gru Documentation

Release 0.1.0

SimilarWeb LTD

Apr 09, 2017

Contents

CHAPTER 1

What is GRU?

GRU is a host-centric inventory management system. It is used to visualize and provide context on individual servers and more importantly, on groups of servers.

It was designed in order to help operations teams, as well as developers to better understand their infrastructure and to provide a unified view of available compute resources.

This is what it looks like:

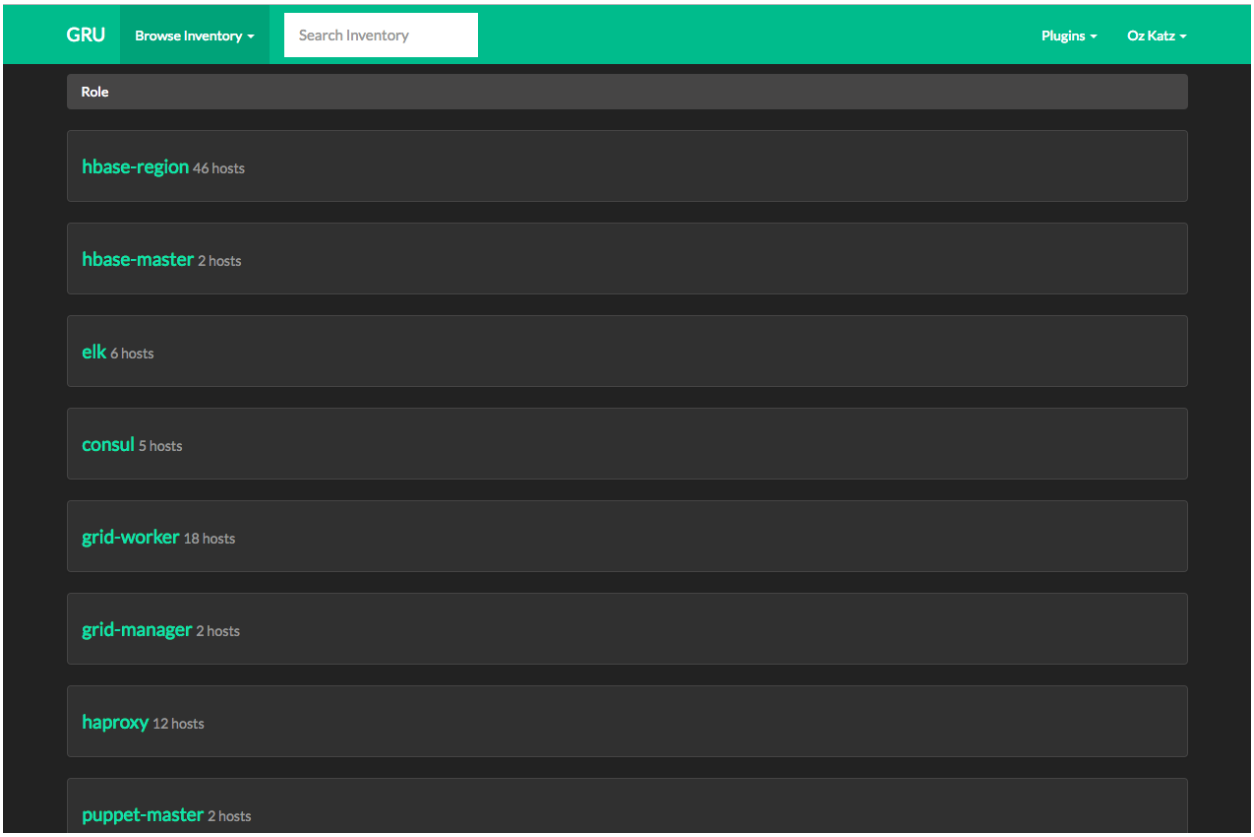

Fig. 1.1: *GRU Host breakdown by role*

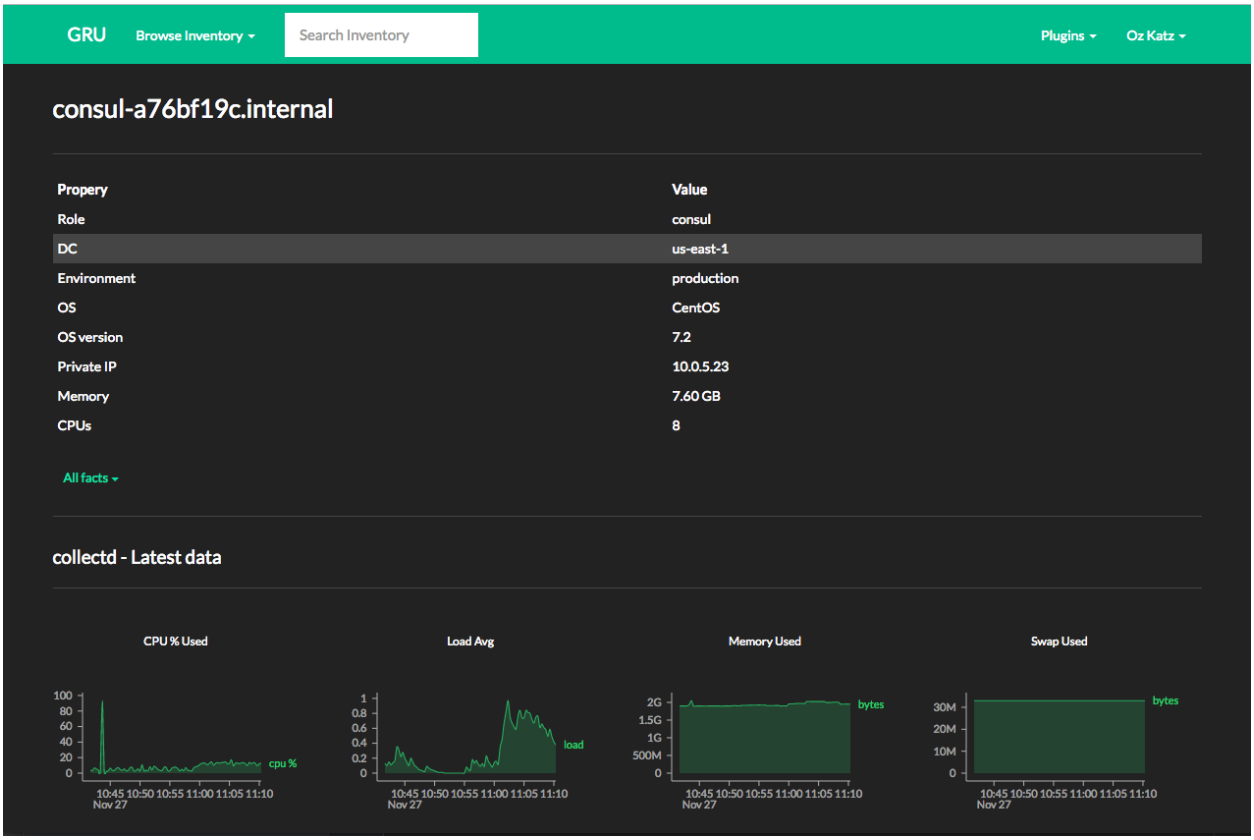

Fig. 1.2: *GRU Host info page*

CHAPTER 2

Motivation

Understanding which compute resources exist (and which business context they serve) can be tricky. In many cases, this information is scattered across several systems.

GRU is designed to give an accurate view of compute resources available to us. The main primitive in GRU is the **Host**. By providing different ways to slice and dice an inventory (or, collection of hosts) and allowing plugins to give external context regarding each host, we can better discover, understand and optimize our infrastructure.

Plugins, that are at the center of GRU, are used to integrate information and actions from external systems. These can include monitoring, CMDB, real time metrics, cost info, service checks and more. By uniting all these into one place we can easily reason about our resources and provide visibility to operations and software engineers.

Contents:

Quick Start

Installation

There are two ways to get started with GRU: Either clone the project and run the python server - or run it using the the Docker image.

We recommend using Docker, since it eliminates the need to install all the different python and OS dependencies.

If you plan on contributing or simply hacking on GRU itself, you should probably use the local server instead. Keep in mind that GRU provides a pretty flexible plugin system. You can develop your own plugins and keep using the docker image.

Basic Configuration

Before we can run GRU, we need to setup a yaml file describing how to access our inventory and optionally how to authenticate and authorize users as well as set other configuration parameters to adjust GRU to our liking.

Here's a very basic example of how this configuration might look like, without any authentication or authorization:

```
---
flask:
  # Keep this one random. used by the session system.
  secret_key: 'fojegj340fuccotnhvi39yombpris'

# Here we set up EC2 as our source of inventory.
inventory:
  provider: gru.contrib.inventory.providers.EC2Provider
  config:
   accounts:
      # setting aws_access_key and aws_secret_access_key is optional
      # otherwise, will be taken from ~/.aws/credentials
      # or environment variables
      - regions: ['us-east-1']
  # Whiche metadata field to use as a hostname
  host_display_name_field: tag:Name
  # Fields to create logical groups of hosts by
  group_by:
    - field: tag:stack
      title: Stack
    - field: tag:role
      title: Role
    - field: region
      title: Region
  # Fields visible when showing a table of hosts
  host_table_fields:
    - field: tag:role
      title: Role
    - field: tag:environment
      title: Env
    - field: architecture
      title: Arch
    - field: instance-state-name
      title: Status
    - field: private-ip-address
      title: IP
  # Fields visible by default when looking at a single host
  # (The user can toggle to view remaining fields)
  host_info_fields:
    - field: tag:role
      title: Role
    - field: region
      title: DC
    - field: tag:environment
      title: Env
    - field: architecture
      title: Arch
    - field: instance-state-name
      title: Status
    - field: private-ip-address
      title: IP
```

Please review the *Configuration Reference* section or check out the `examples/` directory on Github.

Running using Docker

Assuming you already have a yaml configuration file named `gru.yaml`, all we need to do is run the docker image:

```
$ docker run --rm -it \
    -v $PWD:/etc/gru \
    -e GRU_SETTINGS=/etc/gru/gru.yaml \
    -e "AWS_ACCESS_KEY_ID=AKXXXXXXXXXXXXXXXXXX" \  # Set this to your AWS keys
    -e "AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" \
    -p 5000:5000 \
    similarweb/gru:latest
```

If you are using custom plugins, simply mount the module directory and reflect that in your settings file:

```
$ docker run --rm -it \
    -v $PWD:/etc/gru \
    -e GRU_SETTINGS=/etc/gru/gru.yaml \
    -e "AWS_ACCESS_KEY_ID=AKXXXXXXXXXXXXXXXXXX" \  # Set this to your AWS keys
    -e "AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" \
    -v /path/to/plugins:/opt/gru-plugins \
    -p 5000:5000 \
    similarweb/gru:latest
```

This will automatically pull the gru image from docker hub, and run a server using the `gru.yaml` settings file.

Running a local server

Running a local server has been tested on Debian and Ubuntu linux. It should be able to run on OSX and Windows as well, but it hasn't been thoroughly tested.

First, install GRU's dependencies:

```
$ apt-get install --no-install-recommends \
    python-dev \
    libssl-dev \
    libsasl2-dev \
    libldap2-dev
```

Once that's done, we need to clone the GRU project:

```
$ git clone git://github.com/similarweb/gru .
```

Then, install all the required Python dependencies. Preferably, this step should be done in a virtualenv):

```
$ pip install -r requirements.txt
```

Now, run the server itself using the settings file you created:

```
$ GRU_SETTINGS="/path/to/gru.yaml" python app.py runserver
```

A server will be started, listening on http://localhost:5000

Inventory Providers

GRU supports pluggable inventory providers. Inventory providers fill in the list of hosts that make up GRU's view of the world. They allow grouping, listing and searching for hosts, as well as provide metadata for every host in the inventory.

Out of the box, GRU provides two different inventory providers: an ElasticSearch provider that assumes hosts appear as documents in an ElasticSearch cluster; and an EC2 provider that queries the AWS EC2 API

Configuring an Inventory Provider

at the most basic level, you need to set the *inventory.provider* configuration parameter to a fully qualified class name of a class that implements an inventory provider.

ElasticSearch Provider

The ElasticSearch provider assumes that you have configured an ElasticSearch cluster with an index containing hosts as documents. The document IDs in the index correlate to host IDs, and the documents are JSON encoded dictionaries of host metadata.

It's actually pretty simple to create such an index. See the provided mapping file and an example script that populates the hosts index

To enable the ElasticSearch provider, please use `gru.contrib.inventory.providers.ElasticSearchProvider` as the value for `inventory.provider` and refer to the *provider documentation*.

Here's a snippet of such a configuration:

```
---
inventory:
  provider: gru.contrib.inventory.providers.ElasticSearchProvider
  config:
    index: gru-inventory
    timeout_seconds: 10
    hosts:
      # HTTP endpoints to your elasticsearch clusters.
      # No need to specify all servers, they are used for discovery
      - http://10.0.0.1:9200
      - http://10.0.0.2:9200
  ...
```

EC2 Provider

To enable the EC2 provider, please use `gru.contrib.inventory.providers.EC2Proivder` as the value for `inventory.provider` and refer to the *provider documentation*.

The EC2 provider supports aggregation of hosts from several different AWS accounts and/or regions.

Here's a snippet of such a configuration:

```
---
provider: gru.contrib.inventory.providers.EC2Provider
config:
  accounts:
    - aws_access_key_id: AKXXXXXXXXXXXXXXXXXX
      aws_secret_access_key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      regions: ['us-east-1', 'us-west-2']
```

Writing your own

Writing an inventory provider is covered under *Writing your own inventory providers*

Authentication and Authorization

GRU supports pluggable authentication and authorization.

Authorization

To make things simple, authorization is done via user groups: a plugin can configure an `allowed_groups` variable that lists group names that have authorization to use the plugin.

Configuring an Authentication Backend

Configuring the Authentication backend is done via the `authentication.backend` and `authentication.config` configuration parameters. Both are covered in the *configuration documentation*

Dummy authentication backend

By default, gru sets the authentication backend to `gru.contrib.auth.backends.DummyBackend`.

This backend, as its name suggests, does nothing. No need for users to login, and users have no groups associated with them.

If you use GRU in a network where only authorized personnel have access to the web interface, this might be enough. Make sure you protect this network and that you don't accidentally expose it to the world.

For all other situations, it is recommended to use a real authentication backend such as the LDAP one described below.

LDAP authentication backend

The LDAP backend is used to authenticate users against an existing LDAP server (such as Active Directory or OpenLDAP).

Once a user has been authenticated, the LDAP backend also populates user's group names, to allow plugins to authorize only users of specific groups.

To enable the LDAP backend, please use `gru.contrib.auth.backends.LdapBackend` as the value for `authentication.backend` and refer to the *backend documentation* for details on how to populate the `authentication.config` dictionary for the LDAP backend.

Here's a snippet of such a configuration:

```
---
authentication:
  backend: gru.contrib.auth.backends.LdapBackend
  config:
    server: 10.0.0.7
    port: 3268
    bind_user: 'CN=gru,OU=Internal Tools,DC=mycompany,DC=com'
    bind_password: '**************'
    user_query: '(sAMAccountName=$username)'
```

Writing your own

Writing an authentication and authorization backends is covered under *Writing your own authentication backends*

Using plugins

Gru was built with the concept of extendability in mind. Since an inventory of hosts can have many different usages and different companies have a different view of the world (also with regards to how their infrastructure is managed), there's built-in support for 2 main types of plugins: *Host widgets* and *Page plugins*

Host Widgets

Host Widgets are plugins that live within the context of a single host.

They are generally used to provide additional information about the host from external systems (say monitoring, configuration management, cost, etc) or to support actions to be carried out for that host.

Page plugins

Page plugins are plugins that transcend hosts in general. As their name suggests, they are pages that are rendered within GRU and have access to the host inventory system. They are generally designed to complement GRU's use-cases by adding more host/fleet related functionality.

For example, at SimilarWeb, we built a page plugin that displays currently open incidents, linking back to relevant stacks and servers - which is used by on-call personnel.

Loading plugins

In order to load an existing plugin, you'll need to do the following:

1. Make sure the plugin's python module is in gru's `sys.path` (python's search path for modules).

   The easiest way to accomplish this, is to either place the plugins under the default *plugins.directories*.

   Alternatively, you can specify your own directories to override the default paths.

2. For plugins that require configuration, see their documentation and update your GRU settings YAML file accordingly.
3. Restart the GRU web server for the changes to get picked up.

Developing Plugins

To understand the basic concept of plugins and how to use them, refer to the *usage documentation*

This document describes the process and APIs required to develop your own plugins.

Plugin Architecture Overview

GRU Plugins are Python classes that subclass abstract plugin classes.

For example, to write your own *Host Widget*, you'll need to subclass the *gru.plugins.base.hostwidget.HostWidgetPlugin* class.

On startup, the server will do the following:

1. Add the directories listed under *plugins.directories* to Python's `sys.path`
2. It will then iterate over the module names listed in `plugins.modules` and import them
3. For each imported module, the server will iterate over all classes, looking for ones that subclass any of GRU's internal abstract plugin types
4. An object will be instantiated out of each class found. The default `on_load()` method will be called. This is where plugin startup code should live. Things like setting up database connections should be done there
5. For modules that expose views, a Flask Blueprint will be registered and used when serving these views

All plugins subclass the following abstract class:

class `gru.plugins.base.`**`BasePlugin`**

`on_load(self)`:
Defaults to doing nothing. If you have any startup logic, this is the place to put it.

Host Widgets

To write your own Host widget you'll need to provide the following:

1. A subclass of `gru.plugins.base.hostwidget.HostWidgetPlugin`
2. **A template file that extends `plugins/host_widget.html`. This Jinja template defines 3 parts that you'll be able to ov**

    - `pluginc``ss` - will be loaded at the head of the host info page. This is where `<style>` or `<link rel="...">` tags should go.
    - `pluginhtml` - the actual HTML to be rendered inside the host info page body, under the basic host information
    - `pluginjs` - will be placed at the bottom of the HTML body. This is where `<script>` tags should go.

    All template blocks will get the results of the plugin class' `get_context(self, host)` method. That's probably the one you'll need to override.

Here's a reference of the plugin class:

class `gru.plugins.base.hostwidget.`**`HostWidgetPlugin`**

`template_name`
A name of a template file to render. The file must extend `plugins/host_widget.html` and define the blocks listed above.

You can define a `templates/` directory inside your module's parent directory. If it exists, GRU will add it to the template search path.

get_title(self):
You must provide an implementation of this method. It should return a string that will be used as the title for this widget in the host info page.

qualify(self, host):
Given a *gru.plugins.inventory.Host* object, should return `True` or `False`. If `True`, the widget will be rendered for the given host's info page.

This is useful If you have widgets that don't make sense for every host, so you want to display them only where appropriate.

def get_context(self, host):
Given a *gru.plugins.inventory.Host* object, should return a python dictionary to be used as context for the template file. The default implementation simply returns `{}`, so you'll likely want to override it.

def ajax_request(self, request):
Useful if you want the widget to be able to send AJAX requests back to the server. The method will receive a Flask request object, and should return a valid Flask response.

Page Plugins

To write your own Page Plugin you'll need to provide the following:

1. A subclass of *gru.plugins.base.page.PagePlugin*
2. **Optionally, if this page plugin returns an HTML page to be rendered within GRU: A template file that extends `plugins/`**

    - `pluginhtml` - the actual HTML to be rendered. GRU will wrap this with the default navbar, footer, etc.
    - `pluginjs` - will be placed at the bottom of the HTML body. This is where `<script>` tags should go.

3. If the plugin defines a `get_title()` method, it will be linked to from the navigation bar, in the `plugins` dropdown.

Here's a reference of the plugin class:

class `gru.plugins.base.page.`**PagePlugin**

template_name
Optional - a name of a template file to render. The file must extend `plugins/page.html` and define the blocks listed above.

You can define a `templates/` directory inside your module's parent directory. If it exists, GRU will add it to the template search path.

You may also define a `static/` directory, containing static resources such as images, scripts and CSS files. To later include them within your template, use the `plugin_static()` function within your template. Example:

```
<script src="{{plugin_static('myplugin/js/plugin.js')}}"></script>
```

This will be replaced with the URL given to your `static/myplugin/js/plugin.js` file.

get_title(self):
Should return a string. It will be used to register the plugin in the plugins dropdown, in GRU's navigation bar. The default implementation returns `None` meaning it won't be registered in the plugins dropdown.

handler(self, request):
The method will receive a Flask request object, and should return a valid Flask response. You may simply `return self.render()` here, which will render the template you provided using the `template_name` attribute.

Sometimes it's useful to register a PagePlugin which returns a JSON response. Generally it will be called using an AJAX call by another plugin. In such a case, you should keep the default `get_title(self)` implementation and probably leave `template_name` blank as well.

render(self, http_status=200, **kwargs):
A helper method that renders a Jinja template and return a Flask response. Whatever you passed in as `kwargs` will be added to the Jinja context when rendering the template.

The template to render will be taken from the `template_name` attribute.

Inventory Providers

Writing an Inventory Provider requires subclassing *gru.plugins.inventory.InventoryProvider*.

To use the plugin you developed, you'll need to make sure the module is loaded (specified under `plugins.modules`) and that `inventory.provider` points to your subclass' path.

class gru.plugins.inventory.**InventoryProvider**

host_group_breakdown(self, category):
Returns a list of *gru.plugins.inventory.HostCategory* objects. Example: if `category = "datacenter"`, an expected return value would be `[HostCategory("datacenter", "us-east-1", 10), HostCategory("datacenter", "us-west-2", 5)]`

list(self, category, group, sort_by=None, from_ind=None, to_ind=None):
Returns a *gru.plugins.inventory.HostList* object after filter by a field and value. Example: `provider.list("datacenter", "us-east-1")` will return all Hosts in the *us-east-1* datacenter

Use `from_ind` and to_ind`` to support pagination. Both values are zero-based integers.

`sort_by` is optional. If set, the list of *gru.plugins.inventory.Host* objects should be sorted by this field name.

host_search(self, query, from_ind=None, to_ind=None):
Given a query string, perform a search of hosts and returns a *gru.plugins.inventory.HostList* object.

`query` will be a string provided by the user in the search box.

Use `from_ind` and to_ind`` to support pagination. Both values are zero-based integers.

get_host_by_id(self, host_id):
Return a *gru.plugins.inventory.Host* object for the provided host_id.

Host, HostCategory and HostList

class gru.plugins.inventory.**Host**

A Host is the most basic primitive in GRU. A host will generally have some unique ID (a MAC address, an AWS instance ID, or similar) and a key/value mapping of metadata. Common metadata attributes may include OS version, IP address, role, datacenter, etc.

__init__(self, host_id, host_data=None):

Creates a new host by passing in a host identifier (should be unique) and a python dictionary describing host metadata information.

field(self, field_name, default=None):

Returns a field value from the host metadata.

Also supports fetching nested fields using "." notation. i.e. host.field('os.name') will work with this metadata: {'os': {'name': 'Linux'}

---

class gru.plugins.inventory.**HostCategory**

Use this class when returning a list of host categories from the host_group_breakdown() method of a *gru.plugins.inventory.InventoryProvider*.

__init__(self, category, group, count=0):

category - a string specifying the category to breakdown by. e.g. "datacenter"

group - a string specifying the current group. e.g. "us-east-1"

count - an integer specifying the amount of hosts in the group. e.g. 0

---

class gru.plugins.inventory.**HostList**

This class represents a response to a search query or filter by an *gru.plugins.inventory.InventoryProvider*.

__init__(self, hosts=None, total_hosts=0):

hosts -a list of *gru.plugins.inventory.Host* objects.

total_hosts - integer. If the number passsed is bigger than len(hosts), a paginator will appear.

Authentication Backends

Writing an Authentication Backend requires subclassing *gru.plugins.auth.AuthenticationBackend*.

To use the plugin you developed, you'll need to make sure the module is loaded (specified under plugins.modules) and that authentication.backend points to your subclass' path.

class gru.plugins.auth.**AuthenticationBackend**

authenticate(self, username, password):

Given a username and a password, should return a *gru.plugins.auth.User* object if authentication is successful.

Otherwise, return None.

This is the only method an authentication backend has to implement.

member_of(self):

Will check which memebers the currently logged in user is a member of. If not logged in, will return an empty list.

is_logged_in(self):

Returns True if there's a user currently logged in.

login(self, user):
Stores the provided user object as part of the HTTP session

logout(self):
Clears the current session.

get_forbidden_url(self):
Returns a URL for a "Forbidden" page, in case a users tries a forbidden action

get_login_url(self, path):
Returns a URL for the Login form page. If `path` is provided, it will be passed to the login page, which will redirect to it upon successful login.

The User object

class `gru.plugins.auth.`**User**
Represents a logged in user. Has a username, real name, an optional list of groups and optional metadata

__init__(self, username, name, groups=None, user_data=None):
Create a new user object. It will be serialized and stored as part of the session.

`username` - The username used during authentication.

`name` - The user's full name.

`groups` - A list of strings describing the groups this user is a member of. Optional.

`user_data` - A dictionary of additional metadata we want to store about this user. Optional.

Configuration Reference

Setting up a settings file

GRU is configured using a YAML settings file. see the *Quick Start* guide to see how make sure GRU picks up and uses the settings file we'll be providing.

At the most basic level, we need to let GRU know of the following:

1. Which inventory provider we want to use. EC2 and ElasticSearch come out of the box, but you can *write your own*.
2. Which authentication backend you want to use. Dummy (no auth at all) and LDAP authentication are provided, but, again, feel free to *write your own*.
3. How you want to display your inventory: Which metadata should be visible and where
4. Which external plugins you want to load, including any configuration they may require

Below is the full list of configuration parameters that GRU supports, including all the builtin plugins. You can check out the `examples/` directory on Github for complete examples.

All basic configuration parameters

For brevity, nested keys will be denoted using a period (`.`) i.e.:

```
foo.bar = <some value>
```

Is equivalent to the following YAML structure:

```
---
foo:
  bar: <some value>
```

flask.debug

boolean whether to start the underlying flask server in debug mode or not. This will affect the logging verbosity

flask.secret_key

string used to encrypt signed cookies, used by the session system. Any random value will do.

flask.session_seconds

integer how long in seconds to keep user sessions.

Defaults to `604800` (7 days)

ui.name

string name to use in the navbar and titles.

Defaults to `GRU`.

ui.items_per_page

integer how many items to show on paginated pages.

Defaults to `50`.

ui.theme.name

string name of the Bootswatch theme to use.

Defaults to `darkly`.

ui.theme.version

string Bootswatch theme version number.

Defaults to `3.3.7`

plugins.directories

array of strings list of directories that contain GRU plugins.

Defaults to `['~/.gru-plugins', '/opt/gru-plugins']`

See *Loading plugins* for more info

plugins.modules

array of strings list of module names to import and search for plugins

See *Loading plugins* for more info

authentication.backend

string class name for the authentication backend to use.

Out of the box, you can use either `gru.contrib.auth.backends.LdapBackend` or `gru.contrib.auth.backends.DummyBackend`.

inventory.provider

string class name of the inventory provider to use.

Out of the box, you can use either `gru.contrib.inventory.providers.ElasticSearchProvider` or `gru.contrib.inventory.providers.EC2Provider`.

inventory.group_by

array of dictionaries

Describes the list of Host attributes to group the inventory by. This is used by the host breakdown screens. Common fields to group hosts by may include: data center (or AWS region), environment, role and operating system.

Each dictionary should provide at least the `field` key, which corresponds to a Host attribute name, and an optional `title` key which will be used by the UI to give this field a human readable name.

Example:

```
---
inventory:
  group_by:
    - field: os
      title: Operating System
    - field: dc
      title: Data Center
    - field: role
  ...
```

This will appear in the UI under the "Browse Inventory" drop-down.

inventory.host_display_name_field

string

A name for a Host attribute that will be used as the host's display name in various places in the UI.

Examples: `"hostname"`, `"instance-id"`

inventory.host_table_sort_by

string

A name for a Host attribute that will be used when sorting lists of hosts, on supporting Inventory Providers. Out of the box, this is currently only supported by the ElasticSearch provider.

Examples: `"hostname"`, `"instance-id"`

inventory.host_table_fields

array of dictionaries

Describe the list of Host attributes used when showing a table of multiple hosts. Common fields will generally include: role, data center, OS, IP address, # of cores, memory GBs, etc.

Each dictionary should provide at least the `field` key, which corresponds to a Host attribute name, and an optional `title` key which will be used by the UI to give this field a human readable name.

Example:

```
---
inventory:
  host_table_fields:
    - field: os
      title: Operating System
    - field: dc
      title: Data Center
    - field: role
    - field: num_cores
      title: "# Of Cores"
    - field: memory_gb
      title: Memory GB
    - field: ipaddress
      title: IP address
  ...
```

The field names may vary depending on your Inventory Provider.

inventory.host_info_fields

array of dictionaries

Describe the list of Host attributes used when a single host. Other fields will be hidden behind a "show more..." button. Common fields will generally include: role, data center, OS, IP address, # of cores, memory GBs, etc.

Each dictionary should provide at least the `field` key, which corresponds to a Host attribute name, and an optional `title` key which will be used by the UI to give this field a human readable name.

Example:

```
---
inventory:
  host_info_fields:
    - field: os
      title: Operating System
    - field: dc
      title: Data Center
    - field: role
```

```
    - field: num_cores
      title: "# Of Cores"
    - field: memory_gb
      title: Memory GB
    - field: ipaddress
      title: IP address
  ...
```

The field names may vary depending on your Inventory Provider.

LDAP authentication backend configuration parameters

Here's what an LDAP configuration might look like:

```
authentication:
  backend: gru.contrib.auth.backends.LdapBackend
  config:
    server: 10.10.10.10
    port: 3268
    bind_user: 'CN=gru_bind_user,OU=ops,DC=example,DC=com'
    bind_password: 'binduserpassword'
    user_query: '(sAMAccountName=$username)'
```

authentication.config.server

string

LDAP server IP or hostname to use

authentication.config.port

integer

LDAP server port to use

authentication.config.bind_user

string

LDAP user to bind with

authentication.config.bind_password

string

LDAP password to bind with

authentication.config.user_query

string

LDAP query to perform when searching the logging in user. You can use the `$username` interpolation token which will be replaced by the value provided by the logging in user.

Configuration Parameters for the built-in ElasticSearch inventory provider

`inventory.config.index`

string

The ElasticSearch index name to use

`inventory.config.hosts`

array of strings

A list of urls to use when connecting to the ElasticSearch cluster.

`inventory.config.timeout_seconds`

integer

Amount of time in seconds before timing out a request to an ElasticSearch query

Defaults to `30`.

Configuration Parameters for the built-in EC2 inventory provider

`inventory.config.connections`

array of dictionaries

Every connection to the EC2 API is represented by an entry in the connections array.

For each connection, specify the following:

`inventory.config.accounts` - **array** the AWS accounts we want to connect to.

`inventory.config.accounts.[].aws_access_key_id` - **string** *optional* the AWS access key ID to connect with. If omitted, it will be searched in your filesystem and environment variables in the order listed in boto's documentation

`inventory.config.accounts.[].aws_secret_access_key` - **string** *optional* the AWS secret access key to connect with. If omitted, it will be searched in your filesystem and environment variables in the order listed in boto's documentation

`inventory.config.accounts.[].regions` - **array of string** *optional* The AWS regions we wish to connect and pull inventory from. Example: `['us-east-1', 'us-west-2']`

Example:

```
---
provider: gru.contrib.inventory.providers.EC2Provider
config:
  accounts:
    - aws_access_key_id: AKXXXXXXXXXXXXXXXXXX
      aws_secret_access_key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      - regions: ['us-east-1', 'us-west-2']
```

Contributing

How to file an issue

To file an issue, please go to the Github Issues page.

Make sure you follow these guidelines:

1. **Give as much context as you can. Please describe:**
    - The flow of actions leading to the issue
    - The GRU version used
    - The Python version used
    - Docker version used, if applicable
    - Operating system and version
    - Links to all installed open source plugins
    - Your gru.yaml configuration file
2. Be polite. GRU is an open source project. People choose to contribute to it out of their desire to better our community and tools. Please treat them with respect and kindness.
3. Make sure your issue hasn't been filed yet. It is likely that someone else has already encountered the same problem as you. Search the open and recently closed issues for ones similar to yours.

How to contribute

GRU is an open source project and contributions are highly encouraged!

In order to work together productively, please follow these contribution guidelines:

1. Before submitting a pull request with a new feature, please open an issue, tagging it with `Feature Request`. This is a community driven project and there will likely be some discussion on what the optimal solution is. This discussion might affect how and whether this feature should exist.
2. Make sure your code submission respects the coding style of its surrounding code. As a rule of thumb for new files, make sure you respect PEP8.
3. If code changes any documented behaviour, make sure the documentation is updated as well. We will not merge pull requests that end up breaking current documentation.

Road Map

This is GRU's roadmap for the upcoming release:

0.2.0

- Release GRU as open source!
- **Improve the EC2 inventory provider**
    - Parallelize requests to different regions/accounts
    - Cache calls to AWS API for a configurable retention

  - Add proper pagination support to lists and search results
  - Document the search syntax better
- Improve documentation
- Add Unit tests for plugin loader and host pages
- TBD - we'd love your input.

Index

G

T