SOFTWARE ENGINEERING

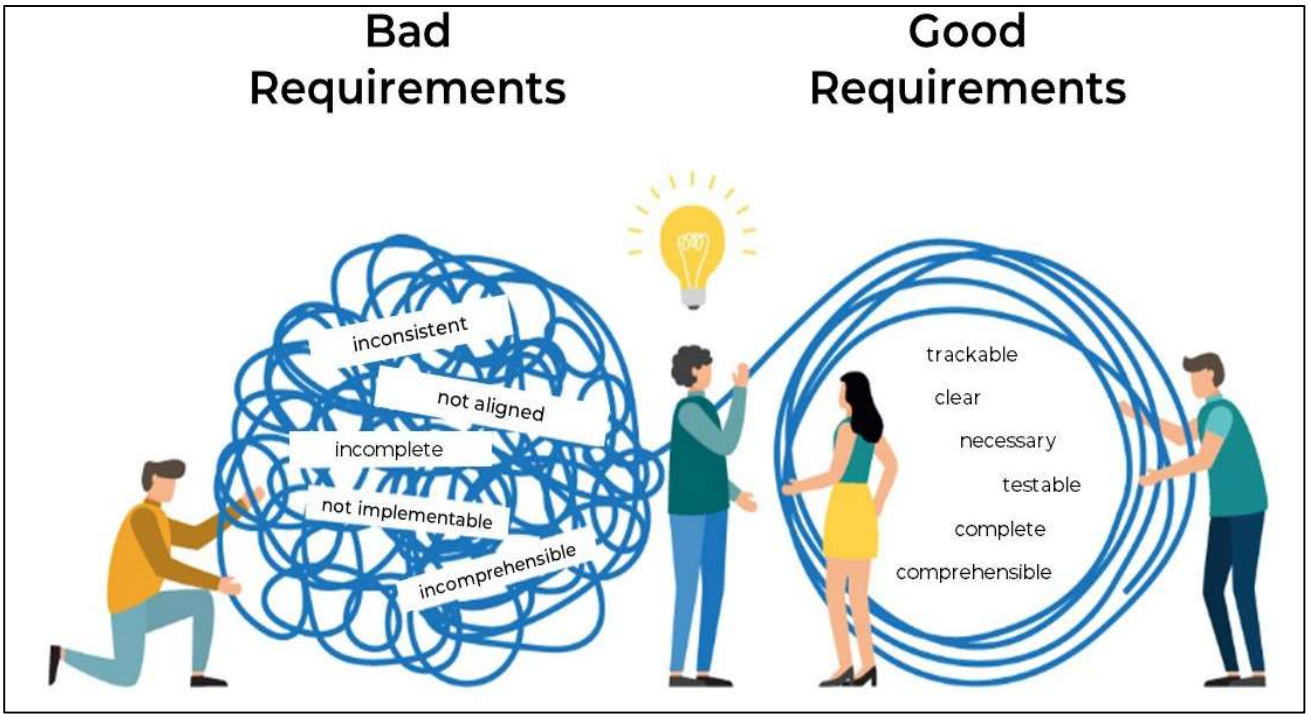
UNIT – 2

TOPIC – 1

UNDERSTANDING REQUIREMENTS

Software requirements describe what a software system should do and how it should perform. These requirements guide developers in building the system the way users expect. Requirements are usually divided into two main types:

1. Functional requirements and
2. Non-functional requirements.



1. Functional Requirements

Functional requirements describe the specific actions the system must perform. They outline the functions or services that the software provides to users. Essentially, they define what the system should do.

Examples of Functional Requirements:

i. A system must allow users to search for items in a database.
ii. Every order made in the system should generate a unique order ID.
iii. The system should let users view documents.

Why Functional Requirements Are Important:

These requirements make sure the software does the tasks it's supposed to do. They cover what happens when a user gives input (like clicking a button) and what the system should output (like showing search results).

Key Qualities of Good Functional Requirements:

i. **Complete:** All required services should be included.
ii. **Consistent:** There should be no contradictions between different requirements.

2. Non-Functional Requirements

Non-functional requirements describe how well the system should perform, not what it should do. These are qualities or attributes that the system should have, such as how fast it should be, how secure, or how reliable.

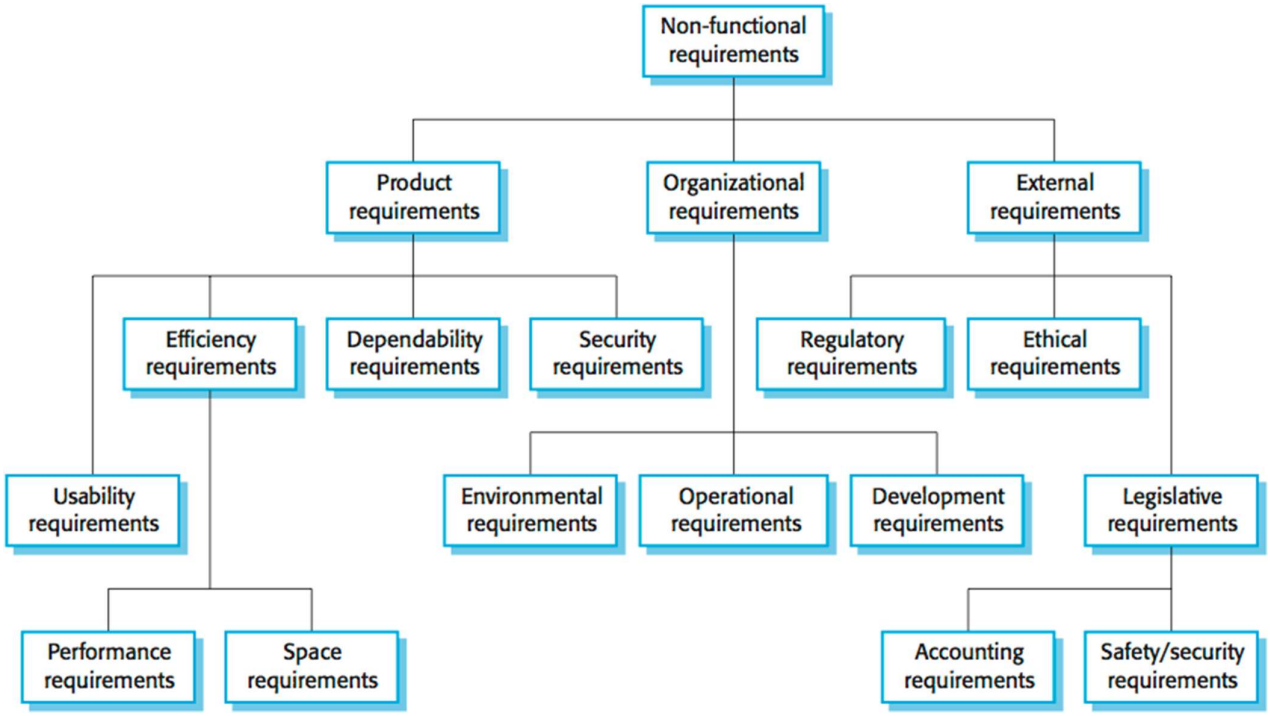
Examples of Non-Functional Requirements:

i. The system must respond to user actions within 2 seconds.
ii. The system should be able to handle up to 1,000 users at the same time.
iii. The system should be available 99% of the time (high reliability).

Why Non-Functional Requirements Are Important:

These requirements ensure that the software meets user expectations beyond just functionality. If the system is too slow or unreliable, it won't satisfy users, even if it performs the right tasks.

Non – Functional Requirements Types:



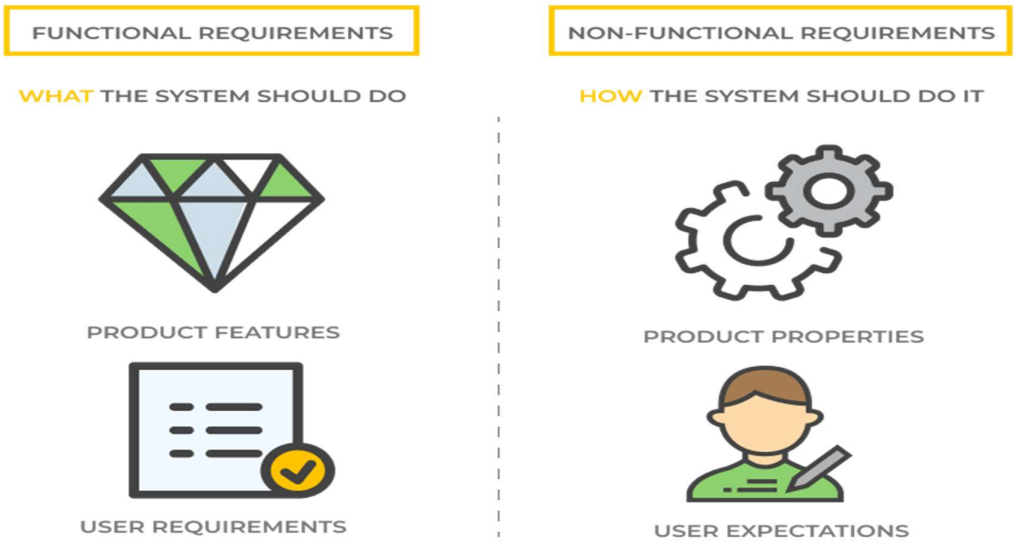
a. **Product Requirements:** These are all about how the software should perform and behave. They focus on what you want the software to do in terms of speed, efficiency, and other technical details. Think of them as the "performance expectations" for your software.
   Speed: How fast should it be? For example, it should find your files in under 2 seconds.
   Memory Usage: How much memory should it use? It shouldn't use more than 500MB of your computer's memory.
   Example: The software must use less than 500MB of memory. This means it should not take up more than 500MB of your computer's memory while running.
b. **Organizational Requirements**: These are related to the company's own rules or policies. They dictate the choices you make when developing software, based on what the company prefers or requires.
   Programming Language: The company might require using a specific language to build the software. For example, they might want it built with Python.
   Example: The software must be built using Python. This means the development team has to use Python for creating the software.

c. **External Requirements:** These come from outside the software itself and usually involve external factors that the software must comply with or interact with.

Laws and Regulations: The software needs to comply with legal rules, like protecting user privacy.

Other Systems: The software might need to work with other systems or services.

Example: The system must follow data privacy laws. This means it has to handle personal information in a way that meets legal privacy standards.



Difference between the functional and non-functional requirements.

Aspect	Functional Requirement	Non-Functional Requirement
What it Describes	Actions or tasks the system should perform	Qualities of the system (speed, security)
Requirement Type	Mandatory	Often not mandatory, but still important
Captured in	Use cases (scenarios of how users interact)	Quality attributes (e.g., performance metrics)
Focus	What the system does	How well the system performs
Testing	Functional testing (e.g., feature testing)	Non-functional testing (e.g., performance tests)

Aspect	Functional Requirement	Non-Functional Requirement
Example	"The system must allow users to place orders."	"The system must handle 1,000 orders per second."

User Requirements

User Requirements are about what users want from the software, written in plain language that's easy to understand. They focus on what the software should do from the user's point of view.

- **Functional User Requirements:** These describe what the software should do. For example, if a user wants to easily manage customer orders, that's what the software needs to be able to do.

Example: "I want the software to allow me to manage my customer orders easily." This means the software should help users handle their orders without any trouble.

- **Non-Functional User Requirements:** These describe how the software should perform or feel. It's about the user's experience, like how fast or easy it is to use.

Example: "The system should be fast and easy to use." This means users want the software to respond quickly and be simple to navigate.

User Requirements Are Gathered by:

- **Interviews:** Talking directly with users to find out what they need.
- **Surveys:** Sending questions to users to get their opinions and needs.
- **Workshops:** Bringing users and developers together to discuss what the software should do.

System Requirements

System Requirements are technical details about how the software should work. They translate user needs into specific instructions for developers to follow when building the software.

- **Functional System Requirements:** These describe what the system should do in technical terms. For example, how the software should handle customer orders.

Example: "The system shall store customer orders in a database and generate unique order IDs." This means the software should save orders in a database and create a unique ID for each one.

- **Non-Functional System Requirements:** These describe how the system should perform in technical terms, like how many users it can handle at once or how quickly it should respond.

Example: "The system shall support up to 500 users at the same time without slowing down." This means the software should be able to handle 500 users using it at the same time without becoming slow.

Purpose of System Requirements:

- **Guide Development:** They tell developers exactly what to build and how to build it.
- **Testing:** They help check if the finished software meets the user's needs.

Difference between the user and system requirements.

Aspect	User Requirements	System Requirements
Purpose	Capture what users want from the system	Define how the system will technically fulfill user needs
Language	Simple, non-technical	Technical, for developers and engineers
Who Uses Them	Users, stakeholders, business analysts	Developers, testers, architects
Detail Level	High-level, broad goals	Detailed, specific instructions
Examples	"The system should allow me to view my orders."	"The system shall retrieve order data from the database and display it."

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 2

SOFTWARE REQUIREMENTS DOCUMENT

Note: This document is the same one previously uploaded for the IEEE SRS document under Lab Experiment 1.b in Unit 1, but it now includes the additional section on the Characteristics of a Good SRS Document.

1. Introduction to SRS (Software Requirements Specification)

- **What is an SRS Document?**
  - **SRS** stands for **Software Requirements Specification**. It's a detailed document that outlines exactly what a software product should do.
  - **Purpose:**
    Think of it as a blueprint for software development, similar to how an architect's plan guides the construction of a building.
  - **Example:**
    If you were building a house, the blueprint would show the size and layout of each room, where the doors and windows go, and what materials to use. Similarly, an SRS document lays out every detail of how the software should function, what features it should have, and how it should look.

2. Why is an SRS Document Needed?

- **Importance of an SRS Document:**

1. **Clarity and Understanding:** It ensures that everyone involved in the project—developers, designers, and stakeholders—understands what the software is supposed to do.
2. **Guides Development:** The SRS document serves as a guide for the development team, helping them build the software correctly.
3. **Communication Tool:** It acts as a communication bridge between different teams and stakeholders, making sure everyone is on the same page.
4. **Manages Changes:** If there are changes needed during development, the SRS document helps manage those changes without confusion.

5. **Helps in Testing:** The SRS document also helps testers by providing clear criteria on what the software should do, so they can check if it meets all the requirements.

3. Good SRS Document

A Software Requirements Specification (SRS) document is a crucial part of software development. It outlines what the software should do, helping both developers and stakeholders understand the project requirements clearly. A well-written SRS document serves as a foundation for designing, coding, testing, and maintaining software.

Characteristics of a Good SRS Document

i. **Correctness:** The document should accurately describe all the system's requirements. Every feature that the software is supposed to have must be included, and any incorrect or missing information should be avoided.

ii. **Unambiguous:** The language used in the SRS should be clear and straightforward. There should be no confusion or multiple interpretations of any requirement. This helps ensure that everyone, from developers to stakeholders, has the same understanding of what needs to be built.

iii. **Complete:** A good SRS covers all aspects of the software. It should include functional requirements (what the system does) and non-functional requirements (like performance, security, usability, etc.), leaving nothing out.

iv. **Consistent:** The SRS must be free of contradictions. For instance, one section should not specify a requirement that another section contradicts. Consistency across all parts of the document ensures smooth development.

v. **Ranked for importance and stability:** Each requirement should be prioritized based on how critical it is to the project. Some requirements may also change over time, so it's essential to identify which parts are more likely to remain stable.

vi. **Verifiable:** Every requirement in the SRS should be measurable and testable. For example, if the document states that the system should respond in under two seconds, this can be tested and verified.

vii. **Modifiable:** A good SRS document is flexible and can be updated when needed. It should be well-organized so that changes can be made without affecting other parts of the document.

viii. **Traceable:** Each requirement should be traceable back to its source, whether that's a customer need or a regulatory requirement. This helps in ensuring that every feature has a valid reason for being in the project.

A good SRS is one that is accurate, clear, complete, consistent, well-organized, and easy to update. This ensures a smooth development process and helps avoid misunderstandings later on.

4. What is IEEE - SRS?

- **Definition:**
  - **IEEE SRS** refers to the **Software Requirements Specification** as defined by the **Institute of Electrical and Electronics Engineers (IEEE)**.
  - **Purpose:**
    The IEEE SRS is a standardized format for writing an SRS document, ensuring that all important aspects of the software are covered consistently.
  - **Example:**
    Just like a recipe book has a standard format for listing ingredients and steps, the IEEE SRS template ensures that all software projects follow a standard way of documenting requirements. This makes it easier for everyone to understand and follow the document.

Software Requirements Specification (SRS) Template

1. **Introduction**
   - **Purpose**: Describes the product that the SRS covers, including the specific version or part of the system.

     Example: "This document outlines the requirements for Version 2.0 of our customer management software."

   - **Document Conventions**: Lists any standards or formatting used in the document, such as fonts, colours, or symbols that indicate priorities.

     Example: "Critical requirements are highlighted in bold and marked with an asterisk (*)."

   - **Intended Audience**: Identifies who should read the SRS, such as developers, managers, or testers, and suggests reading sequences.

Example: "This document is intended for developers, testers, and project managers to ensure a shared understanding of system requirements."

- **Product Scope**: Provides a brief description of the software, its purpose, and how it aligns with business goals.

  Example: "The software aims to automate inventory management, enhancing operational efficiency."

- **References**: Lists other documents or resources related to the SRS.

  Example: "Refer to the User Interface Style Guide, Version 1.1, for design standards."

2. **Overall Description**

- **Product Perspective**: Explains the software's background and its relationship to other systems or components.

  Example: "This tool is an upgrade to the existing analytics module, designed to integrate with our sales platform."

- **Product Functions**: Summarizes the main features the product will perform, organized in a simple list.

  Example: "Key functions include data import, user authentication, and report generation."

- **User Classes and Characteristics**: Defines different user types and their key characteristics, such as skill level or security access.

  Example: "User classes include Admins, who have full access, and Regular Users, who have limited access."

- **Operating Environment**: Details the hardware and software environments where the software will run.

  Example: "The system will operate on Linux servers with Java 11 and MySQL databases."

- **Design and Implementation Constraints**: Outlines any limitations or specific requirements that affect design choices.

Example: "The system must comply with GDPR regulations and use Python for backend development."

- **User Documentation**: Lists the documentation that will be provided to users, such as manuals or online help guides.

  Example: "A user manual and an interactive help guide will be included."

- **Assumptions and Dependencies**: Identifies assumptions that could impact the project, such as reliance on third-party components.

  Example: "The system assumes the availability of an existing LDAP server for user authentication."

3. **External Interface Requirements**

- **User Interfaces**: Describes how the software will interact with users, including screen layouts and navigation standards.

  Example: "Each screen will have a consistent layout with navigation buttons at the top."

- **Hardware Interfaces**: Details how the software will connect with hardware, such as printers or sensors.

  Example: "The software will communicate with RFID scanners via Bluetooth."

- **Software Interfaces**: Defines connections between the product and other software components, including databases and APIs.

  Example: "The software will interact with REST APIs to fetch real-time data."

- **Communications Interfaces**: Specifies requirements for any communication-related functions, like protocols or security needs.

  Example: "The system will use SSL/TLS for secure data transmission over the network."

4. **System Features**

- **Feature Description and Priority**: Provides brief descriptions of each feature and its importance level (high, medium, low).

Example: "User authentication is a high-priority feature to ensure secure access."

- **Stimulus/Response Sequences**: Lists the expected user actions and corresponding system responses.

  Example: "When the user submits a form, the system will validate the data and display a success message."

- **Functional Requirements**: Details the specific tasks the software must perform for each feature.

  Example: "REQ-1: The system must allow users to reset their passwords via email verification."

5. **Other Nonfunctional Requirements**

- **Performance Requirements**: Specifies how the software should perform under different conditions, such as speed and scalability.

  Example: "The application must process up to 500 transactions per second."

- **Safety Requirements**: Defines measures to prevent harm or data loss.

  Example: "The software must include auto-save functionality to protect user data during unexpected shutdowns."

- **Security Requirements**: Lists the security protocols that must be followed, such as data encryption or user authentication.

  Example: "User data must be encrypted using AES-256 encryption standards."

- **Software Quality Attributes**: Describes attributes like usability, reliability, and maintainability.

  Example: "The system should have a 99.9% uptime to ensure high availability."

- **Business Rules**: Outlines operational rules that influence software behaviour.

  Example: "Only users with the admin role can approve transactions over $5,000."

6. **Annexures**

- **Appendix A: Glossary**: Defines terms, abbreviations, and acronyms used throughout the SRS.

Example: "API: Application Programming Interface."

- **Appendix B: Analysis Models**: Includes relevant analysis diagrams, such as data flow diagrams or class diagrams.

Example: "See Figure B.1 for the system's state-transition diagram."

- **Appendix C: To Be Determined List**: Tracks items that are still pending decisions or additional information.

Example: "TBD-1: Decide on the final database technology."

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 3

REQUIREMENTS ENGINEERING AND FEASIBILITY STUDY

1. Requirements Engineering (RE)

Definition:

Requirements Engineering (RE) is a structured process that involves understanding what a software system must do to meet the needs of its users and stakeholders. It involves gathering, analyzing, documenting, validating, and managing the requirements that specify what a system should do.

Key Goals of Requirements Engineering:

- **Define System Requirements:** Clearly describe what the software must accomplish to solve a specific problem or fulfil user needs.
- **Minimize Errors:** Reduce the chance of building a product that doesn't meet user expectations.
- **Guide Development:** Provide a clear blueprint that developers and project managers can follow throughout the software development lifecycle.

Why is Requirements Engineering Important?

- **Improves Communication:** It serves as a communication bridge between stakeholders and developers, making sure everyone understands the project goals.
- **Cost and Time Efficiency:** Catching errors or misunderstandings in the requirements phase is much cheaper and quicker than fixing them during later stages of development.
- **Reduces Risks:** Identifying issues early reduces project risks, such as time overruns, budget issues, and the development of features that are not needed.

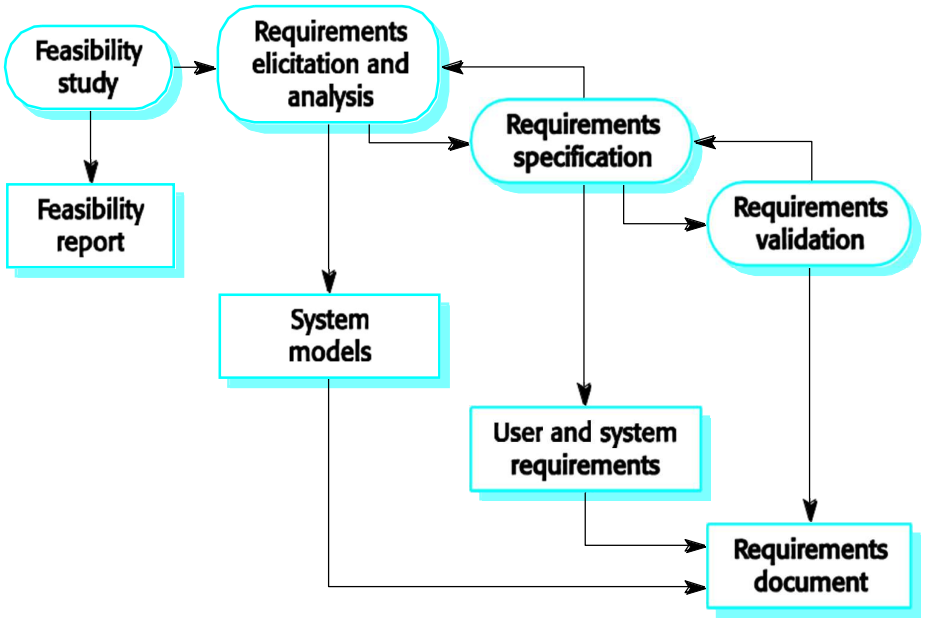
Common Problems if Requirements Engineering is Neglected:

- **Misunderstandings:** Without clear requirements, developers may misunderstand what needs to be built.

- **Scope Creep:** The project may grow out of control because requirements were not clearly defined from the beginning.
- **Project Failure:** Many software projects fail due to incomplete, unclear, or misunderstood requirements.

2. Steps in the Requirements Engineering Process

The Requirements Engineering process is divided into several steps, each contributing to the development of a clear set of software requirements:



1. **Feasibility Study:** A preliminary analysis that determines if the project is viable. Focuses on whether the project can be developed with available resources, technology, and time constraints.
2. **Requirements Elicitation:** Involves collecting the needs of stakeholders, including customers, end-users, and other parties. Techniques used for elicitation include interviews, questionnaires, workshops, and observations.

Example: For a hospital management system, requirements can be gathered from doctors, nurses, administrative staff, and patients to understand their specific needs.

3. **Requirements Analysis:** In this phase, the gathered requirements are analyzed to ensure they are clear, complete, consistent, and feasible. Conflicts between requirements are identified and resolved to ensure that they do not contradict each other.

Example: If a shopping app requires both "fast delivery" and "low shipping cost," analysis is needed to find a balance between these conflicting needs.

4. **Requirements Documentation:** All requirements are documented in a structured format, usually in a Software Requirements Specification (SRS) document. The SRS includes detailed descriptions of both functional requirements (what the system should do) and non-functional requirements (performance, security, usability).

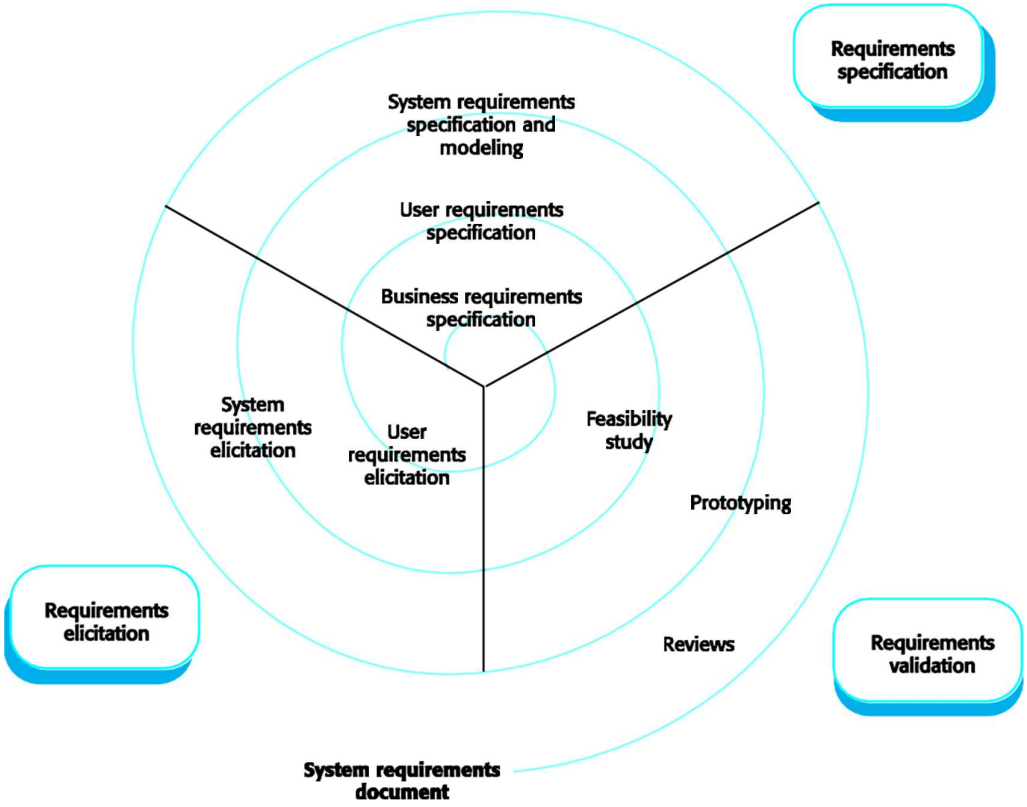
Example: The SRS for a banking software might include functional requirements like "the system shall support account transfers" and non-functional requirements like "transactions should process within two seconds."

5. **Requirements Validation:** Validation ensures that the documented requirements align with what stakeholders actually expect from the system. It involves reviewing the requirements with stakeholders to confirm that they accurately represent their needs.

Example: In a meeting, stakeholders might review and validate that the software's interface design matches their user experience expectations.

6. **Requirements Management:** This step involves handling changes to the requirements as the project evolves. Requirements management ensures that any updates to requirements are systematically controlled and communicated to all team members.

Example: If a new regulation requires added security features, requirements management tracks and integrates these changes into the project plan.



3. Feasibility Study

What is a Feasibility Study?

A feasibility study is a critical assessment conducted to determine if a proposed project or system is practical and achievable. It evaluates various factors like technology, budget, timeline, and overall value to decide whether the project should proceed.

Importance of a Feasibility Study:

- **Risk Mitigation:** Identifies potential risks in technology, costs, and operations before they become problems.
- **Efficient Resource Use:** Ensures that the company's time, money, and effort are invested in projects that are likely to succeed.
- **Informed Decision-Making:** Helps stakeholders decide whether to approve, reject, or modify the project based on concrete data.

4. Types of Feasibility Studies

Feasibility studies can be categorized into several types, each focusing on different aspects of a project's potential success:

1. **Technical Feasibility:** Evaluates whether the technology and tools required for the project are available and suitable.

Considerations: Does the development team have the technical expertise? Is the technology stable and scalable?

Example: Before developing an AI-based customer support system, the company assesses if their infrastructure supports machine learning algorithms and if the team has the skills to manage AI development.

2. **Operational Feasibility:** Assesses whether the project aligns with the organization's existing processes and satisfies user needs.

Considerations: Will the system integrate with current workflows? Will users be able to adapt to the new system?

Example: For a retail company implementing a new inventory management system, operational feasibility would check if employees can transition smoothly to the new system without disrupting sales.

3. **Economic Feasibility:** Analyzes the financial aspects to determine if the project is cost-effective.

Considerations: Do the benefits outweigh the costs? What is the expected return on investment (ROI)?

Example: Developing an e-commerce platform involves significant costs; the company must evaluate if the expected increase in online sales justifies these expenses.

4. **Legal Feasibility:** Ensures that the project complies with legal standards and regulations.

Considerations: Does the system adhere to data protection laws, copyright issues, or industry-specific regulations?

Example: A healthcare application must comply with laws like the Health Insurance Portability and Accountability Act (HIPAA) to protect patient data.

5. **Schedule Feasibility:** Evaluates whether the project can be completed within the desired time frame.

Considerations: Are there adequate resources to meet the deadlines? Is the timeline realistic?

Example: Developing a mobile app to be released in sync with a marketing campaign requires careful scheduling to ensure timely delivery.

5. Example of a Feasibility Study

Let's consider the example of a bank planning to develop a mobile banking app:

- **Technical Feasibility:** Assesses if the bank's IT team can develop the app using the latest mobile technology. Do they have the necessary skills and hardware?
- **Economic Feasibility:** Analyzes whether the app's development and maintenance costs are within the bank's budget and if it will attract enough users to justify the investment.
- **Operational Feasibility:** Evaluates if the mobile app will integrate seamlessly with the bank's existing systems and enhance customer satisfaction.
- **Legal Feasibility:** Ensures that the app will comply with financial industry regulations like secure data handling and user privacy.
- **Schedule Feasibility:** Checks if the development can be completed on time, especially if it's planned to launch during a specific financial quarter.

6. Importance of Feasibility Studies in Software Projects

- **Strategic Alignment:** Helps ensure that the project aligns with the strategic goals of the organization.
- **Avoids Wasted Investment:** Saves resources by identifying non-viable projects early.

- **Supports Business Growth:** Facilitates innovation and growth by investing in feasible projects with a high chance of success.

Requirements Engineering and Feasibility Study are essential components in software development. Requirements Engineering helps define what the software should achieve, ensuring that developers build the right product, while Feasibility Studies ensure that the project is practical, achievable, and beneficial. Together, these processes reduce risks, save costs, and guide the project to success by laying a strong foundation for all software development activities.

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 4

REQUIREMENTS ELICITATION AND ANALYSIS

1. Requirements Elicitation

Definition:

Requirements elicitation is the process of gathering and discovering the needs, expectations, and constraints from users, stakeholders, and anyone who will be affected by the system. This stage focuses on collecting as much relevant information as possible to understand what the software should do.

Goal of Requirements Elicitation:

The primary goal is to capture a complete set of requirements that accurately reflects what stakeholders need from the system. This helps to ensure that the development team has a clear understanding of the desired features and functionalities.

2. Who Are the Stakeholders?

Stakeholders are anyone with an interest in the development and use of the software system. Understanding who these stakeholders are is critical to gathering all relevant requirements.

1. **End-Users:**
    - These are the people who will directly use the software. They have firsthand knowledge of the tasks they need to accomplish using the system.
    - **Example:** In a hospital management system, end-users include doctors, nurses, and administrative staff who interact with the software daily.

2. **Managers:**
    - Managers ensure that the system aligns with the overall business strategy and objectives. They focus on how the software will impact the organization's operations.
    - **Example:** For a retail store, the manager will be interested in how the inventory management system supports sales and improves stock control.
3. **Developers:**
    - Developers are the technical team responsible for building the software according to the requirements gathered.
    - **Example:** Developers need detailed requirements to understand the specific technical tasks they must perform to create an e-commerce website.
4. **Regulators:**
    - Regulators ensure that the system meets industry standards, compliance regulations, and legal requirements.
    - **Example:** In financial software, regulators might include legal experts who ensure compliance with financial reporting standards and data protection laws.

3. Methods of Requirements Elicitation

The process of elicitation involves various methods to gather information from stakeholders, each suited for different situations. These methods aim to cover all aspects of what the software must achieve.

1. **Interviews:** This involves one-on-one or group discussions with stakeholders to understand their needs. It is a direct way to gather specific details.
    - **Types of Interviews:**
        - **Structured Interviews:** Have a predefined set of questions to ensure that all topics are covered systematically.
        - **Unstructured Interviews:** More flexible, allowing stakeholders to speak freely about their thoughts, which can uncover new insights.
    - **Example:** In developing an online banking system, a structured interview with customers might ask specific questions about their needs for security features and ease of transactions.

2. **Brainstorming Sessions:** This technique gathers stakeholders in a creative discussion to generate a wide range of ideas about the system.
    - **Phases of Brainstorming:**
        - **Idea Generation:** All participants share their ideas without evaluating them.
        - **Evaluation Phase:** The ideas are discussed, and the most relevant ones are selected for further development.
    - **Example:** During the design of a new mobile game, a brainstorming session might involve designers, developers, and marketers generating ideas for game mechanics and features.
3. **Facilitated Application Specification Technique (FAST):** FAST is a structured group meeting led by a facilitator, aimed at discussing specific requirements in detail. It encourages collaboration between stakeholders.
    - **Role of the Facilitator:** Ensures that the meeting stays focused, that everyone participates, and that misunderstandings are clarified.
    - **Example:** For a corporate HR management system, the facilitator might lead a session where HR managers, IT staff, and end-users discuss the key features required for tracking employee performance.
4. **Quality Function Deployment (QFD):** QFD is used to transform customer needs into engineering specifications. It involves detailed matrices (often called "House of Quality") to relate customer desires to system capabilities.
    - **Steps in QFD:**
        - Collecting Customer Needs: Understanding what features or services customers value the most.
        - Relating these needs to technical requirements: Defining how each need will be addressed by specific system functionalities.
    - **Example:** A car manufacturer uses QFD to convert customer demands for fuel efficiency and safety into specific design changes and technological innovations.
5. **Use Case Approach:** Use cases describe how different types of users (actors) interact with the system to achieve specific goals. It helps in visualizing the functional requirements of the system.
    - **Components of a Use Case:**
        - **Actors:** The roles that users play when interacting with the system.

        - **Scenarios:** Step-by-step description of the interactions between the actor and the system.
    - **Example:** For an online shopping site, a use case might detail the steps a customer takes to log in, search for products, add items to the cart, and make a purchase.
6. **Ethnography:** Ethnography involves observing users in their natural work environment to understand how they perform tasks and interact with current systems. It focuses on actual behaviour rather than just stated preferences.
    - **Benefits:** Provides insights into workflow inefficiencies and the real-world challenges users face.
    - **Example:** A researcher observing factory workers might notice that the existing software interface is causing delays in processing orders, leading to design improvements.

4. Challenges in Requirements Elicitation

Requirements elicitation is often complex due to various challenges, such as:

- **Communication Barriers:** Different stakeholders might have varying terminologies and perspectives, making it difficult to understand their needs.
- **Changing Requirements:** Stakeholders' needs can evolve over time, resulting in a moving target for requirements.
- **Unstated Requirements:** Users might not be aware of all their needs or assume that certain requirements are obvious.

5. Requirements Analysis

Definition:

Requirements analysis is the process of refining, validating, and prioritizing the gathered requirements. It aims to ensure that the requirements are complete, consistent, and feasible before the development process begins.

Key Objectives of Requirements Analysis:

- **Clarity:** Ensures that every requirement is clear and understandable to all stakeholders, reducing the risk of misunderstandings.
- **Conflict Resolution:** Identifies and resolves any conflicts between different stakeholders' needs and expectations.
- **Prioritization:** Determines the importance of each requirement to focus on the most critical features that deliver maximum value to users.

6. Steps in Requirements Analysis

1. **Gap Analysis:** To identify missing requirements or areas where the system might fail to meet business goals.
    - **Method:** Compare the current state of the system with the desired state to pinpoint gaps.
    - **Example:** For a customer support system, gap analysis might reveal that there is no feature for tracking customer complaints, which is essential for improving service.
2. **Conflict Resolution:** To handle conflicting requirements that arise from different stakeholder groups.
    - **Approach:** Facilitating discussions to find a compromise or prioritizing one requirement over another based-on project goals.
    - **Example:** Marketing wants to add more visual elements to an app, but the technical team is concerned about performance issues. Conflict resolution involves finding a balance between aesthetics and speed.
3. **Feasibility Analysis:** To check whether the requirements can be realistically implemented within the constraints of technology, budget, and time.
    - **Considerations:** Evaluates if the requirements align with the project's scope and the resources available.
    - **Example:** For a startup developing a machine learning product, feasibility analysis might determine that they lack the technical expertise or budget to implement certain advanced features.

7. Importance of Requirements Analysis

- **Avoids Scope Creep:** Ensures that the project remains focused on the agreed-upon requirements and does not expand unnecessarily.
- **Ensures System Quality:** By validating and prioritizing requirements, it helps create a well-defined product that meets user needs effectively.
- **Reduces Costs:** Catching issues early in the analysis phase can prevent expensive rework during later stages of development.

8. Practical Example of Requirements Analysis

Scenario: A transportation company is developing a fleet management system.

- **Stakeholders:** Fleet managers, drivers, IT support, logistics planners, and finance officers.
- **Gap Analysis:** Identified that the current system does not support real-time vehicle tracking, which is crucial for efficient fleet management.
- **Conflict Resolution:** Drivers want an easy-to-use interface, while fleet managers need detailed analytics. The solution involves creating different user views tailored to each group's needs.
- **Prioritization:** The top priority is to implement the vehicle tracking feature due to its impact on operational efficiency.

Requirements Elicitation and Requirements Analysis are critical phases in software development that ensure the final product aligns with user needs and business goals. These processes involve actively engaging stakeholders, employing various techniques to gather information, and rigorously analysing the requirements to ensure clarity, feasibility, and completeness. Properly executed, these activities set the foundation for successful software development, reducing risks, avoiding costly changes, and delivering value to both users and the organization.

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 5

REQUIREMENTS VALIDATION, MANAGEMENT, AND MONITORING

1. Requirements Validation

Definition:

Requirements validation is the process of checking that the documented requirements accurately reflect the customer's needs and are feasible to implement. It ensures that the software being developed will meet the users' expectations and perform the functions it is supposed to.

Importance of Requirements Validation:

1. **Avoiding Costly Errors:**
    - Errors in the requirements phase can be very expensive to fix later in the development process. Catching these issues early can save both time and money.
    - **Example:** Suppose there is a misunderstanding about the security features required for a banking application. If this issue is identified during the validation stage, it can be corrected before the development starts, saving significant costs that would have been incurred if the error were discovered later.
2. **Aligning Expectations:**
    - Validation ensures that the system will function as the customer expects. It aligns what the development team plans to build with what the customer actually needs.
    - **Example:** Validating that an inventory management system will handle stock levels exactly as the client expects, preventing any gaps between the client's needs and the system's capabilities.

Methods of Requirements Validation

1. **Requirements Reviews:**
   - **Description:** This method involves bringing together stakeholders and project team members to review the requirements document in detail.
   - **How It Works:** The team looks for errors, ambiguities, inconsistencies, and missing details in the requirements.
   - **Example:** Requirements review meeting for a Customer Relationship Management (CRM) system might involve sales, marketing, and IT teams examining the document to ensure it captures all their needs.
2. **Prototyping:**
   - **Description:** Creating a simplified version of the system or certain features to visualize and validate requirements.
   - **Benefits:** Prototyping helps stakeholders understand how the system will work and ensures the design meets their expectations.
   - **Example:** Building a prototype of a mobile app's user interface allows users to see what the app will look like and suggest improvements before full development begins.
3. **Test Case Generation:**
   - **Description:** Writing test cases based on the requirements to check if each feature can be tested properly.
   - **Purpose:** Test cases help identify whether the requirements are clear, complete, and executable.
   - **Example:** For an online shopping website, test cases could be created for each feature, like adding items to the cart or processing payments, to verify that the functionality works as described in the requirements.

2. Requirements Management

Definition:

Requirements management is the process of tracking, handling, and controlling changes to the requirements throughout the software development life cycle. It ensures that any updates to the requirements are carefully documented, analysed, and communicated to all relevant stakeholders.

Key Activities in Requirements Management

1. **Tracking Changes:**
   - **Description:** Keeping a detailed record of all changes made to the original requirements during the development process.
   - **Importance:** Helps manage the evolving needs of the project while maintaining a history of what has been modified.
   - **Example:** If a new feature like a wish list option is added to an e-commerce platform, this change is tracked through the requirements management system to ensure everyone is aware of it.
2. **Impact Analysis:**
   - **Description:** Analysing the effect of changes in one requirement on other parts of the system.
   - **Purpose:** To understand how a new or altered requirement might influence the system's architecture, design, or functionality.
   - **Example:** Adding a new feature to a website's shopping cart might affect the entire checkout process and payment integration, requiring an analysis to see the ripple effects on related features.
3. **Communication:**
   - **Description:** Ensuring that all stakeholders, including developers, testers, and customers, are aware of any changes in requirements.
   - **Importance:** Clear communication avoids misunderstandings and ensures that everyone is on the same page regarding the project's direction.
   - **Example:** Informing all team members when a change in project scope affects the development timeline and the overall delivery date.

3. Requirements Monitoring

Definition:

Requirements monitoring involves continuously tracking the progress of the development process to ensure that it stays aligned with the original requirements and project goals. Monitoring helps detect issues early, allowing for quick corrective actions.

Purpose of Requirements Monitoring:

1. **Early Detection of Issues:**
   - Regular monitoring helps identify problems as soon as they arise, enabling the team to fix them before they escalate into bigger issues.
   - **Example:** If a feature in a software project is not being developed according to the agreed requirements, monitoring can reveal this discrepancy early, allowing adjustments to be made.
2. **Keeping the Project on Track:**
   - Ensures that the project remains aligned with the original objectives, goals, and customer expectations.
   - **Example:** Monitoring the development of a new "order tracking" feature in an online store ensures that it is built exactly as specified in the requirements document.

Example of Requirements Management and Monitoring in Practice

Let's consider a scenario of an online booking system for a travel agency:

1. **Requirements Management:**
   - **Tracking Changes:** The project team adds a requirement to support multi-currency payments in the booking system. This change is logged in the management system, and all stakeholders are informed.
   - **Impact Analysis:** Adding multi-currency support affects the payment gateway integration and might also impact the user interface design. An analysis is conducted to understand the adjustments needed in other parts of the system.
   - **Communication:** The development team, testers, and customer service representatives are updated about the new multi-currency feature to ensure they are ready for its implementation.
2. **Requirements Monitoring:**
   - **Early Detection:** The monitoring process reveals that the multi-currency feature is not properly integrated with the discount calculation module. The issue is flagged early, and developers are instructed to fix it.

- **Keeping on Track:** Continuous monitoring ensures that all parts of the booking system align with the specified requirements, providing a consistent experience for the end-users.

Importance of Requirements Validation, Management, and Monitoring

1. **Ensures Quality:**
   - These processes ensure that the final product meets the high standards expected by stakeholders. Validation checks for accuracy, management handles changes systematically, and monitoring keeps the project aligned with its goals.
2. **Reduces Risks:**
   - By validating requirements, managing changes, and monitoring progress, potential issues are addressed proactively, reducing the risks of project delays, cost overruns, and feature mismatches.
3. **Improves Customer Satisfaction:**
   - A well-managed and monitored requirements process means that the final product is more likely to meet or exceed customer expectations, resulting in higher satisfaction and trust in the development team.

Requirements validation, management, and monitoring are critical stages in the software development lifecycle that ensure the project's success. Validation confirms that requirements accurately reflect user needs, management tracks and controls any changes, and monitoring keeps the project on course. Together, these practices prevent costly mistakes, maintain alignment with goals, and deliver a product that truly meets the stakeholders' expectations.

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 6

INTRODUCTION TO VERSION CONTROL SYSTEM, GIT, GITHUB

Introduction to Version Control Systems (VCS)

In software development, managing and organizing code changes is essential. Projects can involve multiple developers who work on different parts of the code simultaneously. Without a way to track changes, things can quickly get confusing, and mistakes can easily happen. This is where Version Control Systems (VCS) come in.

Key Points:

- VCS tools help in **tracking, managing, and organizing code changes**.
- They **store different versions** of the code, so you can go back to previous versions if something goes wrong.
- VCS enables **multiple developers** to work on the same project, allowing easy **collaboration**.
- Example: Think of VCS like a "track changes" feature in a Word document, but for code, where you can see who made changes, when, and why.

What is a Version Control System (VCS)?

A Version Control System (VCS) is a tool that helps keep track of the changes made to files over time. It stores every version of the code, so you can revert to earlier versions if needed. It allows developers to work together without overwriting each other's changes.

Benefits of VCS:

1. **Keeps a history of changes:** Every modification is recorded, and you can see who made which change.

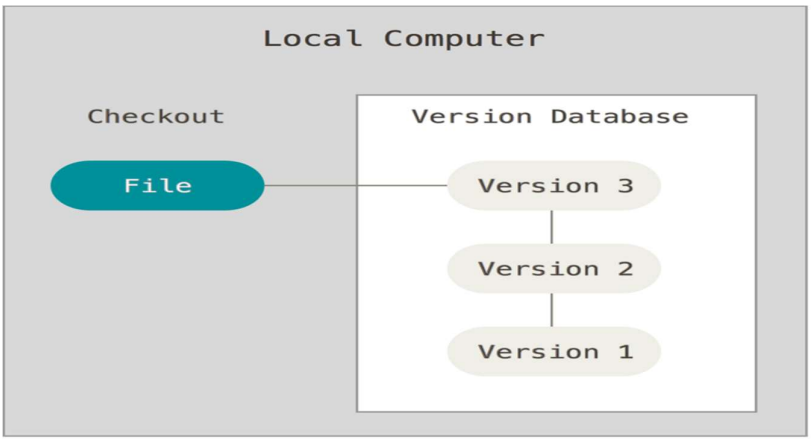
2. **Collaboration:** Multiple developers can work on the same project simultaneously without conflicts.
3. **Easy recovery:** If a mistake is made, you can go back to a previous version without losing any work.

Example: Imagine you're writing a report, and every time you save it, a backup copy is created. You can always go back to an earlier version if the new changes don't work out.

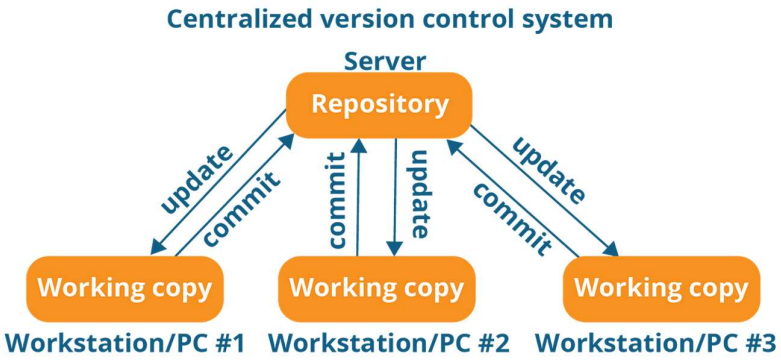
Types of Version Control Systems

There are three main types of VCS, each suited to different situations:

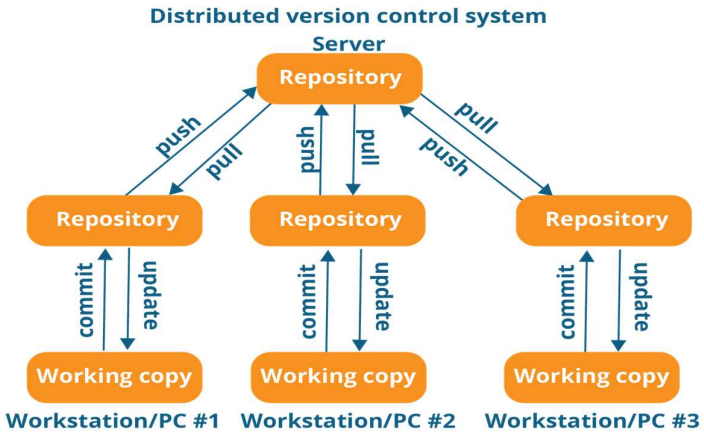
1. **Local Version Control System:**
   - Stores versions on your local computer.
   - **Problem:** If your computer crashes, you lose everything.
   - **Example:** Saving different versions of a file on your desktop.



2. **Centralized Version Control System:**
   - All versions are stored on a central server. Users connect to this server to access the files.
   - **Problem:** If the server crashes, you could lose all your work.
   - **Example:** Like a shared folder on Google Drive that everyone can edit.



3. **Distributed Version Control System (DVCS):**
    - Each user has a complete copy of the project, including its entire history.
    - **Advantage:** Even if the main server fails, copies exist on multiple devices, ensuring data safety.
    - **Example:** Think of it like everyone on the team having their own complete backup of a shared project folder.



Why Distributed VCS is Better:

- Provides **more safety** because data is stored in multiple places.
- Allows **better collaboration** since everyone has a complete copy, making it easy to work offline.

Introduction to Git

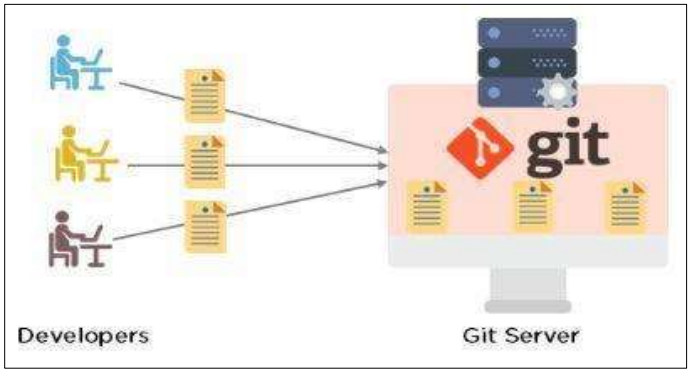
As projects grow larger and involve more people, managing changes can become difficult. This is where **Git**, a Distributed VCS, comes in.

What is Git?

- **Git** is a tool that tracks changes in files, especially code, over time.
- It was created by **Linus Torvalds** in **2005** to manage the development of the Linux kernel.
- Git helps multiple people work on the same project simultaneously without causing conflicts.

Features of Git:

1. **Free and open-source:** Anyone can use Git without cost.
2. **Supports collaboration:** Multiple developers can work on the same project at once.
3. **Non-linear development:** Git allows you to create and manage multiple branches of a project.
4. **Tracks entire history:** Every change is saved, so you can go back to any version.



Why Git is Popular:

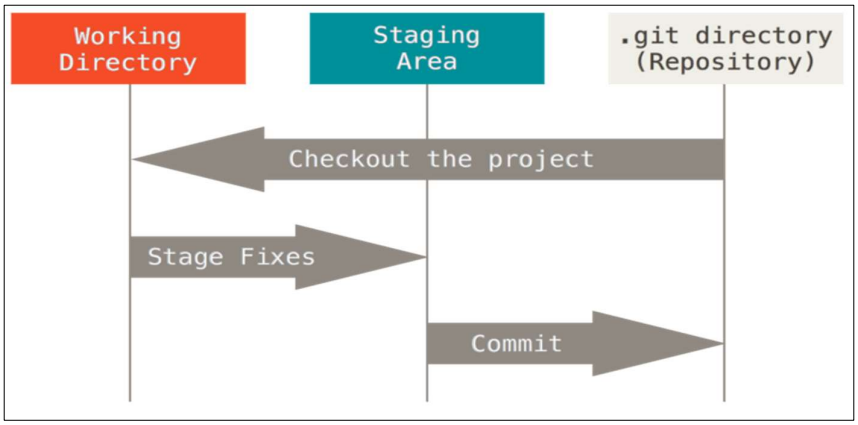
- Efficiently handles large projects with many contributors.
- Allows for **branching**, where developers can create separate copies of the code to work on features without affecting the main codebase.

How Git Works (Git Workflow)

To use Git effectively, it's important to understand its workflow, which involves three main stages:

1. **Working Directory:**
    - This is where you make changes to your files.
    - **Example:** Editing a file on your computer.

2. **Staging Area:**
    - This is where you prepare selected changes to be saved permanently.
    - **Example:** Think of it as placing files in a "to-do list" before adding them to your final project.
3. **Git Repository:**
    - This is where all your changes are stored, including the entire history of your project.
    - **Example:** A folder containing all your backups and old versions of the files.



Steps in Git Workflow:

1. **Edit files** in the Working Directory.
2. **Stage** the files you want to commit (save permanently) in the Staging Area.
3. **Commit** the changes to the Git Repository, where they are saved permanently.

Transition to GitHub: Why Do We Need It?

While Git is great for tracking changes, there's a need for **sharing and collaborating on code** across different locations. This is where **GitHub** comes in.

Problem with Just Git:

- If you're working on a project with people who are not in the same place, it's difficult to share changes and keep everything up-to-date.
- GitHub solves this by providing a **platform** where you can **store your Git projects online**, making it easier for teams to collaborate.

What is GitHub?

GitHub is a **web-based platform** that hosts Git repositories, making it easier for teams to **share, collaborate, and manage** their code projects.

Key Points:

1. **Definition:** GitHub is an online service where developers can store, manage, and share their Git projects.
2. **Purpose:** Allows teams to work on the same project from different locations, making collaboration easy.
3. **Owned by:** Microsoft since 2018.
4. **Example:** Imagine GitHub as a social network for programmers where they can share and discuss their code.

Features of GitHub

GitHub offers many features that make project management and collaboration more efficient:

1. **Project Management:** Helps teams track tasks, assign work, and organize projects.
    - **Example:** Like a to-do list that everyone can see and update.
2. **Secure Code Packages:** Safely store and share pieces of code with your team or publicly.
    - **Example:** Uploading a useful function that others can use.
3. **Team Collaboration:** Tools to communicate, review, and manage code changes.
    - **Example:** Commenting on a teammate's code to suggest improvements.
4. **Pull Requests (Code Review):** Discuss and review code changes before merging them into the main project.
    - **Example:** Like submitting a draft for approval before it's published.
5. **Security Tools:** Detect and fix weaknesses in your code.
    - **Example:** Checking for errors before releasing a product.
6. **Code Hosting:** Millions of repositories hosted, with easy access to project details and history.

Alternatives to GitHub

While GitHub is the most popular platform for hosting Git projects, there are other services that provide similar features:

1. **GitLab:** A complete platform for DevOps and collaboration, offering tools for project management and CI/CD.
2. **Bitbucket:** Often used by smaller teams; integrates well with Atlassian tools like Jira.
3. **AWS Code Commit:** Managed Git hosting provided by Amazon Web Services.
4. **SourceForge:** Known for hosting open-source software projects.

Differences Between Git and GitHub

While Git and GitHub are used together, they serve different purposes. Here's how they compare:

Aspect	Git	GitHub
Type	Software tool for tracking code changes	Online platform for hosting Git projects
Interface	Works through the command line	User-friendly graphical web interface
Location	Installed on your local computer	Accessible through a web browser
Purpose	Manages versions and history of code	Helps in sharing, collaborating, and managing projects
Ownership	Developed by Linus Torvalds	Owned by Microsoft
Focus	Version control and tracking code changes	Collaboration, project management, and code sharing
Example	A tool to save and track changes in your project files	A website where you can share your Git projects with others

Git: An essential tool that tracks changes, manages versions, and helps developers work on code without conflicts.

GitHub: A platform that allows teams to share and collaborate on Git projects from anywhere in the world.

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 7

BASIC GIT COMMANDS - VERSION, CONFIG, INIT, STATUS, ADD, COMMIT, DIFF, HELP

Basic Git Commands

1. **Git Version:**

   The `git --version` command is used to check if Git is installed on your system. This command displays the current version of Git that is available.

   For example, running: `git --version`

might return `git version 2.37.1`, confirming that Git is installed.

Knowing the installed Git version helps in troubleshooting and ensuring compatibility with certain Git features or integrations. It is the first step before using Git functionalities.

2. **Git Config:**

   The `git config` command is used to set up the user identity for commits, including name and email address. This ensures that every commit you make in the project history is associated with your correct identity.

Usage Example:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Explanation:

- **`--global:`** Applies the configuration globally for the system, meaning it applies to all projects.
- **`user.name`** and **`user.email:`** These options specify the identity that will appear in the commit history.

```
kmit@DESKTOP-5OB7B83 MINGW64 ~
$ git config --global user.name "savram674"

kmit@DESKTOP-5OB7B83 MINGW64 ~
$ git config --global user.email "savitharamesh674@gmail.com"

kmit@DESKTOP-5OB7B83 MINGW64 ~
```

Example Output:

```
git config --global user.name "Madhurika"
git config --global user.email "madhurika.kmit@example.com"
```

In this example, Jane Doe's details will be associated with all commits made on this system unless overridden for a specific repository.

3. **Git Init:**

   The **`git init`** command is used to create a new Git repository. This initializes a directory as a new Git repository and creates a **`.git`** subdirectory to store all Git-related files.

Usage Example: **`git init`**

Explanation: This command creates an empty Git project within the directory. After running **`git init`**, the project is now under version control, and you can start tracking changes.

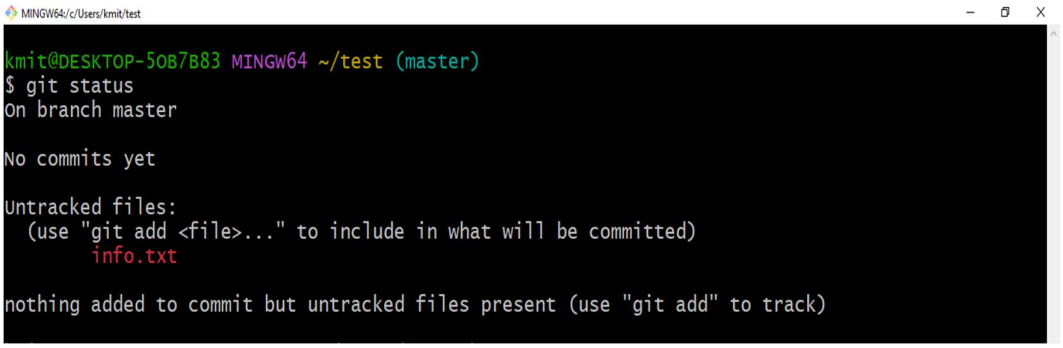
```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test
$ git init
Initialized empty Git repository in C:/Users/kmit/test/.git/
```

Real-World Example: Imagine you have a directory called `my-project`. Running `git init` inside it will turn it into a Git repository, allowing you to track its changes.

4. **Git Status:**

   The `git status` command displays the state of your working directory and staging area. It shows what changes have been staged, what files are not being tracked by Git, and what changes are yet to be committed.

Usage Example: `git status`



Explanation: This command is essential to get a summary of your repository's current state. It lets you know whether files need to be added to the staging area or if there are changes to commit.

5. **Git Add:**

   The `git add` command is used to add changes to the staging area. You must stage changes before committing them to ensure only the desired changes are included in the commit.

Usage Example: `git add .`

Explanation:

- `.` stages all changes in the current directory.
- Specific files can be added by replacing `.` with the file names.

Example: `git add index.html`

This adds the file `index.html` to the staging area.

6. **Git Commit:**

   The `git commit` command saves your changes to the local repository. Each commit represents a snapshot of the repository at a particular point in time. You must include a message explaining what the commit does.

Usage Example: `git commit -m "Added new feature"`

Explanation:

- `-m`: Allows you to add a commit message directly from the command line.
- The commit message should be concise but descriptive enough to explain the change.

```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git commit -m "info file added"
[master (root-commit) cf73846] info file added
 1 file changed, 3 insertions(+)
 create mode 100644 info.txt
```

Example Output:

```
[master (root-commit) 1a2b3c4] Added new feature
```

7. **Git Diff:**

   The `git diff` command shows the differences between files in your working directory and the staging area. It is useful to review changes before committing them.

Usage Example:

```
git diff
```

```
MINGW64:/c/Users/kmit/test
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git commit -m "info file added"
[master (root-commit) cf73846] info file added
 1 file changed, 3 insertions(+)
 create mode 100644 info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ notepad info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git diff
diff --git a/info.txt b/info.txt
index 31c4db7..4111fbe 100644
--- a/info.txt
+++ b/info.txt
@@ -1,3 +1,6 @@
 This is my new file
 username : savram674
-email : savitharamesh674@gmail.com
\ No newline at end of file
+email : savitharamesh674@gmail.com
+
+Department :Information Technology
+Designation: Assistant Professor
\ No newline at end of file
```

Explanation: Running **`git diff`** will display the differences between your modified files and the last commit, helping you see exactly what changes will be staged and committed.

8. Git Help:

The **`git help`** command provides documentation and detailed usage information for any Git command.

Usage Example: **`git help <command>`**

Explanation: You can use **`git help`** to get detailed information about any Git command, such as **`git help add`** to learn more about the **`git add`** command.


```
SSPL-LP-DNS-YT0+Simplilearn@SSPL-LP-DNS-YT01 MINGW64 ~
$ git help add
```

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 8

GIT COMMANDS: WORKING WITH LOCAL AND REMOTE REPOSITORIES - BRANCHES, CHECKOUT, MERGE, REVERT, LOG

Branches in Git

A **branch** in Git is like a **separate copy of your project** where you can make changes without affecting the main version. You might need different branches to work on new features, fix bugs, or prepare for a release.

- Imagine you have a main branch (called **master**) that has the latest stable version of your project. If you need to create a new feature (like adding a search function), you would create a **feature branch**. This allows you to work on the search function independently. Once the feature is complete, you can combine it (merge) with the master branch.

  Types of Branches:

  - **Main Branch (Master)**: The main project everyone uses.
  - **Feature Branch**: A branch to work on specific new features.
  - **Bug Fix Branch**: Specifically for fixing bugs.
  - **Release Branch**: Prepares and tests the project before it is released to users.

Important Commands:

- `git branch <branch_name>:` Creates a new branch, such as `git branch search-feature.`
- `git branch -a:` Shows a list of all branches, including the main branch, feature branches, and remote branches.

- `git branch -d <branch_name>`: Deletes a branch that you no longer need, such as `git branch -d bugfix`.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Sample_Example (master)
$ git branch project_changes

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git branch -a
* master
  project_changes

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git branch -d project_changes
Deleted branch project_changes (was 4e87650).

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git branch -a
* master
```

2. Switching Between Branches: Checkout

The **git checkout** command is used to **switch between branches**. This allows you to move from one task to another easily. For example, you may be working on a new feature in one branch but need to fix a bug in another branch.

- Let's say you're working on a new search feature in a branch called `search-feature`, but suddenly a critical bug is found in the main branch (`master`). You can **switch to the bug fix branch** using `git checkout` and fix the bug. Once the bug is fixed, you can switch back to the `search-feature` branch and continue working without losing your progress.

Important Commands:

- `git checkout <branch_name>`: Switches to an existing branch, like `git checkout bugfix` to fix the bug.
- `git checkout -b <new_branch>:` Creates a new branch and switches to it immediately. For example, `git checkout -b update-design` will create and move to a new branch where you can work on a design update.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git branch project_changes

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git checkout project_changes
Switched to branch 'project_changes'

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ git checkout -b project_bugs
Switched to a new branch 'project_bugs'

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_bugs)
$ git branch -a
  master
* project_bugs
  project_changes
```

3. Combining Work: Merge

After working on a separate branch for a while (e.g., developing a new feature or fixing a bug), you will want to **combine your work** with the main project. This is where **git merge** comes in—it merges the changes from one branch into another, typically merging changes into the **master** branch.

- You've finished working on the `search-feature` branch, and it's time to **merge it into the master branch**. By using `git merge search-feature` while on the `master` branch, you combine all the changes from the search-feature branch into the master. Now, the master branch contains both the original stable version and the new feature you've developed.

Important Command:

- `git merge <branch_name>`: Merges the changes from the specified branch into the current branch. For example, `git merge search-feature` combines the search feature with the master branch.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git branch project_changes

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (master)
$ git checkout project_changes
Switched to branch 'project_changes'

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ ls
a.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ nano b.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ ls
a.txt  b.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ git add .
warning: LF will be replaced by CRLF in b.txt.
The file will have its original line endings in your working directory

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ git commit -m "New file added in new branch project_changes"
[project_changes e525410] New file added in new branch project_changes
 1 file changed, 1 insertion(+)
 create mode 100644 b.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ ls
a.txt  b.txt
```

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_E
xample (project_changes)
$ git checkout master
Switched to branch 'master'

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ ls
a.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ git merge project_changes
Updating 4e87650..e525410
Fast-forward
 b.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 b.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ ls
a.txt  b.txt
```

4. Viewing History: Log

Git keeps a **detailed history** of all changes (called **commits**) made to the project. The **git log** command shows you this history, so you can see who made changes, when they made them, and what the changes were. This is useful for tracking the progress of the project and identifying when bugs were introduced.

- Suppose you want to know when a specific feature was added or who worked on a particular part of the code. By using the `git log` command, you can see a detailed history of all the changes. Each entry in the log will show:
  - **Commit Hash**: A unique identifier for each change.
  - **Author**: The person who made the change.
  - **Date**: When the change was made.
  - **Message**: A short description of the change, like "added search functionality."

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ git log
commit e525410c637e693b0b512bbb2fe67ca8c3763b14 (HEAD -> master)
Author: Madhurika, Budaraju <budarajumadhurika@gmail.com>
Date:   Wed Oct 16 00:16:39 2024 +0530

    New file added in new branch project_changes

commit 4e87650a0f49e0b6f82b53e693099f5987eeb972 (project_bugs)
Author: Madhurika, Budaraju <budarajumadhurika@gmail.com>
Date:   Tue Oct 15 14:37:32 2024 +0530

    Created new file
```

- **Simplified View**:
  If you don't want to see too much detail, you can use the `git log --oneline` command. This shows a summary of each commit, making it easier to quickly scan through the project's history.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ git log --oneline
e525410 (HEAD -> master) New file added in new branch project_changes
4e87650 (project_bugs) Created new file
```

Important Commands:

- `git log`: Shows the complete history of commits in a branch.
- `git log --oneline`: Shows a simpler view with one line per commit, like `f3b9c2b Added search feature.`

5. Undoing Changes: Revert

If a mistake was made in a previous commit, Git provides a way to **undo those changes** using the `git revert` command. This doesn't delete the commit but creates a new commit that cancels out the earlier changes. This way, the history of the project is preserved.

- Imagine you've added a new feature but later discover that it introduced a bug. Instead of deleting that commit, you can use `git revert` to undo the changes made in that commit. For example, if the commit hash for the problematic commit is `f3b9c2b`, you would use `git revert f3b9c2b`. Git will then create a new commit that undoes the changes introduced by that commit without deleting it from the project history.

Important Command:

- `git revert <commit_hash>`: Reverts (undoes) the changes from a specific commit. For example, `git revert f3b9c2b` will undo the changes made in the commit with the ID `f3b9c2b`.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (master)
$ git checkout project_changes
Switched to branch 'project_changes'

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (project_changes)
$ git log --oneline
e525410 (HEAD -> project_changes, master) New file added in new branch project_changes
4e87650 (project_bugs) Created new file

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (project_changes)
$ git revert e525410
[project_changes 7c2e526] Revert "New file added in new branch project_changes"
 1 file changed, 1 deletion(-)
 delete mode 100644 b.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (project_changes)
$ git status
On branch project_changes
nothing to commit, working tree clean

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (project_changes)
$ ls
a.txt
```

Flow of Using Git Commands (with Examples):

1. **Create a Branch**:

   You create a branch to work on a new feature:

```
git branch new-feature
```

2. **Switch to the Branch**:

   You switch to the new branch to start working on the feature:

   ```
   git checkout new-feature
   ```

3. **Develop Your Feature and Commit Changes**:

   You make changes, then commit them:

   ```
   git commit -m "Developed search functionality"
   ```

4. **Switch to Another Branch to Fix a Bug**:

   Suddenly, you need to fix a bug, so you switch to the bug fix branch:

   ```
   git checkout bugfix
   ```

5. **Merge the Bug Fix Into the Main Branch**:

   Once the bug is fixed, you merge it into the master branch:

   ```
   git checkout master
   git merge bugfix
   ```

6. **Check History Using Log**:

   You want to review the commits to make sure everything is correct:

   ```
   git log
   ```

7. **Revert a Mistake**:

   If you discover a mistake in a previous commit, you can revert it:

   ```
   git revert f3b9c2b
   ```

SOFTWARE ENGINEERING

UNIT – 2

TOPIC – 9

GIT COMMANDS: WORKING WITH REMOTE REPOSITORIES - REMOTE, CLONE, PULL, PUSH, FORK

1. Git Remote

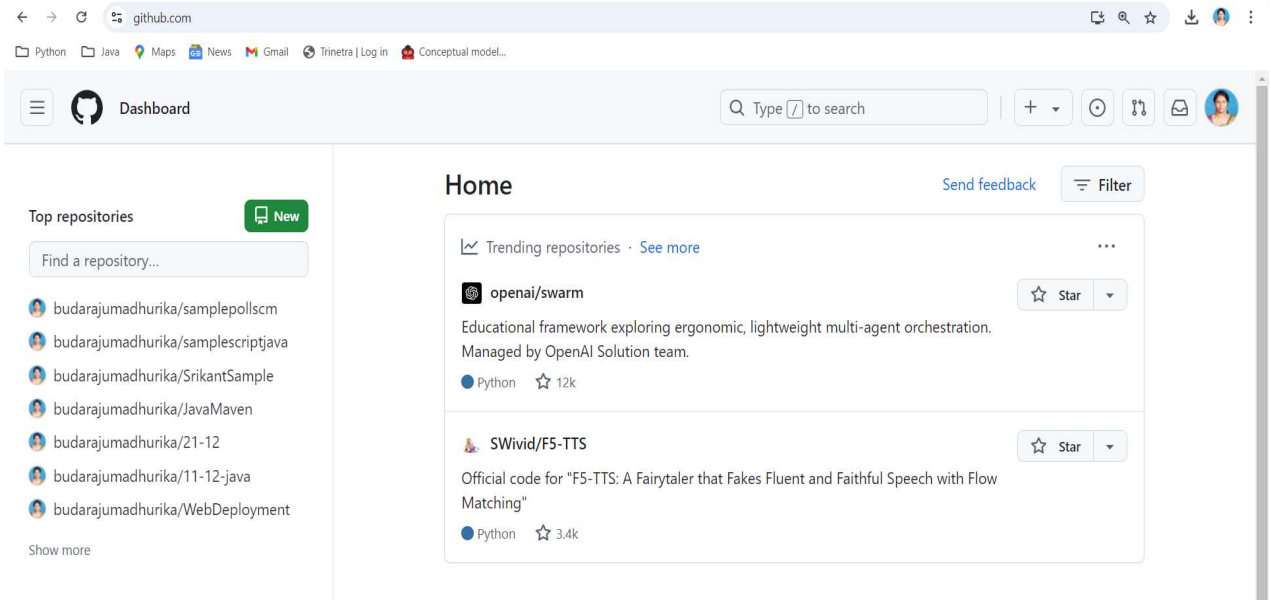
Git remote allows you to manage connections to **remote repositories**, which are Git repositories hosted on the internet (like **GitHub** or **Bitbucket**). A remote repository is where you can save or retrieve your project from a server, and you use this command to add, remove, or view the addresses (URLs) of these repositories.

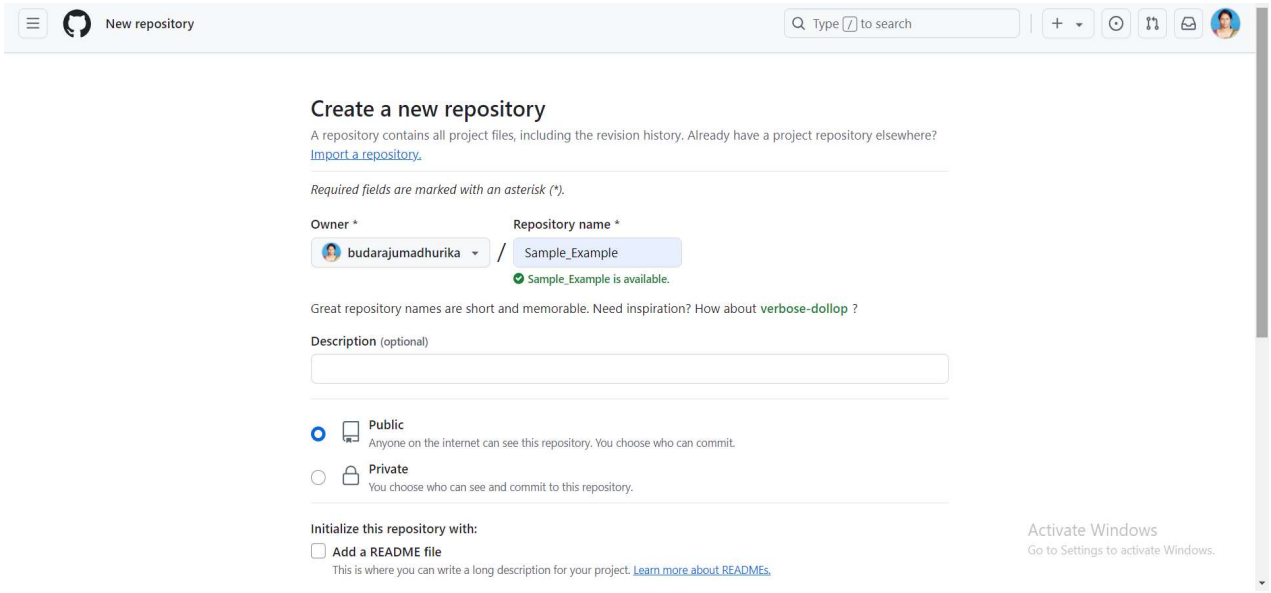
- To add a new remote repository, you use:

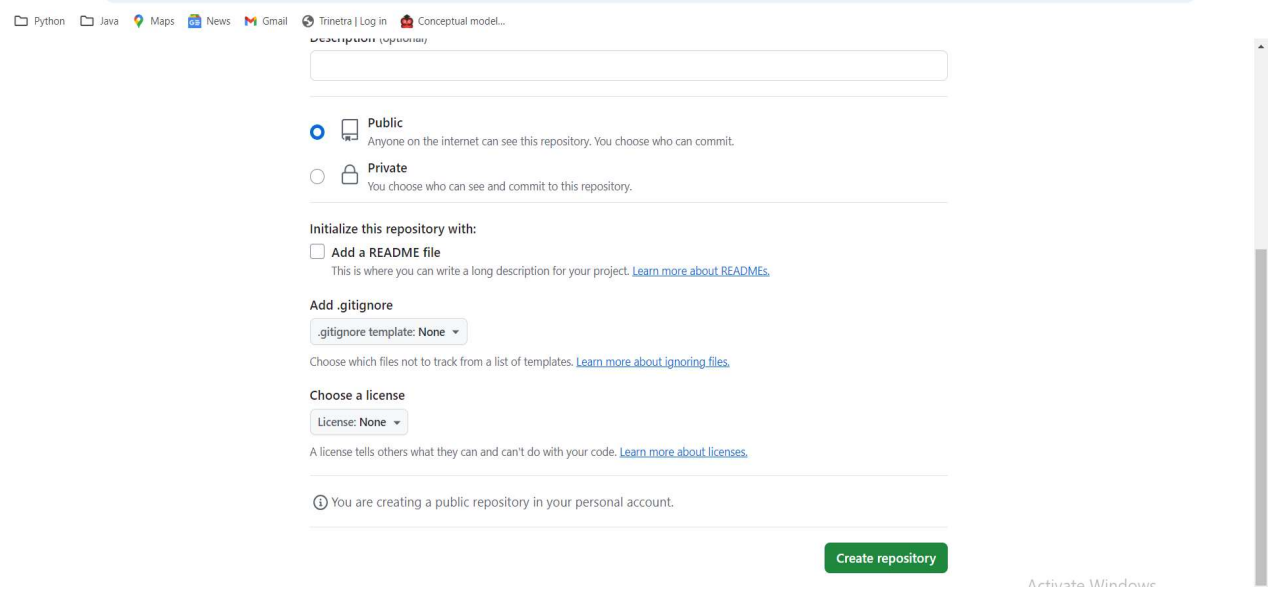
  ```
  git remote add origin https://github.com/user/repo.git
  ```

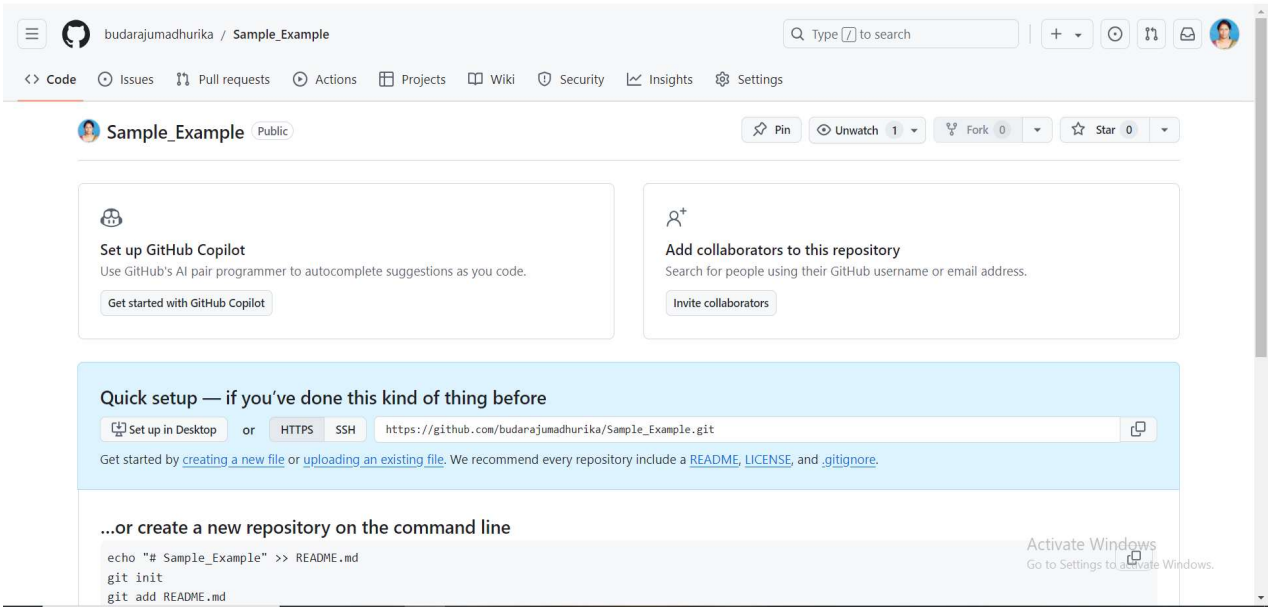
  This adds a new connection to a remote repository called `origin`, which is the common default name for remote repositories. After setting up the remote, you can start interacting with it, like pushing or pulling code.

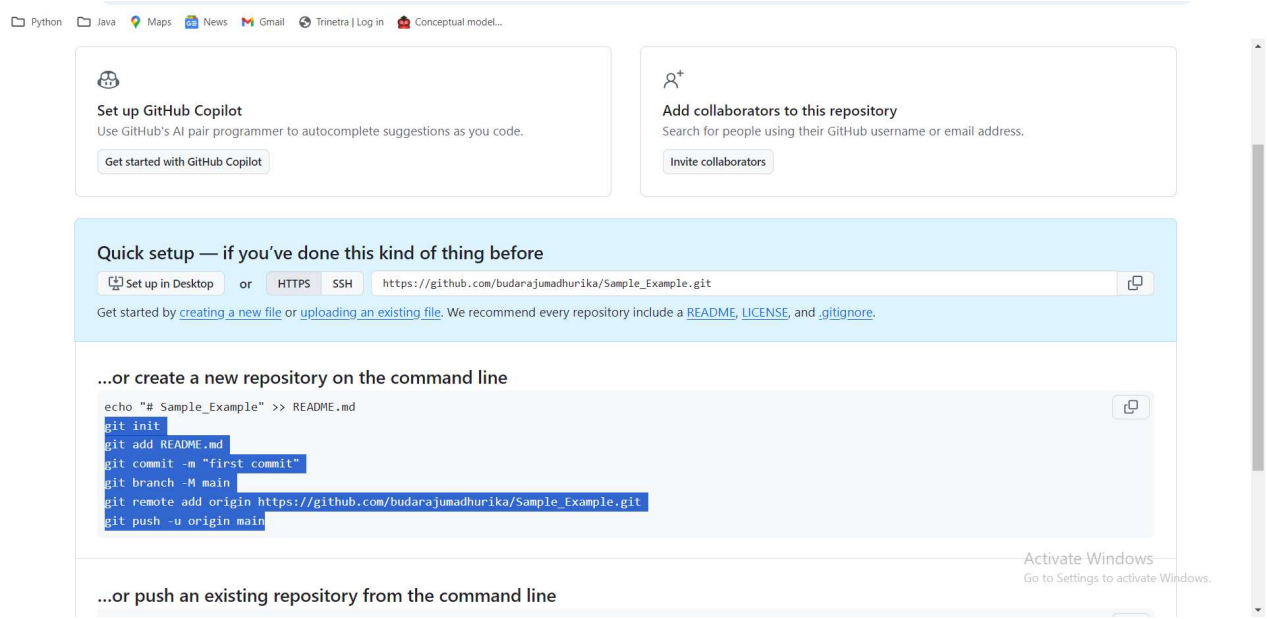
- **Important Commands**:
  - `git remote -v`: Shows the current remotes and their URLs.
  - `git remote add <name> <url>`: Adds a new remote connection.
  - `git remote remove <name>`: Removes an existing remote.

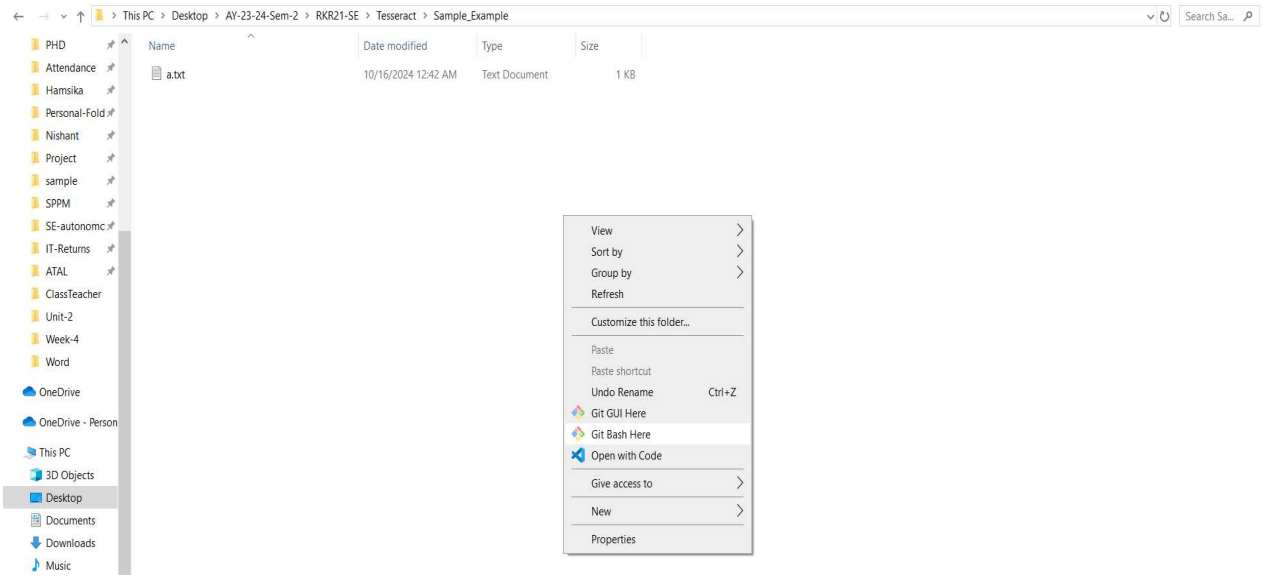

github.com
Python
Java
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Dashboard
Type / to search
Top repositories
New
Find a repository...
budarajumadhurika/samplepollscm
budarajumadhurika/samplescriptjava
budarajumadhurika/SrikantSample
budarajumadhurika/JavaMaven
budarajumadhurika/21-12
budarajumadhurika/11-12-java
budarajumadhurika/WebDeployment
Show more
Home
Send feedback
Filter
Trending repositories · See more
openai/swarm
Star
Educational framework exploring ergonomic, lightweight multi-agent orchestration. Managed by OpenAI Solution team.
Python
12k
SWivid/F5-TTS
Star
Official code for "F5-TTS: A Fairytaler that Fakes Fluent and Faithful Speech with Flow Matching"
Python
3.4k


New repository
Type / to search
Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.
Required fields are marked with an asterisk (*).
Owner *
budarajumadhurika
/
Repository name *
Sample_Example
Sample_Example is available.
Great repository names are short and memorable. Need inspiration? How about verbose-dollop ?
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Add a README file
This is where you can write a long description for your project. Learn more about READMEs.
Activate Windows
Go to Settings to activate Windows.


Python
Java
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Add a README file
This is where you can write a long description for your project. Learn more about READMEs.
Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.
Choose a license
License: None
A license tells others what they can and can't do with your code. Learn more about licenses.
You are creating a public repository in your personal account.
Create repository

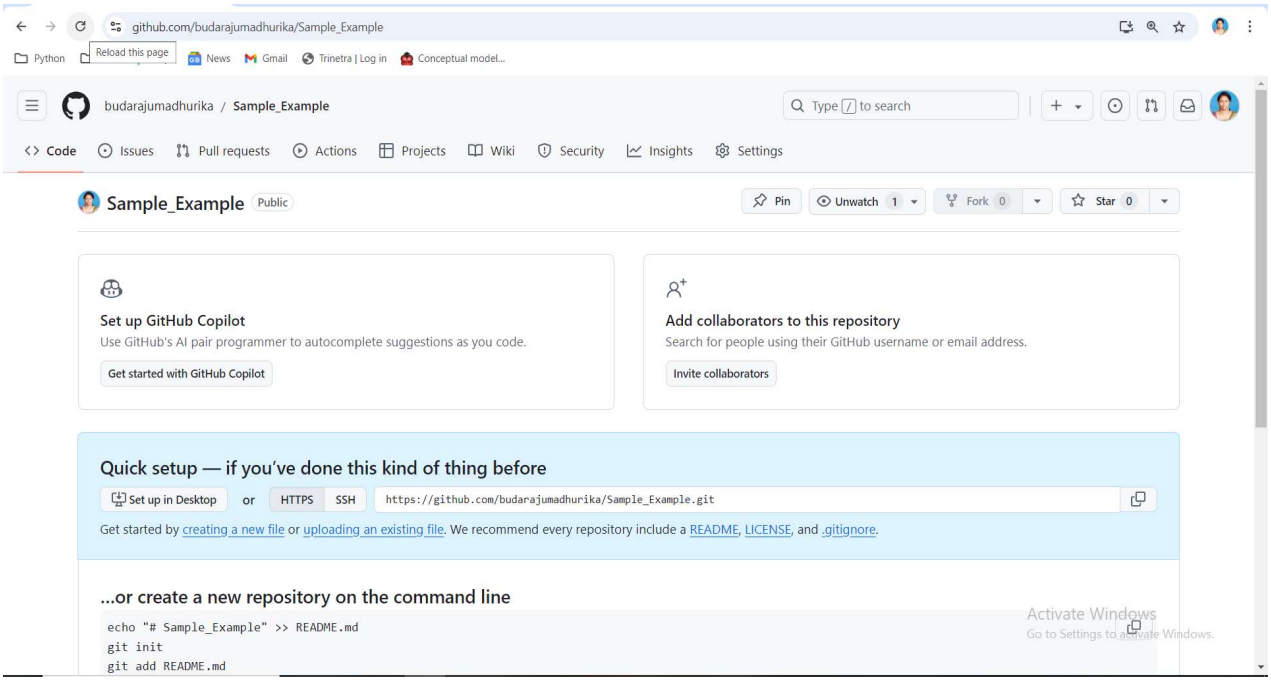

budarajumadhurika / Sample_Example
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Sample_Example
Public
Pin
Unwatch 1
Fork 0
Star 0
Set up GitHub Copilot
Use GitHub's AI pair programmer to autocomplete suggestions as you code.
Get started with GitHub Copilot
Add collaborators to this repository
Search for people using their GitHub username or email address.
Invite collaborators
Quick setup — if you've done this kind of thing before
Set up in Desktop
or
HTTPS
SSH
https://github.com/budarajumadhurika/Sample_Example.git
Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.
...or create a new repository on the command line
echo "# Sample_Example" >> README.md
git init
git add README.md
Activate Windows
Go to Settings to activate Windows.

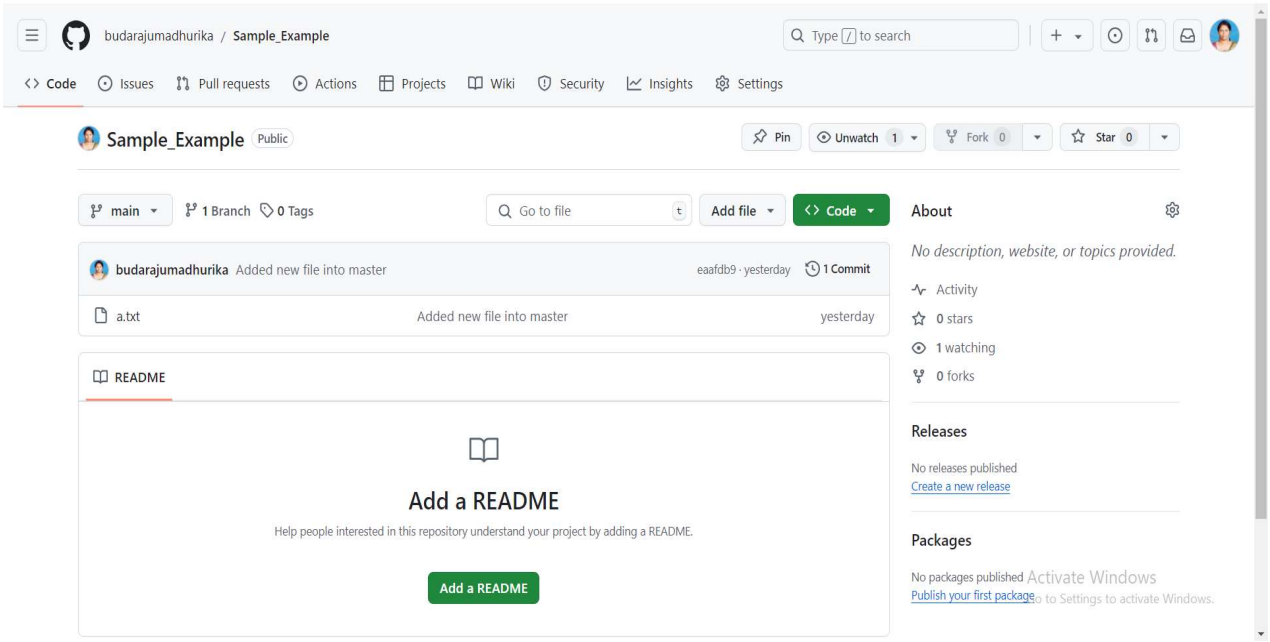





```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Sample_Example (master)
$ git branch -M main

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Sample_Example (main)
$ git remote add origin https://github.com/budarajumadhurika/Sample
_Example.git

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Sample_Example (main)
$ git remote -v
origin  https://github.com/budarajumadhurika/Sample_Example.git (fe
tch)
origin  https://github.com/budarajumadhurika/Sample_Example.git (pu
sh)
```

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Sample_Example (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 248 bytes | 82.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/budarajumadhurika/Sample_Example.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```





2. Connecting Git to GitHub Using SSH Key

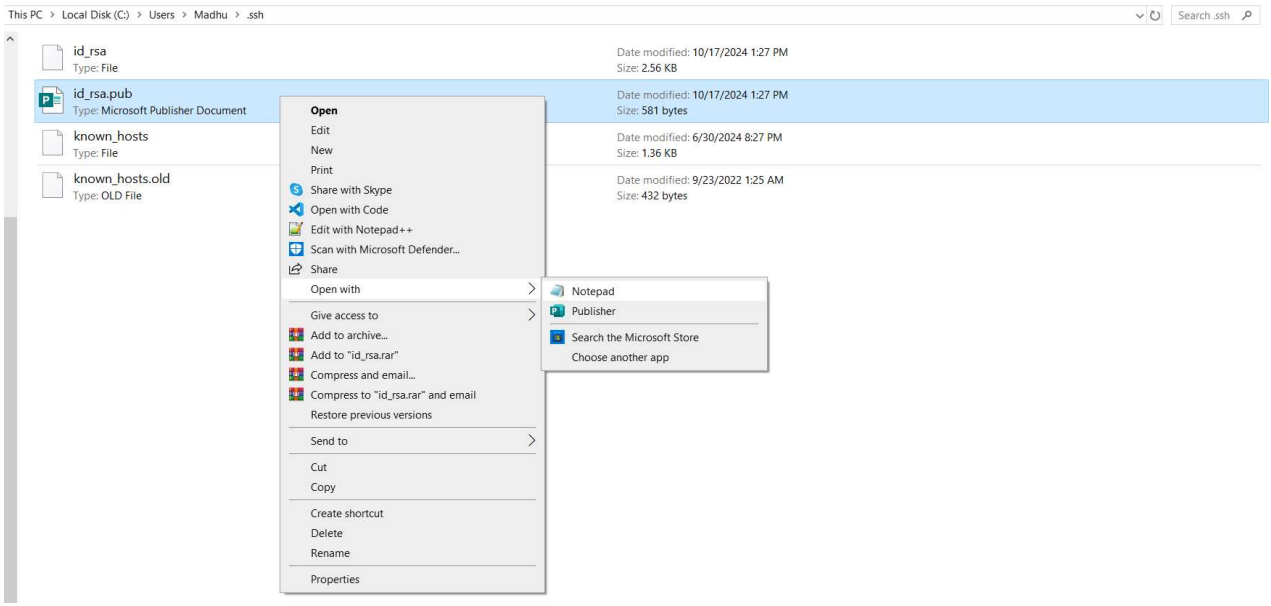
SSH (**Secure Shell**) keys are a way to connect your computer to GitHub without needing to enter a password each time.

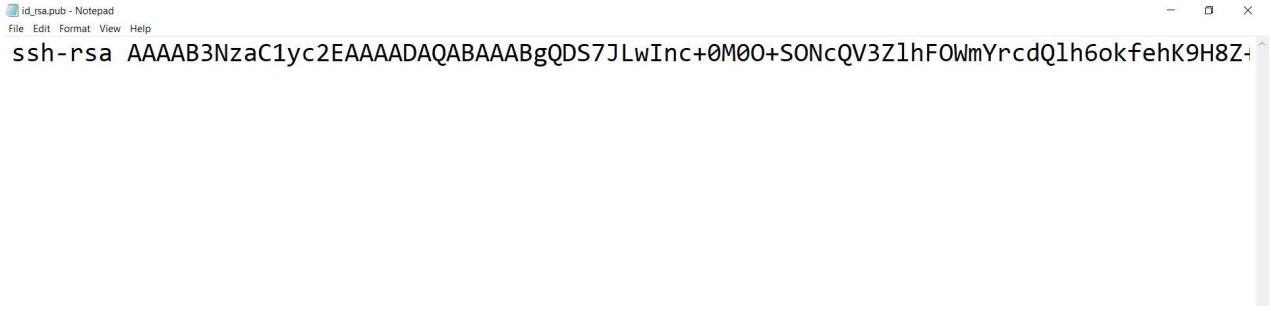
It uses a **key pair**:

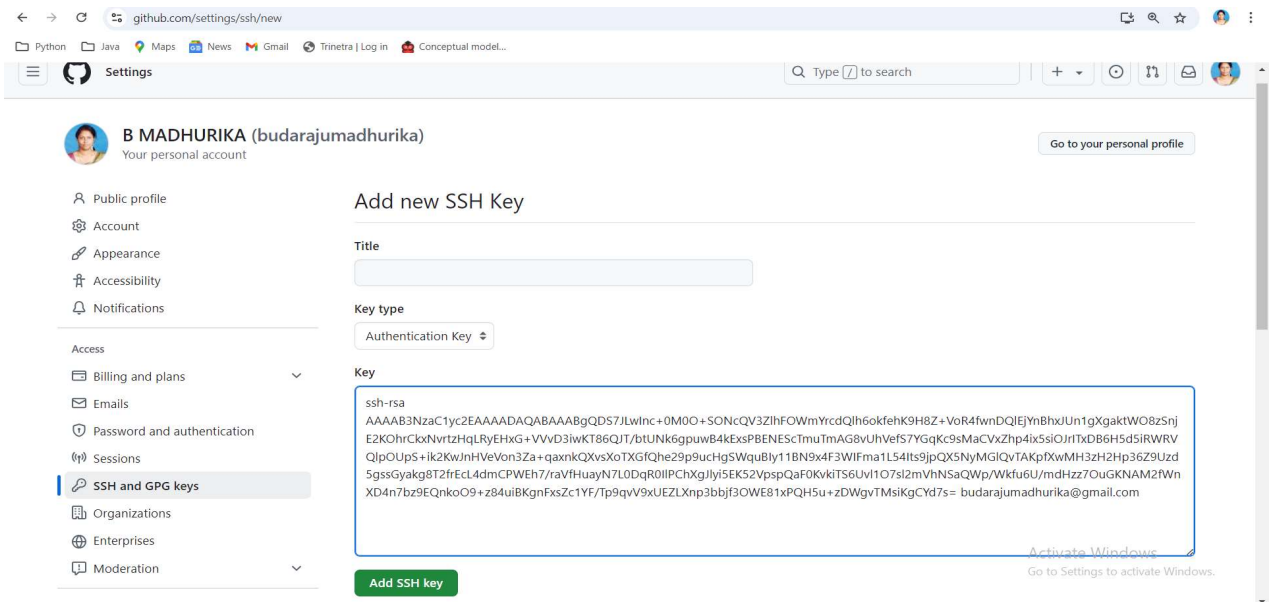
- A **private key** that stays on your computer.
- A **public key** that is shared with GitHub to verify your identity.
- **Steps for Generating SSH Key**:
    1. **Generate SSH Key**: Use the command `ssh-keygen -t rsa` to generate the key on your system.
    2. **Copy the Public Key**: The public key is stored in a file (usually `id_rsa.pub`), and you copy this key.
    3. **Add to GitHub**: In GitHub, go to Settings > SSH and GPG keys > New SSH Key. Paste the copied key into the text box and click "Add SSH key."

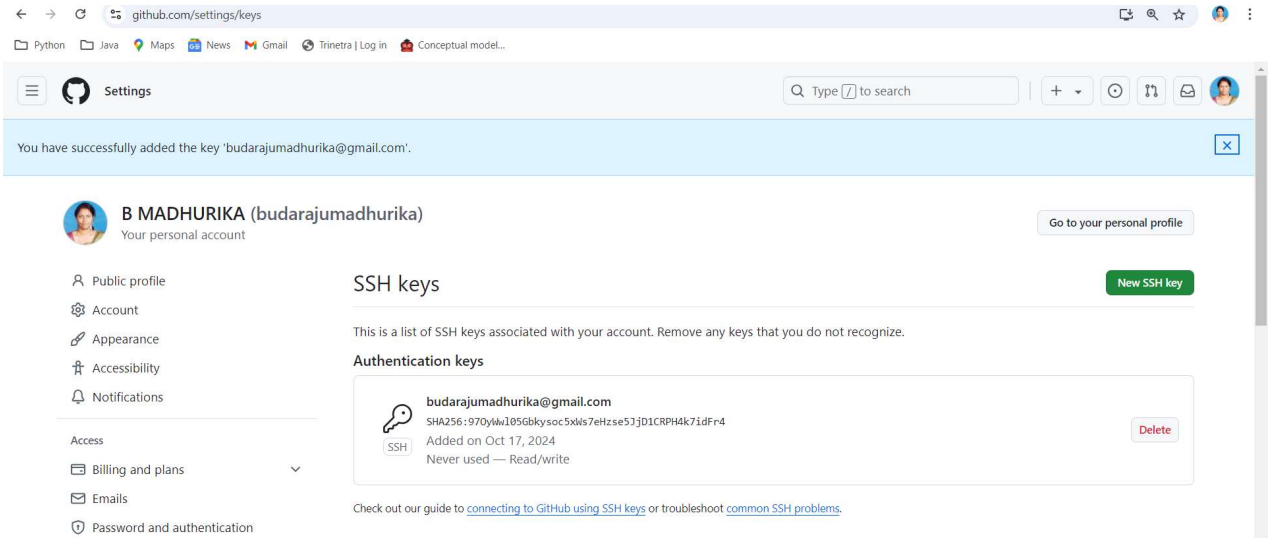
This allows you to securely interact with GitHub without typing your password every time.

```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tes
seract/Example (master)
$ ssh-keygen -t rsa -C budarajumadhurika@gmail.com
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Madhu/.ssh/id_rsa):
/c/Users/Madhu/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Madhu/.ssh/id_rsa
Your public key has been saved in /c/Users/Madhu/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:97OyWwl05Gbkysoc5xWs7eHzse5JjD1CRPH4k7idFr4 budarajumadhurik
a@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|             =.  |
|            B o  |
|           . @ . |
|          o O + .|
|         S B * = |
|        o * B X +|
|         + . % @ |
|           .. O *|
|          o+.oE  |
+----[SHA256]-----+
```


This PC > Local Disk (C:) > Users > Madhu > .ssh
Search .ssh
id_rsa
Type: File
Date modified: 10/17/2024 1:27 PM
Size: 2.56 KB
id_rsa.pub
Type: Microsoft Publisher Document
Date modified: 10/17/2024 1:27 PM
Size: 581 bytes
known_hosts
Type: File
Date modified: 6/30/2024 8:27 PM
Size: 1.36 KB
known_hosts.old
Type: OLD File
Date modified: 9/23/2022 1:25 AM
Size: 432 bytes
Open
Edit
New
Print
Share with Skype
Open with Code
Edit with Notepad++
Scan with Microsoft Defender...
Share
Open with
Give access to
Add to archive...
Add to "id_rsa.rar"
Compress and email...
Compress to "id_rsa.rar" and email
Restore previous versions
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties
Notepad
Publisher
Search the Microsoft Store
Choose another app


id_rsa.pub - Notepad
File Edit Format View Help
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDS7JLwInc+0M0O+SONcQV3ZlhFOWmYrcdQlh6okfehK9H8Z+


github.com/settings/ssh/new
Python Java Maps News Gmail Trinetra | Log in Conceptual model...
Settings
Type / to search
B MADHURIKA (budarajumadhurika)
Your personal account
Go to your personal profile
Public profile
Account
Appearance
Accessibility
Notifications
Access
Billing and plans
Emails
Password and authentication
Sessions
SSH and GPG keys
Organizations
Enterprises
Moderation
Add new SSH Key
Title
Key type
Authentication Key
Key
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQDS7JLwInc+0M0O+SONcQV3ZlhFOWmYrcdQlh6okfehK9H8Z+VoR4fwnDQlEjYnBhxJUn1gXgaktWO8zSnj
E2KOhrCkxNvrtzHqLRyEHxG+VVvD3iwKT86QJT/btUNk6gpuwB4kExsPBENEScTmuTmAG8vUhVefS7YGqKc9sMaCVxZhp4ix5siOJrITxDB6H5d5iRWRV
QlpOUpS+ik2KwJnHVeVon3Za+qaxnkQXvsXoTXGfQhe29p9ucHgSWquBly11BN9x4F3WIFma1L54Its9jpQX5NyMGlQvTAKpfXwMH3zH2Hp36Z9Uzd
5gssGyakg8T2frEcL4dmCPWEh7/raVfHuayN7L0DqR0IlPChXgJlyi5EK52VpspQaF0KvkiTS6Uvl1O7sl2mVhNSaQWp/Wkfu6U/mdHzz7OuGKNAM2fWn
XD4n7bz9EQnkoO9+z84uiBKgnFxsZc1YF/Tp9qvV9xUEZLXnp3bbjf3OWE81xPQH5u+zDWgvTMsiKgCYd7s= budarajumadhurika@gmail.com
Add SSH key
Activate Windows
Go to Settings to activate Windows.



3. Git Clone

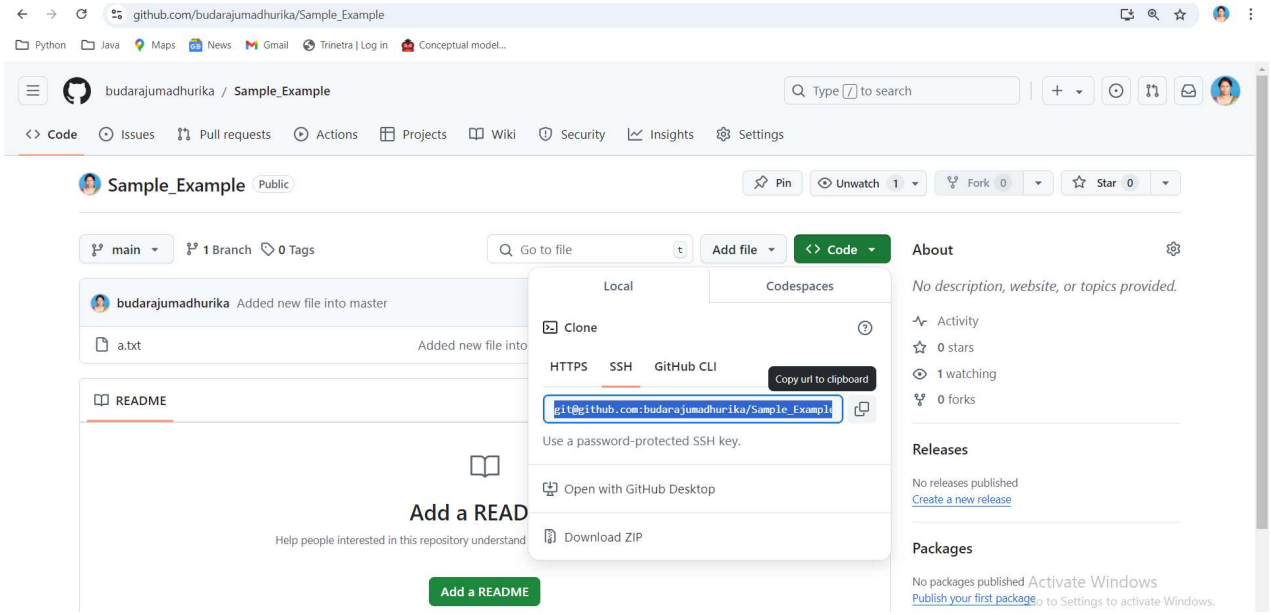
The **git clone** command is used to **download** a copy of a project from a remote repository to your computer. It creates a local version of the project that you can start working on right away.

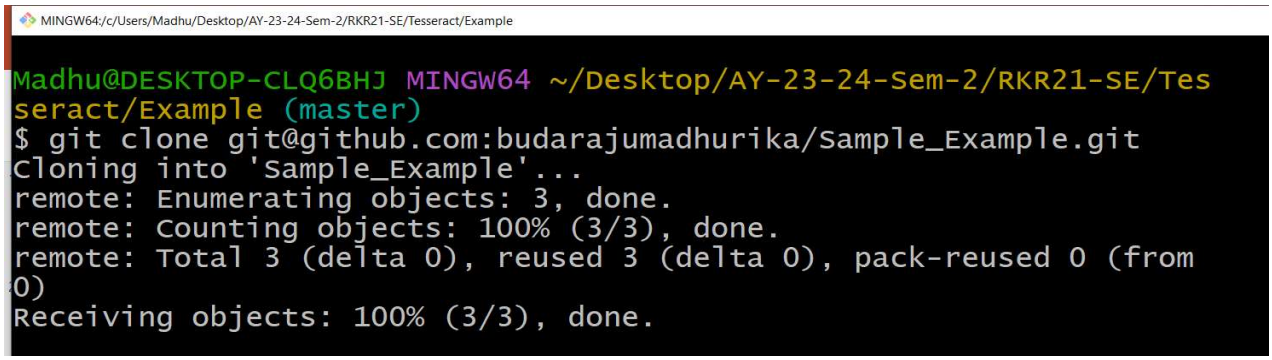
- If you want to work on a project hosted on GitHub, you would use:

  ```
  git clone https://github.com/user/repo.git
  ```

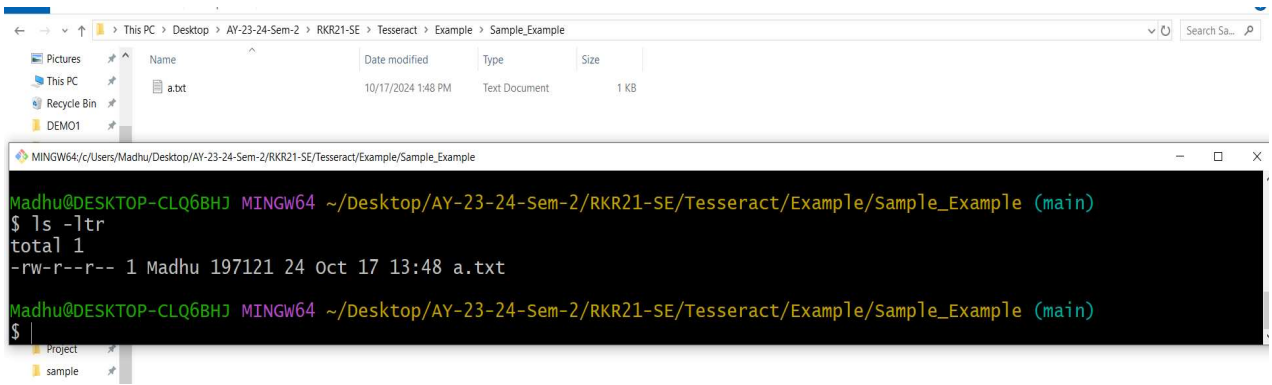
  This creates a local copy of the project so you can begin editing the code on your computer.

- **Other Commands**:
  - `ls -ltr`: Lists the files in the current directory, showing the latest modified files after the cloning is done.


github.com/budarajumadhurika/Sample_Example
budarajumadhurika / Sample_Example
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Sample_Example
Public
Pin
Unwatch 1
Fork 0
Star 0
main
1 Branch
0 Tags
Go to file
Add file
Code
budarajumadhurika Added new file into master
a.txt
README
Local
Codespaces
Clone
HTTPS
SSH
GitHub CLI
Copy url to clipboard
git@github.com:budarajumadhurika/Sample_Exampl
Use a password-protected SSH key.
Open with GitHub Desktop
Download ZIP
Add a README
About
No description, website, or topics provided.
Activity
0 stars
1 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package


MINGW64:/c/Users/Madhu/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example (master)
$ git clone git@github.com:budarajumadhurika/Sample_Example.git
Cloning into 'Sample_Example'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.


Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example (master)
$ ls -ltr
total 0
drwxr-xr-x 1 Madhu 197121 0 Oct 17 13:48 Sample_Example/
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example (master)
$ cd Sample_Example
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$


This PC > Desktop > AY-23-24-Sem-2 > RKR21-SE > Tesseract > Example > Sample_Example
Search Sa...
Pictures
This PC
Recycle Bin
DEMO1
Name
Date modified
Type
Size
a.txt
10/17/2024 1:48 PM
Text Document
1 KB
MINGW64:/c/Users/Madhu/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$ ls -ltr
total 1
-rw-r--r-- 1 Madhu 197121 24 Oct 17 13:48 a.txt
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$
Project
sample

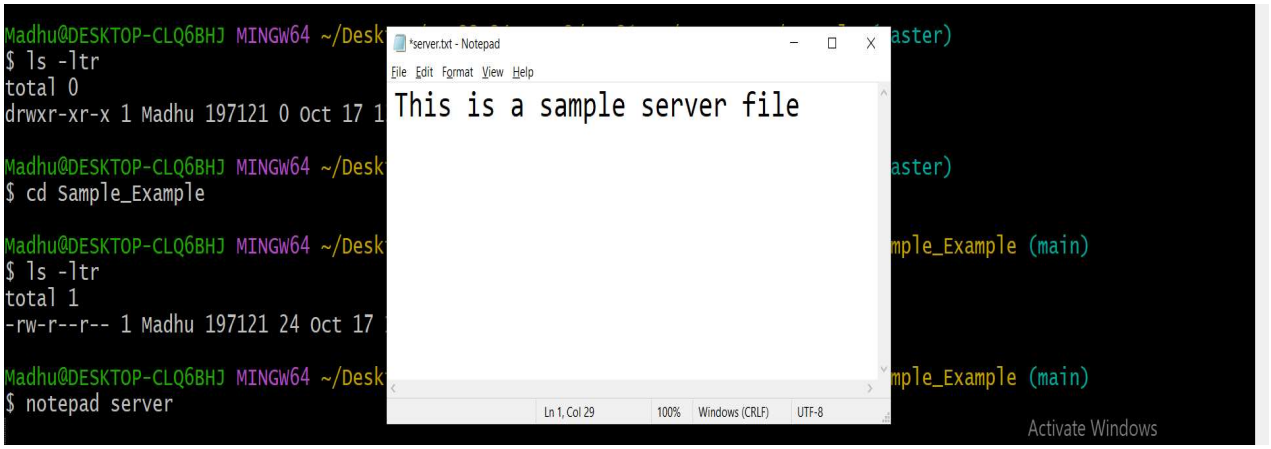
4. Git Push

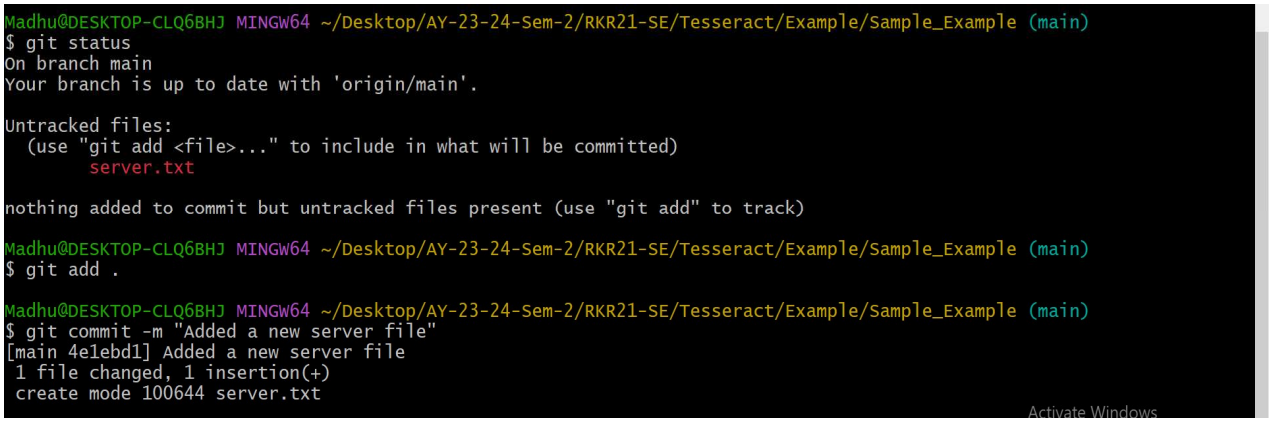
After making changes locally, you use the **git push** command to send these changes from your local repository to the **remote repository** (for example, back to GitHub). This is how you **share your work** with others or keep your remote repository updated with your changes.

- After editing files on your computer and committing the changes, you would push them to the remote repository using:

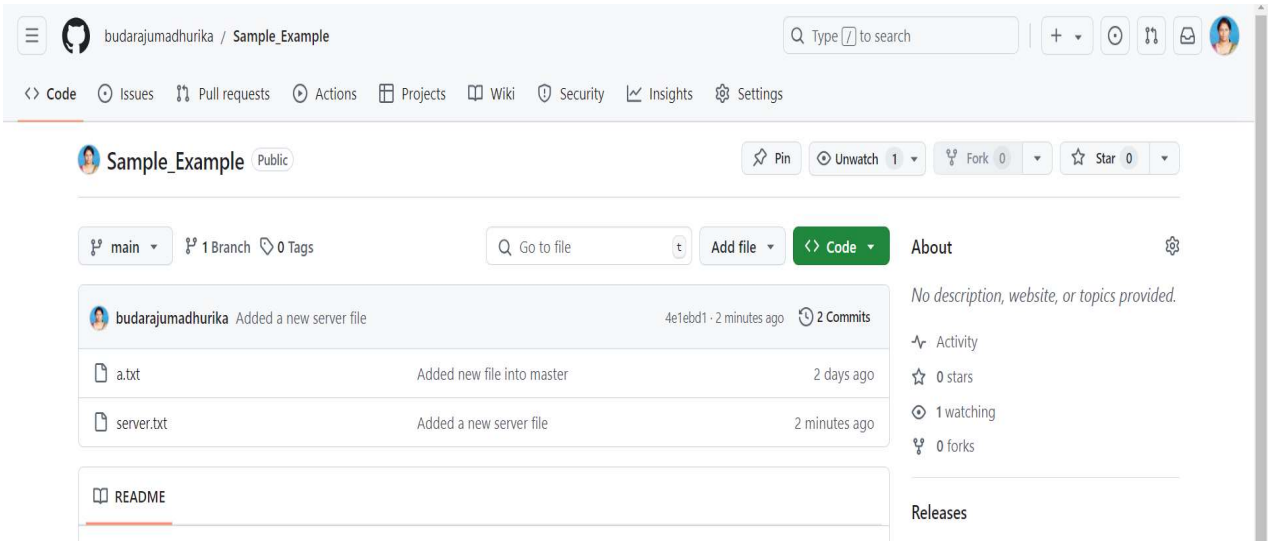
  ```
  git push origin master
  ```

  This pushes the changes from your local `master` branch to the `origin` remote (which is the GitHub repository).





```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$ git remote -v
origin  git@github.com:budarajumadhurika/Sample_Example.git (fetch)
origin  git@github.com:budarajumadhurika/Sample_Example.git (push)

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:budarajumadhurika/Sample_Example.git
   eaafdb9..4e1ebd1  main -> main

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Example/Sample_Example (main)
$
```

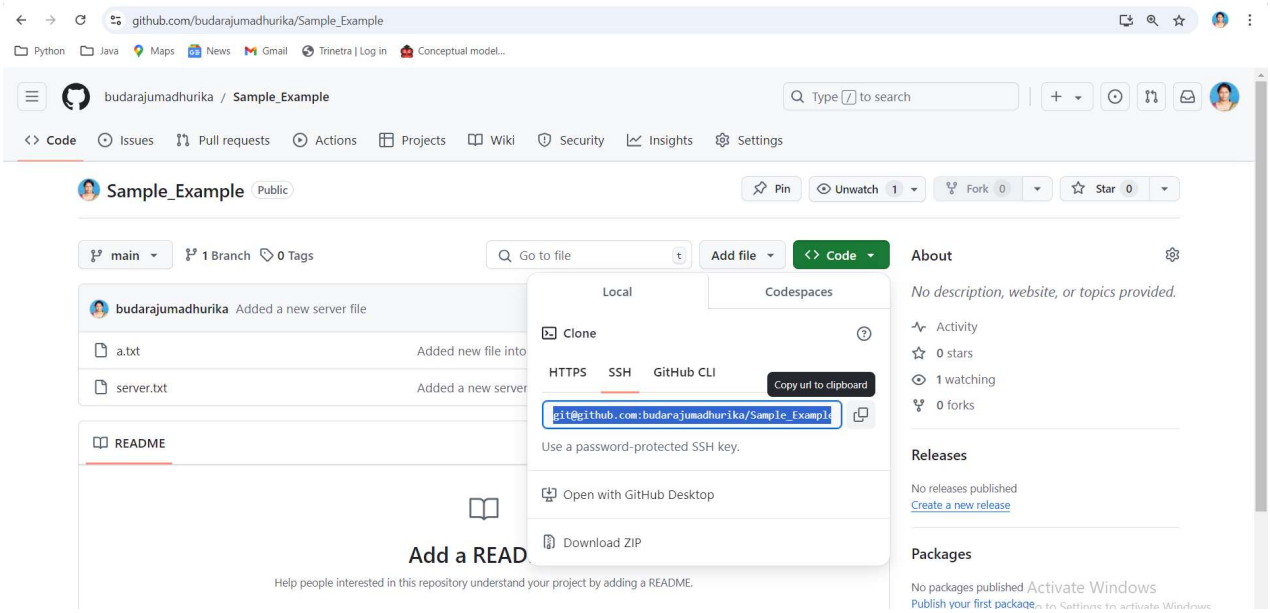
5. Git Pull

The **git pull** command fetches changes from the remote repository and merges them into your local repository. This ensures that you have the latest version of the project, including updates made by your teammates or collaborators.

- If you're working on a project with others and want to get the latest updates they made on GitHub, you would use:

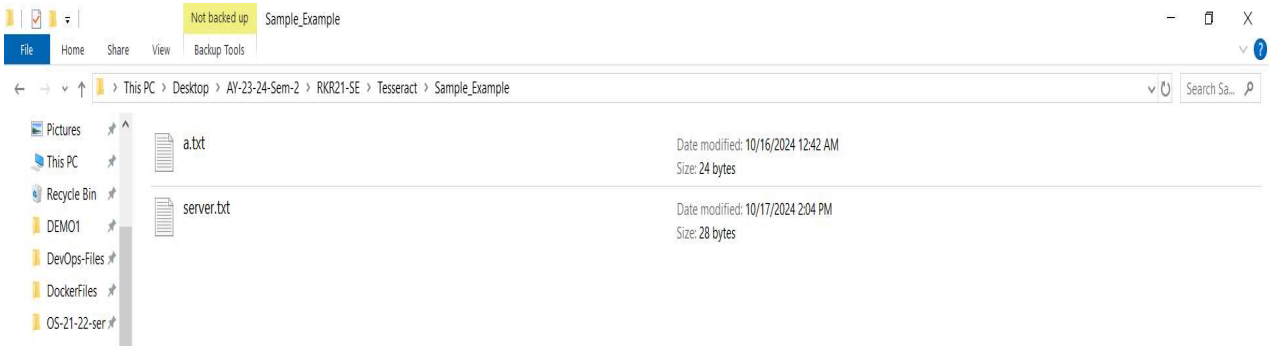
  `git pull origin master`

  This will download any new commits from the `master` branch of the remote repository (`origin`) and merge them with your local code.




```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (main)
$ git remote -v
origin  https://github.com/budarajumadhurika/Sample_Example.git (fetch)
origin  https://github.com/budarajumadhurika/Sample_Example.git (push)

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (main)
$ ls -ltr
total 1
-rw-r--r-- 1 Madhu 197121 24 Oct 16 00:42 a.txt
```

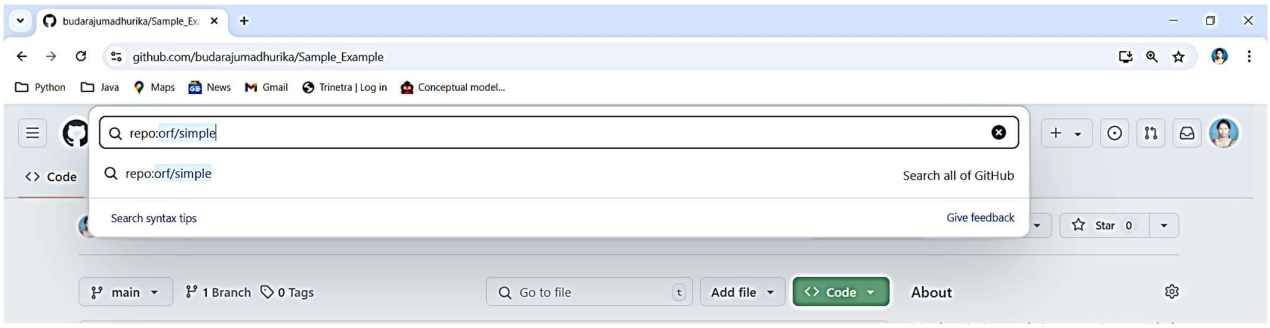
```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (main)
$ git pull origin main
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 294 bytes | 21.00 KiB/s, done.
From https://github.com/budarajumadhurika/Sample_Example
 * branch            main       -> FETCH_HEAD
   eaafdb9..4e1ebd1  main       -> origin/main
Updating eaafdb9..4e1ebd1
Fast-forward
 server.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 server.txt

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/AY-23-24-Sem-2/RKR21-SE/Tesseract/Sample_Example (main
)
$ ls -ltr
total 2
-rw-r--r-- 1 Madhu 197121 24 Oct 16 00:42 a.txt
-rw-r--r-- 1 Madhu 197121 28 Oct 17 14:04 server.txt
```

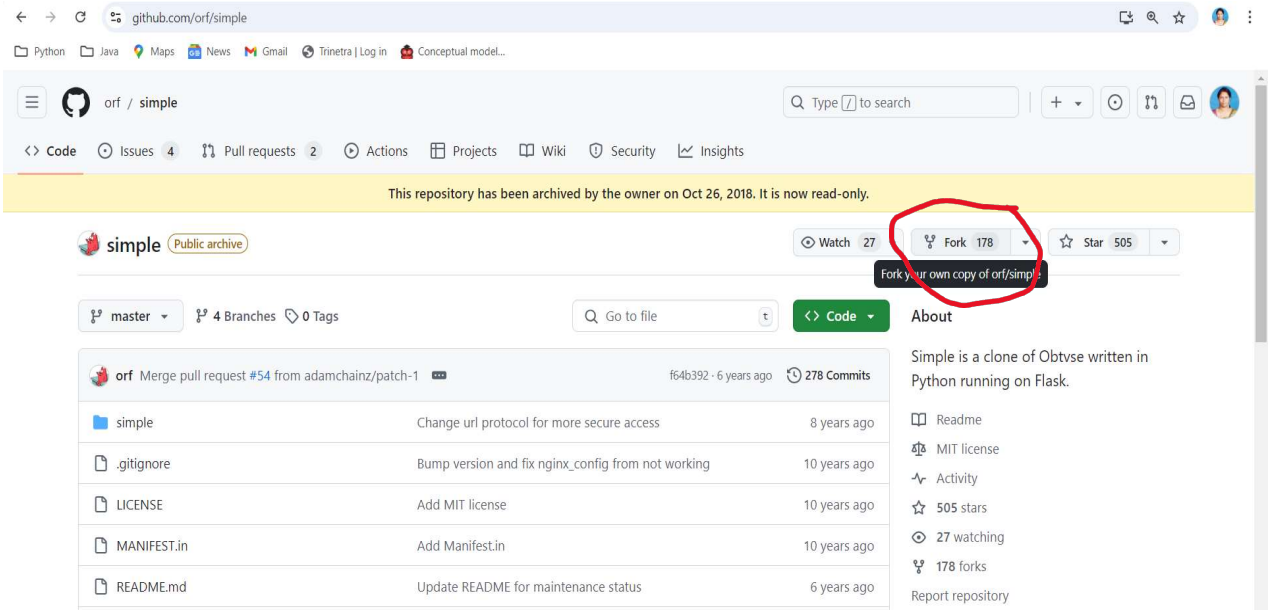


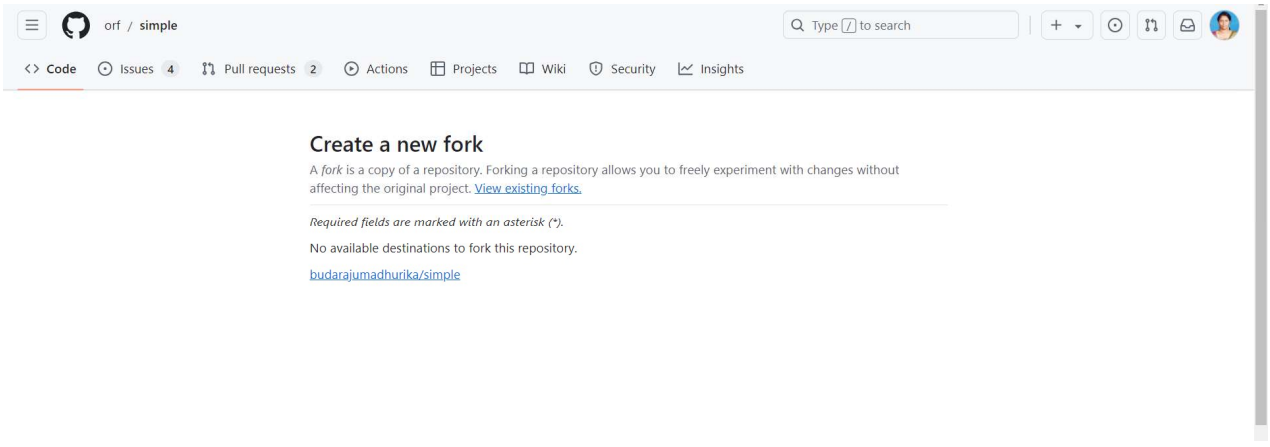
6. Git Fork

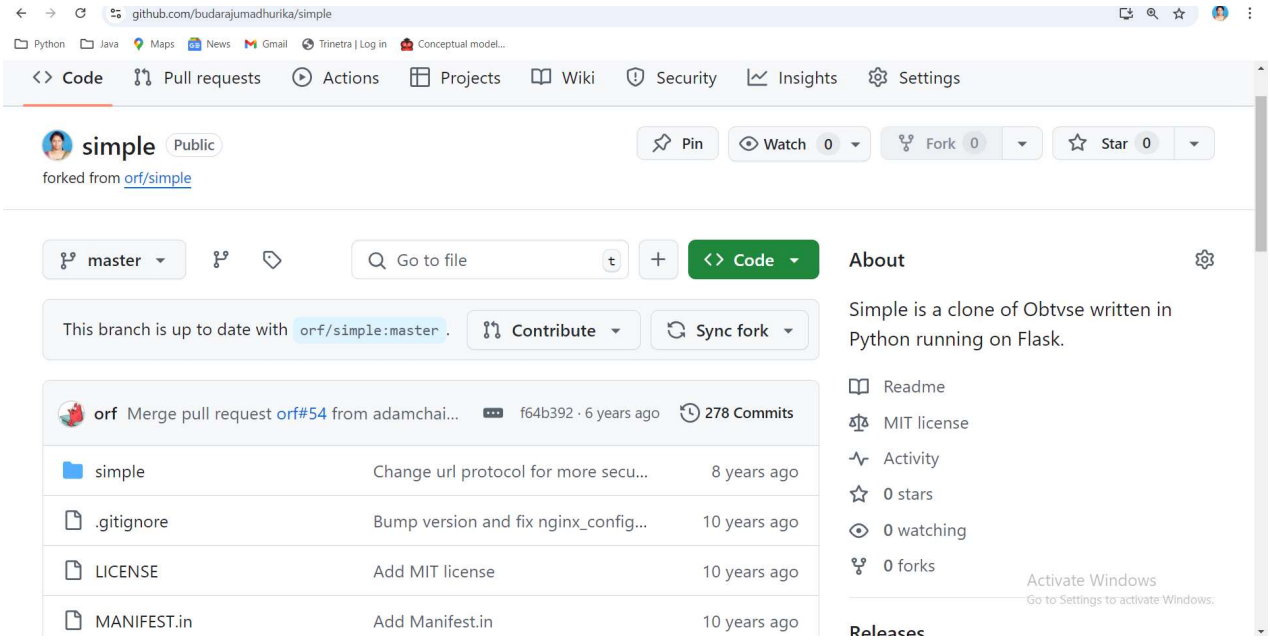
A **fork** is a personal copy of someone else's repository, typically used in open-source projects. When you **fork** a repository, you get your own version of the project on GitHub that you can edit without affecting the original project. You can then propose changes to the original project by making a **pull request**.

- To fork a repository, go to the original repository on GitHub and click the **"Fork"** button. This makes a copy of the repository under your GitHub account where you can make your own changes.




github.com/orf/simple
orf / simple
Code Issues 4 Pull requests 2 Actions Projects Wiki Security Insights
This repository has been archived by the owner on Oct 26, 2018. It is now read-only.
simple Public archive
Watch 27 Fork 178 Star 505
master 4 Branches 0 Tags Go to file Code
About
Simple is a clone of Obtvse written in Python running on Flask.
orf Merge pull request #54 from adamchainz/patch-1 f64b392 · 6 years ago 278 Commits
simple Change url protocol for more secure access 8 years ago
.gitignore Bump version and fix nginx_config from not working 10 years ago
LICENSE Add MIT license 10 years ago
MANIFEST.in Add Manifest.in 10 years ago
README.md Update README for maintenance status 6 years ago
Readme
MIT license
Activity
505 stars
27 watching
178 forks
Report repository


orf / simple
Code Issues 4 Pull requests 2 Actions Projects Wiki Security Insights
Create a new fork
A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. View existing forks.
Required fields are marked with an asterisk (*).
No available destinations to fork this repository.
budarajumadhurika/simple


github.com/budarajumadhurika/simple
Code Pull requests Actions Projects Wiki Security Insights Settings
simple Public
forked from orf/simple
Pin Watch 0 Fork 0 Star 0
master Go to file Code
About
This branch is up to date with orf/simple:master. Contribute Sync fork
Simple is a clone of Obtvse written in Python running on Flask.
orf Merge pull request orf#54 from adamchai... f64b392 · 6 years ago 278 Commits
simple Change url protocol for more secu... 8 years ago
.gitignore Bump version and fix nginx_config... 10 years ago
LICENSE Add MIT license 10 years ago
MANIFEST.in Add Manifest.in 10 years ago
Readme
MIT license
Activity
0 stars
0 watching
0 forks
Activate Windows
Go to Settings to activate Windows.
Releases

Detailed Flow with Examples

1. **Clone a Repository**: If you want to contribute to an open-source project or collaborate with others, you first clone the repository to your computer:

   ```
   git clone https://github.com/user/repo.git
   ```

2. **Make Changes Locally**: Once you have the project on your local machine, you can make changes to the files, such as fixing bugs or adding new features. After editing, you commit the changes:

   ```
   git commit -m "Added new feature"
   ```

3. **Push Changes to Remote**: After committing your changes, you push them back to the remote repository (GitHub):

   ```
   git push origin master
   ```

4. **Pull Changes Made by Others**: If someone else makes changes to the project while you're working on it, you can pull those updates to ensure your version is up-to-date:

   ```
   git pull origin master
   ```

5. **Use SSH Keys for Security**: Instead of typing your password every time you push or pull changes, you can set up an SSH key to make this process secure and automatic:

   ```
   ssh-keygen -t rsa # Generate SSH key
   ```

6. **Forking a Repository**: If you want to make your own changes to someone else's project (such as in open-source), you fork the repository first, creating your own copy:
   - Visit the original project on GitHub.
   - Click the **Fork** button.
   - Make your changes in your personal copy, and then propose your updates back to the original repository by creating a **pull request**.