INTRODUCTION TO DEVOPS AND GITLAB

DevOps With Gitlab

JOAQUIN ANTONIO

Introduction to DevOps and GitLab

1. **Overview of DevOps**
    1. Definition and principles of DevOps
    2. Importance of DevOps in software development
2. **Introduction to GitLab**
    1. Overview of GitLab as an integrated DevOps platform
    2. Key features and benefits
3. **Setting Up GitLab Environment**
    1. Configuring basic settings

Definition and principles of DevOps

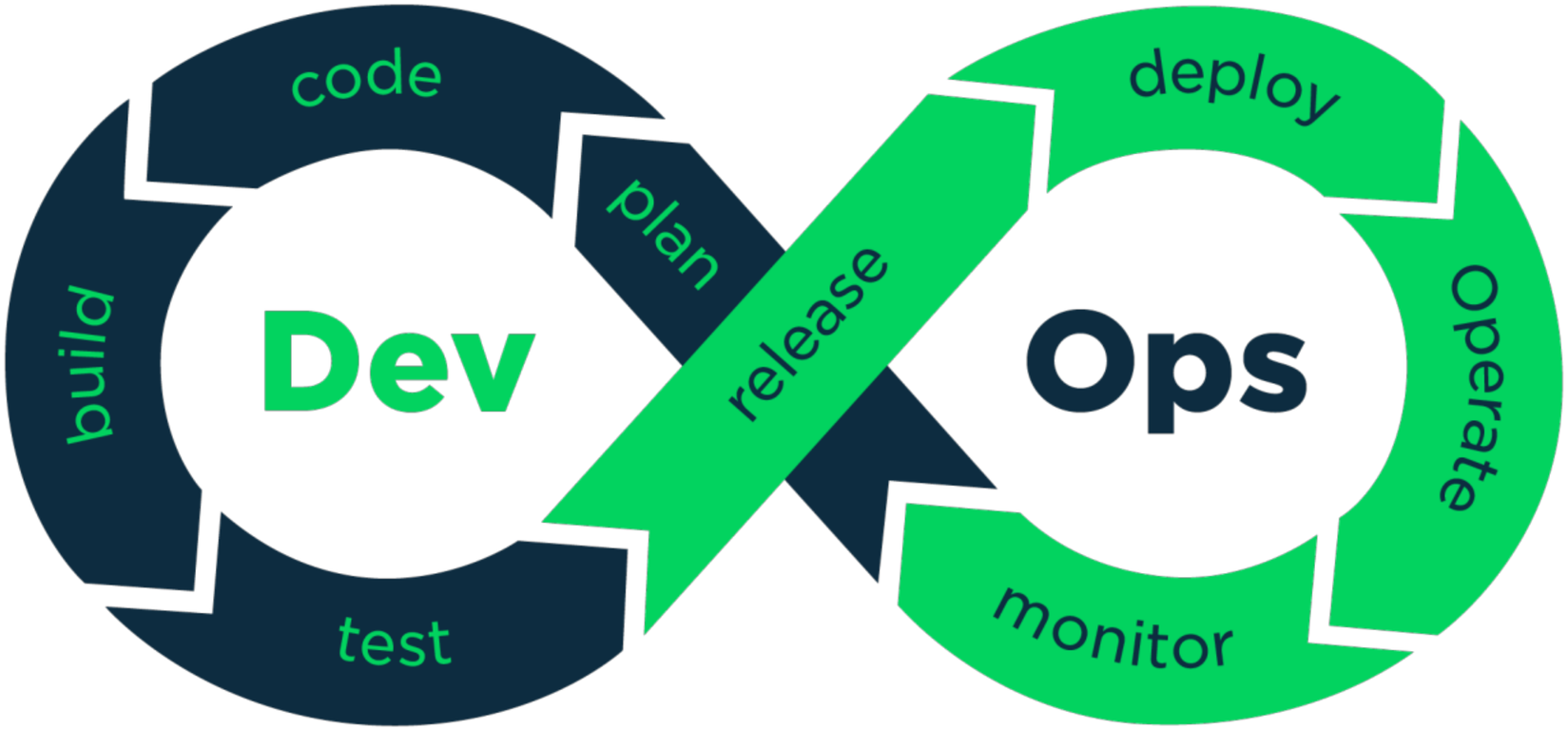
1. Practices and philosophies that aim to improve collaboration and communication between development (Dev) and operations (Ops) teams throughout the software development lifecycle.

2. Goal: enable faster and more reliable delivery of software, fostering a culture of continuous integration, continuous delivery, and continuous deployment.

3. DevOps is not just about tools

    1. Emphasises collaboration, communication, and automation to deliver high-quality software more efficiently and reliably.

DevOps Cycle



Importance of DevOps in software development

1. DevOps plays a crucial role in modern software development
2. Addresses challenges and fosters an environment for faster, more reliable, and efficient delivery of software.
    1. *Accelerated Time-to-Market*
    2. *Improved Collaboration*
    3. *Enhanced Quality and Reliability*
    4. *Efficient Resource Utilisation*
    5. *Increased Flexibility and Adaptability*
    6. *Improved Monitoring and Troubleshooting*
    7. *Cost Efficiency*

DevOps Core Practices

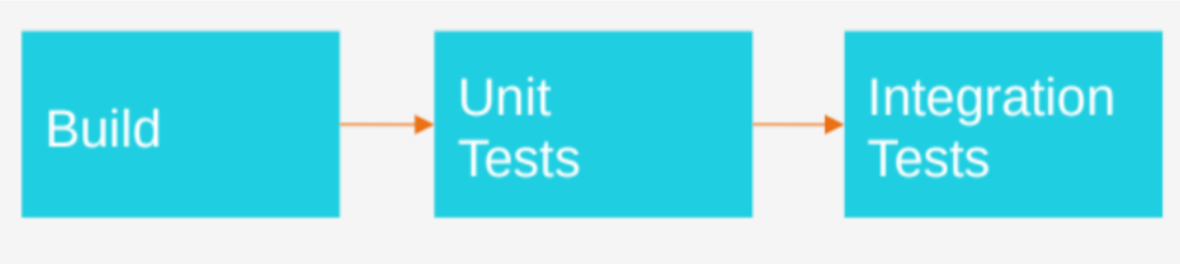
1. **Continuous Integration**

2. **Continuous Delivery**

3. **Continuous Deployment**

1. Continuous Integration

1. “Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.” – Martin Fowler (software engineer)

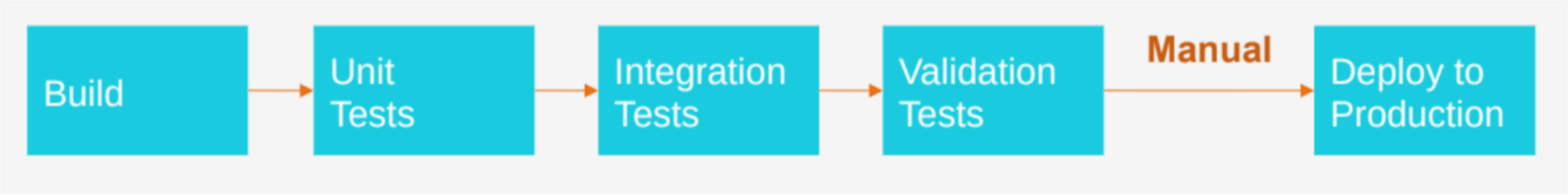


2. Continuous Delivery

1. “The essence of my philosophy to software delivery is to build software so that it is always in a state where it could be put into production. We call this Continuous Delivery because we are continuously running a deployment pipeline that tests if this software is in a state to be delivered.” – Jez Humble, Thoughtworks

Continuous Delivery vs Continuous Integration

1. Continuous Delivery = Continuous Integration + Fully automated test suite
2. Not every change is a release
3. Continuous Delivery is all about testing



3. Continuous Deployment

1. Deployments to the cloud enable all sorts of capabilities you would otherwise not have
    1. Deployment can include the complete creation of a virtual environment
    2. Servers don’t need to have applications redeployed
    3. You simply create new servers for every deployment
    4. Infrastructure can be in source control as a script
    5. You can easily create test environments

Continuous Delivery vs Continuous Deployment

1. When deploying to the cloud, we really want automation
2. Frequent, consistent, reliable deployments
3. One of the most important ways to achieve this in the cloud is through infrastructure as code

Overview of GitLab as an integrated DevOps platform

1. GitLab is a web-based platform that provides a comprehensive set of tools for managing the entire software development lifecycle.

2. It is designed as an integrated DevOps platform offering capabilities that cover:

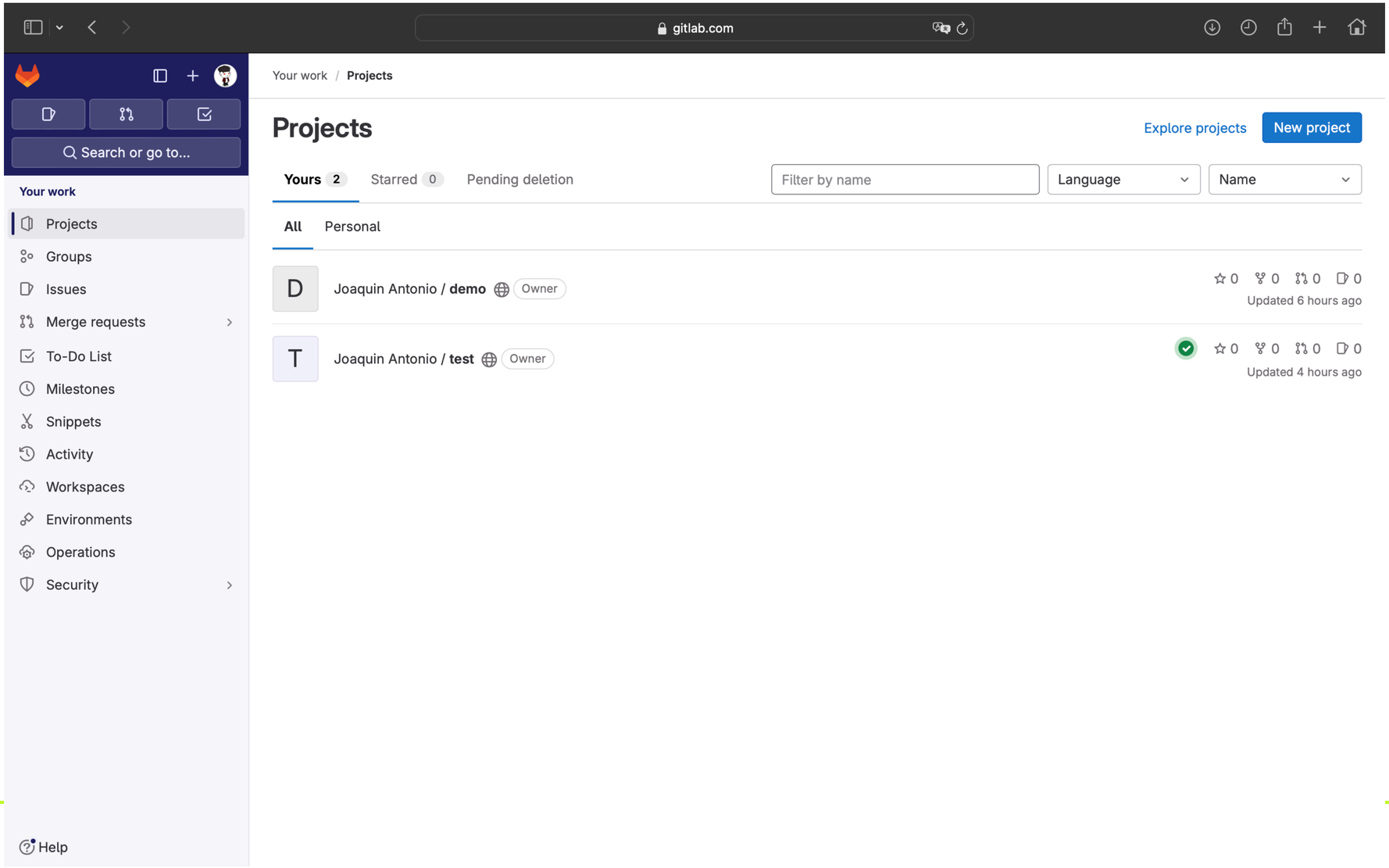
    1. *source code management, continuous integration, continuous delivery, container registry, monitoring, etc*

3. GitLab enables teams to collaborate efficiently and streamline their development processes.

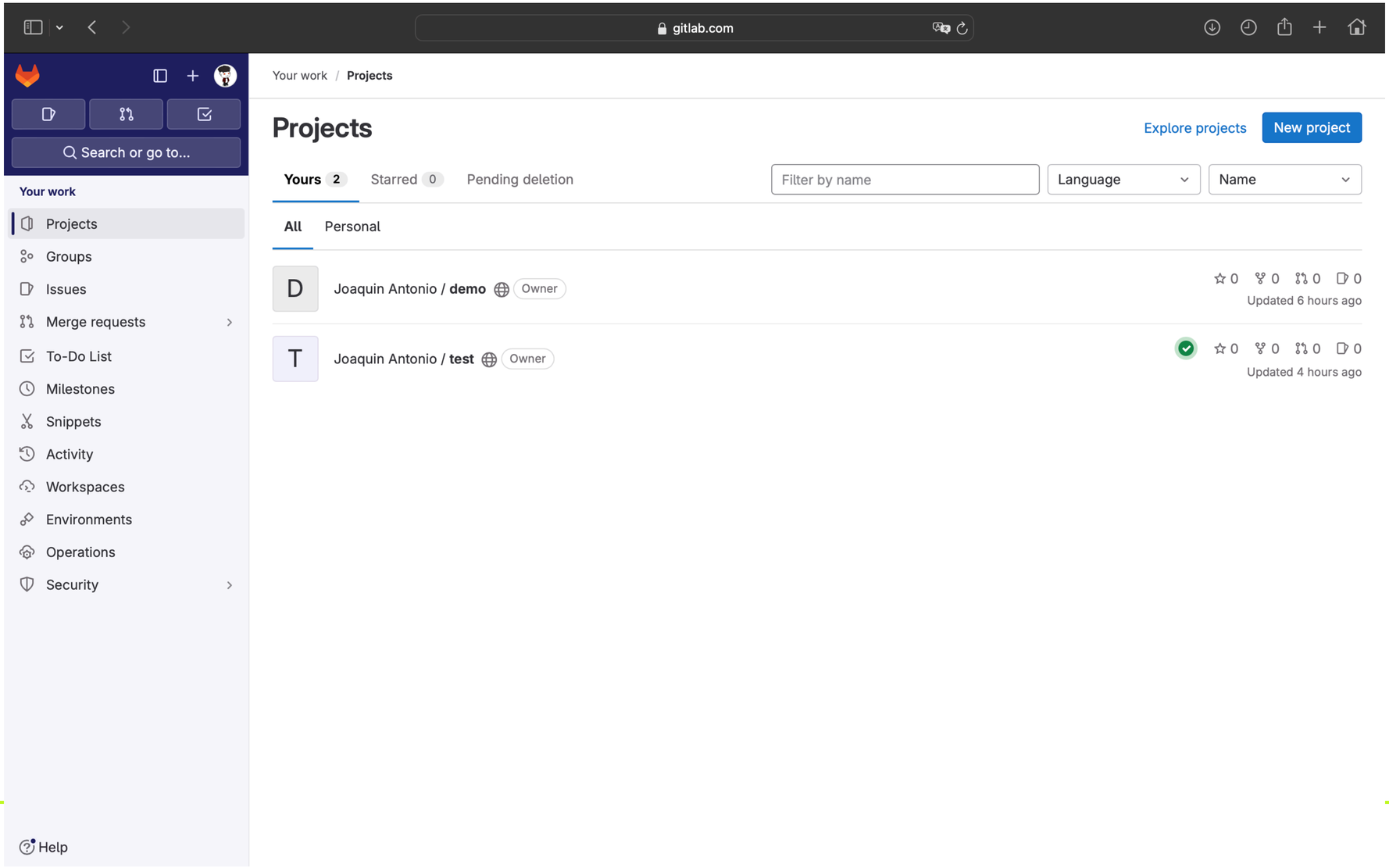
Gitlab vs Github

The major difference between GitHub and GitLab is that **GitHub** enables you to choose your **CI/CD tools after integration**, whereas **GitLab** has **integrated CI/CD tools and DevOps workflows**.

EXERCISE login and a look at Gitlab



EXERCISE


gitlab.com
Search or go to...
Your work
Projects
Groups
Issues
Merge requests
To-Do List
Milestones
Snippets
Activity
Workspaces
Environments
Operations
Security
Help
Your work / Projects
Projects
Explore projects
New project
Yours 2
Starred 0
Pending deletion
Filter by name
Language
Name
All
Personal
D
Joaquin Antonio / demo
Owner
0 0 0 0
Updated 6 hours ago
T
Joaquin Antonio / test
Owner
0 0 0 0
Updated 4 hours ago

Git and Version Control

Git and Version Control

1. **Introduction to Version Control with Git**
    1. Basics of version control
    2. Git concepts: repositories, commits, branches, and merges
2. **GitLab Version Control Workflow**
    1. Overview of GitLab repository structure
    2. Understanding branches in GitLab

Basics of version control

1. **Repository**: a central location where version-controlled files and their history are stored.
2. **Commit**: a snapshot of the code at a specific point in time including changes made to files, a commit message describing the modifications, and a unique identifier (hash).
3. **Branch**: a separate line of development within a repository.
4. **Merge**: combining changes from one branch into another.
5. **Pull Request/Merge Request**: a mechanism for proposing changes to a branch. It allows other developers to review the code before merging it into the main branch.
6. **Clone**: Cloning is the process of copying a repository, creating a local version on a developer's machine.
7. **Push and Pull**: Push involves sending committed changes from a local repository to a remote repository. Pull is the opposite, fetching changes from a remote repository to a local one.
8. **Remote**: a version of the repository stored on a server.
9. **Conflict**: when two or more developers make changes to the same part of a file simultaneously.
10. **Tag**: a named reference to a specific commit often used to mark significant points in a project's history, like a release.

Git concepts: repositories, commits, branches, and merges

1. Repositories:
    1. a data structure that stores the metadata and objects related to a project.
    2. It contains all the files and the entire history of changes for a particular project.
2. Commits:
    1. a snapshot of the project at a specific point in time.
    2. It includes changes made to one or more files and a commit message describing the modifications.
3. Branches:
    1. allows developers to diverge from the main line of development and work on features, fixes, or experiments independently.
4. Merges:
    1. combining changes from one branch (source branch) into another (target branch).

How these concepts work together

1. Developers start with a clean repository, often on the main branch (master or main).
2. They create a new branch to work on a specific task or feature, using the git branch command.
3. Within the branch, they make changes to files and commit those changes using git commit.
4. If working in a team, developers may push their branch to a remote repository, making it available for collaboration.
5. Once the changes are complete, developers merge their branch back into the main branch using git merge.
6. Git automatically performs a merge if there are no conflicts. If conflicts occur, developers resolve them manually.
7. The commit history shows the development path, with branches and merges indicating different lines of work and when they were integrated.

Overview of GitLab repository structure

1. Project Overview:
2. Repository:
3. Commits:
4. Branches:
5. Tags:
6. Merge Requests (MRs):
7. Issues and Boards:
8. Wiki:
9. CI/CD Pipelines:
10. Settings:
11. Web IDE:
12. Container Registry:

Understanding branches in GitLab

1. Every GitLab project has a default branch, often named master or main.
2. Developers create branches to work on new features, bug fixes, or experiments without affecting the main branch.
3. GitLab displays a list of branches in the repository, showing the default branch and any additional branches created by developers.
4. Developers can switch between branches to work on different features or bug fixes.
5. GitLab allows administrators to set branch-specific permissions.
6. Some branches, typically the default branch, can be marked as "protected." This means that only users with sufficient permissions can push changes directly to the branch.
7. Developers create MRs when they want to merge the changes from their feature branch into the default or target branch.
8. GitLab provides a visual representation of the branch history in the form of a branch graph. The graph shows the relationships between different branches and the commit history.

EXERCISE

1. Create a new project/repository in GitLab.
2. Clone the repository to your local machine.
3. Create a branch, make changes, and commit the changes.
4. Push the branch to GitLab.
5. Create a merge request, and merge the changes.

CI/CD with GitLab

CI/CD with GitLab

1. **Introduction to CI/CD**
    1. Definition and benefits
    2. Overview of GitLab CI/CD
2. **Configuring CI/CD Pipelines**
    1. Writing .gitlab-ci.yml files
    2. Defining jobs and stages
3. **Artifact Management and Deployment**
    1. Managing artifacts
    2. Deploying applications using GitLab CI/CD

Definition and benefits

1. Definition: Continuous Integration and Continuous Delivery (or Continuous Deployment), is a set of software development practices aimed at automating the process of integrating code changes, testing them, and delivering the application to production or staging environments in a streamlined and efficient manner.

Definition and benefits

1. Benefits:
    1. Faster Time-to-Market:
    2. Early Detection of Bugs and Issues:
    3. Consistent and Reliable Builds:
    4. Increased Collaboration:
    5. Automated Testing:
    6. Continuous Feedback:
    7. Scalability and Flexibility:
    8. Reduced Manual Errors:
    9. Incremental Updates and Rollbacks:
    10. Improved Security:

Overview of GitLab CI/CD

1. CI/CD Pipelines:
2. Jobs:
3. Runners:
4. .gitlab-ci.yml Configuration File:
5. Stages:
6. Artifacts:
7. Variables:
8. Triggers:
9. Manual Jobs:
10. Review Apps:
11. Multi-Project Pipelines:
12. Integrated Container Registry:

Writing .gitlab-ci.yml files

1. The .gitlab-ci.yml file is a configuration file used by GitLab CI/CD to define the structure and behaviour of the CI/CD pipeline for a project.
2. This YAML file is typically stored in the root directory of the GitLab repository.
3. **Key Concepts:**
    1. Stages:
    2. Variables:
    3. Before and After Scripts:
    4. Jobs:
    5. Script:
    6. Artifacts:
    7. Dependencies:
    8. Variables within Scripts:

Defining jobs and stages

1. Defining jobs and stages in a CI/CD pipeline using GitLab involves structuring your .gitlab-ci.yml file to organise tasks, specify dependencies, and control the flow of your pipeline.

Managing artifacts

1. Artifact management refers to the handling, storage, and sharing of build artifacts produced during the build and testing phases of a pipeline.

2. Artifacts are the output files or directories generated by a job,

    1. often includes compiled binaries, libraries, documentation, and other files necessary for deployment.

Deploying applications using GitLab CI/CD

1. Define Deployment Job:
2. Environment Variables:
3. Artifact Dependency
4. Deployment Scripts
5. Manual Deployments
6. Conditional Deployments
7. Rollbacks
8. Integration with Containerisation

EXERCISE

1. Create CI/CD jobs for building, testing, and deploying your application.
2. Trigger pipelines automatically upon code changes.
3. Analyse pipeline status and troubleshoot issues.

Collaboration and Code Review

Collaboration and Code Review

1. **GitLab Merge Requests**
    1. Creating merge requests
    2. Code reviews and discussions
2. **Code Quality and Static Analysis**
    1. Integrating code quality tools
    2. Using GitLab's built-in code quality features

Creating merge requests

1. Navigate to the Repository
2. Create a New Branch
3. Make Changes
4. Push Changes to GitLab
5. Navigate to the Repository's Merge Requests Section
6. Click on "New Merge Request”
7. Select Source and Target Branches
8. Review Changes

Monitoring and Infrastructure as Code

Monitoring and Infrastructure as Code

1. **Monitoring with GitLab**
    1. Overview of monitoring tools in GitLab
    2. Setting up monitoring for projects
2. **Infrastructure as Code with GitLab**
    1. Introduction to Infrastructure as Code (IaC)
    2. Using GitLab for IaC

Introduction to Infrastructure as Code (IaC)

1. Managing and provisioning computing infrastructure through script files rather than through physical hardware configuration or interactive configuration tools.

2. With IaC, infrastructure configurations are written in code, allowing for automated and consistent deployment, scaling, and management of infrastructure resources.

3. This approach brings several benefits to software development and IT operations.

Key Concepts of Infrastructure as Code

1. Declarative Configuration:
    1. IaC uses a declarative approach, where you specify the desired state of the infrastructure without specifying the step-by-step process to achieve that state.
2. Scripting or Configuration Files:
    1. Typically written using domain-specific languages (DSLs) or general-purpose programming languages.
    2. Common DSLs include HashiCorp Configuration Language (HCL) and YAML
3. Version Control:
    1. Manage and track changes to infrastructure configurations.
    2. This enables collaboration, code reviews, and the ability to roll back to previous states in case of issues.
4. Automation:
    1. Scripts or configuration files are executed by IaC tools to automatically provision and configure infrastructure resources.
    2. This reduces manual errors, ensures consistency, and speeds up the deployment process.

Benefits of Infrastructure as Code

1. Consistency:
2. Scalability:
3. Reproducibility:
4. Collaboration:
5. Auditability and Compliance:
6. Speed and Efficiency:

Use Cases of Infrastructure as Code

1. Cloud Resource Provisioning:
    1. Provision and manage resources in public and private cloud environments.
2. Configuration Management:
    1. Define and manage the configuration of servers and applications.
3. Continuous Integration/Continuous Deployment (CI/CD):
    1. Automate the deployment and scaling of applications in CI/CD pipelines.
4. Environment Replication:
    1. Easily replicate and recreate development, testing, and production environments.
5. Microservices Orchestration:
    1. Manage the deployment and scaling of microservices-based applications.