**What is GitLab?**

* GitLab is a web-based DevOps (Development and Operations) platform that provides a suite of tools for software development and continuous integration/continuous deployment (CI/CD).
* It is a complete version control and collaboration system that helps teams manage code repositories, track issues, and deploy applications.

🡺 GitLab is built around **Git**, a distributed version control system, and it offers a wide range of features to support the software development lifecycle, including:

1. **Source Code Management (SCM):** GitLab allows teams to manage and version control their code using Git repositories. This includes features like branching, merging, pull requests (called "merge requests" in GitLab), and code review.
2. **CI/CD Pipelines:** GitLab offers built-in continuous integration and continuous deployment tools. This allows teams to automate the process of building, testing, and deploying their code whenever changes are made.
3. **Issue Tracking and Project Management:** GitLab provides integrated issue tracking, milestones, and boards (similar to Kanban boards) to help teams manage development tasks, bugs, and feature requests. It also includes the ability to link issues to commits and merge requests.
4. **Collaboration Tools:** It provides features for team collaboration, including code reviews, discussions, and team activity tracking. Merge requests can be used for collaborative code reviews and approvals before changes are merged into the main codebase.
5. **Security Features:** GitLab has built-in security scanning tools that automatically detect vulnerabilities and security issues in your code, dependencies, and containers. This is part of GitLab's focus on DevSecOps, which integrates security practices into the DevOps pipeline.
6. **DevOps Automation:** GitLab provides automation capabilities for the entire DevOps lifecycle—from planning, coding, and building to testing, deploying, and monitoring. It also integrates with other DevOps tools and services.
7. **Self-Hosting and Cloud Hosting:** GitLab can be hosted on your own infrastructure (self-hosted) or used as a cloud service (GitLab.com). The self-hosted version allows for more control and customization, while the cloud-hosted version is easier to set up and use without worrying about infrastructure.
8. **Integration and Extensibility:** GitLab integrates with numerous third-party tools and services, and its open-source nature allows for customization and extension through APIs and plugins.

**Benefits of GitLab:**

* **Single Application for DevOps:** Unlike many other platforms that separate source code management, CI/CD, and project management, GitLab brings everything together in a single interface.
* **Collaboration and Transparency:** GitLab fosters collaboration through merge requests, issue tracking, and real-time communication features.
* **Security and Compliance:** Built-in security tools help ensure the code is secure before deployment, promoting DevSecOps practices.
* **Scalability:** GitLab can scale from small teams to large organizations with complex requirements.

**What is Artifacts in GitHub CI/CD?**

**Artifacts in GitHub CI/CD** refer to files generated during a workflow run, such as build outputs, test reports, or logs. These artifacts are stored so they can be accessed later or shared between different jobs in a pipeline.

In GitHub Actions, artifacts can be uploaded using the actions/upload-artifact action and downloaded using actions/download-artifact. This allows different jobs in a workflow to pass data between each other, making them crucial for tasks like:

* **Debugging** (by reviewing logs or test results)
* **Sharing build outputs** (like binaries or archives)
* **Persisting files** (like documentation or reports)

Artifacts are retained for up to 90 days and can be as large as 2GB.

**GitLab CICD Exit code :** **This exit codes help manage the flow of your pipeline and provide feedback on what went wrong, helping with debugging and improving your CI/CD process.  
Exit Code 0**: Success

**Exit Code 1**: General error

**Exit Code 2**: Misuse of shell builtins

**Exit Code 127**: Command not found

**Exit Code 128**: Invalid exit argument

**Exit Code 130**: Process terminated by Ctrl+C

**Exit Code 137**: Process killed by SIGKILL (often due to memory issues)

**Exit Code 255**: Exit status out of range

**What is CICD?**

**CI/CD** stands for **Continuous Integration** and **Continuous Delivery/Deployment**.

* **Continuous Integration (CI)**: Developers frequently commit code to a shared repository, triggering automated builds and tests to detect issues early.
* **Continuous Delivery (CD)**: Code is automatically prepared for deployment to production, ensuring it's always in a deployable state. Deployment is often manual.
* **Continuous Deployment (CD)**: Every successful change is automatically deployed to production without manual approval.

CI/CD helps automate the software development lifecycle, ensuring faster releases, higher quality, and fewer manual errors

A diagram of a test

Description automatically generated with medium confidence

A screen shot of a computer

Description automatically generated

**GitLab Runners:**

GitLab Runners are agents that execute jobs defined in GitLab CI/CD pipelines. They are responsible for tasks like building, testing, and deploying code automatically.

* Types of Runners:
  + Shared Runners: Provided by GitLab, available for all projects.
  + Specific Runners: Custom runners, often dedicated to a particular project or group.
* Executors: Runners can use different environments to execute jobs, such as Shell, Docker (for containerized jobs), Kubernetes, or SSH (remote servers).
* How it Works: When you push code, GitLab triggers the pipeline, and the runner picks up the job, executes it, and reports the result (success/failure) back to GitLab.
* Use Cases: GitLab Runners help automate testing, building, and deploying software, and they can scale horizontally by adding more runners for parallel job execution.
* Installation: Install GitLab Runner, register it with GitLab using a token, and configure it to run jobs.

**What is Docker?**

**Docker** is an open-source platform that enables developers to automate the deployment, scaling, and management of applications inside lightweight, portable containers. Containers allow applications and their dependencies (e.g., libraries, configurations) to run consistently across different environments.

**Key Concepts of Docker:**

1. **Containers**: Encapsulated environments where applications run. They include the app, its dependencies, and runtime but share the host system's OS kernel.
2. **Images**: Read-only templates used to create containers. Images contain everything needed to run an application, such as code, libraries, and dependencies.
3. **Docker Engine**: The core part of Docker that runs and manages containers on a host machine.
4. **Docker Hub**: A cloud-based registry where Docker images are stored and shared.

**Benefits of Docker:**

* **Portability**: Containers can run on any machine with Docker installed, ensuring the application works the same everywhere.
* **Isolation**: Each container is isolated from others, avoiding conflicts between dependencies.
* **Efficiency**: Docker containers are lightweight and start quickly because they share the host system's kernel.

Docker is widely used in development, CI/CD pipelines, and cloud environments for its ability to simplify application deployment and management.

what is different b/w cache and Artfacts in GitlabCICD

**How to setup SSH Key in GitLab.?**

To set up SSH in GitLab, you need to generate an SSH key pair on your local machine and then add the public key to your GitLab account.

**1. Generate an SSH Key Pair (if you don't have one already)**

1. Open a terminal on your local machine.
2. Run the following command to generate a new SSH key pair. Replace <your-email@example.com> with your actual email address:
3. **ssh-keygen -t rsa -b 4096 -C "<your-email@example.com>"**
   * -t rsa: Specifies the type of key to create (RSA).
   * -b 4096: Specifies the key length in bits (4096 bits is considered secure).
   * -C "<your-email@example.com>": An optional label for the key (typically your email address).
4. You will be prompted to choose a location to save the key. By default, it is saved in the following path:
5. **/home/username/.ssh/id\_rsa (Linux/macOS)**
6. **C:\Users\username\.ssh\id\_rsa (Windows)**

Press Enter to accept the default path (or specify a different path if needed).

1. You will then be prompted to enter a passphrase for the key (optional but recommended for added security). If you don't want a passphrase, just press Enter.

**2. Add the SSH Key to Your GitLab Account**

1. **Copy the SSH Public Key**: The public key file is typically located in ~/.ssh/id\_rsa.pub (on Linux/macOS) or C:\Users\username\.ssh\id\_rsa.pub (on Windows).

Run the following command to copy the content of your public key:

**cat ~/.ssh/id\_rsa.pub**

or on Windows:

type **C:\Users\username\.ssh\id\_rsa.pub**

This will display your public key. Copy the entire key (including ssh-rsa at the beginning and your email at the end).

1. **Add the Key to GitLab**:
   * Log in to your GitLab account.
   * In the top-right corner, click on your profile picture or avatar, and select **Preferences** (or **Settings**).
   * In the left sidebar, click on **SSH Keys** under the **Access Tokens** section.
   * Paste your SSH public key into the **Key** field.
   * Add a title to identify the key (e.g., "My Laptop Key").
   * Click **Add key**.

**3. Test the SSH Connection**

To test that everything is working correctly, run the following command from your terminal:

**ssh -T git@gitlab.com**

You should see a message like:

**Welcome to GitLab, @username!**

This confirms that your SSH key has been successfully added to GitLab.

**4. Configure Git to Use SSH (if needed)**

If you’ve been using HTTPS to access your GitLab repositories and now want to use SSH, you need to update the remote URL for your repository.

1. Navigate to your repository's directory on your local machine.
2. Run the following command to change the remote URL to use SSH:
3. **git remote set-url origin git@gitlab.com:username/repository.git**

Replace username/repository.git with your actual repository path.

**5. Optional: Troubleshooting**

* If you encounter issues with the SSH connection, make sure your SSH agent is running and that your private key is added to it. You can add your SSH key to the SSH agent using:
* **ssh-add ~/.ssh/id\_rsa**

On Windows, you may need to start an SSH agent first. Run:

**eval $(ssh-agent -s)**

**ssh-add C:\Users\username\.ssh\id\_rsa**

* Ensure your SSH client is using the correct key (if you have multiple keys). Check the configuration in ~/.ssh/config (or C:\Users\username\.ssh\config on Windows) if necessary.

**What is GitLab CI CD Runner Setup Gitlab Runner on Windows & Linux?**

GitLab CI/CD Runner is a tool used to execute jobs defined in a GitLab CI/CD pipeline configuration. A GitLab Runner is responsible for pulling code from the GitLab repository, running jobs defined in .gitlab-ci.yml, and reporting back the results. It can be installed and run on various platforms, including Linux and Windows.

**1. GitLab Runner on Linux**

**Step 1: Install GitLab Runner**

You can install GitLab Runner on Linux using the official package manager for your Linux distribution.

**Debian/Ubuntu-based distributions:**

1. Add the official GitLab Runner repository to your system:

* **sudo curl -L --output /etc/apt/trusted.gpg.d/gitlab.asc https://packages.gitlab.com/gpg.key**
* **sudo curl -L https://packages.gitlab.com/install/repositories/gitlab/gitlab-runner/script.deb.sh | sudo bash**

1. Install the GitLab Runner package:

* **sudo apt-get install gitlab-runner**

**Red Hat/CentOS/Fedora-based distributions:**

1. Add the official GitLab Runner repository:

* **sudo curl -L --output /etc/yum.repos.d/gitlab-runner.repo https://packages.gitlab.com/gitlab/gitlab-runner/packages/el/7/gitlab-runner-<version>-1.x86\_64.rpm/download**

1. Install the GitLab Runner package:

* **sudo yum install gitlab-runner**

**Step 2: Register the GitLab Runner**

1. To register the GitLab Runner, you need the GitLab instance URL and a registration token (this can be found under **GitLab CI/CD settings** in your GitLab project).
2. Run the following command to register the runner:

* sudo gitlab-runner register

1. During registration, you'll be prompted for several inputs:
   * **GitLab instance URL**: Enter the URL of your GitLab instance (e.g., https://gitlab.com/).
   * **Registration token**: The token that you can get from your GitLab project’s CI/CD settings.
   * **Description**: A name/description for this runner (e.g., MyLinuxRunner).
   * **Tags**: Optionally, you can specify tags to associate this runner with specific jobs.
   * **Executor**: Choose the executor type. For example, use shell for a basic shell executor.

**Step 3: Start the GitLab Runner**

Once the runner is registered, you can start the GitLab Runner service:

* **sudo gitlab-runner start**

**2. GitLab Runner on Windows**

**Step 1: Download GitLab Runner**

1. Go to the [GitLab Runner releases page](https://gitlab.com/gitlab-org/gitlab-runner/-/releases) and download the gitlab-runner-windows-amd64.exe binary for Windows.
2. Move the binary to a directory of your choice (e.g., C:\GitLab-Runner).

**Step 2: Install GitLab Runner**

To install GitLab Runner as a service:

1. Open a Command Prompt with administrative privileges.
2. Navigate to the folder where you saved the gitlab-runner-windows-amd64.exe file.
3. Install the runner as a service by running:

* **gitlab-runner.exe install**

**Step 3: Register the GitLab Runner**

1. Run the following command to register the GitLab Runner:

* **gitlab-runner.exe register**

1. You’ll be prompted for the following inputs:
   * **GitLab instance URL**: Enter the GitLab server URL (e.g., https://gitlab.com).
   * **Registration token**: Retrieve the registration token from your GitLab project’s **CI/CD settings**.
   * **Description**: Name the runner (e.g., WindowsRunner).
   * **Tags**: Optionally specify tags to associate with the runner.
   * **Executor**: Choose an executor (e.g., shell for basic Windows shell execution).

**Step 4: Start the GitLab Runner**

Once registered, you can start the GitLab Runner service:

1. Open a Command Prompt with administrative privileges.

* **gitlab-runner.exe start**

Alternatively, if you have installed GitLab Runner as a Windows service, you can start it from the **Services** management console.

**3. Configuring .gitlab-ci.yml**

Once the GitLab Runner is set up on your system (either Windows or Linux), you’ll need to configure the .gitlab-ci.yml file in your repository to define the CI/CD pipeline.

Here’s a simple example of **.gitlab-ci.yml**:

**stages:**

**- build**

**- test**

**- deploy**

**build-job:**

**stage: build**

**script:**

**- echo "Building the project..."**

**test-job:**

**stage: test**

**script:**

**- echo "Running tests..."**

**deploy-job:**

**stage: deploy**

**script:**

**- echo "Deploying the project..."**

**4. Monitoring the Runner**

Once the GitLab Runner is set up and the .gitlab-ci.yml file is in place, you can monitor the status of the runner in the **GitLab CI/CD pipelines** section of your project.

* Pipelines will automatically be triggered on code pushes, and the jobs will be picked up by your registered runners.
* You can also view job logs and error messages if something goes wrong.

**Additional Information**

* **Executor Types**:
  + **Shell**: Runs jobs in a shell (e.g., Bash on Linux or Command Prompt/Powershell on Windows).
  + **Docker**: Runs jobs in a Docker container (ideal for isolated environments).
  + **Docker-SSH**: Similar to Docker, but uses SSH to connect to the container.
  + **Kubernetes**: Executes jobs within Kubernetes clusters.
* **GitLab Runner Versions**: It's essential to ensure that you are using the latest stable version of GitLab Runner to benefit from new features and security patches.

**Troubleshooting**

* **Runner not picking up jobs**: Check the GitLab CI/CD settings to ensure the runner is correctly registered and that it matches the tags and requirements for the jobs.
* **Permission issues**: Ensure that the runner has the necessary permissions (e.g., access to the repository or system resources).

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Gitlab CICD what is different b/w Specific Runners and shared Runners?**

**Shared Runners** are GitLab Runners that are available for use across all projects within a GitLab instance or group. They are managed centrally by the GitLab instance or the administrator of a group, and they are shared by multiple projects.

**Specific Runners** (sometimes called "Private Runners") are GitLab Runners that are dedicated to specific projects or groups. These runners are typically installed and configured by the GitLab project or group administrators.

A screenshot of a table

Description automatically generated

A screenshot of a computer program

Description automatically generated

To set up a GitLab CI/CD pipeline for a Maven and Docker project, you can define the stages such as build, test, package, and deploy in a .gitlab-ci.yml file. Below is an example configuration for this setup.

**image: maven:latest**

**# Define the stages of the pipeline**

**stages:**

**- build**

**- test**

**- package**

**- deploy**

**# Define the build stage**

**build:**

**stage: build**

**script:**

**- mvn clean install**

**# Define the test stage**

**test:**

**stage: test**

**script:**

**- mvn test**

**# Define the package stage**

**package:**

**stage: package**

**script:**

**- mvn package**

**# Define the deploy stage**

**deploy:**

**stage: deploy**

**script:**

**- docker build -t my-app:latest .**

**- docker run -d -p 8080:8080 my-app:latest**

**only:**

**- master # Deploy only on the master branch**

**Explanation:**

1. **Image**: Using the maven:latest Docker image as the base for running Maven commands in all the stages.
2. **Stages**:
   * build: This step uses mvn clean install to clean and install dependencies.
   * test: Runs unit tests using mvn test.
   * package: Packages the project into a deployable artifact using mvn package.
   * deploy: This step builds a Docker image (docker build -t my-app:latest .), then runs the container (docker run) on port 8080. The deploy step is set to run only on the master branch.

**Notes:**

* Ensure you have a Dockerfile in your project for the Docker build.
* You can customize the image and script according to your specific needs, such as adding environment variables or connecting to external services.

---------------------------------------------------------------------------------------------------------

**what is different between docker Registry? Gitlab Container Registry.**

**1. Docker Registry:**

A **Docker Registry** is a generic term for a storage location where Docker images are stored and managed. It is a repository that hosts Docker images, making it easier to share images between teams or deploy applications across multiple environments.

**2. GitLab Container Registry:**

The **GitLab Container Registry** is a Docker container registry integrated directly into the GitLab platform. It is specifically designed to work seamlessly with GitLab CI/CD pipelines, enabling you to manage Docker images alongside your code and other artifacts in GitLab.

A screenshot of a computer

Description automatically generated

