you envision it working. For tasks, break down the work into clear steps or requirements. The more context you

provide, the easier it is for others to understand and address the issue. Remember that you might return to this issue months later when the details aren't fresh in your mind—write for future you as well as for others.

6. **Format with Markdown:** The issue description supports Markdown formatting, so you can enhance it with syntax you previously learned (headings, bold, italic, links, code blocks, etc.).
    - **Leveraging Markdown in Issues:** Markdown transforms plain text into well-organized, readable content. Use headings to structure complex issues into sections. Use code blocks to share error messages or code snippets (wrap code in triple backticks). Use checkboxes to create task lists that can be checked off as work progresses (syntax: `- [ ]` for unchecked, `- [x]` for checked). Use bold for emphasis on critical points. Add links to relevant documentation or related resources. Include images to show bugs or mock up desired features. The better formatted your issue, the more likely people are to read and understand it fully.
7. **Preview your formatting:** Click the **"Preview"** tab to see how your markdown will render before submitting.
8. Click **"Submit new issue"** to create the issue.

Assigning Issues

- **The Problem:** In collaborative projects, simply creating an issue isn't enough—you need to ensure it reaches the right person who should work on it.
- **Initial State:** When you first create an issue, it's unassigned. This means it's visible to everyone with access to the repository, but no one is explicitly responsible for it.
- **How to Assign an Issue:**
    1. Navigate to the **"Assignees"** section on the right side of the issue page.
    2. Click on the **gear icon** (⚙️).
    3. You'll see a list of all people who have access to the repository.
    4. Select the person (or people) who should work on this issue.
- **Multiple Assignees:** GitHub allows you to assign multiple people to a single issue. This is useful for complex issues that require collaboration between multiple team members with different expertise (e.g., a frontend developer and a backend developer working together on a feature).
- **Understanding Assignment Semantics:** Assigning someone to an issue serves several purposes. It clarifies ownership and responsibility—the assignee knows this is their task to complete. It enables filtering—team members can view all issues assigned to them to see their workload. It provides accountability—everyone can see who's responsible for each issue. It triggers notifications—assigned users typically receive emails or notifications about updates to the issue. In professional settings, assignment often comes with expectations about when the work will be completed, making it a key tool for project management and coordination.

Tagging Users

- **Syntax:** You can tag a user within an issue by using the **@ symbol** followed by their GitHub username (e.g., `@george-boorman`).
- **When Tagging Happens:** As you type `@` followed by characters, GitHub will autocomplete and suggest usernames of people who have access to the repository.
- **The Notification Effect:** When you tag someone, they receive a notification that they've been mentioned in an issue, even if they're not assigned to it. This brings the issue to their attention without necessarily making them responsible for it.

Assign vs. Tag: The Distinction

- **Assigning:** Clarifies **who should be working on** the issue. Assignment implies ownership and responsibility for resolving or completing the issue.
- **Tagging:** Functions as a **communication tool** to ensure the right people **read the message** or are aware of a discussion. Tagging is about involving someone in the conversation without necessarily expecting them to do the work.
- **Practical Usage Example:** You might assign a bug to a developer to fix, while tagging a product manager to keep them informed of the bug's status. Or you might assign a feature to yourself while tagging a colleague to ask for their opinion on the implementation approach. The assignee is expected to move the issue toward resolution, while tagged users are participants in the discussion surrounding it.

Commenting on Issues

- **Ongoing Communication:** Once an issue is created, the conversation continues through comments. Issues are designed to be living discussions that evolve as work progresses.
- **How to Comment:**
  1. Scroll down to the comment box located below the issue description.
  2. Write your comment using Markdown formatting as desired.
  3. Click the green **"Comment"** button to post.
- **The Value of Issue Comments:** Comments serve multiple purposes. They provide updates on progress (e.g., "Working on this, should have a PR ready by Friday"). They ask clarifying questions when requirements aren't clear. They share findings or challenges encountered while working on the issue. They discuss different approaches to solving the problem. They document decisions made during implementation. This creates a valuable historical record—months later, when someone asks "Why did we implement it this way?" the answer is documented in the issue thread. Good teams maintain active, thoughtful discussions in issue comments, treating them as a form of documentation and knowledge sharing.

Linking to Other Issues

- **Use Case:** Sometimes issues are related to each other, and GitHub makes it easy to create explicit connections between them.

- **How to Link:**
  1. In an issue comment, type a single **hashtag (#)**.
  2. GitHub will automatically show you a dropdown of other issues in the same repository.
  3. Continue typing to filter the list, or select the issue you want to reference.
  4. Clicking on an issue from the dropdown creates a link to that issue (e.g., `#42`).
- **Example from the Course:** When reviewing the Spring campaign in issue #2, you might want to refer back to updates made in the Winter Campaign discussion in issue #1. By typing `#1`, you create a clickable link that lets readers jump directly to that previous discussion.
- **The Web of Context:** Issue linking creates a knowledge graph of your project. Issues that block other issues can be linked to show dependencies. Related bugs can be cross-referenced. Feature implementations can reference the original feature request issues. When an issue is closed, any issues that link to it show up in the "referenced in" section, making it easy to trace the history and relationships between different pieces of work. This interconnection transforms your issue tracker from a simple list into a rich, navigable knowledge base about your project's development.

Quoting Previous Comments

- **Use Case:** In issues with long discussions involving multiple comments, you may want to respond to a specific earlier comment without forcing readers to scroll through the entire thread for context.
- **How to Quote:**
  1. Navigate to the specific comment you want to quote.
  2. Click the **three dots (...)** in the top-right corner of that comment.
  3. Select **"Quote reply"** from the dropdown menu.
  4. This converts that comment into a quoted format in a new comment box where you can add your reply below the quote.
- **Markdown Syntax:** The syntax for creating a quote in Markdown is the **right chevron (>)** symbol at the beginning of each line.
- **Why Quoting Matters:** Quoting makes long discussions easier to follow. It provides immediate context for your response without requiring readers to remember or search for the original comment. It's particularly valuable when multiple threads of discussion are happening within the same issue. By quoting the specific point you're addressing, you keep your response focused and prevent confusion about what you're referring to. This practice is especially important in issues that span days or weeks, where dozens of comments might accumulate.

Closing Issues

- **When to Close:** Once everything in the issue has been addressed and the work is complete, the issue can be closed.

- **How to Close:**
  1. Navigate to the bottom of the issue page.
  2. You can optionally add a final comment explaining the resolution.
  3. Click the **"Close with comment"** button (or just "Close issue" if not adding a final comment).
- **Close Options:**
  1. **Close as completed** (default): Indicates that items in this issue were successfully addressed and the work is done.
  2. **Close as not planned**: Indicates that a fix was not possible, the feature was decided against, or the issue is not going to be addressed for some reason.
- **Accessing the Close Options:** The "Close as not planned" option is accessible via the **dropdown arrow** next to the close button.
- **Understanding Issue Lifecycle:** Issues move through a lifecycle from open (work needed) to closed (work completed or issue resolved). Closing issues is important for project hygiene—it keeps the open issues list focused on current work and prevents it from becoming overwhelming. Closed issues remain accessible and searchable, so closing doesn't mean losing the information. Many teams have regular triage sessions where they review open issues, close ones that are no longer relevant, and ensure remaining open issues are properly categorized and assigned. The distinction between "completed" and "not planned" is valuable for analytics—it helps teams understand what work was actually done versus what was declined or deprioritized.
- **Viewing Closed Issues:** The Issues tab has a filter that defaults to showing only open issues. You can click on "Closed" to view all previously closed issues, which maintains a complete history of all work and discussions on the project.
- **Reopening Issues:** If circumstances change and a closed issue needs attention again, any collaborator can reopen it using the "Reopen issue" button. This flexibility accommodates situations where a bug that was thought to be fixed reappears, or where a declined feature is reconsidered.

3.3 Pull Requests

What is a Pull Request (PR)?

- **Definition:** A PR is a way to **notify others about changes** you would like to make to a branch within a repo.
- **Review Mechanism:** It allows the repo owner or other collaborators to **check the changes before they are added** to the target branch.
- **The Pull Request Philosophy:** Pull requests embody a fundamental principle of professional software development: changes should be reviewed before being integrated. Even experienced developers make mistakes, overlook edge cases, or could benefit from a second pair of eyes. The pull request creates a formal checkpoint where

proposed changes are examined, discussed, tested, and approved before becoming part of the official codebase. This process catches bugs early, ensures code quality, spreads knowledge across the team, and maintains a historical record of why changes were made.

- **Best Practice:** It is best practice to add changes to a **separate branch** from the default `main` branch. This ensures that `main` only contains finished, tested, production-ready work.
- **Branch Isolation Strategy:** The practice of keeping `main` clean and stable is fundamental to professional Git workflows. The `main` branch should always be in a deployable state. If you need to show a demo, deploy to production, or create a release, you should be able to do so from `main` at any moment without worrying about incomplete features or broken functionality. By developing in separate branches and only merging to `main` through reviewed pull requests, you maintain this stability while still enabling rapid development. Feature branches can be experimental, broken, or work-in-progress without any impact on the production codebase.
- **Goal:** The aim of a successful PR is to **merge two branches**—typically merging a feature branch into the main branch.

How to Make a Pull Request

Prerequisites: You have already created a branch for the task you're working on, made your updates, and committed those changes to your branch.

1. Navigate to the **page for your feature branch** on GitHub (in the example, the `update_readme` branch on a forked repo).
2. Click on the **"Pull requests"** tab at the top of the repository.
3. Click the **"New pull request"** button.

Comparing Changes

This step is critical to ensure you're merging the right branches in the right direction.

- **Base Branch:** The branch you want to add your updates **to**. This is the destination or target branch (typically `main`).
- **Compare Branch:** The branch that **contains your updates**. This is the source branch with your new code (e.g., `update_readme`).
- **The Direction Matters:** Understanding which branch is the base and which is the compare is essential. You're asking to merge compare into base. The base branch will receive the changes. The compare branch provides the changes. Getting this backwards would try to merge main into your feature branch, which is the opposite of what you want. Always double-check these settings before proceeding. GitHub usually guesses correctly, but verifying is crucial, especially when working with forks or complex branching structures.

- **Conflict Detection:** After you've confirmed the correct branches, GitHub will analyze them and tell you if there are any **conflicts** (overlapping changes that Git can't automatically merge).
- **Understanding Merge Conflicts:** Conflicts occur when the same lines of code have been modified differently in both branches. For example, if you changed line 10 of a file in your branch, and someone else also changed line 10 in the base branch since you created your branch, Git doesn't know which version to keep. GitHub will alert you to conflicts and prevent automatic merging until they're resolved. Resolving conflicts requires human judgment to decide which changes to keep or how to combine them. If there are no conflicts, GitHub displays a happy message like "Able to merge" with a green checkmark, and you can proceed confidently.
- **Creating the PR:** Once you've verified the branches are correct and there are no conflicts, click **"Create pull request"**.

Filling in Pull Request Details

- **Title:** Add a clear, descriptive title for your PR (e.g., "Update README").
  - **PR Title Best Practices:** The title should clearly describe what the pull request accomplishes. Like issue titles, start with an action verb and be specific. Good PR titles read like a changelog entry: "Add user authentication system," "Fix memory leak in data processing module," "Update dependencies to latest versions," "Refactor database query logic for performance." The title will appear in the project's history and in email notifications, so make it informative. Some teams follow conventions like prefixing with the type of change: "[Feature]," "[Bugfix]," "[Docs]," etc.
- **Description:** You have the option to add a longer description. Use this space to provide context about your changes (e.g., "Adding title and short description to improve project documentation").
  - **Writing Comprehensive PR Descriptions:** A great pull request description answers several key questions. What changes are included? Why were these changes necessary? How do the changes work? Are there any areas reviewers should pay special attention to? Are there any known limitations or future work required? Have tests been added or updated? Does documentation need updating? Including this information upfront saves time in the review process because reviewers don't need to ask these questions—they can focus on evaluating the code quality and correctness. Many teams use PR templates to standardize this information.

Assigning a Pull Request

- **Process:** You can assign a PR to a specific user the same way you assign an issue.
  1. Find the **"Assignees"** section on the right side of the PR page.
  2. Click the gear icon.

3. Select the user who should be responsible for merging this PR (typically the repository owner or project lead).

- **Assignment Responsibility:** The assignee of a PR is typically the person responsible for ultimately deciding whether to merge or close the pull request. This might be a team lead, project maintainer, or code owner for the affected files. Assignment helps clarify who has the final say on accepting the changes.
- **Completing the PR Creation:** Once you've filled in all the details and made any desired assignments, click **"Create pull request"** to officially open the PR.
- **What Happens Next:** Once created, the PR enters the review phase. Other team members will be notified, automated tests may run, and reviewers will examine your changes. The PR remains open and in discussion mode until it's either merged (changes accepted) or closed (changes rejected or abandoned).

3.4 Reviewing Pull Requests

How to Request a Review

- **Starting Point:** You have a PR that's been created and needs review before merging.
- **Process:**
  1. On the PR page, locate the **"Reviewers"** section on the right side.
  2. Click the **gear icon** (⚙) next to Reviewers.
  3. This provides a list of everyone who can review a PR for this repo.
  4. Select the person (or people) you want to review your changes.
- **Understanding Reviewer Selection:** Not everyone with repository access may appear in the reviewers list. GitHub intelligently suggests reviewers based on who has previously edited the files you're changing (since they're familiar with that code), who the code owners are (if the repository has a CODEOWNERS file), and who has previously reviewed similar PRs. You can request reviews from multiple people, which is common for complex changes that touch different areas of the codebase or require sign-off from multiple stakeholders.

Assign vs. Review: The Important Distinction

- **Assignee:** The person **responsible for approving and merging** the PR. The assignee is ultimately accountable for the pull request and has the authority to merge it.
- **Reviewer:** The person **responsible for looking at the changes** being suggested and providing feedback. Reviewers evaluate code quality, correctness, and adherence to standards, but they're not necessarily the ones who click the merge button.
- **In Practice:** While the distinction exists, in many teams the same people serve as both reviewers and assignees. A common pattern is that reviewers are peers who provide feedback, while the assignee is a lead or maintainer who ensures all feedback is

addressed and then performs the merge. In smaller teams or personal projects, these roles often overlap, and the same person might both review and merge.

How to Review a Pull Request

Switching Perspectives: Now we'll look at the PR from the reviewer's point of view—someone who's been requested to review changes.

1. Navigate to the **"Pull requests"** tab in the repository.
2. You'll see a view of the PRs that need your review (these are filtered to show PRs where you've been requested as a reviewer).
3. **Click on the PR** you want to review.
4. Navigate to the **"Files changed"** tab, where you can review all the proposed changes.

Understanding the Files Changed View

- **Diff View:** This view shows all the changed files and exactly what has been added or removed in each file.
- **Color Coding:**
  - **Red highlight** indicates a line that was **removed** (deleted code).
  - **Green highlight** indicates a line that was **added** (new code).
  - **White/Gray** indicates unchanged lines shown for context.
- **Understanding Diffs:** The diff (difference) view is the heart of code review. It shows the delta between the base branch and the compare branch—exactly what will change if this PR is merged. Lines with no highlighting are unchanged and included for context so you can understand the surrounding code. Reviewers should read through all changes carefully, understanding not just what changed, but why it changed and whether the change is correct. Look for bugs, security issues, performance problems, unclear variable names, missing error handling, and opportunities for simplification.
- **Line-by-Line Numbers:** The diff shows line numbers from both the old version (left, in red if removed) and new version (right, in green if added), making it easy to reference specific locations.

Adding Comments to a Pull Request

- **Inline Comments:** You can add comments to specific lines if you have questions or need further clarification.
- **How to Add an Inline Comment:**
  1. **Hover over** the line of code you want to comment on.
  2. Click the **blue square icon with a white plus sign (+)** that appears to the left of the line number.
  3. This opens a comment box where you can type your feedback.
  4. Click **"Add single comment"** to post immediately, or **"Start a review"** to batch multiple comments together.

- **Multi-Line Comments:** If you want to comment on multiple consecutive lines:
  1. **Click and drag** over the line numbers to select all relevant lines.
  2. Then click the plus icon that appears.
  3. Your comment will reference the entire selected range of lines.
- **The Art of Code Review Comments:** Effective code review comments are specific, constructive, and kind. Instead of "This is wrong," explain what the issue is and why: "This will throw a null reference error if the user object is undefined. Consider adding a null check here." Ask questions to understand intent: "Can you help me understand why we're using approach X instead of approach Y?" Praise good code: "Nice refactoring here—much more readable!" Use suggestion blocks (GitHub supports suggesting code changes directly) to make it easy for the author to incorporate feedback. Remember that there's a human on the other end who's invested work in this code; your comments should improve the code while respecting the person.

Responding to a Pull Request

- **Privacy Note:** At this point, your comments are only visible to you. They're not posted until you formally respond to the PR.
- **Submitting Your Review:** Once you've finished examining all the changes and adding comments, click the **"Review changes"** button (green button at the top-right of the Files changed tab).
- **Three Response Options:**
  1. **Comment:** Submit feedback without explicit approval. This is used when you have questions or suggestions but aren't formally approving or blocking the PR.
  2. **Approve:** Submit comments and approve merging the changes. This signals that the PR is ready to merge from your perspective.
  3. **Request changes:** Submit comments that must be addressed before merging. This blocks the PR from being merged until the issues are resolved.
- **Selecting the Appropriate Option:** The choice depends on the nature of your feedback. If you found critical bugs, security issues, or problems that must be fixed, choose "Request changes." If you have minor suggestions or questions but the code is generally acceptable, choose "Approve" (your comments will still be visible for the author to consider). If you need clarification before making a decision, choose "Comment." The example in the course shows selecting "Request changes" because there's one request that must be addressed before merging.

Comment vs. Request Changes: The Distinction

- **Comment:** Implies feedback or suggestions that are **not required** to be incorporated. The comments are advisory—the author can consider them and decide whether to address them.
  - **When to Use Comment:** Use this option when you have minor style preferences, optional optimizations, questions about future plans, or suggestions that aren't

critical to the current PR's success. Comments keep the conversation flowing without blocking progress. They're also useful when you're not a primary reviewer but want to contribute thoughts, or when you're reviewing for educational purposes to learn from others' code.

- **Request Changes:** Confirms that the comments **do include things that need to be incorporated** before the PR can be merged. This formally blocks the PR from being merged until addressed.
    - **When to Request Changes:** Use this when you've found bugs that would break functionality, security vulnerabilities, violations of project coding standards, logic errors, missing tests for new functionality, or anything else that must be fixed before the code can be accepted. Requesting changes is a serious action—it tells the author and other reviewers that this PR is not yet ready, and it typically prevents the PR from being merged (especially if branch protection rules enforce review approvals). Only request changes when necessary, and be clear about exactly what needs to be fixed.

Handling Requested Changes

1. **Author Receives Notification:** Once you select "Request changes" and submit, an alert reaches the PR author informing them that changes have been requested.
2. **Author Makes Changes:** The contributor makes the requested modifications to their code, commits the changes, and pushes them to the same branch.
    - **PR Updates Automatically:** One of the beautiful aspects of pull requests is that they're living entities. When the author pushes new commits to the branch that's associated with the PR, those commits automatically appear in the PR. You don't need to create a new PR—the existing one updates to include the latest commits. This makes the iteration cycle smooth: review, request changes, author updates, review again, approve, merge.
3. **Re-Request Review:** After addressing the feedback, the author can **re-request review** by clicking the "re-request review" icon (circular arrows) next to your name in the Reviewers section.
    - **The Re-Review Cycle:** Re-requesting review notifies the reviewer that the author has addressed their feedback and the PR is ready for another look. The reviewer can then examine the new commits, verify that their concerns were addressed, and potentially add new comments or approve the PR. This cycle can repeat multiple times for complex changes—it's a conversation, not a one-time event. Well-functioning teams embrace this iterative process as a collaboration tool rather than viewing it as bureaucratic overhead.

Approving a Pull Request

1. Once you're satisfied with all changes, navigate back to the **"Files changed"** tab and click **"Review changes"**.
2. This time, select **"Approve"** to indicate that the PR is ready to merge.

- **Approval Significance:** Your approval is your stamp of confidence that this code is ready to be integrated into the project. It means you've reviewed the changes, understand what they do, believe they're correct, and think they meet the project's quality standards. In repositories with branch protection rules requiring multiple approvals, each approval is a gate that must be passed before merging becomes possible. Take approvals seriously—you're vouching for the quality of the code and assuming shared responsibility for anything that might break.

3. Submit your approval.

Merging a Pull Request

Once the PR has the required approvals and all checks have passed:

1. Click the **"Merge pull request"** button.
    - **Pre-Merge Checks:** Before clicking merge, GitHub displays the current status of the PR, including whether all required reviewers have approved, whether automated tests have passed, whether the branch is up-to-date with the base branch, and whether any merge conflicts exist. In repositories with branch protection, the merge button will be disabled (grayed out) until all requirements are met. This prevents accidental merging of code that hasn't been properly reviewed or tested.
2. Click **"Confirm merge"** to finalize the operation.
3. **The Merge Action:** This integrates all the commits from the feature branch into the base branch. The changes are now part of the official codebase.
    - **Merge Strategies:** GitHub offers several merge strategies (which you can select from a dropdown). The default "Create a merge commit" preserves all commits from the feature branch and adds a merge commit. "Squash and merge" combines all commits into a single commit, creating a cleaner history but losing granular commit history. "Rebase and merge" replays the commits from the feature branch onto the base branch without a merge commit, maintaining a linear history. Most teams stick with the default merge commit strategy, but the choice depends on your team's preferences for Git history cleanliness versus preservation of detailed development history.

Deleting a Branch After a PR

- **Best Practice:** Once a PR is complete and merged, it's good practice to **delete the branch** where the changes came from to keep your repository clean.
- **Why Delete Branches:** Merged branches serve no further purpose—their changes are now incorporated into the main branch. Leaving merged branches around clutters your branch list and makes it harder to see which branches represent active work. A clean branch list improves repository navigation and makes it clear which features are in development versus which are completed. Many teams adopt a discipline of immediately deleting merged branches, treating branch creation as temporary scaffolding for development rather than permanent structures.

- **GitHub's Delete Prompt:** After merging, GitHub displays a **"Delete branch"** button on the PR page, making cleanup convenient.
- **Safety Net:** If you need to restore the branch (for example, if you discover an error that needs to be rolled back), you have the option to restore a deleted branch.
    - **When Restoration is Needed:** You might need to restore a branch if you merged too quickly and later discover a bug introduced by those changes, if you want to reference the original commits that were squashed during merge, or if you realize additional work related to that feature is needed. Git never actually deletes the commits—they're still in the repository's object database. Restoring a branch is just recreating the branch pointer to reference those commits again.

How to Restore a Deleted Branch

Option 1: From the PR Window

If you're still viewing the PR page immediately after deleting the branch:

- The **"Delete branch"** button will have changed into a **"Restore branch"** button.
- Simply click it to bring the branch back.

Option 2: From the Pull Requests Tab

If you've navigated away from the PR:

1. Go to the **"Pull requests"** tab.
2. The default view shows **open** PRs. Change the filter to show **closed** PRs (you'll see "0 Open" and the number of closed PRs—click on "Closed").
3. Find and click on the closed PR whose branch you want to restore.
4. Scroll to the bottom of the PR page where you'll see the **"Restore branch"** button.
5. Click it to restore the branch.

- **Restored Branch State:** When you restore a branch, it comes back exactly as it was when deleted, pointing to the same commits. You can then continue working on it, revert the merge if necessary, or extract specific commits as needed. This safety mechanism gives you confidence to delete merged branches knowing you can always recover them if circumstances change.

Chapter 3 Summary: Collaboration is where GitHub truly shines. By mastering cloning and forking, you can work with any repository whether you're a team member or external contributor. Issues provide a structured way to track work, communicate plans, and document decisions. Pull requests create a review gateway that ensures code quality and spreads knowledge across teams. Together, these features transform GitHub from a simple code hosting platform into a complete collaboration environment. The workflows might seem

complex at first—fork, clone, branch, commit, push, pull request, review, merge—but each step serves a purpose in maintaining code quality and enabling safe, effective teamwork. As you practice these workflows, they'll become second nature, and you'll appreciate how they prevent the chaos that can arise when multiple people modify the same codebase without structure. Whether you're contributing to a major open-source project or collaborating with a small team, these are the fundamental patterns that make successful software development possible.