Beginner's Step-by-Step Guide to GitLab CI/CD and Runner for Spring Boot and React Apps

Welcome to the complete beginner's guide for setting up GitLab CI/CD pipelines with runners for modern web applications. This comprehensive guide will walk you through every step needed to automate your development workflow, from initial setup to deployment, using two of today's most popular technologies: Spring Boot for backend services and React for frontend applications.

Introduction to GitLab CI/CD and Runners

What is CI/CD?

Continuous Integration and Continuous Deployment is a software development practice that automates the integration of code changes and deployment processes. CI ensures that code changes are regularly merged and tested, while CD automates the deployment of verified code to production environments.

GitLab CI/CD Role

GitLab CI/CD serves as the orchestration engine for your development pipeline, automatically triggering builds, running tests, and deploying applications whenever code changes are detected. It eliminates manual processes and reduces human error in software delivery.

GitLab Runner

A GitLab Runner is an application that processes jobs in your CI/CD pipeline. It can be installed on various platforms and executes the commands defined in your pipeline configuration. Runners can be shared, group-specific, or project-specific.

For Spring Boot and React projects, GitLab CI/CD provides exceptional value by automating the complex build processes required for both backend Java applications and frontend JavaScript applications. Spring Boot applications require Maven or Gradle builds, dependency management, and JAR packaging, while React applications need Node.js environments, npm package installation, and optimized production builds.

The integration becomes particularly powerful when dealing with full-stack applications where both components need to be built, tested, and deployed in coordination. GitLab's pipeline system can handle parallel builds, conditional deployments, and environment-specific configurations that are essential for professional software development.

> GitLab runners come in three types: **Shared runners** (provided by GitLab), **Group runners** (available to all projects in a group), and **Specific runners** (dedicated to individual projects). For beginners, shared runners are often sufficient to get started.

Overview of the Sample Application Setup

Spring Boot Backend Architecture

Our Spring Boot application will be built using Java 17, leveraging the latest long-term support version for optimal performance and security. We'll use either Maven or Gradle as our build tool, with Maven being slightly more beginner-friendly due to its XML-based configuration and extensive documentation.

The backend will expose REST APIs that follow RESTful principles, providing endpoints for data operations. We'll start with a simple "Hello World" endpoint and expand from there. The application will include standard Spring Boot features like auto-configuration, embedded Tomcat server, and dependency injection.

React Frontend Features

The React frontend will be created using Create React App or Vite for faster development builds. We'll use Node.js as our runtime environment with npm or yarn for package management. The application will start with a simple component structure and can be expanded with routing, state management, and API integration.

- Modern ES6+ JavaScript syntax
- Component-based architecture
- Hot reload development server
- Production-optimized builds

Project Structure

Our project will follow a monorepo approach with separate directories for backend and frontend code. This structure makes it easier to manage dependencies, share configuration, and coordinate deployments between the two components.

```
project-root/
├── backend/        # Spring Boot app
├── frontend/       # React app
├── .gitlab-ci.yml  # CI/CD configuration
└── README.md
```

Version control with Git forms the foundation of our development workflow. GitLab will serve as our remote repository, providing not only code storage but also issue tracking, merge requests, and CI/CD capabilities. Understanding Git basics like commits, branches, and merges is essential for effective collaboration and deployment automation.

The integration between your local development environment and GitLab repository creates a seamless workflow where every code push can trigger automated builds and tests, ensuring code quality and enabling rapid iteration cycles.

Step 1: Initialize Your Spring Boot Application

01

Generate Project with Spring Initializr

Navigate to **start.spring.io** and configure your project settings. Choose Maven or Gradle as your build tool, select Java 17 as your language version, and add the Spring Web dependency. This dependency provides everything needed to build REST APIs including Spring MVC, embedded Tomcat, and JSON serialization.

02

Create REST Controller

Create a simple REST controller class that demonstrates basic API functionality. This controller will serve as our testing endpoint and proof that the application is working correctly. Include a timestamp in the response to verify that the application is dynamically generating responses.

03

Build and Test Locally

Use Maven wrapper (`./mvnw spring-boot:run`) or Gradle wrapper (`./gradlew bootRun`) to build and run your application locally. These wrapper scripts ensure consistent build environments across different development machines.

04

Verify Endpoint

Open your browser and navigate to **http://localhost:8080** to verify that your Spring Boot application is running correctly. You should see your "Hello World" message with a timestamp, confirming that the embedded server is working and your controller is properly mapped.

Sample Controller Code

```
@RestController
public class HelloController {

    @GetMapping("/")
    public Map<String, String> hello() {
        Map<String, String> response = new HashMap<>();
        response.put("message", "Hello World from Spring Boot!");
        response.put("timestamp", Instant.now().toString());
        return response;
    }
}
```

Spring Boot's auto-configuration magic handles most of the setup for you, including configuring the embedded Tomcat server, setting up JSON serialization, and enabling the default error handling.

The Spring Boot framework excels at convention over configuration, meaning that most common development scenarios work out of the box without extensive setup. This approach significantly reduces the time from project creation to having a working application, which is perfect for CI/CD pipelines that need to build and deploy quickly.

When your application runs successfully locally, you'll have confidence that the same code will work in your CI/CD pipeline. The consistency between development and production environments is a key principle in DevOps practices.

Step 2: Initialize Your React Application

Create React App

Use the command `npx create-react-app my-app` to generate a new React application with all necessary dependencies and build tools preconfigured. Alternatively, consider using Vite with `npm create vite@latest my-app -- --template react` for faster builds and improved development experience.

Create Simple Component

Replace the default App component with a custom component that displays "Hello from React" along with some basic styling. This component will serve as your foundation for building more complex user interfaces and will be easily testable in your CI/CD pipeline.

Start Development Server

Run `npm start` or `yarn start` to launch the development server. This server provides hot reloading, error reporting, and automatic browser refresh during development, significantly improving your development experience.

Sample React Component

```
function App() {
  const currentTime = new Date().toLocaleString();

  return (
    <div className="App">
      <header className="App-header">
        <h1>Hello from React!</h1>
        <p>Current time: {currentTime}</p>
        <p>This app was built with Create React
App</p>
      </header>
    </div>
  );
}

export default App;
```

Development Benefits

- Hot reload for instant feedback
- Built-in ESLint for code quality
- Optimized production builds
- Modern JavaScript features
- CSS and asset processing

The development server runs on **http://localhost:3000** by default, providing a convenient environment for testing your React components and ensuring they work correctly before pushing to your GitLab repository.

React's component-based architecture makes it ideal for CI/CD pipelines because components can be individually tested, and the build process generates static files that are easy to deploy to various hosting environments. The npm ecosystem provides excellent tooling for automated testing, linting, and building production-ready applications.

Create React App abstracts away complex webpack configurations while still allowing customization when needed. This balance between simplicity and flexibility makes it an excellent choice for beginners who want to focus on building features rather than configuring build tools.

> Consider using TypeScript for larger projects to benefit from static typing and better IDE support. You can create a TypeScript React app using `npx create-react-app my-app --template typescript`.

Step 3: Create a GitLab Project and Push Your Code

1 GitLab Account Setup

Sign up for a free GitLab account at **gitlab.com** if you don't already have one. GitLab provides generous free tiers that include unlimited private repositories, CI/CD minutes, and collaboration features perfect for personal projects and small teams.

2 Create New Project

Click "New Project" in your GitLab dashboard and choose "Create blank project." Give your project a meaningful name like "spring-boot-react-app" and decide whether to make it public or private. For learning purposes, public repositories can be beneficial for showcasing your work.

3 Initialize Local Git Repository

Navigate to your project root directory in terminal and run `git init` to initialize a new Git repository. This creates the `.git` directory that tracks all changes to your project files.

4 Connect to Remote Repository

Add your GitLab repository as the remote origin using `git remote add origin <your-gitlab-repo-url>`. This establishes the connection between your local repository and GitLab's servers.

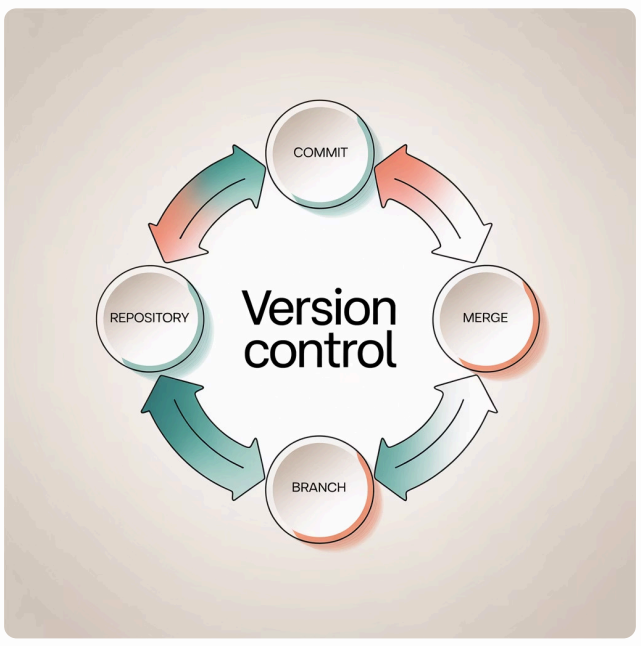
5 First Commit and Push

Stage all your files with `git add .`, create your first commit with `git commit -m "Initial commit"`, and push to GitLab using `git push -u origin main`. The `-u` flag sets up tracking between your local and remote branches.

Essential Git Commands for CI/CD

Understanding these fundamental Git commands is crucial for effective CI/CD workflows. Every push to your repository can trigger pipeline executions, making it important to commit meaningful changes with descriptive messages.

```
# Initialize repository
git init

# Add remote origin
git remote add origin https://gitlab.com/username/project.git

# Stage changes
git add .

# Commit with message
git commit -m "Add Spring Boot and React initial setup"

# Push to GitLab
git push -u origin main

# Check status
git status

# View commit history
git log --oneline
```



⚠ Always review your changes with `git status` and `git diff` before committing to avoid pushing sensitive information like API keys or passwords.

Once your code is pushed to GitLab, you'll have access to a wealth of features including issue tracking, merge requests, and most importantly for our purposes, CI/CD pipelines. GitLab's integrated approach means you can manage your entire software development lifecycle in one platform.

The repository structure should include both your Spring Boot backend and React frontend in separate directories, along with any configuration files needed for CI/CD. This monorepo approach simplifies dependency management and deployment coordination between your frontend and backend components.

Step 4: Install GitLab Runner on Your Machine or Server

Choose Runner Environment

Select the most appropriate environment for your GitLab Runner installation. Local machines are great for development and testing, cloud VMs provide scalability and isolation, while Docker containers offer the ultimate in consistency and portability across different environments.

Download Runner Binary

Visit the official GitLab Runner download page and select the appropriate binary for your operating system. GitLab provides pre-compiled binaries for Linux, Windows, and macOS, as well as package manager installations for popular distributions. https://docs.gitlab.com/runner/installation/windows/

Installation Process

Follow the platform-specific installation instructions. For Ubuntu/Debian systems, you can use the official GitLab package repository for automated updates and easier management of the runner service.

Ubuntu/Debian Installation Example

```
# Add GitLab repository
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh" | sudo bash

# Install GitLab Runner
sudo apt-get install gitlab-runner

# Check installation
gitlab-runner --version

# Register runner using authentication token (replace <your-gitlab-url> and <your-authentication-token>)
sudo gitlab-runner register \
  --url <your-gitlab-url> \
  --token <your-authentication-token> \
  --executor shell \
  --description "my-project-runner" \
  --tag-list "shell,linux"

# Start the runner service
sudo gitlab-runner start
```

Registration Parameters

- **GitLab URL:** https://gitlab.com/
- **Authentication Token:** Obtain from your GitLab instance. Navigate to your Group's Runners page or Instance Admin → CI/CD → Runners, then click "New group runner" or "New instance runner" to generate a new token.
- **Description:** Meaningful runner name
- **Tags:** Optional labels for job targeting
- **Executor:** shell, docker, kubernetes, etc.

The shell executor is simplest for beginners, while Docker executor provides better isolation and consistency across different environments.

3	2400	50000
Executor Types	**Free CI Minutes**	**Shared Runners**
Shell, Docker, and Kubernetes executors each offer different benefits for running your CI/CD jobs	GitLab.com provides generous free CI/CD minutes monthly for private projects	GitLab.com offers thousands of shared runners for immediate pipeline execution

After successful installation and registration, your GitLab Runner will appear in your project's CI/CD settings under the Runners section. The runner will poll GitLab for new jobs and execute them according to your pipeline configuration defined in the `.gitlab-ci.yml` file.

For Spring Boot and React applications, ensure your runner environment has the necessary tools installed: Java 17+ and Maven/Gradle for the backend, and Node.js with npm/yarn for the frontend. Docker executors can provide these environments automatically through container images, while shell executors require manual installation of dependencies.

With your GitLab Runner successfully installed and registered, you're now ready to create your first CI/CD pipeline configuration. The runner will execute your build, test, and deployment jobs automatically whenever you push code changes to your repository, completing the automation circle that makes modern software development so efficient and reliable.

Setting up GitLab Runner with Docker

This guide will walk you through setting up a GitLab Runner using Docker, ensuring persistent configuration and proper integration with your GitLab instance. We'll cover volume creation, runner registration, and container deployment.

1. Create a Docker Volume for Runner Configuration

First, create a Docker volume. This volume will store the GitLab Runner configuration (`config.toml`) persistently, so your settings aren't lost if the container is removed or recreated.

```
docker volume create gitlab-runner-config
```

2. Register the GitLab Runner

Next, you need to register the runner with your GitLab instance. This step links your Docker runner to a specific project or group in GitLab. You will need your GitLab instance URL and a registration token. This token can be found under your project's or group's "Settings > CI/CD > Runners" section. For new setups, use the new project access token method for registration.

Run the following command, which will guide you through an interactive registration process:

```
docker run --rm -it -v gitlab-runner-config:/etc/gitlab-runner gitlab/gitlab-runner:latest register
```

During the registration, you will be prompted for:

- Your GitLab instance URL (e.g., `https://gitlab.com`)
- The registration token (from your GitLab project/group settings)
- A description for your runner
- Tags for your runner (e.g., `docker`, `linux`)
- The executor type (choose `docker`)
- The default Docker image (e.g., `alpine/git`)

3. Run the GitLab Runner Container

After successful registration, you can start your GitLab Runner container. This command will run the runner in detached mode, restart it automatically, and mount the necessary volumes.

```
docker run -d --name gitlab-runner --restart always \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v gitlab-runner-config:/etc/gitlab-runner \
  gitlab/gitlab-runner:latest
```

Explanation of the command options:

- `-d`: Runs the container in detached mode (in the background).
- `--name gitlab-runner`: Assigns a name to your container for easy management.
- `--restart always`: Ensures the container restarts automatically if it stops or Docker is restarted.
- `-v /var/run/docker.sock:/var/run/docker.sock`: Mounts the Docker daemon socket, allowing the GitLab Runner inside the container to execute Docker commands (essential for the Docker executor).
- `-v gitlab-runner-config:/etc/gitlab-runner`: Mounts the named volume created earlier, ensuring the runner uses the `config.toml` generated during registration.
- `gitlab/gitlab-runner:latest`: Specifies the Docker image to use for the GitLab Runner.

With these steps, your GitLab Runner should now be up and running, ready to pick up CI/CD jobs from your GitLab projects.

Step 5: Verify Your GitLab Runner Installation

After setting up your GitLab Runner, it's crucial to verify that it's correctly installed, connected to your GitLab instance, and ready to pick up CI/CD jobs. This section will guide you through the verification process and offer troubleshooting tips.

1. Check Runner Status in GitLab's Web Interface

The first place to check is directly within your GitLab project or group settings. This will show you if GitLab recognizes your new runner and its current status.

- Navigate to your project or group in GitLab.
- Go to `Settings > CI/CD`.
- Expand the `Runners` section.
- You should see your newly registered runner listed. Look for its description or tags to identify it.
- Verify its status. An active, connected runner will typically show a green circle or "Online" status.

2. Verify Runner Connectivity and Operation

To ensure the runner is not only registered but also functional, you can perform a quick check of its logs or trigger a test pipeline.

Check Docker Container Logs

You can inspect the logs of your GitLab Runner Docker container to see if it's polling for jobs and if there are any immediate errors:

```
docker logs gitlab-runner
```

Look for messages indicating successful connection to GitLab and job polling activities.

Run a Test Pipeline

The most conclusive way to verify is to trigger a simple CI/CD pipeline in your GitLab project. Create a `.gitlab-ci.yml` file (or modify an existing one) with a basic job that uses one of your runner's tags:

```
test_job:
  tags:
    - <your-runner-tag>
  script:
    - echo "Hello from GitLab Runner!"
```

Commit this file to your repository. If the runner is working correctly, it should pick up this job and execute it successfully.

3. Understand Runner Status Indicators

In the GitLab UI, the status indicators provide quick insights into your runner's state:

- **Green circle/Online:** The runner is active, connected to GitLab, and ready to accept jobs.
- **Red circle/Offline:** The runner has lost connection to GitLab or is not running. This requires immediate investigation.
- **Yellow circle/Paused:** The runner is registered but has been manually paused. It will not pick up new jobs until unpaused.

4. Common Troubleshooting Tips

If your runner isn't showing as active or jobs are failing, consider these common issues:

- **Check Docker Container Status:** Ensure the `gitlab-runner` container is running. Use `docker ps`. If it's stopped, use `docker start gitlab-runner`.
- **Review Container Logs:** Use `docker logs gitlab-runner` for detailed error messages.
- **Network Connectivity:** Ensure the machine hosting the runner can access your GitLab instance's URL. Check firewalls or proxy settings.
- **Registration Token:** Double-check that the registration token used was correct and still valid.
- **Docker Socket Permissions:** Ensure the Docker socket `/var/run/docker.sock` has appropriate permissions, allowing the runner to interact with the Docker daemon.
- **Configuration File:** Inspect `/etc/gitlab-runner/config.toml` inside the volume to ensure it's correctly configured (e.g., correct GitLab URL, executor settings).

By following these steps, you can effectively verify your GitLab Runner installation and ensure a smooth CI/CD workflow.

Step 6: Create CI/CD Pipeline for Spring Boot

A robust Continuous Integration/Continuous Delivery (CI/CD) pipeline is essential for modern Spring Boot application development. It automates the process of building, testing, and deploying your application, ensuring code quality, faster release cycles, and consistent deployments. This section guides you through creating a `.gitlab-ci.yml` file for a typical Spring Boot project, covering key stages and best practices.

The `.gitlab-ci.yml` file is a YAML file placed in the root directory of your Git repository. It defines the pipeline's structure, including stages, jobs, and their respective scripts. When you push changes to your repository, GitLab automatically detects this file and executes the defined pipeline.

Understanding the Pipeline Stages

A standard CI/CD pipeline for a Spring Boot application typically involves several stages. These stages execute sequentially, with each stage containing one or more jobs that can run in parallel. Here's a common progression:

Build

Compiles source code and resolves dependencies.

Test

Runs unit, integration, and optionally end-to-end tests.

Package

Creates an executable JAR or WAR file.

Deploy

Deploys the packaged application to target environments.

Sample `.gitlab-ci.yml` for Spring Boot

Below is a comprehensive example of a `.gitlab-ci.yml` file tailored for a Maven-based Spring Boot project. Each section is explained in detail afterwards.

```
image: maven:3.9.6-eclipse-temurin-17  # Maven with Java 17 (adjust version if needed)

stages:
  - build
  - test
  - package

variables:
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"  # cache local repo
  MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version"

cache:
  key: ${CI_PROJECT_ID}
  paths:
    - .m2/repository

before_script:
  - echo "Using Java & Maven inside GitLab Runner"
  - java -version
  - mvn -v

build:
  stage: build
  script:
    - mvn $MAVEN_CLI_OPTS clean compile

test:
  stage: test
  script:
    - mvn $MAVEN_CLI_OPTS test
  artifacts:
    when: always
    reports:
      junit: target/surefire-reports/TEST-*.xml

package:
  stage: package
  script:
    - mvn $MAVEN_CLI_OPTS package -DskipTests
  artifacts:
    paths:
      - target/*.jar
    expire_in: 1 week
```

Detailed Explanation of Pipeline Components

1. Image Selection

The `image` keyword defines the Docker image that GitLab Runner will use to execute your jobs. For Spring Boot projects, an OpenJDK image with Maven pre-installed is ideal.

```
image: openjdk:17-jdk
```

This specifies using the OpenJDK 17 Development Kit image, which includes Java and Maven, providing the necessary environment for building your application.

2. Variables

Variables allow you to define parameters or configurations that can be reused across multiple jobs. This improves maintainability and ensures consistency.

- `MAVEN_CLI_OPTS`: Sets Maven to use a local settings file (`.m2/settings.xml`) and run in batch mode, which is suitable for CI environments as it suppresses interactive prompts.
- `MAVEN_OPTS`: Specifies the local Maven repository directory (`.m2/repository`). This is crucial for caching.

3. Caching Maven Dependencies

Caching Maven dependencies significantly speeds up pipeline execution. Instead of downloading dependencies on every run, they are stored and reused.

```
cache:
  paths:
    - .m2/repository
```

This instructs GitLab to cache the `.m2/repository` directory, where Maven stores all its downloaded dependencies.

4. Stages

The `stages` keyword defines the order of execution for different phases of your pipeline.

- `build`: Compiles your Java code.
- `test`: Executes your application's tests.
- `package`: Creates the executable JAR/WAR.
- `deploy_staging`: Deploys the application to a staging environment for further testing.
- `deploy_production`: Deploys the application to the live production environment.

5. Jobs and Scripts

Each job belongs to a stage and defines a set of commands to be executed.

- **`Build Job:`** Uses `mvn compile` to build the project.
- **`Test Job:`** Uses `mvn test` to run all unit tests. Ensure your tests are robust and cover critical functionality.
- **`Package Job:`** Uses `mvn -DskipTests package` to create the runnable JAR file. Tests are skipped here since they've already been run in the 'test' stage.

6. Artifact Management

Artifacts are files or directories generated by a job that you want to save for later use, such as passing them to subsequent stages or downloading them.

```
artifacts:
  paths:
    - target/*.jar
  expire_in: 1 week
```

The `package_job` stores the generated JAR file from the `target` directory. This artifact will be available for download or use by downstream jobs (like deployment jobs).

7. Deployment Options (Staging & Production)

Deployment jobs are typically environment-specific. GitLab's `environment` keyword provides deployment tracking.

- `environment`: Defines the name and URL of the environment being deployed to, which helps visualize deployments in GitLab's UI.
- `rules`: Controls when a job is executed.
  - `deploy_staging_job` runs only when changes are pushed to the `develop` branch.
  - `deploy_production_job` runs only for the `main` branch and requires a `when: manual` action, ensuring human oversight before deploying to production.
- The actual deployment scripts (`script` section) will vary depending on your infrastructure (e.g., SSH commands, Docker deployments, Kubernetes manifests, cloud provider CLI commands).

8. Runner Tags

The `tags` keyword ensures that your jobs are executed by specific GitLab Runners that have these tags assigned during registration. This is crucial if you have different runners for different purposes (e.g., Docker, Kubernetes, specific hardware).

```
tags:
  - docker
  - linux
```

This specifies that only runners tagged with "docker" and "linux" can pick up these jobs, matching the Docker-based runner we set up in previous steps.

Best Practices for Spring Boot CI/CD

- **`Keep`** `.gitlab-ci.yml` **`Clean:`** Avoid complex logic directly in the YAML. Use shell scripts or makefiles for intricate tasks.
- **`Fast Feedback:`** Prioritize fast-running jobs (build, unit tests) early in the pipeline to catch issues quickly.
- **`Separate Environments:`** Always have distinct pipelines or stages for staging and production. Never deploy directly from feature branches to production.
- **`Security Scans:`** Integrate security scanning tools (SAST, DAST) into your pipeline, ideally before deployment.
- **`Dockerize Your Application:`** For more consistent deployments, consider building a Docker image of your Spring Boot application as part of the pipeline's package stage.
- **`Secrets Management:`** Use GitLab's CI/CD variables for sensitive information like API keys or database credentials, rather than hardcoding them in the `.gitlab-ci.yml` file.

By implementing this CI/CD pipeline, your Spring Boot development workflow will become more efficient, reliable, and automated, allowing your team to deliver high-quality software faster.

Generate SSH key (if you don’t already have one)

Open Git Bash or PowerShell and run:

ssh-keygen -t ed25519 -C "your_email@example.com"

If ed25519 isn’t supported, use:

ssh-keygen -t rsa -b 4096 -C "your_email@example.com"

Press Enter to accept default location (~/.ssh/id_ed25519 or id_rsa).

Enter a passphrase (optional, recommended).

This will generate:

Private key → ~/.ssh/id_ed25519

Public key → ~/.ssh/id_ed25519.pub

1. Copy your public key

Run:

cat ~/.ssh/id_ed25519.pub

(or open the .pub file in Notepad)

Copy the whole string starting with ssh-ed25519 ....

1. Add the SSH key to GitLab

Go to GitLab → Profile (top-right avatar) → Settings → SSH Keys (direct link: https://gitlab.com/-/profile/keys)

Paste your public key.

Give it a title (like My Laptop Key).

Click Add key.

1. Test the connection

In your terminal:

ssh -T git@gitlab.com

Expected output:

Welcome to GitLab, @your-username!

1. Tell Git to use SSH instead of HTTPS

If your repo is cloned via HTTPS, update it:

git remote set-url origin git@gitlab.com:username/reponame.git

Now pushes/pulls use SSH.

Troubleshooting GitLab Runner Docker Configuration

This section provides a comprehensive guide to troubleshoot common issues encountered when configuring GitLab Runners with Docker executors, particularly for Java/Spring Boot projects. We'll cover incorrect executor settings, volume mounting problems, and provide step-by-step instructions for re-registering and configuring your runners.

Common Issues

Incorrect Executor Configuration (`docker-windows` vs. `docker`)

A frequent problem is using the wrong Docker executor. If your GitLab Runner is installed on a Windows machine but intended to run Linux containers, you must specify the `docker` executor, not `docker-windows`. The `docker-windows` executor is for running Windows containers, which is less common for typical Spring Boot projects unless explicitly required.

Volume Mounting Issues

Improper volume mounting can lead to various problems, such as slow builds due to repeated dependency downloads or inability to access necessary files within the container. Ensuring the correct cache volume and permissions is crucial for efficient builds.

Step-by-Step Troubleshooting Guide

1. Unregistering Existing Runners

Before reconfiguring, it's often best to unregister any misconfigured runners to start fresh.

```
gitlab-runner unregister --name "Your Runner Name" --url https://gitlab.com/ --token YOUR_RUNNER_TOKEN
```

Alternatively, you can unregister by passing the runner's configuration path:

```
gitlab-runner unregister --all-runners
```

Or manually delete the runner from your GitLab project/group settings under "CI/CD > Runners".

2. Re-registering GitLab Runner with Docker Executor

Now, register your runner, ensuring you select the correct executor and default Docker image.

```
gitlab-runner register
```

Follow the prompts and provide the following information:

- Enter your GitLab instance URL (e.g., `https://gitlab.com/`): [YOUR_GITLAB_URL]
- Enter the registration token: [YOUR_REGISTRATION_TOKEN]
- Enter a description for the runner (e.g., `My Docker Runner for Spring Boot`): [RUNNER_DESCRIPTION]
- Enter tags for the runner, separated by commas (e.g., `docker, java, spring-boot`): `docker, java`
- Enter the executor: `docker`
- Enter the default Docker image (e.g., `maven:3.8.7-openjdk-17-slim`): `maven:3.8.7-openjdk-17-slim`

3. Configuring Runner for Spring Boot Projects

After registration, you might need to adjust the runner's `config.toml` file (usually located in `/etc/gitlab-runner/` on Linux or `C:\GitLab-Runner\` on Windows) for optimal Spring Boot builds.

Volume Mounting for Maven Cache

To speed up Maven builds by caching dependencies, configure a dedicated volume for the Maven repository.

```
[runners.docker]
  # ... other docker settings
  volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
```

This line tells the runner to mount a Docker volume named `/cache`. GitLab Runner will automatically manage this volume for caching. If you need to mount a specific host path, change it to `"/path/on/host:/path/in/container:rw"`. For Maven, the `/cache` should correspond to the `.m2` directory within the build process.

Shared Memory Size

For large builds or complex tests, increasing shared memory size can prevent out-of-memory errors in Docker containers.

```
[runners.docker]
  # ... other docker settings
  shm_size = 256 # In MB
```

Full Example (`config.toml` snippet)

A common configuration for a Spring Boot project might look like this:

```
[[runners]]
  name = "My Docker Runner for Spring Boot"
  url = "https://gitlab.com/"
  token = "YOUR_RUNNER_TOKEN"
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "maven:3.8.7-openjdk-17-slim"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
    shm_size = 256
    pull_policy = "if-not-present" # Improves build speed by not pulling image on every run if already present
  [runners.cache]
    Type = "docker"
    Shared = true
```

After modifying `config.toml`, remember to restart the GitLab Runner service:

```
sudo gitlab-runner restart
```

4. Verification Steps

Check Runner Status

Ensure your runner is online and healthy:

```
gitlab-runner status
```

Or check its status in the GitLab UI under your project/group CI/CD settings.

Run a Test Pipeline

Create a simple `.gitlab-ci.yml` in your Spring Boot project to verify the configuration:

```
stages:
  - build

build-job:
  stage: build
  tags:
    - docker
    - java
  script:
    - echo "Building Spring Boot project with Maven..."
    - mvn clean install -DskipTests=true
    - echo "Build successful!"
```

Observe the pipeline logs to ensure it's using the correct Docker image, caching dependencies, and completing successfully without volume-related errors.

Additional Tips

- **Permissions for Volumes:** Ensure that the user running the GitLab Runner service has appropriate permissions to access any host paths mounted as volumes.
- **Docker-in-Docker (Dind):** If your Spring Boot project also involves building Docker images within the CI/CD pipeline, you might need to enable `privileged = true` and add `"/var/run/docker.sock:/var/run/docker.sock"` to your volumes in `config.toml` to allow Docker commands to run inside the runner container.

Step 1: Enter the GitLab Runner container

Run:

```
docker exec -it gitlab-runner /bin/bash
```

This opens a shell inside the container.

Step 2: Locate config.toml

Inside the container, the runner's config is usually at:

```
/etc/gitlab-runner/config.toml
```

Check if it exists:

```
ls /etc/gitlab-runner/
```

You should see config.toml.

Step 3: Edit config.toml

Open it with vi or nano (nano may not be installed):

```
vi /etc/gitlab-runner/config.toml
```

Look for your runner entry, it will look like this:

```
[[runners]]
  name = "test"
  url = "https://gitlab.com/"
  token = "YOUR_TOKEN"
  executor = "docker-windows"
  [runners.docker]
    image = "gitlab/gitlab-runner"
    volumes = ["c:\\cache"]
```

Step 4: Update executor and volume paths

Change:

```
executor = "docker-windows"
```

to:

```
executor = "docker"
```

And update volumes:

```
volumes = ["/cache"]
```

Full [runners.docker] section should look like this:

```
[runners.docker]
  tls_verify = false
  image = "gitlab/gitlab-runner"
  privileged = false
  disable_entrypoint_overwrite = false
  oom_kill_disable = false
  disable_cache = false
  volumes = ["/cache"]
```

Step 5: Save and exit

In vi, press:

```
Esc -> :wq -> Enter
```

Step 6: Restart the runner container

Exit the container:

```
exit
```

Then restart the container:

```
docker restart gitlab-runner
```

✅ Step 7: Test a job

Run a simple pipeline in GitLab to check if the runner executes without the invalid volume specification error.

If you want, I can also give you a **ready-to-copy** config.toml that will work **on Windows host with Docker executor**, so you don't have to manually edit anything.

Do you want me to prepare that?

You said:

bash: vi: command not found

ChatGPT said:

Ah! That makes sense—your GitLab Runner container doesn't have vi or nano installed. No worries, we can still edit config.toml in a few ways. Here are the easiest options:

Option 1: Edit the file from the host

Since the runner's config is mounted as a **volume on your host**, you can edit it directly on your host machine.

1. Find the container's config location:

```
docker inspect gitlab-runner | grep Config
```

Or check if you mapped a volume when creating the container:

```
docker inspect gitlab-runner --format '{{ json .Mounts }}' | jq
```

You're looking for something like:

```
/etc/gitlab-runner/config.toml -> C:\gitlab-runner\config.toml
```

1. Open that file with any text editor on your host (Notepad, VSCode, etc.).
2. Change the executor and volumes as I showed:

```
executor = "docker"
[runners.docker]
volumes = ["/cache"]
```

1. Save the file and restart the container:

```
docker restart gitlab-runner
```

Option 2: Install an editor inside the container

If you want to edit inside the container:

```
docker exec -it gitlab-runner bash
apt-get update
apt-get install -y nano
nano /etc/gitlab-runner/config.toml
```

After editing, restart the container.

Option 3: Use sed to update config directly

If you want a quick command-line fix without opening an editor:

```
docker exec -it gitlab-runner bash -c \
"sed -i 's/executor = \"docker-windows\"/executor = \"docker\"/' /etc/gitlab-runner/config.toml && \
sed -i 's#volumes = \[\"c:\\\\cache\"\]#volumes = [\"/cache\"]#' /etc/gitlab-runner/config.toml"
```

Then restart:

```
docker restart gitlab-runner
```

Sample GitLab Runner config.toml File

Below is a comprehensive example of a `config.toml` file tailored for GitLab Runners using a Docker executor, particularly useful for Spring Boot projects. Remember to replace placeholder values with your actual GitLab instance details and project-specific configurations.

```
concurrent = 1
check_interval = 0
connection_max_age = "15m0s"
shutdown_timeout = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "test"
  url = "https://gitlab.com/"
  id = 49795542
  token = "glrt-....."
  token_obtained_at = 2025-09-14T14:44:55Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  shell = "bash"
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "gitlab/gitlab-runner"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
    network_mtu = 0
    pull_policy= "always"
  [runners.docker.auth_config]
    username = "<dockerhub username>"
    password = "<dockerhub passwd>"
```

Understanding the Configuration Sections

Each section of the `config.toml` plays a crucial role:

- `concurrent` **setting:** This global setting at the top defines the maximum number of jobs your GitLab Runner instance can execute simultaneously across all its registered runners. Adjust this based on your server's resources.
- `[[runners]]` **section:** This is where individual runner configurations are defined. You can have multiple `[[runners]]` blocks for different purposes.
  - `name`: A descriptive name for your runner. Make it unique and easily identifiable.
  - `url`: The URL of your GitLab instance (e.g., `https://gitlab.com/`).
  - `token`: A unique token generated when you register the runner with your GitLab project, group, or instance. **DO NOT SHARE THIS TOKEN.**
  - `executor`: Specifies how the jobs will be executed. Here we use `"docker"`.
  - `limit`: The maximum number of concurrent jobs for this *specific* runner. This allows you to fine-tune resource allocation per runner.
- `[runners.docker]` **section:** These settings are specific to the Docker executor.
  - `image`: The default Docker image that will be used for jobs if not specified in the `.gitlab-ci.yml`. For Spring Boot, an OpenJDK image is a good starting point.
  - `privileged`: Set to `true` if your CI/CD jobs need to run Docker-in-Docker or access privileged system resources. Use with caution.
  - `volumes`: Defines volumes that will be mounted into the job container. `"/cache"` is common for caching build artifacts or dependencies.
  - `shm_size`: Sets the shared memory size. Increase if your jobs involve memory-intensive tasks like browser automation tests.
  - `memory`, `cpuset_cpus`: Optional settings for resource limiting within the Docker container. Uncomment and configure if you need to control how much memory or CPU a job can consume.
- `[runners.cache]` **section:** Configures caching for your build artifacts and dependencies, significantly speeding up pipeline execution.
  - `Type = "s3"`: Specifies Amazon S3 (or S3-compatible storage like MinIO) for distributed caching. This is highly recommended for shared runners or when you need cache persistence across runner restarts.
  - `Shared = true`: Allows multiple jobs (even across different runners if using distributed cache) to share the same cache.
  - `[runners.cache.s3]`**:** Specific S3 configuration.
    - `ServerAddress`: The endpoint for your S3 service (e.g., `s3.amazonaws.com`, or your custom MinIO host).
    - `AccessKey` / `SecretKey`: Your S3 credentials for accessing the bucket. **Keep these secure and manage them carefully.**
    - `BucketName`: The name of the S3 bucket where caches will be stored.
    - `Insecure`: Set to `true` if your S3 endpoint does not use HTTPS (e.g., for local development with MinIO). Not recommended for production.

Customization Guidance

- **Executor:** While this sample uses `docker`, GitLab Runner supports other executors like `shell`, `kubernetes`, `virtualbox`, etc. Choose the one that best fits your infrastructure.
- **Docker Image:** For Spring Boot, you might switch `image` to a specific JDK version, or even a custom image that pre-installs dependencies or tools your project needs.
- **Resource Limits:** Adjust `memory` and `cpuset_cpus` in the Docker section based on the demands of your CI jobs to prevent resource starvation or over-provisioning.
- **Caching Strategy:** If you don't use S3, you can opt for `Type = "gcs"` for Google Cloud Storage, or rely on local `Type = "in-memory"` (less persistent) or `Type = "file"` (local filesystem caching).
- **Multiple Runners:** For different project types or resource requirements, consider registering multiple runners, each with its own `[[runners]]` block and specific configurations.

Next Steps and Best Practices

Building on the foundation of your GitLab Runner configuration for Spring Boot projects, this section outlines crucial next steps and best practices to further enhance your CI/CD pipeline. These recommendations aim to improve development efficiency, ensure robust security, optimize performance, and guide you towards continuous learning and improvement.

Extending Your CI/CD Pipeline

To fully leverage your CI/CD infrastructure, consider integrating the following aspects:

- **React Frontend Integration:** If your project includes a React (or any modern JavaScript framework) frontend, extend your `.gitlab-ci.yml` to include separate build and test stages for the frontend. This might involve using a Node.js Docker image, running `npm install`, `npm test`, and `npm build`, and then packaging the static assets for deployment alongside your Spring Boot application.
- **Automated Deployment Strategies:** Move beyond manual deployments. Explore options like:
  - **Kubernetes:** Deploying your Spring Boot application as Docker containers to a Kubernetes cluster for scalability and resilience.
  - **Cloud Providers (AWS, Azure, GCP):** Utilizing cloud-specific services like AWS CodeDeploy, Azure DevOps, or Google Cloud Deploy for seamless, automated deployments.
  - **Container Registries:** Automate pushing your built Docker images to a private container registry (e.g., GitLab Container Registry, Docker Hub, AWS ECR).
- **Monitoring and Alerting:** Integrate monitoring tools into your deployment process. Set up automated checks and alerts for application performance, error rates, and resource utilization using tools like Prometheus, Grafana, ELK Stack, or cloud-native monitoring services.

Security Best Practices

Security should be a paramount concern throughout your CI/CD pipeline:

- **Secrets Management:** Never hardcode sensitive information (like S3 keys, database passwords, API tokens) directly in your `config.toml` or `.gitlab-ci.yml`. Utilize GitLab CI/CD variables, HashiCorp Vault, or cloud secrets managers.
- **Container Image Security:** Use trusted base images (e.g., official OpenJDK images). Implement image scanning tools (e.g., Trivy, Clair) within your pipeline to detect vulnerabilities in your Docker images. Regularly update images to patch known vulnerabilities.
- **Code Quality and Security Scans:** Integrate static application security testing (SAST) tools (like SonarQube, Bandit) and dynamic application security testing (DAST) tools into your pipeline to identify code vulnerabilities before deployment.
- **Least Privilege Principle:** Ensure your runner's user and the Docker containers running jobs operate with the minimum necessary permissions. Avoid using `privileged = true` in Docker executor unless absolutely essential and fully understood.

Performance Optimization

Optimize your pipeline for speed and resource efficiency:

- **Effective Caching:** Beyond basic S3 caching, fine-tune your cache keys in `.gitlab-ci.yml` to ensure caches are invalidated only when necessary. Cache dependencies (e.g., Maven `.m2` directory, npm modules) that don't change frequently.
- **Job Parallelization:** Break down large jobs into smaller, parallelizable tasks. GitLab CI/CD offers features like `parallel` and `dependencies` to manage this effectively.
- **Resource Allocation:** Monitor your runner's resource usage (CPU, memory, disk I/O). Adjust the `concurrent` setting and Docker resource limits (`memory`, `cpuset_cpus`, `shm_size`) in `config.toml` to match job requirements and prevent bottlenecks.
- **Optimized Dockerfiles:** Create efficient Docker images for your Spring Boot application by using multi-stage builds to keep the final image small and fast to pull.

Further Learning and Resources

Continuous learning is key in the evolving CI/CD landscape:

- **GitLab Documentation:** The official GitLab documentation is an invaluable resource for advanced runner configurations, CI/CD syntax, and best practices.
- **Community Forums:** Engage with the GitLab community to learn from others' experiences and contribute your own insights.
- **Industry Blogs & Tutorials:** Stay updated with blogs and tutorials from DevOps experts on topics like containerization, cloud-native development, and CI/CD.
- **Certifications:** Consider pursuing certifications in cloud platforms or DevOps tools to deepen your expertise.

By implementing these next steps and best practices, you can transform your basic GitLab Runner setup into a robust, secure, and highly efficient CI/CD pipeline, driving faster and more reliable software delivery for your Spring Boot projects.