GitLab CI/CD Cheatsheet

A comprehensive cheat sheet for GitLab CI/CD, covering essential concepts, syntax, and best practices for automating your software development pipeline.

GitLab CI/CD Basics

Core Concepts

CI/CD: Continuous Integration and Continuous Delivery/Deployment. Automates the software development lifecycle.

GitLab CI/CD: Integrated CI/CD tool within GitLab for building, testing, and deploying code.

.gitlab-ci.yml: Configuration file defining the CI/CD pipeline. Located in the root of your repository.

Pipeline: A set of stages and jobs defining the CI/CD process.

Stage: A logical division within a pipeline. Stages run sequentially.

Job: An individual task within a stage. Jobs run in parallel within a stage.

Runner: Executes the jobs defined in the `.gitlab-ci.yml` file. Can be shared or specific to a project/group.

Artifacts: Files or directories generated by a job that can be used by subsequent jobs or downloaded.

.gitlab-ci.yml Structure

```yaml
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building..."
    - ./build_script.sh

test_job:
  stage: test
  script:
    - echo "Testing..."
    - ./test_script.sh

deploy_job:
  stage: deploy
  script:
    - echo "Deploying..."
    - ./deploy_script.sh
```

Key Directives

`stages`	Defines the stages of the pipeline (e.g., build, test, deploy).
`image`	Specifies the Docker image to use for the job.
`script`	Commands to execute within the job.
`stage`	Assigns the job to a specific stage.
`only` / `except`	Controls when a job runs based on branch, tags, etc.
`variables`	Defines environment variables for the job.

Advanced Configuration

Variables

Define variables in `.gitlab-ci.yml`:	`variables:`
	`  MAVEN_CLI_OPTS: "-s .m2/settings.xml --batch-mode"`

Precedence (highest to lowest):	CI/CD variables -> Project variables -> Group variables -> Instance variables
Masked variables:	Sensitive variables can be masked in the GitLab UI to prevent them from being printed in job logs.

Artifacts

```yaml
job_name:
  stage: ...
  script: ...
  artifacts:
    paths:
      - path/to/artifact1
      - path/to/artifact2
    expire_in: 1 week
```

`paths` - Specifies the files/directories to store as artifacts.

`expire_in` - Sets the expiration time for the artifacts.

Artifacts can be downloaded or passed to subsequent jobs.

Caching

```yaml
cache:
  key: "$CI_COMMIT_REF_SLUG"
  paths:
    - .m2/repository
```

`key` - A unique key for the cache. Using `$CI_COMMIT_REF_SLUG` caches per branch.

`paths` - Specifies the directories to cache.

Caching can significantly speed up build times by reusing dependencies and build outputs.

Conditional Execution & Triggers

Only/Except

`only`	Run job only for specified refs (branches, tags).
`except`	Run job for all refs except specified ones.

Example:

```
job_name:
  stage: ...
  script: ...
  only:
    - main
    - tags
```

Rules

More flexible conditional execution based on various conditions.

```
job_name:
  stage: ...
  script: ...
  rules:
    - if: '$CI_PIPELINE_SOURCE ==
"merge_request_event"'
      when: always
    - when: never
```

`if` - Specifies the condition.

`when` - Specifies when to run the job (`always`, `on_success`, `on_failure`, `manual`, `delayed`, `never`).

Pipeline Triggers

Trigger pipelines from other pipelines or external sources.

```
trigger_job:
  stage: deploy
  trigger:
    project: group/project
    branch: main
```

Use `trigger:` to specify the project and branch to trigger.

Best Practices & Tips

Security

Use masked variables for sensitive information (passwords, API keys).

Avoid storing secrets directly in `.gitlab-ci.yml`.

Regularly audit your CI/CD configuration.

Use GitLab's security scanning tools to identify vulnerabilities in your code and dependencies.

Performance

Use caching to reduce build times.

Optimize your Docker images for size and performance.

Run jobs in parallel whenever possible.

Use GitLab Runner autoscaling to dynamically scale your runner infrastructure based on demand.

Maintainability

Keep your `.gitlab-ci.yml` file organized and well-documented.

Use templates to reuse common CI/CD configurations across multiple projects.

Regularly update your CI/CD configuration to take advantage of new features and improvements.

Test your CI/CD pipeline thoroughly to ensure it is working as expected.