SiSock Documentation

Release 0.2.13+1.g8bd5ec6.dirty

Simons Collaboration

Nov 14, 2019

Contents

sisock is a python library and collection of components for serving quicklook data over websockets, designed initially for the Simons Observatory. The goal is to define an API for implementing a `DataNodeServer`, which returns the desired data, whether from memory or from disk. Data is passed through a crossbar server using the WebSockets protocol.

sisock plays a key role in the Simons Observatory housekeeping live monitor, which is based on the open source analytics and monitoring platform Grafana. A simple webserver sits between sisock and Grafana, allowing Grafana to query the sisock DataNodeServers.

CHAPTER 1

User's Guide

Start here for information about the design and use of sisock.

1.1 Live Monitoring Setup Guide

This guide will walk you through the complete setup and basic operation of the Simons Observatory live housekeeping monitor. By the end of this guide you will be able to command your hardware systems using OCS, record their data to disk in the so3g file format, and monitor their output in real time using a Grafana dashboard.

1.1.1 Dependencies

The system is designed to be distributed across many computers, however it can also all be run on a single computer. This modular architecture means the software and hardware requirements are quite flexible. Below are the minimum hardware and software requirements for getting the live monitor running on a single computer.

Software Requirements

Installation will be covered in the next section, but these are the required software dependencies:

- Docker - Containerization software used for deploying several SO written packages.
- Docker Compose - CLI tool for running multi-container Docker applications.
- OCS - The observatory control system, for running clients locally.

Hardware Requirements

You will need a Linux computer running Ubuntu 18.04. Other Operating Systems can be used, but will not be supported.

Note: Docker stores its images in the root filesystem by default. If the computer you are using has a small / partition you might run into space constraints. In this case you should get in touch with Brian for advice on how best to proceed.

Networking Requirements

This Linux machine will need to go on the same network as whatever hardware you're controlling with OCS. Live monitoring remotely (i.e. not sitting directly at the computer) is facilitated if your IT department allows it to have a public IP address, and if you are able to setup a secure webserver. Doing so, however, is beyond the scope of this guide.

Warning: If you do have a public IP and traffic is allowed to all ports, you are strongly recommended to enable a firewall as described in *Configuring a Firewall*. Care should also be taken, when exposing ports in Docker, to expose your services, especially the crossbar server, to only your localhost (i.e. 127.0.0.1). This is the default in all templates provided by the SO DAQ group.

Note: If you do not have a public IP, but do have access to a gateway to your private network, then port forwarding can be used to view the live monitor remotely, as described in *Port Forwarding to View Remotely*.

1.1.2 Software Installation

This page provides brief instructions, or links to external resources where appropriate, for installation of software related to the live monitor.

Installing Docker

Docker is used to run many of the components in the live monitor. While the system can be run without Docker, it is the recommended deployment option. To install, please follow the installation documentation on the Docker website.

Note: The docker daemon requires root privileges. We recommend you run using sudo.

Warning: While it is possible to run docker commands from a user in the `docker` group, users in this group are considered equivalent to the `root` user.

When complete, the docker daemon should be running, you can check this by running `sudo systemctl status docker` and looking for output similar to the following:

```
$ sudo systemctl status docker
 docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; disabled; vendor preset:␣
↪enabled)
   Active: active (running) since Tue 2018-10-30 10:57:48 EDT; 2 days ago
     Docs: https://docs.docker.com
 Main PID: 1472 (dockerd)
```

If you see it is not active, run `sudo systemctl start docker`. To ensure it runs after a computer reboot you should also run `sudo systemctl enable docker`.

Installing Docker Compose

Docker Compose facilitates running multi-container applications. This will allow us to pull and run all the containers we need in a single command. To install see the Docker Compose documentation.

When complete you should be able to run:

```
$ docker-compose --version
docker-compose version 1.22.0, build 1719ceb
```

Note: The version shown here might not reflect the latest version available.

Installing OCS

Install OCS with the following:

```
$ git clone https://github.com/simonsobs/ocs.git
$ cd ocs/
$ pip3 install -r requirements.txt
$ python3 setup.py install
```

Note: If you want to install locally, not globally, throw the *–user* flag on both the pip3 and setup.py commands.

Warning: The master branch is not guaranteed to be stable, you might want to checkout a particular version tag before installation depending on which other software you are working with. See the latest tags.

These directions are presented in the OCS repo, which likely has the most up to date version. If you need to update OCS, be sure to stash any changes you've made before pulling updates from the repo.

1.1.3 Environment Setup

All the required software should now be installed. The next step is to properly configure the environment. Scripts to help with parts of this setup are provided in the ocs-site-config repository. In this repository is a directory for each SO site, currently this means one for each test institution (i.e. yale, penn, ucsd). Start by cloning this repository, and if your site does not have a directory, copy the templates directory to create one.:

```
$ git clone https://github.com/simonsobs/ocs-site-configs.git
$ cp -r templates/ yale/
```

Setup Scripts

There are many steps to perform in setting up a new system. In an attempt to streamline these setup steps we have provided several setup scripts. These need to each be run once on your system. In the future they will likely be combined into a single script, but for now we deal with the individual parts.

TLS Certificate Generation

The crossbar server can handle secure connections using TLS certificates. The live monitor uses this secure connection capability, and as a result we need to generate a set of self-signed TLS certificates. To do this we just need to run the `setup_tls.py` script. Simply enter your new directory and run it (swap `yale` for your institution):

```
$ cd yale/
$ ./setup_tls.py
```

This will generate the required certificates and put them in a directory called `.crossbar/` (which already existed in the copied template directory).

Warning: Make sure your `.crossbar/config.json` file exists. Missing the dot directory when copying files from the template is a common mistake. A missing crossbar configuration will cause the entire system not to work.

Docker Environment Setup

If this is your first time using Docker then we need to do some first time setup. In the site-config `templates/` directory (and thus in your copy of it for your institution) there should be a script called `init-docker-env.sh`. Running this creates a storage volume for Grafana so that any configuration we do survives when we remove the container. To setup the Docker environment run the script:

```
$ sudo ./init-docker-env.sh
```

Manual Setup Steps

These steps haven't been included in any scripts yet, and must be performed manually. These only need to be performed once per system.

OCS User/Group and Data Directory Creation

The OCS aggregator agent runs as a user called *ocs*, with a UID of 9000. We will setup the same *ocs* user on the host system, as well as an *ocs* group. The data written by the aggregator will belong to this user and group:

```
$ groupadd -g 9000 ocs
$ useradd -u 9000 -g 9000 ocs
```

Next we need to create the data directory which the aggregator will write files to. This can be any directory, however we suggest using `/data`, and will use this in our example:

```
$ mkdir /data
$ chown 9000:9000 /data
```

Finally, we should add the current user account to the *ocs* group, replace *user* with your current user:

```
$ sudo usermod -a -G ocs user
```

OCS Config Setup

The OCS configuration file is named after a given site, i.e. `yale.yaml`. In order for OCS to know where to find your configuration file we need to do two things.

First, add the following to your `.bashrc` file:

```
export OCS_CONFIG_DIR='/path/to/ocs-site-configs/<your-institution-directory>/'
```

Next, within your site config directory, symlink your configuration file to `default.yaml`:

```
$ ln -s yale.yaml default.yaml
```

Note: If you're proceeding in the same terminal don't forget to source your `.bashrc` file.

Login to Docker Registry

The Docker images which we will need to run the live monitor are hosted on a private Docker registry at Yale. Until things are hosted publicly we need to login to the private. (The password can be found on the SO wiki.) To do so run:

```
$ sudo docker login grumpy.physics.yale.edu
Username: simonsobs
Password:
```

You will see output along the lines of:

```
WARNING! Your password will be stored unencrypted in /home/user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

You will now be able to pull images from the registry.

The system is now ready to configure. In the next section we will discuss both the *docker-compose* and *ocs* configuration files.

1.1.4 Configuration

Next we need to configure both the *docker-compose* and *ocs* environments, each with their own configuration files. These files will differ for each site depending on your hardware setup. Below we cover a simple configuration for each. Later we discuss more advanced configuration.

docker-compose.yaml

Docker is used extensively in deploying several parts of the live monitor. Docker Compose is used to manage all the containers required to run the live monitor software. The Docker Compose configuration file defines the containers that we will control.

The ocs site configs templates provide a simple template *docker-compose.yaml* file which configures some of the essential container.

Note: The filename is important here, as the *docker-compose.yaml* path is the default one parsed by the docker-compose tool. A configuration file can be specified with the *-f* flag.

Note: If you are interested in developing containers related to sisock, you might be interested in checking out the "dev-mode" template.

The template configuration does not contain all available containers. Details about more containers can either be found in the sisock documentation or in the socs and ocs documentation.

The template *docker-compose.yaml* file, looks something like this:

```
version: '2'
volumes:
  grafana-storage:
    external: true
services:
  # ------------------------------------------------------------------------
  # Grafana for the live monitor.
  # ------------------------------------------------------------------------
  grafana:
    image: grafana/grafana:5.4.0
    restart: always
    ports:
      - "127.0.0.1:3000:3000"
    environment:
      - GF_INSTALL_PLUGINS=grafana-simple-json-datasource, natel-plotly-panel
    volumes:
      - grafana-storage:/var/lib/grafana

  # ------------------------------------------------------------------------
  # sisock Components
  # ------------------------------------------------------------------------
  sisock-crossbar:
    image: grumpy.physics.yale.edu/sisock-crossbar:latest
    ports:
      - "127.0.0.1:8001:8001" # expose for OCS
    volumes:
      - ./.crossbar:/app/.crossbar
    environment:
        - PYTHONUNBUFFERED=1

  sisock-http:
    image: grumpy.physics.yale.edu/sisock-http:latest
    depends_on:
      - "sisock-crossbar"
    volumes:
      - ./.crossbar:/app/.crossbar:ro

  weather:
    image: grumpy.physics.yale.edu/sisock-weather-server:latest
    depends_on:
      - "sisock-crossbar"
      - "sisock-http"
    volumes:
```

(continues on next page)

(continued from previous page)

```
    - ./.crossbar:/app/.crossbar:ro

# ----------------------------------------------------------------------
# sisock Data Servers
# ----------------------------------------------------------------------
LSA23JD:
  image: grumpy.physics.yale.edu/sisock-thermometry-server:latest
  environment:
      TARGET: LSA23JD # match to instance-id of agent to monitor
      NAME: 'LSA23JD' # will appear in sisock a front of field name
      DESCRIPTION: "LS372 in the Bluefors control cabinet."
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"

# ----------------------------------------------------------------------
# OCS Agents
# ----------------------------------------------------------------------
ocs-registry:
  image: grumpy.physics.yale.edu/ocs-registry-agent:latest
  hostname: ocs-docker
  volumes:
    - ${OCS_CONFIG_DIR}:/config:ro
  depends_on:
    - "sisock-crossbar"

ocs-aggregator:
  image: grumpy.physics.yale.edu/ocs-aggregator-agent:latest
  hostname: ocs-docker
  user: "9000"
  volumes:
    - ${OCS_CONFIG_DIR}:/config:ro
    - "/data:/data"
  depends_on:
    - "sisock-crossbar"
```

Warning: Bind mounts are a system unique property. This is especially true for ones which use absolute paths. If they exist in any reference configuration file, they will need to be updated for your system.

Understanding what is going on in this configuration file is key to getting a system that is working smoothly. The Docker Compose reference explains the format of the file, for details on syntax you are encouraged to check the official documentation.

In the remainder of this section we will go over our example. We first define the use of an external docker volume, `grafana-storage`, which we created using the `init-docker-env.sh` script.

Every block below `services:` defines a Docker container. Let's look at one example container configuration. This example does not represent something we would want to actually use, but contains configuration lines relevant to many other container configurations:

```
g3-reader:
  image: grumpy.physics.yale.edu/sisock-g3-reader-server:latest
  restart: always
  hostname: ocs-docker
  user: "9000"
```

(continues on next page)

(continued from previous page)

```
  ports:
    - "127.0.0.1:8001:8001" # expose for OCS
  volumes:
    - /data:/data:ro
    - ./.crossbar:/app/.crossbar
  environment:
      MAX_POINTS: 1000
      SQL_HOST: "database"
      SQL_USER: "development"
      SQL_PASSWD: "development"
      SQL_DB: "files"
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
    - "database"
```

The top line, `g3-reader`, defines the name of the service to docker-compose. These must be unique. `image` defines the docker image used for the container. A container can be thought of as a copy of an image. The container is what actually runs when you startup your docker service. `restart` allows you to define when a container can be automatically restarted, in this instance, always. `hostname` defines the hostname internal to the container. This is used in the OCS container configurations in conjunction with the ocs-site-configs file. `user` defines the user used inside the container. This is only used on the aggregator agent configuration.

`ports` defines the ports exposed from the container to the host. This is used on containers like the crossbar container and the grafana container. `volumes` defines mounted docker volumes and bind mounts to the host system. The syntax here is `/host/system/path:/container/system/path`. Alternatively the host system path can be a named docker container, like the one used for grafana. `environment` defines environment variables inside the container. This is used for configuring behaviors inside the containers. `depends_on` means Docker Compose will wait for the listed containers to start before starting this container. This does not mean the services will be ready, but the container will be started.

Note: Environment variables can be used within a docker-compose configuration file. This is done for the *OCS_CONFIG_DIR* mount for the OCS agents in the default template. For more information see the docker compose documentation.

For more details on configurations for individual containers, see the service documentation pages, for instance in the sisock documentation or in the respective ocs agent pages.

OCS

OCS has a separate configuration file which defines connection parameters for the crossbar server, as well as the Agents that will run on each host, whether they are on the host system, or in a Docker container. This configuration file allows default startup parameters to be defined for each Agent.

We will look at a simple example and describe how deploying Agents in containers should be handled. For more details on the OCS site configuration file see OCS Site Configuration. Here is an example config:

```
# Site configuration for a fake observatory.
hub:

  wamp_server: ws://localhost:8001/ws
  wamp_http: http://localhost:8001/call
  wamp_realm: test_realm
```

(continues on next page)

(continued from previous page)

```
  address_root: observatory
  registry_address: observatory.registry

hosts:

    ocs-docker: {

        'agent-instances': [
            # Core OCS Agents
            {'agent-class': 'RegistryAgent',
             'instance-id': 'registry',
             'arguments': []},
            {'agent-class': 'AggregatorAgent',
             'instance-id': 'aggregator',
             'arguments': [['--initial-state', 'record'],
                           ['--time-per-file', '3600'],
                           ['--data-dir', '/data/']]},

            # Lakeshore agent examples
            {'agent-class': 'Lakeshore372Agent',
             'instance-id': 'LSA22YE',
             'arguments': [['--serial-number', 'LSA22YE'],
                           ['--ip-address', '10.10.10.4']]},

            {'agent-class': 'Lakeshore240Agent',
             'instance-id': 'LSA22Z2',
             'arguments': [['--serial-number', 'LSA22Z2'],
                           ['--num-channels', 8]]},
        ]
    }
```

The *hub* section defines the connection parameters for the crossbar server. This entire section will likely remain unchanged, unless you are running a site with multiple computers, in which case other computers will need to either run their own crossbar server, or point to an already configured one.

Under *hosts* we have defined a single host, *ocs-docker*. This configuration example shows an example where every OCS Agent is running within a Docker container. The hostname *ocs-docker* must match that given to your docker containers in the `docker-compose.yaml` file. We recommend naming the docker hosts based on your local hostname, however the configuration shown here will also work on a simple site layout.

Note: To determine your host name, open a terminal and enter `hostname`.

Each item under a given host describes the OCS Agents which can be run. For example look at the first 372 Agent:

```
{'agent-class': 'Lakeshore372Agent',
 'instance-id': 'LSA22YE',
 'arguments': [['--serial-number', 'LSA22YE'],
               ['--ip-address', '10.10.10.4']]},
```

The `agent-class` is given by the actual Agent which will be running. This must match the name defined in the Agent's code. The `instance-id` is a unique name given to this agent instance. Here we use the Lakeshore 372 serial number, *LSA22YE*. This will need to be noted for later use in the live monitoring. Finally the arguments are used to pass default arguments to the Agent at startup, which contains the serial number again as well as the IP address of the 372.

1.1.5 Running Docker

Our dependencies are met, the environment setup, and the configuration files configured; Now we're ready to run Docker. From your institution configuration directory (where the `docker-compose.yml` file is), run:

```
$ sudo -E docker-compose up -d
```

Note: The `-d` flag daemonizes the containers. If you remove it the output from every container will be attached to your terminal. This can be useful for debugging.

Note: The `-E` flag on sudo preserves the existing environment variables. This is needed as we use the *OCS_CONFIG_DIR* variable within the docker-compose file.

You can confirm the running state of the containers with the `docker ps` command:

```
$ bjk49@grumpy:~$ sudo docker ps
CONTAINER ID        IMAGE                                                                 ␣
↪    COMMAND                  CREATED             STATUS              PORTS             ␣
↪           NAMES
f325b0a95384        grumpy.physics.yale.edu/ocs-lakeshore240-agent:latest                 ␣
↪    "python3 -u LS240_ag..."   47 hours ago        Up 47 hours                           ␣
↪              prod_ocs-LSA22ZC_1_2cc23a32f274
e27946e2806f        grumpy.physics.yale.edu/ocs-lakeshore240-agent:latest                 ␣
↪    "python3 -u LS240_ag..."   47 hours ago        Up 47 hours                           ␣
↪              prod_ocs-LSA22Z2_1_e8ae8bdfcbe1
123c43ade64c        grumpy.physics.yale.edu/ocs-lakeshore240-agent:latest                 ␣
↪    "python3 -u LS240_ag..."   47 hours ago        Up 47 hours                           ␣
↪              prod_ocs-LSA24R5_1_81cb5b556c75
d0484abc5e22        grumpy.physics.yale.edu/ocs-lakeshore372-agent:latest                 ␣
↪    "python3 -u LS372_ag..."   2 days ago          Up 2 days                             ␣
↪              prod_ocs-LSA22YE_1_345860de361e
fb1274ec0983        grumpy.physics.yale.edu/ocs-lakeshore372-agent:latest                 ␣
↪    "python3 -u LS372_ag..."   2 days ago          Up 2 days                             ␣
↪              prod_ocs-LSA22YG_1_eccac22afb71
c4994af324f7        grumpy.physics.yale.edu/sisock-weather-server:v0.2.11                 ␣
↪    "python3 -u server_e..."   2 days ago          Up 2 days                             ␣
↪              prod_weather_1_b7f76f317d75
fed155bfcfad        grumpy.physics.yale.edu/sisock-g3-reader-server:v0.2.11-1-
↪g1ff12ac   "python3 -u g3_reade..."   2 days ago          Up 2 days                     ␣
↪                        prod_g3-reader_1_9e7e53ec96b0
70288c5d6ce6        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11             ␣
↪    "python3 -u thermome..."   2 days ago          Up 2 days                             ␣
↪              prod_LSA22YG_1_cd64f9656cfe
dd4906561ed1        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11             ␣
↪    "python3 -u thermome..."   2 days ago          Up 2 days                             ␣
↪              prod_LSA23JD_1_9a57b3fa29df
5956786cd5b4        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11             ␣
↪    "python3 -u thermome..."   2 days ago          Up 2 days                             ␣
↪              prod_LSA22YE_1_b5f1673d913f
810258e8893c        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11             ␣
↪    "python3 -u thermome..."   2 days ago          Up 2 days                             ␣
↪              prod_LSA22Z2_1_e6316efdbb2d
d8db9af9a1de        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11             ␣
↪    "python3 -u thermome..."   2 days ago          Up 2 days                             ␣
↪              prod_LSA24R5_1_19e6469ef97b
```

(continues on next page)

(continued from previous page)

```
91ecab00bd26        grumpy.physics.yale.edu/sisock-thermometry-server:v0.2.11
↪    "python3 -u thermome..."   2 days ago          Up 2 days
↪             prod_LSA22ZC_1_e1436bd60b9b
d92bcdf8468a        grumpy.physics.yale.edu/sisock-http:v0.2.11
↪    "python3 -u grafana_..."   2 days ago          Up 2 days
↪             prod_sisock-http_1_aeeb14fced5e
2a782c1aa9c4        eee74fd50cf5
↪    "python3 -u registry..."   2 days ago          Up 2 days
↪             prod_ocs-registry_1_ecacce7345b6
7e8e3d7372ca        grumpy.physics.yale.edu/ocs-aggregator-agent:latest
↪    "python3 -u aggregat..."   2 days ago          Up 47 hours
↪             prod_ocs-aggregator_1_5ed8fe90f913
8e7129ab199d        grumpy.physics.yale.edu/sisock-crossbar:v0.2.11
↪    "crossbar start"         2 days ago          Up 2 days           127.0.0.1:8001->
↪8001/tcp   prod_sisock-crossbar_1_7b0eb9ec21ff
a98066cc4569        grumpy.physics.yale.edu/sisock-g3-file-scanner:v0.2.11-1-g1ff12ac
↪    "python3 -u scan.py"     6 days ago          Up 6 days
↪           prod_g3-file-scanner_1_99d392723812
ddd6f1a63821        grafana/grafana:5.4.0
↪    "/run.sh"                6 days ago          Up 6 days           127.0.0.1:3000->
↪3000/tcp   prod_grafana_1_817207e03f75
cc0ef28deef0        e07bb20373d8
↪    "docker-entrypoint.s..."   6 days ago          Up 6 days           3306/tcp
↪             prod_database_1_a7c15d7039b9
```

This example shows all the containers running at Yale at the time of this writing.

Note: Since all the OCS Agents are configured to run in containers (which is our recommendation for running your system), there is no additional startup of OCS Agents. Previously these were either started individually by calling the agent script in a terminal, or using the *ocsbow* tool.

If your system is still setup to use these methods you can move to the Docker configuration by adding the required Agents to your docker-compose configuration and moving the Agent configurations in the ocs config file to a docker host block.

Using Docker

Many users may not be familiar with using Docker. Here are some useful tips for interacting with docker.

Viewing Logs

Each container has its own logs to which the output from the program running inside the container is written. The logs can be viewed with:

```
$ sudo docker logs <container name>
```

If you want to follow the logs (much like you would `tail -f` a file) you can run:

```
$ sudo docker logs -f <container name>
```

Updating Containers

If you have made changes to the docker compose configuration you need to update your containers by running the up command again:

```
$ sudo -E docker-compose up -d
```

This will rebuild any containers that have either been updated or depend on one which has updated.

To update a single service only you can run:

```
$ docker-compose stop <service name>
$ docker-compose up <service name>
```

Where you need to replace `<service name>` with the name of the service you have configured in your docker-compose configuration file.

Restarting Containers

If need to just restart a container, and haven't made any changes to your docker-compose configuration file you can do so with:

```
$ sudo docker-compose restart <container name>
```

Shutting Down All Containers

All the containers started with Compose can be stopped with:

```
$ sudo docker-compose down
```

1.1.6 Running OCS Clients

All the OCS Agents should now be running in Docker containers. To command them we run an OCS Client. Examples of OCS Clients can be found in the ocs/clients directory. As an example, we can start data acquisition on a Lakeshore using `ocs/clients/start_new_therm.py`:

```
$ python3 start_new_therm.py --target=LSA23JD
2019-01-10T11:53:52-0500 transport connected
2019-01-10T11:53:52-0500 session joined: SessionDetails(realm=<test_realm>,␣
↪session=1042697241527250, authid=<GJJU-4YG3-3UCG-CSMJ-TQTW-PWSM>, authrole=<server>,
↪ authmethod=anonymous, authprovider=static, authextra=None, resumed=None,␣
↪resumable=None, resume_token=None)
2019-01-10T11:53:52-0500 Entered control
2019-01-10T11:53:52-0500 Registering tasks
2019-01-10T11:53:52-0500 Starting Aggregator
2019-01-10T11:53:52-0500 Starting Data Acquisition
```

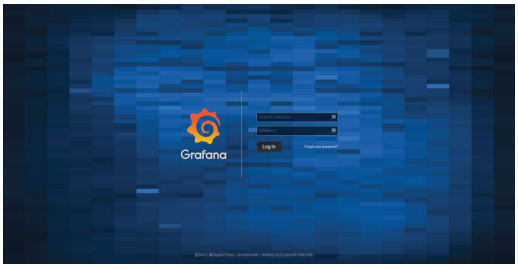
Once you have started data collection, the data is being acquired by the Lakeshore Agent and published over the crossbar server to the Aggregator Agent. There it is being written to disk at the location you have configured (*/data* in our example). The data is also passed to the live monitor, where it can be displayed in Grafana. We still discuss configuring Grafana in the next section.

1.1.7 Using Grafana

Our environment is configured, OCS Agents are running, and we have commanded them to acquire some data. We are now ready to configure Grafana. The configuration is not challenging, however dashboard configuration can be time consuming. We start with the first time setup steps.

Set a Password

When you first navigate to http://localhost:3000 in your web browser you will see the following page:

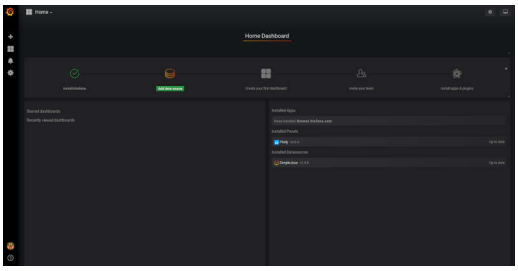


The default username/password are *admin*/*admin*. Once you enter this it will prompt you to set a new admin password. Select something secure, especially if your computer has a public IP address.

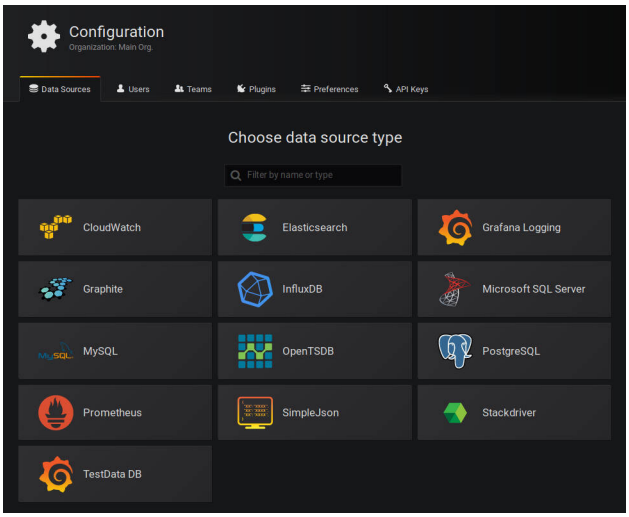
Note: Grafana can be a great way to check the housekeeping data at your site from anywhere you can access a web browser. For a more user friendly and secure connection you should setup a web server to proxy the connection to the Grafana container. This, however, is currently beyond the scope of this guide.

Configuring the Data Source

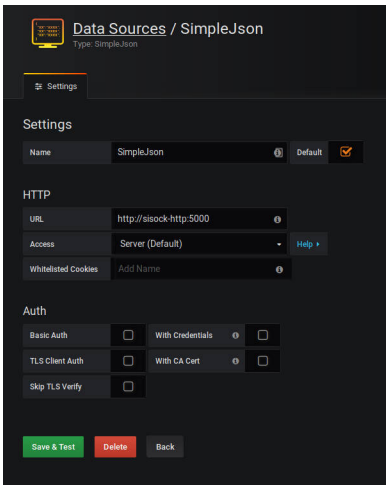
After setting the password you will end up on this page:



Click on the highlighted “Add data source” icon. (This is also accessible under the gear in the side menu as “Data Sources”.) You should then see this:

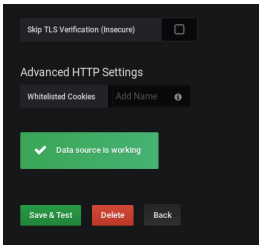


These are all the supported sources from which Grafana can pull data. We will use the SimpleJson source. Clicking on that icon will get you here:

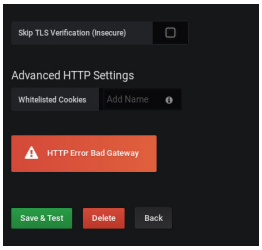


You can fill in what you want for a name, or keep the default. Make sure the "Default" checkbox is checked, as this will be our default data source when creating a new Dashboard. Finally, the URL must be `http://sisock-http:5000`. This is the name for the HTTP server we set in the `docker-compose.yml` file with the default port we assigned it.

When you click "Save & Test" a green alert box should show up, saying "Data source is working", like this:



If the Data Source is not working you will see an HTTP Error Bad Gateway in red:

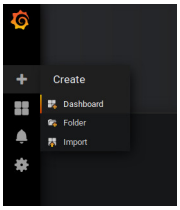


If this occurs it could be several things.

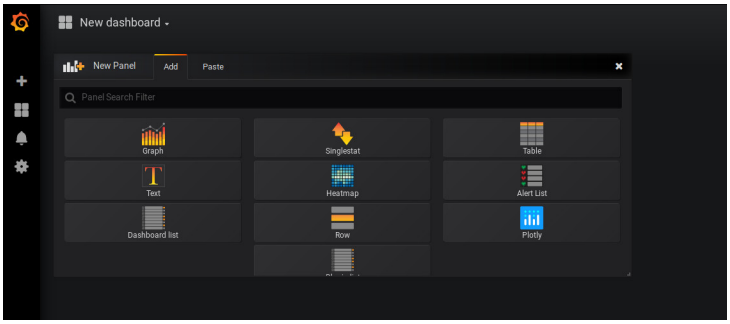
- Check the URL is correct
- Make sure you select the SimpleJson data source Type
- Check the sisock-http container is running

Configuring a Dashboard

Now that we have configured the Data Source we can create our first Dashboard. If you press back on the previous screen you will end up on the Data Sources menu. From any page you should have access to the sidebar on the left hand side of your browser. You may need to move your mouse near the edge of the screen to have it show up. Scroll over the top '+' sign and select "Create Dashboard", as shown here:

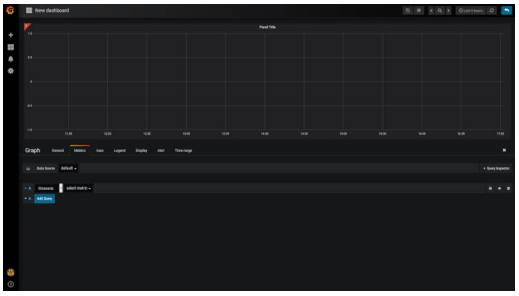


You will then see a menu like this:



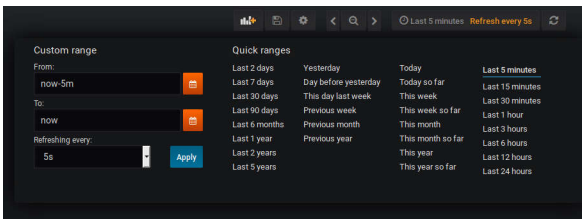
In this menu we are selecting what type of Panel to add to our Dashboard. We will add a Graph. When we first add the Graph it will be blank:

Click on the "Panel Title", and in the drop down menu, click "Edit". This will expand the plot to the full width of the page and present a set of tabbed menus below it.



We start on the "Metrics" tab. Here is where we add the fields we wish to plot. The drop down menu that says "select metric" will contain fields populated by the sisock data servers. Select an item in this list, for instructional purposes we will select the "apex_archive::apex_radiometer" metric, which is from an example data server which contains a small sample of data to verify the live monitor is working properly. Data should appear in the plot if you are running the example weather data server container and have selected the date range between July 13, 2017 and July 28, 2017.

You can configure the time interval and update intervals by clicking on the time in the upper right, it most likely by default says "Last 6 hours":



Note: The update intervals can be further customized by editing the dashboard. This can be done by clicking on the gear icon near the time settings. Keep in mind though that the more data you load the less you want to be rapidly querying the backend for displaying it. Performance may be impacted if you query large datasets rapidly.

Viewing the Live Monitor

Now we should start to see data in our live monitor.

Warning: If no data is showing up, you may have to select the metrics drop down menu again when first starting up. This is a known bug. Selecting the metric drop down should get data showing again. This is likely only a problem after you have a configured panel and restart the live monitor containers.

Here are some examples of what fully configured panels may look like:

Fig. 1: The diode calibration setup at Penn. Six diodes are readout on a single Lakeshore 240. The top plot shows the calibrated diode, reporting temperature in Kelvin. While the bottom plot shows the 5 uncalibrated diodes. The Top element is a SingleStat panel which shows the current temperature of the 4K plate via the calibrated diode.

Fig. 2: A demo Lakeshore 372 readout at Yale. The Lakeshore switches over 15 channels, reading each out for a few seconds before moving onto the next. Here the first eight channels are shown on the left plot, and the last seven shown on the right plot. There are 15 single stat panels below the plots showing the current values for each given channel.

1.1.8 Other Info

Grafana

The `grafana-storage` volume that we initialized will allow for persistent storage in the event the container is rebuilt. Dashboards can also be backed up by exporting them to a `.json` file.

Warning: This should be a one time setup, however, if you destroy the grafana-storage volume you will lose your configuration. We encourage you to export your favorite dashboards for backup.

Backing up Panels

Networking

Configuring a Firewall

Note: This problem is solved in part by the explicit exposure of the crossbar server port to localhost in our `docker-compose.yml` file in the line `ports: "127.0.0.1:3000:3000"`. This ensures port 3000 is only available to the localhost. If this is not done (i.e. "ports: 3000:3000") Docker will manipulate the iptables to make port 3000 available anywhere, so if your computer is publicly facing anyone online can (and will) try to connect. This will be evident in your crossbar container's logs.

That said, the firewall setup is not totally necessary, though still is good practice, so I will leave this information here.

If you have convinced your university IT department to allow you to have a Linux machine on the public network we should take some precautions to secure the crossbar server, which currently for OCS does not have a secure authentication mechanism, from the outside world. The simplest way of doing so is by setting up a firewall.

Ubuntu should come with (or have easily installable) a simple front end for iptables called ufw (Uncomplicated Firewall). This is disabled by default. Before configuring you should consider any software running on the machine which may require an open port. We will configure it to have ports 22 and 3000 open, for ssh and Grafana, respectively.

`ufw` should be disabled by default:

```
$ sudo ufw status
Status: inactive
```

You can get a list of applications which `ufw` knows about with:

```
$ sudo ufw app list
Available applications:
  CUPS
  OpenSSH
```

We can then allow the ssh port with:

```
$ sudo ufw allow OpenSSH
Rules updated
Rules updated (v6)
```

This opens port 22. And finally, we can allow Grafana's port 3000:

```
$ sudo ufw allow 3000
Rules updated
Rules updated (v6)
```

Lastly we have to enable `ufw`:

```
$ sudo ufw enable
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup
```

You should then see that the firewall is active:

```
$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
```

(continues on next page)

(continued from previous page)

```
3000                       ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
3000 (v6)                  ALLOW       Anywhere (v6)
```

Port Forwarding to View Remotely

If the computer you are running Grafana on is not exposed to the internet you can still access the web interface if you forward port 3000 to your computer.

You will need a way to ssh to the computer you are running on, so hopefully there is a gateway machine. To make this easier you should add some lines to your `.ssh/config`:

```
Host gateway
    HostName gateway.ip.address.or.url
    User username

Host grafana
    HostName ip.address.of.grafana.computer.on.its.network
    User username
    ProxyCommand ssh gateway -W %h:%p
```

Here you should replace "gateway" and "grafana" with whatever you want, but note the two locations for "gateway", namely the second in the ProxyCommand. This will then allow you to ssh through the gateway to "grafana" with a single command.

You can then forward the appropriate ports by running:

```
$ ssh -N -L 3000:localhost:3000 <grafana computer>
```

You should now be able to access the grafana interface on your computer by navigating your browser to `localhost:3000`.

1.2 Components

There are currently three components that make up a functioning sisock stack (in the context of live montioring, anyway). There's the crossbar server, the intermediate web server, and then all the `DataNodeServers`. This page documents how to get containers of these components and configure them in your `docker-config.yml`.

Components

1.2.1 sisock-crossbar

The crossbar server, an implementation of a WAMP router, serves as the communication channel between all the various components, as well as to parts of the Observatory Control System (OCS). This is provided by an image at `grumpy.physics.yale.edu/sisock-crossbar`.

Alongside this server runs the sisock Hub, which keeps track of all the running `DataNodeSevers`.

Configuration

Option	Description
container_name	sisock_crossbar - a currently hardcoded value in sisock, used for DNS resolution within Docker
ports	Exposes the crossbar container on localhost at port 8001, used for OCS communication. The 127.0.0.1 is *critical* if the computer you are running on is exposed to the internet.
volumes	We need the TLS certs bind mounted.
PYTHONBUFFERED	Force stdin, stdout, and stderr to be unbuffered. For logging in Docker.

```
sisock-crossbar:
  image: grumpy.physics.yale.edu/sisock-crossbar:0.1.0
  container_name: sisock_crossbar # required for proper name resolution in sisock code
  ports:
    - "127.0.0.1:8001:8001" # expose for OCS
  volumes:
    - ./.crossbar:/app/.crossbar
  environment:
      - PYTHONUNBUFFERED=1
```

1.2.2 sisock-http

Between Grafana and sisock sits an web server. This serves as a data source for Grafana, translating the queries from Grafana into data and field requests in sisock. The result is data from a `DataNodeServer` is passed through the crossbar server to the sisock-http server and then to Grafana over http, displaying in your browser.

Configuration

The image is provided at `grumpy.physics.yale.edu/sisock-http`, and depends on the crossbar server running to function. Since it communicates over the secure port on crossbar we need the TLS certificates mounted. The HTTP server defaultly runs on port 5000, but you can change this with the environment variable 'PORT', as shown in the example. There is also a 'LOGLEVEL' environment variable which can be used to set the log level. This is useful for debugging. txaio is used for logging.

```
sisock-http:
  image: grumpy.physics.yale.edu/sisock-http:0.1.0
  depends_on:
    - "sisock-crossbar"
  environment:
    PORT: "5001"
    LOGLEVEL: "info"
  volumes:
    - ./.crossbar:/app/.crossbar:ro
```

1.2.3 g3-file-scanner

This component will scan a directory for .g3 files, opening them and storing information about them required for the g3-reader DataNodeServer in a MySQL database. It scans at a given interval, defined as an environment variable. It also requires connection parameters for the SQL database, and a top level directory to scan.

Configuration

The image is provided at `grumpy.physics.yale.edu/sisock-g3-file-scanner`, and only depends on the database we store file information in.

```
g3-file-scanner:
  image: grumpy.physics.yale.edu/sisock-g3-file-scanner:0.2.0
  volumes:
    - /home/koopman/data/yale:/data:ro # has to match the mount in g3-reader
  environment:
      SQL_HOST: "database"
      SQL_USER: "development"
      SQL_PASSWD: "development"
      SQL_DB: "files"
      DATA_DIRECTORY: '/data/'
      SCAN_INTERVAL: 3600 # seconds
  depends_on:
    - "database"
```

1.2.4 Common Configuration

There are some environment variables which are common among all sisock components. These mostly relate to connection settings for the crossbar server. The defaults will work for a simple, single node, setup. However, moving to multiple nodes, in most cases, will require setting some of these.

Option	Description
WAMP_USER	The username configured for connecting to the crossbar server. This is the "role" in the crossbar config.
WAMP_SECRET	The associated secret for the WAMP_USER.
CROSSBAR_HOST	IP or domain name for the crossbar server.
CROSS-BAR_TLS_PORT	The port configured for secure connection to the crossbar server. In default SO configurations this is 8080.
CROSS-BAR_OCS_PORT	The port configured for open connection to the crossbar server. In default SO configurations this is 8001.

Warning: The default *WAMP_SECRET* is not secure. If you are deploying your crossbar server in a public manner, you should not use the default secret.

1.3 DataNodeServers

sisock components are made up of individual DataNodeServers which know how to retrieve a specific set of data. Each DataNodeServer (abbreviated DaNS, so as to avoid the commonly known DNS acryonym) runs within its own

Docker container. To use a DaNS you simply add a copy of its configuration to your `docker-compose.yml` file and edit it accordingly. Below you will find details for how to configure each DaNS.

Note: A quick note about versions for the Docker images. The version number, as of this writing, corresponds to the tagged version of sisock. You can view the tags on Github under releases. It is perhaps most safe, from a stability standpoint, to use a specific version number (i.e. 0.1.0), rather than "latest", which changes its underlying version number as new releases are made. The version number in these examples might not reflect the latest release, so you should check the releases page.

DataNodeServers

1.3.1 example-weather

An example `DataNodeServer` for demonstrating reading data from disk. A good choice to include for debugging your live monitor.

This container includes a small set of raw text files that contain weather data from the APEX telescope. The dataset runs from 2017-07-13 to 2017-07-27, so be sure to set your date range accordingly.

Configuration

The image is called `sisock-weather-server`, and should have the general dependencies. It also communicates over the secure port with the crossbar server, and so needs the bind mounted `.crossbar` directory.

```
weather:
  image: grumpy.physics.yale.edu/sisock-weather-server:latest
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
  volumes:
    - ./.crossbar:/app/.crossbar:ro
```

1.3.2 example-sensors

An example `DataNodeServer` for demonstrating the use of live data. The data is generated within the container through use of the lm-sensors program, which is already installed in the container. This returns the tempearture of your computer's CPU cores.

> **Warning:** There are some known problems getting this to run on some systems. It should work on an Ubuntu 18.04 box, but if you're having trouble and really want to get it running get in touch with someone from the DAQ group.

This is a fun early demo, but probably won't be useful to many users.

Configuration

The image is called `sisock-sensors-server`, and should have the general dependencies. It also communicates over the secure port with the crossbar server, and so needs the bind mounted `.crossbar` directory.

```
sensors:
  image: grumpy.physics.yale.edu/sisock-sensors-server:latest
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
  volumes:
    - ./.crossbar:/app/.crossbar:ro
```

1.3.3 apex-weather

A `DataNodeServer` based on the `example-weather` server, which reads weather data from the APEX telescope as archived by the ACT team. Used in production at the ACT site.

Configuration

The image is called `sisock-apex-weather-server`, and should have the general dependencies. It also communicates over the secure port with the crossbar server, and so needs the bind mounted `.crossbar` directory. In addition, you will need to mount the location that the data is stored on the host system.

There is an environment variable, `MAX_POINTS`, that can be used to configure the maximum number of points the server will return, this is useful for looking at large time ranges, where fine resolution is not needed.

```
apex-weather:
  image: grumpy.physics.yale.edu/sisock-apex-weather-server:latest
  volumes:
    - ./.crossbar:/app/.crossbar:ro
    - /var/www/apex_weather:/data:ro
  environment:
      MAX_POINTS: 1000
  depends_on:
    - "sisock_crossbar"
    - "sisock_grafana_http"
```

1.3.4 data-feed

A `DataNodeServer` which is able to subscribe to, cache, and serve live data from an OCS agent which publishes to an OCS Feed. This `DataNodeServer` communicates with the crossbar server on an unencrypted port so as to enable subscription to the OCS data feeds.

Data published by OCS Agents is cached in memory for up to an hour. Any data with a timestamp older than an hour is removed from the cache.

Configuration

The image is called `sisock-data-feed-server`, and should have the general dependencies.

There are several environment variables which need to be set uniquely per instance of the server:

Variable	Description
TARGET	Used for data feed subscription, must match the "instance-id" for the Agent as configured in your site-config file.
FEED	Used for data feed subscription. This must match the name of the ocs Feed which the ocs Agent publishes to.
NAME	Used to uniquely identify the server in Grafana, appears in sisock in front of the field name.
DESCRIPTION	Description for the device, is used by Grafana.
CROSSBAR_HOST	Address for the crossbar server
CROSS-BAR_TLS_PORT	Port for TLS communication to the crossbar server

The "TARGET" and "FEED" variables are used to construct the full crossbar address which is used for the subscription. This address ultimately looks like "observatory.TARGET.feeds.FEED". Failure to match to an address which has data published to it will result in no data being cached.

The "CROSSBAR_HOST" and "CROSSBAR_TLS_PORT" variables are useful when setting up a multi-node system. If hosted on the same computer the host is typically 'sisock-crossbar', else it will be the IP of the computer hosting it. The TLS port, unless changed in the crossbar configuration, should be 8080.

```
bluefors:
  image: grumpy.physics.yale.edu/sisock-data-feed-server:latest
  environment:
      TARGET: bluefors # match to instance-id of agent to monitor, used for data feed␣
↪subscription
      NAME: 'bluefors' # will appear in sisock a front of field name
      DESCRIPTION: "bluefors logs"
      FEED: "bluefors"
  logging:
    options:
      max-size: "20m"
      max-file: "10"
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
```

1.3.5 ucsc-radiometer

A `DataNodeServer` based on the `example-weather` server, which reads weather data from the UCSC radiometer located on Cerro Toco. Used in production at the ACT site.

Configuration

The image is called `sisock-radiometer-server`, and should have the general dependencies. It also communicates over the secure port with the crossbar server, and so needs the bind mounted `.crossbar` directory. In addition, you will need to mount the location that the data is stored on the host system.

There is an environment variable, `MAX_POINTS`, that can be used to configure the maximum number of points the server will return, this is useful for looking at large time ranges, where fine resolution is not needed.

```
ucsc-radiometer:
  image: grumpy.physics.yale.edu/sisock-radiometer-server:latest
  volumes:
    - ./.crossbar:/app/.crossbar:ro
    - /var/www/Skymonitor:/data:ro
  environment:
      MAX_POINTS: 1000
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
```

1.3.6 g3-reader

A `DataNodeServer` which reads data from g3 files stored on disk. This operates with the help of a MySQL database, which runs in a separate container. This database stores information about the g3 files, such as the filename, path, feed name, available fields and their associated start and end times. This enables the g3-reader DataNodeServer to determine which fields are available via a query to the database and to determine which files to open to retrieve the requested data.

The server will cache any data opened from a .g3 file. The data cache takes the form of a dictionary with the full path to the file as a key. The value is a dictionary with structure related to the structure within the .g3 file. The design of the cache allows loaded files to be popped out of the dictionary to prevent the cache from growing too large (though currently a good cache clearing scheme is not implemented).

Configuration

The image is called `sisock-g3-reader-server`, and should have the general dependencies. It also communicates over the secure port with the crossbar server, and so needs the bind mounted `.crossbar` directory. In addition, you will need to mount the location that the data is stored on the host system.

There is an environment variable, `MAX_POINTS`, that can be used to configure the maximum number of points the server will return, this is useful for looking at large time ranges, where fine resolution is not needed.

Additionally, there are environment variables for the SQL connection, which will need to match those given to a mariadb instance. Both configurations will look like:

```
g3-reader:
  image: grumpy.physics.yale.edu/sisock-g3-reader-server:latest
  volumes:
    - /home/koopman/data/yale:/data:ro
    - ./.crossbar:/app/.crossbar
  environment:
      MAX_POINTS: 1000
      SQL_HOST: "database"
      SQL_USER: "development"
      SQL_PASSWD: "development"
```

(continues on next page)

(continued from previous page)

```
      SQL_DB: "files"
  depends_on:
    - "sisock-crossbar"
    - "sisock-http"
    - "database"

database:
  image: mariadb:10.3
  environment:
    MYSQL_DATABASE: files
    MYSQL_USER: development
    MYSQL_PASSWORD: development
    MYSQL_RANDOM_ROOT_PASSWORD: 'yes'
  volumes:
    - database-storage-dev:/var/lib/mysql
```

CHAPTER 2

Developer's Guide

If you are interested in the details of how individual components work, look here.

2.1 g3 File Scanner

Reading back housekeeping data from g3 files written to disk is a key feature required of the housekeeping monitor. This task is performed in two parts. The first is to scan the files. Each g3 file is read and metadata about what is stored in the files is recorded into a MySQL database. The component that performs this first task is called the *g3-file-scanner*. The second, is the opening and caching of the data in specific g3 files which are requested. This is the data server, *g3-reader*. This page describes the inner workings of the *g3-file-scanner*.

Contents

2.1.1 Overview

The *g3-file-scanner* (referred to here as the "file scanner") will scan a given directory for files with a *.g3* extension, open them, and record metadata about them required for the *g3-reader* data server in a MySQL database. The scan occurs on a regular interval, set by the user as an environment variable.

Note: The first scan of a large dataset will take some time, depending on how much data you have.

2.1.2 SQL Database Design

The SQL database is split into two tables, the "feeds" table and the "fields" table. "feeds" stores the filename and path to the file along with the *prov_id* and *description* from within the g3 file. The *description* will be used to assemble a unique sisock field name. Additionally, the "feeds" table keeps track of whether a scan has completed on the given file (with the *scanned* column), and assigns each file a unique *id*.

A description and example of the "feeds" table is shown here:

```
MariaDB [files]> describe feeds;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id          | int(11)      | NO   | PRI | NULL    |       |
| filename    | varchar(255) | YES  | MUL | NULL    |       |
| path        | varchar(255) | YES  |     | NULL    |       |
| prov_id     | int(11)      | YES  |     | NULL    |       |
| description | varchar(255) | YES  |     | NULL    |       |
| scanned     | tinyint(1)   | NO   |     | 0       |       |
+-------------+--------------+------+-----+---------+-------+
6 rows in set (0.010 sec)

MariaDB [files]> select * from feeds limit 3;
+----+-----------------------+-------------+---------+--------------------+---------
↪+
| id | filename              | path        | prov_id | description        |␣
↪scanned |
+----+-----------------------+-------------+---------+--------------------+---------
↪+
|  1 | 2019-03-18-16-52-46.g3 | /data/15529 |       0 | observatory.LSA22ZC |       1␣
↪|
|  2 | 2019-03-18-16-52-46.g3 | /data/15529 |       1 | observatory.LSA23JD |       1␣
↪|
|  3 | 2019-03-18-16-52-46.g3 | /data/15529 |       3 | observatory.LSA22YG |       1␣
↪|
+----+-----------------------+-------------+---------+--------------------+---------
↪+
3 rows in set (0.001 sec)
```

The "fields" table has a row for each ocs field within a file (i.e. "Channel 1", "Channel 2", channels for a given Lakeshore device), the start and end times for the field, and the correspoding 'id' in the feeds id, stored here as "feed_id".

A description and example of the "fields" table is shown here:

```
MariaDB [files]> describe fields;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| feed_id | int(11)      | NO   | MUL | NULL    |       |
| field   | varchar(255) | YES  |     | NULL    |       |
| start   | datetime(6)  | YES  |     | NULL    |       |
| end     | datetime(6)  | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
4 rows in set (0.001 sec)

MariaDB [files]> select * from fields limit 3;
+---------+----------+----------------------------+----------------------------+
```

(continues on next page)

(continued from previous page)

```
| feed_id | field     | start                      | end                        |
+---------+-----------+----------------------------+----------------------------+
|       1 | Channel 1 | 2019-03-18 16:51:55.762230 | 2019-03-18 17:01:56.772258 |
|       1 | Channel 2 | 2019-03-18 16:51:55.762230 | 2019-03-18 17:01:56.772258 |
|       1 | Channel 3 | 2019-03-18 16:51:55.762230 | 2019-03-18 17:01:56.772258 |
+---------+-----------+----------------------------+----------------------------+
3 rows in set (0.001 sec)
```

The "description" table is a simple, single column, table containing the combined ocs field and ocs descriptions distinctly across all .g3 files. This is used as the field list that the g3-reader will return.

A description and example of the "description" table is shown here:

```
MariaDB [files]> describe description;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| description | varchar(255) | NO   | PRI | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
1 row in set (0.001 sec)

MariaDB [files]> select * from description limit 3;
+---------------------------------+
| description                     |
+---------------------------------+
| observatory.LSA22YE.channel_01_r |
| observatory.LSA22YE.channel_01_t |
| observatory.LSA22YE.channel_02_r |
+---------------------------------+
3 rows in set (0.000 sec)
```

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 API

This page contains the auto-generated documentation for the sisock package.

3.1.1 sisock.base module

Sisock: serve Simons data over secure sockets (`sisock`)

Classes

sisock.base.DataNodeServer([config])	Parent class for all data node servers.

Functions

sisock.base.uri(s)	Compose a full URI for pub/sub or RPC calls.
sisock.base.sisock_to_unix_time(t)	Convert a sisock timestamp to a UNIX timestamp.

Constants

WAMP_USER Username for servers/hub to connect to WAMP router.

WAMP_SECRET Password for servers/hub to connect to WAMP router.

WAMP_URI Address of WAMP router.

REALM Realm in WAMP router to connect to.

BASE_URI The lowest level URI for all pub/sub topics and RPC registrations.

class sisock.base.**DataNodeServer**(*config=None*)

Bases: autobahn.twisted.wamp.ApplicationSession

Parent class for all data node servers.

Variables

- ***name*** (*string*) – Each data node server inheriting this class must set its own name. The hub will reject duplicate names.
- ***description*** (*string*) – Each data node server inheriting this class must provide its own, human- readable description for consumers.

after_onJoin(*details*)

This method is called after onJoin() has finished.

This method can be overridden by child classes that need to run more code after the parent onJoin method has run.

Parameters **details** (autobahn.wamp.types.SessionDetails) – Details about the session, as passed to onJoin.

description = None

get_data(*field*, *start*, *end*, *min_stride=None*)

Request data.

This method can overridden by child classes if a get_data runs in the main reactor thread, otherwise it's the get_data_blocking method that needs to be overridden

Parameters

- **field** (*list of strings*) – The list of fields you want data from.
- **start** (*float*) – The start time for the data: if positive, interpret as a UNIX time; if 0 or negative, begin *start* seconds ago.
- **end** (*float*) – The end time for the data, using the same format as *start*.
- **min_stride** (float or None) – If not None then, if necessary, downsample data such that successive samples are separated by at least *min_stride* seconds.

Returns

On success, a dictionary is returned with two entries.

- **data** [A dictionary with one entry per field:]
  - field_name : array containing the timestream of data.
- **timeline** [A dictionary with one entry per timeline:]
  - timeline_name : An dictionary with the following entries.
  - t : an array containing the timestamps
  - finalized_until : the timestamp prior to which the presently requested data are guaranteed not to change; None may be returned if all requested data are finalized

If data are not available during the whole length requested, all available data will be returend; if no data are available for a field, or the field does not exist, its timestream will be an empty array. Timelines will only be included if there is at least one field to which it corresponds with available data. If no data are available for any of the fields, all arrays will be empty.

If the amount of data exceeds the data node server's pipeline allowance, `False` will be returned.

Return type dictionary

get_fields(*start*, *end*)
Get a list of available fields and associated timelines available within a time interval.

Any field that has at least one available sample in the interval *[start, stop)* must be included in the reply; however, be aware that the data server is allowed to include fields with zero samples available in the interval.

This method should be overridden by child classes if the fields are obtained in a non-blocking way (running in the reactor thread).

Parameters

- **start** (*float*) – The start time for the field list. If positive, interpret as a UNIX time; if 0 or negative, get field list *t* seconds ago.
- **end** (*float*) – The end time for the field list, using the same format as *start*.

Returns

Two dictionaries of dictionaries, as defined below.

- **field** [the field name is the key, and the value is:]
  - description : information about the field; can be *None*.
  - timeline : the name of the timeline this field follows.
  - type : one of "number", "string", "bool"
  - units : the physical units; can be *None*
- **timeline** [the field name is the key, and the value is:]
  - interval : the average interval, in seconds, between readings; if the readings are aperiodic, `None`.
  - field : a list of field names associated with this timeline

The *field* dictionary can be empty, indicating that no fields are available during the requested interval.

Return type dictionary

name = None

onChallenge(*challenge*)
Fired when the WAMP router requests authentication.

Parameters **challenge** (`autobahn.wamp.types.Challenge`) – The authentication request to be responded to.

Returns **signature** – The authentication response.

Return type string

onConnect()
Fired when session first connects to WAMP router.

onJoin(*details*)
Fired when the session joins WAMP (after successful authentication).

After registering procedures, the hub is requested to add this data node server.

Parameters **details** (autobahn.wamp.types.SessionDetails) – Details about the session.

sisock.base.**sisock_to_unix_time**(*t*)

Convert a sisock timestamp to a UNIX timestamp.

Parameters **t** (*float*) – A sisock timestamp.

Returns **unix_time** – If *t* is positive, return *t*. If *t* is zero or negative, return $time.time() - t$.

Return type float

sisock.base.**uri**(*s*)

Compose a full URI for pub/sub or RPC calls.

Parameters **s** (*The final part of the URI to compose.*) –

Returns **uri** – The string returned is "%s.%s" % (BASE_URI, s).

Return type string

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index