2. Documentation and Version Control Systems

Miquel Àngel París i Peñaranda

Web Application Deployment

2nd C-VET Web Application Development

Index of contents

1. Goals.

1. Install and configure web servers on virtual machines and/or the cloud.
2. Perform functional tests on web and application servers.
3. Document the installation and configuration processes performed on web and application servers.

2. The Git Version Control System.

In this section, we introduce the distributed version control system Git, one of the most widely used systems today, utilized by most public software repositories like GitHub, GitLab, or Bitbucket, among others.

Git is a distributed version control system that allows us to collaborate on a project with several developers. This does not mean we cannot work locally. In this section, we will start using Git locally and later work in a distributed manner. First, we will look at the most important concepts related to Git and the typical workflow.

Git was developed to support the distributed development of the Linux kernel. Its main characteristics are:

- It is **fast** and **scalable**, adaptable to both individual projects and large projects with a high number of developers.
- It uses a **complete copy** with the full development history on each local user. If the main server becomes unavailable, it can be restored from any local copy. It also allows us to revert to a previous project state (what we call snapshots or commits) or view differences between two states without external tools.
- It facilitates **distributed development** by allowing simultaneous and independent development with a full repository copy, avoiding continuous synchronization.
- It allows **local work** without needing a connection to other networks since commits are made locally, and external connections occur when we want to collaborate with others or fetch the latest changes.
- It enables working with **branches** and easily merging them, allowing non-linear development.
- Instead of working with change lists, Git uses **snapshots** of the file system in each commit.
- It supports server connections using **multiple protocols**: HTTP, HTTPS, and SSH, along with its own protocol.

- It is a **robust** system that internally uses the SHA-1 encryption algorithm to detect accidental or malicious modifications in the repository.
- It is **free software**, following the philosophy of most collaborative projects, and is distributed under the GNU/GPL 2.0 license.

2.1. Concepts about Git

Here are some concepts used by Git, some of which may be familiar from other VCS systems:

- **Repository**: A container or database where the historical changes in the project's files are stored, saved through a commit.
- **Commit**: The action of saving a snapshot of the project in the repository, along with information about this change (usually the author, date, and description of the changes).
- **Git Areas**: A file can exist locally in three areas:
  - **Working Directory**: The visible area of the repository where we will usually work.
  - **Staging Area** (or Index): An intermediate area, not accessible from the terminal but through Git, where snapshots of the files to be committed are stored.
  - **Repository**: The actual storage of changes after making commits.

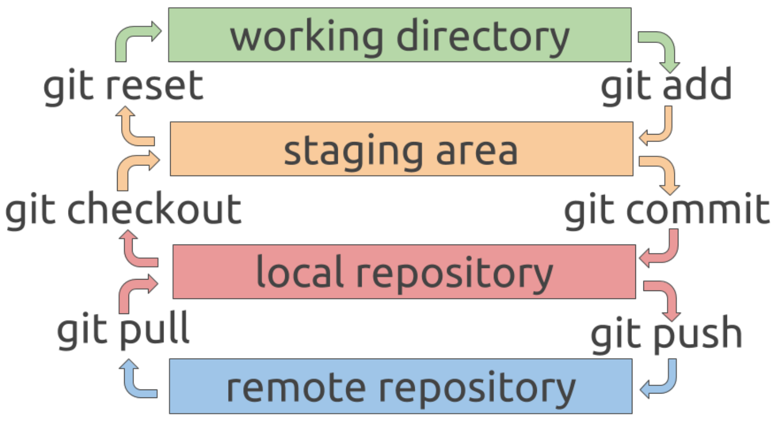


Fig. 3.4. Areas in Git

- **File States**: Files in the working directory can either be **untracked** or **tracked** by Git. The tracked files can exist in the following states:
    - **Committed** (or unmodified): The content of the file is the same in the working directory, staging area, and the repository.
    - **Modified**: The content of the file in the working directory is different from that in the staging area and repository. This means the file has been changed but not yet committed to the database.
    - **Staged**: The content of the file in the working directory and the staging area is the same, but it differs from the content in the repository. This happens when the file is marked to be included in the next commit.

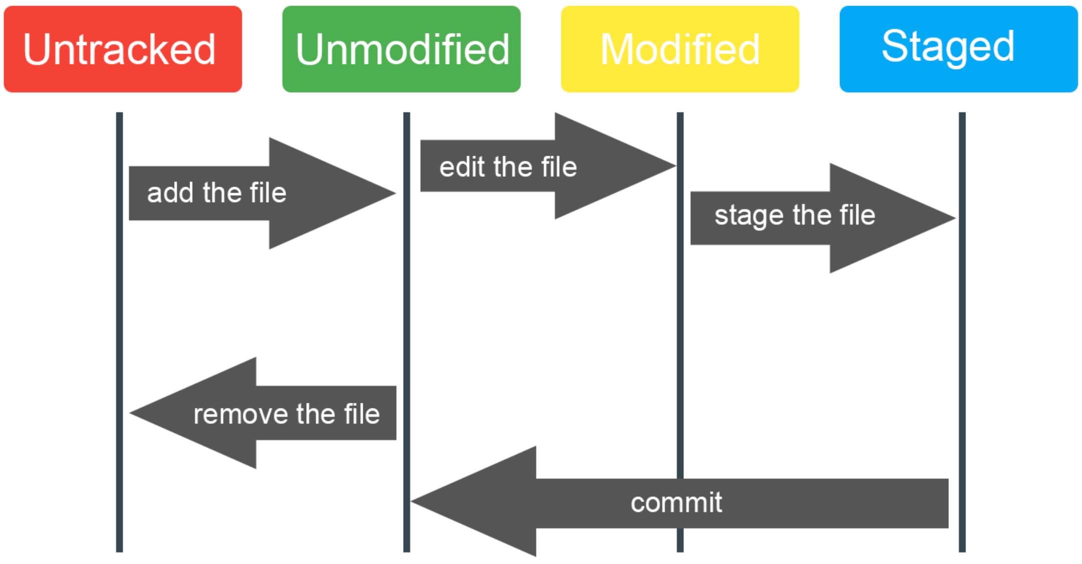


Fig. 3.5. File States in Git

- **SHA-1** and **HEAD**: The SHA-1 is a cryptographic function applied to a sequence of bytes of any size, such as an entire file or a set of files from a commit, resulting in a 40-character hexadecimal string (from 0 to 9 and A to F). Any modification to the file would produce a completely different SHA-1 **hash**. This is the mechanism Git uses to detect when a file has been modified or corrupted, ensuring the **integrity** of repository files. Each time a commit is made, a unique SHA-1 is calculated to identify it. Every commit, except the first, references the previous commit, stored in a **short SHA-1** format that uses at least the first four characters to uniquely identify the commit as long as there is no ambiguity.

HEAD: This is a reference pointing to the most recent commit.

- **Branches**: Branches are the alternative paths we take at a certain point to fix errors, add new features, or test new code that may or may not be viable (i.e., we are not sure if the new code will work, and we may need to undo everything we have done). Most of the time, we work on the main branch, known as master or main, from which we can create other branches and switch between them, changing the working directory accordingly.

2.2. Workflow

Once we have initialized a repository, the workflow consists of adding the files we want to the version control system, modifying them, and preparing successive commits to incorporate these changes into the database.

We can observe this in depth in the following diagram, noting the changes in the state of the files:

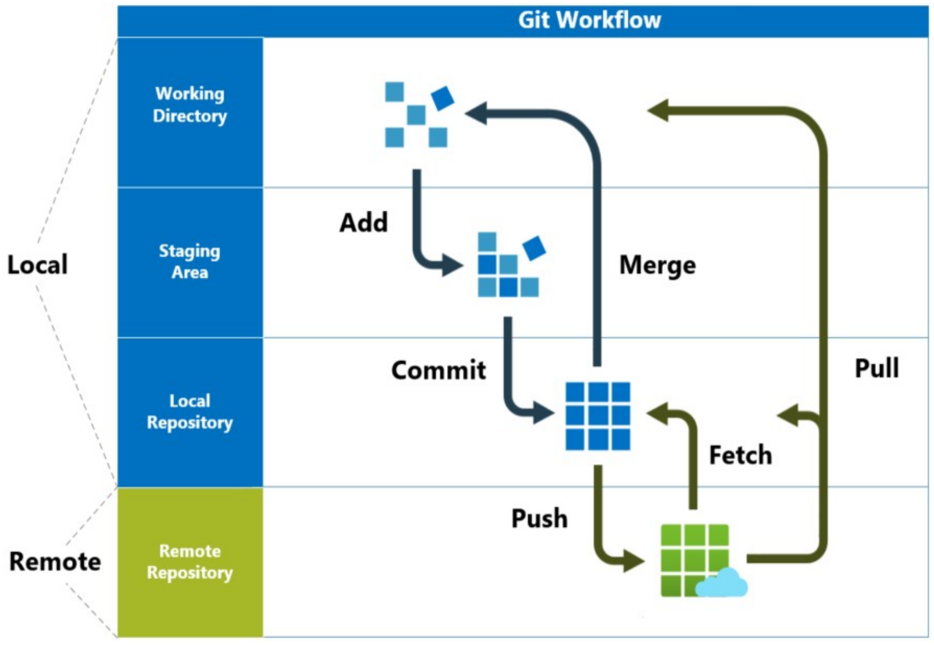


Fig. 3.6. Workflow.

1. We create the necessary files in the project folder. Initially, these files will be untracked by the version control system. We can create, edit, or delete them. Until they are added to version control, they will remain in this state.
2. We add the files to version control using the command `git add`. At this point, the files are tracked and also staged for the next commit. From this staged state, two things can happen:
   a) If we commit the changes, the modifications made for this commit will be added to the repository. The file will now be in the committed or unmodified state.
   b) Alternatively, we might modify the file we just created. In this case, the file moves to the modified state. To include these changes in the next commit, we need to stage the file again (using `git add`). If we don't add these changes, the unmodified version of the file will be added to the repository in the next commit, and we would need to run `git add` again to prepare the file.
3. We repeat the previous step, where we add new modifications and prepare new commits.

Throughout this workflow, each commit we make will generate a new SHA signature that points to the SHA signature of the previous commit. The HEAD will always point to the last valid commit.

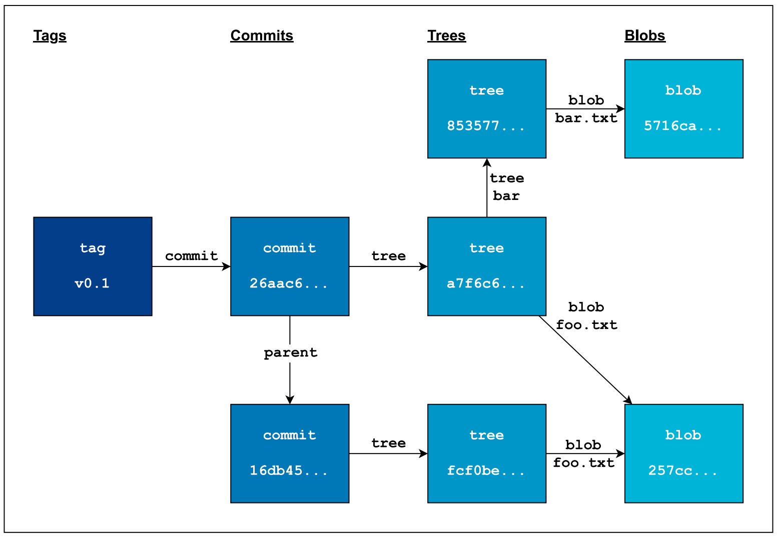


Fig. 3.7. Commits and SHA-1.

3. Remote Repositories with Git. GitHub.

3.1. Remote Repositories

So far, we've seen how to create and manage repositories with Git and how to work with different branches, all locally. When working in a distributed team, we need to share our local work. This is where Git's distributed architecture comes into play.

As we already know, Git is a distributed version control system (VCS). This means that the entire repository can be located on different computers, rather than being centralized on a single server. However, this doesn’t mean that a server isn’t required to maintain what would be the main repository.

Working directories and bare repositories

In previous practical cases, we used the `git init` command to initialize a directory as a working directory, allowing us to perform version control locally. In this working directory, Git generates a hidden `.git` directory at the root, where all the necessary information for local version control is stored.

When multiple developers need to share their work, they will do so through a **bare repository**, which could be translated as "uncovered," "empty," or "clean" repository, and it will be located on a centralized server. The team members will use a copy of this repository to work locally and will synchronize their work with the central repository.

This architecture could look like the following:

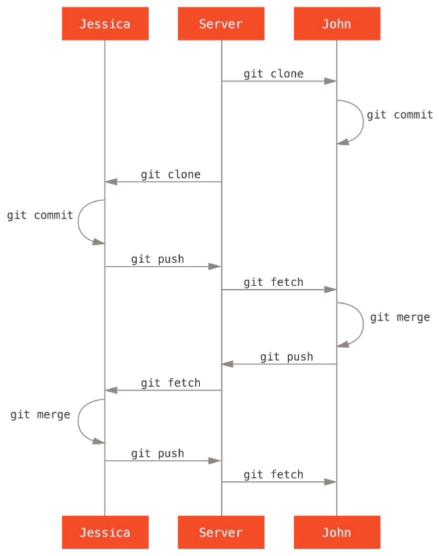


Fig. 3.9 Git Architecture.

This is still a distributed architecture since the repository is located on the different teams' machines in the form of a local repository. Additionally, there is a server that stores the bare repository. This repository will not have a working directory, so its content will have the same structure as the `.git` directory in local repositories.

The workflow in these cases will be as follows:

1. Create a bare repository on the server (`git init --bare`).
2. Clone the repository on each team's machine (`git clone repository_url`).
3. Work on the local copy of the repository in the working directory. Here, resources are added or removed (`git add/rm`), and changes are committed to the local repository (`git commit`), etc.
4. Synchronize between the local repository and the server. When sending data to the server, a **PUSH** operation is performed (`git push`), and when we want to fetch data from the server, **FETCH** (`git fetch`) or **PULL** (`git pull`) is used.

3.2. GitHub

When we want to create projects for team collaboration, or simply have a centralized repository on the Internet to manage version control for our projects using Git, the Internet offers us a wide range of possibilities.

Today, perhaps the most popular option is GitHub. However, since Microsoft acquired GitHub, and due to the concerns this acquisition raised in the community, other platforms like GitLab and Bitbucket have gained more traction in the market.

As stated on its homepage, GitHub has over 100 million developers, spread across more than three million organizations, and hosts over 413 million repositories.

GitHub defines itself as a collaborative development platform. In addition to hosting and providing version control through Git, it also offers various tools for project management and team collaboration, such as a wiki or documentation websites, among many other features.

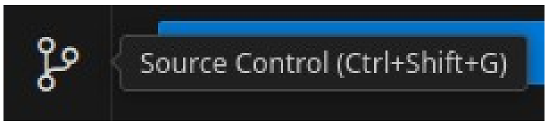
3.3. Git and Development Environments

So far, we have explored how to work with Git via the command line. Development environments also allow us to manage version control for our code, either through integrated tools or via extensions.

A. Visual Studio Code and Git

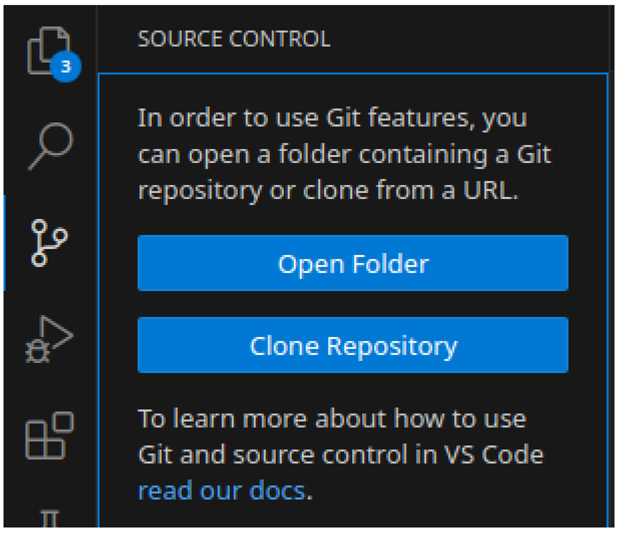
In addition to code editing, Visual Studio Code (VSCode) integrates many advanced features, including the Git version control system. The only requirement to use this feature in VSCode is having Git installed on your machine.

If you look at the activity bar on the left, there's a section called **Source Control** that allows you to interact with version control:



(Fig. 3.10: Accessing source control.)

If you open the **Source Control** section without any folder open, it will give you the option to either open a folder to manage version control or to clone a repository directly.

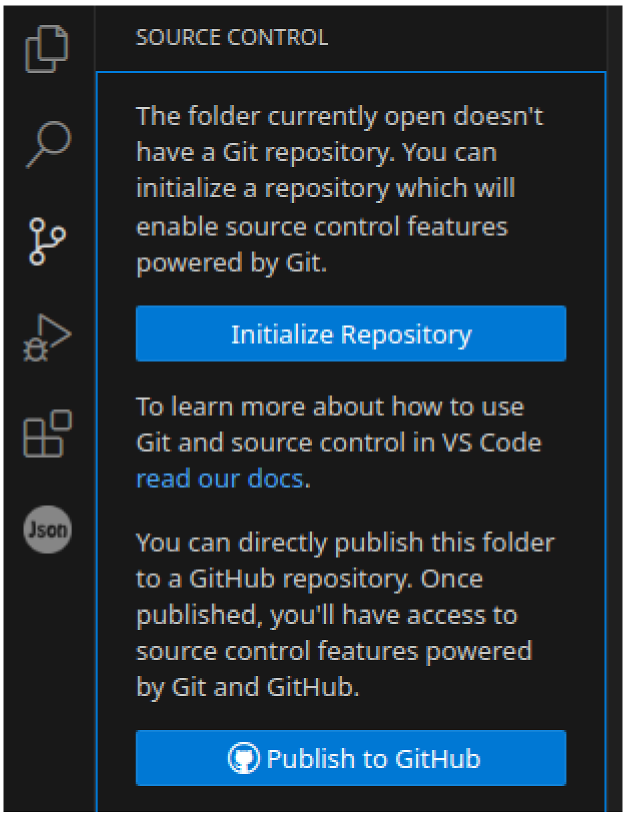


In order to use Git features, you can open a folder containing a Git repository or clone from a URL.

To learn more about how to use Git and source control in VS Code read our docs.

(Fig. 3.11: Source Control activity.)

Alternatively, if you open a folder that doesn't have version control but want to initialize it, you can do so from the integrated terminal or directly within VSCode by initializing the repository from this activity. At this point, it will also allow you to publish the folder's contents to GitHub.



The folder currently open doesn't have a Git repository. You can initialize a repository which will enable source control features powered by Git.

To learn more about how to use Git and source control in VS Code read our docs.

You can directly publish this folder to a GitHub repository. Once published, you'll have access to source control features powered by Git and GitHub.

(Fig. 3.12: Ways to initialize a folder with Git.)

Once you have a folder open with a repository, either cloned or initialized, this activity will provide information about the status of version control. For example, if you modify the `README.md` file and click on the refresh icon in the Source Control activity, you'll see that a change is detected in the repository. Additionally, a letter "M" (for modified) will appear next to the filename, indicating that the file has been changed. If you create a new file and follow the same process, it will appear in the Source Control with a "U" next to it, indicating that the file is untracked (not under version control yet).

Hovering over any of these files will show different options to open the file, revert the changes, or stage the changes (add). Additionally, from the title bar of this activity, there's an icon to commit changes.

To apply the changes, click the respective buttons to stage the changes (add) for both files, and you will see that they move to the staged (ready) area. Then, add a descriptive commit message and confirm the changes (commit).

Once this is done, all changes are confirmed in the working directory. If you are managing local version control, you’re finished. However, if you’re working on a cloned repository from GitHub, you need to perform a push. To do so, open the "Views and More Actions" menu in the title bar and look for the option "Pull," "Push," or "Synchronize."

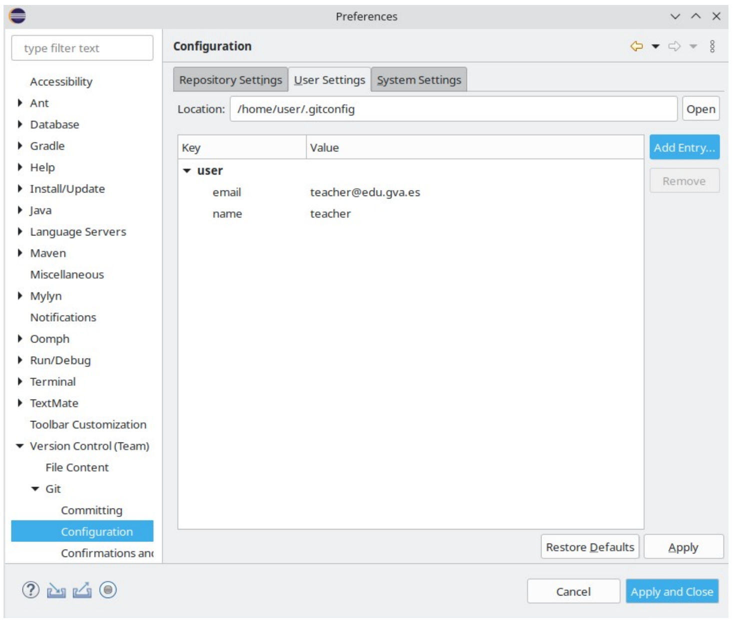
At this point, the environment will inform you that a pull will be performed from the repository, followed by a push, and will ask for confirmation. Additionally, since this is the first time synchronizing the repository, it will inform you that you are not logged into GitHub and will request permission to log in. The first time, you will be asked for your personal access token (PAT):

As you’ve seen, from the "More Actions" menu, you can access all Git functionalities supported by VSCode. Moreover, note that the status bar at the bottom not only provides status information but also allows you to perform certain actions like synchronizing or working with branches.

B. Eclipse and Git

The default installation of the Eclipse IDE includes Git integration. To use it, you need to have your **name** and **email** parameters configured. You can check this by going to the **Preferences** menu (Window > Preferences) and navigating to the Git configuration (Version Control (Team) > Git > Configuration).

If these parameters are already configured in your system, they will appear in this window; if not, you can add them using the **Add Entry** button:

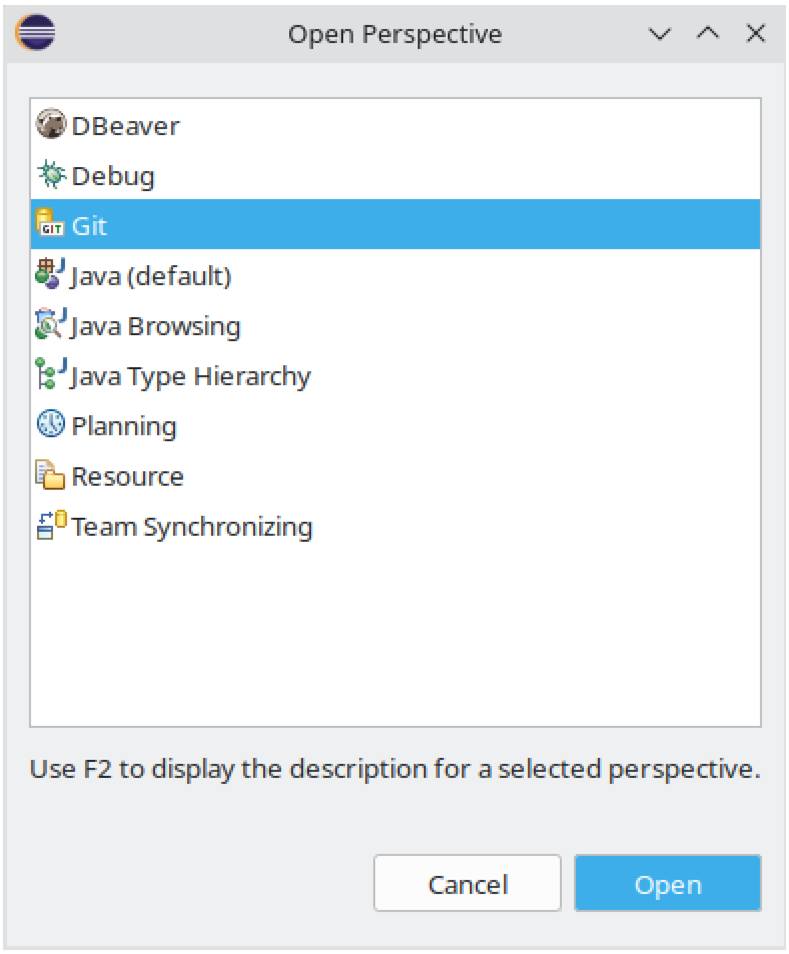


Location: /home/user/.gitconfig

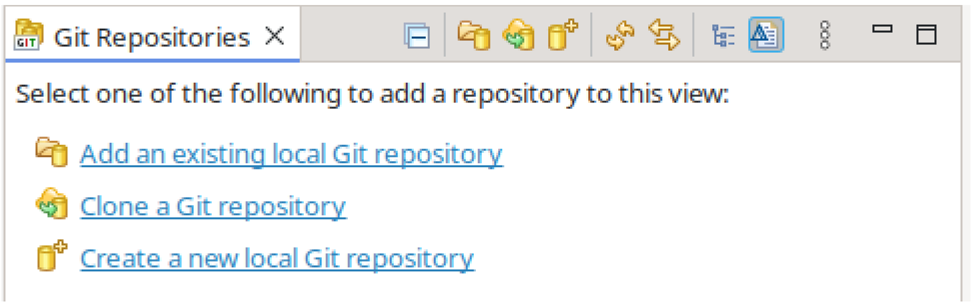
Key	Value
user	
email	teacher@edu.gva.es
name	teacher

(Fig. 3.17: Git configuration in Eclipse.)

To work with Git, you need to open the appropriate perspective (Window > Perspective > Open Perspective > Other, or the "Open Perspective" button on the right of the toolbar):



(Fig. 3.18: Opening the Git perspective.)


Git Repositories
Select one of the following to add a repository to this view:
Add an existing local Git repository
Clone a Git repository
Create a new local Git repository

(Fig. 3.19: Git perspective.)

From this perspective, you can perform graphical Git operations such as adding local repositories, cloning remote repositories, creating empty repositories, etc.

When working with a folder that already contains a repository, you can view its branches, working directory, and remote repositories, among other options.

Additionally, in the view on the lower-right corner, you can check and update the changes you're making in the repository.

4. Documentation

Creating effective documentation for a web application involves using the right tools to generate, manage, and store documentation, ensuring it's accessible and secure, and implementing systems for version control and continuous integration.

Effective documentation is a cornerstone of any successful web application. It not only aids in development and maintenance but also serves as a valuable resource for users, administrators, and support teams.

4.1. Documentation Generation Tools

Documentation generation tools create structured and formatted documents from comments or annotations within the code. Here are some popular tools:

- **JSDoc (JavaScript)**: Ideal for JavaScript, it creates documentation from comments in your code.
- **Swagger (OpenAPI)**: For APIs, it generates interactive documentation and helps visualize API endpoints.
- **Sphinx (Python)**: Primarily used with Python, it's flexible and suitable for creating entire project documents.
- **Doxygen (C++, C, PHP)**: Handles languages like C++, C and PHP, generating documents in multiple formats.
- **phpDocumentor(PHP)**: is the de-facto documentation application for PHP projects.

phpDocumentor is a documentation generator specifically designed for PHP applications. It allows developers to create structured, professional documentation based on comments within the code itself, following PHPDoc standard annotations.

1. Setting Up phpDocumentor

Installation: Ensure you have PHP installed. Then install phpDocumentor globally with Composer:

```
composer global require phpdocumentor/phpdocumentor
```

Basic Usage: To generate documentation, navigate to your project directory and run:

```
phpdoc
```

This generates HTML documentation from PHPDoc comments in your codebase.

2. Writing PHPDoc Comments

phpDocumentor generates documentation from specially formatted comments. Here’s a breakdown of key annotations:

- **@param** – Describes parameters of a function.
- **@return** – Describes what the function returns.
- **@throws** – Specifies exceptions the function might throw.
- **@var** – Defines the type of a variable.
- **@author**
- **@version**

Example Code with PHPDoc Annotations

Let’s start with a basic PHP class and add PHPDoc annotations.

```
<?php
/**
* Calculator class for basic arithmetic operations.
*
* This class provides methods to perform addition, subtraction,
* multiplication, and division.
*
* @package MyCalculatorApp
*/
class Calculator {
        /**
        * Adds two numbers.
        *
        * @param float $a The first number.
        * @param float $b The second number.
        * @return float The result of addition.
        */
        public function add(float $a, float $b): float {
            return $a + $b;
        }

        /**
        * Subtracts one number from another.
        *
        * @param float $a The first number.
        * @param float $b The second number.
        * @return float The result of subtraction.
        */
        public function subtract(float $a, float $b): float {
            return $a - $b;
        }
    }
```

3. Generating Documentation

Run phpDocumentor in your project directory:

```
phpdoc
```

This generates a `docs/` folder containing HTML files for each class, method, and attribute in your project.

4. Exercise: Document a More Complex Class

Objective: Write PHPDoc comments for a class named `UserManager` that performs user account operations.

Guidelines:

- Add @param and @return tags for methods.
- Include a brief description for the class and each method.

Example Class:

```
<?php
/**
* Manages user accounts.
*/
class UserManager {
    /**
```

```
        * Registers a new user.
        *
        * @param string $username The username.
        * @param string $password The password.
        * @return bool True if successful, false otherwise.
        */
        public function register(string $username, string $password): 
        bool {
                // Registration logic
        }

        /**
        * Logs a user in.
        *
        * @param string $username The username.
        * @param string $password The password.
        * @return bool True if login successful, false otherwise.
        */
        public function login(string $username, string $password): bool
        {
                // Login logic
        }
}
?>
```

Run phpDocumentor to check the generated documentation for completeness and accuracy.

5. Advanced phpDocumentor Configurations

- Custom Configuration File: You can create a `phpdoc.xml` file to customize options, like output directories, ignored files, and template selection.
- Example phpdoc.xml Configuration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpdoc>
        <files>
                <directory>src</directory>
        </files>
        <parser>
                <ignore>vendor/</ignore>
        </parser>
        <transformer>
                <target>docs/api</target>
```

```
    </transformer>
</phpdoc>
```

6. Exercise: Add Exception Documentation

Expand `UserManager` to throw custom exceptions and document them with `@throws`.

Add Exceptions:

```
/**
* Registers a new user.
*
* @param string $username The username.
* @param string $password The password.
* @return bool True if successful, false otherwise.
* @throws Exception if the username is taken.
*/
public function register(string $username, string $password): bool {
    if ($this->isUsernameTaken($username)) {
        throw new Exception("Username is already taken.");
    }
    // Registration logic
}
```

Generate Documentation and confirm the `@throws` tags display under each method.

7. Explore Different Templates

phpDocumentor comes with a few built-in templates for styling. Try generating documentation with the `clean` template:

```
phpdoc -t clean
```

8. Additional Resources

- phpDocumentor Official Documentation: phpDocumentor Manual
- PHPDoc Standard: PHPDoc on phpDocumentor

4.2. Platform-Specific Generators for Software Documents

Many platforms offer tools that align with their ecosystem, providing enhanced features and language support:

- **Java (JavaDoc)**: JavaDoc generates documentation directly from source code comments in Java projects.
- **.NET (Sandcastle)**: Sandcastle produces documentation for .NET projects, compatible with Visual Studio.
- **Android (Dokka)**: Used for Kotlin and Android, Dokka generates structured documentation compatible with Android Studio.
- **JSDoc**: A tool for generating documentation for JavaScript code.

JSDoc is an excellent documentation tool for JavaScript, allowing you to generate structured, readable documentation from your code. It uses special comments to describe classes, functions, parameters, and other aspects of JavaScript code.

1. Setting Up JSDoc

Installation

If you don’t have JSDoc installed, start by installing it globally with npm:

```
$ npm install -g jsdoc
```

You can also add it to your project’s dependencies:

```
npm install jsdoc --save-dev
```

Basic Setup

To generate documentation, run:

```
jsdoc path/to/your/file.js
```

This will create a `out/` directory with HTML documentation files.

2. Documenting a Function

Let’s start with a simple function and learn how to document it using JSDoc.

Example

```
/**
* Adds two numbers together.
* @param {number} a - The first number.
* @param {number} b - The second number.
* @returns {number} The sum of the two numbers.
*/
function add(a, b) {
        return a + b;
}
```

Explanation

- `@param` tags describe the parameters (`a` and `b`).
- `@returns` tag describes the return type and value.

3. Working with Object and Array Types

JSDoc supports more complex data types, such as objects and arrays, which can be documented with specific notation.

Example

```
/**
* Get user information.
* @param {Object} user - The user object.
* @param {string} user.name - The user's name.
* @param {number} user.age - The user's age.
* @param {string[]} user.hobbies - An array of hobbies.
* @returns {string} A formatted string with user info.
*/
function getUserInfo(user) {
        return `${user.name} is ${user.age} years old and enjoys $
        {user.hobbies.join(", ")}.`;
}
```

Explanation

- `{Object}` type for `user` object.
- `{string[]}` denotes an array of strings for `user.hobbies`.

4. Documenting Classes and Methods

JSDoc is very useful for documenting classes and their methods.

Example

```
/**
```

```
* Represents a Car.
* @class
*/
class Car {
      /**
      * Create a car.
      * @param {string} make - The make of the car.
      * @param {string} model - The model of the car.
      */
      constructor(make, model) {
            this.make = make;
            this.model = model;
      }

      /**
      * Get the car's details.
      * @returns {string} The car's make and model.
      */
      getDetails() {
            return `${this.make} ${this.model}`;
      }
}
```

Explanation

- `@class` tag documents the `Car` class.
- Constructor and methods are documented within the class.

5. Using Custom Tags

You can create custom tags in JSDoc to add extra information, such as `@example` for code examples.

Example

```
/**
* Calculates the area of a rectangle.
* @param {number} width - The width of the rectangle.
* @param {number} height - The height of the rectangle.
* @returns {number} The area of the rectangle.
* @example
* const area = calculateArea(5, 10);
* console.log(area); // 50
*/
function calculateArea(width, height) {
        return width * height;
    }
```

Explanation

- `@example` tag provides a code snippet example of how to use the function.

6. Generating Documentation HTML

After adding JSDoc comments, you can generate HTML documentation for a more readable format.

Example

In the project root, run:

```
jsdoc path/to/your/file.js
```

This will generate an `out` folder with HTML documentation. Open `out/index.html` to view it in a browser.

Customizing Output

You can specify a configuration file for JSDoc to customize output location, themes, and other options.

4.3. Various Documentation Formats

Documentation can be stored in different formats based on needs:

- **HTML**: Interactive and easy to navigate, HTML works well for online documents.
- **PDF**: Good for sharing and offline use, PDF is often created for formal documentation.
- **Markdown (MD)**: Simple to write and read, Markdown is widely used for README files on GitHub.
- **YAML/JSON (OpenAPI, Swagger)**: Used in API documentation, these formats allow configuration of API parameters and requests.

4.4. Collaborative Tools for Creation and Maintenance of Documentation

Collaborative tools make it easier for teams to contribute to and maintain documentation:

- **Confluence**: Used to create, share, and collaborate on documents in real time.
- **Notion**: Allows teams to collaborate on project documentation, with support for images, links, and tables.
- **Google Docs**: A free and versatile option for collaborative writing and version history.
- **GitHub Wiki**: Used alongside GitHub repositories, it enables teams to write and update documentation collaboratively.

4.5. Ensuring Accessibility and Security in Version Control Systems (VCS)

Version Control Systems (VCS) like Git ensure secure storage of your documentation, code, and assets. They provide access control, track changes, and allow rollback if needed. Some key considerations:

- **Branching**: Use branches (e.g., “main” for stable docs, “dev” for edits) to manage documentation changes.
- **Permissions**: Assign appropriate read/write access to protect sensitive sections.
- **Encryption**: Encrypt repositories and maintain secure access credentials.
- **Backups**: Regularly backup documentation to prevent loss.

4.6. Tools for Continuous Code Integration

Continuous Integration (CI) automates tests, builds, and deployment. By adding documentation generation to CI, you can ensure the latest documentation is published with each update:

- **GitHub Actions**: Automates tasks in GitHub, like building, testing, and deploying documentation.
- **Jenkins**: A powerful CI tool with plugins for documentation generation.
- **CircleCI**: CI/CD with Docker support, often used to test and deploy web apps.
- **Travis C**I: A cloud-based CI service that works with GitHub and other platforms.

CI Pipeline for Documentation:

1. Trigger: A change is pushed to the documentation repository.
2. Build: The documentation is built using the chosen tool.
3. Test: The documentation is tested for errors and accessibility issues.
4. Deploy: The documentation is deployed to a web server or static site hosting service.

Web Links

In this section, you will find the relevant links of interest necessary to expand and explore the contents of the unit.

-