1. Create a new GitHub repository.

* Clone the repository to your local machine using SSH (generate an SSH key if needed, add the public key to your GitHub account).
* Create a new branch named after your username (e.g., Tutedude).
* Add your Flask project files to this branch.
* Commit the changes and merge the branch into the main branch.

Solution:-

# generate key :-

ssh-keygen -t ed25519 -C "your\_email@example.com"

eval "$(ssh-agent -s)"; ssh-add ~/.ssh/id\_ed25519

# clone

git clone git@github.com:YOUR\_GITHUB\_USERNAME/my-flask-app.git

cd my-flask-app

# create branch, add files, commit

git checkout -b Tutedude

# copy your Flask files into this folder

git add .

git commit -m "Add Flask project files"

# push branch, create PR or merge locally

git push -u origin Tutedude

# either create PR on GitHub or

git checkout main

git pull origin main

git merge --no-ff Tutedude

git push origin main

2. Create a new branch named <your\_name>\_new (e.g., Tutedude\_new).

* Update the content of the JSON file used for the /api route in this branch.
* Merge the <your\_name>\_new branch into the main branch.
* If there are conflicts during the merge, resolve them by accepting the changes from the <your\_name>\_new branch.
* Add the resolved changes to the staging area, commit them, and push the updates to the remote repository.

Solution:-

1. Create new branch (Tutedude):

git checkout -b Tutedude\_new

Explanation:-

* git checkout -b creates and switches you to a new branch called Tutedude\_new.
* This branch is independent of main, so you can experiment safely.

1. Update the JSON file in this branch:-

Before (data.json):

{

"students": ["Saie", "Sanika"]

}

After (data.json):

{

"students": ["Saie", "Sanika", "Tutedude"]

}

Explanation:

You are making a change only in the branch Tutedude\_new, so the main branch is untouched.

3)Stage and commit the changes:-

git add data.json

git commit -m "Update JSON file with new content in Tutedude\_new branch"

Explanation:

git add stages the file for commit.  
git commit permanently records the change in the branch history.

4) Merge <Tutedude>\_ new into main

First, switch back to main:

git checkout main

Make sure it’s up-to-date:

git pull origin main

Now merge:

git merge Tutedude\_new

Example :-

* This tries to combine the work from Tutedude\_new into main.
* If there are no conflicts, the merge will succeed immediately.
* If there **are conflicts**, Git will tell you which files are in conflict.

5)Add resolved changes to staging area and commit:-

git add data.json

git commit -m "Resolved merge conflict by accepting changes from Tutedude\_new"

Explanation:-

* After fixing conflicts manually, you must add the fixed file.
* Then commit records the resolution.

6)Push the updates to the remote repository:-

git push origin main

Explanation:-

* This uploads your merged branch (with resolved changes) to GitHub.
* Now main contains the new JSON changes from your branch.

3. **Branch Creation**:

* Create two branches: master\_1 and master\_2 from the main branch.
* **Feature Development in** master\_1:
* In the master\_1 branch, create a **To-Do Page** in the frontend.
  + The page should contain a form with the following fields:
    - **Item Name**
    - **Item Description**
* **Backend API in** master\_2:
* In the master\_2 branch, create a backend route named /submittodoitem.
* This route will:
  + Accept itemName and itemDescription via a POST request.
  + Store these details in a MongoDB database.
* **Merging Changes**:
* Merge the changes from both master\_1 and master\_2 into the main branch.

Solution:-

1)Branch Creation:-

git checkout main

git pull origin main # make sure it’s up to date

# create two branches

git checkout -b master\_1

git checkout main

git checkout -b master\_2

Explanation:-

* master\_1 → for frontend changes.
* master\_2 → for backend changes.

2) **Feature Development in master\_1 (Frontend To-Do Page)**

### **Example frontend file: templates/todo.html**

<!DOCTYPE html>

<html>

<head>

<title>To-Do Page</title>

</head>

<body>

<h2>Add a To-Do Item</h2>

<form action="/submittodoitem" method="POST">

<label>Item Name:</label><br>

<input type="text" name="itemName" required><br><br>

<label>Item Description:</label><br>

<textarea name="itemDescription" required></textarea><br><br>

<button type="submit">Submit</button>

</form>

</body>

</html>

Update Flask app (app.py) to serve frontend:

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route("/todo")

def todo\_page():

return render\_template("todo.html")

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

Commit frontend changes:-

git add templates/todo.html app.py

git commit -m "Add To-Do page with form in master\_1 branch"

git push -u origin master\_1

## **Backend API in master\_2**

Switch to backend branch:

git checkout master\_2

Install MongoDB client for Flask:

pip install flask pymongo

Update app.py with /submittodoitem route:

from flask import Flask, request, jsonify

from pymongo import MongoClient

app = Flask(\_\_name\_\_)

# connect to MongoDB (local or Atlas cluster)

client = MongoClient("mongodb://localhost:27017/") # replace with your MongoDB URI

db = client["todo\_db"]

collection = db["items"]

@app.route("/submittodoitem", methods=["POST"])

def submit\_todo\_item():

item\_name = request.form.get("itemName")

item\_description = request.form.get("itemDescription")

if not item\_name or not item\_description:

return jsonify({"error": "Both fields are required"}), 400

# insert into MongoDB

todo\_item = {

"name": item\_name,

"description": item\_description

}

collection.insert\_one(todo\_item)

return jsonify({"message": "To-Do item added successfully!"}), 201

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

Commit backend changes:-

git add app.py

git commit -m "Add /submittodoitem backend route with MongoDB integration in master\_2"

git push -u origin master\_2

Merge Changes into main:-

Merge master\_1:-

git checkout main

git pull origin main

git merge master\_1

Merge master\_2:-

git merge master\_2

from flask import Flask, render\_template, request, jsonify

from pymongo import MongoClient

app = Flask(\_\_name\_\_)

# MongoDB setup

client = MongoClient("mongodb://localhost:27017/")

db = client["todo\_db"]

collection = db["items"]

# Frontend route

@app.route("/todo")

def todo\_page():

return render\_template("todo.html")

# Backend route

@app.route("/submittodoitem", methods=["POST"])

def submit\_todo\_item():

item\_name = request.form.get("itemName")

item\_description = request.form.get("itemDescription")

if not item\_name or not item\_description:

return jsonify({"error": "Both fields are required"}), 400

collection.insert\_one({

"name": item\_name,

"description": item\_description

})

return jsonify({"message": "To-Do item added successfully!"}), 201

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

1. Save the file, stage, and commit the resolved merge:  
   git add [app.py](http://app.py)

git commit -m "Resolved merge conflicts: combined frontend and backend changes"

Push final main branch:-

git push origin main

Testing:-

python app.py

4) **Enhancing the To-Do Form in** master\_1:

* In the master\_1 branch, add the following fields to the To-Do form:
  + **Item ID**
  + **Item UUID**
  + **Item Hash**
* **Committing in Sequence**:
* Add and commit each field separately in the following order:
  + **First commit**: Add **Item ID** field.
  + **Second commit**: Add **Item UUID** field.
  + **Third commit**: Add **Item Hash** field.
* **Merging to** main:
* Merge the master\_1 branch into the main branch.
* **Git Reset and Commit Deletion**:
* In the main branch, use **Git Reset** to roll back to the commit where only the **Item ID** field was added.
* Use git reset --soft to ensure changes remain staged.
* Re-commit this state to the main branch.
* Merge this updated state to the main branch.
* **Rebasing Changes**:
* Rebase the updated changes in the main branch to the master\_1 branch.  
  **Clarification**:
  + During rebasing, **preserve individual commits** to maintain the commit history for each change (i.e., do not squash commits).
  + Use git rebase main master\_1 to integrate changes from the main branch back into the master\_1 branch.

Solution:-

# in master\_1

git add templates/todo.html

git commit -m "Add Item ID field"

git add templates/todo.html

git commit -m "Add Item UUID field"

git add templates/todo.html

git commit -m "Add Item Hash field"

# merge into main

git checkout main

git merge master\_1

git push origin main

# reset main to Item ID commit

git log --oneline

git reset --soft <commit\_hash\_of\_Item\_ID>

git commit -m "Re-commit only Item ID field state"

git push origin main --force

# rebase master\_1 on top of updated main

git checkout master\_1

git rebase main