Lab: Create a GitLab Pages website from scratc

This lab shows you how to create a Pages site from scratch using the Jekyll Static Site Generator (SSG). You start with a blank project and create your own CI/CD configuration file, which gives instructions to a [runner]. When your CI/CD [pipeline] runs, the Pages site is created.

This example uses Jekyll, but other SSGs follow similar steps. You do not need to be familiar with Jekyll or SSGs to complete this lab.

To create a GitLab Pages website:

- Step 1: Create the project files
- Step 2: Choose a Docker image
- Step 3: Install Jekyll
- Step 4: Specify the `public` directory for output
- Step 5: Specify the `public` directory for artifacts
- Step 6: Deploy and view your website

Prerequisites

You must have a blank project in GitLab.

Create the project files

Create three files in the root (top-level) directory:

- `.gitlab-ci.yml` : A YAML file that contains the commands you want to run. For now, leave the file's contents blank.
- `index.html` : An HTML file you can populate with whatever HTML content you'd like, for example:

  ```
  <html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
  </html>
  ```

- `Gemfile` : A file that describes dependencies for Ruby programs.

  Populate it with this content:

  ```
  source "https://rubygems.org"

  gem "jekyll"
  ```

Choose a Docker image

In this example, the runner uses a [Docker image] to run scripts and deploy the site.

This specific Ruby image is maintained on DockerHub.

Edit your `.gitlab-ci.yml` file and add this text as the first line:

```
image: ruby:2.7
```

If your SSG needs NodeJS to build, you must specify an image that contains NodeJS as part of its file system. For example, for a Hexo site, you can use `image: node:12.17.0`.

Install Jekyll

To run Jekyll in your project, edit the `.gitlab-ci.yml` file and add the installation commands:

```
script:
  - gem install bundler
  - bundle install
  - bundle exec jekyll build
```

In addition, in the `.gitlab-ci.yml` file, each `script` is organized by a `job`. A `job` includes the scripts and settings you want to apply to that specific task.

```
job:
  script:
  - gem install bundler
  - bundle install
  - bundle exec jekyll build
```

For GitLab Pages, this `job` has a specific name, called `pages`. This setting tells the runner you want the job to deploy your website with GitLab Pages:

```
pages:
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build
```

Specify the `public` directory for output

Jekyll needs to know where to generate its output. GitLab Pages only considers files in a directory called `public`.

Jekyll uses a destination flag (`-d`) to specify an output directory for the built website. Add the destination to your `.gitlab-ci.yml` file:

```
pages:
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
```

Specify the `public` directory for artifacts

Now that Jekyll has output the files to the `public` directory, the runner needs to know where to get them. The artifacts are stored in the `public` directory:

```
pages:
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
```

Your `.gitlab-ci.yml` file should now look like this:

```
image: ruby:2.7

pages:
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
```

Deploy and view your website

After you have completed the preceding steps, deploy your website:

1. Save and commit the `.gitlab-ci.yml` file.
2. Go to **CI/CD > Pipelines** to watch the pipeline.
3. When the pipeline is finished, go to **Settings > Pages** to find the link to your Pages website.

   If this path is not visible, select **Deployments > Pages**. [This location is part of an experiment]. When this `pages` job completes successfully, a special `pages:deploy` job appears in the pipeline view. It prepares the content of the website for the GitLab Pages daemon. GitLab runs it in the background and doesn't use a runner.

Deploy specific branches to a Pages site

You may want to deploy to a Pages site only from specific branches.

First, add a `workflow` section to force the pipeline to run only when changes are pushed to branches:

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

pages:
  script:
    - gem install bundler
    - bundle install
```

```
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
```

Then configure the pipeline to run the job for the [default branch] (here, `main`) only.

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

pages:
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

Specify a stage to deploy

There are three default stages for GitLab CI/CD: build, test, and deploy.

If you want to test your script and check the built site before deploying to production, you can run the test exactly as it runs when you push to your [default branch] (here, `main`).

To specify a stage for your job to run in, add a `stage` line to your CI file:

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

pages:
  stage: deploy
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  environment: production
```

Now add another job to the CI file, telling it to test every push to every branch **except** the `main` branch:

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

pages:
  stage: deploy
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  environment: production

test:
  stage: test
  script:
    - gem install bundler
    - bundle install
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: $CI_COMMIT_BRANCH != "main"
```

When the `test` job runs in the `test` stage, Jekyll builds the site in a directory called `test`. The job affects all branches except `main`.

When you apply stages to different jobs, every job in the same stage builds in parallel. If your web application needs more than one test before being deployed, you can run all your tests at the same time.

Remove duplicate commands

To avoid duplicating the same scripts in every job, you can add them to a `before_script` section.

In the example, `gem install bundler` and `bundle install` were running for both jobs, `pages` and `test`.

Move these commands to a `before_script` section:

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

before_script:
```

```
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  environment: production

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: $CI_COMMIT_BRANCH != "main"
```

Build faster with cached dependencies

To build faster, you can cache the installation files for your project's dependencies by using the `cache` parameter.

This example caches Jekyll dependencies in a `vendor` directory when you run `bundle install`:

```
image: ruby:2.7

workflow:
  rules:
    - if: $CI_COMMIT_BRANCH

cache:
  paths:
    - vendor/

before_script:
  - gem install bundler
  - bundle install --path vendor

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

```
  environment: production

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: $CI_COMMIT_BRANCH != "main"
```

In this case, you need to exclude the `/vendor` directory from the list of folders Jekyll builds. Otherwise, Jekyll tries to build the directory contents along with the site.

In the root directory, create a file called `_config.yml` and add this content:

```
exclude:
  - vendor
```

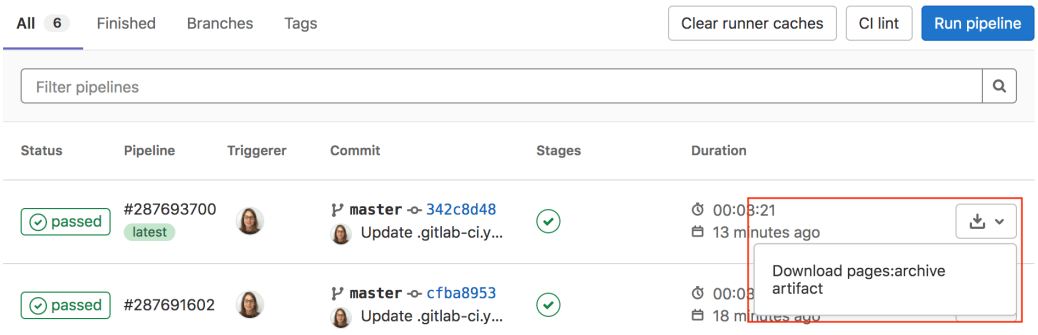
Now GitLab CI/CD not only builds the website, but also:

- Pushes with **continuous tests** to feature branches.
- **Caches** dependencies installed with Bundler.
- **Continuously deploys** every push to the `main` branch.

Download job artifacts

You can download job artifacts or view the job archive:

1. On the **Pipelines** page, to the right of the pipeline:



2. On the **Jobs** page, to the right of the job:

3. On a job's detail page. The **Keep** button indicates an expire_in value was set:

4. On a merge request, by the pipeline details:

5. When browsing an archive:

If GitLab Pages is enabled in the project, you can preview HTML files in the artifacts directly in your browser. If the project is internal or private, you must enable GitLab Pages access control to preview HTML files.