zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzxTopic 1: Github, git and gitbash.

Github:

GitHub is a web-based platform for version control and collaboration that allows developers to host and manage their Git repositories. It provides a centralized location for storing and sharing code, enabling multiple people to work on the same project simultaneously. GitHub offers features such as issue tracking, pull requests, project management tools, and collaboration functionalities, making it popular among developers for open-source projects and team-based software development.

Gitbash:  
  
Git Bash is a command-line interface (CLI) program that provides a Unix-like environment on Windows. It allows you to interact with Git and run Unix/Linux commands. Git Bash provides a way to use Git commands and access the Git repository on Windows without requiring a full Unix-like operating system.

Git:

Git is a distributed version control system that allows developers to track changes in their code over time. It is designed to handle projects of any size with speed and efficiency. With Git, you can create branches to work on different features or bug fixes independently, merge changes from multiple contributors, and easily revert to previous versions of your code if needed. Git also enables collaboration by allowing multiple developers to work on the same project simultaneously and resolving conflicts that may arise when merging changes.

Comands:

1. **git config** is a command in Git used to configure various aspects of Git's behavior. It allows you to set or modify configuration settings that control how Git works on your local machine.

Here are some common **git config** commands:

* **git config --global user.name "Your Name"**: Sets your name that will be associated with your Git commits globally.
* **git config --global user.email "youremail@example.com"**: Sets your email address associated with your Git commits globally.
* **git config --global core.editor "editor"**: Specifies the default text editor to be used for Git operations, where "editor" can be replaced with the name of your preferred text editor.
* **git config --global color.ui true**: Enables colored output in Git to improve readability.
* **git config --list**: Displays the current Git configuration settings.

These commands are just a few examples of how you can use **git config** to customize your Git environment.

Top of Form

git config --global user.email [deviswati32@gmail.com](mailto:deviswati32@gmail.com)

1. **git init** is a command in Git used to initialize a new Git repository in a directory. When you run **git init** in a project directory, it sets up a new repository with the necessary data structures and metadata that Git uses to track changes in your code.

Here's how **git init** works:

1. Open your terminal or Git Bash.
2. Navigate to the directory where you want to create the Git repository.
3. Run the command **git init**.

After running **git init**, the directory will become a Git repository, and you can start using Git to track changes, create branches, commit your code, and perform various other Git operations within that repository. Git will create a hidden **.git** directory in the project directory to store all the necessary files and metadata for version control.

Git Status:  
  
git add .

Git status:

Git commit -m “success” .

git remote add origin <https://github.com/swati-thakur/food-ordering-management-system.git>

git config --global user.email [uic.19mca8157@gmail.com](mailto:uic.19mca8157@gmail.com)

git config --global user.name "swati-thakur"

git push -u origin master

git branch -M main

git push -u origin main

git push origin master

code. : open visual studio code.

1. `git init`: Initializes a new Git repository in the current directory, creating a `.git` folder to track changes.

2. `git clone <repository\_url>`: Creates a copy of a remote repository on your local machine.

3. `git add <file(s)>`: Adds file(s) to the staging area, preparing them to be committed.

4. `git commit -m "Commit message"`: Records changes to the repository with a descriptive commit message.

5. `git status`: Shows the current status of the repository, including modified, staged, and untracked files.

6. `git branch`: Lists all branches in the repository and indicates the current branch with an asterisk.

7. `git checkout <branch\_name>`: Switches to the specified branch.

8. `git merge <branch\_name>`: Combines the specified branch's changes into the current branch.

9. `git pull`: Fetches and merges changes from a remote repository into the current branch.

10. `git push`: Pushes local changes to a remote repository.

11. `git log`: Displays the commit history, including commit messages, authors, and timestamps.

12. `git remote add <remote\_name> <repository\_url>`: Adds a remote repository with a given name and URL.

13. `git remote -v`: Lists all remote repositories associated with the current repository.

14. `git diff`: Shows the differences between the working directory and the staging area.

15. `git stash`: Temporarily saves changes that are not ready to be committed, allowing you to switch branches or perform other operations.

These are just a few examples of the many Git commands available. You can explore more Git commands and their options by referring to Git's documentation or using the `git --help` command for a list of available commands and options.

1. Untracked: Untracked files are files that Git is not currently aware of. These files have not been added to the repository or the staging area. They can be new files or files that were previously tracked but have been deleted.
2. Unmodified: Unmodified files are files that have been previously committed and have not been changed since the last commit. Git recognizes these files as being in a stable state.
3. Modified: Modified files are files that have been changed since the last commit. Git detects the differences between the current file and the committed version. These changes are not yet staged for the next commit.
4. Staged: Staged files are modified files that have been added to the staging area using the **git add** command. Staging files means you have explicitly marked them for inclusion in the next commit. These changes will be part of the next commit snapshot.

**git commit** is a command used to record changes to the repository. It creates a new commit, which represents a snapshot of the current state of your project, including any changes you have staged (using **git add**) since the last commit.

Here's the basic workflow for creating a commit:

1. Make changes to your project files.
2. Use **git add** to stage the changes you want to include in the commit.
3. Run **git commit -m "Commit message"** to create a new commit with a descriptive message.

The commit message is essential as it helps others understand the purpose or context of the changes made in the commit.

Additionally, you can use various options with the **git commit** command to customize your commits, such as:

* **git commit -a**: Automatically stages all modified files before creating the commit.
* **git commit --amend**: Modifies the last commit by adding or changing files or updating the commit message.
* **git commit -m "Commit message" --author="Author Name <email@example.com>"**: Sets the author information for the commit.

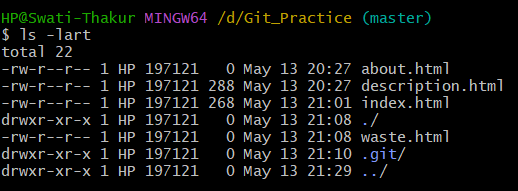
Remember that commits create a permanent record of changes in your Git history. It's good practice to create meaningful and granular commits, focusing on specific changes or features, and to keep the commit history clean and organized.

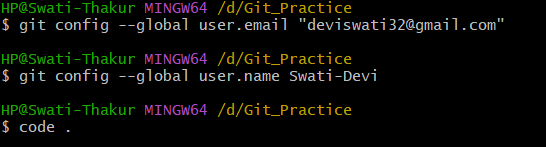
Vim is a powerful and popular text editor that is available on most Unix-based systems, including Linux and macOS. It is known for its efficiency, extensibility, and versatility. Vim operates in different modes, such as normal mode, insert mode, and command-line mode, allowing you to perform various editing tasks.

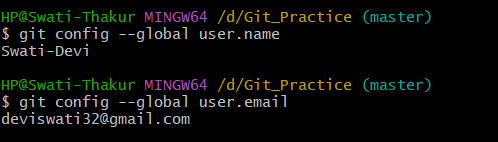
Here are some essential Vim commands to get you started:

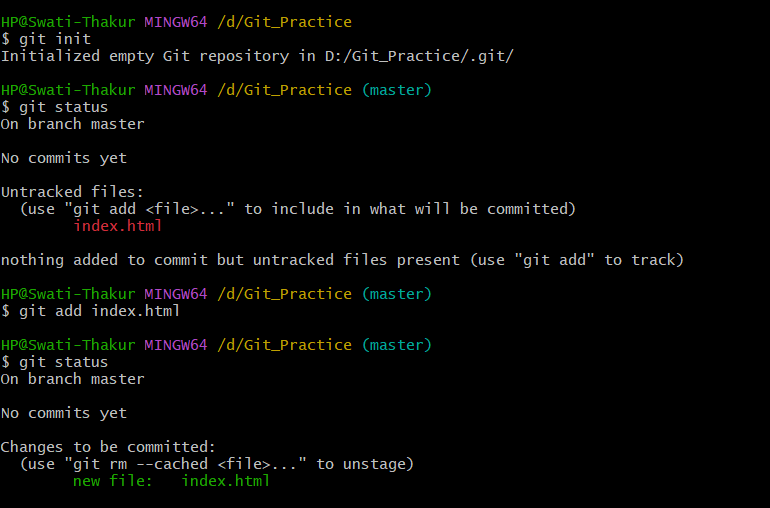
1. Opening a file:
   * **vim filename**: Opens the specified file in Vim.
   * While in Vim, you can use the **:e <filename>** command to open a new file or switch to an existing file.
2. Navigating within Vim:
   * Use the arrow keys to move the cursor.
   * **h**, **j**, **k**, **l**: Move the cursor left, down, up, and right respectively.
   * **gg**: Move the cursor to the beginning of the file.
   * **G**: Move the cursor to the end of the file.
   * **:<line\_number>**: Moves the cursor to the specified line number.
3. Editing and Saving:
   * **i**: Switch to insert mode, allowing you to enter text.
   * **Esc**: Switch back to normal mode from insert mode.
   * **:w**: Save the changes made to the file.
   * **:q**: Quit Vim.
   * **:wq** or **:x**: Save the changes and quit Vim.
4. Deleting and Copying:
   * **x**: Delete the character under the cursor.
   * **dd**: Delete the current line.
   * **yy**: Copy the current line.
   * **p**: Paste the copied or deleted text after the cursor.
5. Searching and Replacing:
   * **/search\_term**: Search forward for the specified search term.
   * **?search\_term**: Search backward for the specified search term.
   * **:s/old/new**: Replace the first occurrence of 'old' with 'new' in the current line.
   * **:s/old/new/g**: Replace all occurrences of 'old' with 'new' in the current line.
   * **:%s/old/new/g**: Replace all occurrences of 'old' with 'new' in the entire file.

These are just a few fundamental Vim commands. Vim has many more features and commands, including advanced text manipulation, split windows, macros, and plugins. It has a steep learning curve but provides a highly customizable and efficient editing experience once mastered.





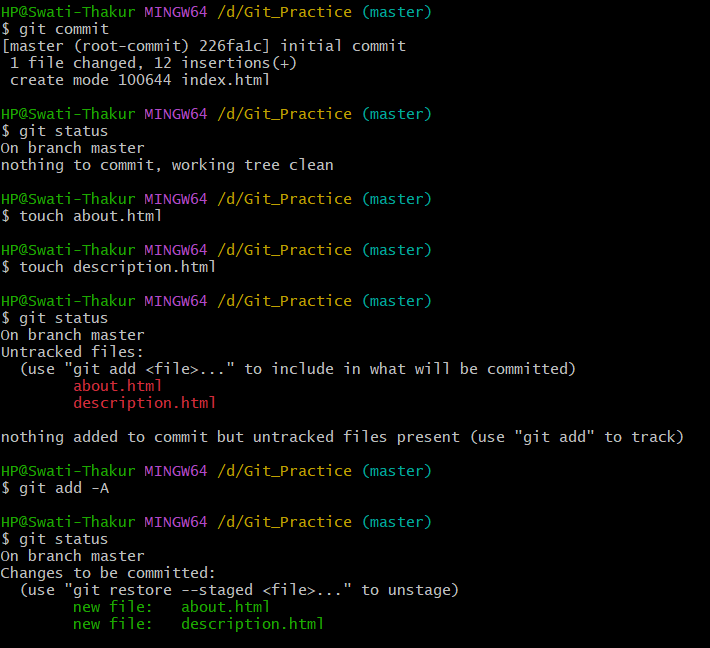


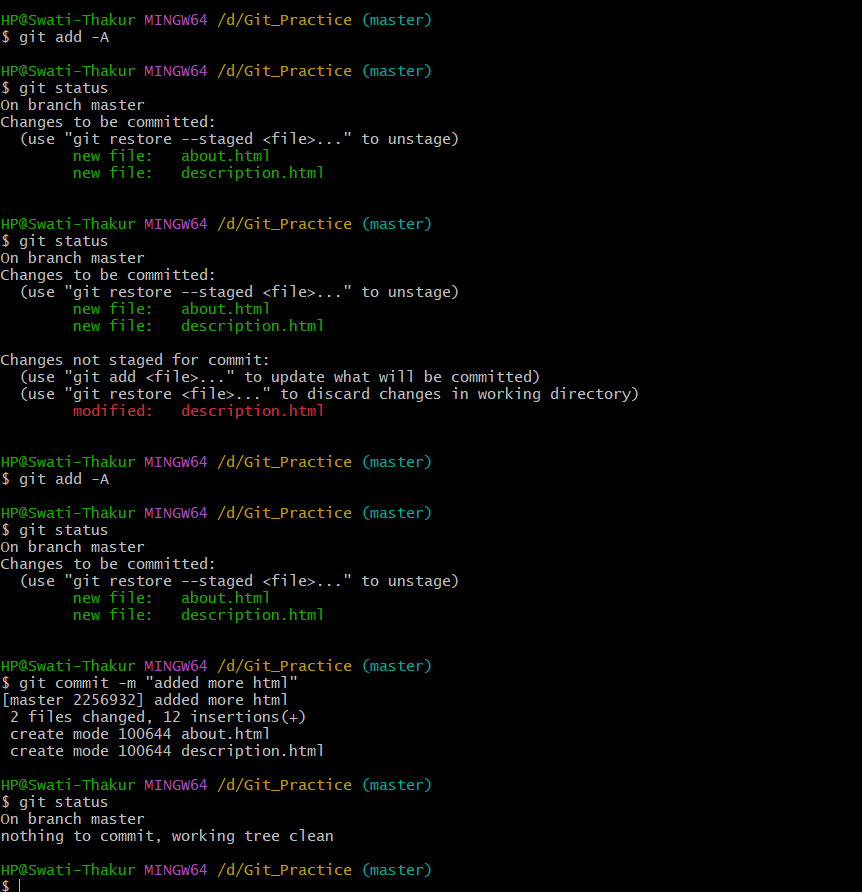


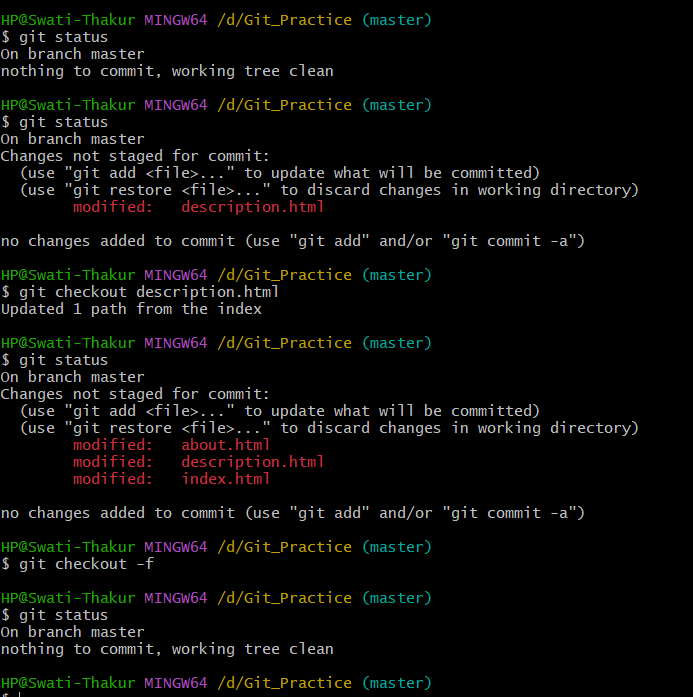


Then vim editor will open and press I to enter then type something then press esc then :wq and enter.





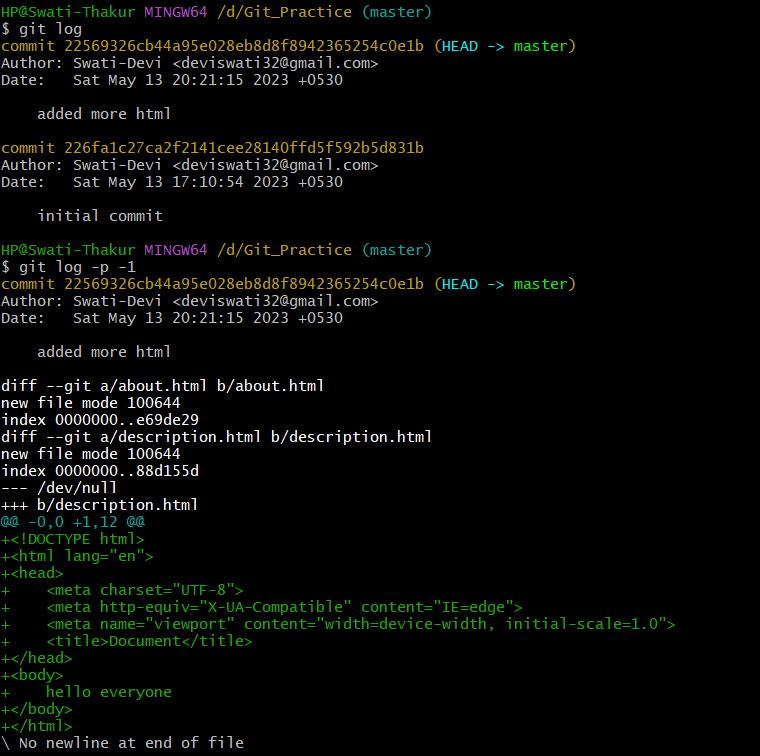




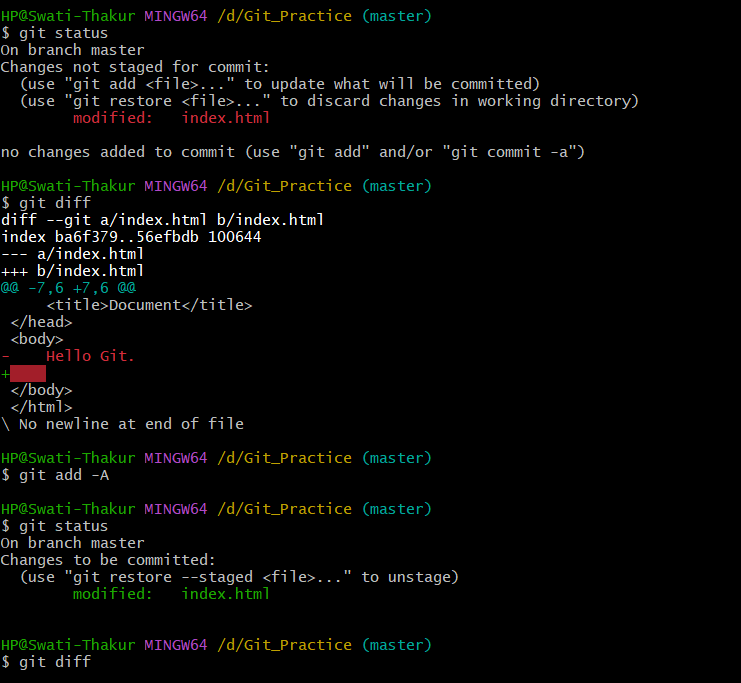
The **git checkout** command is used in Git to switch branches or restore files from previous commits. Here are some common use cases for **git checkout**:

1. Switching branches:
   * **git checkout <branch\_name>**: Switches to the specified branch, updating the working directory to reflect the contents of that branch.
2. Creating and switching to a new branch:
   * **git checkout -b <new\_branch\_name>**: Creates a new branch with the given name and immediately switches to it.
3. Discarding changes to a file:
   * **git checkout -- <file\_path>**: Restores the specified file to the state it was in at the last commit, discarding any local changes.
4. Checking out a specific commit:
   * **git checkout <commit\_hash>**: Detaches the HEAD from the current branch and puts it at the specified commit, allowing you to inspect or make changes at that specific point in history.
5. Checking out a specific file from a different branch or commit:
   * **git checkout <branch\_name> -- <file\_path>**: Retrieves a specific file from another branch and overwrites the corresponding file in the current branch.

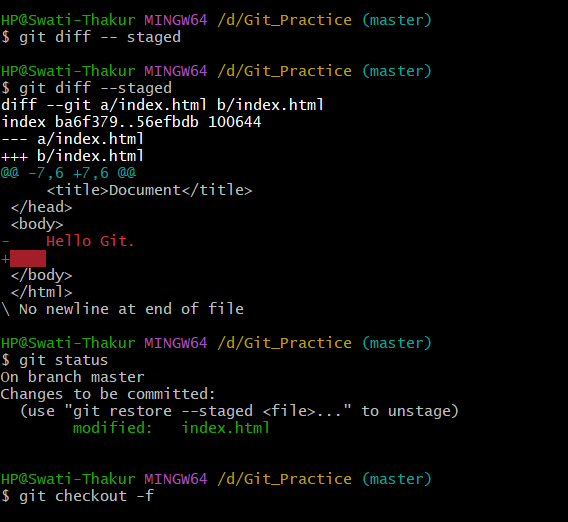
It's important to note that using **git checkout** can discard local changes, so it's recommended to commit or stash your changes before switching branches or restoring files.



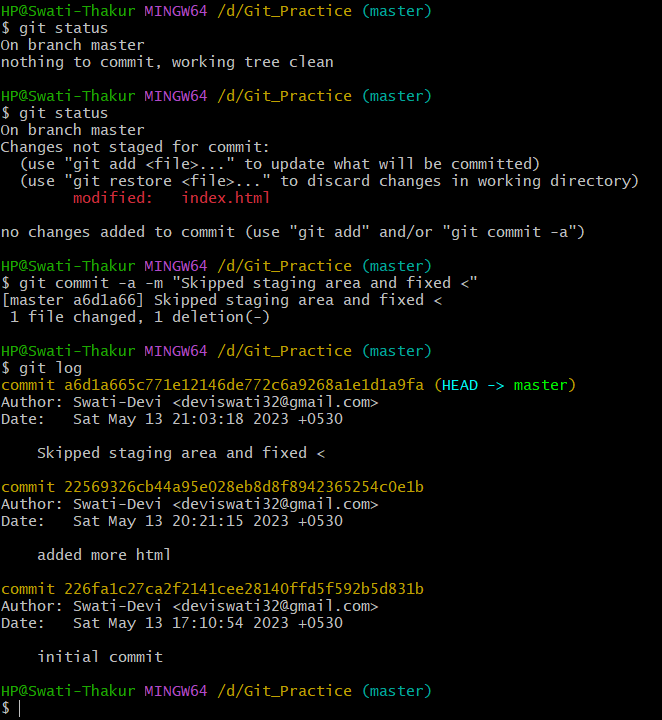
Git diff: is use to compare the difference between working file and the stage area file.

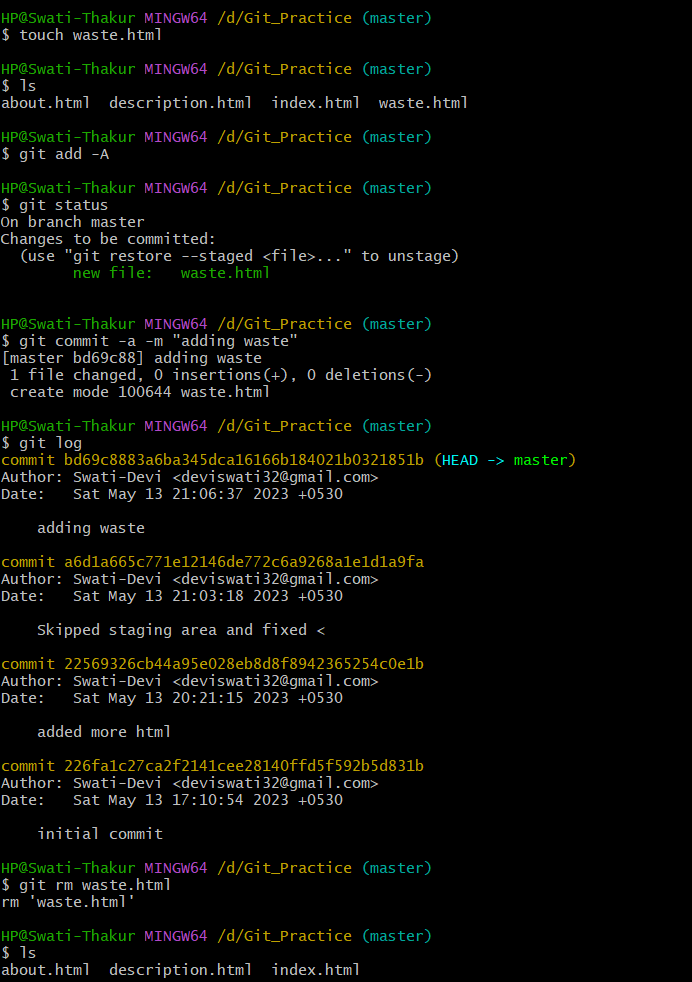


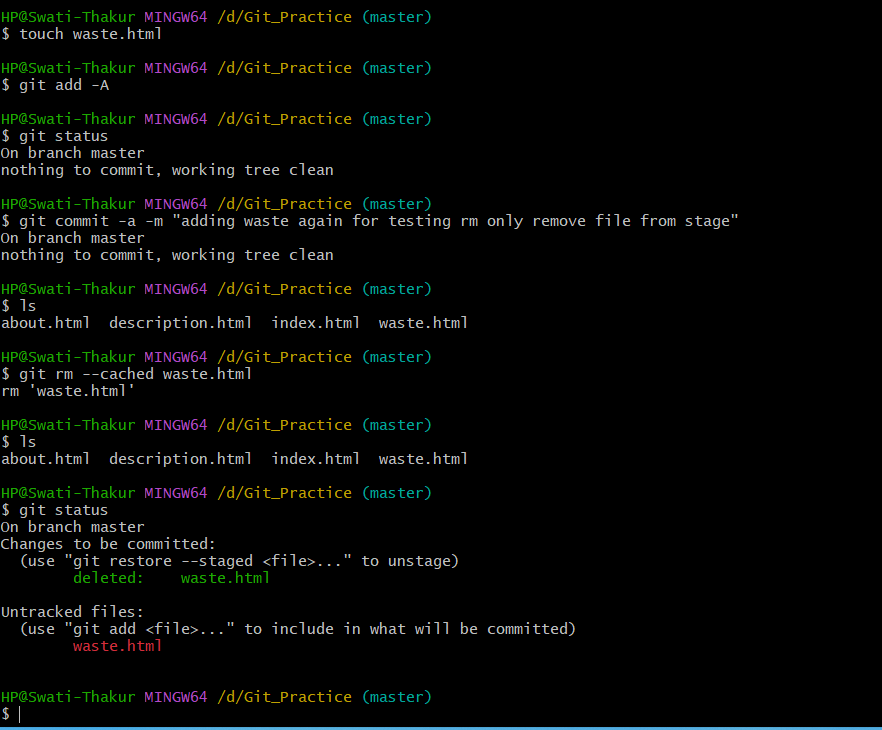
Git diff –staged: is use for comparing difference between staged area file and last commit file.

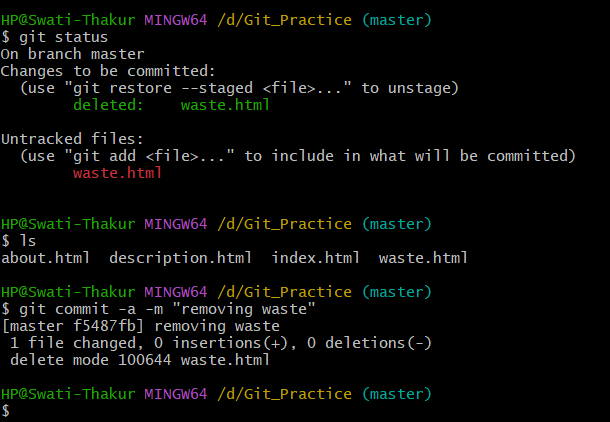


Git commit -a -m: is use to directly commit file with add it to stage area.





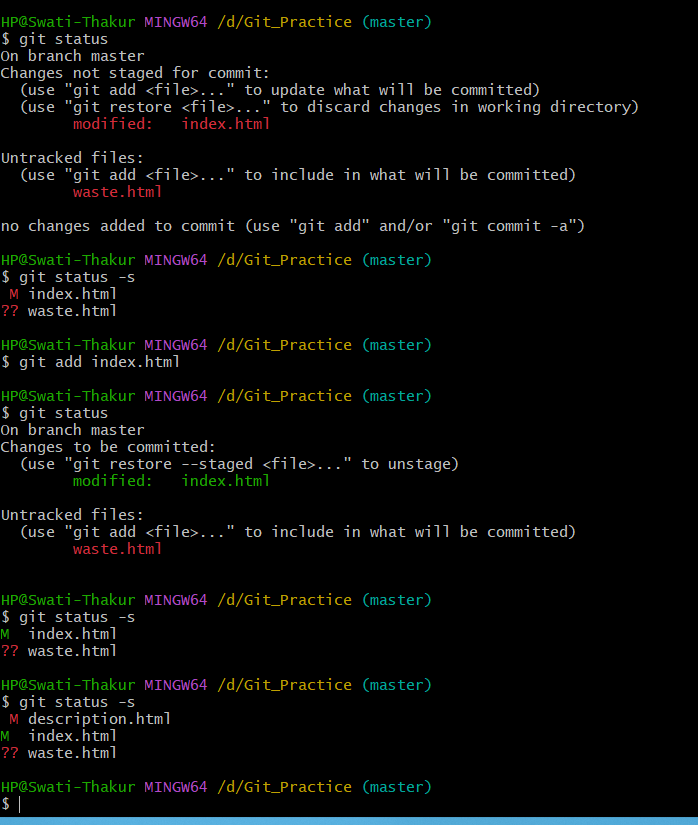


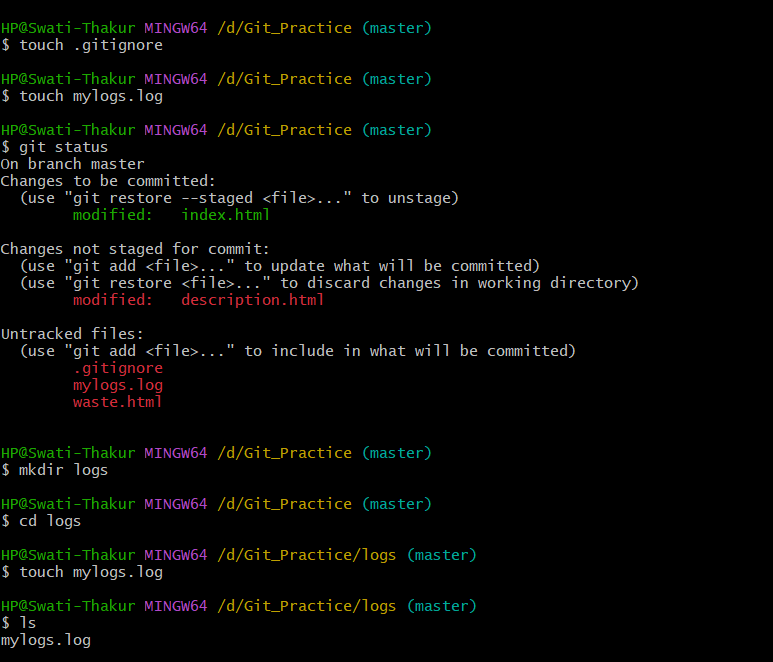


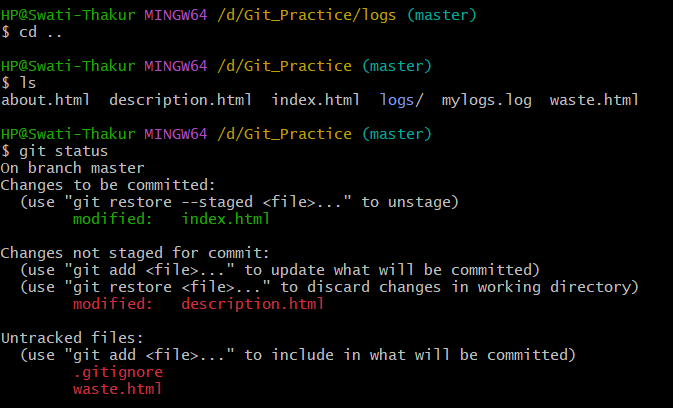
Git status -s:

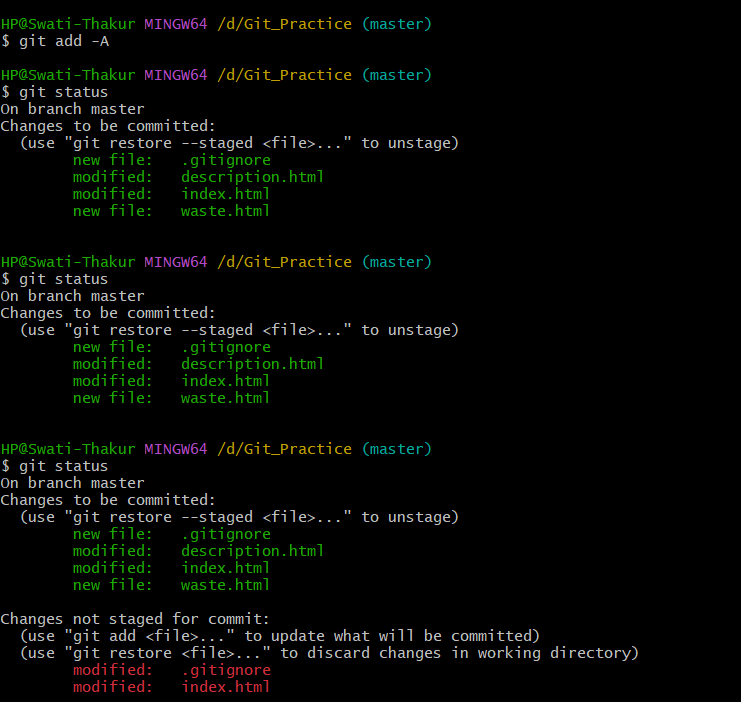
Green M means file is it stage area.

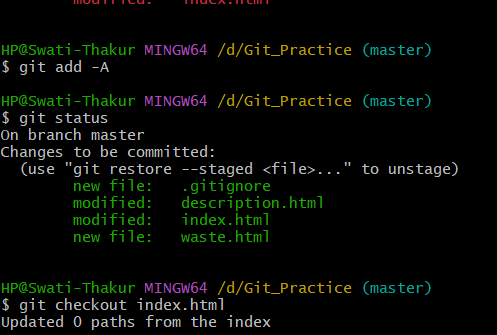
Red M means file in working tree.

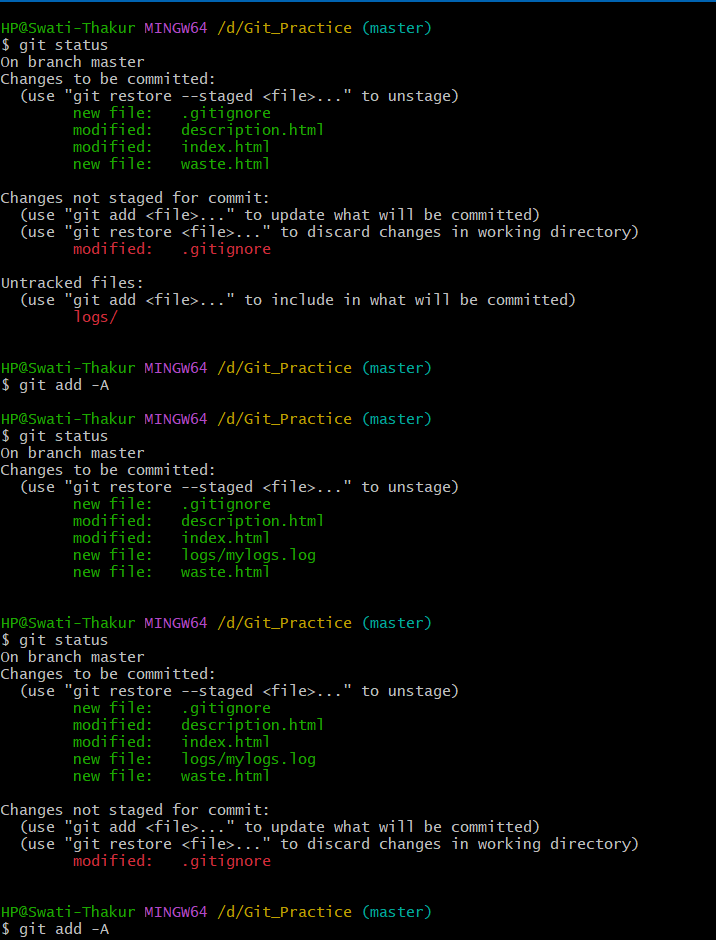
?? file is untracked.  
  
 



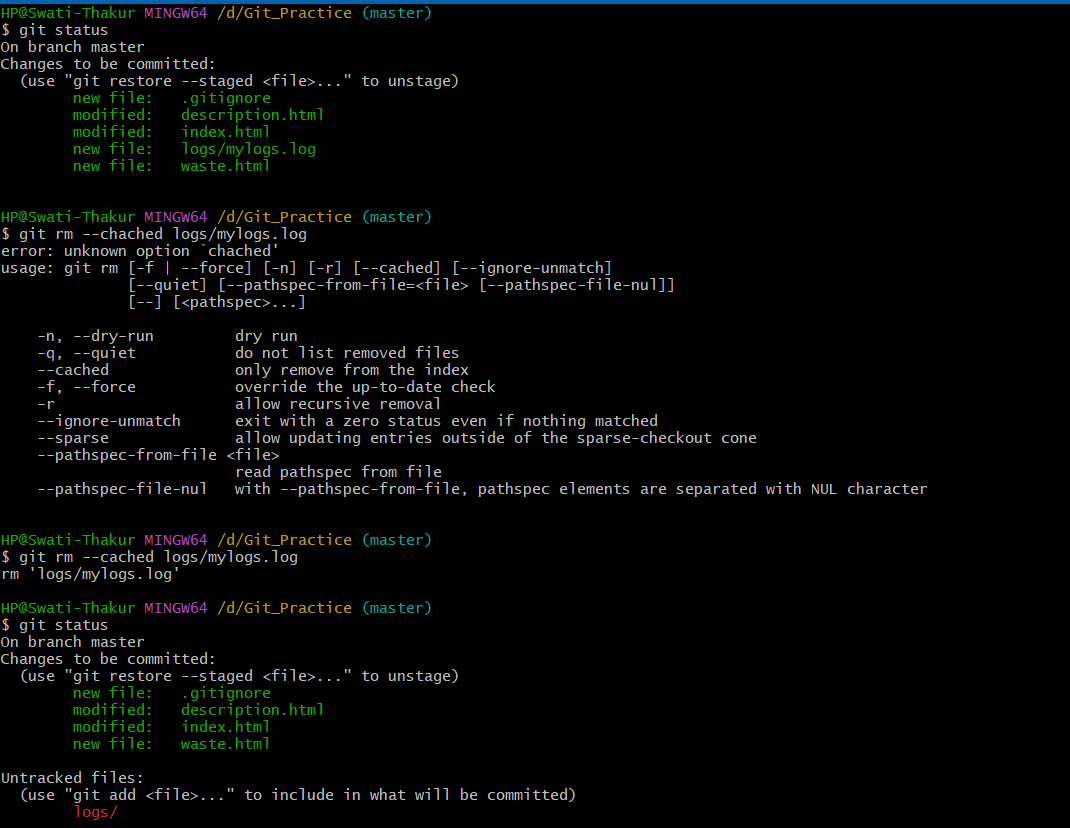




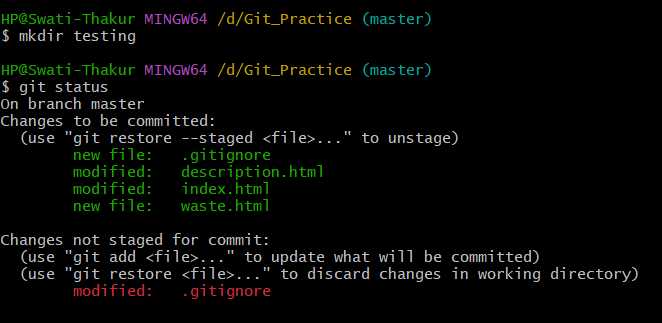


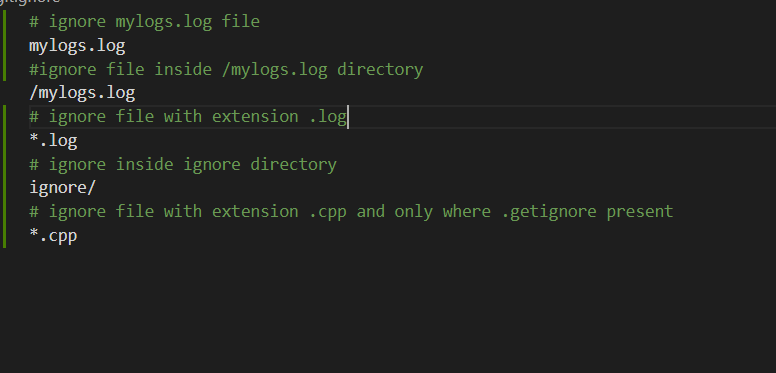


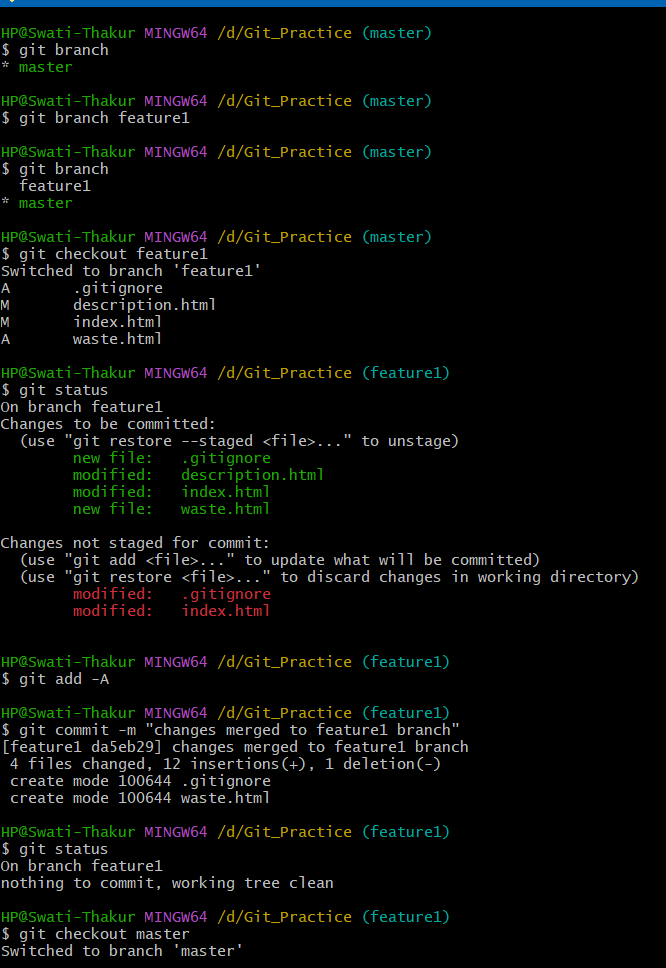
When file is tracked and you add that file .gitignore file then firstlt you have to remove that file from stagged.

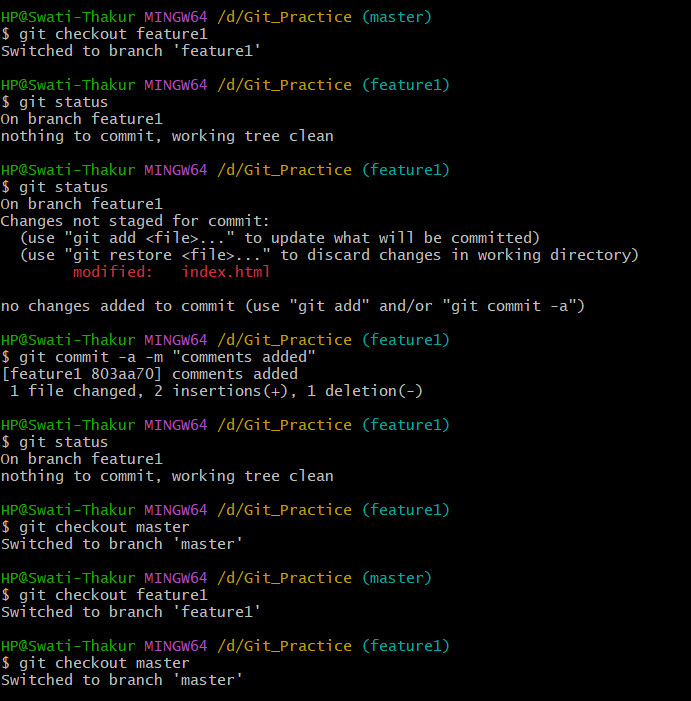


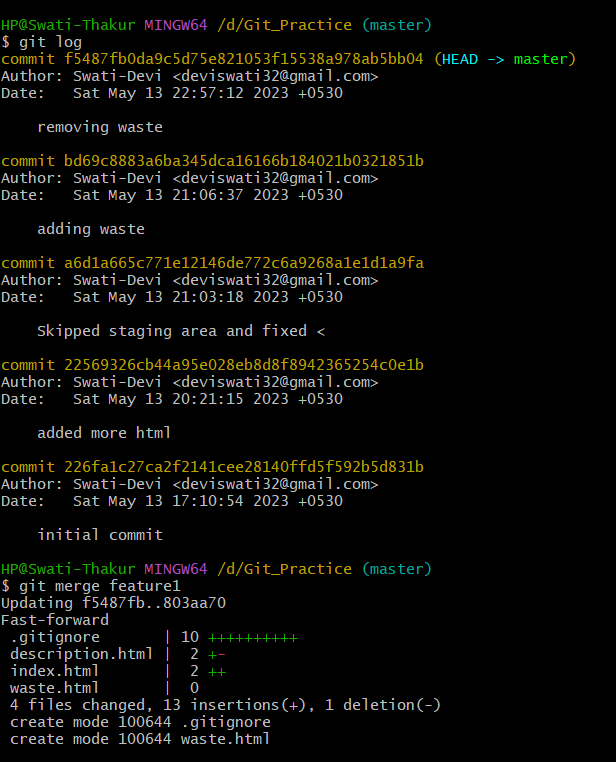
Empty directory not tracked in git. If is contain file then that file is tracked.

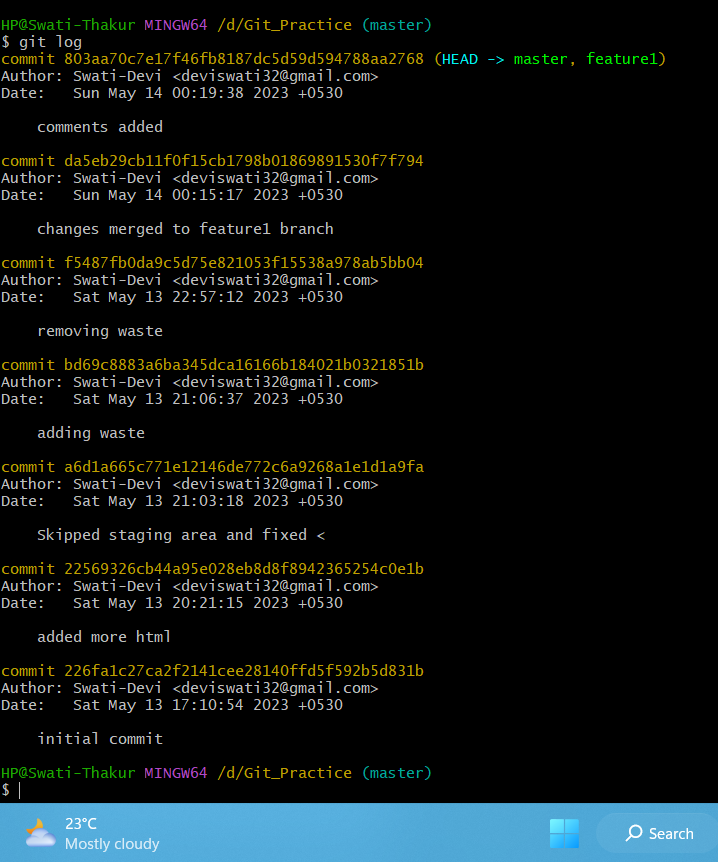


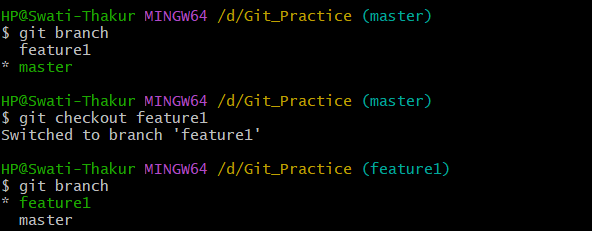




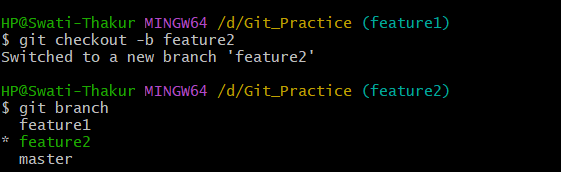


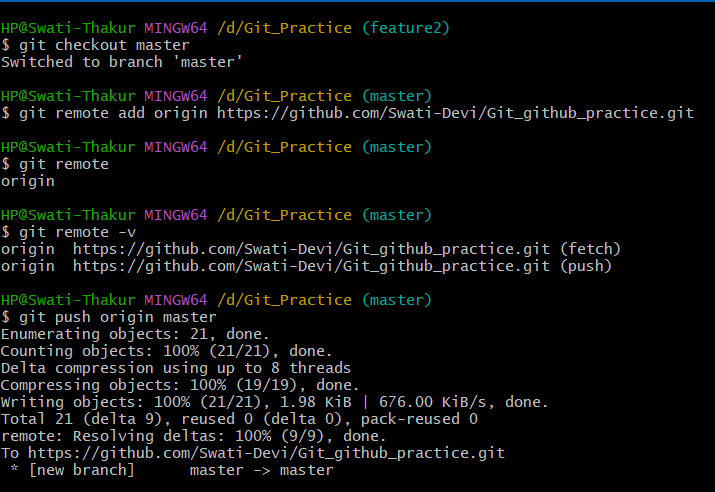


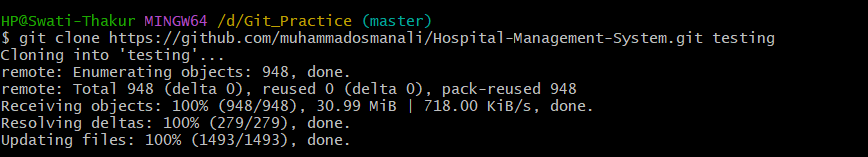




Create new branch and switch directly to it.

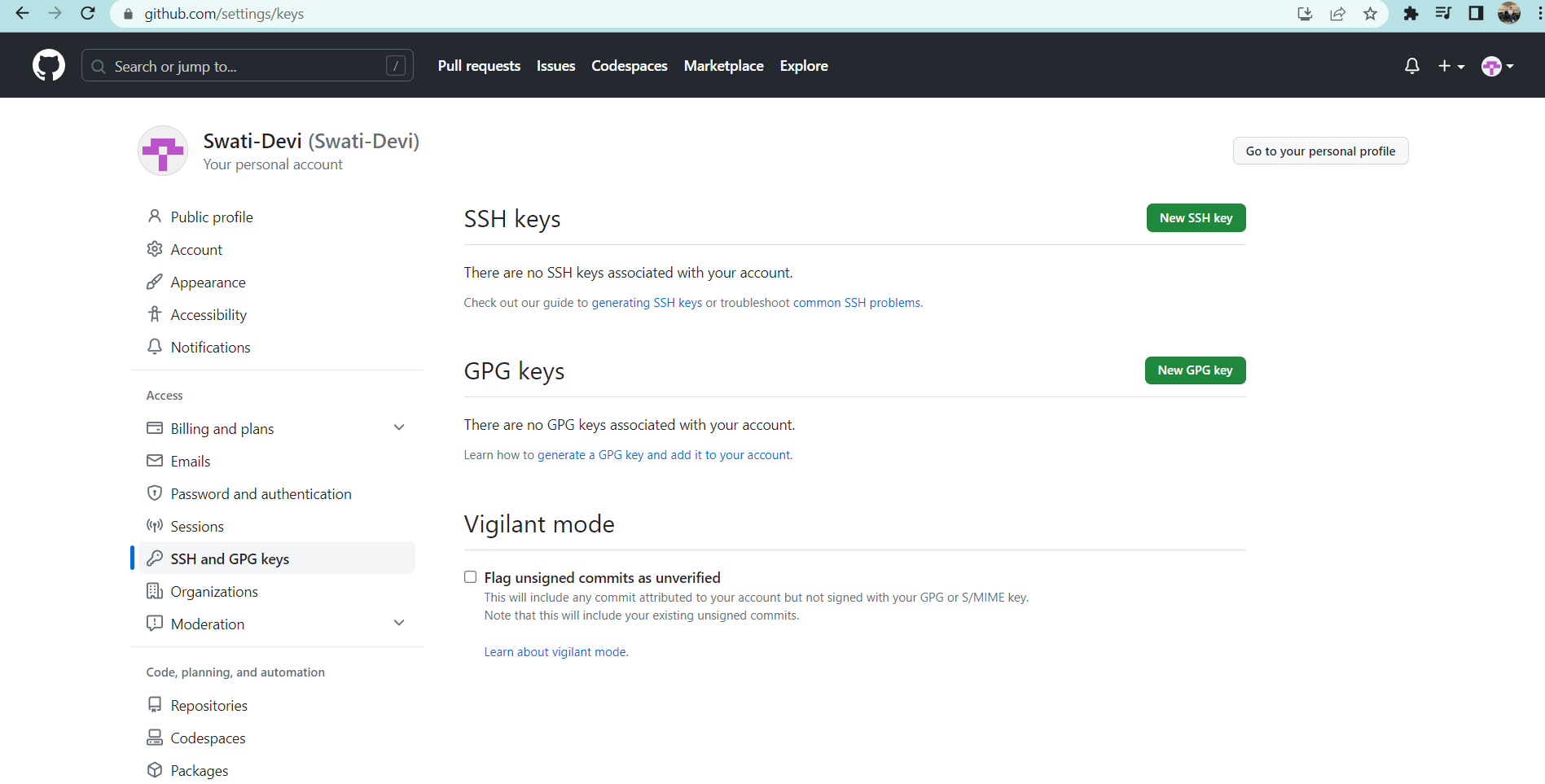


If your remote repository is public   


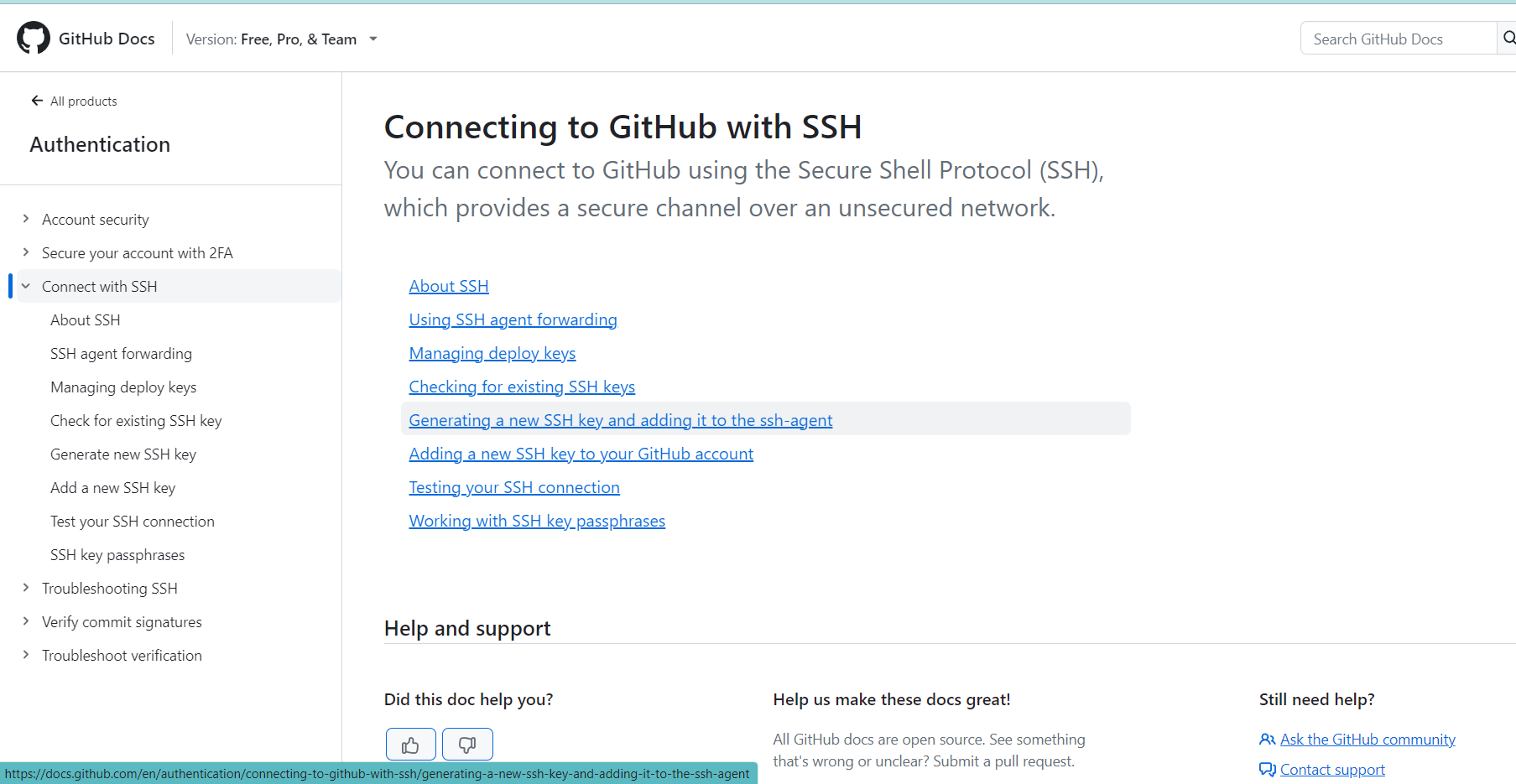


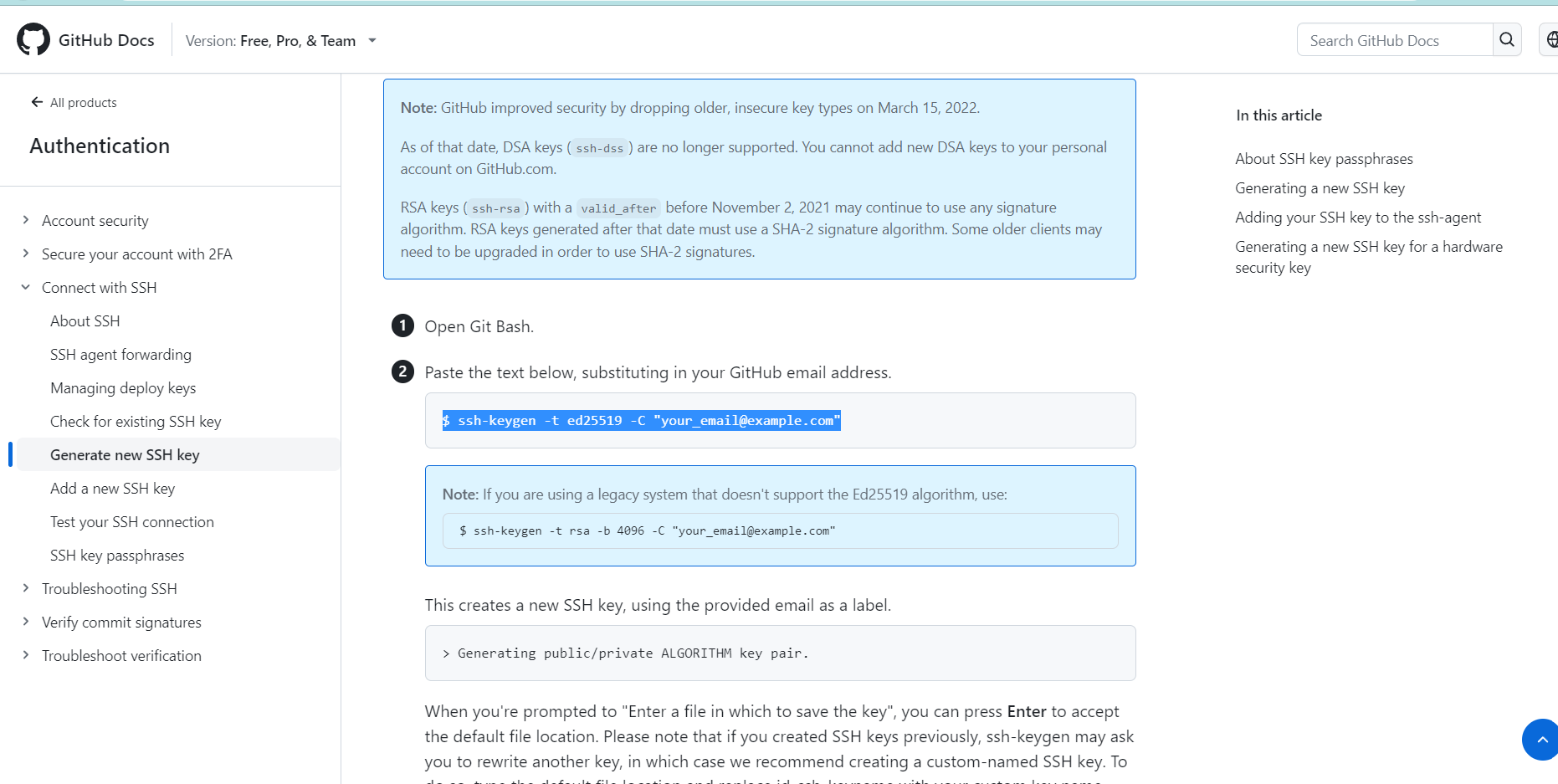


Go to setting open ssh and gpg keys.



Click on generating ssh key.

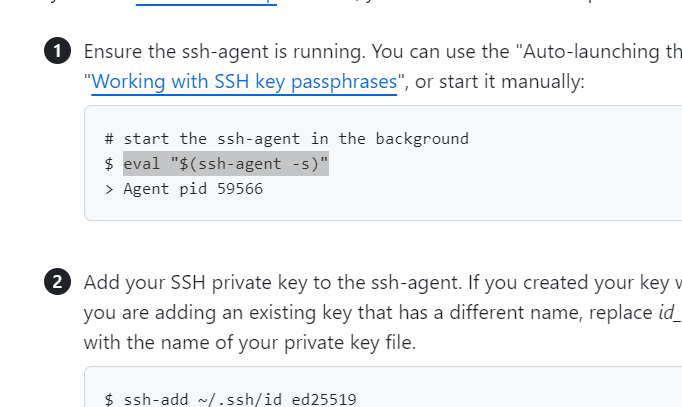


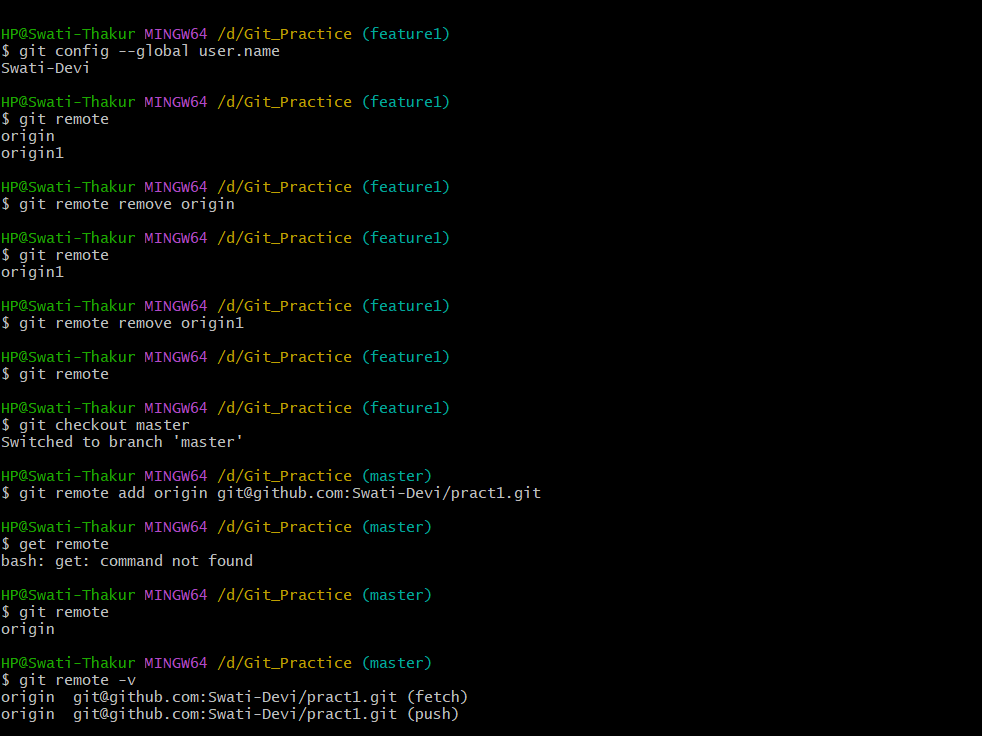


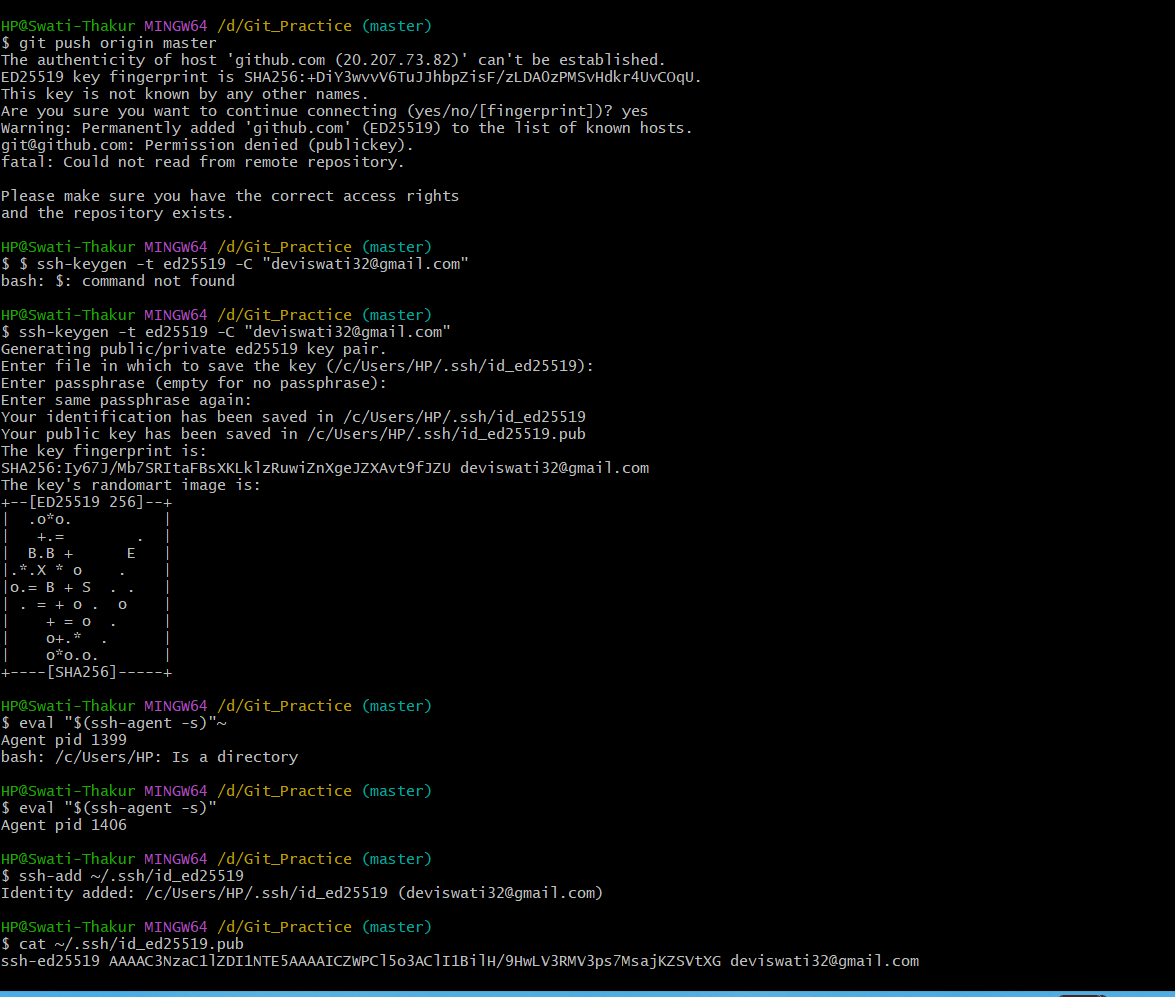
Run below command in git.

$ ssh-keygen -t ed25519 -C [deviswati32@gmail.com](mailto:deviswati32@gmail.com)

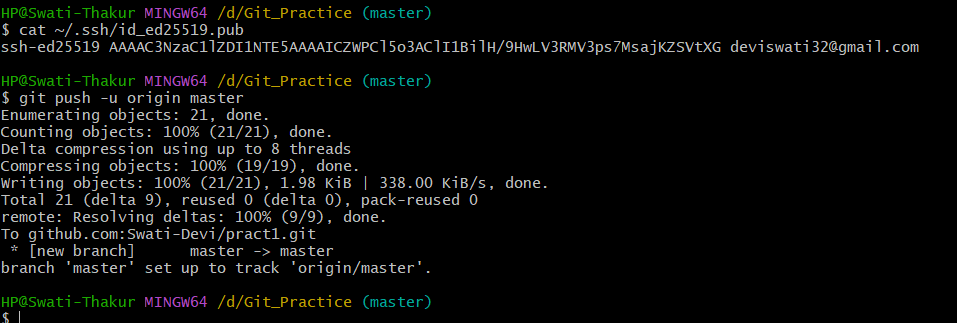
After that run below command in git







Copy the content of cat command and add it to add ssh ket in github.

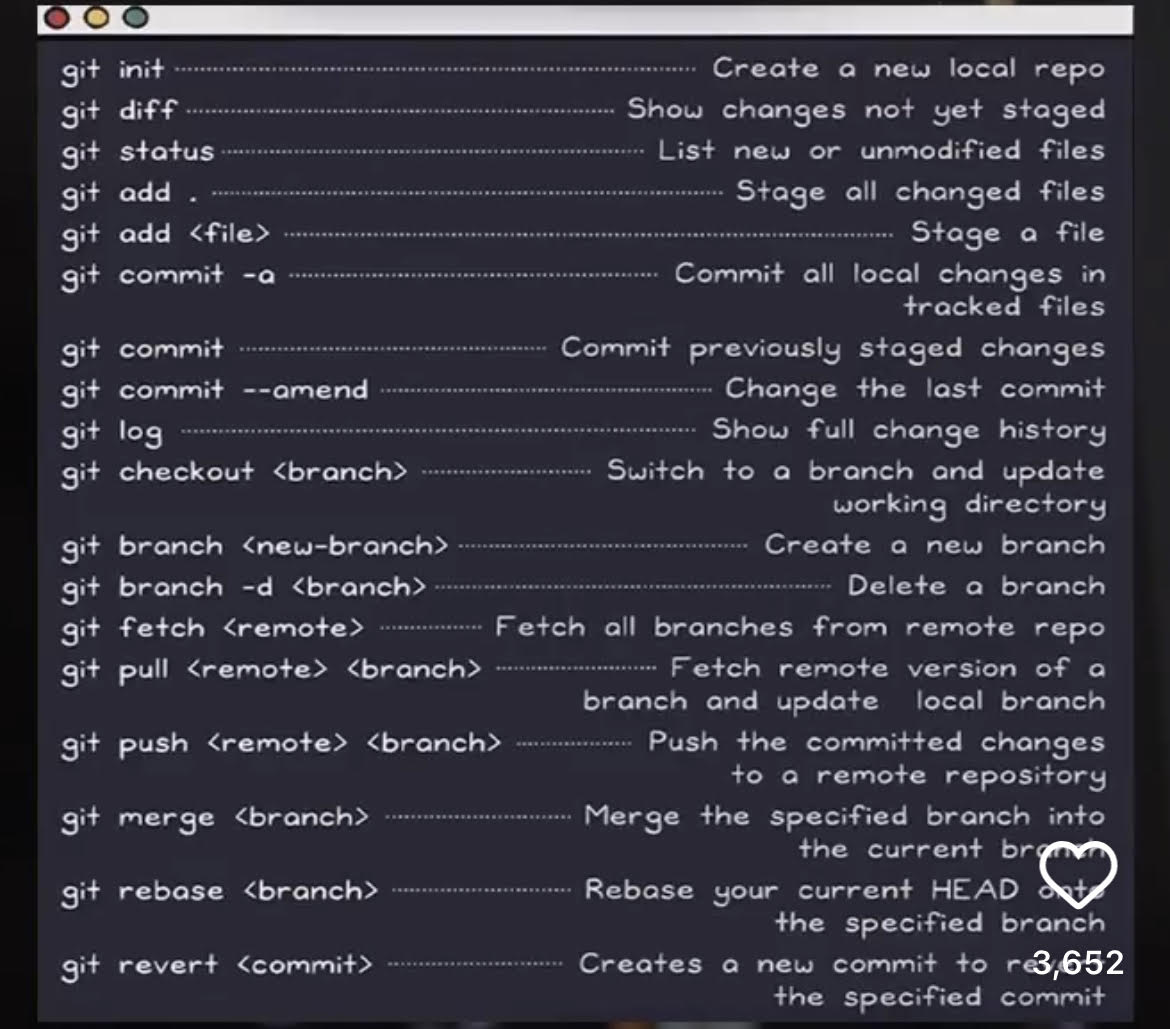


If incase above thing not workout then change repository to ssh type and copy url and paste in below command



Then again type push command.



Git pull command is used to pull change which made by other developer from remote repo or if someone made change in remote and that change will not reflect in our local without pulling them.  
  
  
  
  
  
  
  
NOTE: Shortcut for delete all mail at one time byt add filter in search. Eg : is:unread -is:starred

1. MYSQL | SQL | POSTGReSQL | DJANGO | FLASK  
     
     
   **SQL**:
   * **Definition**: SQL stands for Structured Query Language. It's a standard language used for accessing, manipulating, and managing relational databases.
   * **Usage**: Used in various database systems for creating, querying, and manipulating data.
   * **Characteristic**: It's a language, not a software.
2. **MySQL**:
   * **Definition**: MySQL is an open-source relational database management system (RDBMS).
   * **Usage**: Used for storing, retrieving, and managing data in databases.
   * **Characteristic**: Uses SQL as its query language. It's one of the most popular database systems, known for its fast performance and reliability.
3. **Postgres (PostgreSQL)**:
   * **Definition**: PostgreSQL is another open-source RDBMS, known for its extensibility and SQL compliance.
   * **Usage**: Like MySQL, it's used for managing data in databases but offers some advanced features like table inheritance, function overloading, etc.
   * **Characteristic**: While both MySQL and Postgres are RDBMS, they have differences in terms of features, extensibility, licensing, and performance. Postgres often gets attention for its strong support for custom data types and other advanced features.
4. **Flask**:
   * **Definition**: Flask is a lightweight Python web framework.
   * **Usage**: It's used to build web applications. Flask gives developers more flexibility on which components (like authentication, databases) to use.
   * **Characteristic**: Flask follows the WSGI standard and provides tools, libraries, and technologies to build web applications. It's considered "micro" because it doesn't provide advanced functionalities like Django does out of the box, but this can be added with extensions.
5. **Django**:
   * **Definition**: Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.
   * **Usage**: It's used for building web applications. Django comes with an ORM (Object-Relational Mapping) layer out of the box.
   * **Characteristic**: Django follows the "batteries-included" philosophy, meaning it provides a wide array of functionalities without the need for third-party extensions. For instance, it includes its own admin interface and ORM.

**In summary**:

* SQL is a language for databases.
* MySQL and PostgreSQL are RDBMS software that use SQL as their query language.
* Flask and Django are web frameworks in Python for developing web applications. Flask is lightweight and flexible, while Django provides a more comprehensive set of features out of the box.

Each of these technologies has its own strengths and is suited for different scenarios, so your choice would depend on your project requirements.

## RDBMS

RDBMS stands for Relational Database Management System.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.  
  
  
  
In SQL, joins are used to combine data from two or more tables based on a related column between them. There are several types of joins, each serving a different purpose. Here are the main types of joins with explanations and examples:

1. **INNER JOIN:** The INNER JOIN returns only the rows that have matching values in both tables.

**Syntax:**

sqlCopy code

SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;

**Example:** Consider two tables, **Customers** and **Orders**. We want to retrieve a list of customers who have placed orders.

sqlCopy code

SELECT Customers.CustomerName, Orders.OrderDate FROM Customers INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

1. **LEFT JOIN (or LEFT OUTER JOIN):** The LEFT JOIN returns all rows from the left table and the matching rows from the right table. If there's no match, NULL values are used for missing values from the right table.

**Syntax:**

sqlCopy code

SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;

**Example:** Using the same **Customers** and **Orders** tables, we want to retrieve a list of all customers along with their orders (if any).

sqlCopy code

SELECT Customers.CustomerName, Orders.OrderDate FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

1. **RIGHT JOIN (or RIGHT OUTER JOIN):** The RIGHT JOIN returns all rows from the right table and the matching rows from the left table. If there's no match, NULL values are used for missing values from the left table.

**Syntax:**

sqlCopy code

SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;

**Example:** If we want to retrieve a list of all orders along with their customers (if any), we can use a RIGHT JOIN.

sqlCopy code

SELECT Customers.CustomerName, Orders.OrderDate FROM Customers RIGHT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

1. **FULL JOIN (or FULL OUTER JOIN):** The FULL JOIN returns all rows when there's a match in either the left or the right table. If there's no match, NULL values are used for missing values.

**Syntax:**

sqlCopy code

SELECT columns FROM table1 FULL JOIN table2 ON table1.column = table2.column;

**Example:** Suppose we want to retrieve a list of customers and their associated orders, including customers who have no orders and orders with no customers.

sqlCopy code

SELECT Customers.CustomerName, Orders.OrderDate FROM Customers FULL JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

1. **CROSS JOIN:** The CROSS JOIN (or Cartesian Join) returns the Cartesian product of two tables, resulting in every possible combination of rows.

**Syntax:**

sqlCopy code

SELECT columns FROM table1 CROSS JOIN table2;

**Example:** If we have tables **Colors** and **Sizes**, and we want to generate all possible combinations of colors and sizes:

sqlCopy code

SELECT Colors.ColorName, Sizes.SizeName FROM Colors CROSS JOIN Sizes;

These are the main types of joins in SQL, each serving different purposes when combining data from multiple tables.

# **SQL TOP, LIMIT, FETCH FIRST or ROWNUM Clause**

**Note:** Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses FETCH FIRST n ROWS ONLY and ROWNUM. **SQL Server / MS Access Syntax:**

SELECT TOP *number*|*percent* *column\_name(s)*  
FROM *table\_name*WHERE *condition*;

**MySQL Syntax:**

SELECT *column\_name(s)*  
FROM *table\_name*WHERE *condition*  
LIMIT *number*;

**Oracle 12 Syntax:**

SELECT *column\_name(s)*  
FROM *table\_name*ORDER BY *column\_name(s)*  
FETCH FIRST *number* ROWS ONLY;

**Older Oracle Syntax:**

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE ROWNUM <= *number*;

**Older Oracle Syntax (with ORDER BY):**

SELECT \*  
FROM (SELECT *column\_name(s)*FROM *table\_name* ORDER BY *column\_name(s)*)  
WHERE ROWNUM <= *number*;

## The SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

### **MIN() Syntax**

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### **MAX() Syntax**

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

SELECT MIN(cnt) AS min\_count

FROM (

SELECT COUNT(DISTINCT City) AS cnt

FROM customers

) AS counts;

## The SQL COUNT(), AVG() and SUM() Functions

The COUNT() function returns the number of rows that matches a specified criterion.

### **COUNT() Syntax**

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

The AVG() function returns the average value of a numeric column.

### **AVG() Syntax**

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

The SUM() function returns the total sum of a numeric column.

### **SUM() Syntax**

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

The SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

* The percent sign (%) represents zero, one, or multiple characters
* The underscore sign (\_) represents one, single character

**Note:** MS Access uses an asterisk (\*) instead of the percent sign (%), and a question mark (?) instead of the underscore (\_).

The percent sign and the underscore can also be used in combinations!

## SQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

### **Wildcard Characters in MS Access**

### **Wildcard Characters in SQL Server**

All the wildcards can also be used in combinations!

## Here are some examples showing different LIKE operators with '%' and '\_' wildcards: The SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

### **IN Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (value1, value2, ...);

or:

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (*SELECT* STATEMENT);

## The SQL BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

### **BETWEEN Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name BETWEEN value1 AND value2;

## MySQL Aliases

Aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

### **Alias Column Syntax**

SELECT column\_name AS alias\_name  
FROM table\_name;

### **Alias Table Syntax**

## SELECT column\_name(s) FROM table\_name AS alias\_name; SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

## SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

### **INNER JOIN Syntax**

## SELECT column\_name(s) FROM table1 INNER JOIN table2ON table1.column\_name = table2.column\_name; SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

### **LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2ON table1.column\_name = table2.column\_name;

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

### **RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name = table2.column\_name;

The SQL UNION Operator

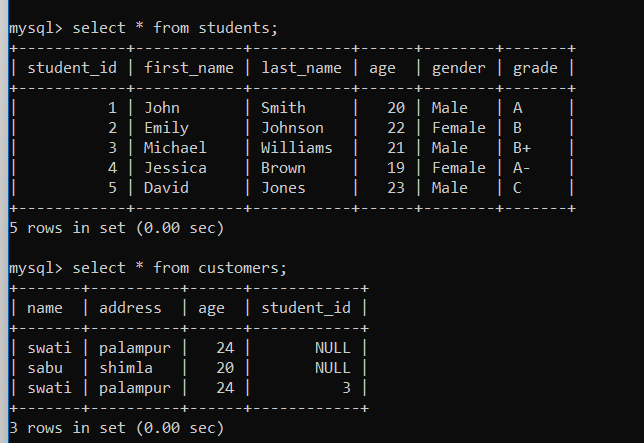
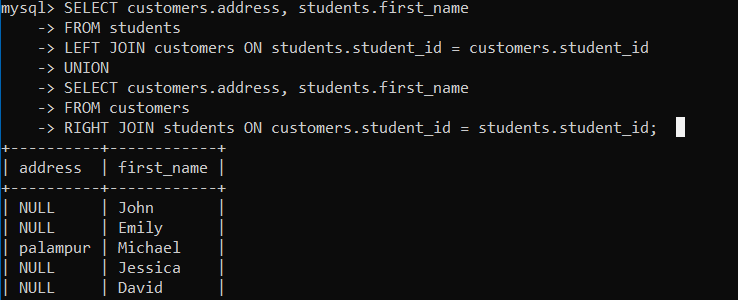
The UNION operator is used to combine the result-set of two or more SELECT statements.

* Every SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in every SELECT statement must also be in the same order

### **UNION Syntax**

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

**Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement.  
  
  
 ***full outer join alternatice is left, right and union:***

## The SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

### **GROUP BY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s);

## The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

### **HAVING Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

Aggregate functions in SQL are used to perform calculations on sets of values and return a single value as the result. These functions are often used with the **SELECT** statement to summarize data in a meaningful way. Here are some common aggregate functions:

1. **COUNT()**: Counts the number of rows that match a specific condition.

Example:

sqlCopy code

SELECT COUNT(\*) FROM orders;

1. **SUM()**: Calculates the sum of values in a column.

Example:

sqlCopy code

SELECT SUM(total\_amount) FROM orders;

1. **AVG()**: Calculates the average (mean) of values in a column.

Example:

sqlCopy code

SELECT AVG(price) FROM products;

1. **MIN()**: Retrieves the minimum value from a column.

Example:

sqlCopy code

SELECT MIN(order\_date) FROM orders;

1. **MAX()**: Retrieves the maximum value from a column.

Example:

sqlCopy code

SELECT MAX(price) FROM products;

1. **GROUP\_CONCAT()**: Concatenates values from multiple rows into a single string, grouped by a specified column.

Example:

sqlCopy code

SELECT category, GROUP\_CONCAT(product\_name) FROM products GROUP BY category;

1. **HAVING**: Works with the **GROUP BY** clause to filter aggregated results based on conditions.

Example:

sqlCopy code

SELECT category, AVG(price) as avg\_price FROM products GROUP BY category HAVING avg\_price > 100;

These functions are often used in combination with the **GROUP BY** clause to group data before applying the aggregation. The **GROUP BY** clause divides the result set into groups based on one or more columns, and then the aggregate functions are applied to each group.

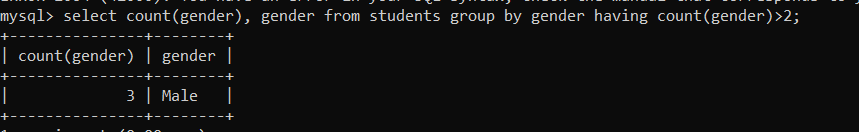
For example:

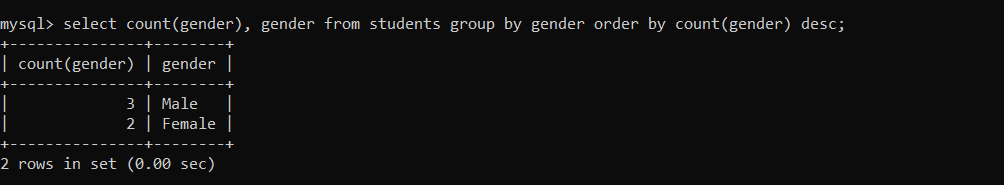
sqlCopy code

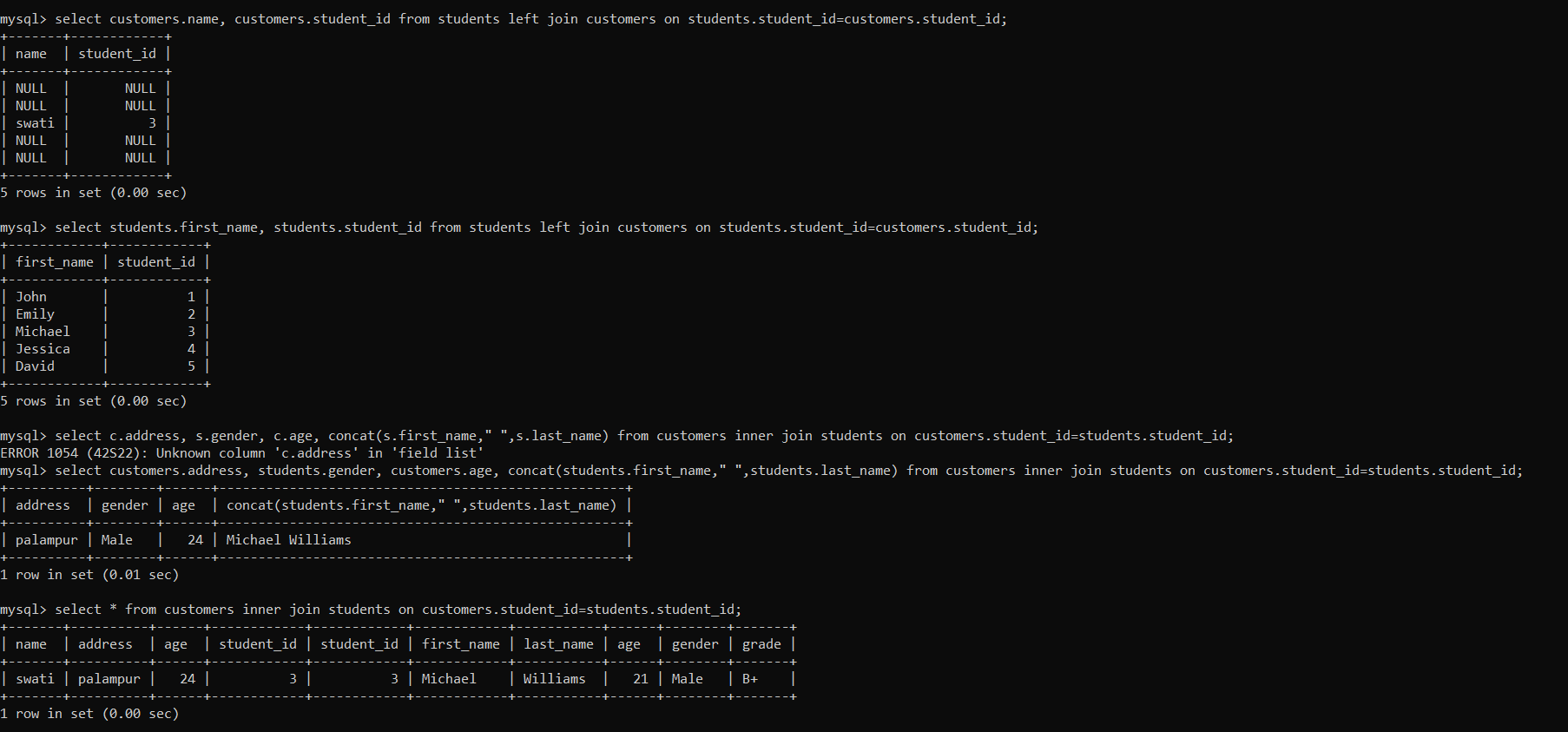
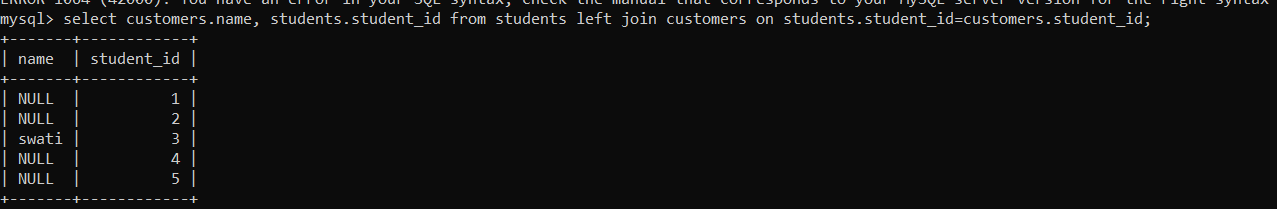
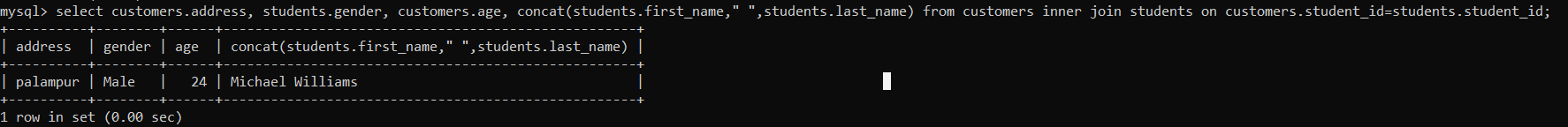
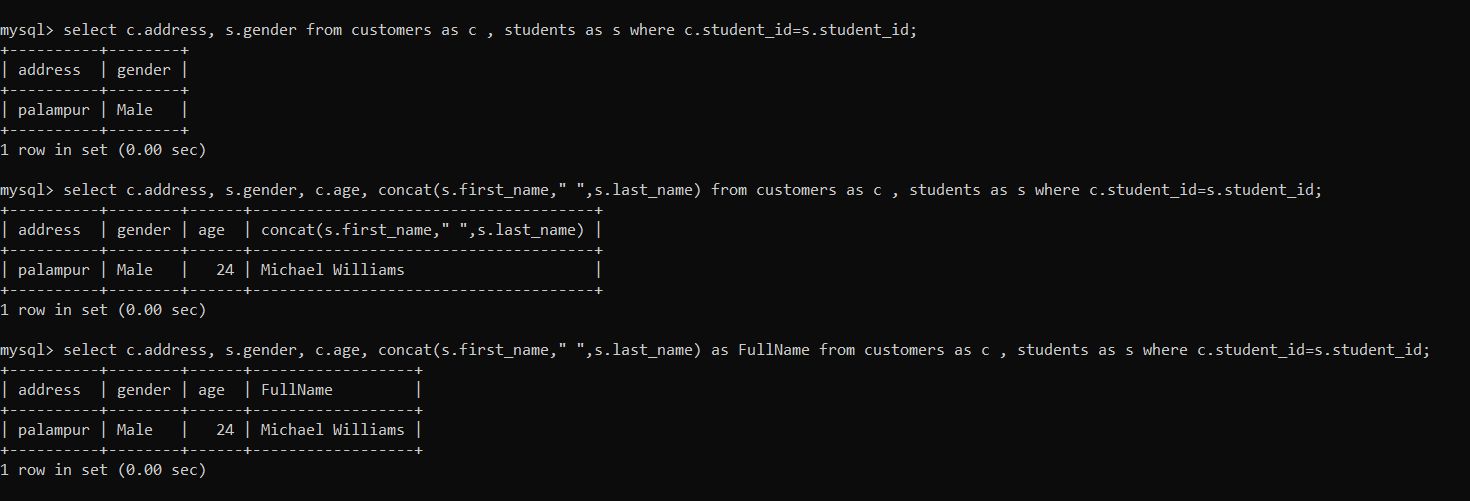
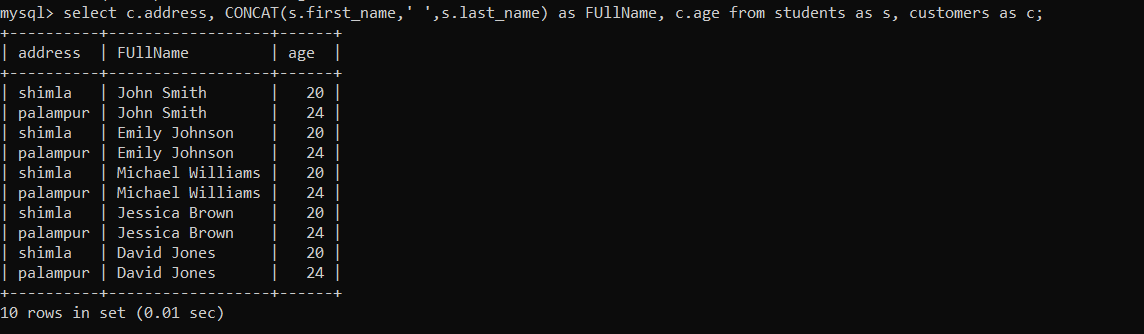
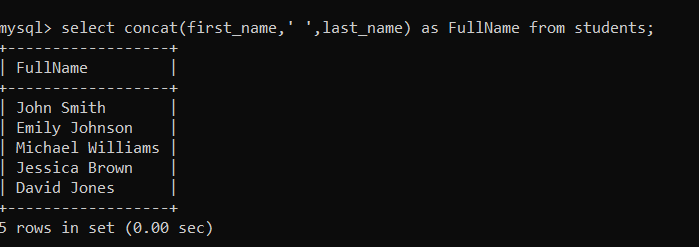
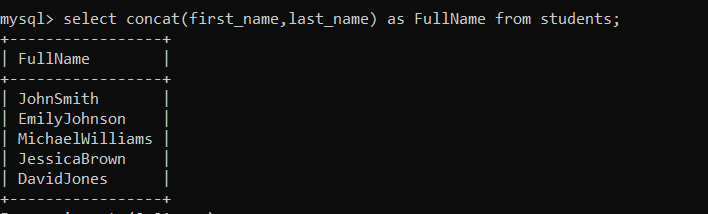
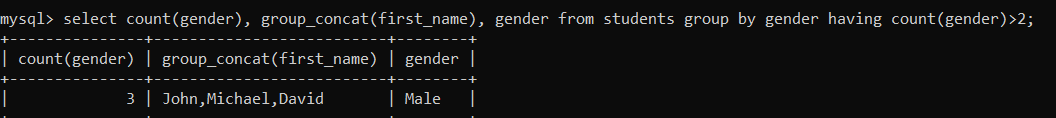
SELECT department, AVG(salary) as avg\_salary FROM employees GROUP BY department;

In this query, the **AVG()** function calculates the average salary for each department group specified by the **GROUP BY** clause.

Keep in mind that different database systems might have slight variations in their supported aggregate functions and syntax. It's always a good idea to refer to the documentation for the specific database system you're using for accurate and detailed information on aggregate functions.





## The SQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records.

### **EXISTS Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE EXISTS  
(SELECT column\_name FROM table\_name WHERE condition);

## The SQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

## The SQL ANY Operator

The ANY operator:

* returns a boolean value as a result
* returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

### **ANY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name operator ANY  
  (SELECT column\_name  FROM table\_name  WHERE condition);

## The SQL ALL Operator

The ALL operator:

* returns a boolean value as a result
* returns TRUE if ALL of the subquery values meet the condition
* is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

### **ALL Syntax With SELECT**

SELECT ALL column\_name(s)  
FROM table\_name  
WHERE condition;

### **ALL Syntax With WHERE or HAVING**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name operator ALL  
  (SELECT column\_name  FROM table\_name  WHERE condition);

**Note:** The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

## The SQL SELECT INTO Statement

The SELECT INTO statement copies data from one table into a new table.

### **SELECT INTO Syntax**

Copy all columns into a new table:

SELECT \*  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

Copy only some columns into a new table:

SELECT column1, column2, column3, ...  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.