Session – 1

Date: August 30, 2025

Topics: GitHub Concepts, Machine Learning Overview, APIs, and Full Stack App Integration.

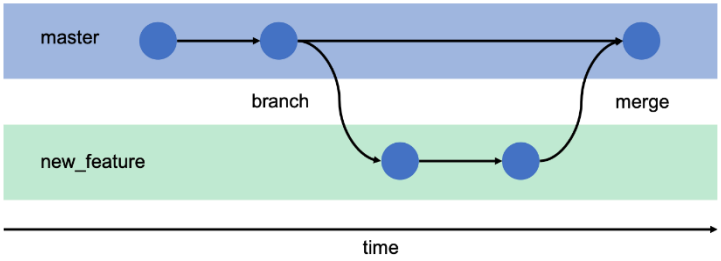
Part 1: Collaborative Development with GitHub

The initial part of the session focused on GitHub concepts with practical use cases to foster effective teamwork and collaboration.

Core Concepts Explained The session covered essential commands and workflows:

1. **Branches (Local & Remote)**

   Think of a branch as a **parallel timeline** for your project. It allows you to work on a new feature (e.g., "add-user-login") or a bug fix ("fix-payment-glitch") in isolation without affecting the main, stable version of your code (which is usually in a branch called main or master).



   This is the cornerstone of collaborative development.

- **Local Branches**: These exist only on your computer. You create them to do your work.
- **Remote Branches**: These are versions of your branches that live on a shared server like GitHub. You **push** your local branch to the remote to share it with your team.

Common Commands

- **List all branches** (local and remote):

  git branch -a

- **Create a new local branch**:

  git branch <new-branch-name>

- **Switch to a branch** to start working on it:

  git switch <branch-name> # or the older command: git checkout <branch-name>

- **Create and switch to a new branch** in one step:

  git switch -c <new-branch-name>

  # or the older command: git checkout -b <new-branch-name>

- **Push a new local branch** to the remote repository (GitHub) to share it:

  git push -u origin <branch-name>

2. **Push, Pull, and Pull Request**

This is the fundamental workflow for sharing and integrating code with a team.

1. **git pull: Get the Latest Changes**

Before you start working, you should always update your local code with the latest changes from the remote repository. git pull fetches the changes from the remote and merges them into your current branch.

- **Syntax**:

  git pull origin <branch-name>

2. **git push: Share Your Work**

After you have committed your changes locally, you use git push to upload those commits to a remote branch on GitHub, making them available to your team.

- **Syntax**:

  git push origin <branch-name>

3. **Pull Request (PR): Propose Your Changes**

A Pull Request is **not a command**, but a feature on GitHub. After you've pushed your branch, you create a PR to ask for your changes to be merged into the main branch. It's a formal request for code review.

The Process:

1. **Push** your branch to GitHub.
2. Go to your repository on GitHub. You will often see a prompt to "Compare & pull request".
3. You select the branch you want to merge (feature/add-user-login) and the branch you want to merge it into (main).
4. You give the PR a title and description, explaining what your changes do.
5. Your team can then review the code, leave comments, and request changes before it is approved and merged.

3. **Merge Conflict**

A merge conflict happens when Git can't automatically merge two branches. This typically occurs when two people have edited the **same lines in the same file**, and Git doesn't know which version to keep.

How It Looks

Git will stop the merge and mark the conflicted area in the file with special symbols:

```
<<<<<<< HEAD
// Code from your current branch (e.g., your new feature)
const color = "blue";
=======
// Code from the other branch you are merging in
const color = "red";
>>>>>>> other-branch-name
```

- <<<<<<< HEAD: Marks the beginning of your changes.
- =======: Separates your changes from the incoming changes.
- >>>>>>>: Marks the end of the incoming changes.

How to Resolve It Open the conflicted file in your code editor.

I. **Manually edit the file**. Decide which code to keep. You might keep your version, the other version, or a combination of both.

II. **Delete all the conflict markers** (<<<<<<<, =======, >>>>>>>).

III. **Save** the file.

IV. **Stage the resolved file** using git add.

```
git add <path/to/conflicted/file.js>
```

V. **Commit the merge** to finalize the resolution.

```
git commit -m "Resolved merge conflict in file.js"
```

4. **Issues**

Like Pull Requests, **Issues** are a project management feature on GitHub, not a Git command. They are a centralized place to track tasks, enhancements, and bugs for your project.

Key Features:

- **Bug Reports**: Users and developers can report bugs in detail.
- **Feature Requests**: A place to propose and discuss new ideas.
- **Task Lists**: You can create checklists within an issue to track progress on a complex task.
- **Labels**: Categorize issues (e.g., bug, enhancement, documentation, help-wanted) for easy filtering.
- **Assignees**: Assign an issue to a specific team member to give them ownership.
- **Linking**: You can link Issues to Pull Requests. For example, when the PR that fixes a bug is merged, the corresponding issue can be closed automatically.

Hands-On Practice Labs

To ensure practical understanding, completed three guided labs:

1. **Introduction to GitHub**: https://github.com/skills/introduction-to-github
2. **Review Pull Requests**: https://github.com/skills/review-pull-requests
3. **Resolve Merge Conflicts**: https://github.com/skills/resolve-merge-conflicts

Part 2: Intro to Machine Learning & API Integration

The second half of the session learned a high-level overview of Machine Learning basics and how to connect an ML model to a full-stack application. **The Machine Learning Lifecycle** the session walked through the typical stages of an ML project at a surface level.

The Main Steps of an ML Project

Think of the ML lifecycle as a recipe for building an intelligent system. It's a structured process that takes you from a raw idea to a functioning application.

1. Data & Segregation

- **Key Concept:** We split our data into two main parts:
    - **Training Set:** This is the majority of the data (e.g., 80%). The model studies this data to learn patterns. It's like the textbook and practice problems you use to study for an exam.
    - **Testing Set:** This is the smaller, leftover part (e.g., 20%). The model has **never** seen this data before. We use it to evaluate how well the model learned. This is like the final exam itself.
- **Why split?** To avoid "cheating." If the model just memorizes the training data, it won't perform well on new, unseen data. The test set gives us an honest grade of its performance.

2. Algorithms & Model Creation

- **Algorithm:** This is the "learning" method we choose. It's the process that finds patterns, rules, and relationships within the training data. Examples include Linear Regression, Decision Trees, etc.
- **Model:** The **output** of the training process is the model. You can think of the model as the "brain" that has been trained. It's a file containing all the patterns the algorithm learned.

3. Prediction & Tuning

- **Prediction:** Now we use our trained model on new data (like our test set) to make educated guesses, or "predictions."
- **Tuning:** The first version of your model is rarely perfect. Tuning is the process of adjusting the model's settings (called **hyperparameters**) to improve its accuracy. It's like tuning a guitar to get the perfect sound.

A Quick Word on Neural Networks

- This is a more advanced subfield of ML, inspired by the structure of the human brain. Neural networks are very powerful and are used for complex tasks like image recognition and natural language processing.

From a Trained Model to a Real Application

A trained model on your computer isn't very useful to others. We need a way to let other applications (like a website or a mobile app) use it.

- Use pickle for learning purposes, but know that joblib (for scikit-learn) and ONNX or TF SavedModel (for production) are more standard.
- Once a model is trained, you need to save it.
- A common way to do this in Python is by using the pickle library, which saves your model as a **.pkl file**.
- This file essentially "freezes" your trained model's brain, so you can load it and use it anytime without having to retrain it.

2. API Integration

- **API (Application Programming Interface):** An API is like a waiter in a restaurant. Your application (the "customer") sends a request for data (an "order") to the API. The API takes that request to the model (the "kitchen"), gets the result, and brings it back to your application.
- **How it works:** A full-stack application can send user input (e.g., text from a form) to an API. The server running the API loads your saved. pkl model, uses it to make a prediction on the input, and sends the result back.

3. Live API Demo: How They Talk

APIs communicate using a set of rules. Key things to know are:

- **Method Types:** How you want to interact. Common ones are GET (to retrieve data) and POST (to send data to be processed).
- **Headers:** Extra information, like the format of the data.
- **Payload (or Body):** The actual data you're sending to the API (used with methods like POST).
- **Response:** The data the API sends back, often in a format called **JSON**.

1. Fetching Data from an API (Client-Side JS) Frontend

In modern JavaScript, you use the built-in **fetch API** to make network requests. It's the equivalent of Python's requests library and works in all modern browsers.

GET Request

This code fetches data from a specified URL. The fetch function returns a Promise, so we use .then() to handle the response once it arrives.

JavaScript

```
const getUserUrl = "https://jsonplaceholder.typicode.com/users/1";
```

```
fetch(getUserUrl)
 .then(response => {
  if (!response.ok) {
   throw new Error(` Network response was not ok: ${response.statusText}` );
  }
  return response.json();})
 .then(data => {
  console.log("GET Request Successful:");
  console.log(data); })
 .catch(error => {
  console.error("There was a problem with the fetch operation:", error); });
```

POST Request

To send data, you need to provide a second argument to fetch with an options object that specifies the method, headers, and body.

JavaScript

```
const postUrl = "https://jsonplaceholder.typicode.com/posts";

const newPostData = {
 title: 'My Awesome New Post',
 body: 'This is the content of my post, created from JavaScript!',
 userId: 10
};

const requestOptions = {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json'
 },
 body: JSON.stringify(newPostData)
};

fetch(postUrl, requestOptions)
 .then(response => response.json())
 .then(data => {
  console.log("POST Request Successful:");
  console.log(data);
 })
 .catch(error => {
  console.error("Error during POST request:", error);
 });
```

Assignments:

1. **Learn Python**
2. **Learn the Math Behind ML** YouTube playlist

   https://www.youtube.com/watch?v=GGPIjGyUQok&list=PLLUFZCmqicc-vkvCCax14T3OneUyEo2X0.

3. **Build a Small Full Stack App**

   build a small application with API calls between the frontend and backend.