Cloud Infotech Solutions

8688253560

Why We Need GitHub

1. **Version Control**: Tracks changes in the codebase, allowing developers to work on the same project without overwriting each other's work.
2. **Collaboration**: Facilitates teamwork by providing tools to merge changes, review code, and manage contributions.
3. **Backup**: Acts as a remote repository, ensuring code is safely stored and accessible from anywhere.
4. **Continuous Integration/Deployment**: Automates testing, builds, and deployments using GitHub Actions or other tools.
5. **Community Support**: Encourages open-source contributions, allowing developers to learn, share, and improve collectively.

Who Mostly Uses GitHub?

1. **Software Developers**: For managing and collaborating on projects.
2. **DevOps Engineers**: To integrate CI/CD workflows and manage infrastructure as code.
3. **Open Source Contributors**: To share and collaborate on community-driven projects.
4. **Tech Companies**: To host private repositories for internal projects.
5. **Students and Educators**: For learning and teaching version control and collaboration.

Benefits for Software Developers

- **Collaboration Tools**: Pull requests, code reviews, and issue tracking.
- **Showcase Work**: Acts as a portfolio to demonstrate expertise and contributions.
- **Integration**: Works with CI/CD pipelines, IDEs, and other tools.
- **Learning Resource**: Access to open-source projects and contributions.

Who Creates GitHub Repositories and Manages Them?

- **Repository Creators**: Typically project leads, developers, or DevOps engineers.
- **Management**: Includes creating branches, merging pull requests, and handling permissions.
- **Real-Time Updates**: Developers push changes, and GitHub reflects these updates for the team to view and collaborate.

Cloud Infotech Solutions

8688253560

GitHub Features

1. **Collaborators**: Team members added to a repository with specific permissions.
2. **Tokens**: Personal Access Tokens (PAT) used for secure authentication in API calls and CI/CD workflows.
3. **GitHub Actions**: Automates workflows such as testing, building, and deploying.
4. **Permissions**: Granular access control, including roles like owner, admin, write, and read.

Few Common Git Commands

- `git init`: Initializes a new Git repository.
- `git clone <URL>`: Clones an existing repository.
- `git add <file>`: Adds changes to the staging area.
- `git commit -m "message"`: Saves changes with a descriptive message.
- `git push`: Sends commits to the remote repository.
- `git pull`: Fetches and merges changes from the remote repository.
- `git branch`: Lists, creates, or deletes branches.
- `git merge <branch>`: Merges a branch into the current branch.
- `git status`: Displays the status of the working directory.
- `git log`: Shows the commit history.

Common Git Interview Questions

1. **What is Git and why is it used?**
2. **What is the difference between Git and GitHub?**
3. **Explain the concept of branching in Git.**
4. **What is a pull request and how is it handled?**
5. **How does Git handle merge conflicts?**
6. **What is the difference between `git fetch` and `git pull`?**
7. **What are Git tags, and how do you use them?**
8. **How do you revert a commit in Git?**
9. **Explain GitHub Actions and their use in CI/CD.**
10. **What is the purpose of a `.gitignore` file?**

Cloud Infotech Solutions

8688253560

====

General Notes:

what is git ?

local vcs tool , which track working area always and store data in special db with vc , linus troval who develops ,open source , free,light wieght , any platform

Types of scm ?
local(git) and Remote (github,bitbuckt)

stages of git :

WA -- SA -- Repository(local db in pc)commit area

install gitbash tool in windows /linux (git) or else **ide(integrated development environmet) example: vcs, pycham** tool integration

how to configure git global profile ?
Open gitbash , create directory project, cd project , then do these command
git config --global user.name "jhon"
git config --global user.email "jhone33@gmail.com"

how to initialize (convert regular to vcs)?

create folder(project2) switch project2, git init <--
it will create .git folder

Cloud Infotech Solutions

8688253560

how to push code from working area to staging ?
create some files , touch file1 file2 file3
note: important to configure user profile to git
git add . (filename) <-- (to add files from wA to SA)
---

how to check staging area (To check stage area from git to git hu b)

git status <--
 it will show these information ,..branch ,untracked or changes to be commited and files

how to commit (from stage area , to send to local repository db)
git commit -m "first commit"

how to check commit logs ?
git log

how to check commit contents ?
git show commit-id
===

How to push local repo/commited db to github of currect branch.
after commited source code to local repository then push local reposited files to github:
1.git remote add rb https://github.com/cloudinfotechsolutions/riyadhbank.git
2.create temp credentials token on github, copy

Cloud Infotech Solutions

8688253560

(goto github--account --settings- developer settings-- personal access token -- classic, generate new token (copy)
==== if you have already git credentials stored in windows then follow this steps or if not asking code then do these steps===
--- add windows credentails : generic --add
git:http://username@address
user : saicloud90@github.com
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
==========
3.git push -u rb master <--
it will ask token , paste token and sign in , see 100% push or not
===
cross check :
goto github -- click your created repo-- see ur files uploaded successfully.

differences :

- ☐ Push : To upload git local repo to remote github/bitbucket
  - ☐ Git push -u repo-origin branch :
- ☐ Pull command its like download from gh to git and merge updated data (it will commit)
  - ☐ Pull url
- ☐ Fetch : it is also download but does not merge into local repo. (it keeps into working dir)
  - ☐ Fetch url
- ☐ Clone : when new user joins on going project ,want to have all source code from gh to his/her local repo.
  - ☐ Clone url

—

default commit id is very large name which is not easy to memorize so tag can be use to assign custom name? example:

Cloud Infotech Solutions

8688253560

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

Tag : its label to commits rather than using default long commit id

Is it possible to do undo after adding to stage ?

Yes (git restore)

Is it possible to do undo from commit ?

yes (git revert)

git branching

what branching ?

its method of creating new space in existing repository to do additional task on the same project (add features, bugfix) ,if task is approved by teamlead then it need to merge to master .. to maintain mulitple .git database file in single directory (every branch will maintain its own .git repository)..

note:
when u create new branch it will copy master branch db

how to create new branch ?
git branch branch-name <-- (patch)

how to switch to one branch to another ?
git checkout branch-name <-- (patch)

How to see branches ?
git branch

how to delete branch ?

git branch -d branch-name <--

how to merge sub-branch to master ?
create branch (patch) , swtich to patch , create a file & edit data , git add , git commit... , then switch back to master , Git Merge Patch.

git init , config , add,status,commit,logs,push ,pull,fetch,clone,braching

Assignment :

diff clone , fetch , pull
why cherry-pick ,rebase ,stash
what is merge conflicts?
reset/
clone
fetch
pull
craete github account?
create repository (master , feature, bug-fix)

how to pull repo branch from git hub to git ?
git pull repo url (github url)
its a download of repository , use case : dev-team-A push code to github and other side Team-B pull the code to continue other modules..
merge conflicts
cherry-pick
===

ASSignmnet :

create 2 (directory in your laptop) using gitbash for 2 projects :

Cloud Infotech Solutions

8688253560

--1)rentcar -- 2)ecommerceapp
then initialize these directory , set git profile for both directory
create branches :
master (create files login,reg,search car, becomedirver,payments) commit,
then create new feature , bugfix in each repo
goto feature create some f1 f2 f3 files , commit
then merge into master branch .
goto bugfix , fix1 fix2 fix4... commit
push master .git to master branch of gh ,
push bugfix branch to bugfix in gh
--

create 2 acccount for github :
create repositoy rentcar and ecommerceapp
then create branch as u did local repo.
==
search simple java app github, simple nodejs app , python app github and do fork to your github account
---
reposiory names:
master, feature, bugfix
=====
invite collorabortor to your github account , invite give github account name , he/she need approve /accept then you can give some previlleges
note: realtime you create private account
===

how to push file larger than 100mb ?

larger than 100 mb how to push
git lfs install
git lfs track '*.nc'
git lfs track '*.csv'

It will create a file named .gitattributes, and now you can perform add & commit operations as normal.
Push the files to the LFS,
Push the pointers to GitHub.

git lfs push --all rb master

git push -u origin master

====

- Tag : its label to commits rather than using default long commit id
- Is it possible to do undo after adding to stage ?
- git restore --staged file-to-unstage.txt
- git reset file-to-unstage.txt
- git rm --cached file-to-unstage.txt
- Is it possible to do undo from commit ?
- Git log –-oneline
- Git revert commitid ß exit the vi, then commit and push your code.
- ===
- git reset is useful when you want to undo commits that have not been pushed to a remote repository.
- git reset [commit ID] particular commit
- Note :However, it's important to note that Git revert should only be used for small, isolated changes. If you need to undo multiple changes or large sections of code, it's generally better to use Git reset.

What is a Git Merge Conflict?

A **Git merge conflict** occurs when Git cannot automatically reconcile differences in the code between two branches being merged. This usually happens when:

1. Changes are made to the same line of a file in both branches.
2. A file is edited in one branch and deleted in another.
3. Overlapping changes are made to a section of code in two branches.

Git highlights these conflicts and leaves them unresolved so that the developer can manually decide how to proceed.

How to Resolve a Git Merge Conflict: Step-by-Step

Example Scenario:

1. You have two branches: `main` and `feature-branch`.
2. Both branches have conflicting changes in a file, `example.txt`.

Steps to Resolve the Conflict:

1. **Start the Merge Process**

Switch to the branch you want to merge into (e.g., `main`):
bash
Copy code
```
git checkout main
```

-

Merge the feature branch:
bash
Copy code
```
git merge feature-branch
```

-

Git Identifies the Conflict Git will stop the merge process and notify you about the conflict:
sql
Copy code
```
Auto-merging example.txt
CONFLICT (content): Merge conflict in example.txt
```

```
Automatic merge failed; fix conflicts and then commit
the result.
```

2.

Open the Conflicted File The conflicting file, `example.txt`, will include markers like this:
plaintext
Copy code

```
<<<<<<< HEAD
Line from the main branch
=======
Line from the feature branch
>>>>>>> feature-branch
```

3.
    - **`HEAD`** section: Content from the current branch (`main`).
    - **`feature-branch`** section: Content from the branch being merged.
4. **Resolve the Conflict** Decide how to handle the conflict:
    - Keep content from one branch.
    - Combine changes.
    - Manually rewrite the section.

Example resolution:
plaintext
Copy code

```
Combined content from both branches
```

5.

Mark the Conflict as Resolved After editing the file, mark it as resolved:
bash

Copy code
```
git add example.txt
```

6. **Complete the Merge** Commit the merge:
bash
Copy code
```
git commit -m "Resolved merge conflict in example.txt"
```

7. **Push the Changes** Push the resolved merge to the remote repository:
bash
Copy code
```
git push
```

8.

Example: Resolving a Conflict in `example.txt`

Before Resolution:
plaintext
Copy code
```
<<<<<<< HEAD
Line from the main branch
=======
Line from the feature branch
>>>>>>> feature-branch
```

After Resolution:
plaintext

Copy code
```
Final merged line (e.g., a combination of both lines)
```

Commands:

bash
Copy code
```
git add example.txt
git commit -m "Resolved conflict in example.txt"
git push
```

Tips for Avoiding Merge Conflicts:

- **Pull frequently**: Regularly pull changes from the remote repository to keep your branch updated.
- **Communicate**: Coordinate with team members about significant changes.
- **Work on smaller files**: Divide large files into smaller modules to reduce conflicts.

Merge conflicts are a normal part of collaborative development, and resolving them effectively ensures a smooth workflow.

What are GitHub Collaborators?

GitHub collaborators are individuals who are granted access to a private or public repository. They can contribute by cloning, pushing, pulling, and reviewing code depending on their permissions. Collaboration is key when multiple developers are working on the same project.

Cloud Infotech Solutions

8688253560

Steps to Use GitHub Collaborators for Multiple Developers

Scenario: You and your team are building a web application using GitHub.

---

1. Set Up a Repository

1. Create a new repository or use an existing one:
    - Go to GitHub.
    - Click on **New Repository**.
    - Name it and initialize it with a `README.md` (optional).

Clone the repository locally (for everyone involved):
bash
Copy code

```
git clone
https://github.com/username/repository-name.git
```

2.

---

2. Add Collaborators

1. Go to the repository on GitHub.
2. Navigate to the **Settings** tab.
3. Click on **Collaborators and teams** (under "Access").
4. Add collaborators by searching for their GitHub usernames or email addresses.
5. Assign the appropriate **role** (e.g., Read, Write, Admin).

---

3. Workflow for Multiple Developers

Step 1: Create a Branch for Each Feature

Each developer should work on their own branch:

bash
Copy code
```
git checkout -b feature-branch-name
```

Step 2: Commit Changes

After editing files, commit the changes:

bash
Copy code
```
git add .
git commit -m "Add feature or fix bug"
```

Step 3: Push Branch to GitHub

Push your branch to the remote repository:

bash
Copy code
```
git push origin feature-branch-name
```

Step 4: Create a Pull Request (PR)

1. Go to the repository on GitHub.
2. Click on the **Pull Requests** tab.
3. Click **New Pull Request** and select your branch.
4. Assign reviewers (other collaborators) to review your code.

Step 5: Review and Merge the PR

- Collaborators review the code and leave comments or approve it.

- Once approved, merge the pull request into the `main` or another target branch.

---

4. Example Workflow

1. Developer 1: Alice

- Creates a branch: `feature-login-page`.
- Adds login functionality.

Pushes the branch:
bash
Copy code
```
git push origin feature-login-page
```

-
- Opens a pull request.

2. Developer 2: Bob

- Reviews Alice's PR.
- Suggests changes or approves it.
- Merges the PR into `main`.

3. Developer 3: Charlie

Pulls the latest `main` branch:
bash
Copy code
```
git pull origin main
```

-
- Starts work on their feature branch: `feature-signup-page`.

---

5. Best Practices

1. **Branch Naming:** Use descriptive names like `feature-login`, `bugfix-profile`.

Pull Frequently: Regularly pull changes from the `main` branch to avoid conflicts.
bash
Copy code
```
git pull origin main
```

2.
3. **Review Code Thoroughly:** Use PR reviews to maintain code quality.
4. **Use Issues:** Track tasks using GitHub Issues for better collaboration.

Summary

By using GitHub collaborators and following a branch-based workflow, multiple developers can efficiently work on the same repository while maintaining code quality and avoiding conflicts. This ensures a smooth and collaborative development process.

GitHub Branch Protection and Copilot Use Case for DevOps Engineers

---

1. GitHub Branch Protection

Branch protection is a feature in GitHub that prevents direct changes to important branches (e.g., `main` or `production`) without meeting specified conditions. It ensures code quality, security, and reduces the risk of accidental changes.

Why Branch Protection is Useful for DevOps Engineers?

- Prevents accidental deployment of untested or insecure code.
- Enforces code reviews and automated checks (e.g., CI/CD pipelines).
- Ensures branch stability in critical environments like production or staging.

Key Features of Branch Protection

1. **Require Pull Request Reviews:**
   - Ensure that every change is reviewed by peers before merging.
2. **Require Status Checks:**
   - Enforce successful CI/CD pipeline checks (e.g., build, tests).
3. **Restrict Pushes:**
   - Only specific users or teams can push to the branch.
4. **Require Signed Commits:**
   - Enforce cryptographic signatures for commits to verify the authenticity of contributors.

Example: Setting Up Branch Protection for `main`

1. Navigate to your repository on GitHub.
2. Go to **Settings > Branches**.
3. Under **Branch Protection Rules**, click **Add Rule**.
4. Set the rule for the `main` branch:
   - **Require pull request reviews** with at least 2 reviewers.
   - **Require status checks to pass** (e.g., CI/CD pipeline).
   - Enable **Restrict who can push** to the branch.
5. Save changes.

Use Case for DevOps Engineers

Scenario: Managing Infrastructure as Code (IaC)

1. DevOps engineers manage Terraform or Ansible scripts in a GitHub repository.
2. The `main` branch contains the production-ready configuration files.

Workflow with Branch Protection:

1. **Feature Branches for Changes:**
   Engineers create branches like `feature-update-aws-vpc` for updates.
2. **Pull Request Reviews:**
   Before merging, pull requests must pass:
   - Peer reviews.
   - Automated checks (e.g., `terraform plan` output validation).
3. **Restrict Pushes:**
   Only senior DevOps engineers or leads can merge changes into `main`.

---

2. GitHub Copilot

GitHub Copilot is an AI-powered code suggestion tool that helps developers write code faster by offering context-aware suggestions.

Why Use Copilot for DevOps?

- Speeds up writing automation scripts for tools like Terraform, Ansible, Kubernetes, and CI/CD pipelines.
- Assists in creating boilerplate code for configuration files and deployment scripts.
- Reduces human errors by suggesting syntax and best practices.

Examples of Copilot Use for DevOps Engineers

1. Writing Kubernetes YAML Manifests

With Copilot, you can quickly draft complex Kubernetes manifests:

yaml
Copy code

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: nginx
          ports:
            - containerPort: 80
```

- Copilot auto-suggests the fields and values based on the context.

2. Automating Infrastructure with Terraform

You need a Terraform script to create an AWS S3 bucket:

hcl

Copy code

```
resource "aws_s3_bucket" "example" {
  bucket = "my-example-bucket"
  acl    = "private"
}
```

- Copilot can autocomplete resource attributes and suggest common configurations.

3. Writing CI/CD Pipeline Scripts

When creating a GitHub Actions workflow for deployment:

yaml
Copy code

```
name: Deploy to AWS
on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Configure AWS CLI
        uses: aws-actions/configure-aws-credentials@v2
        with:
```

```
          aws-access-key-id: ${{
secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{
secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Deploy using Terraform
        run: terraform apply -auto-approve
```

- Copilot suggests entire sections of the YAML file based on the project context.

Combined Use Case: Branch Protection + Copilot

Scenario: Secure Infrastructure Updates

1. **Branch Protection:**
   - Protect the `main` branch to prevent direct changes to Terraform files.
   - Require CI/CD pipelines (e.g., GitHub Actions) to validate Terraform scripts.
2. **Copilot Assistance:**
   - Use Copilot to draft and improve Terraform or Ansible scripts.
   - Generate efficient Kubernetes manifests or CI/CD workflows.
3. **Workflow:**
   - DevOps engineers work on feature branches using Copilot for code suggestions.
   - Push changes and open a pull request.
   - Branch protection ensures code reviews and pipeline validations before merging.

Benefits for DevOps Engineers

- **Improved Collaboration:** Branch protection enforces team standards.
- **Faster Development:** Copilot reduces time spent on repetitive tasks.
- **Enhanced Security:** Prevent unauthorized or untested changes in critical branches.
- **Consistency:** AI-driven suggestions promote best practices and reduce errors.

By integrating branch protection and GitHub Copilot, DevOps engineers can maintain a secure, efficient, and automated workflow.

====

What is a GitHub App?

A **GitHub App** is an integration that connects tools, services, or custom workflows directly to GitHub. GitHub Apps allow developers to automate tasks, integrate external services, and extend GitHub's functionality to suit specific needs.

GitHub Apps can:

- Listen to events (e.g., pull requests, issues, pushes).
- Perform actions (e.g., post comments, label issues, merge pull requests).
- Access specific repositories or organizations with granular permissions.

Key Features of GitHub Apps

1. **Granular Permissions:**
    - Apps can request specific permissions for read/write access to certain resources like issues, pull requests, or repositories.
2. **Event-Driven:**
    - GitHub Apps respond to webhook events (e.g., when a pull request is opened).
3. **Scoped Access:**
    - Installed per repository or organization, limiting scope to only what is required.
4. **API Access:**
    - Apps can use the GitHub REST or GraphQL APIs to automate workflows.

Use Cases for GitHub Apps

1. **CI/CD Pipelines:**
    - Automate builds, tests, and deployments when code is pushed to the repository.
2. **Code Quality Checks:**
    - Integrate tools like ESLint, Prettier, or SonarQube to review code automatically.
3. **Issue and PR Management:**
    - Automatically label, assign, or comment on issues and pull requests.
4. **Custom Workflows:**
    - Create custom integrations for organization-specific processes like approval workflows.
5. **Security Automation:**
    - Identify vulnerabilities in dependencies and enforce compliance policies.

Cloud Infotech Solutions

8688253560

How to Use GitHub Apps

1. Install a GitHub App

1. Go to the GitHub Marketplace.
2. Search for an app (e.g., Dependabot, Codecov, or Actions apps).
3. Click on the app and select **Install**.
4. Choose the repositories or organizations where the app should be installed.

---

2. Create a Custom GitHub App

Step 1: Register the App

1. Navigate to **Settings > Developer Settings > GitHub Apps**.
2. Click **New GitHub App**.
3. Fill in details like:

Best Practices for Using GitHub Apps

1. **Minimal Permissions:**
   - Only request the permissions necessary for the app's functionality.
2. **Security:**
   - Use secure storage for private keys and tokens.
3. **Error Handling:**
   - Implement robust error handling for webhook events and API calls.
4. **Logging and Monitoring:**
   - Track app activity and webhook delivery for debugging and performance optimization.

Popular GitHub Apps for DevOps

1. **Dependabot:**
    - Automatically updates dependencies to the latest versions.
2. **CodeQL:**
    - Performs static analysis to detect vulnerabilities.
3. **Codecov:**
    - Tracks code coverage during testing.
4. **Jenkins X:**
    - Integrates Jenkins pipelines with GitHub.
5. **Snyk:**
    - Scans for vulnerabilities in open-source dependencies.

GitHub Apps are versatile tools for automating workflows, improving productivity, and enhancing collaboration, making them essential for modern DevOps practices.