General information about the subject:
During this course, we will cover the basics of working with the C# language and the construction and architecture of server applications. We will focus on various practices and architectural patterns useful during application development. We will communicate with a relational database. At the end of the course, we will spend some time on building graphical applications.

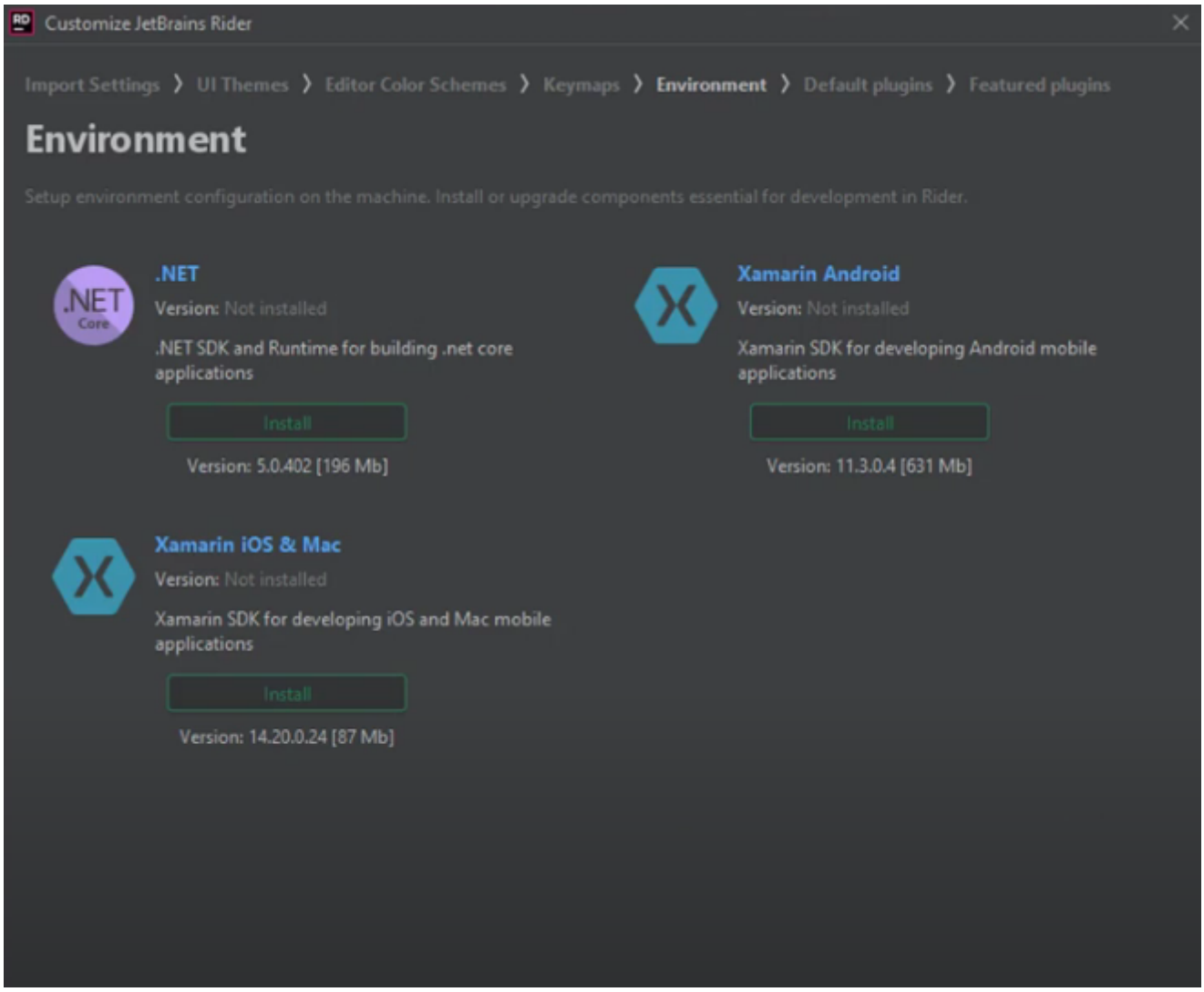
Passing rules:

- Exercises - 12 points (need to score at least 6),
- Midterm 1 - 20 points (need to score at least 10),
- Midterm 2 - 20 points (need to score at least 10),
- Lecture – 5 points (points awarded based on attendance)
- Project - non-compulsory, for grade 5
- Total - 57 points.
- There is an exam during the exam session

What to install to work with .NET?

IDE options:

1. Rider (recommended) - https://www.jetbrains.com/rider/download/#section=mac
   During Rider installation you can at the same time install .NET SDK (we do not need Xamarin).



2. Visual Studio Community - https://visualstudio.microsoft.com/pl/vs/community/
3. Visual Studio Code - https://code.visualstudio.com/

Installing .NET SDK separately

1. .NET SDK - version 7.0 or above should be fine. Unfortunately on school computers we have version 7.0, so most likely this one will be the most safe. https://dotnet.microsoft.com/en-us/download/visual-studio-sdks

---

Working with terminal

During our classes we will be using from time to time terminal. We assume that each of you understand the basics terminal commands.

As a quick reminder below you can find the list of the most basics command which will be useful during our classes.

Navigation

- cd folderName
- cd ..
- cd "C:\some\path"

Creating directory

- mkdir nameOfDir
- rmdir nameOfDir

Listing the content of the location:

- ls

Clear the console

- clear

Display current path:

- pwd

In case you need a longer reminder you can check the following materials:
https://www.youtube.com/watch?v=5XgBd6rjuDQ
https://www.youtube.com/watch?v=Us3G-nJYru0

Problem

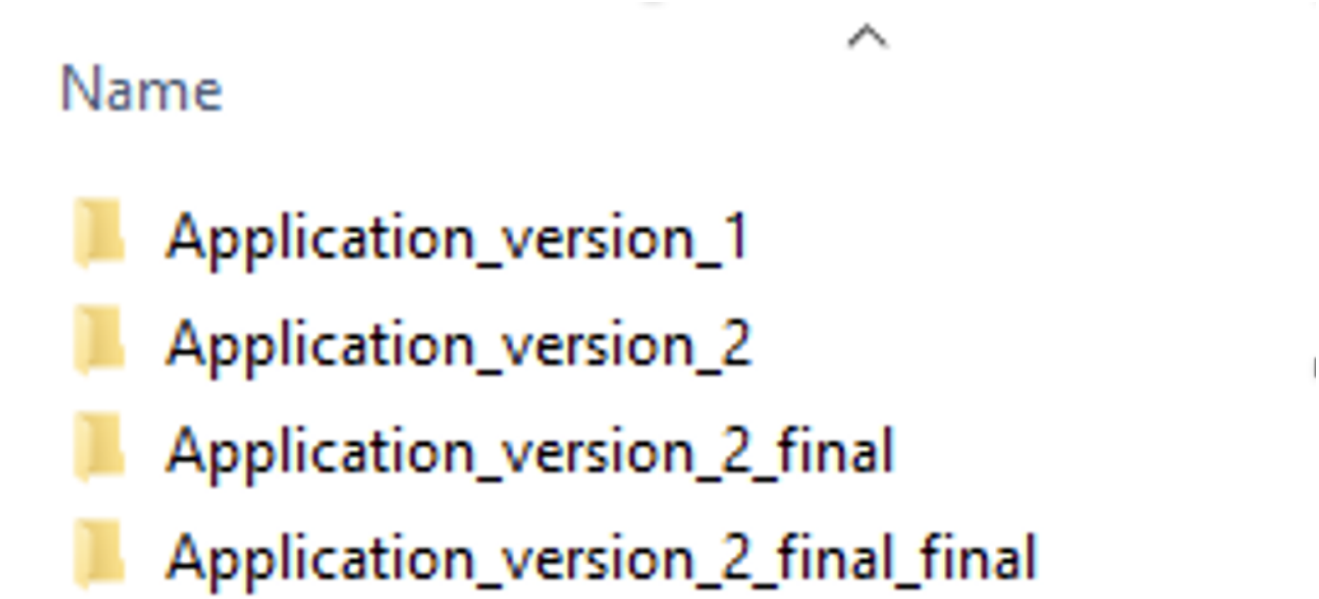
When we work on a IT project we can encounter a few problems:

- usually you will have to share your code with other developers

- even if you are working on your personal project - you do not want the data to get lost
- usually you would like to get a clear view and be able to get back to the previous version of your code
- maybe you are working on multiple versions of the same codebase and you want to easily work together

---

We do not want to do this



Version control systems

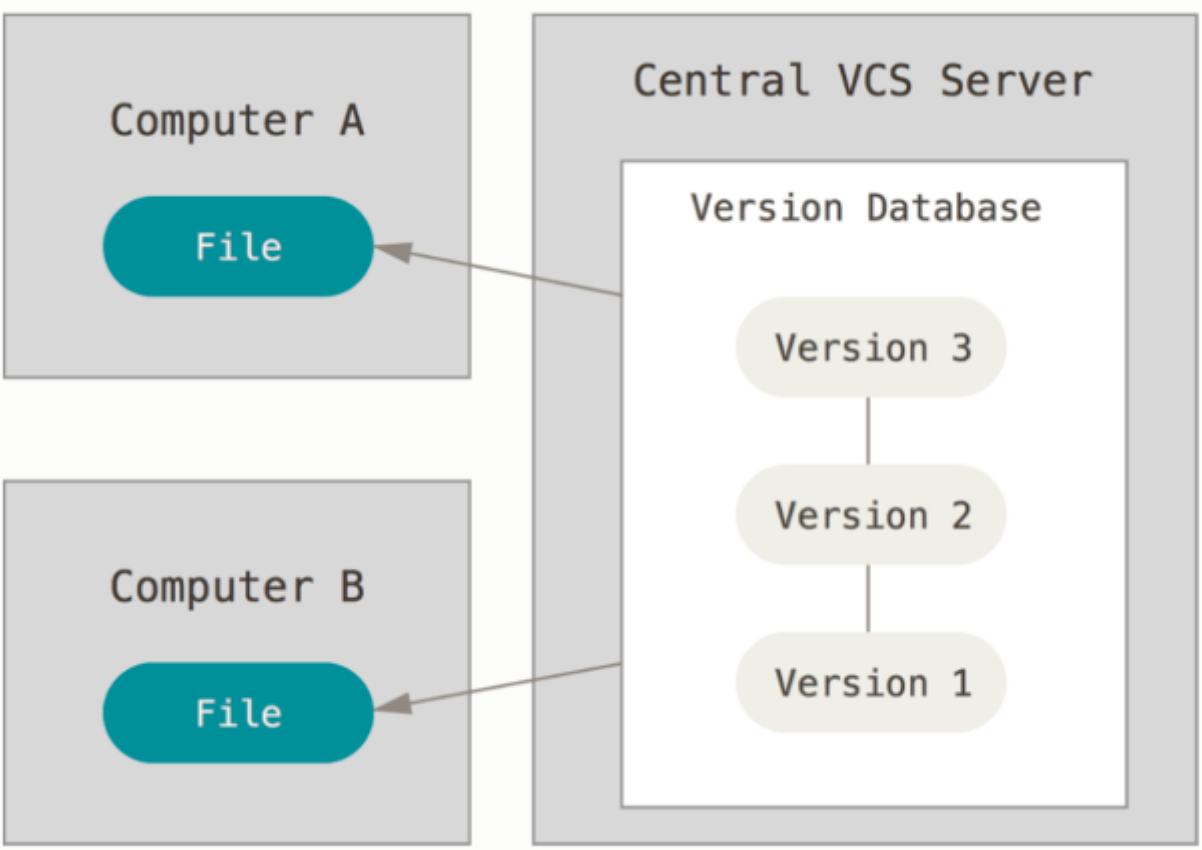
•Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

•If you are a graphic or web designer and want to keep every version of an image or layout you have designed

•If you are a programmer you want to keep track of the changes in code (especially if you are working in a team).

•It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.

Centralized version control systems



Centralized Version Control Systems (CVCS) are systems where all the versioned files are stored on a central server, and team members interact with this central repository to check out files, make changes, and then commit these changes back to the repository.

This model offers a simple and straightforward approach to version control, where the central server acts as the single source of truth for the project's history and current state.

Key Characteristics of Centralized Version Control Systems

Single Repository: All project files and historical data are stored in one central location. Users need to connect to this central server to upload or download changes.

Access Control: The central server can enforce access controls, determining who can read from or write to the repository, which helps in managing permissions and security.

Dependencies on Network: Since the repository is central, users must have network access to interact with it, which can be a limitation for offline work.

Visibility: With a single repository, it's easier for administrators to monitor and manage the project's development, including tracking changes and backups.

Examples of Centralized Version Control Systems

Subversion (SVN): One of the most popular CVCS, known for its reliability and support for binary files.

Concurrent Versions System (CVCS): An earlier system that was widely used before being largely replaced by more modern systems like SVN and distributed version control systems (DVCS).

Limitations of Centralized Version Control Systems

Single Point of Failure: If the central server goes down, no one can collaborate or save versioned changes until it's back up.

Scalability: Large projects with many contributors may face challenges due to the centralized nature, especially in terms of performance and workflow
bottlenecks.

Less Flexibility for Offline Work: Since changes need to be pushed to and pulled from the central server, working offline is more cumbersome compared to distributed systems.

Distributed version control systems

Distributed Version Control Systems (DVCS) represent a model of version control where each contributor to a project maintains a personal copy of the entire repository, including its full history.

This approach decentralizes version control, allowing for more flexibility and redundancy compared to centralized version control systems (CVCS). In DVCS, synchronization of changes between repositories is facilitated through push and pull operations among peers or through a central repository that acts as a hub.

Key Characteristics of Distributed Version Control Systems

Full Repository Copies: Each user has a complete local copy of the project repository, including all branches and history. This allows for a wide range of actions to be performed offline, such as commits, history viewing, and branching.

Peer-to-Peer Sharing: Changes can be shared directly between users or through a central server. This flexibility enhances collaboration and allows for various workflows.

Enhanced Redundancy: Since every contributor has the full repository, the risk of data loss due to server failure is minimized. Each clone is effectively a full backup of the repository.

Branching and Merging: DVCS tend to handle branching and merging more efficiently and intuitively than CVCS. This supports a more dynamic development workflow with less risk of merge conflicts.

Increased Performance: Many operations are faster in DVCS because they only involve local files and not network communication. This includes viewing history, committing changes, and switching between branches.

Access Control: While DVCS allows every user to have a full copy of the repository, access control mechanisms can still be implemented, especially on central or official repositories used for collaboration.

Examples of Distributed Version Control Systems

Git: Perhaps the most popular DVCS today, known for its efficiency and support for complex branching models.

Mercurial: Similar to Git in many ways, Mercurial is praised for its simplicity and ease of use.

Bazaar: Another DVCS that focuses on usability and flexibility, suitable for everything from small projects to very large projects.

Advantages of Distributed Version Control Systems

Resilience: The distributed nature means there's no single point of failure. If a central repository becomes unavailable, any of the cloned repositories can be used to restore it

Flexibility: Developers can work independently on their local repository with all version control features available, then synchronize their changes with others as needed.

Collaboration: DVCS supports various collaboration models, from highly centralized to completely distributed, allowing teams to choose the workflow that best fits their project.

Challenges of Distributed Version Control Systems

Learning Curve: DVCS can be more complex to understand and use effectively, especially for those accustomed to the centralized model.

Repository Size: Since each clone includes the entire history, the repository size can become an issue, particularly for projects with large binaries or long histories.

Birth of Git

Back in 2005, the Linux kernel (the core of the Linux operating system) was growing rapidly, and developers from all over the globe were contributing to it.

Linus Torvalds, the creator of Linux, faced a predicament. They were using a version control system that suddenly wasn't available

anymore.
Linus needed a new tool that could handle the massive scale and speed of development without causing chaos. So, Linus created Git.

Git - description

Git allows developers to take "snapshots" of their projects. Each snapshot captures the entire state of the project at a moment in time. You can go back to any snapshot if you need to see who made which changes, or if you just want to return to a version before you accidentally introduced a bug that turned your project into digital confetti.

Git lets you create parallel realities (called "branches") for your project. In one branch, you could be adding new features, while in another, you're fixing bugs.
These branches can then be merged back into the main project, combining all your work seamlessly.

Git's capabilities quickly made it the go-to version control system for developers worldwide. It's not just for coding, though. Writers, designers, and anyone who needs to track changes and collaborate on digital projects can use Git.

GitHub

GitHub - a social network meets Git repository hosting service, where developers share their projects, contribute to others, and collaborate on code. We will be using GitHub during our classes.

How to install Git?

Git is a console (command line)tool. ==
Event though there are GUI tools for Git, at least at the beginning ==we will use git from within the console.
This should help us understand exactly how Git works and how to use all it's capabilities. Usually none of the GUI tools allows you to use all it's functionalities.

For Windows

1. Download the Installer: Go to the official Git website (git-scm.com) and download the latest Git for Windows installer.
2. Install: Run the downloaded installer. During the installation, you can stick with the default settings, which are suitable for most users. Adjust options if you have specific preferences.
3. Verify Installation: Open Command Prompt (cmd) or Git Bash (a special terminal for Git commands) and type git --version. If Git is installed correctly, you'll see the installed version number.

For macOS

1. Install with Homebrew: If you have Homebrew (a package manager for macOS), you can install Git by opening the Terminal and running brew install git.
2. Manual Install: Alternatively, you can download the installer from the Git website, similar to Windows, and follow the prompts.
3. Verify Installation: Open the Terminal and type git --version to check if Git was installed successfully.

For Linux

1. Use a Package Manager: Most Linux distributions include Git in their package managers. For Debian/Ubuntu-based systems,

open a terminal and run sudo apt-get install git. For Fedora, use sudo dnf install git.

2. Verify Installation: In the terminal, type git --version to ensure Git is installed.

Git - configuration

Git configuration is a flexible and powerful feature that allows you to customize your Git environment.
Configurations in Git are stored in text files and can be managed at three levels: system, global, and local.

System-Level Configuration

Where It's Stored: System-level configuration applies to every user on the system and all their repositories. The configuration file is located in a system-wide directory (/etc/gitconfig on Linux, for example).

How to Set: Use git config --system to set configurations at this level.

Global-Level Configuration

Where It's Stored: Global configuration is specific to your user profile on the system. The configuration file is typically found in your home directory (~/.gitconfig or ~/.config/git/config).

How to Set: Use git config --global to set configurations that will apply to all of your projects.

Common Settings: It's recommended to set your user.name and user.email at this level, as these identifiers will be used in your

commits across all projects on your system, ensuring consistency and traceability.

Local-Level Configuration

Where It's Stored: Local configuration is specific to a single repository and is stored in .git/config within the repository's directory.

How to Set: Use git config --local (default if you're within a repository) to set configurations that should only apply to the current project.

Common Settings: Project-specific settings like remote repository addresses or branch configurations are best set at this level.

Important Configurations to Set Up

user.name and user.email: These are crucial for identifying the author of commits in Git. They help maintain accountability and collaboration in team environments, as each commit is associated with the author's name and email.

auto.crlf: This setting is important for projects shared across different operating systems. It handles the automatic conversion of line endings to LF (Linux, macOS) or CRLF (Windows) on commits and checkouts. Setting auto.crlf to true on Windows or input on Linux and macOS helps prevent issues with line endings in a cross-platform development environment.

For example

```
git config --global user.name "Piotr Gago"
git config --global user.email "pgago@pjwstk.edu.pl"
```

For Windows users:

```
git config --global core.autocrlf true
```

For macOS users:

```
git config --global core.autocrl input
```

Git - help

You can get help about specific command in git using:

```
git config --help
```

In the example above we get help about config command.

Additional console tools

You can install additional tool which will make using the git from within a console a little for visually appealing.
Here is a plugin for Windows Power Shell terminal.
You can find similar plugins for other types of terminal.

How to initialize a new repository in the current location?

Git allows us to track all the changes within a specific git repository. Usually this simply means that all the changes within specific directory are tracked. Then we can save those changes in a local database stored in **.git directory**.

To initialise the repository in the current location you can run the following command:

```
git init
```

This command will intialize new repository. Witin the same locatino we should see new folder being created.
This folder is called .git and it will represent our local database.
Remember that by default this folder is hidden so you might need to change the setting sin your respective OS to show the hidden files and folders.

How Git stores data?

Git stores the data using the **key-value store** (or content addressable file system).
We can insert into the database any kind of file/data. Each file will get a key which we can later use to retrieve it.

We call this tyle of store "object database". Our values are called "**objects**".
They key is generated using SHA1 algorithm base on object's content. This give us a key with 160 bits length.

Key (SHA1)	Example Content
0cc175b9c0f1b6a831c399e269772661	Example file content for a blob object.
92eb5ffee6ae2fec3ad71c777531578f	Another example file content for a different blob object.
4a8a08f09d37b73795649038408b5f33	Example directory listing for a tree object.
8277e0910d750195b448797616e091ad	Commit object example content with parent commit, tree reference, and commit message.

Key (SHA1)	Example Content
0cc175b9c0f1b6a831c399e269772661	Hello, World! This is a simple text file.
92eb5ffee6ae2fec3ad71c777531578f	`function helloWorld() { console.log('Hello, world!'); }`
4a8a08f09d37b73795649038408b5f33	`100644 blob 0cc175b9c0f1b6a831c399e269772661 hello.txt\n100644 blob 92eb5ffee6ae2fec3ad71c777531578f hello.js`
8277e0910d750195b448797616e091ad	`tree 4a8a08f09d37b73795649038408b5f33\nparent 0000000000000000000000000000000000000000\nauthor Author Name <author@example.com> 1616262626 +0200\ncommitter Committer Name <committer@example.com> 1616262626 +0200\n\nInitial commit.`

```
Key=SHA1(object)
```

Remember that SHA1 will return the same key for the same input data. This means that the same objects will have the same key.

To minimize the size of our database, Git is compressing the objects using zlib library.

Key (SHA1)	Compressed Content
0cc175b9c0f1b6a831c399e269772661	`x□óHÍÉÉ×QÏ/ÊIQTÉÈ,V□¢DàÌÜ□□T□Ô□□´Ì□T=□$o□`
92eb5ffee6ae2fec3ad71c777531578f	`x□K+ÍK.ÉÌÏSÈHÍÉÉ□Ï/ÊIÑÐT¨VHÎÏ+ÎÏIÕËÉO×P÷□Éé(□□d□Õ5□j□□j□□`
4a8a08f09d37b73795649038408b5f33	`x□MË;□□O□ÐÝSö□¦ô□å8□!Æ4é □Ç×Mßþ F,%È□□¢*P□Öè Ø[□ÍÌ□□□"□×ncÌõº¯□~□□Iu7ÃnÉMsß□□□j□JÍ¿y□□Þê□□`
8277e0910d750195b448797616e091ad	`x□+)JMUOI´H4°H3°L160276·453±40¶01°H2M36æ*H,JÍ+QO □p%□□dä□)8B(¿ÄÜT□□□□CjEbnANª^r~®□□i□i□□□*h□□□□Es3KJR□□□á,□v □□¸<ó2K2□s□ □ö□Dÿ@ò`

.git directory

Database is stored locally in .git folder within specific repository. All the objects are stored in objects directory. The content of each object is stored in a file.

For example:

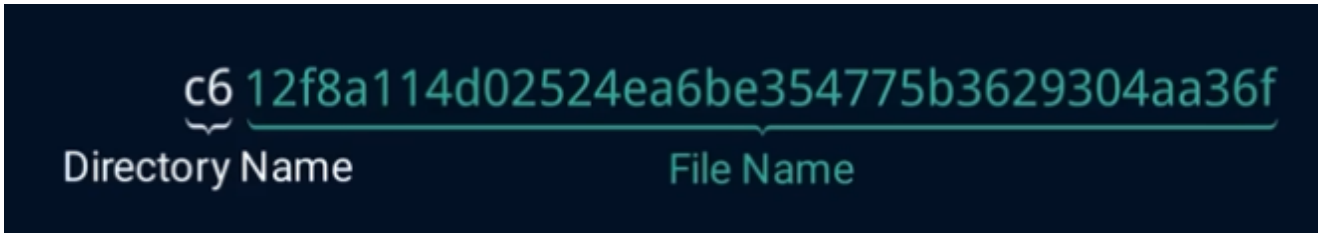
```
ls -a .git/objects/
```

The object file is named after its key.

```
ls -a .git/objects/1d
```

Object name

The first byte of the key is used to name the subdirectory. Remaining part is used for file containing the object.



When we create the new object with the name above - the first byte (c6) is used to create subdirectory.

Why Git is creating those subdirectories?

Git is using this technique to limit the number of files in single directory. This is because some of the file systems have certain limitations when it comes to number of files in a single directory. for ex. FAT32 - allow only around 65000 files per folder.

Example

Let create a file and store it in our object database.
First let's create new repository and add a single Java file.
Then let's add this file to the database.

```
git hash-object src/Program.java -w
```

This gives us the key for this object.

Now let's read the content of the file from our database.

```
git cat-file blob 6ab2c....
```

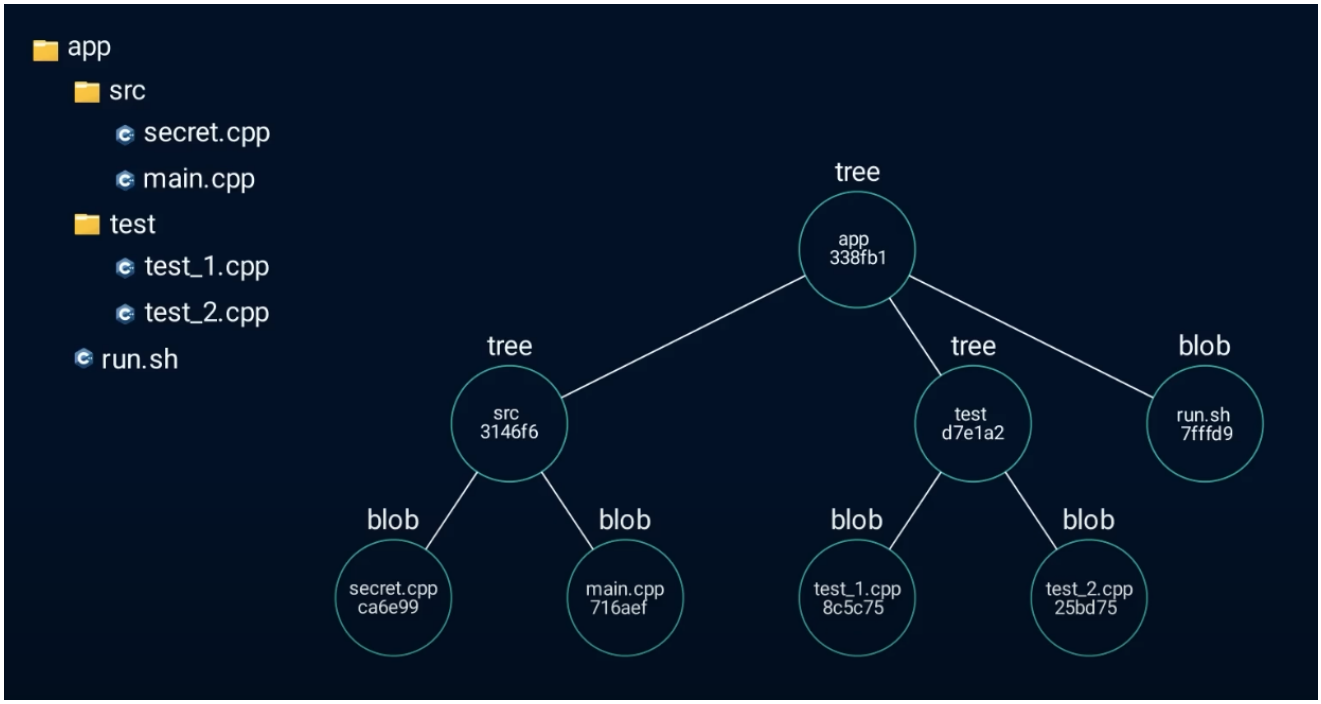
This will return us the content of the file.
Notice that within this object we are not storing anywhere metadata related to file. **This file is called "blob".**

Now let's modify our Program.java file and store it again.
As you see it should get a different key - this is because the key is generated based on the content of the file.

Tree - answer to the question where is the metadata?

Tree is another object type - other than a blob.

We can think of tree as a directory. It can contain other files and directories (trees).



Tree

We inspect one of the tree using the following commands:

```
git log --oneline
git ls-tree 4655cf33...
```

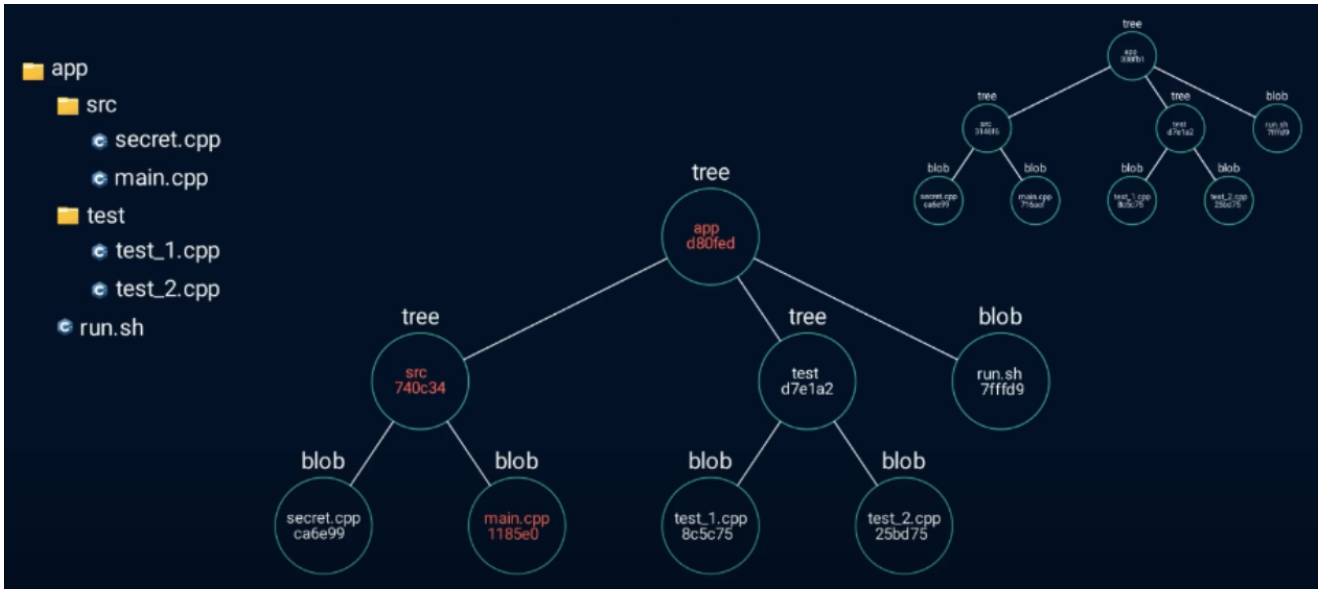
This will display all objects within the selected tree.

```
freezing@technodrom /tmp/app (master) $ git log --oneline
465cf55 (HEAD -> master) Initial commit
freezing@technodrom /tmp/app (master) $ git ls-tree 465cf55
100644 blob 313d088bdfddb59ae68acdfec4205335be3c6df7    run.sh
040000 tree e2767aea34481a41c629fa7d80533d35a1cfac5e    src
040000 tree b214487763e9e4695abd5b0030d18f72e9e261e7    test
freezing@technodrom /tmp/app (master) $ git ls-tree e2767aea34481a41c629fa7d80533d35a1cfac5e
100644 blob 96a1a5ad3f96db243404b5288772539f45920cda    main.cpp
100644 blob 123516dc181bbdd925722fd2928d3113ba1003f1    secret.cpp
freezing@technodrom /tmp/app (master) $ git ls-tree b214487763e9e4695abd5b0030d18f72e9e261e7
100644 blob 246cfcf58639958d2ab1ac293f2ee528dfc81ec0    test_1.cpp
100644 blob 1dd71c1800c59d44964b1d043da7f0f6bc4ab2d6    test_2.cpp
freezing@technodrom /tmp/app (master) $
```

What happens when we modify a file?

Usually the steps involved are:

1. we generate the new key and we store the object in database (our .git directory)
2. we generate new tree and store the modified file within this tree

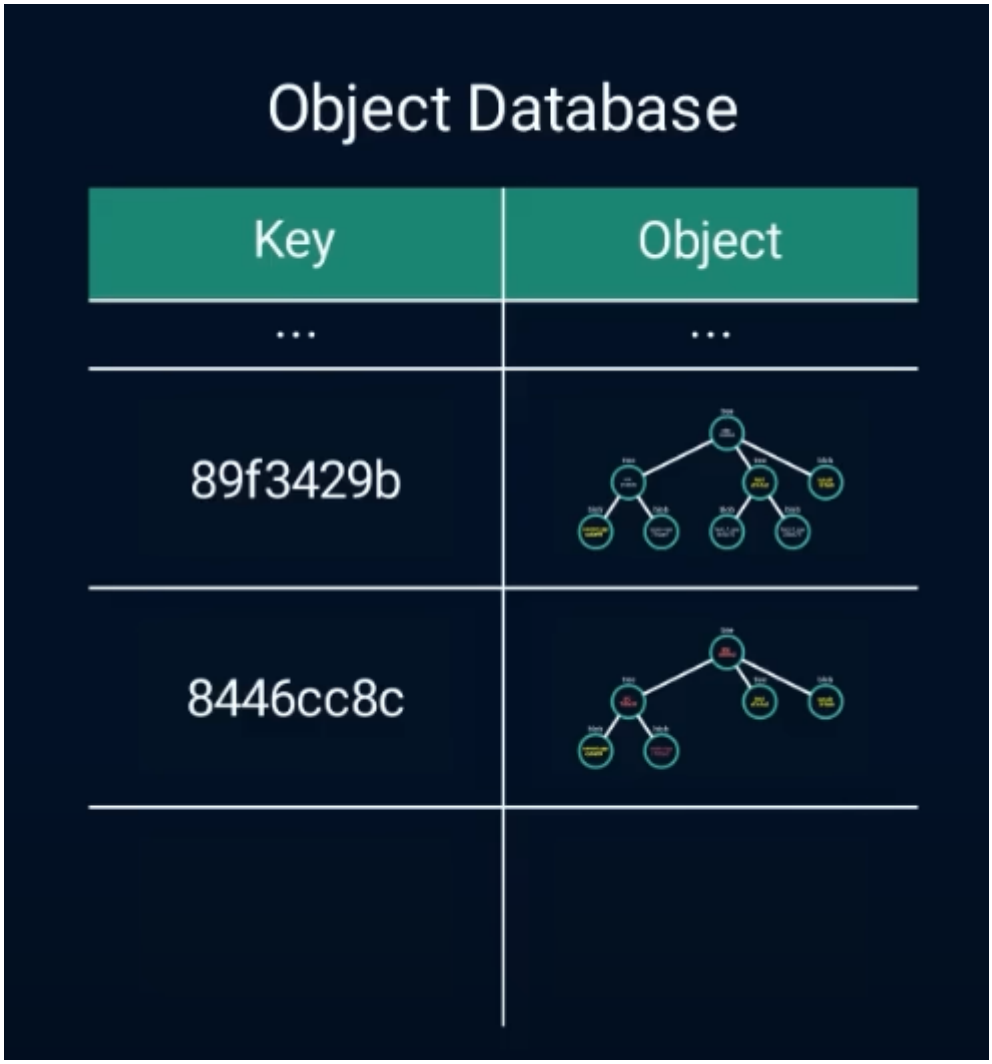


On the example above let's assume we modified the main.cpp. All the nodes up to the root has to be modified and generated again, but we see that portion of the tree do not have to regenerated

This works very well for large projects because we usually do not modify all the files.

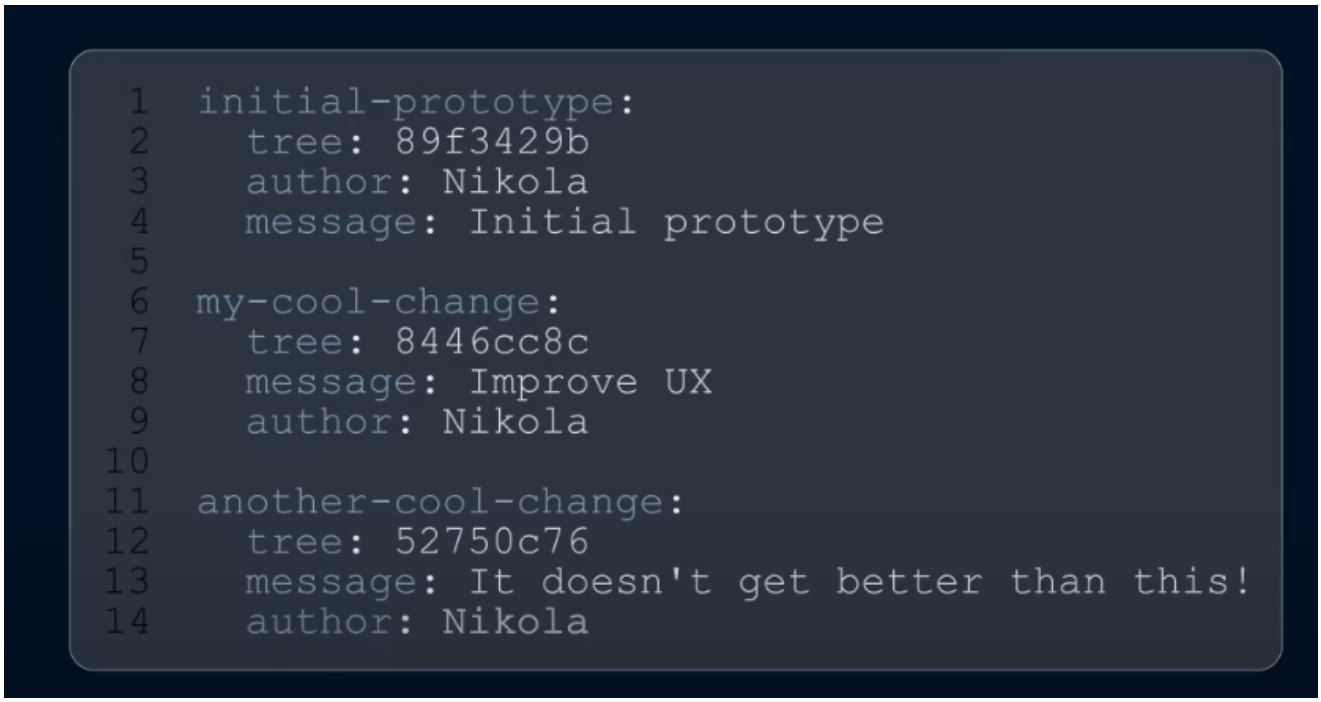
Tree

Out tree is stored in our object database.



Commit

Commit is another type of object which we can store.



It would be pretty hard to remember all those keys (sha1 values) and quickly identify where our data is located. This is why we added to our set of objects commits.

Commit is another type of the object we can store in database.
Commit id represented by SHA1 values. Commit represents change. Each
commit has a reference to the previous commit (or change).

Moreover each commit stores the hash value of the tree it is associated with.

We can use commit id to retrieve the state of the project from any time.

Branch

Unfortunately, commit id values are not easy to remember.

Branch is a reference to commit. We can give a branch any name we want.

Mutable and immutable

Commit ids are immutable - the ids are generated based on commit content.

Branches are mutable - often updated every time we make a change, branch reference is updated to point to the newest commit.

Each time we commit a new commit - current branch is updated automatically.

Summary

- Git stores object in the object database
- Object types: commits, trees and blobs
- A blob represent content of a file without metadata
- A tree represents a directory and it consists of other trees and blobs
- A commit is a snapshot of the project with additional information like author, time, message. It points to previous commit and it has a reference to root tree.
- Commit ids are SHA1 hashes
- A branch is a reference to a commit id using a friendly name

---

Scenario 1: using Git in a single person project

Knowing all the information let's see how we can save the changes.

```
// Step 1 - create new repository
mkdir gakko-app
cd gakko-app
git init
```

Let's create new Java console project within the created directory.

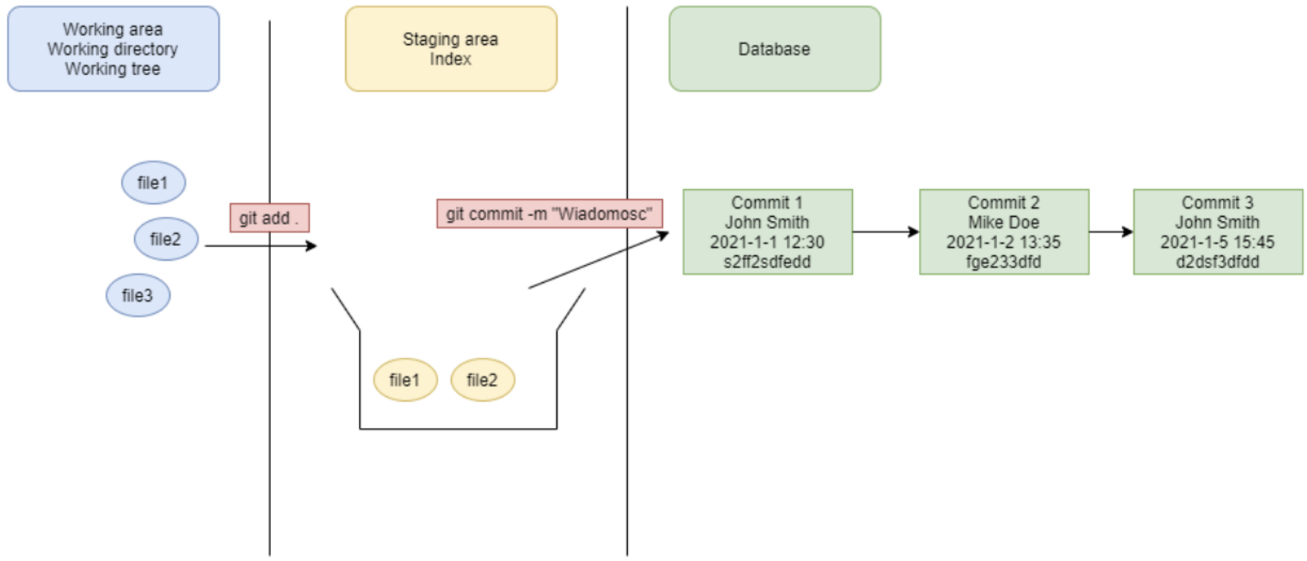
```
// Step 2 - we try to create new project
git status
git add .
git commit -m "Added new project"
```

---

1. **`git status`**:
   - This command is used to display the state of the working directory and the staging area. It shows you the changes that have been staged (with `git add`), changes that are not staged yet, and files that are not being tracked by Git. It's a good way to get an overview of what's happening in your repository before you commit any changes.
2. **`git add .`**:
   - This command adds changes from all files in the current directory (and its subdirectories) to the staging area, preparing them to be included in the next commit. The dot `.` represents the current directory, so this command tells Git to track changes in all files in the current directory and below. This is useful when you've made changes to many files and want to commit them all at once.

3. `git commit -m "Added new project"`:
    - This command is used to save your changes to the local repository. With the `-m` option, you're able to add a commit message directly from the command line, where `"Added new project"` is the message describing what changes the commit contains. Commit messages are important for keeping a clear history of what changes have been made and why. This command only affects your local repository; it doesn't affect the remote repository until you push the changes.

And now all the steps in a single picture



GitHub

https://github.com/

Sites like GitHub are used as platforms for version control and collaboration, allowing individuals and teams to work together on projects from anywhere.

These platforms are built on top of Git, which is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Here's a bit more detail on why we use sites like GitHub and their association with Git:

- Version control
- Collaboration
- Open Source and Private Projects
- Integration and Automation
- Documentation and Community

Please create account on GitHub using school email address. If you already have the account - simply add school email address to already existing account

---

Scenario 2 - GitHub and Git

This time we will try to use Git in conjunction with GitHub. We would like to work on our project and also upload the changes in our code to remote repository.

Step 1 - create new repository on GitHub

Create new repository on GitHub. During the creation of the repository you can immediately create README.md file and .gitignore file. Git ignore is a file which allows us to ignore certain files which should not be a part of the repository.

Step 2 - clone the repository

Then we would like to copy the repository to your desktop.

```
git clone url
```

Step 3 - add your project and create new commit

Create new Java console project and put it in the created directory.

```
git status
git add .
git commit –m "Initial project"
```

Step 4 - push the changes to the server

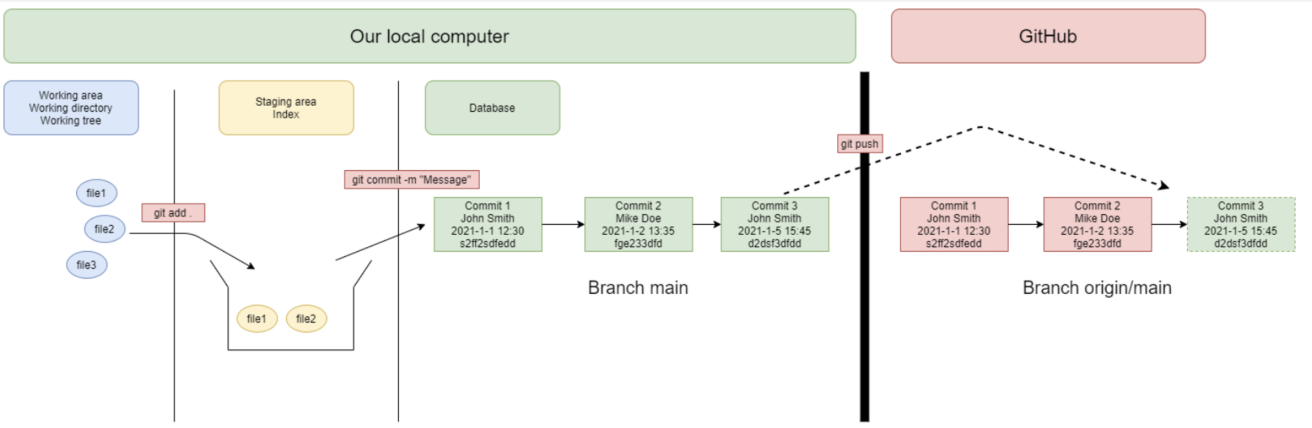
Let's push the changes to the server.

```
git push
```

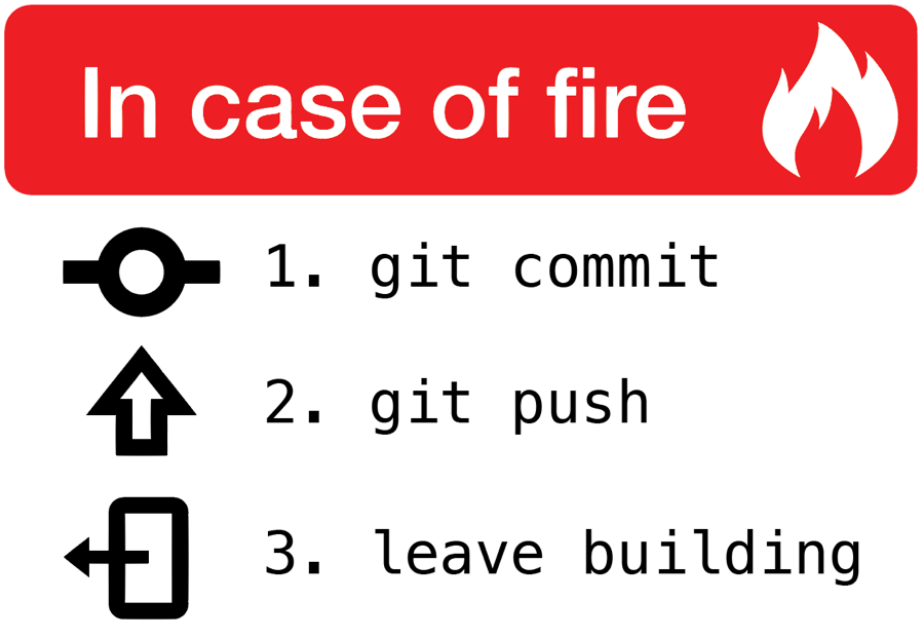
Check whether the changes are visible on GitHub.

Summary of scenario 2

Below we have all the previous steps in a single picture.

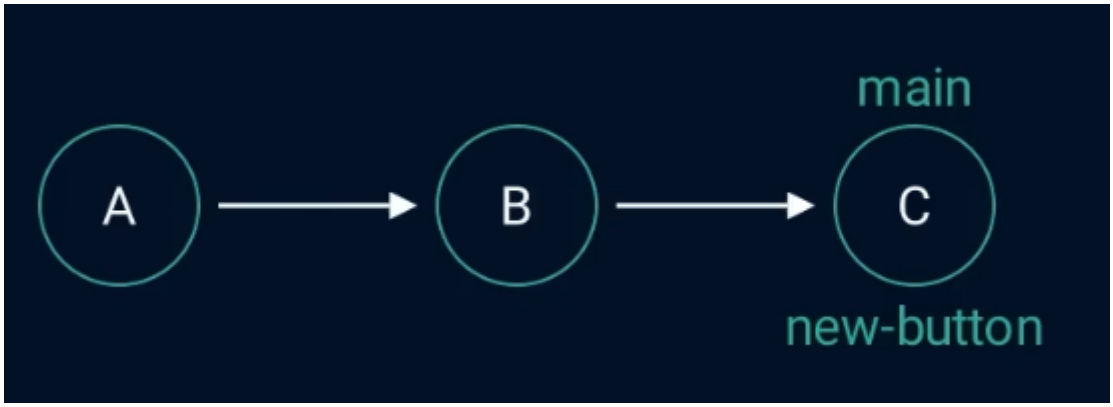


Remember to push your changes



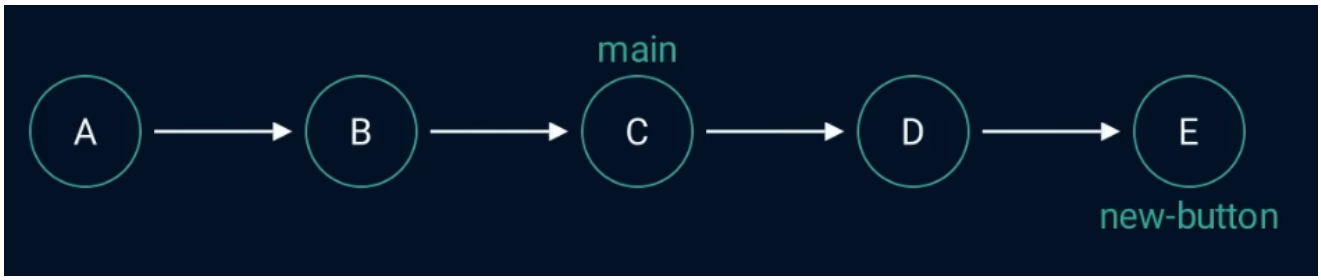
Scenario 3 - working on a new feature on a different branch

Let's assume we have a branch. On this branch we already have a few commits. This is a main branch.
We are asked to work on a new feature - adding a new button to the application. We decide to work on a new feature on a different branch - "new-button".



At the beginning we can see that our main branch and new-button branch are both pointing to the same commit.

After that we added a few commits to implement new feature. After that our branch looks like this.



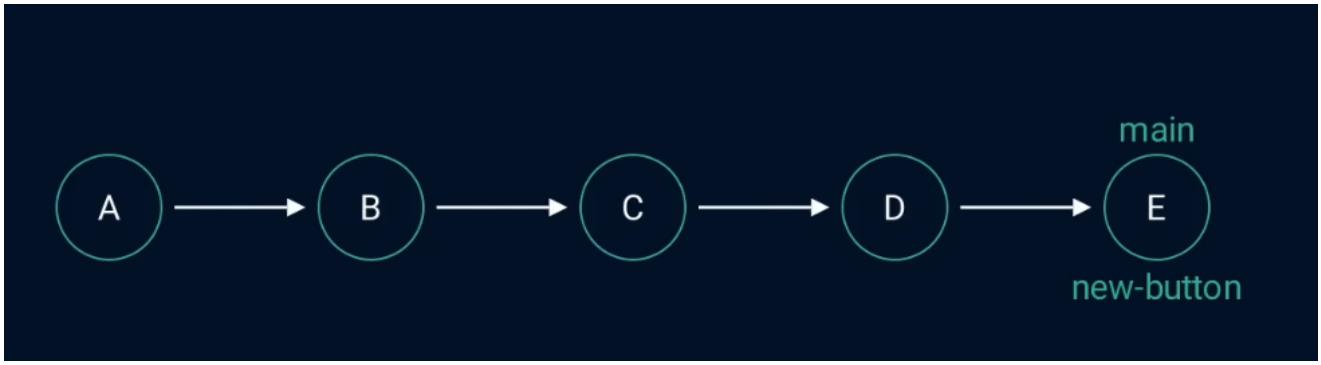
Merging - fast-forward merge (two-point merge)

After finishing the new feature we would like to update the main branch to point to the new commit.
Joining two branches together is usually referred to as **merging**.

To perform a merge we can use:

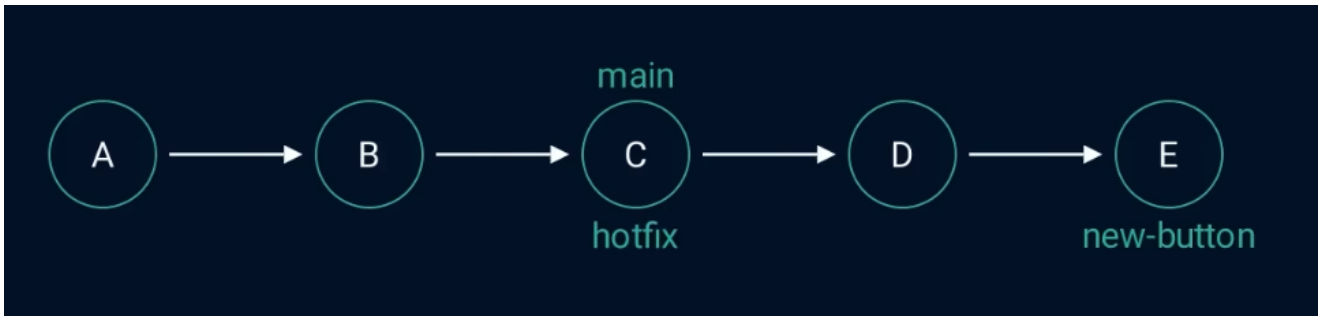
```
git checkout main //first we should switch to the branch we are merging into
git merge new-button //then we are performing the merge
```

After the operation the branch should look similar to the image below.

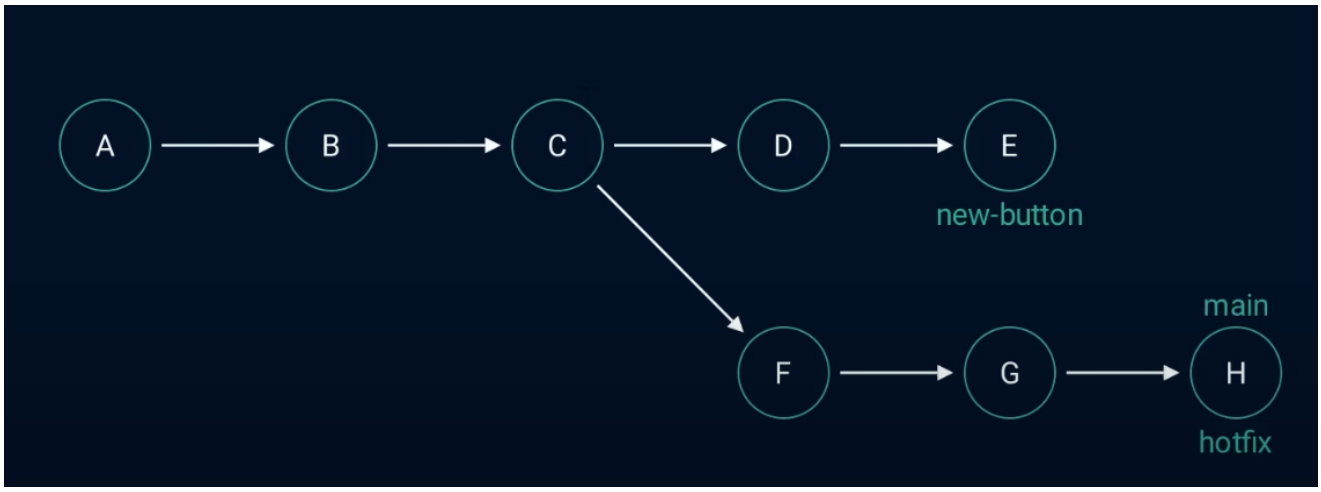


Scenario 4 - new-feature and divergent branches

Let's assume that while we were working on the new feature - we had to introduce a new hotfix to the main branch of our application. Then we merged the hotfix into the main branch.

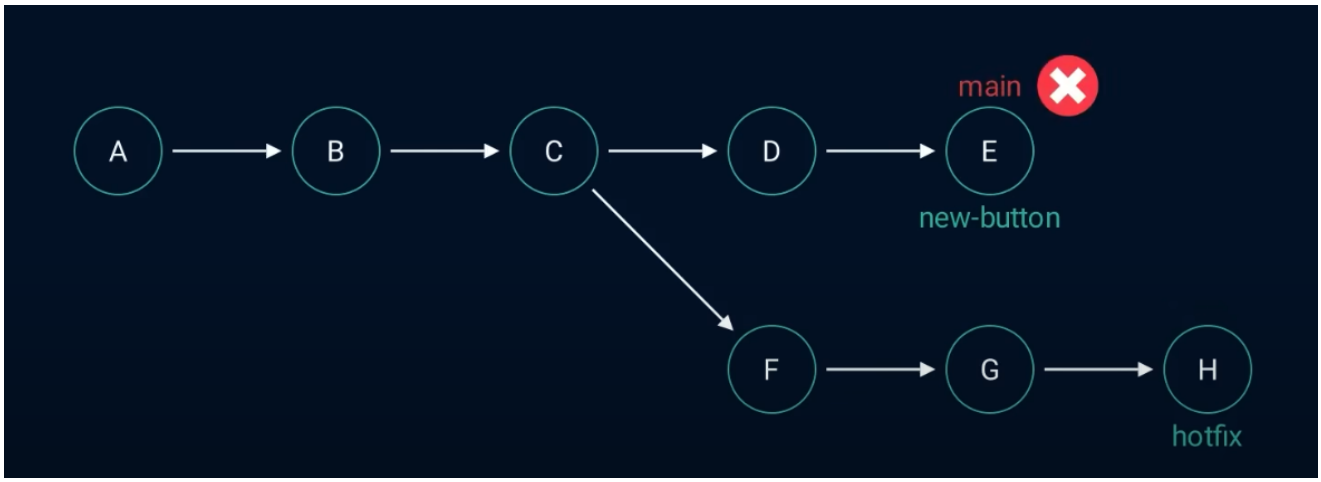


After performing those operations the history of our repository looks similar to the picture below. As we can see - the branches diverge in C node.



Now what will happen when we would like to merge the new-button branch into main branch. We cannot do it easily. This is because both branches contain different code versions. We have to somehow join togehter two different version of the code.

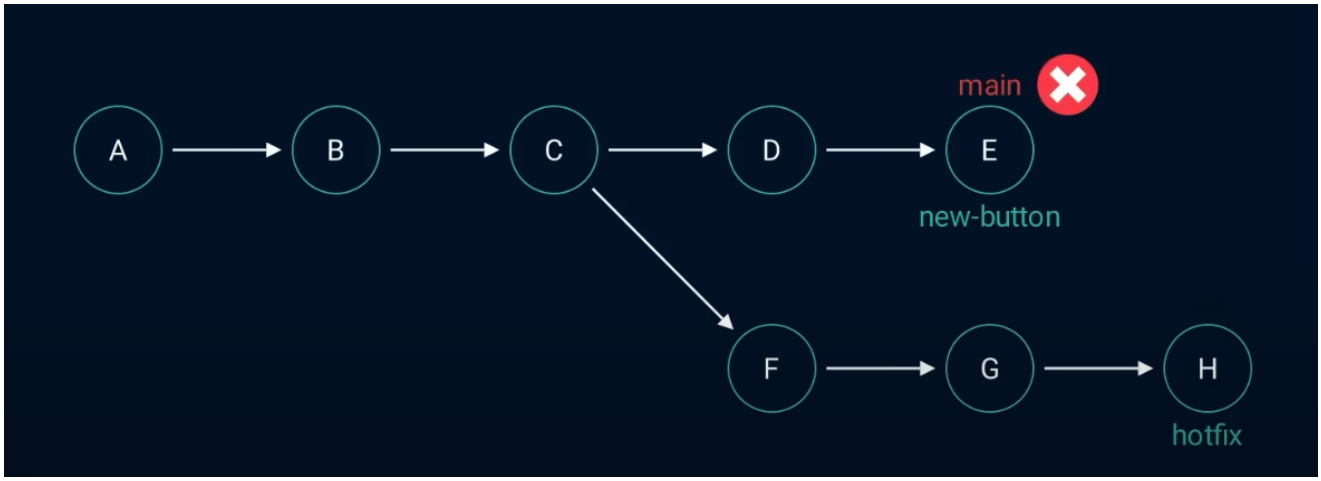
If we simply move the main pointer to the new-button commit - we would loose changes introduced on the hotfix branch.



Strategy 1 - 3-way merge

One way to deal with our problem is to perform a 3-way merge.

1. We find commit common to both branches
2. We compare snapshots from both branches (main and new-button) to the common commit - C.



We compare the branches:

- E \ C = D1 - we compare commit E and C and get the difference between them D1
- H \ C = D2 - we do the same with commit H and C

We can use the diff command to see the difference between two commits.

```
git diff 4566 34222
```

Option 1 - changes D1 and D2 do not overlap

If developers edited different files on both branches - then changes most likely do not overlap. We can safely merge both changes D1 and D2 into a single commit. We will refer to this commit as "merge commit".

Option 2 - changes D1 and D2 overlap (same file, but different lines of code)

Let's assume that developers edited the same file, but thankfully they edited different lines of code.

This means that we should be able to merge those changes into a single one automatically.

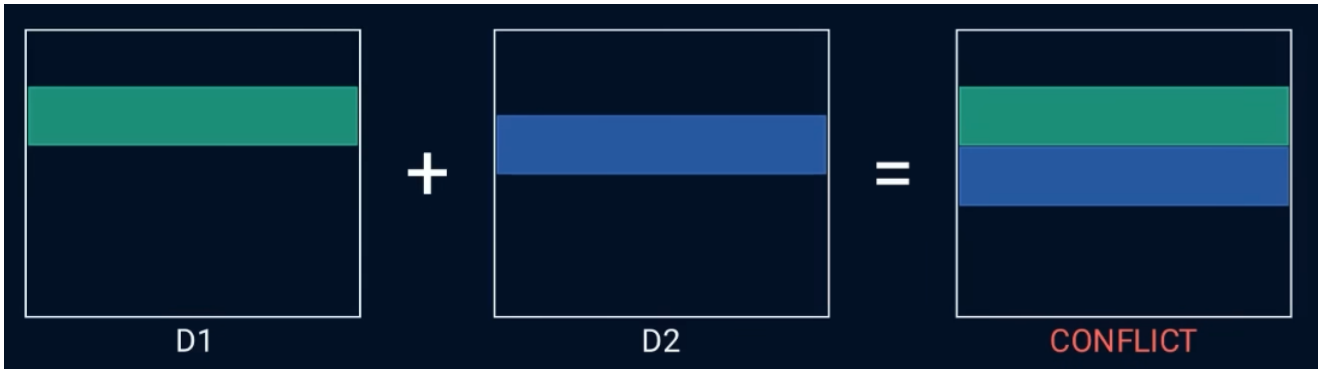
Just as before, git will add a new "merge commit" with the merged changes.

Option 3 - changes D1 and D2 overlap (same file, same lines of code)

Let's assume that developers edited the same file and same line of code. This means that we cannot automatically decide how to merge those changes. We do not know whose changes are more important, or maybe we should keep both the changes.

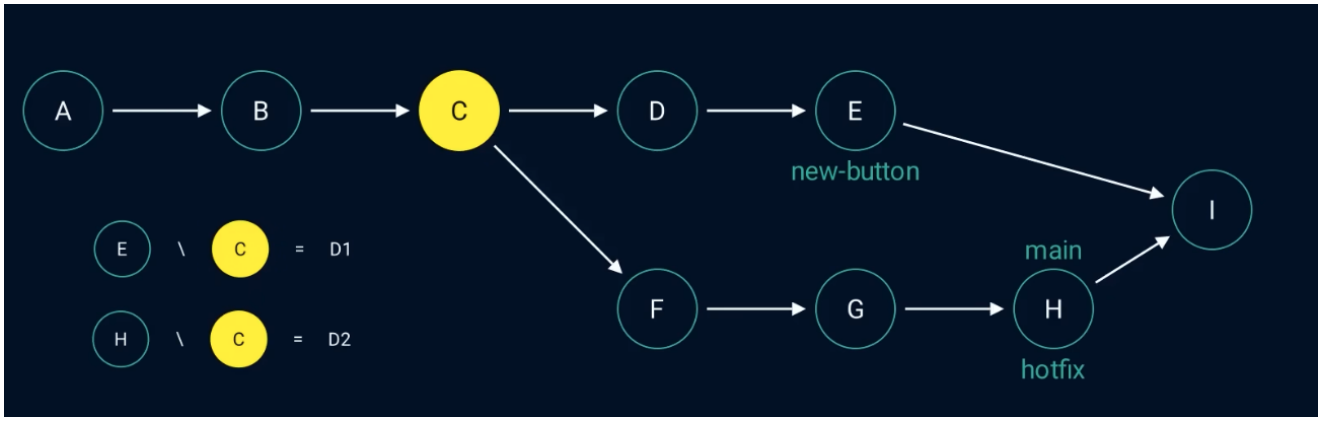
This kind of situation is referred to as **"conflict"**.

In this situation we can keep both changes in the file and add special markers showing us all the places with conflicts. Thanks to those markers we can manually resolve the conflict.



After resolving the conflict git will add a "merge commit".

Below is our history of commits after performing a merge.



In this example we can see the exact history of how the branches were created and how they were merged into other branches.

Although this shows the exact history - sometimes it can clutter the history. There are other ways of performing the merge like - rebase and squashing. Depending on the company you can be asked to use specific merging strategy.

Different strategies to work with Git

There are various ways of working with Git. We can say that basically say that every company has it's own way of working with Git. However we can say that are two well known approaches to work with Git.

Gitflow workflow

The GitFlow workflow is a branching model designed to enhance the management of larger projects by utilizing a structured approach to branching and releasing. This model, introduced by Vincent Driessen in 2010, outlines a strict branching model designed around the project release.

As you will see this way of working proved to be quite complex and may lead to a lot of conflicts. Because of this another way of working

was created - "trunk based development".
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

---

Trunk based development

Trunk-based development is a software development strategy where developers collaborate on code in a single branch called the "trunk", "mainline", or "master", depending on the version control system.

This approach minimizes the complexity of merging and integrating changes by encouraging frequent commits to the main branch and discouraging long-lived feature branches.
https://trunkbaseddevelopment.com/

Useful git commands

Configuration

```
git config –global user.name “Jan Kowalski”
git config –global user.email kowalski@gmail.com
git config –global core.editor “code –wait” //Ustawienie domyślnego IDE
git config –global core.autocrlf true (input – Mac, Linux) //Ustawienie auto-commit’owania LF
```

Initializing a repository

```
git init
```

Stage files

```
git add file1.js //Stages a single file1
git add file1.js file2.js //Stages multiple files
git add *.js //Stages all files with extension “js”
git add Dir1/* //Stages all files in directory “Dir1”
git add . //Stages all the modifications in current
directory
```

View the status of the repository

```
git status //Full status
git status -s //Short status
```

Committing the staged files

```
git commit -m “Message” //Commits with a one-line
message
git commit //Opens the default editor to type a long
message
```

Skipping the staging area

```
git commit -am “Message”2
```

Removing files

```
git rm file1.js //Removes file from working directory
and staging area (index)
git rm –cached file1.js //Removes from staging area only
```

Viewing the staged/unstaged changes

```
git diff //Shows unstaged changes
git diff –staged //Shows staged changes
git diff –cached //Same above
```

Viewing the history

```
git log //Full history
git log –oneline //Summary
git log –reverse //Lists the commits from the oldest to
the newest
```

Viewing a commit

```
git show 92a2ff //Shows the given commit
git show HEAD //Shows the last commit
git show HEAD~2 //Two steps before the last commit
git show HEAD:file.js
```

Unstaging files (undoing git add)

```
git restore –staged file.js
```

Discarding local changes

```
git restore file.js
git restore file1.js file2.js
git restore .
git clean –fd3
```

Restoring an earlier version of a file

```
git restore –source=head~2 file2.js
```

Viewing the history

```
git log –stat
git log –patch
```

Filtering the history

```
git log -3
git log –author=”Piotr”
git log –before=”2020-01-01”
git log –after=”one week ago”
git log –grep=”gui”
git log -s”gui”
git log hash1..hash2
git log file.txt
```

Viewing a commit

```
git show head~2
git show head~2:file1.txt
```

Comparing commits

```
git diff head~3 head
git diff head~3 head file.txt
```

Checkout out a commit*

```
git checkout dad47ed
git checkout master4
```

Finding a bad commit*

```
git bisect start
git bisect bad
git bisect good ca49180
git bisect reset
```

Finding contributors

```
git shortlog
```

Viewing the history of a file

```
git log file.txt
git log –stat file.txt
git log –path file.txt
```

Finding the author of lines

```
git blame file.txt
```

Tagging

```
git tag v1.0
git tag v1.0 5e7a828
git tag
git tag –d v1.0
```

Managing branches

```
git branch bugfix
git checkout bugfix
git switch bugfix
bug switch -c bugfix5
git branch -d bugfix
```

Comparing branches

```
git log master..bugfix
git diff master..bugfix
```

Stashing

```
git stash push -m “changed login”
git stash list
git stash show stash@{1}
git stash show 1
git stash apply 1
git stash drop 1
git stash clear
```

Git merging

```
git merge bugfix
git merge –no-ff bugfix
git merge –squash bugfix
git merge –abort
```

Viewing the merged branches

```
git branch –merged
```

```
git branch –no-merged
```

Rebasing

```
git rebase master6
```

Cloning a repository

```
git clone url
```

Syncing with remotes

```
git fetch origin master
git fetch origin
git fetch
git pull
git push origin master
git push
```

Sharing tags

```
git push origin v1.0
git push origin –delete v1.0
```

Sharing branches

```
git branch –r
git branch –vv
git push –u origin bugfix
git push –d origin bugfix
```

Managing remotes

```
git remote
git remote add upstream url
git remote rm upstream
```

Additional links

Here you can download Git

https://git-scm.com/downloads

Book about Git

https://git-scm.com/book/en/v2

Gitflow workflow

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Trunk based development

https://trunkbaseddevelopment.com/