GitLab Configuration as Code

Release 0.1

Hoffmann-La Roche

Aug 01, 2022

CONTENTS

When configuring your GitLab instance, part of the settings you put in Omnibus or Helm Chart configuration, and the rest you configure through GitLab UI or API. Due to tons of configuration options in UI, making GitLab work as you intend is a complex process.

We intend to let you automate things you do through now UI in a simple way. The Configuration as Code has been designed to configure GitLab based on human-readable declarative configuration files written in Yaml. Writing such a file should be feasible without being a GitLab expert, just translating into code a configuration process one is used to executing in the web UI.

CHAPTER

ONE

CONTENTS:

1.1 Installation

1.2 Usage

1.2.1 Docker image

Image is available in Docker Hub.

GCasC Docker image working directory is `/workspace`. Thus you can quickly launch `gcasc` with:

```
docker run -v $(pwd):/workspace hoffmannlaroche/gcasc
```

It will try to find both GitLab client configuration and GitLab configuration in `/workspace` directory. You can modify the behavior by passing environment variables:

- `GITLAB_CLIENT_CONFIG_FILE` to provide path to GitLab client configuration file
- `GITLAB_CONFIG_FILE` to provide a path to GitLab configuration file

```
docker run
    -e GITLAB_CLIENT_CONFIG_FILE=/gitlab/client.cfg
    -e GITLAB_CONFIG_FILE=/gitlab/config.yml
    -v $(pwd):/gitlab
    hoffmannlaroche/gcasc
```

You can also configure a GitLab client using environment variables. More details about the configuration of GitLab client is *here*.

1.2.2 CLI

1.3 Client Configuration

GCasC uses a very particular configuration source order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. configuration file
2. environment variables (due to limitations in `python-gitlab` if using configuration file only `GITLAB_CLIENT_TOKEN` environment variable will be used)

Important! GitLab does not allow authentication using API with username and password. The preferred approach is to use personal access tokens. For more about it see *getting personal access token*.

1.3.1 Configuration file

Configuration file can have any name, but must contain have following structure (do not omit `[global]` line):

```
[global]
url = https://gitlab.yourdomain.com
ssl_verify = true  # optional
timeout = 5  # optional
private_token = <personal_access_token>
api_version = 4  # optional, assumes latest
```

By default *GCasC* is trying to find client configuration file in following paths:

```
/etc/python-gitlab.cfg
/etc/gitlab.cfg
~/.python-gitlab.cfg
~/.gitlab.cfg
```

You can provide another path to your configuration file in `GITLAB_CLIENT_CONFIG_FILE` environment variable.

1.3.2 Environment variables

You can use set up environment variables to configure your API client:

1.3.3 Getting personal access token

You **must** have personal access token if you want to use *GCasC*. Personal access token is mandatory in any client configuration approach. Unfortunately there is no way to configure it via API or get it automatically on instance setup. Thus you must first have GitLab running (for fresh deploys), then go to the UI and follow these instructions to get personal access token.

Recommendation is to limit scopes to minimal set required by the token. Additionally limit the time how long token is valid. It may not be the most convenient approach for CI/CD pipelines, but gives you additional significant security.

1.3.4 Setting client certificate`

GCasC allows setting up client certificate in case your GitLab instance requires mutual TLS authentication. You can configure it same way when using either configuration file or environment variables for client.

Just provide both of these environment variables. If one of them is missing, error will be raised.

1.4 Configuration

1.4.1 Appearance

GCasC allows configuring instance Appearance. Appearance can be configured either through UI (under Apperance in Admin Area) or API. Using this you can apply branding to your GitLab instance and provide basic information to your users.

Reference: https://docs.gitlab.com/12.7/ee/api/appearance.html

Appearance structure is flexible. It starts with a root key `appearance`. Then you provide configuration options as defined in these docs. For example

```
appearance:
  title: "GitLab instance title"
  description: "Some description of GitLab instance"
  header:
    logo: "http://path-to-your-logo.com/logo.png"
    message: "This is message to show in header"
```

Note: Any invalid keys will be discarded, warn message will be presented, but *GCasC* will continue execution.

1.4.2 Application Settings

GCasC allows configuring Application Settings. It consists of plenty of configuration options, that can be set only through UI or API. They are key to make your GitLab instance work as you intend to.

Reference: https://docs.gitlab.com/12.4/ee/api/settings.html

Settings the structure is flexible. It starts with a root key `settings`. Then you provide configuration options as defined in these docs. For example

```
settings:
  elasticsearch:
    url: http://elasticsearch.mygitlab.com
    username: elastic_user
    password: elastic_password
```

and

```
settings:
  elasticsearch_url: http://elasticsearch.mygitlab.com
  elasticsearch_username: elastic_user
  elasticsearch_password: elastic_password
```

are exactly the same and match `elasticsearch_url`, `elasticsearch_username` and `elasticsearch_password` settings. This means you can flexibly structure your configuration Yaml, where a map child keys are prefixed by parent key (here `elasticsearch` parent key was a prefix for `url`, `username` and `password` keys). Simply:

```
settings:
 prefix1:
   prefix2:
     value21: 'value21'
   value1: 'value1'
 prefix1_value2: 'value2'
```

will try to configure following properties: `prefix_value1`, `prefix_value2` and `prefix1_prefix2_value21`. You only need to follow available Application Settings.

Note: Any invalid keys will be discarded, warn message will be presented, but *GCasC* will continue execution.

You can adjust your Yamls by splitting them into multiple or injecting environment variables into certain values using `!include` or `!env` directives respectively. Example is shown below:

```
settings:
  elasticsearch: !include config/elasticseach.yml
  terms: !include tos.md
```

where:

- `settings.elasticsearch` is injected from file under `./config/elasticsearch.yml` path. Its configuration may look like this:

  ```
  url: http://elasticsearch.mygitlab.com
  username: !env ELASTICSEARCH_USERNAME
  password: !env ELASTICSEARCH_PASSWORD
  ```

  Note that here also `ELASTICSEARCH_USERNAME`, `ELASTICSEARCH_PASSWORD` are used to inject username and password from environment variables

- `settings.terms` is injected from `./tos.md` file

1.4.3 Instance Feature Flag

GitLab comes with some functionality configurable using feature flags. Part of the GitLab functionality is turned off, where to enable it you need to use API, cause it does not offer UI for setting up feature flags.

Reference: https://docs.gitlab.com/ee/api/features.html

Important! This is authoritative configuration, thus any existing Feature Flags will be removed and replaced with the ones defined in config file. If none are defined in config file, existing Feature Flags will remain untouched.

Features offered by GitLab are not collected in a single documentation page, but they are scattered. Please reference to GitLab documentation for them. Features yaml structure starts with a root key `features` . It's structure is defined below:

```
features: [list]
  - name: [string]
    value: [bool/int]
    feature_group: [string|optional]
    groups: [list(string)|optional]
    projects: [list(string)|optional]
    users: [list(string)|optional]
```

To configure certain feature for a limited set of:

- users, by specifying `users` by their username.
- groups, by specifying `groups` by group short name.
- projects, by specifying `groups` with format `group_name/project_name`.

Example of complex features configuration:

```yaml
features:
  - name: some_percentage_feature
    value: 25
    users:
      - user1
      - user2
  - name: some_percentage_feature
    value: 50
    users:
      - myuser
    groups:
      - mygroup
    projects:
      - mygroup1/myproject
      - mygroup1/myproject2
```

It will configure `some_percentage_feature` with value `25` for users `user1` and `user2`, while with value `50` for user `myuser`, group `mygroup` and projects `mygroup1/myproject`, `mygroup1/myproject2`.

1.4.4 Instance CI/CD Variables

GCasC allows configuring CI/CD Instance Variables. Instance variables are useful for no longer needing to manually enter the same credentials repeatedly for all your projects. Instance-level variables are available to all projects and groups on the instance.

Reference: https://docs.gitlab.com/ee/api/instance_level_ci_variables.html

Properties

Instance variables configuration starts with a root key `instance_variables`. Then you can either define

1. simple *key-value* property, where *key* is a name of variable and *value* is its value.
2. complex property to provide additional variables configuration. Property *key* is a name of variables

Key must be one line, using only letters, numbers, or _ (underscore), with no spaces.

Note: You can reference variables in other variables, e.g. you can set `MY_VARIABLE: 'the other variable is $OTHER_VARIABLE`.

Example

```yaml
instance_variables:
  MY_VARIABLE: 'value of my instance variable'
  ANOTHER_VARIABLE:
    value: !env SOME_PASSWORD
    masked: true
    protected: false
  SOME_FILE_VARIABLE:
    value: |
      long file data
    variable_type: file
```

1.4.5 License

Only for Enterprise Edition or gitlab.com. FOSS/Community Edition instance will fail when trying to configure license

GCasC offers a way to manage your GitLab instance licenses. The clue is that despite license is just a single file, you need to configure other properties of license so *GCasC* do not upload new (but already used) license with every execution. That way it is able to recognize that exactly the same license is already in use and skips uploading new one. Otherwise you could end with very long license history.

Reference: https://docs.gitlab.com/12.4/ee/api/license.html

Properties

Important! Beware of storing your license in `data` field directly as text. This is insecure and may lead to leakage of your license. Use `!env` or `!include` directives to inject license to `license.data` field securely from external source. Also keep your license file itself safe and secure!

Examples

Full license configuration::

```
license:
  starts_at: 2019-11-17
  expires_at: 2019-12-17
  plan: starter
  user_limit: 30
  data: |
    azhxWFZqbk1BOUsrTVxug6AdfzIzWXI1WUVsdWNKRk53V2hiV1FlTUN2TTRS
    NkhSVFFhZ3hCajd4bGlLMkhhcUxhd1EySHh2TjJTXG40U3ZNUWM0ZzhqYTE5
    T1lcbkJnNERFOVBORkpxK3FsaHZxNFFVSG9GL0NEWWF0elkyOE9SUE41Ny9v
```

Injecting license data from external file::

```
license:
  starts_at: 2019-11-17
  expires_at: 2019-12-17
  plan: ultimate
  user_limit: 30
  data: !include /etc/gitlab/my_gitlab_license.lic
```

Injecting license data from environment variable::

```
license:
  starts_at: 2019-11-17
  expires_at: 2019-12-17
  plan: ultimate
  user_limit: 30
  data: !env GITLAB_LICENSE
```

GitLab configuration is defined in a YAML. Providing configuraton for your GitLab instance is as simple as this:

```yaml
appearance:
  title: "Your GitLab instance title"
  logo: "http://path-to-your-logo/logo.png"

settings:
  elasticsearch:
    url: http://elasticsearch.mygitlab.com
    username: !env ELASTICSEARCH_USERNAME
    password: !env ELASTICSEARCH_PASSWORD
  recaptcha_enabled: yes
  terms: !include toc.md
  plantuml:
    enabled: true
    url: 'http://plantuml.url'

features:
  - name: sourcegraph
    value: true
    groups:
      - mygroup1
    projects:
      - mygroup2/myproject
    users:
      - myuser

instance_variables:
  MY_VARIABLE: 'value of my instance variable'
  ANOTHER_VARIABLE:
    value: !env SOME_PASSWORD
    masked: true
    protected: false

license:
  starts_at: 2019-11-17
  expires_at: 2019-12-17
  plan: premium
  user_limit: 30
  data: !include gitlab.lic
```

You can customize where *GCasC* searches for configuration file or if any changes should be applied on instance using environment variables.

Yaml directives

Custom Yaml directives give you enhanced way of defining your GitLab configuration YAML, where you can split your configuration into multiple Yaml files or inject environment variables.

- `!include` to provide path to another Yaml or plain text file which will be included file under certain key, e.g.

  ```yaml
  settings:
    terms: !include toc.md
    elasticsearch: !include config/elasticsearch.yml
  ```

  It searches for relative paths in current working directory tree AND in directory tree where GitLab configuration file is present.

- `!env` to inject values of environment variables under certain key, e.g.

```
settings:
  elasticsearch_username: !env ELASTICSEARCH_USERNAME
  elasticsearch_password: !env ELASTICSEARCH_PASSWORD
```

Note: Use `!env` directive to inject secrets into your Yaml. Never put secrets directly in Yaml file!

1.5 FAQ

I'm getting "gcasc.ClientInitializationError: GitLab token was not provided. It must be defined in `GITLAB_CLIENT_TOKEN` environment variable"

> It is likely that you provided invalid GitLab client configuration. If you use configuration file, verify if it has all required configuration parameters and that `GITLAB_CLIENT_CONFIG_FILE` environment variable is set to a path where your config file is. If you use environment variables, verify that you provided all necessary variables. See the :ref:`client configuration <client_configuration>` for details.