**DevOps**

Dev (Development) + Ops (Operation), DevOps is not a single tool or technology basically a way of doing things or you can say a series of best practices.

Best practices means: - Building and deploying applications to server and managing it and monitoring it after words.

**Plan ----> Code ---> Tested --- > Security --- > Build --- > Deploy --- > Monitor**

**Dev: -** Write a code and test in local machine.

**OPS: -** Plan ----> Code ---> Tested --- > Security --- > Build --- > Deploy --- > Monitor

**Why do we need DevOps:-**

* **Streamlined: -** We have very clear understanding what stages we go through in order to build and deploy the application.
* **Managed: -** Before DevOps there was no proper way to track it, different person write different stages now we create a pipeline and all stages in one place.

**Key Concepts of DevOps: -**

1. **CI/CD (Continuous Integration and Continuous Deployment/Delivery) :** It is a backbone of DevOps

**Scenario:** A company wants to ensure that any code changes made by developers are automatically tested and deployed.

**CI: -** Once Developer push their code that code should be tested then security Check then application build and it should be ready to deploy called Continuous Integration.

**CD (Continuous Deployment): -** Once CI is completed, automatically triggered called continuous Deployment.

**CD (Continuous Delivery): -** Once CI is completed, we need manually triggered called continuous Deployment.

**Detailed Flow:**

* A developer writes new code and commits it to the version control system (e.g., Git).
* The CI server detects the commit and triggers a build.
* Automated tests are run to ensure the new code doesn't break existing functionality.
* If tests pass, the code is automatically deployed to a staging environment.
* Further integration tests are performed in staging.
* If all checks pass, the code is deployed to production.

1. **Infrastructure as Code (IaC) :-**

**Scenario:** An organization wants to manage and provision its infrastructure using code rather than manual processes.

**Practice:** Using tools like Terraform or AWS Cloud Formation, the infrastructure is defined in code, allowing for version control, reproducibility, and automation.

**Detailed Flow:**

* Define infrastructure requirements in a configuration file (e.g., Terraform .tf files).
* Store these configuration files in a version control system.
* When changes are needed, update the configuration files and commit the changes.
* Use a CI/CD pipeline to automatically apply these changes to the cloud environment, ensuring the infrastructure is always up-to-date and consistent with codebase.

1. **Configuration Management : -**

**Scenario:** An organization needs to ensure that its servers are configured consistently and securely.

**Practice:** Use configuration management tools like Ansible, Puppet, or Chef to automate the setup and maintenance of server configurations.

**Detailed Flow:**

* Write configuration scripts (e.g., Ansible playbooks) that describe how to set up the servers (installing software, configuring services, etc.).
* Store these scripts in a version control system.
* Use a CI/CD pipeline to apply these configurations to servers automatically.
* Periodically run these scripts to ensure servers remain in the desired state, and any drift from the configuration is corrected.

1. **Microservices Architecture : -**

**Scenario:** An e-commerce company wants to break down its monolithic application into smaller, independently deployable services.

**Practice:** Develop each microservice to handle specific business functions (e.g., user authentication, product catalog, order processing) and deploy them independently.

**Detailed Flow:**

* Decompose the monolithic application into microservices, each with its own codebase and database.
* Use containerization (e.g., Docker) to package each microservice.
* Orchestrate these containers using Kubernetes or Docker Swarm for automated deployment, scaling, and management.
* Implement service discovery and API gateways to manage communication between microservices.
* Use CI/CD pipelines to independently deploy and update each microservice.

1. **Monitoring and Logging : -**

**Scenario:** A company needs to ensure its applications are performing well and to quickly identify and troubleshoot issues.

**Practice:** Implement comprehensive monitoring and logging using tools like Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), or Splunk.

**Detailed Flow:**

* Instrument the application and infrastructure to emit metrics and logs.
* Use monitoring tools (e.g., Prometheus) to collect and store these metrics.
* Use visualization tools (e.g., Grafana) to create dashboards for real-time monitoring.
* Set up alerting rules to notify the team when certain thresholds are exceeded (e.g., high CPU usage, errors).
* Use logging tools (e.g., ELK Stack) to aggregate and analyze logs for troubleshooting and auditing purposes.

**Benefits of DevOps**

**Faster Time to Market:** Automated processes and continuous delivery pipelines speed up the release cycle.

**Improved Collaboration:** Breaking down silos fosters better communication and collaboration between development and operations teams.

**Increased Efficiency:** Automation reduces manual tasks, allowing teams to focus on higher-value work.

**Enhanced Reliability:** Continuous testing and monitoring ensure that issues are detected and resolved quickly.

**Scalability:** Infrastructure as Code and container orchestration allow for easy scaling of applications to meet demand.

**Tools Commonly Used in DevOps**

**CI/CD:** Jenkins, GitLab CI, CircleCI, Travis CI, GitHub Actions

**IaC:** Terraform, AWS CloudFormation, Ansible, Chef, Puppet

**Configuration Management:** Ansible, Puppet, Chef, SaltStack

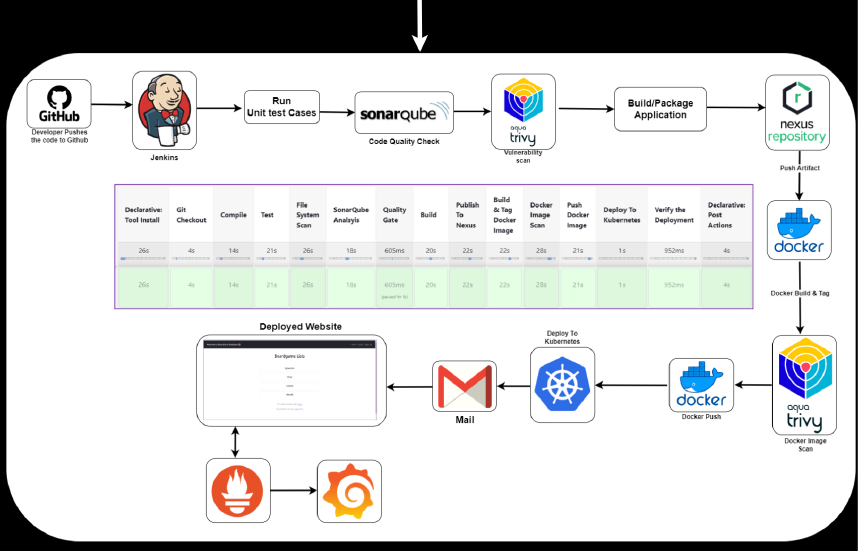
**Monitoring:** Prometheus, Grafana, Nagios, Zabbix

**Logging:** ELK Stack, Splunk, Fluentd, Graylog

**Containerization:** Docker

**Orchestration:** Kubernetes, Docker Swarm, OpenShift

**Project Workflow Documentation: -**

****

* Once Developer confirms code is correct then pushed into GitHub repository.
* Once repo is pushed, automatically pushing of the code will be triggering a pipeline CI/Cd.
* As soon as pipeline is triggered first thing that is going to happen is compilation of the code (in your code might have some issue in your syntax like} ; is missed), if we receive any error then inform to developer to check the code.
* Once compilation is successfully done then to run unit test cases to test different unit of application.
* Once Unit test is completed then move to sonarcube (code quality check and code coverage) like less number of bugs and less number of vulnerabilities and % of code covered by test cases.
* Once sonarcube is done then move to trivy( it is a security tool perform certain type of checks like dependencies checks)
* Once trivy is done then move to maven (build the code), after build the code it will generate in jar, war format called artifact.
* One build process completed then we need to store this different jar file in Nexus repository.
* Then we need to write a file in docker and one docker image build it will scan by trivy, then we need to store docker image (Push docker image).
* Now we need to prepare the yaml manifest file (like RAM, CPU and which docker image should be used for deployment), after we are going to deploy the application on kuberneties server using this yaml manifest file, also need to secure kuberneties server with kubeaudit.
* Once deployment is done we need to write a pipeline to receive a notification that our deployment is successful.
* Now we can access our application is browser and also set a monitoring tool to monitor the application.

**Detailed Workflow**

1. **Client Feature Request (Jira Ticket)**

**Purpose:** To capture and track the client's request for a new feature in the application.

1. **Discussion and Assignment**

**Purpose:** To discuss the feature request, refine the requirements, and assign the task to a developer.

1. **Development and Local Testing**

**Purpose:** To develop the new feature and ensure it functions correctly in a local development environment.

**Local Testing**: Run the application locally to test the new feature.

1. **Code Management (Feature Branch and Pull Request)**

**Purpose:** To manage the feature code in the version control system and prepare it for integration into the main development branch.

1. **CI/CD Pipeline Execution**

**Purpose:** To automate the build, test, and deployment processes using a CI/CD pipeline.

**Pipeline Stages:**

* Git Checkout: Fetching the latest source code from the repository.
* Fresh Dependency Installation: Installing all required dependencies from scratch.
* Executing Test Cases: Running automated tests to validate the code.
* Performing SonarQube Analysis: Conducting static code analysis to ensure code quality.
* Trivy Vulnerability Scanning: Scanning for vulnerabilities in the codebase.
* Package/Build Application: Compiling and packaging the application.
* Publish Artifacts to Nexus: Uploading built artifacts to Nexus for version control and management.
* Build Docker Images: Creating Docker images for the application.
* Scan Docker Images Using Trivy: Ensuring Docker images are free from vulnerabilities.
* Deploy Application to Kubernetes Cluster: Deploying the application using Kubernetes manifest files.
* Perform Penetration Testing Using OWASP ZAP: Conducting security testing on the deployed application.
* Post Actions (Send Mail Notifications): Sending notifications about the pipeline status and results.

**Detailed Pipeline Script Example:**

pipeline {

agent any

stages {

stage('Git Checkout') {

steps {

git branch: 'develop', url: 'https://github.com/your-repo/project.git'

}

}

stage('Install Dependencies') {

steps {

sh 'rm -rf node\_modules'

sh 'npm install'

}

}

stage('Execute Test Cases') {

steps {

sh 'npm test'

}

post {

always {

junit 'test-results.xml'

}

}

}

stage('SonarQube Analysis') {

steps {

withSonarQubeEnv('SonarQube') {

sh 'sonar-scanner'

}

}

}

stage('Trivy Vulnerability Scanning') {

steps {

sh 'trivy fs --exit-code 1 --severity HIGH,CRITICAL .'

}

}

stage('Build Application') {

steps {

sh 'npm run build'

}

}

stage('Publish to Nexus') {

steps {

nexusPublisher nexusInstanceId: 'nexus', nexusRepositoryId: 'releases', packages: [

[$class: 'MavenPackage', mavenAssetList: [[classifier: '', extension: 'jar', filePath: 'target/app.jar']], mavenCoordinate: [artifactId: 'app', groupId: 'com.example', version: '1.0.0']]

]

}

}

stage('Build Docker Image') {

steps {

script {

def app = docker.build("your-repo/app:${env.BUILD\_NUMBER}")

}

}

}

stage('Scan Docker Image') {

steps {

sh 'trivy image --exit-code 1 --severity HIGH,CRITICAL your-repo/app:${env.BUILD\_NUMBER}'

}

}

stage('Deploy to Kubernetes') {

steps {

kubernetesDeploy configs: 'k8s/', kubeconfigId: 'kubeconfig'

}

}

stage('OWASP ZAP Penetration Testing') {

steps {

sh 'zap-baseline.py -t http://your-app-url -r zap-report.html'

}

post {

always {

archiveArtifacts artifacts: 'zap-report.html'

}

}

}

stage('Post Actions') {

steps {

mail to: 'team@example.com',

subject: "Pipeline ${currentBuild.fullDisplayName} - ${currentBuild.currentResult}",

body: "Pipeline ${currentBuild.fullDisplayName} finished with status: ${currentBuild.currentResult}. Please check the Jenkins console output for more details."

}

}

}

}

**Deployment Strategies in DevOps**

Deployment strategies are essential for ensuring that new software updates are delivered to users with minimal disruption. Different strategies can be chosen based on the specific needs and constraints of a project. Below are detailed descriptions of various deployment strategies with examples.

1. **Recreate Deployment Strategy**

**Description:** This is the simplest deployment strategy where the existing version of the application is stopped and replaced with the new version. It involves downtime since the old version is terminated before the new one starts.

**Use Case:** Suitable for non-critical applications where a brief downtime is acceptable.

**Example:**

**Scenario:** Deploying a new version of a web application.

* 1. Process: Stop the running application.
  2. Deploy the new version.
  3. Start the new version.

1. **Rolling Update Deployment Strategy**

**Description:** This strategy updates instances of the application incrementally. New instances are started before the old ones are terminated, ensuring that the application remains available during the deployment.

**Use Case:** Ideal for applications where zero downtime is crucial.

**Example:**

**Scenario:** Deploying a new version of a microservice.

**Process:**

Start a new instance of the application.

Gradually route traffic to the new instance.

Terminate the old instance once the new instance is running correctly.

1. **Blue-Green Deployment Strategy**

**Description:** This strategy involves maintaining two environments, Blue and Green. The current version runs on the Blue environment, while the new version is deployed to the Green environment. Once the Green environment is verified, traffic is switched from Blue to Green.

**Use Case:** Suitable for applications requiring minimal downtime and easy rollback capabilities.

**Example:**

**Scenario:** Deploying a new version of a banking application.

**Process:** Deploy the new version to the Green environment.

* 1. Run tests on the Green environment.
  2. Switch traffic from Blue to Green.
  3. If issues are found, switch traffic back to Blue.

1. **Canary Deployment Strategy**

**Description:** In this strategy, a new version is deployed to a small subset of users before gradually rolling it out to the entire user base. This allows for monitoring and verification with minimal risk.

**Use Case:** Suitable for applications where gradual rollout and real-time feedback are critical.

**Example:**

**Scenario:** Deploying a new version of an e-commerce application.

**Process:**

Deploy the new version to a small percentage of servers.

Monitor the performance and gather feedback.

Gradually increase the percentage of servers running the new version.

1. **Shadow Deployment Strategy**

**Description:** In this strategy, the new version runs in parallel with the current version, but the traffic is duplicated and sent to both versions. The new version does not affect the actual user traffic but is monitored to ensure it behaves as expected.

**Use Case:** Ideal for validating new features and performance without impacting the live environment.

**Example:**

**Scenario:** Deploying a new analytics service.

* 1. **Process:**
  2. Deploy the new version alongside the current version.
  3. Duplicate and route traffic to both versions.
  4. Monitor the performance and behavior of the new version.

**Environments in DevOps**

In DevOps, environments refer to different stages or settings where code is developed, tested, and deployed. Each environment serves a specific purpose in the software development lifecycle. Here is a detailed explanation of each environment and the promotion process:

1. **Development (DEV) Environment**

**Purpose:** The DEV environment is where developers write, debug, and initially test their code. It’s used for active development and experimentation.

* 1. **Characteristics:**
  2. Frequent code changes and updates.
  3. Limited or no restrictions on who can access and modify the code.
  4. Basic testing, often unit tests, are performed here.
  5. **Promotion Criteria:** Code is promoted from DEV to QA when it has passed initial testing and is deemed stable enough for more rigorous testing.

1. **Quality Assurance (QA) Environment**

**Purpose:** The QA environment is used for extensive testing, including functional, integration, and performance testing. QA engineers validate that the application works as expected.

* 1. **Characteristics:**
  2. More stable than the DEV environment.
  3. Automated and manual tests are run here.
  4. Simulates the production environment closely.
  5. **Promotion Criteria:** Code is promoted from QA to PPD when it passes all the tests and meets the predefined acceptance criteria.
  6. **3. Pre-Production (PPD) Environment**

**Purpose:** The PPD environment, also known as staging, is a near-identical replica of the production environment. It is used for final testing and validation before the code goes live.

* 1. **Characteristics:**
  2. Mimics the production environment in terms of configuration, data, and load.
  3. User acceptance testing (UAT) and final performance testing occur here.
  4. Limited access to ensure stability.
  5. **Promotion Criteria:** Code is promoted from PPD to PROD when it passes final acceptance tests and stakeholders approve it for release.

1. **Production (PROD) Environment**

**Purpose:** The PROD environment is where the application is live and accessible to end-users. It’s the final environment where the software operates under real-world conditions.

* 1. **Characteristics:**
  2. Highly stable and secure.
  3. Monitored for performance, availability, and security.
  4. All changes are strictly controlled and reviewed.
  5. **Promotion Criteria:** Code is deployed to PROD after thorough testing and approval from relevant stakeholders.

1. **Disaster Recovery (DR) Environment**

**Purpose:** The DR environment is a backup environment used to recover from catastrophic failures in the PROD environment. It ensures business continuity and data recovery.

* 1. **Characteristics:**
  2. Kept in sync with the PROD environment, often in a different geographical location.
  3. Includes all necessary data and configurations to restore service quickly.
  4. Regularly tested to ensure it can handle the switch from PROD in case of an emergency.
  5. **Promotion Criteria:** The DR environment is not typically part of the regular promotion process but is updated to mirror the PROD environment. It is activated when a disaster occurs.

==================🡺==========================🡺================

**GIT**

**Introduction to Git:** Git is a distributed version control system used for tracking changes in source code during software development. It enables multiple collaborators to work on the same codebase simultaneously while maintaining a history of changes, making it easier to manage, collaborate, and track progress.

**Centralized vs. Distributed**

* **SVN:** Centralized version control system. All version history is stored on a central server. Developers check out files from the server, make changes, and commit them back to the server.

* **Git:** Distributed version control system. Each developer has a complete copy of the repository, including its entire history. Changes are committed locally, and then pushed to a central repository

**Git, GitLab and Github**

Git is a tool that used to manage the code (like comment, pull, and push)

Git lab and Github basically hosting the repository, these are simply used to store the code.

**Key Concepts:**

1. **Repository:** A repository (repo) is a collection of files and directories along with their complete history of changes.
2. **Branches:** A branch is a separate line of development that allows you to work on features, bug fixes, or experiments without affecting the main codebase. The default branch is usually called "master" or "main."
3. **Commit:** A snapshot of the changes in the repository
4. **Merge:** Merging combines changes from one branch into another.
5. **Pull Request (PR):** A method of submitting contribution to a project.
6. **Staging Area:** A place where you can group changes to prepare them for a commit.
7. **Master/Main:** The default primary branch in Git.
8. **Head:** A pointer to the current branch latest commit.
9. **Clone:** A copy of repository
10. **Fork:** A personal copy of someone’s repository.
11. **Rebase:** Reapplying commits on top of another base tip.
12. **Conflict:** Occurs when changes from different commits interfere with each other.
13. **Remote:** A version of your project that is hosted on the internet or network
14. **Fetch:** Download commit, files from a remote repository
15. **Pull:** Fetches and merge changes from a remote repository to your working directory.
16. **Push:** Upload local changes to remote repository.
17. **Tag:** A reference to a specific point in the Git history.

**How to Generate Token in Git:-**

Go to your github account name --> Setting --> Developer setting-- > Personal Access token -- > Token classic -- > Generate new token-- > Generate new token classic -- > Name and period.

ghp\_3s7IVYbE8NImAF4hr1Z1RfCpORLfK31HZbd1

**How to clone, add, commit and push:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code

$ git clone <https://github.com/flexis007/Batch.git>

Cloning into 'Batch'...

remote: Enumerating objects: 4, done.

remote: Counting objects: 100% (4/4), done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

Receiving objects: 100% (4/4), done.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code

$ ls

**Batch**/

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code

$ cd Batch/

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ touch 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git status

On branch main

Your branch is up to date with 'origin/main'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

1.txt

nothing added to commit but untracked files present (use "git add" to track)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git add 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git status

On branch main

Your branch is up to date with 'origin/main'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git commit -m "V1" 1.txt

Author identity unknown

\*\*\* Please tell me who you are.

Run

git config --global user.email "you@example.com"

git config --global user.name "Your Name"

to set your account's default identity.

Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'Farhan@DESKTOP-IR4NRGD.(none)')

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git config --global user.email "imti5791@gmail.com"

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git config --global user.name "Imtiyaj"

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git commit -m "V1" 1.txt

[main b95b0d1] V1

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git push origin main

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

To https://github.com/flexis007/Batch.git

ab38d8a..b95b0d1 main -> main

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

**Git Branch:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git branch

\* main

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git branch develpo

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git checkout develpo

Switched to branch 'develpo'

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$ ls

1.txt 2.txt LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$ git push origin develpo

Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

remote:

remote: Create a pull request for 'develpo' on GitHub by visiting:

remote: https://github.com/flexis007/Batch/pull/new/develpo

remote:

To https://github.com/flexis007/Batch.git

\* [new branch] develpo -> develpo

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$

**Delete Branch:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git branch -D new

Deleted branch new (was 1e25cd2).

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$

**Checking Logs:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$ git log

commit f191cfbcdf74a829eb4bcb0eab005bdcd7ab47b6 (**HEAD** -> **develpo**, **origin/main**, **origin/HEAD**, **main**)

Author: flexis007 <158850838+flexis007@users.noreply.github.com>

Date: Thu Aug 15 16:29:07 2024 +0530

Create 2.txt

commit b95b0d15d582d2a0e967c0f0654eb9413fb8e9e9

Author: Imtiyaj <imti5791@gmail.com>

Date: Thu Aug 15 16:20:54 2024 +0530

V1

commit ab38d8ac96d6c81d3c6d0ca451a9cf5204485f09

Author: flexis007 <158850838+flexis007@users.noreply.github.com>

Date: Thu Aug 15 16:01:22 2024 +0530

Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$ git log --oneline

f191cfb (**HEAD** -> **develpo**, **origin/main**, **origin/HEAD**, **main**) Create 2.txt

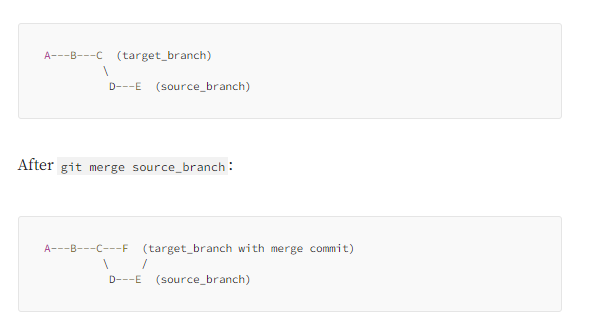
b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (develpo)

$

**Git Merge VS Git Rebase:**

****

**Merge Example:** Create a new branch from github portal

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git pull

From https://github.com/flexis007/Batch

\* [new branch] new -> origin/new

Already up to date.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git checkout new

Switched to a new branch 'new'

branch 'new' set up to track 'origin/new'.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ touch abc

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git add abc

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git commit -m "merge" abc

[new f8db722] merge

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 abc

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git push origin new

Enumerating objects: 3, done.

Counting objects: 100% (3/3), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (2/2), 246 bytes | 246.00 KiB/s, done.

Total 2 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/flexis007/Batch.git

f191cfb..f8db722 new -> new

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git log --oneline

f8db722 (**HEAD** -> **new**, **origin/new**) merge

f191cfb (**origin/main**, **origin/HEAD**, **develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git checkout main

Switched to branch 'main'

Your branch is ahead of 'origin/main' by 2 commits.

(use "git push" to publish your local commits)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git log --oneline

df05b90 (**HEAD** -> **main**, **origin/develop**, **develop**) V4

45a17b8 V3

f191cfb (**origin/main**, **origin/HEAD**, **develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git merge new

Merge made by the 'ort' strategy.

abc | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 abc

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git log --oneline

358dc95 (**HEAD** -> **main**) Merge branch 'new' ##(Merge shows)

f8db722 (**origin/new**, **new**) merge

df05b90 (**origin/develop**, **develop**) V4

45a17b8 V3

f191cfb (**origin/main**, **origin/HEAD**, **develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$

**Git Rebase Example:**

****

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git checkout new

Switched to branch 'new'

Your branch is up to date with 'origin/new'.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ touch xyz

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ touch qwe

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git add xyz

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git commit -m "new" xyz

[new c8ee029] new

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 xyz

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git add qwe

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git commit -m "new1" qwe

[new 1e25cd2] new1

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 qwe

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git push origin new

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Delta compression using up to 4 threads

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 419 bytes | 419.00 KiB/s, done.

Total 4 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (2/2), completed with 1 local object.

To https://github.com/flexis007/Batch.git

f8db722..1e25cd2 new -> new

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git log --oneline

1e25cd2 (**HEAD** -> **new**, **origin/new**) new1

c8ee029 new

f8db722 merge

f191cfb (**develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (new)

$ git checkout main

Switched to branch 'main'

Your branch is up to date with 'origin/main'.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git log --oneline

358dc95 (**HEAD** -> **main**, **origin/main**, **origin/HEAD**) Merge branch 'new'

f8db722 merge

df05b90 (**develop**) V4

45a17b8 V3

f191cfb (**develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git rebase new

Successfully rebased and updated refs/heads/main.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git log --oneline

822e732 (**HEAD** -> **main**) V4

ac3ce75 V3

1e25cd2 (**origin/new**, **new**) new1

c8ee029 new

f8db722 merge

f191cfb (**develpo**) Create 2.txt

b95b0d1 V1

ab38d8a Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$

**Git Stash/pop:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ touch 9.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ vim 9.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git add 9.txt

warning: in the working copy of '9.txt', LF will be replaced by CRLF the next time Git touches it

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git commit -m "stash" 9.txt

warning: in the working copy of '9.txt', LF will be replaced by CRLF the next time Git touches it

[main ecabdd7] stash

1 file changed, 2 insertions(+)

create mode 100644 9.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ vim 9.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ cat 9.txt

This is stash

jsnflksnflkfnfrjg;otjg;lmbdf;lbv

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git stash

warning: in the working copy of '9.txt', LF will be replaced by CRLF the next time Git touches it

Saved working directory and index state WIP on main: ecabdd7 stash

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git push origin main

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 267 bytes | 267.00 KiB/s, done.

Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/flexis007/Batch.git

0f4605f..ecabdd7 main -> main

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ git stash pop

On branch main

Your branch is up to date with 'origin/main'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: 9.txt

modified: xyz

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (60eaec3187b2d91bfe8c124225c1e979e7ab5775)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$ cat 9.txt

This is stash

jsnflksnflkfnfrjg;otjg;lmbdf;lbv

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/code/Batch (main)

$

**Git Cherrypick:**

Git cherry-pick is a command that allows you to apply a specific commit from one branch to another. It's like picking a single cherry from a tree and placing it on another

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ ls

LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ touch 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ touch 2.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ touch 3.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git add 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git commit -m "1" 1.txt

[main c778dc1] 1

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 1.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git add 2.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git commit -m "2" 2.txt

[main 208f293] 2

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 2.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git add 3.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git commit -m "3" 3.txt

[main 6853308] 3

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 3.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git log --oneline

6853308 (**HEAD** -> **main**) 3

208f293 2

c778dc1 1

2314c06 m1

9b1ba1e my1

53f254e (**test**) newver

52c7cfe (**origin/main**, **origin/HEAD**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git checkout test

Switched to branch 'test'

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (test)

$ git log --oneline

53f254e (**HEAD** -> **test**) newver

52c7cfe (**origin/main**, **origin/HEAD**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (test)

$ git cherry-pick 208f293

[test 2aad7dc] 2

Date: Thu Aug 15 19:16:17 2024 +0530

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 2.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (test)

$ git log --oneline

2aad7dc (**HEAD** -> **test**) 2

53f254e newver

52c7cfe (**origin/main**, **origin/HEAD**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (test)

$

**Git Merge Conflicts:**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ echo " Line1: change" > example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git checkout -b imti

Switched to a new branch 'imti'

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ git checkout main

Switched to branch 'main'

D abc.html

M example.txt

D xyz

Your branch is ahead of 'origin/main' by 1 commit.

(use "git push" to publish your local commits)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git add example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git commit -m "m1" example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

[main 109f87b] m1

1 file changed, 1 insertion(+), 1 deletion(-)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git checkout imti

Switched to branch 'imti'

D abc.html

D xyz

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ ls

1.txt 2.txt 3.txt LICENSE README.md abc example.txt java qwe sql

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ cat example.txt

Line1: Initial

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ echo " Line2: change" >> example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ git add example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ git commit -m "m2" example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

[imti 627c110] m2

1 file changed, 1 insertion(+)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ cat example.txt

Line1: Initial

Line2: change

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (imti)

$ git checkout main

Switched to branch 'main'

D abc.html

D xyz

Your branch is ahead of 'origin/main' by 2 commits.

(use "git push" to publish your local commits)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ echo " Line2: main change" >> example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git add example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git commit -m "m3" example.txt

warning: in the working copy of 'example.txt', LF will be replaced by CRLF the next time Git touches it

[main 9ef161d] m3

1 file changed, 1 insertion(+)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ cat example.txt

Line1: change

Line2: main change

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$ git merge imti

Auto-merging example.txt

CONFLICT (content): Merge conflict in example.txt

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

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ vim example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ cat example.txt

<<<<<<< HEAD

Line1: change

Line2: main change

=======

Line1: Initial

Line2: change

>>>>>>> imti

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ git checkout imti

error: you need to resolve your current index first

example.txt: needs merge

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ cat example.txt

<<<<<<< HEAD

Line1: change

Line2: main change

=======

Line1: Initial

Line2: change

>>>>>>> imti

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ git checkout main

error: you need to resolve your current index first

example.txt: needs merge

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ vim example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

$ git add example.txt

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main|MERGING)

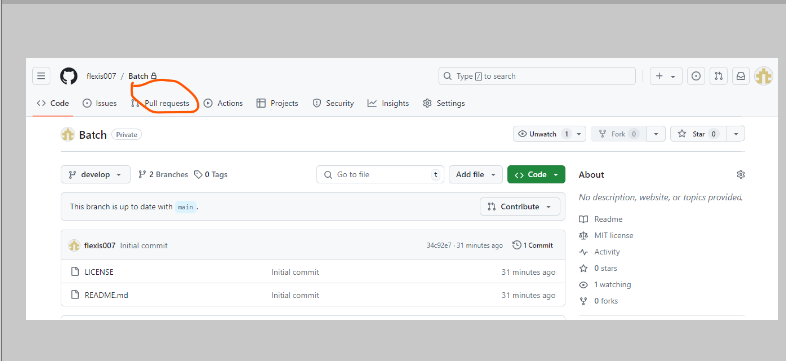
$ git commit -m "resolveconflict"

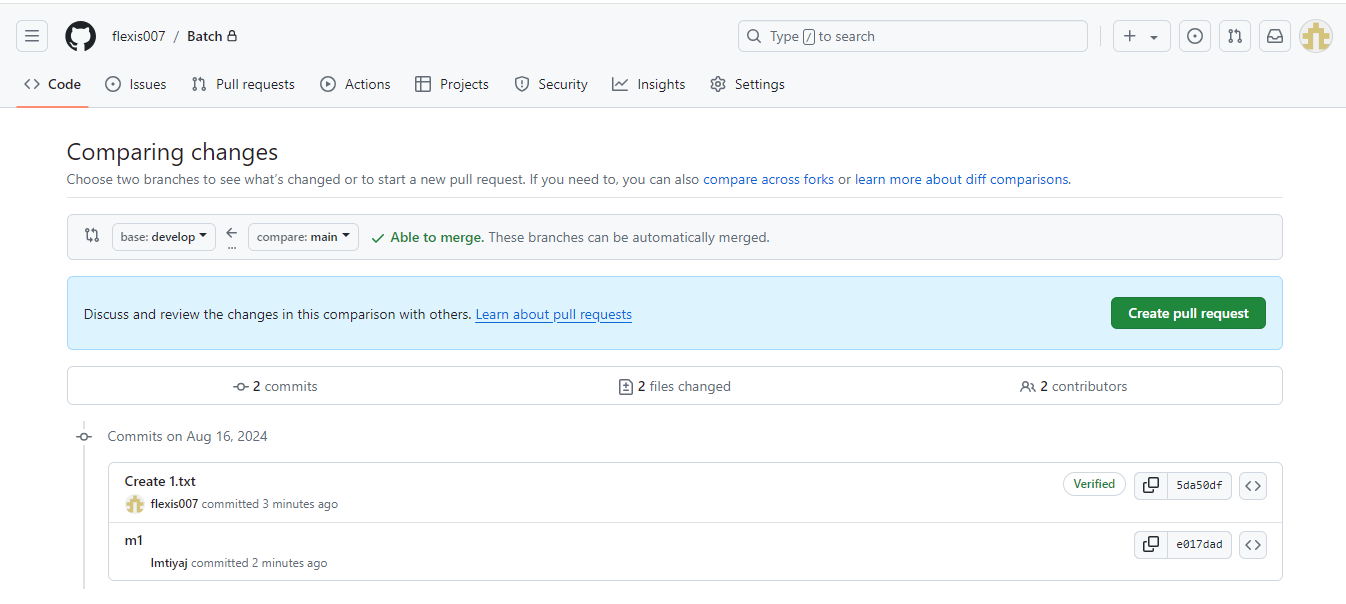
[main 5bee2c6] resolveconflict

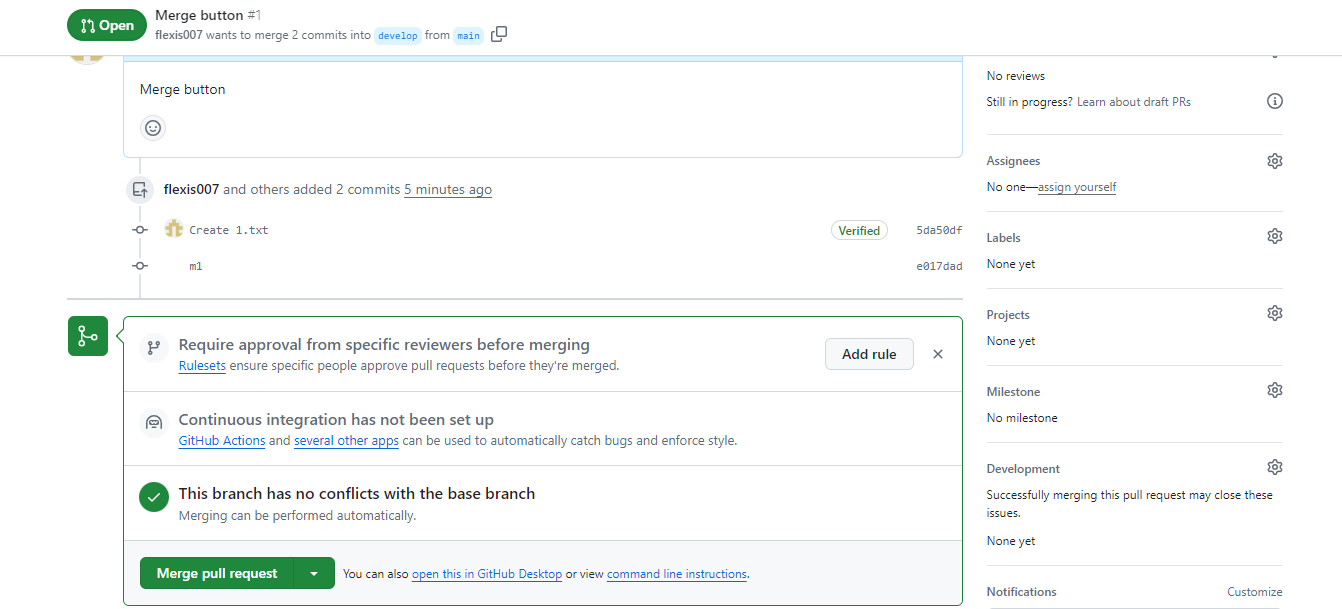
Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/test/stash (main)

$

**Pull Request:**

****

****

****

**Branching Strategy**

1. **Main/Branch:**
   1. **Name: main or master**
   2. **Purpose:** This is the main branch that holds production-ready code. It's always stable and should ideally reflect the code running in the production environment.
2. **Development Branch:**

**Name: develop or dev**

**Purpose:** All ongoing development work takes place in this branch. New features and bug fixes are merged into this branch. It should be relatively stable but not necessarily production-ready at all times.

1. **QA Branch:**

**Name: qa or testing**

**Purpose:** Once features are considered complete in the develop branch, they are merged into the qa branch for testing. This branch should reflect a stable state for testing purposes.

1. **Pre-Production Branch:** 
   1. **Name: ppd or staging**
   2. **Purpose:** This branch is used to simulate the production environment closely. After successful QA testing, code is merged from the qa branch to the ppd branch for final validation before deployment.

**5 Production Branch:**

* 1. **Name: prod or release**
  2. **Purpose:** Once the code is thoroughly tested in the ppd environment and ready for deployment, it's merged into the prod branch and deployed to the production environment.

**6 Disaster Recovery Branch:**

* 1. **Name: dr or backup**
  2. **Purpose:** This branch holds code that is identical to the currently deployed production code. It's useful for disaster recovery scenarios, allowing rapid deployment of the latest stable code in case of critical issues.

**Workflow:**

1. 1. Developers work on feature branches derived from develop.
2. 2. Once a feature is complete, it's merged into develop.
3. 3. Regular integration and automated tests take place in develop.
4. 4. Periodic merges from develop to qa for testing.
5. 5. After successful QA, merge to ppd for final validation.
6. 6. After final validation in ppd, merge to prod for deployment.
7. 7. Maintain a mirror of the production code in the dr branch.

**Git Reset:**

**Soft Reset: - remove from the commit they are currently staging area**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ ls

1.txt 2.txt 3 4 5 LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git status

On branch main

Your branch is ahead of 'origin/main' by 3 commits.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

dc2dff7 (**HEAD** -> **main**) 5

62205b6 4

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git reset --soft eab4a0f

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ ls

1.txt 2.txt 3 4 5 LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git status

On branch main

Your branch is ahead of 'origin/main' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: 4

new file: 5

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

eab4a0f (**HEAD** -> **main**) 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$

**Mix Reset: - remove from the commit and staging area**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ ls

1.txt 10 11 12 2.txt 3 4 5 8 9 LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

b29764f (**HEAD** -> **main**) 12

68397c0 11

4d96963 10

2a21356 9

e9bcfe1 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git reset 2a21356

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git status

On branch main

Your branch is ahead of 'origin/main' by 3 commits.

(use "git push" to publish your local commits)

Untracked files:

(use "git add <file>..." to include in what will be committed)

10

11

12

4

5

nothing added to commit but untracked files present (use "git add" to track)

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

2a21356 (**HEAD** -> **main**) 9

e9bcfe1 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$

**Hard Reset: Files removed permanently**

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ ls

1.txt 10 11 12 14 15 16 2.txt 3 4 5 8 9 LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

0c72b09 (**HEAD** -> **main**) 16

6a19f45 15

4d8421c 14

2a21356 9

e9bcfe1 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git reset --hard e9bcfe1

HEAD is now at e9bcfe1 8

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

e9bcfe1 (**HEAD** -> **main**) 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$



**Git Revert**

git revert is used to create a new commit that undoes the changes introduced by a previous commit. It's a safe way to undo changes while preserving the commit history.

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ ls

1.txt 10 11 12 2.txt 20 21 22 23 24 3 4 5 8 LICENSE README.md

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

d8df0fa (**HEAD** -> **main**) 24

5c4cfd1 23

923cc61 22

f48d02c 21

61577d0 20

e9bcfe1 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git revert 61577d0

[main 5f119b1] undo

1 file changed, 0 insertions(+), 0 deletions(-)

delete mode 100644 20

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$ git log --oneline

5f119b1 (**HEAD** -> **main**) undo

d8df0fa 24

5c4cfd1 23

923cc61 22

f48d02c 21

61577d0 20

e9bcfe1 8

eab4a0f 3

e017dad (**origin/main**, **origin/HEAD**) m1

5da50df Create 1.txt

34c92e7 (**origin/develop**) Initial commit

Farhan@DESKTOP-IR4NRGD MINGW64 ~/Documents/faru/Batch (main)

$

**50 Git commands**

1. git init Initializes a new Git repository.

$ git init

Initialized empty Git repository in /path/to/repository/

2. git clone Clones a remote repository to your local machine.

$ git clone https://github.com/username/repository.git

Cloning into 'repository'...

3. git add Stages changes for commit.

$ git add file.txt

4. git status Shows the status of the working directory and staged changes.

$ git status

5. git commit Commits staged changes.

$ git commit -m "Added new feature"

6. git log Displays commit history.

$ git log

7. git diff Shows differences between working directory and staged changes.

$ git diff

8. git branch Lists branches.

$ git branch

9. git checkout Switches branches or restores files.

$ git checkout branch\_name

10. git merge Merges changes from one branch into another.

$ git merge feature\_branch

11. git pull Fetches and integrates changes from a remote repository.

$ git pull origin master

12. git push Pushes changes to a remote repository.

$ git push origin master

13. git remote Manages remote repositories.

$ git remote add origin <https://github.com/username/repository.git>

14. git fetch Downloads objects and refs from a remote repository.

$ git fetch origin

15. git stash Temporarily stores changes to work on something else.

$ git stash

16. git tag Creates and manages tags for specific commits.

$ git tag v1.0.0

17. git reset Unstages changes or moves the HEAD to a specific commit.

$ git reset HEAD file.txt

18. git rebase Reapplies commits on top of another base.

$ git rebase master

19. git config Sets configuration options.

$ git config --global user.name "Your Name"

$ git config --global user.email [your.email@example.com](mailto:your.email@example.com)

20. git log --oneline Displays compact commit history.

$ git log –oneline

21. git show Shows information about a commit.

$ git show commit\_hash

22. git cherry-pick Applies a commit from one branch to another.

$ git cherry-pick commit\_hash

23. git rm Removes files from the working directory and stages the removal.

$ git rm file.txt

24. git revert Creates a new commit that undoes changes from a previous commit.

$ git revert commit\_hash

25. git reflog Displays the history of HEAD positions.

$ git reflog

26. git clean Removes untracked files and directories from the working directory.

$ git clean -n # Dry-run

$ git clean -f # Force removal

27. git tag -a Creates an annotated tag with a message.

$ git tag -a v1.0.0 -m "Version 1.0.0"

28. git log --graph Displays commit history as a graph.

$ git log --graph –oneline

29. git config --list Lists all Git configuration settings.

$ git config –list

30. git log --since / git log --until Displays commit history within a time range.

$ git log --since="2 weeks ago"

$ git log --until="2023-07-01"

31. git cherry Shows commits that have not been merged.

$ git cherry master feature\_branch

32. git revert --no-commit Reverts changes interactively without committing.

$ git revert --no-commit commit\_range

33. git log --author Filters commit history by author.

$ git log --author="John Doe"

34. git log --stat Displays file statistics with commit history.

$ git log –stat

35. git blame Shows who last modified each line in a file.

$ git blame file.txt

36. git tag -d Deletes a tag.

$ git tag -d v1.0.0

37. git log -p Displays commit history with patch diffs.

$ git log -p

38. git rev-parse Converts a revision string into a SHA-1 hash.

$ git rev-parse HEAD

39. git remote -v Lists remote repositories and their URLs.

$ git remote -v

40. git log --decorate Displays references (branches, tags) in commit history.

$ git log –decorate

41. git bisect Performs a binary search to find a faulty commit.

$ git bisect start

$ git bisect good <commit>

$ git bisect bad <commit>

$ git bisect reset

42. git log --grep Searches commit messages for a specific keyword.

$ git log --grep="bug fix"

43. git log --name-only Displays only file names in commit history.

$ git log --name-only

44. git rebase -i Interactively rewrites commit history.

$ git rebase -i HEAD~3

45. git log --before / git log --after Displays commit history before/after a specific date.

$ git log --before="2023-01-01"

$ git log --after="2022-01-01"

46. git checkout -b Creates a new branch and switches to it.

$ git checkout -b new\_feature

47. git log --cherry-pick Shows commits that have been cherry-picked.

$ git log --cherry-pick master..feature\_branch

48. git log -S Searches for changes that added or removed a specific string.

$ git log -S "function\_name"

49. git reflog expire Expires old reflog entries.

$ git reflog expire --expire=30.days refs/heads/master

50. git commit --amend Modifies the most recent commit.

$ git commit –amend

======================🡺 ========================🡺===============

**Maven**

**Maven: -** It is building automation tools for java-based projects.

**Structure of project: -** src folder contains all code.

pom.xml contains the dependencies of the application.

Dependencies means any project you build that requires files like some support files, these files are needed or mandatory for application to work well, these files can be in different format like jar, war and pom.

Dependencies need to add pom.xml file like from <https://mvnrepository.com/>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>2.3.232</version>

<scope>test</scope>

</dependency>

**Maven Lifecycle: -**

1. **Clean Lifecycle: -** mvn clean (delete target folder and create fresh target folder)
2. **Default Lifecycle: -** Validate, Compile, test, package, verify, install, deploy (Validate and verify not used)

* **compile: -** mvn compile, found any syntax-based error like;}
* **test: -** mvn test, for unit testing
* **package: -** mvn package, try to be packing/building the whole application then convert into executable format/file like jar (java archive), war (web application archive) and ear (enterprise application archive) and it is stored in target folder.
* **Install: -** mvn install, after running the mvn install new jar will be created and installed on local maven repository, like .m2 where jar is stored.
* **Deploy: -** mvn deploy means upload the jar file on third party external repo like Nexus or jfrog

1. **Site Lifecycle: -** In site lifecycle we have commands to generate documentation with respect to maven (Very less used)

**Jar vs war vs ear: -**

* jar is standalone application not required server (java -jar app.jar)
* war is website-based application so required servers like tomcat, nginx server (java -jar app.war)
* ear is used in very big application directly deployed on server.
* mvn package -DskipTests=true (skip test and direct run package)

**Maven Practical: -**

* sudo apt update
* java
* sudo apt install openjdk-17-jre-headless -y
* mvn
* sudo apt install maven -y
* git clone <https://github.com/flexis007/Boardgame.git>
* cd Boardgame/
* ls
* mvn compile
* mvn test
* mvn package
* cd target/
* java -jar database\_service\_project-0.0.4.jar
* <http://13.232.169.122:8080/>
* cd ..
* mvn clean package
* mvn package
* mvn install

**Deploy nodejs Server: -**

* install nodejs <https://nodejs.org/en/download/package-manager>
* git clone <https://github.com/flexis007/demo_nodejs_webpage.git>
* cd demo\_nodejs\_webpage
* npm install
* npm start
* ip:8081

======================🡺 ========================🡺===============

**Jenkins**

Jenkins is an open-source automation server that is widely used for building, testing, and deploying software projects. It is one of the most popular and powerful tools in the field of Continuous Integration and Continuous Deployment (CI/CD). Here are the key features of Jenkins in detail:

1. **Automation of Repetitive Tasks:**

Jenkins automates repetitive tasks such as building, testing, and deploying code, freeing up developers from manual, error-prone processes.

1. **Integration with Version Control Systems:**

Jenkins can integrate with various version control systems like Git, Subversion, Mercurial, and more, allowing it to trigger builds automatically upon code changes.

1. **Extensive Plugin Ecosystem:**

Jenkins has a rich ecosystem of plugins (over 1,500 at the time of my last update), which extend its functionality to integrate with various tools and technologies, making it highly customizable.

1. **Distributed Build Support:**

Jenkins supports distributed builds, allowing you to set up a master-slave architecture where build tasks can be distributed across multiple machines to improve performance and scalability.

1. **Customizable Dashboards:**

Jenkins provides a web-based user interface with customizable dashboards and widgets, allowing teams to monitor build and deployment statuses, trends, and test results.

1. **Notification and Alerts:**

Jenkins can send notifications and alerts through email, chat services (Slack, Microsoft Teams), or other communication channels to keep the team informed about build and deployment outcomes.

**Plugins: -** In Jenkins, plugins are extensions that add new features or enhance existing functionalities. Jenkins itself is a powerful automation server, but plugins allow you to customize it to fit your specific needs.

1. **Pipeline Plugin:** This plugin enables you to define and manage your build and deployment pipelines as code using Jenkinsfile.
2. **Git Plugin:** Allows Jenkins to integrate with Git repositories, making it easy to trigger builds based on code changes.
3. **GitHub Integration Plugin:** Provides deeper integration with GitHub, enabling you to trigger builds and update GitHub statuses.
4. **Docker Plugin:** Integrates Jenkins with Docker, allowing you to build, push, and run Docker containers as part of your CI/CD process.
5. **ArtifactDeployer Plugin:** Facilitates the archiving and deployment of build artifacts to various repositories.
6. **Maven Plugin:** Streamlines the integration of Apache Maven with Jenkins, making it easy to build and deploy Java projects.
7. **JIRA Integration Plugin:** Enables integration with Atlassian JIRA for tracking and managing issues related to builds and deployments.
8. **Email Extension Plugin:** Allows you to send customizable email notifications upon build success, failure, or other events.
9. **Copy Artifact Plugin:** Lets you copy build artifacts from one job to another, useful for passing artifacts between pipeline stages.
10. **Workspace Cleanup Plugin:** Automatically cleans up workspaces to free up disk space after builds.
11. **Build Timeout Plugin:** Adds the ability to set build timeouts, ensuring that builds don't hang indefinitely.
12. **Credentials Plugin:** Provides a secure way to manage and use credentials, such as API keys, passwords, and SSH keys, within your Jenkins jobs.
13. **NodeJS Plugin:** Simplifies the installation and management of Node.js versions on Jenkins agents.
14. **Blue Ocean Plugin:** Offers a modern, user-friendly UI for creating and visualizing Jenkins pipelines, making it easier to understand and troubleshoot builds.

**15. Slack Notification Plugin:** Integrates Jenkins with Slack, allowing you to send build notifications and updates to Slack channels

**Jenkins practical: -**

4gb ram and 20 gb storage minimum

apt install openjdk-17-jre-headless -y

install Jenkins from google

systemctl start Jenkins && systemctl enable Jenkins

**Manage Jenkins: -**

1. **System: -** Whichever specific servers you want to install within Jenkins, for example if you want to perform SonarQube then two things you need sonar server and sonar scanner

Sonar scanners analyze your source code and generate report and report will publish on sonar server

1. **Plugins: -** Plugins are just like extension, Example

A close-up of a white background

Description automatically generated

1. **Tools: -** Where you going to configure the plugins, example you want multiple jdk in your pipeline

A screenshot of a computer

Description automatically generated

**Types of Jenkins Jobs: -**

1. **Freestyle: -** Simple, easy to set up, best for single tasks.
2. **Pipeline: -** More powerful and flexible, suitable for complex workflows  
   (used for Single branch) written in groovy script
3. **Multibranch Pipeline: -** Automates management of multiple branches with individual pipelines, ideal for collaborative environments.

(Used for multibranch like dev----feature, qa, test etc.)

**Create your 1st job freestyle: -**

**discard old builds** in a Freestyle Jenkins job, you can set up build retention policies directly in the job configuration

Dashboard---New---jobname---freestyle---discard old build (max build to keep 2) ---jdk (which you want to select) ---git (paste git URL) ---build steps (Invoke top-level Maven targets---maven ---package) ---apply---build now.

**Build Triggers: -**

**1. Build Periodically**

Description: Schedule builds to run at specific intervals using cron syntax.

Usage: Useful for regular tasks like nightly builds or periodic testing.

Example Cron Syntax: H 2 \* \* \* (runs daily at 2 AM).

**2. Poll SCM**

Description: Jenkins checks your source code management (SCM) system at regular intervals for changes.

Usage: Automatically triggers a build if there are changes in the repository.

Configuration: Specify the polling interval (e.g., every minute: \* \* \* \* \*).

**3. GitHub hook trigger for GITScm polling**

Description: Automatically triggers a build when a push occurs in a GitHub repository.

Usage: Requires a GitHub webhook to be set up in the repository.

Configuration: Enable the "GitHub hook trigger for GITScm polling" option.

**4. Trigger builds remotely (e.g., from scripts)**

Description: Allows you to trigger builds via an HTTP request.

Usage: Useful for integrating with other systems or custom scripts.

Configuration: You need to specify an authentication token for security.

**5. Build after other projects are built**

Description: Allows you to trigger this job after another job has completed.

Usage: Useful for creating a chain of jobs where one depends on the success of another.

Configuration: Specify the projects to trigger and set whether to wait for them to succeed.

**Create your 2nd job pipeline: -**

Dashboard---New---jobname---pipeline-- discard old build (max build to keep 2) ---pipeline –pipeline script (write your pipeline) ---build now

pipeline {

agent any

tools {

maven 'maven3'

jdk 'jdk18'

}

stages {

stage('Git Checkout') {

steps {

git branch: 'main', url: 'https://github.com/flexis007/FullStack-Blogging-App.git'

}

}

stage('Compile') {

steps {

sh "mvn compile"

}

}

stage('Test') {

steps {

sh "mvn test"

}

}

stage('package') {

steps {

sh "mvn package"

}

}

}

}

We can also copy this pipeline and on git repo, create a file name Jenkinsfile and paste this content in this file--- Pipeline—Pipeline scipt from SCM

**Write code in Jenkinsfile**

pipeline {

agent any

tools {

maven 'maven3'

jdk 'jdk18'

}

stages {

stage('Compile') {

steps {

sh "mvn compile"

}

}

stage('Test') {

steps {

sh "mvn test"

}

}

stage('package') {

steps {

sh "mvn package"

}

}

}

}

**Create your 3rd job Multibranch pipeline: -**

Lets assume you have main, develop, dev, feature1, feature2 branch then it will check Jenkinsfile in all branch, in case if Jenkinsfile not now available in any branch they ignore this branch.

Dashboard---New---jobname---Multibranchpipeline—branch source (git)— discard old build (max build to keep 2)

For check in stage view you can install view plugin.

**Setting Up a Trigger for a Pipeline Using a Generic Webhook Trigger**

**Step 1: Install Generic Webhook Trigger Plugin**

1. Go to Jenkins dashboard.
2. Navigate to Manage Jenkins -> Manage Plugins.
3. In the Available tab, search for "Generic Webhook Trigger."
4. Install the plugin and restart Jenkins if necessary.

**Step 2: Configure Jenkins Job**

1. Create or open the Jenkins job you want to trigger.
2. In the job configuration, scroll down to the Build Triggers section.
3. Check the checkbox for Generic Webhook Trigger.

**Step 3: Configure Post Parameters**

* 1. In the Post Parameters section, add the following: Variable: ref
  2. Expression (JSON): $.ref

**Step 4: Write a string as Token**

1. Write any keyword string as token that will be used in the webhook URL.

**Step 5: Configure Optional Filter**

1. In the Optional Filter section, configure the following: Expression: refs/heads/branch\_name (Replace branch\_name with the name of your branch)
2. Text: $ref

**Step 6: Configure GitHub Webhook**

1. Go to your GitHub repository settings.
2. Navigate to Webhooks.
3. Click Add webhook.

1. In the Payload URL field, enter the following URL:

Jenkins\_URL/generic-webhook-trigger/invoke?token=github\_token

Replace Jenkins\_URL with the URL of your Jenkins instance and github\_token with the token generated earlier. Example:

http://65.0.31.109:8080/generic-webhook-trigger/invoke?token=github\_token

1. Set the Content type to application/json.
2. Select the events you want to trigger the webhook for (e.g., Pushes).

**Step 7: Save Changes**

1. Save your Jenkins job configuration.

Now, your Jenkins job is configured to trigger via a generic webhook whenever a push event occurs on the specified branch in your GitHub repository.

**Setting Up a Trigger for a Multibranch Pipeline Using Multibranch Scan Webhook Trigger Plugin**  
**Step 1: Install Multibranch Scan Webhook Trigger Plugin**

1. Navigate to Jenkins dashboard.
2. Go to Manage Jenkins -> Manage Plugins.
3. In the Available tab, search for "Multibranch Scan Webhook Trigger."
4. Install the plugin and restart Jenkins if required.

**Step 2: Configure Multibranch Pipeline Job**

1. Create or open the multibranch pipeline job you want to trigger.
2. In the job configuration, navigate to Scan Multibranch Pipeline Triggers.
3. Select Scan by webhook.
4. Enter any trigger token in the Trigger token field.

**Step 3: Configure GitHub Webhook**

1. Go to your GitHub repository settings.
2. Navigate to Webhooks.
3. Click Add webhook.
4. In the Payload URL field, enter the following URL:

Jenkins\_URL/multibranch-webhook-trigger/invoke?token=token\_value

Replace Jenkins\_URL with the URL of your Jenkins instance and token\_value with the trigger token you configured in the Jenkins job. Example:

http://65.0.31.109:8080/multibranch-webhook-trigger/invoke?token=Devopsshack

1. Set the Content type to application/json.
2. Select the events you want to trigger the webhook for (e.g., Pushes).

**Step 4: Save Changes**

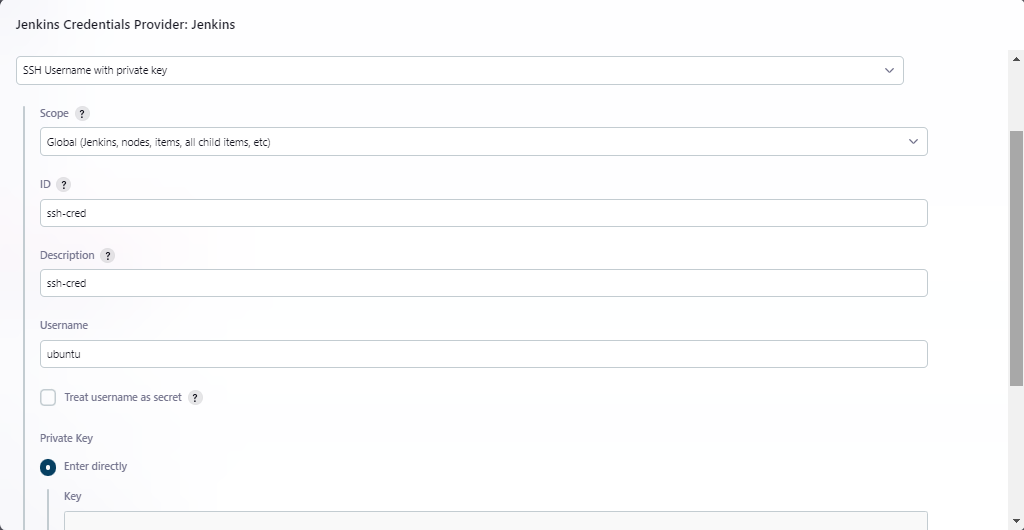
1. Save your Jenkins job configuration.

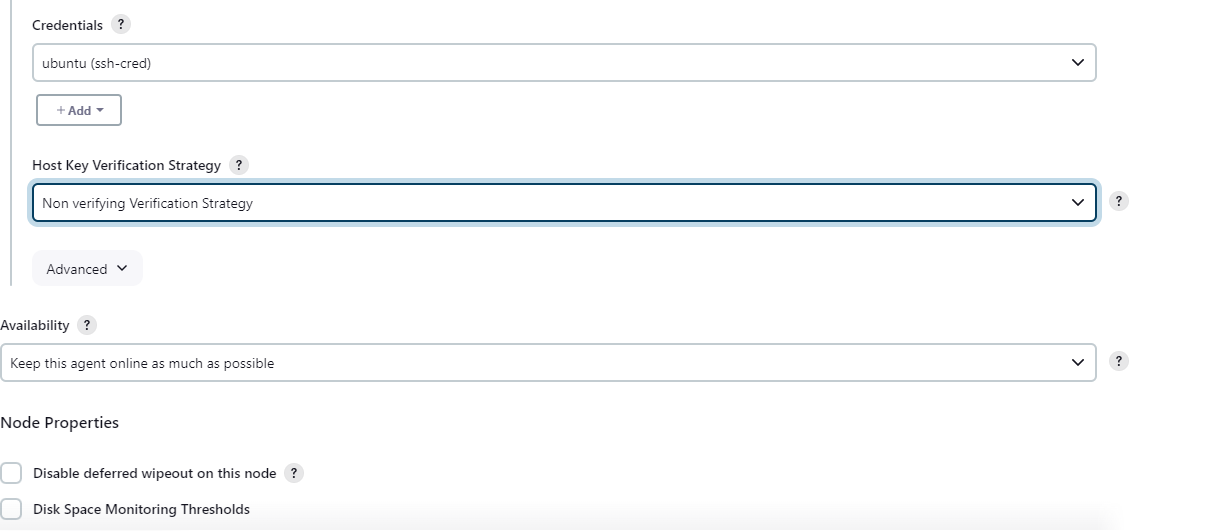
Now, your multibranch pipeline job is configured to trigger via webhook whenever a push event occurs on your GitHub repository.

**Master Slave architecture: -**

Create a slave machine---install java  
mkdir slave (store all jobs here)  
manage Jenkins---nodes---new node---slave-1—create--- Number of executors (How many jobs run at a same time) --- Remote root directory (directory pwd /home/ubuntu/slave) --- Labels (agent will be identified slave-1)---usage (only build jobs)---launch method (via ssh)---

  
host---master IP  
instead of /root/slave --- /home/ubuntu/slave





**Save.**

Dashboard---New---jobname---pipeline-- discard old build (max build to keep 2) ---pipeline –pipeline script (write your pipeline) ---build now

pipeline {

agent { label ‘slave-1’ }

tools {

maven 'maven3'

jdk 'jdk18'

}

stages {

stage('Git Checkout') {

steps {

git branch: 'main', url: 'https://github.com/flexis007/FullStack-Blogging-App.git'

}

}

stage('Compile') {

steps {

sh "mvn compile"

}

}

stage('Test') {

steps {

sh "mvn test"

}

}

stage('package') {

steps {

sh "mvn package"

}

}

}

}

**Parameter: -**

**1st type:-**

Create a pipeline job

A line of blue lines

Description automatically generated with medium confidence

A graph with green lines

Description automatically generated

A screenshot of a computer

Description automatically generated

**2nd type direct in pipeline:-**

**S**tring

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**2nd Choices:-**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**User Authentication: -   
  
first need to install** [**Matrix Authorization Strategy Plugin**](https://plugins.jenkins.io/matrix-auth)

A screenshot of a computer

Description automatically generated

**Second need to create users and group**

**Manage Jenkins---users**

**Third need to provide access of this user**

**Manage Jenkins---Security—**

**A screenshot of a computer

Description automatically generated**

**Upstream and Downstream job in Jenkins**

**I want to trigger 1 Jenkins job based on completion on another Jenkins job**

**Job triggers another job--- upstream**

**Job getting triggered or started--- downstream**

**Create a job ---pipeline**

A screenshot of a computer

Description automatically generated

Create another job---pipeline

A screenshot of a computer

Description automatically generated

B job should be triggered completion of A. (A trigger B)

**Jenkins Backup 1st type**

1. Create a Jenkins server---create a job and build
2. Create a S3 bucket
3. Systemctl stop Jenkins
4. tar -zcvf jenkinsbackup.tar.gz /var/lib/jenkins/
5. IAM Roles with s3 full access policy and attach with Server
6. aws s3 cp jenkinsbackup.tar.gz s3:// jenkinsbackupaseem/jenkinsbackup.tar.gz
7. Now terminate the server
8. Create new Instance and install Jenkins for restore and attach IAM role
9. Systemctl stop Jenkins
10. aws s3 cp s3://jenkinsbackupaseem/jenkinsbackup.tar.gz jenkinsbackup.tar.gz
11. rm -rf /var/lib/Jenkins
12. tar -zxvf jenkinsbackup.tar.gz -C /
13. system start Jenkins

**Jenkins Backup 2nd type**

Install the plugin thin backup and configure it set crontab for automatic backup

**Jenkins Shared Libraries**

**Jenkins Troubleshoot**

Gitcheckout --- repo url /user/token ---- 128 error if not provide user and token/password

Gitcheckout --- repo url /user/token ---403 provide wrong user and token/password

Pipeline fails --- 137 not enough memory

127---command or tool not found like mvn not found

When we run sonar analysis with Jenkins and we forget to configure the sonar server--- sonar not available

500 – You should check immediately server up or not (if push nexus artifact and getting error then check nexus server up or not if it is up then restart the server)