ARCHITECTURE OF THE KARANA BACK END

MICHAEL GÖTTE*

May 10, 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*

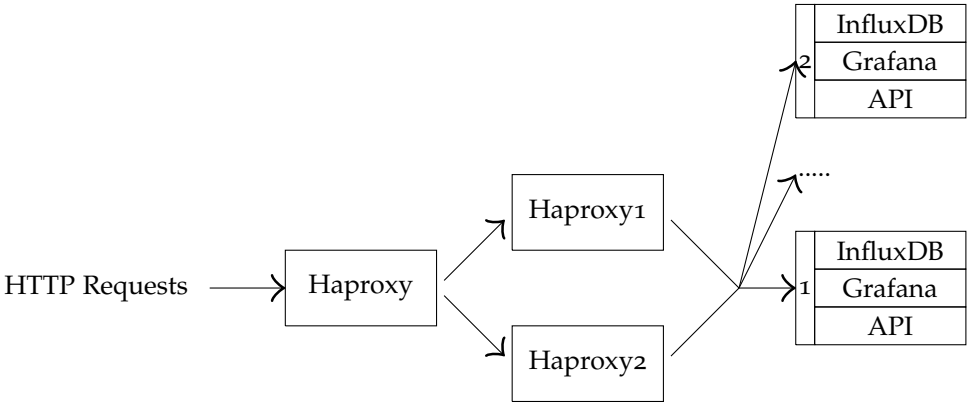

Figure 1: The Karana Back End IT infrastructure

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. Those doesn't have to be on the same LXD host. 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

This section introduces the underlying IT infrastructure. It first gives some comments on the used components LXD and HAproxy and then describes how to set it up running Ansible scripts. The script for that can be found on our Gitlab page (Projectname: backend-deploy).
Note: All the paths in here are relative to the base folder of this gitlab project.

2.1 Description of the Infrastructure

2.1.1 *LXD and Virtual Machines*

LXD was chosen as "Virtualization Platform", because it has a very small amount of infrastructure needed for its usage. Any other container solution (Kubernetes, OpenVZ, Docker...) and even full virtualization solutions (KVM, HyperV, Xen...) can be used as well. But LXD can be used even within small vServer instances and are therefore a good choice in smaller

setups. This setup leads to a lot of containers (at least three for every customer), which helps later to balance them between different hosts and decouple the systems of the customers.

2.1.2 *HAproxy and Load Balancing*

HAproxy works as the guard for the IT infrastructure. Every HTTP request made to the server is processed by HAproxy and deligated to the appropriate lxd container in the virtual network. Since a malformed configuration of HAproxy may stop the HAproxy-service, the first outer HAproxy is configured with static simple configuration which sends the traffic to at least two different HAproxy behind it. Those second level Balancers can be taken out of production and updated with a new configuration. This method allows on the fly configuration changes without affecting the running production setup. This second level balancers do the routing to the right containers and TLS termination. Later those balancing services may run even on different hosts, to assure a higher availability.

2.1.3 *Achieving Higher Availability*

Even if our single services (Influxdb, KBE and Grafana) are not setup with an HA-schema, but as single instances, a monitoring of the instances may create „new“ containers with restored backups on the fly. Our Karana data collection devices do have mostly bad internet connection, so they need to be resilient on this aspect anyway. Which meeans that even downtimes of several minutes should not affect the system.

2.2 Build the infrastructure with Ansible scripts

To roll out the backend one only needs to configure Ansible properly, install python 2 on the target server and run the Ansible scripts *playbook/lxd.yml*. and *playbook/haproxy.yml*

2.2.1 *Configure Ansible*

The following steps need to be done to configure Ansible. You need a local Linux computer with Python Pip installed and a (target) server with a domain name (tested with Ubuntu Xenial on the target server).

1 Install Ansible, e.g. with pip

2 Create your ansible hosts file, see also *playbook/hosts* and Listing 1 where you put in the server of your domain name

3 Copy the file to the right location

```
$ cp playbook/hosts /etc/ansible/hosts
```

4 Ansible works on SSH, so you need to have SSH access to your target server, please configure SSH such that you can use it through authorized keys, to tell Ansible how this works you need to configure the file *playbook/ssh.cfg* and put in your server domain name, see also Listing 2

5 install Python 2 on your target server

Listing 1: Example of Ansible hosts file

```
[example]
<your-server-domain-name>
```

Listing 2: Example of Ansible ssh.cfg file

```
Host 10.0.12.*
StrictHostKeyChecking no
ProxyCommand ssh -o "StrictHostKeyChecking=no" root@<your-server-domain-name>


Host <your-server-domain-name>
ForwardAgent yes
ControlPath ~/.ssh/cm-%r@%h:%p
ControlMaster auto
ControlPersist 10m
StrictHostKeyChecking no
```

The configuration of *playbook/ssh.cfg* makes your target server an SSH bastion. Since we will build the virtual network in the IP space 10.0.12.* you can then directly use SSH to go to the containers on your target server.

2.2.2 *Run the script to set up the lxd network*

Then you can run in the folder *playbook/* the following command

```
$ ansible-playbool lxd.yml -u root
```

The execution of the command takes a while but if everything goes well the script does the following:

- creates an ssh key on the target server
- installs lxd, ipython and python pip on the target server
- initializes lxd on the target server
- gets an image for the containers on the target server
- creates a virtual network on the target server
- creates for lxd containers on the target server, namely three haproxy instances and one default nginx instance
- adds the four containers to the virtual network and makes them ssh accessible.

2.2.3 *Run the script to set up the HAproxys*

The three haproxy instances need to be configured. For that there are configuration files in *files/standard_files*. The setup is as described above in Subsection 2.1.2. The command

```
$ ansible-playbool haproxy.yml -u root
```

when executed

- installs python on the four lxd containers
- installs haproxy on the three haproxy containers

- pushes the configuration files *files/standard_files/haproxy.cfg* on the central haproxy and *files/standard_files/haproxy.dist.cfg* on the two parallel haproxys
- installs nginx on the fourth container
- configures the iptables on the target server with the file *files/standard_files/rules.v4*, this sends all incoming requests on the ports 80, 222 and 443 to the central haproxy container

3 CONFIGURING AN INSTANCE

Having everything setup as in Section 2 to create an instance for a project is now fairly simple. **Note (again):** All the paths in here are relative to the base folder of this gitlab project backend-deploy as in Section 2.

3.1 Configuration

The file *playbook/tenant_var.yml* holds the variables which will be used during the execution of the Ansible scripts. An example file looks like in Listings 3.

Listing 3: Example of Ansible hosts file

```
pname: <name>
influxip: 10.0.12.101
grafanaip: 10.0.12.102
apiip: 10.0.12.103
domain: <your-server-domain-name>
path_b: ../../backend-api
path_f_r: ../../frontend-react/karana-app-test
influx_conf: influxdb_1_2.conf
```

You can give the project a name and enter your target server domain. The IP addresses are addresses in the virtual network on your target server. Make sure that they are still free. The path variables should point to the KBE base folder and to the frontend of choice. Furthermore, influxDB changes the configuration files from version to version. If the file changed again you can put in *files/templates/* and change the reference in the configuration file.

3.2 Create an Instance

Now in the folder *playbook/* run the command

```
$ ansible-playbool instance.yml -u root
```

This

- creates three influx containers (one for grafana, one for the api, one for influxDB)
- adds them to the virtual network on the target server with the defined IP addresses in the file *playbook/tenant_var.yml*
- pushes the authorized keys to the containers for remote access

- installs python and ssh on the containers

So this scripts creates the raw containers to put in the three different instances.

3.3 Setup Influx Container

Now run the command

```
$ ansible-playbool influx.yml -u root
```

This

- install influx in the container with the influx IP
- adds the instance to the two parallel HAproxys by turning the first one off and then turning the second one off, the registered containers are stored in a tinyDB file (you might need to install tinyDB with pip) holding all containers, from the tinyDB entries the template *playbook/templates/haproxy.dist.cfg* is configured using Jinja2 syntax
- creates and admin user on the influxDB instance and writes the credentials in the file *files/templates/<projectname>_config.ini*
-

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.
Note: The *config.ini* file can be configured with ansible scripts if needed.

4.1.1 *The config.ini File*

An example *config.ini* looks like in Listing 4.

Listing 4: 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. If one uses the IT infrastructure described in Section 2 it might be necessary to also specify the external host. The IP's used here are the once of the internal LXD network which resides on 10.0.12.*.

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 (PAYG). A resource definition section for one resource looks like in Listing 5.

Listing 5: 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',\
  }\
}
```

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

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 schema 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 schema 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 6.

Listing 6: 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/",
  }
}
```

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

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 POST http://localhost:8000/v1/users/login
user=admin@example.com password=admin
```

Note: In test mode the API creates an admin user with the above credentials. 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\"}]
}"
```

To use this to do a success full request one needs to take that token and put into the Authorization header of the HTTP request, for example:

```
http GET http://localhost:8000/v1/users/ Authorization:"bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJsb2dpbm5hbWUiOiJtaWNoYUBle
C5kZSIsImRhdGEiOnsicm9sZSI6ImFkbWluIn19.
Fx9txrY2nBXOKG7BVTYIWEpW2nrPhmjEEu8UUC4TVGE"
```

With response:

HTTP Verb	Endpoint	Description
POST	v{NR}/SYNC/ALL/DB	triggers the syncing process with the other instance

Table 3: Endpoint for syncing with the db


```
HTTP/1.0 200 OK
Date: Thu, 02 Mar 2017 16:54:46 GMT
Server: WSGIServer/0.2 CPython/3.5.2
access-control-allow-headers: content-type
access-control-allow-origin: http://localhost:3000
content-length: 472
content-type: application/json

"{\"results\": [
{\"4e1fcc06-7ccc-4b81-be93-e7bae73daf2a\":
{\"credentials\":
{\"password\":
\"$6$tZPL4iZZV5Idmnfy$jACMfQGZLun176F.
LOaV46g.ea4YI2LWkl.uOlvJ.LhY62cqLbQE9zp7CmNMXltImo/
qbLrzIUTd/FeppRENd1\",
\"login\": \"micha@ex.de\"},
\"uuid\": \"4e1fcc06-7ccc-4b81-be93-e7bae73daf2a\",
\"created_at\": \"2017-03-02T15:26:11.767166+00:00\",
\"email\": \"micha@ex.de\",
\"password_influx\": \"UUTQAMMMLZoZ3RTK\",
\"role\": \"admin\",
\"karanas\": [],
\"name\": \"Micha\"
}}]}"
```


4.2.3 *Syncing the API*

There is one more endpoint this endpoint is triggered when sending a HTTP Post request to the endpoint, the endpoint is described in Table 3. Each time a new resource is created, updated or modified it is noted as being not in sync. When sending a POST request against the endpoint in Table 3 each resource which is not in sync is synchronized. Resources that need syncing are karanas and users. The users and karanas are synced with the Grafana and InfluxdDB instance associated with the API. This enables karanas to be configured and send data into InfluxDB. ANd it enables to visualize data with Grafana by connecting the karana with the Grafana instance.

4.3 Marshmallow and Model Creation

Marshmallow is a helpful tool to validate data models. Each resource (e.g. user, karana) has certain properties. For example a user has an id, a name, an email, credentials etc. and a karana has an id, a name, a configuration, an owner etc. These properties have certain dimensions, for example a password has a minimal length, an email a certain format, the id is of type uuid version 4. These dimension need to be checked to prevent the database from being corrupted, for this validation we use Marshmallow twice. Once

for checking the properties when a resource is created and once when a resource is imported. The difference between these two processes is that during creation an uuid is generated while during import an uuid needs to be already present. The models are defined in *src/schema.py*. An example is Listing 7. The ResourceSchema is meant for creation and the ResourceDBSchema is meant for import, as can be seen by the distribution of the required attribute.

Listing 7: Example for a marshmallow scheme, here user

```
class UserSchema(Schema):
  uuid = fields.UUID(dump_only=True)
  name = fields.Str(required=True)
  email = fields.Email(required=True)
  password_influx = fields.Str()
  credentials = fields.Nested(CredentialsSchema, many=False)
  created_at = fields.DateTime(dump_only=True)
  role = fields.Str()
  karanas = fields.List(fields.UUID(), validate=validate.Length(max=1000))

  @post_load
  def make_user(self, data):
    return User(**data)


class UserDbSchema(Schema):
  uuid = fields.UUID(required=True, dump_only=True)
  name = fields.Str(required=True)
  email = fields.Email(required=True)
  password_influx = fields.Str(required=True)
  credentials = fields.Nested(CredentialsSchema, required=True, many=False)
  created_at = fields.DateTime(required=True, dump_only=True)
  role = fields.Str(required=True)
  karanas = fields.List(fields.UUID(), validate=validate.Length(max=1000),
  required=True)
```

Each schema also has a model class associated with it. When creating a resource this is called through the post_load decorator. Such a user class is shown in Listing 8. As one can see on initialization a date and an uuid version for are created. Furthermore, a random string generator is called to generate a password. When a resource is created a Python dictionary of the resource is created and stored. Note that in this example credentials is also a scheme having a model class. This can be used to nest schemes.

Listing 8: Example for a marshmallow scheme, here user

```
class User(object):
  def __init__(self, name, email, credentials, role='client'):
    self.uuid = uuid.uuid4()
    self.name = name
    self.credentials = Credential(**credentials)
    self.email = email
    self.password_influx = id_generator()
    self.created_at = dt.datetime.now()
    self.role = role
    self.karanas = []
```

Key	Description
METADATA	this is right now an artifact
TABLES	holds the actual data of the resources
TABLES_META	holds the metadata of the resources as specified in *src/configuration.py*
UUID_INDEX	this is right now an artifact
UNIQUENESS_INDEX	this holds for each unique field in the resource definition in *src/configuration.py* defined in UNIQUE_SCHEMA_FIELDS, this helps to access resource tables through there unique fields
SYNC_STATE	this holds a list of each resource with a boolean indicating if the resource is in sync with the InfluxDB and Grafana instance, see also Subsection 4.2.3
CREDENTIALS_INDEX	this is right now an artifact

Table 4: Description of the main state of the in memory database of the API

If one wants to add a new model one need to write three class, two marshmallow schemes for creation and import and one model for actually getting a python object. Once one has done that one needs to add the resource to the configuration file as described in Subsection 4.1.2. When the API is started one then can use the endpoints described in Subsection 4.2.

4.4 Database: JSON Dumps and GitPython

The database is always in memory while the API is running. This design is chosen due to the multi tenant concept. This means since each project gets its own infrastructure the total size of the database will not be big. If one day this becomes an issue we need to replace this approach by a postgreSQL solution. Every time the database is changed it is written to the hard disc (the location is specified in Listing 6 by the keyword TABLE_DB_FOLDER, see Subsection 4.1.2). When written to the hard disc the Python library GitPython is used to stage and commit the changes and if wanted pushed to a remote. This is automatically the backup system of the database.

The file handling all of this is *src/db.py*.

- What will be dumped?
  We dump all resources and the associated configuration defining the resources.
- When do we read from the database?
  Only when the API is started the dumped file is read into memory. When this is done a validation check see Subsection 4.3 is performed to check if data is corrupted. This means especially that reading in production is looking up in a Python directory.
- How does the state in memory looks like? The main state is an attribute of the class KARANADBWRAPPER in *src/db.py* and called MAIN_STATE. There are several keys in the main state. This is described in Table 4. Only the values behind the keys TABLES and TABLES_META are written to disc and versionized.

4.5 InfluxDBWrapper and GrafanaWrapper: Configuring Remotely

The syncing process relies on communication with the Grafana and the InfluxDB instance configured as described in Subsection 4.1.2. For that we wrote two Python wrappers which are used in the syncing process described in Subsection 4.2.3. These to wrappers rely on the standard Python library for sending HTTP requests. Building the HTTP request is taken care of and the wrapper offer several functions to simplify communication with the other instances. The wrappers are stored in

- GrafanaWrapper: *src/grafana.py*
- InfluxDBWrapper: *src/influx.py*, including the HTTP preparation class DBHTTPSetup

These wrappers are used to perform idempotent syncing through the class SynchInflux in *src/synch.py*. This class offers methods which check if a certain action needs to be performs and it offers methods which perform the action if the check fails.

4.6 Py.test and Test Driven Development

The folder *tests/* holds several tests which can used by the Test Driven Development framework Py.Test. An example call could be

```
py.test-3.5 -vvvs tests/
```

The tests include testing the wrappers from Subsection 4.5 and tests the Authentication from Subsection 4.2.2 and the Endpoints for the resources user and karana described in Subsection 4.2.

4.7 JournalCTL and logging

The file *src/logger.py* provides logging to log files but also to journalCTL. How to log including verbosity (INFO, DEBUG, ERRORS etc.) can be defined through the YAML file src/logging.yaml. There is a template called src/logging.yaml.sample. If no specification is given there will only be simple logging to the system logs.

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.