ARCHITECTURE OF THE KARANA BACK END

MICHAEL GÖTTE*

March 7, 2017

CONTENTS

ABSTRACT

Karana is a project with the aim of developing a cheap, open source monitoring framework. This document describes the design of the Karana Back End (KBE). The KBE is the IT backbone of the hardware monitoring device. The KBE combines a well established open source approach for time series based data collection and data visualization with a small scale management API framework and an automated deployment scheme. While there is plenty of information on the Internet how to combine InfluxDB with Grafana we focus in this documentation on the architecture of the management API and the automated deployment. The managment API is written in the Python framework Hug which is an API extension of the WSGI framework Falcon. The automated deployment is a collection of scripts written with the powerful automation tool Ansible.

* *MicroEnergy International, Berlin, Germany*

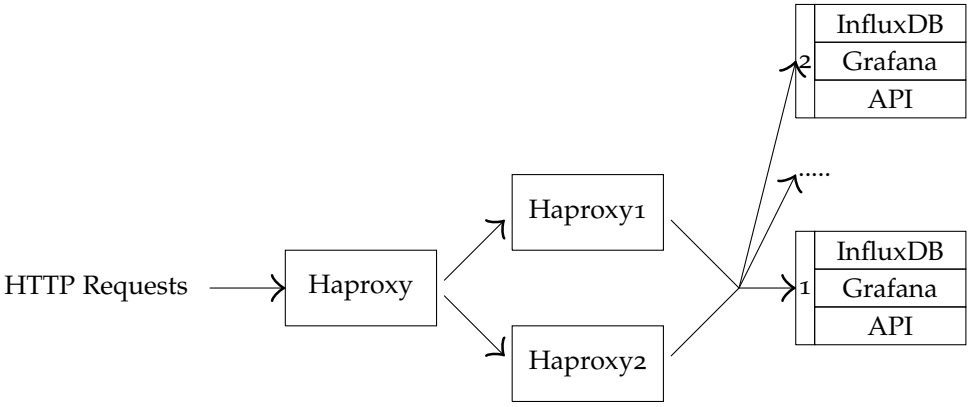

Figure 1: The Karana Back End IT infrastructure

1 INTRODUCTION

The KBE is an IT solution for monitoring data. The KBE provides an IT infrastructure suitable for multiple independent projects on one server, but can also be enrolled separately on different servers for each project.

The infrastructure relies on several virtual machines managed by LXD. All incoming requests to the server are directed to a load balancer which forwards the requests based on availability to one of two load balancer which distribute the requests to the correct instances of a project. The load balancing is realized with HAproxy. An instance is composed of three LXD containers holding an influxDB instance, a Grafana instance and an management API instance. Having a plain Linux (tested on Xenial, Ubuntu 16.04) server with only python installed (a dependency for Ansible) the Ansible script will deploy the IT infrastructure visualized in Figure 1 almost automatically.

The next sections will explain the architecture of the IT infrastructure and how to build it with Ansible. After that we will discuss the software design of the management API and how to use the API.

2 BUILDING THE INFRASTRUCTURE

2.1 LXD and Virtual Machines

2.2 HAproxy and Load Balancing

2.3 Achieving Higher Availability

3 CONFIGURING AN INSTANCE

4 MANAGMENT API

Simply spoken the management API enables maintaining the KBE without using command line tools. The API defines several end points which react on incoming HTTP requests. The body format of the HTTP requests (for GET, POST, PUT, PATCH, DELETE) is JSON. Subsection 4.2 will specify the behavior of these endpoint. These endpoints can be used to write front

ends for better user experience. The API is written in Python using the Hug framework.

For the rest of this Section we assume that the reader has cloned the git repository Git:KBE. This means that references to files are always with respect to the base folder of the project.

4.1 Configuration

Configuring the API is fairly simple if one has the credentials of the associated influxDB and Grafana instances. Ideally the configuration of the API is done by configuring the instance as described in Section 3. For completeness we will show how to configure the API by hand.

There are basically two files which need modification for configuration, the *config.ini* file which holds the information for the credentials of the influxDB and the Grafana instance and the *src/configuration.py* file which holds information on what models are used.

4.1.1 *The config.ini File*

An example *config.ini* looks like in Listing 1.

Listing 1: Example of a *config.ini* file

```
[influxdb]
host = 10.0.12.101
port = 8083
user = admin
pass = influxpw

[grafana]
host = 10.0.12.102
port = 3000
user = karanaadmin
pass = karanapasswort


[test]
host = localhost
port = 8000
user = test
pass = test
```

One needs to specify the host, the port, the user, and a password.

4.1.2 *The src/configuration.py File*

For an example of a *src/configuration.py* file see the project repository. In general there are two sections, a resource definition section and a tenant definition section. The resource definition section defines which resources should be used. There should be at least the resource user and karana, since these are the two core resources. For more complex projects one can also choose other resources related to technical maintenance of karana or pay as you go. A resource definition section for one resource looks like in Listing 2.

key	description	note
RES_TABLE_ID	the unique id of a resource description	this is an artifact and not used in the code
NAME	the name of a resource	resources must be plural
UNIQUE _SCHEMA_FIELDS	a list of resource attributes which need to be unique	needs to be a list
CREDENTIALS_LOGIN _FIELD	the resource attribute used for login in	is NONE if resource is not allowed to login, needs to be in UNIQUE_SCHEMA_FIELDS

Table 1: Keys of a resource definition

Listing 2: Example for a resource definiton in the *src/configuration.py* file, here for the user resource

```
users =
{
  'metadata':
  {
    'res_table_id': 1,\
    'schema':
    {
      'entry_create_schema': "UserSchema",\
      'entry_import_schema': "UserDbSchema"
        },\
    'name': "users",\
    'unique_schema_fields': ['uuid', 'email'],\
    'credentials_login_field': 'email',\
  }\
}
```

A resource definition is a python dictionary, holding one key, METADATA. The description of each of the subkeys is stored in Table 1 and in the list on SCHEMA below. The SCHEMA divides into two further keys. Since in the configuration file only meta information is stored one needs to link the resource to the actual model. How to write a model for a resource will be explained later in Subsection 4.3. Some models are already written, for example the user model. The SCHEMA tells the API which models to used. The models are defined explicitly in the file *src/schema.py*.

- ENTRY_CREATE_SCHEMA: This is the scheme for creation. During creation some attributes are set by the creator (e.g. for user: email, name) and some are created by the system (e.g. for user: uuid, created_at). So only the presence of the non system created attributes is checked, i.e. is an email an email, is the length of a password sufficient etc.

- ENTRY_IMPORT_SCHEMA: This is the scheme for import. When importing a resource to make it accessible it is checked if all entries are present and have the defined properties, e.g. length, is email etc. This is used always when the API is started and is especially important when migrating one data set to another API instance.

An example for the tenant definition section is Listing 3

Listing 3: Example for a tenant definiton in the *src/configuration.py* file

```
api_metadata =
{
  'tenant_id': 'SMNTYQIUB4YTC',
  'tenant_customer_name': 'Bintumani e.V.',
  'tenant_login_credentials_resource': 'users',
  'tenant_login_credentials_field': 'credentials',
  'tenant_used_login_credentials': ['login', 'pwhash'],
  'logging_config_file': 'src/logging.yaml',
  'db_dump':
  {
    'table_db_path': "storage/table_db.json",\
    'table_meta_db_path': "storage/tablestate_db.json",
    'table_db_folder': "storage/",
  }
}
```

4.2 REST Endpoints: How to use them?

This subsection deals with the endpoint description. The basic concept is generic. So for each resource there are the same endpoints present. The endpoints are written with Hug as mentioned before and are defined in the file *src/main.py*. This file is also the file which is called when starting the API. **Note:** one important advantage of using Hug is having the possibility of serving different endpoints. An endpoint version is defined inside the Hug decorator (decorators are Python specific and start with an @).

4.2.1 *Resource Endpoints*

There are five standard endpoints for each resource. They are described in Table 2. They can be used to get, create change, and delete resources. If you send a body the body needs to be a JSON with one key DATA holding a string which is a JSON. For example using the HTTP library httpie:

```
http post http://localhost:8000/v1/users/new
data='{"name":"Micha","role":"admin","email":"micha@all.de",
"credentials":{"login":"micha@all.de","password":"12345"}}'
```

4.2.2 *Java Web Tokens and Authentication*

The authentication and authorization is managed through JWT. If a loginable resource (this is a resource having credentials and a role like the user resource) once to login it sends its credentials to the appropriate login endpoint (see also Table 2). In return it gets a JSON web token (JWT) holding decrypted information on the creation of the token, which resource id it belongs to, and which role this resource has. This token needs to be send as a bearer in the authorization header of each HTTP request. The Hug framework provides middleware for authentication which decodes the token and confirms that it is valid to use the requested endpoints. The role can limit the number of endpoints, e.g. someone who can see the users may not be allowed to modify them. An example request looks like this:

HTTP Verb	Endpoint	Description
GET	v{NR}/{RESNAME}/{RESID}	if RESID is not present it returns all resource if it is present it returns this resource
POST	v{NR}/{RESNAME}/NEW	this creates a new resource
POST	v{NR}/{RESNAME}/LOGIN	this generates a token to use the endpoints see also Subsection 4.2.2
PUT	v{NR}/{RESNAME}/{RESID}	this updates a resource needs at least two changes with given id
PATCH	v{NR}/{RESNAME}/{RESID}	this modifies a resource only one change with given id
DELETE	v{NR}/{RESNAME}/{RESID}	this deletes a resource with given id

Table 2: Endpoint description for a resource

```
http POST http://localhost:8000/v1/users/login user=micha@ex.de
password=Mypassword
```

An response would look like this:

```
HTTP/1.0 200 OK
Date: Thu, 02 Mar 2017 16:54:24 GMT
Server: WSGIServer/0.2 CPython/3.5.2
access-control-allow-headers: content-type
access-control-allow-origin: http://localhost:3000
content-length: 185
content-type: application/json

"{
\"results\": [
{\"token\":
\"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJsb2dpbm5hbWUiOiJtaWNoYUBle
C5kZSIsImRhdGEiOnsicm9sZSI6ImFkbWluIn19.
Fx9txrY2nBXOKG7BVTYIWEpW2nrPhmjEEu8UUC4TVGE\"}]
}"
```

REFERENCES

[1] A. J. Figueredo and P. S. A. Wolf. Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330, 2009.