

Collaborating & Automation

Exploring GitLab

Outline

- Sharing & Collaborating via remotes
- The GitLab flow (Merge Requests)
- Project Management in GitLab
- Forks (Intro)
- Automating with GitLab CI/CD
- GitLab Pages (Intro)
- Auto DevOps

Remotes

Local references to external repositories

- You'll reference one or more *remotes* from your local repository
- Not a live link / No auto-synchronization

```
# add a remote called "origin"
git remote add origin git@gitlab.com:rm-training/a.git

# inspect
git remote -vv

# upload a branch
git push origin my-branch
```

GitLab

A platform for the entire software development lifecycle

- Very full-featured
- Open-source, enterprise and on-site available
- Release updates monthly
- Devops focused solutions

Alternatives include: GitHub, Bitbucket, Gerrit, etc...

High level features

- Project Management
- Code Management
 - Code review
 - Common workflow
- Web IDE
- Tight CI/CD Integration
 - Reporting & Monitoring
 - DevOps

Demo: I'll share my repository

Let's explore GitLab while I push up my personal repository

To create a new repository...

1. Create an empty repository on the host
2. Add a *remote reference* called `origin`
3. *Push* your `master` branch up

```
git remote add <remote name> <url of repo>
git push <remote name> <branch name>
```

Optionally *track* remote branches

```
git push origin master -u
```

Project management in GitLab

Teams can be organized around projects, groups and sub-groups

GitLab Projects

Projects represent individual code repositories

- Can have own **Issues, Boards and Milestones**
- Releases map to Git `tag`s
- Merge Requests
- CI/CD settings

Repository Rule-Setting

You should define who can push and which branches require special approval before merging

- Protect special branches
- Require reviews
- Limit who can push to where
- *Auto-sync* across other repositories

SSH vs HTTPS

You'll likely want to [set up your SSH keys](#) if you haven't already

Lab: Share your repository

Make sure you have an account!

1. Create a new repository on the host, called "About Me"

- Do not *initialize* with a readme

2. Reference the remote in your local repository

```
git remote add origin <url>
```

3. Push your master branch up, with tracking enabled

```
git push origin master -u
```

4. Visit the project on GitLab! Refresh to see the updates.

5. Locally, list your remotes and all branches

```
git branch -a
```

```
git remote -vv
```

push to share

Push uploads only the necessary commits reachable in the branch you push

- Typically one branch at a time (but you can push many)
- Keep branch names the same
- *It does not matter where you are when you push*

```
git push origin bug-1  
git push upstream master  
  
# send all  
git push origin --all
```

fetch to refresh local data

Fetch updates your repository with the latest data from the remote

- **It does not merge anything**
 - You'll need to merge manually
- *Safe to do from anywhere*

```
git fetch origin  
  
# then update your local master  
git checkout master  
git merge origin/master
```

pull to update a branch

Pull will get the latest data and update the branch you are on

- It does a `fetch` and then `merge` to your current branch
- *It does matter where you are - it affects the branch you are on!*

```
# update master  
git checkout master  
git pull origin master
```

```
# update a branch you are working on  
git checkout 1-my-profile  
git pull origin master
```

```
# update a branch with changes from your team mate  
git checkout 2-our-index  
git pull origin 2-our-index
```

Remote Branches

Remote branches can be *referenced* locally but they are not really branches for you to work on...

- The *refs* look like `remote-name/branch-name`
- Need to be pruned manually
- Not a branch - more like a "tag"

```
git branch --all  
git branch --remote  
  
git fetch origin --prune  
git push origin --delete 1-feature-work
```

Want to work on a remote branch?

Just check it out, git will auto-create a local copy and set up tracking, too.

```
$> git branch -a
master
remotes/origin/master
remotes/origin/5-feature-work

$> git checkout 5-feature-work
Branch 'pipeline' set up to track remote branch...
Switched to a new branch 'pipeline'
```

Tracking Branches

Indicates that a branch is *related* to another branch on the remote

- Does not auto-sync
- Gives you extra info & shortcuts
- Can configure through `branch` or `push`

```
git pull # error! needs tracking  
  
git branch --track 1-bug origin/1-bug  
git pull # success!  
  
# view tracking details of your branches  
git branch -vv
```

Sharing tags

Tags must be explicitly pushed or configured to do so

```
# push one by one  
git push origin v1  
  
# push all reachable tags  
git push origin --tags  
  
# configure git to always include tags  
git config push.followTags true
```

Merge Requests

Merge (or Pull) Requests encapsulate a set of changes (in a branch) you want to introduce to the project

- "Please merge my branch"
- Enables discussion, code review & automation around that branch
- Issues can be related to *merge requests*, too
- You can mark it `WIP` in *GitLab*
- aka *Pull Requests* on other platforms

Quiz

1. How do you share your branch?
2. How do you get your `master` up to date?
3. How do you get your `1-super-feature` branch up to date with the latest from `master` ?
4. What is the difference between `fetch` and `pull`
5. Does it matter what branch you're on?
 - when you `fetch`
 - when you `pull`
 - when you `push`

Lab: Push & Pull (the GitLab Flow)

On your personal project, make a change and get it merged with a Merge Request, then update your local `master`

Sharing work

- Make a change in your local repo on a new branch, pushing it up to the remote.

Integrate your work

- Open a pull/merge request into `master`, review and merge it on GitLab

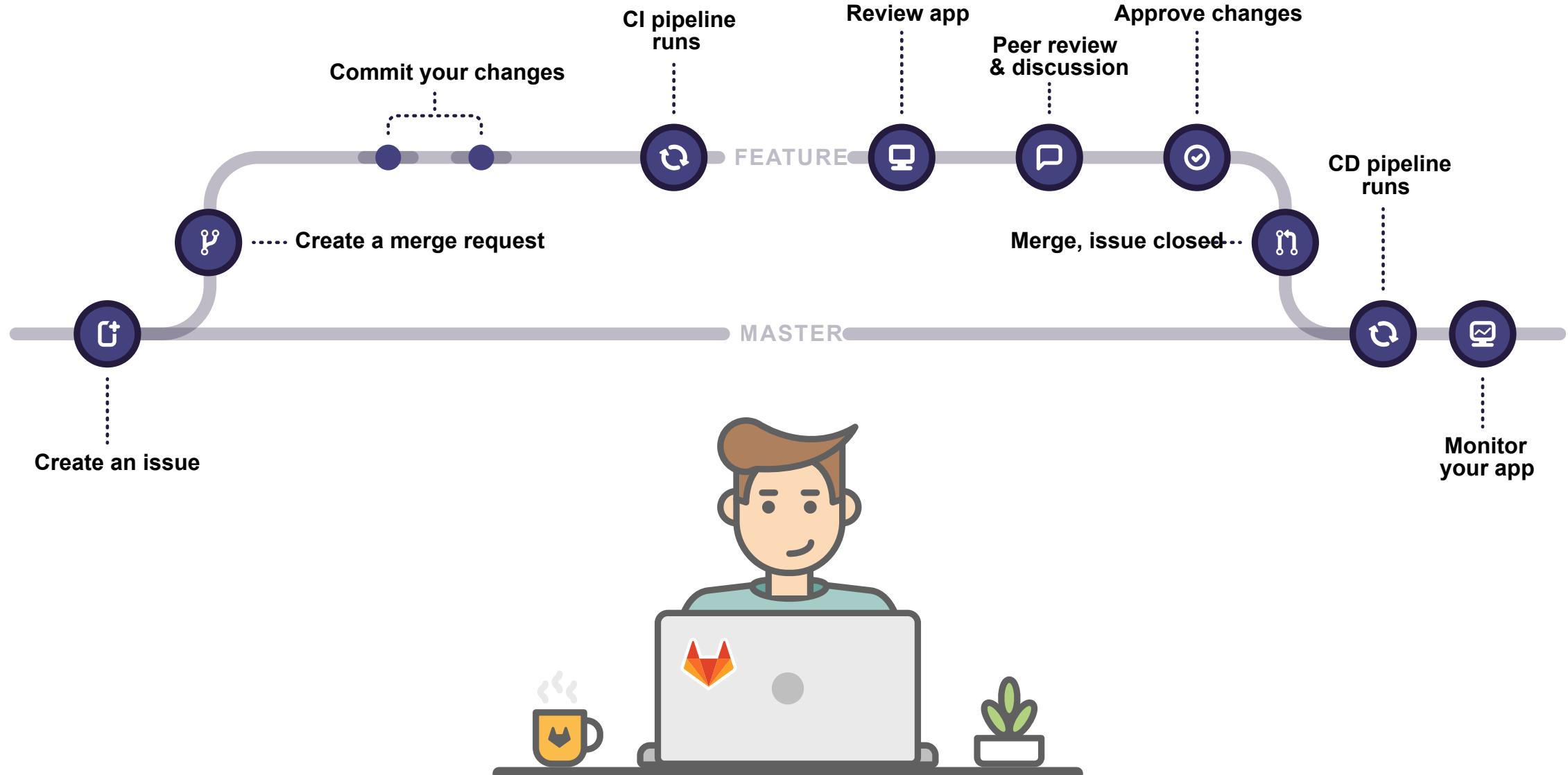
Get your local up to date

- Use `pull` to update your local `master`

Bonus

- Delete your branch & prune the remote

This is the GitLab Flow



Remote vs Local

Two copies of the repository are now at play

Our **local** keeps track of the remote's branches

The **remote** has no idea we exist...

next up

Collaborating in a repository

Demo - Team Group and a project

I'll set up a public sub-group & "About Us" project for us to work in

- Define some branch & merge rules
- Add some tasks to our board, maybe milestones
 - #1 "Add a Readme" <-- *I'll do this one*
 - #2 "Add Ryan's Profile"
 - #3 "Add an [index.md](#)"
 - #4 "Add your name to the index"
 - #5 "Build HTML & Deploy"

When I'm done, you should: Request Access to the Sub Group!

GitLab Groups

Groups serve to assemble related projects in a "folder-like" structure

- Give access to several projects at once
- Can be nested to match org structure
- They own Epics (along with boards & milestones)

We'll use a group to define a "team" with their own project(s) that they can access

Namespaces in GitLab

Projects will exist in group or user namespaces

```
http://gitlab.example.com/username  
http://gitlab.example.com/groupname  
http://gitlab.example.com/groupname/subgroup_name
```

Issue Management

- Epics > Milestones (sprints)
- Issues
 - Support markdown, special actions
 - `/tableflip`
- Labels
- Boards

Some features are only available at both a *project* and *group* level.

clone to copy a repository

Clone will download a full copy of a repository from a remote

- Sets up `master` branch and `origin` remote
- You can specify a target folder

```
git clone git@gitlab.com/user/repo_name  
cd repo_name
```

Lab: Add & Resolve an Issue

- Make a new issue in the **About Us** Project and *assign it to yourself*
 - Ex: `Add my <name> Profile`
- Go local and do the work for the issue
 - `clone` the team's "About Us" repository
 - Create a new branch from `master`, using the issue # in the name
 - Do your work and `push` it up
- On GitLab, create a **Merge Request**
 - Request to merge your branch into `master`
- Go *review* your teammates' Merge Requests

Don't merge anything -- I'll merge

Lab: Update your branch

Pretend I reviewed your code and requested a small change, ex: add your favorite animal or clean up some formatting.

You need to get your merge request updated with the change, so...

- Make the tweak locally using the same branch
- Push it up to the same branch
- Refresh your merge request in your browser

Keeping your branches up to date

- Update master?
 - Checkout master and do a `git pull`
- Get the latest master?
 - Merge the remote `master` into your topic branch
 - You can also pull from the `origin master`
- Get updates from the topic branch itself?
 - Merge the remote topic branch into your local topic branch
 - You can also do a `git pull` if you have tracking

Demo: I'll add an `index.md` file if I haven't already

Lab: Update & Collaborate

Add your name to the *list* in the `index.md` file

- Make sure your local `master` is up to date

```
git checkout master  
git pull
```

- Then do the work, making sure to `branch` off `master`
- Finally, share your work
 - `push` it up and open a merge/pull request

Don't merge it

Conflicts with Remotes

When you get a conflict in a merge/pull request, you'll need to make the fix locally

You need to introduce the conflict locally so you can resolve it locally, then push it up

- Checkout your topic branch that won't merge cleanly into master
- merge in the latest copy of master , causing the conflict
- Resolve the conflict, stage, commit
- Finally, push up the updated branch

Lab: Dealing with conflicts

Fix the conflict and update your merge/pull request

- Make sure `master` is up to date
- Update your branch to cause the conflict
 - Merge `master` into your branch
 - or simply `pull origin master` into your branch
- Resolve the conflict
 - Fix, stage, commit
- Share
 - Push it up!
 - View your updated merge/pull request

Forks

Forks are full copies of a repository in the hosting platform

- Like a "clone" across platforms or accounts
 - But forks maintain a relationship, clones do not
- Break one project down into many sub-projects
- Copy another team's project so you can use it or contribute to it
- Typical of open source contributors
- GitLab/GitHub thing, not a git thing

Typical fork flow

Your personal fork will act as a *go-between* for you to contribute to the "main" repository that you don't own

```
Main Repo    -> Fork to my account -> Clone to local  
(upstream)          (my origin)
```

- push to your personal origin
- Merge Request into the main (upstream) repo
- pull from the main repo, the upstream

GitLab Repository Sync

Similar to a fork - but it automatically pulls or pushes between two repository's branches.

Will fail if the main branch `master` diverges.

You can force override *all other* branches.

next up

GitLab Pages

GitLab Pages

Host a static site on GitLab:

```
<username or groupname>.gitlab.io/<project name>
```

- Lots of templates out there
- Anything in a `./public` folder will be exposed
 - If you build this folder you'll need to export as an `artifact`
- Requires GitLab CI & a `job` labeled `pages`

Getting started with Pages

Create a `.gitlab-ci.yml` file

```
image: alpine:latest

pages:
  stage: deploy
  script:
    - echo 'Nothing to do...'
artifacts:
  paths:
    - public
only:
  - master
```

We could get our team repo to build HTML and deploy to "pages"

Demo: Team Pipeline

| Let's get a pipeline going for our team project

- I want to build html from our markdown
- I want to deploy it to "GitLab Pages"

These have working builds & pipelines I could steal from:

<https://gitlab.com/rm-training/fork-me-for-the-team>

<https://gitlab.com/rm-training/demo-about-me>

next up

CI/CD in GitLab

Modern Software Lifecycle

We want to plan, implement and release to market as quickly as possible, then collect feedback, rinse and repeat.

And to do it with:

- Speed
- Agility
- Stability
- Flexibility

How do we get there?

At the DevOps level

- Tightly integrate dev & operations teams
- Automate *All The Things*
- Keep configuration close to code
- Treat infrastructure like code

And at the feature/code level

- Make small units of changes
- Which are easier to review, test and release
- Merge quickly to avoid conflicts
- Reduce risk and increase speed to market

Continuously

The faster we can release, the more feedback we get and the better we can respond.

- Integrate
- Build, Test, Analyze
- Deploy

There are a lot of things we may want to do:

- Write code
- Run tests
- Check code quality
- Check for security issues
- Containerize
- Build
- Deploy
- Monitor

And we want to do it *frequently* and *quickly*

GitLab wants to be the central platform for your entire software development pipeline from code to deploy & monitoring

Unify your development and operations teams on one platform.

Reduce and remove "brittle" connections between many disparate platforms/services.

"spend more time writing code, less time maintaining the tool chain"

One end-to-end tool

We'll need a few additional tools to pull this off in GitLab

- YAML (for configs)
 - And knowledge of their pipelines
- Basic knowledge of containers (ie: docker)
- Going cloud native (containerize) will make this easier

Containerization

- *Virtualized* instances of your application bundled with all dependencies and services
- Cloud-native (but doesn't have to be)
- Docker & Kubernetes

YAML

YAML Ain't Markup Language

- Flexible and data-oriented
- Indentation matters!
- YAML primer [here](#) and [here](#)

YAML Example

```
# comment line
# use 2 spaces to indent
a_nested_map:
  key: value
  another_key: value

# inline sequence
[milk, eggs, juice]

a_sequence_or_array:
  - Item 1
  - Item 2
  -
    - Another
    - sequence indice

data: |
  Block of content
  One new lines
```

GitLab Pipelines

Just requires a `.gitlab-ci.yml` file to be present.

Broken down into *stages* & *jobs*

- Stages
 - Logical grouping of work
 - ie: build, test, deploy
- Jobs
 - Each stage can contain one or more
 - Can pass files between stages

Demo: CI/CD with Pipelines

I'll set up a basic pipeline while we learn some of the configurations

I can force a failure: `test -f README.md && return 1 || return 0`

Stages

- `stages` map defines order of stages
 - defaults to `build`, `test` and `deploy`
- The default `stage` for any *job* is `test`
- You can skip with `.`
- `.pre` and `.post` are always first/last

```
stages:  
  - build  
  - test  
  - deploy  
  
magical_flower:  
  stage: build  
  script: "echo 1"
```

Jobs

- Run in parallel
- When all jobs are complete, next stage begins
- A job will fail if any non-0 value is returned
 - But jobs can be allowed to fail
- Must contain `script` clause
- No limit to the # of jobs
- Read up on [configuration parameters](#)

Example jobs

```
job1:  
  stage: test  
  script: "./build.sh"  
  allow_failure: true  
  
test:  
  script:  
    - echo "OK, Going to test"  
    - npm install  
    - npm test
```

Pipeline Images

Can define a container image for the entire pipeline, or per job.

```
image: alpine

test:
  image: node:latest
  script:
    - echo "OK, Going to test"
    - npm install
    - npm test
```

Need custom dockerfiles each build? [Use the docker-in-docker image](#)

Control *when* a job or stage runs

`when` controls when the job runs, such as `manual`, `delayed`, etc...

```
cleanup_job:
  stage: cleanup
  script:
    - ./cleanup.sh
  when: on_failure

deploy:
  stage: deploy
  environment: production
  when: manual
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --api-key=$HEROKU_API_KEY --app=gitlab-demo-ci
```

Caching between stages

```
cache:  
  key: ${CI_COMMIT_REF_SLUG}  
  paths:  
    - node_modules/  
  
before_script:  
  - yarn install
```

Artifacts

- You can pass data between Stages (not jobs)
- Can be set to expire
- Can be downloaded or pushed into reporting tools
- [Read more in the docs](#)

```
build:  
  stage: build  
  script: make build  
  artifacts:  
    - build  
    - .config
```

[Read more in the docs](#)

Only / Except

- Can run jobs or stages *only* on specific branches
 - `only` defines when the job will run
 - `except` defines when it will *not* run
- They are inclusive (you can use both together)
- Default `only` is `['branches', 'tags']`

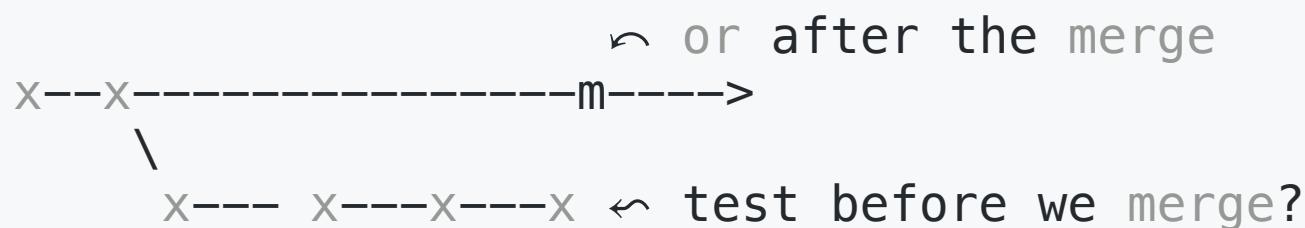
```
review:  
only:  
  - master
```

Warning: may be deprecated in favor of `rules`

Pipelines for Merge Requests

Protected against "race condition" conflicts

- Set `only: merge_requests` and it will detach the merge request and test it (not testing the result)
- In combo with `Attempt to test the merge result` it will test the merge between the MR and master



Merge Trains

Ability to queue a chain of merge requests, each built against the previous merge

- Requires "pipelines for merge results" enabled
- Queues merges but *immediately* begins their pipelines on the result
 - This can be wasteful if you skip the queue
- Each merge will be done against the previous branch in the chain
- Max of 20 will be run in parallel

[Read more](#)

Environment Vars in the Pipeline

These can be defined per project in the UI

As well as within the pipelines for all stages, or individual stages and jobs.

```
variables:  
TEST: "HELLO WORLD"
```

Deploying

- You can deploy anywhere!
 - S3, EC2 on AWS
 - Heroku
 - Google Cloud
 - Azure
 - bare metal...
- Tight integration with Kubernetes

Example: Deploying with DPL

- Use **DPL** as your deployment cli tool

```
staging:  
  stage: deploy  
  script:  
    - gem install dpl  
    - dpl --provider=heroku --app=my-app-staging --api-key=$HEROKU_STAGING_API_KEY
```

Example: Deploy to AWS

- Use the AWS image to get cli commands
- Add your AWS key/secret to your env vars
- Specify the image for your deployment

```
deploy:  
  stage: deploy  
  image: registry.gitlab.com/gitlab-org/cloud-deploy:latest # see the note below  
  script:  
    - aws s3 ...  
    - aws create-deployment ...
```

[Read the docs](#)

Example: Deploy to Google App Engine

- Create a service account role in google App engine
- Create a JSON key and add two env vars, PROJECT_ID & SERVICE_ACCOUNT

```
image: google/cloud-sdk:alpine

deploy_production:
  stage: deploy
  environment: Production
  only:
    - master
  script:
    - echo $SERVICE_ACCOUNT > /tmp/$CI_PIPELINE_ID.json
    - gcloud auth activate-service-account --key-file /tmp/$CI_PIPELINE_ID.json
    - gcloud --quiet --project $PROJECT_ID app deploy app.yaml dispatch.yaml
```

(Read the docs) [<https://medium.com/google-cloud/automatically-deploy-to-google-app-engine-with-gitlab-ci-d1c7237cbe11>]

Lab: Basic Pipeline Playground

Create a basic pipeline

- Start a new Project
- Create the `.gitlab-ci.yml` file
- Set up 3 `stages` with at least 5 `jobs`
- Pass an `index.html` through each stage
- Have it "deploy to pages"
 - By putting html files in `./public`

Bonus:

- Make the deploy step `manual`
- Add a merge request that fails the pipeline

GitLab Runners

- Executes pipeline *Jobs*
- Open Source
- Scalable

Runner Types

- Shared (on [GitLab.com](#))
 - but may be slow, unavailable and uses "minutes"
 - may want a more secure environment
- Local
- Self-hosted

Running Runners

- You can install them locally or on another instance
- Runner tags can define runners that only handle specific jobs
- Can use a runner locally to test your CI config

next up

Auto DevOps

| Zero-config CI/CD

Demo: Auto DevOps

I'll set up a new project from the Node/Express template and turn on Auto DevOps

- View the pipeline(s) created
- Make a merge request to edit the title of the site

I have a project that is ready to look at as an "end state" of the demo:

[Demo Auto DevOps with Node](#)

Auto DevOps

- Pretty awesome... when your app fits into the requirements
 - Deploys automatically into Kubernetes clusters (GKE, Amazon)
- Can be customized & extended
- Uses Herokuish and buildpacks to detect and auto-build, test, etc...
- "Best Practices" built-in

- Auto Build
- Auto Test
- Auto Code Quality
- Auto SAST (Static Application Security Testing)
- Auto Dependency Scan
- Auto License Compliance Scan
- Auto Container Scanning
- Auto "Review Apps"
- Auto DAST (Dynamic...)
- Auto Browser Performance Testing
- Auto Deploy
- Auto Cleanup

[Read up](#)

Extending Auto DevOps

Easy to customize and extend pipelines

Copy [the Auto DevOps template](#) as needed.

```
include:
  - template: Auto-DevOps.gitlab-ci.yml

build_merge:
  stage: build
  extends: build
  only:
    - merge_requests
```

You can also disable jobs via ENV vars, ex `CODE_QUALITY_DISABLED`

Define your own Dockerfile

Auto DevOps can be limiting, if you have your own Dockerfile, [GitLab will use that](#).

But you may need the *Container Registry* enabled.

Example Dockerfile

```
FROM node:8.11-alpine

WORKDIR /usr/src/app

ARG NODE_ENV
ENV NODE_ENV $NODE_ENV

COPY package.json /usr/src/app/
RUN npm install

COPY . /usr/src/app

ENV PORT 5000
EXPOSE $PORT
CMD [ "npm", "start" ]
```

Deploying to Kubernetes Clusters

Kubernetes

- Can use Docker, or other container systems
- Automates provisioning, load balancing, etc
- Manages a collection of *nodes*
- Cluster
 - Node(s) (Worker Machine)
 - Pod(s)