# Geofront Documentation

***Release 0.2.1***

**Hong Minhee**

July 08, 2014

Contents


**1 Situations** **3**

**2 Idea** **5**

**3 Prerequisites** **7**

**4 Author and license** **9**

**5 Missing features** **11**

**6 User's guide** **13**
- 6.1 Installation . . . 13
- 6.2 How to contribute . . . 15
- 6.3 Geofront Changelog . . . 16

**7 References** **19**
- 7.1 HTTP API . . . 19
- 7.2 CLI . . . 24
- 7.3 Configuration . . . 25
- 7.4 `geofront` — Simple SSH key management service . . . 30

**8 Indices and tables** **49**

**HTTP Routing Table** **51**

**Python Module Index** **53**

Geofront is a simple SSH key management server. It helps to maintain servers to SSH, and `authorized_keys` list for them. Read the docs for more details.

CHAPTER 1

# Situations

- If the team maintains `authorized_keys` list of all servers owned by the team:
  - When someone joins or leaves the team, all lists have to be updated.
  - *Who* do update the list?
- If the team maintains shared private keys to SSH servers:
  - These keys have to be expired when someone leaves the team.
  - There should be a shared storage for the keys. (Dropbox? srsly?)
  - Everyone might need to add `-i` option to use team's own key.
- The above ways are both hard to scale servers. Imagine your team has more than 10 servers.

CHAPTER 2

# Idea

1. Geofront has its own *master key*. The private key is never shared. The master key is periodically and automatically regened.
2. Every server has a simple `authorized_keys` list, which authorizes only the master key.
3. Every member registers their own public key to Geofront. The registration can be omitted if the key storage is GitHub, Bitbucket, etc.
4. A member requests to SSH a server, then Geofront *temporarily* (about 30 seconds, or a minute) adds their public key to `authorized_keys` of the requested server.

CHAPTER 3

# Prerequisites

- Linux, BSD, Mac
- Python 3.3+
- Third-party packages (automatically installed together)
  - Paramiko 1.13.0+
  - Werkzeug 0.9+
  - Flask 0.10+
  - Apache Libcloud 0.14.0+
  - Waitress 0.8.8+
  - singledispatch (only if Python is older than 3.4)

CHAPTER 4

# Author and license

Geofront is written by Hong Minhee, maintained by Spoqa, and licensed under AGPL3 or later. You can find the source code from GitHub:

```
$ git clone git://github.com/spoqa/geofront.git
```

CHAPTER 5

# Missing features

- Google Apps backend [#3]
- Bitbucket backend [#4]
- Fabric integration
- PuTTY integration

(Contributions would be appreciated!)

CHAPTER 6

# User's guide

## 6.1 Installation

You can easily install Geofront server using **pip**:

```
$ pip install Geofront
```

### 6.1.1 Running server

Geofront server requires a configuration file. Configuration file is a typical Python script. The server is sensitive to the values of some uppercase variables like `TEAM`, `KEY_STORE`, and `MASTER_KEY_BITS`. The filename of the configuration is not important, but recommend to use `.cfg.py` suffix. You also can find an example configuration in the Geofront repository: `example.cfg.py`.

**See also:**

*Configuration* The reference manual for Geofront server configuration.

If a configuration file is ready you can run the server right now. Suppose the configuration file is `geofront.cfg.py`.

**geofront-server** command provides several options like `--host`, and requires a configuration filename as its argument.

```
$ geofront-server -p 8080 geofront.cfg.py
```

It might be terminated with the following error message:

```
$ geofront-server -p 8080 geofront.cfg.py
usage: geofront-server [...] FILE
geofront-server: error: no master key;
try --create-master-key option if you want to create one
```

It means `MASTER_KEY_STORE` you configured has no master key yet. `--create-master-key` option creates a new master key if there's no master key yet, and then stores it into the configured `MASTER_KEY_STORE`.

```
$ geofront-server -p 8080 --create-master-key geofront.cfg.py
no master key;  create one...
created new master key: 2b:d5:64:fd:27:f9:7a:6a:12:7d:88:76:a7:54:bd:6a
serving on http://0.0.0.0:8080
```

If it successfully starts serving it will show you the bound host and port.

### 6.1.2 Reverse proxy

Application servers typically run behind the reverse proxy like Nginx. Here's an example configuration for Geofront server behind Nginx reverse proxy:

```
# Redirect all HTTP requests to HTTPS.
# We highly recommend to expose Geofront server only through HTTPS.
server {
  listen 80;
  server_name geofront-example.org;
  rewrite ^(.*)$ https://geofront-example.org$1;
}

# Forward all requests to https://geofront-example.org to internal
# http://127.0.0.1:8080.
server {
  listen 443 ssl;
  server_name geofront-example.org;
  access_log  /var/log/nginx/geofront/access.log;
  error_log /var/log/nginx/geofront/error.log;

  ssl on;
  ssl_certificate /path/to/ssl_cert_chain.pem;
  ssl_certificate_key /path/to/ssl_cert.pem;

  # HSTS: https://developer.mozilla.org/en-US/docs/Web/Security/HTTP_strict_transport_security
  add_header Strict-Transport-Security "max-age=31536000";

  location / {
      proxy_pass http://127.0.0.1:8080;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

### 6.1.3 Using geofront-cli

Every team member who want to use Geofront has to install a client for Geofront server. geofront-cli is the reference implementation of Geofront client. It can be installed using **pip**:

```
$ pip install --allow-external dirspec \
              --allow-unverified dirspec \
              geofront-cli
```

To setup what Geofront server to use use **geofront-cli start** command. It will show a prompt:

```
$ geofront-cli start
Geofront server URL:
```

Type the server URL, and then it will open an authentication page in your default web browser:

```
$ geofront-cli start
Geofront server URL: https://geofront-example.org/
Continue to authenticate in your web browser...
Press return to continue
```

That's done. Setup process is only required at first. You can show the list of available remotes using **geofront-cli remotes**:

```
$ geofront-cli remotes
web-1
web-2
...
```

For more details on **geofront-cli**, read the manual of its `README.rst`, or use *`geofront-cli --help`* option.

### 6.1.4 Remote colonization

Until a remote server authorizes the master key you can't access to the remote using **geofront-cli**. So the master key needs to be added to remote's `authorized_keys` list. Geofront calls it *colonization*. You can colonize a remote using **geofront-cli colonize** command. Surely the following command has to be run by who can access to it:

```
$ geofront-cli remotes
web-1
web-2
...
$ geofront-cli colonize web-1
```

You can understand **geofront-cli colonize** is **ssh-copy-id** for Geofront. Once colonized remote is accessible by every team member unless you configured more fine-grained ACL. (See also `PERMISSION_POLICY` if you're interested in ACL.)

### 6.1.5 SSH through Geofront

If a remote is once colonized any team member can **ssh** to it through Geofront. Use **geofront-cli ssh** command:

```
$ geofront-cli ssh web-1
Last login: Sat May  3 16:32:15 2014 from hong-minhees-macbook-pro.local
$
```

## 6.2 How to contribute

### 6.2.1 License agreement

All contributed codes have to be free software licensed under the terms of the GNU Affero General Public License Version 3 or any later version. We treat all pull requests imply agreement of it, but if a significant amount of code is involved, it is safest to mention in the pull request comments that you agree to let the patch be used under the GNU Affero General Public License Version 3 or any later version as part of the Geofront code.

### 6.2.2 Coding style

- Follow PEP 8 except you can limit all lines to a maximum of 80 characters (not 79).
- Order `imports` in lexicographical order.
- Prefer relative `imports`.
- All functions, classes, methods, attributes, and modules should have the docstring.

### 6.2.3 Tests

- All code patches should contain one or more unit tests of the feature to add or regression tests of the bug to fix.
- You can run the test suite using `runtests.sh` script. It installs librearies for testing as well if not installed.
- Or you can simply run `py.test` command if you have all dependencies for testing.
- Some tests would be skipped unless you give additional options. You can see the list of available options in *custom options* section of `py.test --help`.
- All commits will be tested by Travis CI.

## 6.3 Geofront Changelog

### 6.3.1 Version 0.2.1

Released on June 16, 2014.

- Fixed an authentication bug of `TwoPhaseRenewal` raised due to not specify login username.
- More detailed message logging of exceptions that rise during master key renewal.

### 6.3.2 Version 0.2.0

Released on May 3, 2014.

- Added `list_groups()` method to `Team` interface.
- Added `list_groups()` method to `GitHubOrganization` class.
- Removed an unnecessary dependency to enum34 on Python 3.3.
- Added `geofront.backends.cloud` module.
  - `geofront.masterkey.CloudMasterKeyStore` is moved to `geofront.backends.cloud.CloudMasterKeyStore`.
  - `geofront.remote.CloudRemoteSet` is moved to `geofront.backends.cloud.CloudRemoteSet`.
- `Remote` now has `metadata` attribute.
- `CloudRemoteSet` fills `metadata` of the resulted `Remote` objects if the given driver supports.
- Now depends on singledispatch if Python is older than 3.4.
- Added `PermissionPolicy` interface.
- Added `DefaultPermissionPolicy` class.
- Added `GroupMetadataPermissionPolicity` class.
- Added new `PERMISSION_POLICY` configuration.
- Added `geofront.backends.dbapi` module.
- Added **geofront-key-regen** command.
- HTTP APIs became more RESTful. Now it has the root endpoint which provides the link to create a new token, and the token API provides several links to subresources as well.
- Added new `MASTER_KEY_BITS` configuration.

- Added new `bits` optional parameters to `renew_master_key()`, `PeriodicalRenewal`, and `regenerate()`.
- Added `CloudKeyStore`. [#2]
- Added `CloudMasterPublicKeyStore`. [#2]

### 6.3.3 Version 0.1.1

Released on April 22, 2014.

- Fixed `TypeError` that rises when `CloudMasterKeyStore` is used with AWS S3 driver.
- Added *`--trusted-proxy`* option to **geofront-server** command. It's useful when the server is run behind a reverse proxy.
- Added token no-op API: `GET /tokens/(token_id:token_id)/`.

### 6.3.4 Version 0.1.0

First alpha release. Released on April 21, 2014.

CHAPTER 7

# References

## 7.1 HTTP API

### 7.1.1 Server version

The release policy of Geofront follows Semantic Versioning, and the HTTP API which this docs covers also does the same. You can treat what you could do on Geofront 1.2.3:

- might be broken on Geofront 2.0.0;
- shouldn't be broken 1.3.0;
- must not be broken on Geofront 1.2.4.

Also broken things on Geofront 1.2.3 might be fixed on Geofront 1.2.4.

So how does the server tell its version through HTTP API? It provides two headers that are equivalent:

***Server*** Which is a standard compliant header. The form follows also the standard e.g. `Geofront/1.2.3`.

***X-Geofront-Version*** Which is a non-standard extended header. The form consists of only the version number e.g. `1.2.3`.

These headers even are provided when the response is error:

```
HTTP/1.0 404 Not Found
Content-Length: 9
Content-Type: text/plain
Date: Tue, 01 Apr 2014 17:46:36 GMT
Server: Geofront/0.9.0
X-Geofront-Version: 0.9.0

Not Found
```

### 7.1.2 Endpoints

**GET /**
The endpoint of HTTP API which provide the url to create a new token.

```
GET / HTTPS/1.1
Accept: application/json
```

```
HTTP/1.0 200 OK
Content-Type: application/json
Link: <https://example.com/tokens/>; rel=tokens

{
  "tokens_url": "https://example.com/tokens/"
}
```

**Response Headers**

- **Link** – the url to create a new token. the equivalent to the response content

**Status Codes**

- **200** – when the server is available

New in version 0.2.0.

**POST /tokens/** (**token_id:** *token_id*) **/remotes/**
*alias***/** Temporarily authorize the token owner to access a remote. A made authorization keeps alive in a minute, and then will be expired.

```
POST /tokens/0123456789abcdef/remotes/web-1/ HTTPS/1.1
Accept: application/json
Content-Length: 0

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "success": "authorized",
  "remote": {"user": "ubuntu", "host": "192.168.0.5", "port": 22},
  "expires_at": "2014-04-14T14:57:49.822844+00:00"
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity
- **alias** (`str`) – the alias of the remote to access

**Status Codes**

- **200** – when successfully granted a temporary authorization
- **404** – (`not-found`) when there's no such remote

**GET /tokens/** (**token_id:** *token_id*) **/keys/**
**fingerprint:** *fingerprint***/** Find the public key by its `fingerprint` if it's registered.

```
GET /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47/ HTTPS/1.1
Accept: text/plain

HTTPS/1.1 200 OK
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity
- **fingerprint** (`bytes`) – the fingerprint of a public key to find

**Status Codes**

- **200** – when the public key is registered
- **404** – (`not-found`) when there's no such public key

**DELETE /tokens/**(**token_id:** *token_id*) **/keys/**
**fingerprint:** *fingerprint***/** Delete a public key.

```
DELETE /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47/ HTTPS/1.1
Accept: application/json

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "72:00:60:24:66:e8:2d:4d:2a:2a:a2:0e:7b:7f:fc:af":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCom2CDLekY...5CeYsvSdrTWA5 ",
  "78:8a:09:c8:c1:24:5c:89:76:92:b0:1e:93:95:5d:48":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA16iSKKjFHOgj...kD62SYXNKY9c= ",
  "ab:3a:fb:30:44:e3:5e:1e:10:a0:c9:9a:86:f4:67:59":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAzzF8c07pzgKk...r+b6Q9VnWWQ== "
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity
- **fingerprint** (`bytes`) – the fingerprint of a public key to delete

**Status Codes**

- **200** – when the public key is successfully deleted
- **404** – (`not-found`) when there's no such public key

**GET /tokens/**(**token_id:** *token_id*) **/authenticate/**
Finalize the authentication process. It will be shown on web browser.

**Parameters**

- **token_id** (`str`) – token id created by `create_access_token()`

**Status Codes**

- **400** – when authentication is failed
- **404** – when the given `token_id` doesn't exist
- **403** – when the `token_id` is already finalized
- **200** – when authentication is successfully done

**GET /tokens/**(**token_id:** *token_id*) **/masterkey/**
Public part of the master key in OpenSSH authorized_keys (public key) format.

```
GET /tokens/0123456789abcdef/masterkey/ HTTPS/1.1
Accept: text/plain
```

```
HTTPS/1.1 200 OK
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity

**Status Codes**

- **200** – when the master key is available
- **500** – when the master key is unavailable

**GET /tokens/** (**token_id:** *token_id*) **/remotes/**
List all available remotes and their aliases.

```
GET /tokens/0123456789abcdef/remotes/ HTTPS/1.1
Accept: application/json

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "web-1": {"user": "ubuntu", "host": "192.168.0.5", "port": 22},
  "web-2": {"user": "ubuntu", "host": "192.168.0.6", "port": 22},
  "web-3": {"user": "ubuntu", "host": "192.168.0.7", "port": 22},
  "worker-1": {"user": "ubuntu", "host": "192.168.0.25", "port": 22},
  "worker-2": {"user": "ubuntu", "host": "192.168.0.26", "port": 22},
  "db-1": {"user": "ubuntu", "host": "192.168.0.50", "port": 22},
  "db-2": {"user": "ubuntu", "host": "192.168.0.51", "port": 22}
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity

**Status Codes**

- **200** – when listing is successful, even if there are no remotes

**GET /tokens/** (**token_id:** *token_id*) **/keys/**
List registered keys to the token owner.

```
GET /tokens/0123456789abcdef/keys/ HTTPS/1.1
Accept: application/json

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q== ",
  "72:00:60:24:66:e8:2d:4d:2a:2a:a2:0e:7b:7f:fc:af":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCom2CDLekY...5CeYsvSdrTWA5 ",
  "78:8a:09:c8:c1:24:5c:89:76:92:b0:1e:93:95:5d:48":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA16iSKKjFHOgj...kD62SYXNKY9c= ",
  "ab:3a:fb:30:44:e3:5e:1e:10:a0:c9:9a:86:f4:67:59":
```

```
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAzzF8c07pzgKk...r+b6Q9VnWWQ== "
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity

**Status Codes**

- **200** – when listing is successful, even if there are no keys

**POST /tokens/** (**token_id:** *token_id*) **/keys/**

Register a public key to the token. It takes an OpenSSH public key line through the request content body.

```
POST /tokens/0123456789abcdef/keys/ HTTPS/1.1
Accept: application/json
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==

HTTPS/1.1 201 Created
Content-Type: text/plain
Location: /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity

**Status Codes**

- **201** – when key registration is successful
- **400** – (`unsupported-key-type`) when the key type is unsupported, or (`invalid-key`) the key format is invalid, or (`deuplicate-key`) the key is already used
- **415** – (`unsupported-content-type`) when the *`Content-Type`* is not *`text/plain`*

**PUT /tokens/** (**token_id:** *token_id*) **/**

Create a new access token.

```
PUT /tokens/0123456789abcdef/ HTTPS/1.1
Accept: application/json
Content-Length: 0

HTTPS/1.1 202 Accepted
Content-Type: application/json
Date: Tue, 15 Apr 2014 03:44:43 GMT
Expires: Tue, 15 Apr 2014 04:14:43 GMT
Link: <https://example.com/login/page/?redirect_uri=...>; rel=next

{
  "next_url": "https://example.com/login/page/?redirect_uri=..."
}
```

**Parameters**

- **token_id** (`str`) – an arbitrary token id to create. it should be enough random to avoid duplication

**Status Codes**

- **202** – when the access token is prepared

**Response Headers**

- **Link** – the link owner's browser should redirect to

**GET /tokens/(token_id: *token_id*)/**

The owner identity that the given token holds if the token is authenticated. Otherwise it responds 403 Forbidden, 404 Not Found, 410 Gone, or 412 Precondition Failed. See also `get_identity()`.

```
GET /tokens/0123456789abcdef/ HTTPS/1.1
Accept: application/json

HTTPS/1.0 200 OK
Content-Type: application/json
Link: <https://example.com/tokens/0123456789abcdef/remo...>; rel=remotes
Link: <https://example.com/tokens/0123456789abcdef/keys/>; rel=keys
Link: <https://example.com/tokens/0123456789abcdef/ma...>; rel=masterkey

{
  "identifier": "dahlia",
  "team_type": "geofront.backends.github.GitHubOrganization",
  "remotes_url": "https://example.com/tokens/0123456789abcdef/remotes/",
  "keys_url": "https://example.com/tokens/0123456789abcdef/keys/",
  "master_key_url": "https://example.com/tokens/0123456789abcdef/mas..."
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity

**Response Headers**

- **Link** – the url to list remotes (`rel=remotes`), public keys (`rel=keys`), and master key (`rel=masterkey`)

**Status Codes**

- **200** – when the token is authenticated

Changed in version 0.2.0: The response contains `"remotes_url"`, `"keys_url"`, and `"master_key_url"`, and equivalent three *`Link`* headers.

## 7.2 CLI

**See also:**

*Configuration*

### 7.2.1 geofront-server

Simple SSH key management service

**file**
geofront configuration file (Python script)

**-h, --help**
show this help message and exit

**--create-master-key**
create a new master key if no master key yet

**-d, --debug**
debug mode

**-v, --version**
show program's version number and exit

**-H** <host>**, --host** <host>
host to bind [0.0.0.0]

**-p** <port>**, --port** <port>
port to bind [5000]

**--renew-master-key**
renew the master key before the server starts. implies –create-master-key option

**--trusted-proxy**
IP address of a client allowed to override url_scheme via the X-Forwarded-Proto header. useful when it runs behind reverse proxy. -d/–debug option disables this option

### 7.2.2 geofront-key-regen

Regen the Geofront master key

**file**
geofront configuration file (Python script)

**-h, --help**
show this help message and exit

**--create-master-key**
create a new master key if no master key yet

**-d, --debug**
debug mode

**-v, --version**
show program's version number and exit

## 7.3 Configuration

The **geofront-server** command takes a configuration file as required argument. The configuration is an ordinary Python script that defines the following required and optional variables. Note that all names have to be uppercase.

config.**TEAM**
(geofront.team.Team) The backend implementation for team authentication. For example, in order to authorize members of GitHub organization use GitHubOrganization implementation:

```
from geofront.backends.github import GitHubOrganization

TEAM = GitHubOrganization(
    client_id='GitHub OAuth app client id goes here',
    client_secret='GitHub OAuth app client secret goes here',
    org_login='your_org_name'  #  in https://github.com/your_org_name
)
```

Or you can implement your own backend by subclassing `Team`.

**See also:**

**Module `geofront.team` — Team authentication** The interface for team authentication.

**Class `geofront.backends.github.GitHubOrganization`** The `Team` implementation for GitHub organizations.

`config.`**`REMOTE_SET`**

(`collections.abc.Mapping`) The set of remote servers to be managed by Geofront. It can be anything only if it's an mapping object. For example, you can hard-code it by using Python `dict` data structure:

```
from geofront.remote import Remote

REMOTE_SET = {
    'web-1': Remote('ubuntu', '192.168.0.5'),
    'web-2': Remote('ubuntu', '192.168.0.6'),
    'web-3': Remote('ubuntu', '192.168.0.7'),
    'worker-1': Remote('ubuntu', '192.168.0.25'),
    'worker-2': Remote('ubuntu', '192.168.0.26'),
    'db-1': Remote('ubuntu', '192.168.0.50'),
    'db-2': Remote('ubuntu', '192.168.0.51')
}
```

Every key has to be a string, and every valye has to be an instance of `Remote`. `Remote` consits of an user, a hostname, and the port to SSH. For example,if you've **ssh**-ed to a remote server by the following command:

```
$ ssh -p 2222 ubuntu@192.168.0.50
```

A `Remote` object for it should be:

```
Remote('ubuntu', '192.168.0.50', 2222)
```

You can add more dynamism by providing custom `dict`-like mapping object. `collections.abc.Mapping` could help to implement it. For example, `CloudRemoteSet` is a subtype of `Mapping`, and it dynamically loads the list of available instance nodes in the cloud e.g. EC2 of AWS. Due to Apache Libcloud it can work with more than 20 cloud providers like AWS, Azure, or Rackspace.

```
from geofront.backends.cloud import CloudRemoteSet
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

driver_cls = get_driver(Provider.EC2_US_WEST)
driver = driver_cls('access id', 'secret key')
REMOTE_SET = CloudRemoteSet(driver)
```

**See also:**

**Class `geofront.remote.Remote`** Value type that represents a remote server to **ssh**.

**Class `geofront.backends.cloud.CloudRemoteSet`** The Libcloud-backed dynamic remote set.

**Module collections.abc — Abstract Base Classes for Containers** This module provides abstract base classes that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

config.**TOKEN_STORE**
(werkzeug.contrib.cache.BaseCache) The store to save access tokens. It uses Werkzeug's cache interface, and Werkzeug provides several built-in implementations as well e.g.:

- MemcachedCache
- RedisCache
- FileSystemCache

For example, in order to store access tokens into Redis:

```
from werkzeug.contrib.cache import RedisCache

TOKEN_STORE = RedisCache(host='localhost', db=0)
```

Of course you can implement your own backend by subclassing BaseCache.

Although it's a required configuration, but when *-d*/*--debug* is enabled, SimpleCache (which is all expired after **geofront-server** process terminated) is used by default.

**See also:**

**Cache — Werkzeug** Cache backend interface and implementations provided by Werkzeug.

config.**KEY_STORE**
(geofront.keystore.KeyStore) The store to save *public keys* for each team member. (Not the *master key*; don't be confused with MASTER_KEY_STORE.)

If TEAM is a GitHubOrganization object, KEY_STORE also can be GitHubKeyStore. It's an adapter class of GitHub's per-account public key list.

```
from geofront.backends.github import GitHubKeyStore

KEY_STORE = GitHubKeyStore()
```

You also can store public keys into the database like SQLite, PostgreSQL, or MySQL through DatabaseKeyStore:

```
   import sqlite3
   from geofront.backends.dbapi import DatabaseKeyStore

   KEY_STORE = DatabaseKeyStore(sqlite3,
                                '/var/lib/geofront/public_keys.db')
```

```
Some cloud providers like Amazon EC2 and Rackspace (Next Gen) support
*key pair service*.  :class:`~geofront.backends.cloud.CloudKeyStore`
helps to use the service as a public key store::

    from geofront.backends.cloud import CloudKeyStore
    from libcloud.storage.types import Provider
    from libcloud.storage.providers import get_driver

    driver_cls = get_driver(Provider.EC2)
    driver = driver_cls('api key', 'api secret key')
    KEY_STORE = CloudKeyStore(driver)
```

New in version 0.2.0: Added `DatabaseKeyStore` class. Added `CloudKeyStore` class.

`config.`**`MASTER_KEY_STORE`**
(`geofront.masterkey.MasterKeyStore`) The store to save the *master key*. (Not *public keys*; don't be confused with `KEY_STORE`.)

The master key store should be secure, and hard to lose the key at the same time. Geofront provides some built-in implementations:

**`FileSystemMasterKeyStore`** It stores the master key into the file system as the name suggests. You can set the path to save the key. Although it's not that secure, but it might help you to try out Geofront.

**`CloudMasterKeyStore`** It stores the master key into the cloud object storage like S3 of AWS. It supports more than 20 cloud providers through the efforts of Libcloud.

```
from geofront.masterkey import FileSystemMasterKeyStore

MASTER_KEY_STORE = FileSystemMasterKeyStore('/var/lib/geofront/id_rsa')
```

`config.`**`PERMISSION_POLICY`**
(`PermissionPolicy`) The permission policy to determine which remotes are visible for each team member, and allowed them to SSH.

The default is `DefaultPermissionPolicy`, and it allows everyone in the team to view and access through SSH to all available remotes.

If your remote set has metadata for ACL i.e. group identifiers to allow you can utilize it through `GroupMetadataPermissionPolicy`.

If you need more subtle and complex rules for ACL you surely can implement your own policy by subclassing `PermissionPolicy` interface.

New in version 0.2.0.

`config.`**`MASTER_KEY_BITS`**
The number of bits the generated master key should be. 2048 by default.

New in version 0.2.0.

`config.`**`MASTER_KEY_RENEWAL`**
(`datetime.timedelta`) The interval of master key renewal. `None` means never. For example, if you want to renew the master key every week:

```
import datetime

MASTER_KEY_RENEWAL = datetime.timedelta(days=7)
```

A day by default.

`config.`**`TOKEN_EXPIRE`**
(`datetime.timedelta`) The time to expire each access token. As shorter it becomes more secure but more frequent to require team members to authenticate. So too short time would interrupt team members.

A week by default.

### 7.3.1 Example

```
# This is a configuration example.  See docs/config.rst as well.

# Scenario: Your team is using GitHub, and the organization login is @YOUR_TEAM.
# All members already registered their public keys to their GitHub accounts,
```

```
# and are using git through ssh public key authorization.

# First of all, you have to decide how to authorize team members.
# Geofront provides a built-in authorization method for GitHub organizations.
# It requires a pair of client keys (id and secret) for OAuth authentication.
# You can create one from:
#
# https://github.com/organizations/YOUR_TEAM/settings/applications/new
#
# Then import GitHubOrganization class, and configure a pair of client keys
# and your organization login name (@YOUR_TEAM in here).
from geofront.backends.github import GitHubOrganization

TEAM = GitHubOrganization(
   client_id='0123456789abcdef0123',
   client_secret='0123456789abcdef0123456789abcdef01234567',
   org_login='YOUR_TEAM'
)

# Your colleagues have already registered their public keys to GitHub,
# so you don't need additional storage for public keys.  We'd use GitHub
# as your public key store.
from geofront.backends.github import GitHubKeyStore

KEY_STORE = GitHubKeyStore()

# Unlike public keys, the master key ideally ought to be accessible by
# only Geofront.  Assume you use Amazon Web Services.  So you'll store
# the master key to the your private S3 bucket named your_team_master_key.
from geofront.backends.cloud import CloudMasterKeyStore
from libcloud.storage.types import Provider
from libcloud.storage.providers import get_driver

driver_cls = get_driver(Provider.S3)
driver = driver_cls('aws access key', 'aws secret key')
container = driver.get_container(container_name='your_team_master_key')
MASTER_KEY_STORE = CloudMasterKeyStore(driver, container, 'id_rsa')

# You have to let Geofront know what to manage remote servers.
# Although the list can be hard-coded in the configuration file,
# but you'll get the list dynamically from EC2 API.  Assume our all
# AMIs are Amazon Linux, so the usernames are always ec2-user.
# If you're using Ubuntu AMIs it should be ubuntu instead.
from geofront.backends.cloud import CloudRemoteSet
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

driver_cls = get_driver(Provider.EC2_US_WEST)
driver = driver_cls('aws access id', 'aws secret key')
REMOTE_SET = CloudRemoteSet(driver, user='ec2-user')

# Suppose your team is divided by several subgroups, and these subgroups are
# represented in teams of the GitHub organization.  So you can control
# who can access each remote by specifying allowed groups to its metadata.
# CloudRemoteSet which is used for above REMOTE_SET exposes each EC2 instance's
# metadata as it has.  We suppose every EC2 instance has Allowed-Groups
# metadata key and its value is space-separated list of group slugs.
# The following settings will allow only members who belong to corresponding
```

```
# groups to access.
from geofront.remote import GroupMetadataPermissionPolicy

PERMISSION_POLICY = GroupMetadataPermissionPolicy('Allowed-Groups')

# Geofront provisions access tokens (or you can think them as sessions)
# for Geofront clients.  Assume you already have a Redis server running
# on the same host.  We'd store tokens to the db 0 on that Redis server
# in the example.
from werkzeug.contrib.cache import RedisCache

TOKEN_STORE = RedisCache(host='localhost', db=0)
```

## 7.4 geofront — Simple SSH key management service

### 7.4.1 geofront.backends — Backend implementations

#### geofront.backends.cloud — Libcloud-backed implementations

This module provides built-in implementations of Geofront's some core interfaces through libcloud. Libcloud is "a library for interacting with many of the popular cloud service providers using unified API."

New in version 0.2.0.

**class** geofront.backends.cloud.**CloudKeyStore**(*driver: libcloud.compute.base.NodeDriver*, *key_name_format: str=None*)

Store public keys into the cloud provider's key pair service. Note that not all providers support key pair service. For example, Amazon EC2, and Rackspace (Next Gen) support it.

```
from geofront.backends.cloud import CloudKeyStore
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

driver_cls = get_driver(Provider.EC2)
driver = driver_cls('api key', 'api secret key')
KEY_STORE = CloudKeyStore(driver)
```

**Parameters**

- **driver** (libcloud.compute.base.NodeDriver) – libcloud compute driver
- **key_name_format** (str) – the format which determines each key's name used for the key pair service. default is DEFAULT_KEY_NAME_FORMAT

**DEFAULT_KEY_NAME_FORMAT = '{identity.team_type.__module__}.{identity.team_type.__qualname__} {identity.identifi**

(str) The default key_name_format. The type name of team followed by identifier, and then key fingerprint follows e.g. 'geofront.backends.github.GitHubOrganization dahlia 00:11:22:..:ff'.

**class** geofront.backends.cloud.**CloudMasterKeyStore**(*driver: libcloud.storage.base.StorageDriver*, *container: libcloud.storage.base.Container*, *object_name: str*)

Store the master key into the cloud object storage e.g. AWS S3. It supports more than 20 cloud providers through the efforts of Libcloud.

```
from geofront.backends.cloud import CloudMasterKeyStore
from libcloud.storage.types import Provider
from libcloud.storage.providers import get_driver

driver_cls = get_driver(Provider.S3)
driver = driver_cls('api key', 'api secret key')
container = driver.get_container(container_name='my-master-key-bucket')
MASTER_KEY_STORE = CloudMasterKeyStore(container)
```

**Parameters**

- **driver** (`libcloud.storage.base.StorageDriver`) – the libcloud storage driver
- **container** (`libcloud.storage.base.Container`) – the block storage container
- **object_name** (`str`) – the object name to use

**See also:**

**Object Storage — Libcloud** Storage API allows you to manage cloud object storage and services such as Amazon S3, Rackspace CloudFiles, Google Storage and others.

**class** `geofront.backends.cloud.`**`CloudMasterPublicKeyStore`**(*driver: libcloud.compute.base.NodeDriver*, *key_pair_name: str*, *master_key_store: geofront.masterkey.MasterKeyStore*)

It doesn't store the whole master key, but stores only public part of the master key into cloud provider's key pair registry. So it requires the actual `master_key_store` to store the whole master key which is not only public part but also private part.

It helps to create compute instances (e.g. Amazon EC2) that are already colonized.

**Parameters**

- **driver** (`libcloud.compute.base.NodeDriver`) – libcloud compute driver
- **key_pair_name** (`str`) – the name for cloud provider's key pair registry
- **master_key_store** (`MasterKeyStore`) – "actual" master key store to store the whole master key

New in version 0.2.0.

**class** `geofront.backends.cloud.`**`CloudRemoteSet`**(*driver: libcloud.compute.base.NodeDriver*, *user: str='ec2-user'*, *port: numbers.Integral=22*)

Libcloud-backed remote set. It supports more than 20 cloud providers through the efforts of Libcloud.

```
from geofront.backends.cloud import CloudRemoteSet
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

driver_cls = get_driver(Provider.EC2_US_WEST)
driver = driver_cls('access id', 'secret key')
REMOTE_SET = CloudRemoteSet(driver)
```

If the given `driver` supports metadata feature (for example, AWS EC2, Google Compute Engine, and OpenStack support it) the resulted `Remote` objects will fill their `metadata` as well.

**Parameters**

- **driver** (`libcloud.compute.base.NodeDriver`) – libcloud compute driver
- **user** (`str`) – the username to **ssh**. the default is `'ec2-user'` which is the default user of amazon linux ami
- **port** (`numbers.Integral`) – the port number to **ssh**. the default is 22 which is the default **ssh** port

**See also:**

**Compute — Libcloud** The compute component of libcloud allows you to manage cloud and virtual servers offered by different providers, more than 20 in total.

Changed in version 0.2.0: It fills `metadata` of the resulted `Remote` objects if the `driver` supports.

### `geofront.backends.dbapi` — Key store using DB-API 2.0

**See also:**

**PEP 249** — Python Database API Specification v2.0

New in version 0.2.0.

**class** `geofront.backends.dbapi.`**`DatabaseKeyStore`**(*db_module: module*, **args*, ***kwargs*)

Store public keys into database through DB-API 2.0. It takes a module that implements DB-API 2.0, and arguments/keywords to its `connect()` method. For example, the following code stores public keys into SQLite 3 database:

```
import sqlite3
DatabaseKeyStore(sqlite3, 'geofront.db')
```

The following code stores public keys into PostgreSQL database through psycopg2:

```
import psycopg2
DatabaseKeyStore(psycopg2, database='geofront', user='postgres')
```

It will create a table named `geofront_public_key` into the database.

**Parameters**

- **db_module** (`types.ModuleType`) – **PEP 249** DB-API 2.0 compliant module
- ***args** – arguments to `db_module.connect()` function
- ***kwargs** – keyword arguments to `db_module.connect()` function

### `geofront.backends.github` — GitHub organization and key store

**class** `geofront.backends.github.`**`GitHubKeyStore`**

Use GitHub account's public keys as key store.

**class** `geofront.backends.github.`**`GitHubOrganization`**(*client_id: str*, *client_secret: str*, *org_login: str*)

Authenticate team membership through GitHub, and authorize to access GitHub key store.

Note that group identifiers `list_groups()` method returns are GitHub team *slugs*. You can find what team slugs are there in the organization using GitHub API:

```
$ curl -u YourUserLogin https://api.github.com/orgs/YourOrgLogin/teams
Enter host password for user 'YourUserLogin':
[
  {
    "name": "Owners",
    "id": 111111,
    "slug": "owners",
    "permission": "admin",
    "url": "https://api.github.com/teams/111111",
    ...
  },
  {
    "name": "Programmers",
    "id": 222222,
    "slug": "programmers",
    "permission": "pull",
    "url": "https://api.github.com/teams/222222",
    ...
  }
]
```

In the above example, `owners` and `programmers` are team slugs.

**Parameters**

- **client_id** (`str`) – github api client id
- **client_secret** (`str`) – github api client secret
- **org_login** (`str`) – github org account name. for example `'spoqa'` in https://github.com/spoqa

geofront.backends.github.**request**(*access_token*, *url: str*, *method: str='GET'*, *data: bytes=None*)

Make a request to GitHub API, and then return the parsed JSON result.

**Parameters**

- **access_token** (`str`, `Identity`) – api access token string, or `Identity` instance
- **url** (`str`) – the api url to request
- **method** (`str`) – an optional http method. `'GET'` by default
- **data** (`bytes`) – an optional content body

### 7.4.2 `geofront.identity` — Member identification

**class** geofront.identity.**Identity**(*team_type: type*, *identifier: collections.abc.Hashable*, *access_token=None*)

Hashable value object which purposes to identify the owner of each public key in the store.

**Parameters**

- **team_type** (`type`) – a sbclass of `Team`
- **identifier** (`collections.abc.Hashable`) – any hashable identifier for the owner. it's interpreted by `team_type`
- **access_token** – an optional access token which may used by key store

**access_token = None**
An optional access token which may be used by key store.

**Note:** The attribute is ignored by ==, and = operators, and `hash()` function.

**identifier = None**
(`collections.abc.Hashable`) Any hashable identifier for the owner. It's interpreted by `team_type`.

**team_type = None**
(`type`) A subclass of `Team`.

### 7.4.3 geofront.keystore — Public key store

geofront.keystore.**KEY_TYPES = {'ssh-rsa': <class 'paramiko.rsakey.RSAKey'>, 'ssh-dss': <class 'paramiko.dsskey.DSS**
(`collections.Mapping`) The mapping of supported key types.

**exception** geofront.keystore.**AuthorizationError**
Authorization exception that rise when the given identity has no required permission to the key store.

**exception** geofront.keystore.**DuplicatePublicKeyError**
Exception that rise when the given public key is already registered.

**class** geofront.keystore.**KeyStore**
The key store backend interface. Every key store has to guarantee that public keys are unique for all identities i.e. the same public key can't be registered across more than an identity.

**deregister**(*identity: geofront.identity.Identity*, *public_key: paramiko.pkey.PKey*)
Remove the given `public_key` of the `identity`. It silently does nothing if there isn't the given `public_key` in the store.

**Parameters**

- **ientity** – the owner identity
- **public_key** (`paramiko.pkey.PKey`) – the public key to remove

**Raises** geofront.keystore.AuthorizationError when the given `identity` has no required permission to the key store

**list_keys**(*identity: geofront.identity.Identity*) → collections.abc.Set
List registered public keys of the given `identity`.

**Parameters** **identity** (`Identity`) – the owner of keys to list

**Returns** the set of `paramiko.pkey.PKey` owned by the `identity`

**Return type** `collections.abc.Set`

**Raises** geofront.keystore.AuthorizationError when the given `identity` has no required permission to the key store

**register**(*identity: geofront.identity.Identity*, *public_key: paramiko.pkey.PKey*)
Register the given `public_key` to the `identity`.

**Parameters**

- **ientity** – the owner identity
- **public_key** (`paramiko.pkey.PKey`) – the public key to register

**Raises**

- **geofront.keystore.AuthorizationError** – when the given `identity` has no required permission to the key store
- **geofront.keystore.DuplicatePublicKeyError** – when the `public_key` is already in use

**exception** `geofront.keystore.`**`KeyStoreError`**
Exceptions related to `KeyStore` are an instance of this.

**exception** `geofront.keystore.`**`KeyTypeError`**
Unsupported public key type raise this type of error.

`geofront.keystore.`**`format_openssh_pubkey`**(*key: paramiko.pkey.PKey*) → str
Format the given `key` to an OpenSSH public key line, used by `authorized_keys`, `id_rsa.pub`, etc.

**Parameters** **key** (`paramiko.pkey.PKey`) – the key object to format

**Returns** a formatted openssh public key line

**Return type** `str`

`geofront.keystore.`**`get_key_fingerprint`**(*key: paramiko.pkey.PKey*, *glue: str=':'*) → str
Get the hexadecimal fingerprint string of the `key`.

**Parameters**

- **key** (`paramiko.pkey.PKey`) – the key to get fingerprint
- **glue** (`str`) – glue character to be placed between bytes. `':'` by default

**Returns** the fingerprint string

**Return type** `str`

`geofront.keystore.`**`parse_openssh_pubkey`**(*line: str*) → paramiko.pkey.PKey
Parse an OpenSSH public key line, used by `authorized_keys`, `id_rsa.pub`, etc.

**Parameters** **line** (`str`) – a line of public key

**Returns** the parsed public key

**Return type** `paramiko.pkey.PKey`

**Raises**

- **ValueError** – when the given `line` is an invalid format
- **KeyTypeError** – when it's an unsupported key type

### 7.4.4 `geofront.masterkey` — Master key management

Master key renewal process:

1. Create a new master key without updating the master key store.
2. Update every `authorized_keys` to authorize both the previous and the new master keys.
3. Store the new master key to the master key store, and remove the previous master key.
4. Update very `authorized_keys` to authorize only the new master key.

For more details, see also `TwoPhaseRenewal`.

Changed in version 0.2.0: `CloudMasterKeyStore` is moved from this module to `geofront.backends.cloud`. See `CloudMasterKeyStore`.

**exception** `geofront.masterkey.`**`EmptyStoreError`**
Exception that rises when there's no master key yet in the store.

**class** geofront.masterkey.**FileSystemMasterKeyStore**(*path: str*)
Store the master key into the file system. Although not that secure, but it might help you to try out Geofront.

**Parameters** **path** (str) – the path to save file. it has to end with the filename

**Raises** OSError when the path is not writable

**class** geofront.masterkey.**MasterKeyStore**
The master key store backend interface. It can have only one master key at the most.

**load**() → paramiko.pkey.PKey
Load the stored master key.

**Returns** the stored master key

**Return type** paramiko.pkey.PKey

**Raises** geofront.masterkey.EmptyStoreError when there's no master key yet in the store

**save**(*master_key: paramiko.pkey.PKey*)
Remove the stored master key, and then save the new master key. The operation should be atomic.

**Parameters** **master_key** (paramiko.pkey.PKey) – the new master key to replace the existing master key

**class** geofront.masterkey.**PeriodicalRenewal**(*servers: collections.abc.Set*, *key_store: geofront.masterkey.MasterKeyStore*, *interval: datetime.timedelta*, *bits: int=2048*, *start: bool=True*)
Periodically renew the master key in the separated background thread.

**Parameters**

- **servers** (collections.abc.Set) – servers to renew the master key. every element has to be an instance of Remote
- **key_store** (MasterKeyStore) – the master key store to update
- **interval** (datetime.timedelta) – the interval to renew
- **bits** (int) – the number of bits the generated key should be. it has to be 1024 at least, and a multiple of 256. 2048 by default
- **start** (bool) – whether to start the background thread immediately. True by default

New in version 0.2.0: The bits optional parameter.

**terminate**()
Graceful termination.

**class** geofront.masterkey.**TwoPhaseRenewal**(*servers: collections.abc.Set*, *old_key: paramiko.pkey.PKey*, *new_key: paramiko.pkey.PKey*)
Renew the master key for the given servers. It's a context manager for with statement.

```
# State: servers allow only old_key;
#        old_key is in the master_key_store
with TwoPhaseRenewal(servers, old_key, new_key):
    # State: *servers allow both old_key and new_key;*
    #        old_key is in the master_key_store
    master_key_store.save(new_key)
    # State: servers allow both old_key and new_key;
    #        *new_key is in the master_key_store.*
# State: *servers allow only new_key;*
#        new_key is in the master_key_store
```

**Parameters**

- **servers** (collections.abc.Set) – the set of Remote servers to renew their master key
- **old_key** (paramiko.pkey.PKey) – the previous master key to expire
- **new_key** (paramiko.pkey.PKey) – the new master key to replace old_key

geofront.masterkey.**read_private_key_file**(*file_: io.IOBase*) → paramiko.pkey.PKey

Read a private key file. Similar to PKey.from_private_key() except it guess the key type.

**Parameters** **file** (io.IOBase) – a stream of the private key to read

**Returns** the read private key

**Return type** paramiko.pkey.PKery

**Raises** paramiko.ssh_exception.SSHException when something goes wrong

geofront.masterkey.**renew_master_key**(*servers: collections.abc.Set*, *key_store: geofront.masterkey.MasterKeyStore*, *bits: int=2048*) → paramiko.pkey.PKey

Renew the master key. It creates a new master key, makes servers to authorize the new key, replaces the existing master key with the new key in the key_store, and then makes servers to deauthorize the old key. All these operations are done in a two-phase renewal transaction.

**Parameters**

- **servers** (collections.abc.Set) – servers to renew the master key. every element has to be an instance of Remote
- **key_store** (MasterKeyStore) – the master key store to update
- **bits** (int) – the number of bits the generated key should be. it has to be 1024 at least, and a multiple of 256. 2048 by default

**Returns** the created new master key

**Return type** paramiko.pkey.PKey

New in version 0.2.0: The bits optional parameter.

### 7.4.5 geofront.regen — Regen master key

New in version 0.2.0.

geofront.regen.**main**()

The main function of **geofront-key-regen** CLI program.

geofront.regen.**main_parser**(*parser: argparse.ArgumentParser=None*) → argparse.ArgumentParser

Create an ArgumentParser object for **geofront-key-regen** CLI program. It also is used for documentation through sphinxcontrib-autoprogram.

**Returns** a properly configured ArgumentParser

**Return type** argparse.ArgumentParser

geofront.regen.**regenerate**(*master_key_store: geofront.masterkey.MasterKeyStore*, *remote_set: collections.abc.Mapping*, *bits: int=2048*, ***, *create_if_empty: bool*, *renew_unless_empty: bool*)

Regenerate or create the master key.

### 7.4.6 geofront.remote — Remote sets

Every remote set is represented as a mapping (which is immutable, or mutable) of alias `str` to `Remote` object e.g.:

```
{
    'web-1': Remote('ubuntu', '192.168.0.5'),
    'web-2': Remote('ubuntu', '192.168.0.6'),
    'web-3': Remote('ubuntu', '192.168.0.7'),
    'worker-1': Remote('ubuntu', '192.168.0.25'),
    'worker-2': Remote('ubuntu', '192.168.0.26'),
    'db-1': Remote('ubuntu', '192.168.0.50'),
    'db-2': Remote('ubuntu', '192.168.0.51')
}
```

However, in the age of the cloud, you don't have to manage the remote set since the most of cloud providers offer their API to list provisioned remote nodes.

Geofront provides builtin `CloudRemoteSet`, a subtype of `collections.abc.Mapping`, that proxies to the list dynamically made by cloud providers.

Changed in version 0.2.0: `CloudRemoteSet` is moved from this module to `geofront.backends.cloud`. See `CloudRemoteSet`.

**class** `geofront.remote.`**AuthorizedKeyList**(*sftp_client: paramiko.sftp_client.SFTPClient*)

List-like abstraction for remote `authorized_keys`.

Note that the contents are all lazily evaluated, so in order to pretend heavy duplicate communications over SFTP use `list()` to eagerly evaluate e.g.:

```
lazy_list = AuthorizedKeyList(sftp_client)
eager_list = list(lazy_list)
# ... some modifications on eager_list ...
lazy_list[:] = eager_list
```

**Parameters** **sftp_client** (`paramiko.sftp_client.SFTPClient`) – the remote sftp connection to access `authorized_keys`

**FILE_PATH = '.ssh/authorized_keys'**

(`str`) The path of `authorized_keys` file.

**class** `geofront.remote.`**DefaultPermissionPolicy**

All remotes are listed and allowed for everyone in the team.

New in version 0.2.0.

**class** `geofront.remote.`**GroupMetadataPermissionPolicy**(*metadata_key: str*, *separator: str=None*)

Allow/disallow remotes according their metadata. It assumes every remote has a metadata key that stores a set of groups to allow. For example, suppose there's the following remote set:

```
{
    'web-1': Remote('ubuntu', '192.168.0.5', metadata={'role': 'web'}),
    'web-2': Remote('ubuntu', '192.168.0.6', metadata={'role': 'web'}),
    'web-3': Remote('ubuntu', '192.168.0.7', metadata={'role': 'web'}),
    'worker-1': Remote('ubuntu', '192.168.0.25',
                       metadata={'role': 'worker'}),
    'worker-2': Remote('ubuntu', '192.168.0.26',
                       metadata={'role': 'worker'}),
    'db-1': Remote('ubuntu', '192.168.0.50', metadata={'role': 'db'}),
```

```
    'db-2': Remote('ubuntu', '192.168.0.51', metadata={'role': 'db'})
}
```

and there are groups identified as `'web'`, `'worker'`, and `'db'`. So the following policy would allow only members who belong to the corresponding groups:

GroupMetadataPermissionPolicy('role')

**Parameters**

- **metadata_key** (`str`) – the key to find corresponding groups in metadata of each remote
- **separator** (`str`) – the character separates multiple group identifiers in the metadata value. for example, if the groups are stored as like `'sysadmin,owners'` then it should be `','`. it splits group identifiers by all whitespace characters by default

New in version 0.2.0.

**class** `geofront.remote.`**`PermissionPolicy`**

Permission policy determines which remotes are visible by a team member, and which remotes are allowed to SSH. So each remote can have one of three states for each team member:

**Listed and allowed** A member can SSH to the remote.

**Listed but disallowed** A member can be aware of the remote, but cannot SSH to it.

**Unlisted and disallowed** A member can't be aware of the remote, and can't SSH to it either.

**Unlisted but allowed** It is possible in theory, but mostly meaningless in practice.

The implementation of this interface has to implement two methods. One is `filter()` which determines whether remotes are listed or unlisted. Other one is `permit()` which determines whether remotes are allowed or disallowed to SSH.

New in version 0.2.0.

**`filter`**(*remotes: collections.abc.Mapping*, *identity: geofront.identity.Identity*, *groups: collections.abc.Set*) → collections.abc.Mapping

Determine which ones in the given `remotes` are visible to the `identity` (which belongs to `groups`). The resulted mapping of filtered remotes has to be a subset of the input `remotes`.

**Parameters**

- **remotes** (`collections.abc.Mapping`) – the remotes set to filter. keys are alias strings and values are `Remote` objects
- **identity** (`Identity`) – the identity that the filtered remotes would be visible to
- **groups** (`collections.abc.Set`) – the groups that the given `identity` belongs to. every element is a group identifier and `collections.abc.Hashable`

**`permit`**(*remote: geofront.remote.Remote*, *identity: geofront.identity.Identity*, *groups: collections.abc.Set*) → bool

Determine whether to allow the given `identity` (which belongs to `groups`) to SSH the given `remote`.

**Parameters**

- **remote** (`Remote`) – the remote to determine
- **identity** (`Identity`) – the identity to determine
- **groups** (`collections.abc.Set`) – the groups that the given `identity` belongs to. every element is a group identifier and `collections.abc.Hashable`

**class** geofront.remote.**Remote**(*user: str*, *host: str*, *port: numbers.Integral=22*, *metadata: collections.abc.Mapping={}*)

Remote node to SSH.

**Parameters**

- **user** (str) – the username to **ssh**
- **host** (str) – the host to access
- **port** (numbers.Integral) – the port number to **ssh**. the default is 22 which is the default **ssh** port
- **metadata** (collections.abc.Mapping) – optional metadata mapping. keys and values have to be all strings. empty by default

New in version 0.2.0: Added optional `metadata` parameter.

**host = None**
(Address) The hostname to access.

**metadata = None**
(collections.abc.Mapping) The additional metadata. Note that it won't affect to `hash()` of the object, nor ==/= comparison of the object.

New in version 0.2.0.

**port = None**
(numbers.Integral) The port number to SSH.

**user = None**
(str) The username to SSH.

geofront.remote.**authorize**(*public_keys: collections.abc.Set*, *master_key: paramiko.pkey.PKey*, *remote: geofront.remote.Remote*, *timeout: datetime.timedelta*) → datetime.datetime

Make an one-time authorization to the `remote`, and then revokes it when `timeout` reaches soon.

**Parameters**

- **public_keys** (collections.abc.Set) – the set of public keys (paramiko.pkey.PKey) to authorize
- **master_key** (paramiko.pkey.PKey) – the master key (*not owner's key*)
- **remote** (Remote) – a remote to grant access permission
- **timeout** (datetime.timedelta) – the time an authorization keeps alive

**Returns** the expiration time

**Return type** datetime.datetime

### 7.4.7 geofront.server — Key management service

Although Geofront provides **geofront-server**, a CLI to run the server, it also provides an interface as a WSGI application as well. Note that there might some limitations like lack of periodical master key renewal.

First of all, the server need a configuration, there are several ways to configure it.

**app.config.from_pyfile()** If you can freely execute arbitrary Python code before start the server, the method is the most straightforward way to configure the server. Note that the argument should be an absolute path, because it interprets paths relative to the path of Geofront program, not the current working directory (CWD).

There also are other methods as well:

- `from_object()`
- `from_json()`
- `from_envvar()`

**GEOFRONT_CONFIG** If you can't execute any arbitrary Python code, set the `GEOFRONT_CONFIG` environment variable. It's useful when to use a CLI frontend of the WSGI server e.g. **gunicorn**, **waitress-serve**.

```
$ GEOFRONT_CONFIG="/etc/geofront.cfg.py" gunicorn geofront.server:app
```

Then you can run a Geofront server using your favorite WSGI server. Pass the following WSGI application object to the server. It's a documented endpoint for WSGI:

```
geofront.server:app
```

`geofront.server.`**AUTHORIZATION_TIMEOUT = datetime.timedelta(0, 60)**
(`datetime.timedelta`) How long does each temporary authorization keep alive after it's issued. A minute.

**class** `geofront.server.`**FingerprintConverter**(**args*, ***kwargs*)
Werkzeug custom converter which accepts valid public key fingerprints.

**class** `geofront.server.`**Token**
(`type`) The named tuple type that stores a token.

**expires_at**
Alias for field number 1

**identity**
Alias for field number 0

**class** `geofront.server.`**TokenIdConverter**(**args*, ***kwargs*)
Werkzeug custom converter which accepts valid token ids.

`geofront.server.`**add_public_key**(*token_id: str*)
Register a public key to the token. It takes an OpenSSH public key line through the request content body.

```
POST /tokens/0123456789abcdef/keys/ HTTPS/1.1
Accept: application/json
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==

HTTPS/1.1 201 Created
Content-Type: text/plain
Location: /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters** **token_id** (`str`) – the token id that holds the identity

**Status 201** when key registration is successful

**Status 400** (`unsupported-key-type`) when the key type is unsupported, or (`invalid-key`) the key format is invalid, or (`deuplicate-key`) the key is already used

**Status 415** (`unsupported-content-type`) when the *`Content-Type`* is not *`text/plain`*

`geofront.server.`**app = <Flask 'geofront.server'>**
(`flask.Flask`) The WSGI application of the server.

`geofront.server.`**authenticate**(*token_id: str*)
Finalize the authentication process. It will be shown on web browser.

**Parameters** **token_id** (`str`) – token id created by `create_access_token()`

**Status 400** when authentication is failed

**Status 404** when the given `token_id` doesn't exist

**Status 403** when the `token_id` is already finalized

**Status 200** when authentication is successfully done

`geofront.server.`**`authorize_remote`**(*token_id: str*, *alias: str*)

Temporarily authorize the token owner to access a remote. A made authorization keeps alive in a minute, and then will be expired.

```
POST /tokens/0123456789abcdef/remotes/web-1/ HTTPS/1.1
Accept: application/json
Content-Length: 0

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "success": "authorized",
  "remote": {"user": "ubuntu", "host": "192.168.0.5", "port": 22},
  "expires_at": "2014-04-14T14:57:49.822844+00:00"
}
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity
- **alias** (`str`) – the alias of the remote to access

**Status 200** when successfully granted a temporary authorization

**Status 404** (`not-found`) when there's no such remote

`geofront.server.`**`create_access_token`**(*token_id: str*)

Create a new access token.

```
PUT /tokens/0123456789abcdef/ HTTPS/1.1
Accept: application/json
Content-Length: 0

HTTPS/1.1 202 Accepted
Content-Type: application/json
Date: Tue, 15 Apr 2014 03:44:43 GMT
Expires: Tue, 15 Apr 2014 04:14:43 GMT
Link: <https://example.com/login/page/?redirect_uri=...>; rel=next

{
  "next_url": "https://example.com/login/page/?redirect_uri=..."
}
```

**Parameters** **token_id** (`str`) – an arbitrary token id to create. it should be enough random to avoid duplication

**Status 202** when the access token is prepared

**Resheader Link** the link owner's browser should redirect to

geofront.server.**delete_public_key**(*token_id: str*, *fingerprint: bytes*)
Delete a public key.

```
DELETE /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47/ HTTPS/1.1
Accept: application/json

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "72:00:60:24:66:e8:2d:4d:2a:2a:a2:0e:7b:7f:fc:af":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCom2CDLekY...5CeYsvSdrTWA5 ",
  "78:8a:09:c8:c1:24:5c:89:76:92:b0:1e:93:95:5d:48":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA16iSKKjFHOgj...kD62SYXNKY9c= ",
  "ab:3a:fb:30:44:e3:5e:1e:10:a0:c9:9a:86:f4:67:59":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAzzF8c07pzgKk...r+b6Q9VnWWQ== "
}
```

**Parameters**

- **token_id** (str) – the token id that holds the identity
- **fingerprint** (bytes) – the fingerprint of a public key to delete

**Status 200** when the public key is successfully deleted

**Status 404** (not-found) when there's no such public key

geofront.server.**get_identity**(*token_id: str*) → geofront.identity.Identity
Get the identity object from the given token_id.

**Parameters** **token_id** (str) – the token id to get the identity it holds

**Returns** the identity the token holds

**Return type** Identity

**Raises** werkzeug.exceptions.HTTPException 404 Not Found (token-not-found) when the token does not exist. 412 Precondition Failed (unfinished-authentication) when the authentication process is not finished yet. 410 Gone (expired-token) when the token was expired. 403 Forbidden (not-authorized) when the token is not unauthorized.

geofront.server.**get_key_store**() → geofront.keystore.KeyStore
Get the configured key store implementation.

**Returns** the configured key store

**Return type** KeyStore

**Raises** RuntimeError when 'KEY_STORE' is not configured, or it's not an instance of KeyStore

geofront.server.**get_master_key_store**() → geofront.masterkey.MasterKeyStore
Get the configured master key store implementation.

**Returns** the configured master key store

**Return type** MasterKeyStore

**Raises** RuntimeError when 'MASTER_KEY_STORE' is not configured, or it's not an instance of MasterKeyStore

geofront.server.**get_permission_policy**() → geofront.remote.PermissionPolicy
Get the configured permission policy.

**Returns** the configured permission policy

**Return type** PermissionPolicy

**Raises** RuntimeError if 'PERMISSION_POLICY' is not configured, or it's not an instance of PermissionPolicy

New in version 0.2.0.

geofront.server.**get_public_key**(*token_id: str*, *fingerprint: bytes*) → paramiko.pkey.PKey
Internal function to find the public key by its fingerprint.

**Parameters**

- **token_id** (str) – the token id that holds the identity
- **fingerprint** (bytes) – the fingerprint of a public key to find

**Returns** the found public key

**Return type** paramiko.pkey.PKey

**Raises** werkzeug.exceptions.HTTPException (not-found) when there's no such public key

geofront.server.**get_remote_set**() → collections.abc.Mapping
Get the configured remote set.

**Returns** the configured remote set

**Return type** collections.abc.Mapping

**Raises** RuntimeError if 'REMOTE_SET' is not configured, or it's not a mapping object

geofront.server.**get_team**() → geofront.team.Team
Get the configured team implementation, an instance of team.Team.

It raises RuntimeError if 'TEAM' is not configured.

geofront.server.**get_token_store**() → werkzeug.contrib.cache.BaseCache
Get the configured token store, an instance of werkzeug.contrib.cache.BaseCache.

It raises RuntimeError if 'TOKEN_STORE' is not configured, but it just warns RuntimeWarning when it comes to debug mode.

**Returns** the configured session store

**Return type** werkzeug.contrib.cache.BaseCache

**Raises** RuntimeError when 'TOKEN_STORE' is not configured, or the value is not an instance of werkzeug.contrib.cache.BaseCache

geofront.server.**list_public_keys**(*token_id: str*)
List registered keys to the token owner.

```
GET /tokens/0123456789abcdef/keys/ HTTPS/1.1
Accept: application/json

HTTPS/1.1 200 OK
Content-Type: application/json

{
  "50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q== ",
  "72:00:60:24:66:e8:2d:4d:2a:2a:a2:0e:7b:7f:fc:af":
    "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCom2CDLekY...5CeYsvSdrTWA5 ",
  "78:8a:09:c8:c1:24:5c:89:76:92:b0:1e:93:95:5d:48":
```

```
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA16iSKKjFHOgj...kD62SYXNKY9c= ",
  "ab:3a:fb:30:44:e3:5e:1e:10:a0:c9:9a:86:f4:67:59":
    "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAzzF8c07pzgKk...r+b6Q9VnWWQ== "
}
```

**Parameters** **token_id** (`str`) – the token id that holds the identity

**Status 200** when listing is successful, even if there are no keys

`geofront.server.`**`main`**`()`
The main function for **geofront-server** CLI program.

`geofront.server.`**`main_parser`**`()` → argparse.ArgumentParser
Create an `ArgumentParser` object for **geofront-server** CLI program. It also is used for documentation through sphinxcontrib-autoprogram.

**Returns** a properly configured `ArgumentParser`

**Return type** `argparse.ArgumentParser`

`geofront.server.`**`master_key`**(*token_id: str*)
Public part of the master key in OpenSSH authorized_keys (public key) format.

```
GET /tokens/0123456789abcdef/masterkey/ HTTPS/1.1
Accept: text/plain

HTTPS/1.1 200 OK
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters** **token_id** (`str`) – the token id that holds the identity

**Status 200** when the master key is available

**Status 500** when the master key is unavailable

`geofront.server.`**`public_key`**(*token_id: str*, *fingerprint: bytes*)
Find the public key by its `fingerprint` if it's registered.

```
GET /tokens/0123456789abcdef/keys/50:5a:9a:12:75:8b:b0:88:7d:7a:8d:66:29:63:d0:47/ HTTPS/1.1
Accept: text/plain

HTTPS/1.1 200 OK
Content-Type: text/plain

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDAEMUvjBcX.../MuLLzC/m8Q==
```

**Parameters**

- **token_id** (`str`) – the token id that holds the identity
- **fingerprint** (`bytes`) – the fingerprint of a public key to find

**Status 200** when the public key is registered

**Status 404** (`not-found`) when there's no such public key

`geofront.server.`**`remote_dict`**(*remote: geofront.remote.Remote*) → collections.abc.Mapping
Convert a `remote` to a simple dictionary that can be serialized to JSON.

**Parameters** **remote** (Remote) – a remote instance to serialize

**Returns** the converted dictionary

**Return type** collections.abc.Mapping

geofront.server.**server_endpoint**()
The endpoint of HTTP API which provide the url to create a new token.

```
GET / HTTPS/1.1
Accept: application/json

HTTP/1.0 200 OK
Content-Type: application/json
Link: <https://example.com/tokens/>; rel=tokens

{
  "tokens_url": "https://example.com/tokens/"
}
```

**Resheader Link** the url to create a new token. the equivalent to the response content

**Status 200** when the server is available

New in version 0.2.0.

geofront.server.**server_version**(*response: flask.wrappers.Response*) → flask.wrappers.Response
Indicate the version of Geofront server using *Server* and *X-Geofront-Version* headers.

geofront.server.**token**(*token_id: str*)
The owner identity that the given token holds if the token is authenticated. Otherwise it responds 403 Forbidden, 404 Not Found, 410 Gone, or 412 Precondition Failed. See also get_identity().

```
GET /tokens/0123456789abcdef/ HTTPS/1.1
Accept: application/json

HTTPS/1.0 200 OK
Content-Type: application/json
Link: <https://example.com/tokens/0123456789abcdef/remo...>; rel=remotes
Link: <https://example.com/tokens/0123456789abcdef/keys/>; rel=keys
Link: <https://example.com/tokens/0123456789abcdef/ma...>; rel=masterkey

{
  "identifier": "dahlia",
  "team_type": "geofront.backends.github.GitHubOrganization",
  "remotes_url": "https://example.com/tokens/0123456789abcdef/remotes/",
  "keys_url": "https://example.com/tokens/0123456789abcdef/keys/",
  "master_key_url": "https://example.com/tokens/0123456789abcdef/mas..."
}
```

**Parameters** **token_id** (str) – the token id that holds the identity

**Resheader Link** the url to list remotes (rel=remotes), public keys (rel=keys), and master key (rel=masterkey)

**Status 200** when the token is authenticated

Changed in version 0.2.0: The response contains "remotes_url", "keys_url", and "master_key_url", and equivalent three *Link* headers.

### 7.4.8 geofront.team — Team authentication

Geofront doesn't force you to manage team members by yourself. Instead it hides how to manage team members, and offers `Team`, the layering interface to implement custom team data provider e.g. `GitHubOrganization`.

It is theologically possible to implement a straightforward RDBMS-backed team provider, but we rather recommend to adapt your existing team data instead e.g. GitHub organization, Google Apps organization, Bitbucket team.

**exception** `geofront.team.`**`AuthenticationError`**
Authentication exception which rise when the authentication process has trouble including network problems.

**class** `geofront.team.`**`Team`**
Backend interface for team membership authentication.

Authorization process consists of three steps (and therefore every backend subclass has to implement these three methods):

1. `request_authentication()` makes the url to interact with the owner of the identity to authenticate. I.e. the url to login web page of the backend service.
2. `authenticate()` finalize authentication of the identity, and then returns `Identity`.
3. `authorize()` tests the given `Identity` belongs to the team. It might be a redundant step for several backends, but is a necessary step for some backends that distinguish identity authentication between team membership authorization. For example, Any Gmail users can authenticate they own their Gmail account, but only particular users can authenticate their account belongs to the configured Google Apps organization.

**`authenticate`**(*auth_nonce: str*, *requested_redirect_url: str*, *wsgi_environ: collections.abc.Mapping*) → geofront.identity.Identity
Second step of authentication process, to create a verification token for the identity. The token is used by `authorize()` method, and the key store as well (if available).

**Parameters**

- **auth_nonce** (`str`) – a random string to guarantee it's a part of the same process to `request_authentication()` call followed by this which is the first step
- **requested_redirect_url** (`str`) – a url that was passed to `request_authentication()`'s `redirect_url` parameter
- **wsgi_environ** (`collections.abc.Mapping`) – forwarded wsgi environ dictionary

**Returns** an identity which contains a verification token

**Return type** `Identity`

**Raises** **geofront.team.AuthenticationError** when something goes wrong e.g. network errors, the user failed to verify their ownership

**`authorize`**(*identity: geofront.identity.Identity*) → bool
The last step of authentication process. Test whether the given `identity` belongs to the team.

Note that it can be called every time the owner communicates with Geofront server, out of authentication process.

**Parameters** **identity** (`Identity`) – the identity to authorize

**Returns** `True` only if the `identity` is a member of the team

**Return type** `bool`

**`list_groups`**(*identity: geofront.identity.Identity*) → collections.abc.Set
List the all groups that the given `identity` belongs to. Any hashable value can be an element to represent a group e.g.:

```
{1, 4, 9}
```

Or:

```
{'owners', 'programmers'}
```

Whatever value the set consists of these would be referred by `Remote` objects.

Some team implementations might not have a concept like groups. It's okay to return always an empty set then.

**Parameters** **identity** (`Identity`) – the identity to list his/her groups

**Returns** the set of groups associated with the `identity`

**Return type** `collections.abc.Set`

New in version 0.2.0.

**request_authentication**(*auth_nonce: str*, *redirect_url: str*) → str

First step of authentication process, to prepare the "sign in" interaction with the owner. It typically returns a url to the login web page.

**Parameters**

- **auth_nonce** (`str`) – a random string to guarantee it's a part of the same process to following `authenticate()` call which is the second step
- **redirect_url** (`str`) – a url that owner's browser has to redirect to after the "sign in" interaction finishes

**Returns** a url to the web page to interact with the owner in their browser

**Return type** `str`

### 7.4.9 geofront.util — Utilities

geofront.util.**typed**(*function: function*) → function

Automatically check argument types using function's annotated parameters. For example, the following code will raise `TypeError`:

```
>>> @typed
... def add(a: int, b: int):
...     return a + b
...
>>> add('strings are not ', 'accepted')
```

**Parameters** **function** (`types.FunctionType`) – a function to make automatically type checked

### 7.4.10 geofront.version — Version data

geofront.version.**VERSION = '0.2.1'**

(`str`) The version string e.g. `'1.2.3'`.

geofront.version.**VERSION_INFO = (0, 2, 1)**

(`tuple`) The triple of version numbers e.g. `(1, 2, 3)`.

CHAPTER 8

# Indices and tables

- *genindex*
- *modindex*
- *search*

## /

GET /, 19

## /tokens

PUT /tokens/(token_id:token_id)/, 23
GET /tokens/(token_id:token_id)/, 24
GET /tokens/(token_id:token_id)/authenticate/,
21
POST /tokens/(token_id:token_id)/keys/,
23
GET /tokens/(token_id:token_id)/keys/,
22
DELETE /tokens/(token_id:token_id)/keys/(fingerprint:fingerprint)/,
21
GET /tokens/(token_id:token_id)/keys/(fingerprint:fingerprint)/,
20
GET /tokens/(token_id:token_id)/masterkey/,
21
GET /tokens/(token_id:token_id)/remotes/,
22
POST /tokens/(token_id:token_id)/remotes/(alias)/,
20

## c

config, 25

## g

geofront, 30
geofront.backends, 30
geofront.backends.cloud, 30
geofront.backends.dbapi, 32
geofront.backends.github, 32
geofront.identity, 33
geofront.keystore, 34
geofront.masterkey, 35
geofront.regen, 37
geofront.remote, 37
geofront.server, 40
geofront.team, 46
geofront.util, 48
geofront.version, 48