Igor Documentation

Release 0.99.2

Jack Jansen

Mar 15, 2021

Contents:

This documentation was generated for Igor version 0.99.2

CHAPTER 1

Introduction

Igor is named after the Discworld characters of the same name. You should think of it as a butler (or valet, or major-domo, whatever the difference is) that knows everything that goes on in your household, and makes sure everything runs smoothly. It performs its tasks without passing judgement and maintains complete discretion, even within the household. It can work together with other Igors (lending a hand) and with lesser servants such as Iotsa-based devices.

Home page is https://github.com/cwi-dis/igor. This software is licensed under the MIT license by the CWI DIS group, http://www.dis.cwi.nl.

1.1 What Igor does

Igor performs its tasks of managing your household by knowing three things:

- what is going on at the moment,
- what needs to happen when, and
- how to make that happen.

For a freshly-installed Igor all of these three categories are pretty much empty, so basically Igor sits there doing nothing.

But then you can add plugins for sensors that allow Igor to get an understanding of what is going on: environmental sensors help it determining the indoor temperature, but also probes to test whether your internet connection is working or which personal devices (mobile phones) are currently connected to the local WiFi network.

Next you can add plugins for controlling devices: turning on or off electrical appliances, lights, music, etc.

Finally you add rules to explain to Igor what needs to happen when. Igor can now ensure that there is music playing between 08.00 and 09.00 on weekdays, but only if there are people at home.

1.2 Technical description

Igor is basically a hierarchical data store, with three basic operating agents:

- Plugins for sensing devices modify the data (for example recording a fact such as *"device with MAC address 12:34:56:78:9a:bc has obtained an IP address from the DHCP service"*).
- Rules trigger on data changes and modify other entries in the database (for example *"if device 12:34:56:78:9a:bc is available then Jack is home"* or *"If Jack is home the TV should be on"*).
- Actions trigger on data changes and allow control over external hardware or software (for example *"If the TV should be on emit code 0001 from the infrared emitter in the living room"*).

1.3 Comparison to other IoT solutions

Igor can be completely self-contained, it is not dependent on any cloud infrastructure. This means your thermostat should continue controlling your central heating even if the Google servers are down. Moreover, Igor allows you to keep information in-house, so you get to decide which information you share with whom (so you do not have to share your room temparature with Google if you do not want to). That said, Igor can work together with cloud services, so you can use devices for which vendor-based cloud use is the only option, but at least *you* get to combine this information with other sensor data.

Igor is primarily state-based, unlike ITTT (If This Then That) and many other IoT platforms which are primarily event-based. The advantage of state-based is that it allows you to abstract information more easily. In the example of the previous section, if you later decide to detect *"Jack is home"* with a different method this does not affect the other rules.

Igor has a fine-grained access control mechanism, unlike most (all?) other IoT solutions. This makes it possible to give some people (or automatic agents) access to high-level abstracted information without giving them access to the low-level information that causes Igor to known this high-level information. In the example from the previous paragraph, you can give a person (or service) access to the state variable *"Jack is home"* without giving them the MAC address of his phone.

1.4 Usage

After initial installation Igor will usually be started at system boot time on some machine in the household that has access to the network and that is always on. A Raspberry PI sitting somewhere in a corner is an option, but any MacOSX or Linux machine can be used.

Igor then presents a user interface at port 9333. So if the hostname of the machine Igor is running on is *igor.local* you can access this user interface by browsing to <https://igor.local:9333/>.

The user interface allows you to inspect all items in the Igor database that are considered fully accessible, in other words: all information about your household that you consider to be available to anyone.

You can also log in. Initially Igor knows about a single user, *admin*, with no password. You can add a password (you should). When logged in you can access more information; for instance the *admin* user can access everything in the database. You can then add accounts for other users, such as yourself and other people in the family. You can grant specific rights (access permissions) to different users.

In the user interface you can then add devices, sensors and other plugins (for example to determine the health of your internet connection, or whether backups of important machines are up-to-date). You can also modify access control rules, examine the Igor log file, etc.

In addition various plugins have their own user interface allowing you to control them or to inspect their current status.

There are also some command line tools that are meant primarily for use in shell scripts but can also be used to manually control Igor.

1.5 Implementation overview

Igor is implemented in Python (and works with both version 2 as 3). At the core of Igor is an XML datastore with locking for concurrent access, and an XPath 1.0 implementation to allow searching, selecting and combining (using expressions) of data.

Alongside that is a webserver with either *http* or *https* access that allows REST access to the data (GET, PUT, POST and DELETE methods), by default on port 9333. The server handles conversion between the internal (XML) format and external XML, JSON or plain text formats.

In addition to data access, the web server exposes internal functionality (for example for saving the database) and more general XPath expressions over the database. It can also serve static content and template-based content (using the Jinja2 template engine and data from the database).

1.5.1 Plugins

There is a plugin mechanism that allows adding plugins that can control external devices based on variables in the database changing. Or they can change database variables to reflect the state of external devices. Or both.

A number of plugins is included. Some of these are generally useful, some should be considered example code to help you develop your own plugins. See *Plugin implementation* for a description of the plugin architecture and *Standard Plugins* for the standard plugins.

Some of the plugins come with helper utilities or servers. See helpers/readme.md for details.

1.5.2 Actions

Then there is an `actions` module, populated from a special section of the database, that allows actions to be triggered by events. Here, *actions* are REST operations (on the database itself or on external URLs) using data constructed from the database, and *events* are one or a combination of:

- periodic timers,
- specific incoming REST requests,
- changes to database nodes that match specific XPath selectors.

1.5.3 Security and privacy

Igor has an optional capability-based access control mechanism that allows fine-grained control over which agent (user, external device, plugin, etc) is allowed to do which operation. Human users can log in to the Igor server to gain access to their set of capabilities, external devices can carry their capabilities in requests. Igor can handle signing those capabilities with a secret key shared between the device and Igor. Actions and plugins can also carry a private set of capabilities, so you can design things so that the action has privileges that its caller does not have.

For convenience on a local subnet Igor can also function as a Certificate Authority (CA), signing the SSL certificates needed to allow trusted *https* access between Igor and external devices (and any other local services you have).

1.5.4 External interfaces

There are a number of *Command line utilities* and *Python modules*, such as `igorVar` to allow access to the database REST interface from shell scripts, `igorSetup` to initialize and control the database or `igorCA` to access the Certificate Authority.

And of course there is the main REST interface described in *Rest entry points*.

1.5.5 Missing functionality

There are two common operations that cannot currently be done through the user interface, at least not easily:

- Modifying individual data items,
- Adding (or changing or deleting) action rules.

For now you have to use the *Command line utilities*, or stop Igor and edit the XML database manually.

The user interface is currently not very logically organized, and it is completely unstyled and ugly.

There is no friendly user interface yet to manually modify the database.

There is no friendly user interface yet to modify actions.

Mirroring and distributing the database over multiple Igor instances is planned but not implemented yet.

A method for easy installation (and updating and removal) of externally supplied plugins is not implemented yet.

CHAPTER 2

Installation

2.1 Prerequisites

You need to have Python 3.6 or later installed.

You need *pip* and *setuptools* (if not already included in your Python installation). Installing Python's package installation program *pip* will also install *setuptools*.

Your system might well have both Python 2.7 and Python 3.X installed, for that reason it is best to always use the commands `python3` and `pip3` (as opposed to `python` and `pip`, which could refer to 2 or 3 depending on circumstances).

See also https://packaging.python.org/tutorials/installing-packages/.

2.2 Installing from PyPi

Assuming you are using Python 3, run the following command:

```
[sudo] python3 -m pip install igor-iot
```

Depending on your system installation you may be able to run it without `sudo`, but even if this gives no permission errors it may actually still install in your home directory in stead of in the system directories, which may cause problems later if you want to run Igor as a system daemon. But, to make matters more complicated, some times it is better to run *without* `sudo`, for example when installing on MacOS with a `brew` installed Python. Sorry, you have to find out yourself what works best...

Note that installing with `--user` or installing in a *virtualenv* has not been tested, at least not extensively.

After installing the software you can continue with *Initial setup*.

2.3 Installing from source

Alternatively you can build Igor from source. Download the source via https://github.com/cwi-dis/igor.

```
git clone https://github.com/cwi-dis/igor
```

Then change into the `igor` directory, and install with

```
[sudo] python3 -m pip install .
```

This will allow uninstalling as well, so it is the preferred method of installation. You can also supply the `--editable` flag if you know what you are doing.

Side note: If you really want you can also install with setuptools, but this is not recommended:

```
[sudo] pip3 install -r requirements.txt
python3 setup.py build
[sudo] python3 setup.py install
```

This installs the main binary `igorServer` and the utilties `igorVar`, `igorSetup`, `igorControl` and `igorCA`.

These instructions should install Igor and the required dependencies for all users on your system. Installing for the current user only may be possible but is untested.

You may also want to install some of the helper utilities from the `helpers` subdirectory.

2.3.1 Testing the software (optional)

There is a unittest-based test suite in the `test` subdirectory. The easiest way to run the tests is to first install the software (as per the instructions above) and then run

```
python3 setup.py test
```

This will run all tests in various configurations (with and without https support, with and without capability support, etc). This will run the tests in complete isolation, using a separate temporary install and separate Igor instances running on different ports, etc. So it will not interfere with your installed igor.

If you want more control (for example to specify test options and such) you can in stead use the following command line (which will use the Igor version installed on your system):

```
python3 -m test.test_igor
```

It is also possible to test the performance of Igor (again with the various configurations):

```
python3 -m test.perf_igor
```

will run a set of actions similar to the unittests (for a minimum number of calls and a minimum duration) and report number of calls, average runtime per call and standard deviation of the runtimes.

2.4 Updating the software

Note: currently the database format (particularly the schema) may change between releases. You should check the release notes to ensure your database is still compatible, and otherwise convert it manually after updating.

Stop the server if necessary:

```
igorControl -u http://localhost:9333 stop
```

In the `igor` directory, do

```
git pull
```

and repeat the installation step steps from earlier. Either

```
[sudo] python3 -m pip install .
```

or

```
[sudo] pip3 install -r requirements.txt
python3 setup.py build
[sudo] python3 setup.py install
```

Restart the server:

```
igorServer
```

CHAPTER 3

Initial setup

Make sure you have first installed the Igor software, according to the instructions in *Installation*.

3.1 Setup the database

Create an initial empty database with

```
igorSetup initialize
```

The default database is stored in `~/.igor`. Currently, Igor databases are *not* compatible between versions, so if you have used an older version of Igor you should first remove the old database.

Next add the standard plugins:

```
igorSetup addstd systemHealth ca user device actions editData
igorSetup addstd lan home say
```

The first set of those are plugins that are used for Igor administration. Technically they are optional, i.e. Igor itself will work fine without them, but practically they are needed to allow you to administer your Igor server.

The second set are really optional, but they provide convenience functions such as checking that the internet works, and determining how any people ar currently at home. You may want to skip installing these right now and add them later via the *Administration* interface.

You should now be able to run the server with

```
igorServer
```

and point your browser at http://localhost:9333 to see Igor in action.

At any time the server is not running you can check the consistency of the database, with

```
igorServer --check
```

or alternatively you can try to automatically fix it with

```
igorServer --fix
```

3.2 Setup security

It is advisable to run Igor with the secure *https* protocol as opposed to the completely open *http* protocol. Igor can use any SSL certificate, but simplest is to configure Igor as a Certificate Authority.

3.2.1 Setup Igor as a CA

Enabling Igor as a Certificate Authority (CA) for the `.local` domain is the generally the best option (but see below for alternatives). Details on using Igor as a CA are in *Command line utilities* but here is the information to get started.

A CA needs a well-known name, so that people who receive a certificate signed by it can see who they ultimately trust by trusting that certificate. Even though it is rather pointless for our `.local` CA you will still have to specify the mandatory fields *country*, *state*, *organization* and it is a good idea to specify *common name*. You also need a different *common name* for the intermediate issuer. Initialize your CA with the following command, but replace *igor.local*, *NL*, *Netherlands* and *Jack Jansen* with values that make sense for your situation:

```
igorCA initialize '/CN=root.ca.igor.local/C=NL/ST=Netherlands/O=Jack Jansen' '/
↪CN=intermediate.ca.igor.local/C=NL/ST=Netherlands/O=Jack Jansen'
```

(After this you don't really need the CA root key on your machine anymore, because the CA intermediate key will be used for everything. The directory `~/.igor/ca/root` has been made inaccessible, but if you are really security-conscious you can put its content on a USB stick, put it in a safe and remove all of `~/.igor/ca/root`).

Now you can use your newly-created CA to sign the certificate for the Igor server:

```
igorCA self igor.local localhost 127.0.0.1 ::1
```

The `self` command should be given all hostnames and IP addresses via which you expect to access Igor, and the "canonical name" should be first. So, the `igor.local` in the example above should be replaced by the DNS or mDNS name you normally use to access this host.

If you ever want to access Igor from Windows you should be aware of the fact that Windows does not have good support for mDNS *.local* names. You must either install some extension that supports this, or you must ensure that your Igor host has a fixed IP address and also add that address to the list of hostnames and IP addresses.

Finally you need to install the root certificate for the Igor CA into Igor and (if you want to access Igor with a browser or other software) into your system set of trusted root certificates. How this is done depends on whether you run Linux or OSX and which version you run (google for *"install root certificate"* with your OS name) but you get the Igor CA root certificate chain with the following command:

```
igorCA getRoot
```

More commands are forthcoming here.

3.2.2 Alternative: Use another Igor as CA

If you already have another instance of Igor running on the local network and that other Igor has been setup as a CA you can set things up so that this Igor uses the CA of the other Igor.

Let's say the other igor is running on machine *masterigor.local*. You can create a secret key and a Certificate Signing Request, and then ask the other Igor to sign the certificate with the following command:

```
igorCA --remote --url https://masterigor.local:9333/data/ --noverify self igor.local␣
↪localhost 127.0.0.1 ::1
```

And again, you have to get and install the root certificate:

```
igorCA --remote --url https://masterigor.local:9333/data/ --noverify getRoot
```

Note: You may need to pass username and password to the `igorCA` commands if the master Igor needs them. Add arguments of the form `--credentials username:password` after the `--noverify`.

3.2.3 Alternative: Self-signed Certificate

Deprecated since version 0.9: Enabling Igor as a CA is better than using a self-signed certificate, because with a self-signed certificate you will have to go through a lot of ominous-sounding security dialogs for each browser with which you want to access Igor.

To use a self-signed certificate for Igor, run

```
igorSetup certificate
```

And restart Igor. Igor will detect that it has a certificate and start up in secure mode.

Now connect your browser to https://localhost:9333. You will get a number of warnings about an untrusted website (because you used a self-signed certificate), read these and select all the answers that indicate you trust this website. This needs to be done only once per browser per user per machine.

3.2.4 Alternative: Use a real certificate

If you happen to have access to a real trusted CA and your Igor runs on a machine with a public DNS domain name you can use the following command (after supplying the correct hostname) to create a secret key and Certificate Signing Request:

```
igorCA selfCSR igor.your.domain.name
```

This will store the secret key in the file *~/.igor/igor.key* and output the CSR (certificate signing request). You send this CSR to your CA, which will sign it and return you a certificate. You store this certificate in *~/.igor/igor.crt* (in PEM format).

3.2.5 Alternative: run without https

It is possible to run Igor without *https*, using only *http* access, but this is only advisable in very specific situations where you know you network is physically secure and completely isolated from the internet. You simply don't run any *igorCA* commands, and Igor will start up using the http protocol (after issuing a warning). If you had previously already created certificates and keys and such and you want to revert to http mode you can remove `~/.igor/ca`, `~/.igor/igor.key` and `~/.igor/igor.crt`.

3.3 Capability-based access control

Igor has support for experimental fine grained access control, using capabilities. On top of that there is user-based (login) access control.

This feature is incomplete, especially the documentation is lacking, therefore it is not enabled by default. If you want to experiment you can use first

```
igorServer --capabilities --fix
```

to add the required set of minimal capabilities to your database, and then run

```
igorServer --capabilities
```

to run your server in capability-based acess control mode. You will probably need various visits to the */users.html*, */devices.html* and */capabilities.html* administrative interfaces to get anything to work.

It is also possible to let Igor go through all the motions of capability-based access control, but allowing the operations even if the capabilities would disallow it. This can be handy while converting your database to use capabilities: you will get all the error messages about missing capabilities, but as warnings only. Therefore your Igor server will function as if no capabilities were in use. Enable this mode with

```
igorServer --warnCapabilities
```

3.4 Igor configuration

You will need to configure your Igor to do something useful. On the Igor landing page there are links to pages that allow you to add *devices*, *plugins* and *users*.

> Note: this functionality is currently incomplete, so some things will have to be configured manually. Specifically: actions cannot be created through a user interface.

3.4.1 Manual configuration

The database is an XML file, so it can be edited in a normal text editor. But: you should make sure Igor is not running while you are editing, or it may override your changes.

See *Igor database schema* and *~/.igor directory structure* for information on how to add things manually.

The following command helps you with stopping Igor during an edit and restarting it afterwards:

```
igorSetup edit
```

3.5 Starting automatically

Igor can be started automatically at system boot with the following command:

```
igorSetup runatboot
```

On OSX and Linux this should start Igor as a deamon process. Igor will run under your user ID, and use the *.igor* database in your home directory.

3.6 Running a temporary igor

Sometimes you want to run a copy of igor, for example to debug a plugin. This is fairly easy, because there are environment variables that are honoured by `igorServer` and by all the utlities.

Create and run a fresh igor on port 19333 with:

```
igorSetup sandbox
```

That command is roughly equivalent to the following sequence of commands:

```
export IGORSERVER_DIR=/tmp/temp-igor
export IGORSERVER_PORT=19333
igorSetup initialize
igorSetup addstd systemHealth ca user device actions editData
igorServer
```

3.7 Debugging Igor

It is also possible to run the igorServer with a `python` command line. This may be useful if you want to use `python3 -i` so you can then `import pdb ; pdb.pm()` to debug an igor crash.

```
python3 -m igor
```

if you send a running igorServer a `SIGQUIT` signal (with `kill --QUIT` or by typing control-\ interactively) igor will print a stack trace for all of its threads.

CHAPTER 4

Administration

To be supplied. For now, the Igor homepage has a number of quick links to allow you to:

- login (use *admin* to administer the service)
- change your password
- add, edit and delete users (through *user - manage igor users*)
- add, edit and delete devices (through *device - manage igor devices*)
- add, edit and delete actions (through *action - manage igor actions*)
- edit any data item (through *editData - edit entries in the database*)
- administration of SSL certificates (through *ca - Certificate Authority*)
- administration of capabilities
- install and delete plugins (but for device plugins use the device interface)
- view the high-level status of everything (through *systemHealth - look after health of services and other infrastructure*)
- view the service log
- view the internal REST action status

CHAPTER 5

Standard Plugins

Igor comes with a set of standard plugins. Some of these can be used as-is, and installed using (for example):

```
igorSetup addstd copytree
```

Some are more examples that you should copy and adapt to your own need, or use as inspiration.

Plugins take their name (for use in `plugindata`, for example) from the name they are installed under (which can be different than the standard name). So you can install multiple independent copies (for example as *say* and *sayInBedroom*) and use different plugindata to control each copy of the plugin.

Various plugins should be considered standard to Igor operations and usually installed:

- *ca* allows access to the Certificate Authority
- *device* allows adding and removing devices
- *user* allows adding and removing users
- *systemHealth* implements the self-checks and health-checks of Igor
- *user* allows managing users: adding new users, changing passwords, etc

5.1 ble

Listens to Bluetooth Low Energy advertisements to determine which devices are in range. See the ble readme for details.

5.2 buienradar

Stores rain forecast for the coming hour. See buienradar/readme.md for details.

5.3 ca

Certificate Authority. Programming interface to the Igor SSL certificate authority that allows *https:* access to local services (like Igor itself). See ca/readme.md for details.

5.4 copytree

Copies subtree `src` to `dst`. Both src and dst can be local or remote, allowing mirroring from one Igor to another (or actually between any two REST servers). Optional arguments `mimetype` and `method` are available.

5.5 device

Low level API to add devices and their capabilities and secret keys, and allow devices to call certain actions. User interface is provided.

5.6 dhcp

Queries database of DHCP server on local machine and stores all active dhcp leases. See dhcp/readme.md for details.

5.7 fitbit

Retrieves health data from Fitbit devices using the Fitbit cloud API and stores this in `sensors/fitbit`. See fitbit/readme.md for details.

Note the underscore: the plugin is called `fitbit` because otherwise it would have a name clash with the underlying Python module it uses.

As of October 2018 this is the first plugin to have a user interface through a `setup.html` page, it can be used as an example of such. It is also currently the only plugin that implements *OAuth2* to retrieve data from external websites.

5.8 homey

Example Homey integration. See homey/readme.md for details.

Needs work.

5.9 iirfid

Example RFID reader integration. See iirfid/readme.md for details.

5.10 iotsaDiscovery

A plugin to discover devices based on the iotsa architecture and configure those devices (install certificates, install Igor capabilities, etc). The *NeoClock*, *Plant* and *Smartmeter_iotsa* devices below are examples of *iotsa* devices. User interface is provided.

5.11 kaku

Turns lights (or other devices) on and off using a KlikAanKlikUit device. See kaku/readme.md for details.

5.12 lan

Determines whether a local (or remote) internet service is up and running. See lan/readme.md for details.

5.13 lastFileAccess

Determine when a file was last modified or accessed (for example to check when some program was last used).

5.14 lcd

Displays messages on an LCD display. See lcd/readme.md for details.

5.15 logparse

Incomplete.

5.16 neoclock

Driver for NeoClock internet-connected clock. See neoclock/readme.md for details.

5.17 netatmo

Driver for NetAtmo weather stations. See netatmo/readme.md for details.

5.18 philips

Example of controlling a Philips television. See philips/readme.md for details.

5.19 plant

Move an internet-connected plant up and down. See plant/readme.md for details.

5.20 say

Speaks messages to the user. See say/readme.md for details.

5.21 smartmeter_iotsa

Reads current energy use in the household using a iotsa SmartMeter reader. See smartmeter_iotsa/readme.md for details.

5.22 smartmeter_rfduino

Older Bluetooth-based version of *smartmeter___iotsa*, reads current energy use in the household using a RFDuino-based SmartMeter reader.

5.23 systemHealth

Collects error messages from `services/*`, `devices/*` and `sensors/*` and stores these in `environment/systemHealth`. See systemHealth/readme.md for details.

5.24 testPlugin

A Python-based plugin that simply shows all the arguments it has been passed. This can be used as the starting point for a python-based plugin.

5.25 timemachine

Checks last time that Apple Time Machine backed up specific machines. See timemachine/readme.md for details.

5.26 user

Low-level interface to add and delete users, including capabilities and such, and change passwords. User interface is provided.

5.27 watchdog

Reboots the host system in case of problems. See watchdog/readme.md for details.

5.27.1 action - manage igor actions

This plugin is an absolutely minimal user interface to allow inspection of the current set of Igor actions, and adding, deleting and changing those actions.

You should be logged in as *admin* to use it.

5.27.2 ble - Bluetooth LE plugin

Listens to Bluetooth Low Energy advertisements to determine which devices are in range.

requirements

Only tested on a Raspberry PI with an external Bluetooth LE USB dongle and the Bluez bluetooth stack.

Requires the *bleServer*, see `../../../helpers`.

Stores raw data (mac-address, last time seen) data for all devices in `sensors/ble`, and current availability by name for specific "known" devices (programmable via `plugindata/bleDevices`) into `environment/devices`.

schema

- `sensors/ble`: Stores per-device entries exactly as they are read from *bleServer*.
- `environment/devices`: Stores availability of known devices.
- `status/devices/ble`: Device status updated on every action.
- `plugindata/bleDevices/bleDevice`: Maps hardware address to name, for known devices (which will be stored in `environment/devices`:
  - `id`: Mac address (string, 6 colon-separated hex bytes).
  - `name`: User-visible name.

actions

Two actions:

- Pull data from *bleServer* into `sensors/ble` every 60 seconds.
- Update `environment/devices` for known BLE devices.

5.27.3 buienradar - buienradar.nl plugin

Stores rain forecast for the coming hour, from data provided by buienradar.nl.

requirements

GPS location in `environmnt/location` is used to get rain forecast data, and this location must probably be in the Netherlands.

schema

- `sensors/buienradar/lastupdate`: timestamp of last update.
- `sensors/buienradar/data`: one 5-minute rain forecast. Multiple elements can exist, they should be ordered temporally. Fields:
  - `mm`: amount of rain expected, in millimeters (float)
  - `level`: logarithmic value of above, see buienradar API for details (integer)
  - `time`: date and time for which this measurement is valid (isotime)
  - `hour`: hour of `time` (int)
  - `minute`: minute of `time` (int)
- `plugindata/buienradar`: GPS location for which to present forecast.

actions

Three actions:

- Use buienradar.nl API to fill `sensors/buienradar` every 5 minutes.
- Copy `environment/location/lon` to `plugindata/buienradar/lon` on Igor start and whenever it changes.
- Copy `environment/location/lat` to `plugindata/buienradar/lat` on Igor start and whenever it changes.

5.27.4 ca - Certificate Authority

Igor includes a certificate authority implementation that allows local use of SSL communication, and thereby *https:* access to Igor and devices without using self-signed certificates.

By installing the Igor root certificate on all machines that need access to Igor (or devices) all https accesses become transparent to the user (and to automatic scripts, and Igor itself).

the *ca* plugin is only a frontend (and UI) to the *igorCA* command line tool. In turn, that tool is only a front end op the *openssl* command which does all the heavy lifting of generating keys, certificate signing requests, certificates, etc.

schema

- `plugindata/ca/ca`: if non-empty this should be the URL for the Igor that this Igor uses as its CA. If empty (or if it doesn't exist) this Igor is its own CA.

Using Igor ca certificates

This section assumes the CA is already initialized and Igor is configured for using a ca-signed certificate (see below for instructions).

To install the root certificate chain on your machine download it through Igor, from */plugin/ca/root*. This may be accessible as http://igor.local:9333/plugin/ca/root or https://igor.local:9333/plugin/ca/root. In the second case you will have to accept the Igor certificate once (because it isn't trusted yet, because you have not installed the root certificate yet).

MacOS

On MacOS, open the certificate in *Keychain Access*. Add it to the *login* keychain or the *System* keychain (the latter installs it for all users on the system). Select the newly-installed certificate, open it, open the *Trust* section and set *X.509 Basic Policy* to *Always Trust*.

Linux

On Linux, rename the certificate so that it has a *.crt* extension. Copy it into */usr/share/ca-certificates*. Run *update-ca-certificates*. The latter two steps may have to be done using *sudo*. Maybe (unsure when) you have to run `dpkg-reconfigure ca-certificates` in stead of the update command and select your new certificate.

Windows

To be provided.

Checking that it worked

After installing the certificate you can check that it worked by pointing your browser at *https://igor.local:9333* or (even better, because you previously manually trusted the igor certificate) use some other tool that uses SSL access, for example

```
curl https://igor.local:9333
```

Initializing the ca

To initialize the Certificate Authority run

```
igorCA initialize
```

This creates a root key and certificate (you get to supply all the details such as country and organization and such). The root key is protected by a password (which you supply). After the initialize command is finished you can remove the root key and certificate (from *~/.igor/ca/root*) and keep them offline, if you want, for added security. You only need them again if your system has been compromised.

Next it creates an intermediate key and certificate (supply identical details as for the root) and signs it with the root key. This intermediate key is not password-protected, and will be used for normal signing operations. The root and intermediate certificates are concatenated for installation in other systems (see previous section) so that trust can be established.

All infrastructure is kept in *~/.igor/ca*. Revocation is not implemented yet but possible.

Enabling https for Igor

After the CA has been initialized the following command will create a key and certificate for Igor itself:

```
igorCA self
```

This creates a private key *~/.igor/igor.key* and a Certificate Signing Request *~/.igor/igor.csr* and uses the CA to sign that, giving a certificate *~/.igor/igor.crt*.

At startup, the existence of *~/.igor/igor.key* and *~/.igor/igor.crt* will cause Igor to serve on *https* in stead of *http* (but still on the default *9333* port).

Enabling https for other services

To create a secret key and certificate for a service *foo* (name for your personal enjoyment only) listening to addresses *foo.local* and *192.168.4.1* (names externally visible) run the following command in some temporary directory:

```
igorCA gen foo foo.local 192.168.4.1
```

This creates a number of files in the current directory: *foo.key* and *foo.crt* are the key and certificate. Copy these to the service. The other files (*foo.csr* and *foo.csrconfig*) are temporary files.

Using igorCA on a different machine

The `igorCA` command will normally use local filesystem access to obtain keys and certificates for signing, but it can also use the *ca plugin* as an intermediate to do the actual signing.

The command line to sign a key for a local igor server, using a master igor as the CA:

```
igorCA --remote --url https://masterigor.local:9333/data/ self
```

And the command line to sign a key for a service *foo* using the default Igor:

```
igorCA --remote gen foo foo.local 192.168.4.1
```

5.27.5 device - manage igor devices

This plugin is primarily a set of user interfaces to allow you to inspect, add and delete devices and sensors. You should probably be logged in as *admin*.

The device listing has links to the corresponding per-device data and plugin data, and to any user interface the device plugin provides.

When you install a new device this plgugin will install the corresponding plugins, create capabilities and secret shared keys and actions.

5.27.6 dhcp - Device availability based on wifi

Queries database of DHCP server on local machine and stores all active dhcp leases. Updates human-visible devices whenever a known device has a lease.

requirements

The plugin parses the internal files of the Linux DHCP service, therefore it only works if the DHCP server runs on the same machine as Igor.

schema

- `environment/devices`: Updated with known devices, as they are available.
- `devices/dhcp/lease`: information about a single DHCP lease:
  - `hardware`: Hardware MAC-address (string, 6 colon-separated hex bytes).
  - `ip-address`: IP address (string).
  - `client-hostname`: Name of the device, if known to DHCP (string)
  - `arp`: Boolean, true if IP-address is in the ARP-cache.
  - `ping`: Boolean, true if device reacts to ping.
  - `alive`: Boolean, true if either *arp* or *ping* (or both) is true.
- `plugindata/wifiDevices/wifiDevice`: MAC-address to name mapping for a single device:
  - `id`: MAC-address (string, 6 colon-separated hex bytes).
  - `name`: user-visible name.

actions

Three actions:

- Every 60 seconds the DHCP database is parsed and `sensors/dhcp` is filled.
- When a known device availability changes `environment/devices` is updated.

5.27.7 editData - edit entries in the database

This plugin is an absolutly minimal data editor. It allows you to inspect any item in the database (in rather unreadable XML form) and to update it.

You can only view and edit items for which you have permission, so you may have to login as *admin*.

You cannot edit items that have any hidden content (such as capabilities or plugin ownership annotations) because these would get lost. If you really need to edit such subtrees you should stop Igor and edit the database with a text editor.

5.27.8 fitbit - get health data from fitbit cloud service

Retrieves health data from Fitbit devices using the Fitbit cloud API and stores this in `sensors/fitbit`.

It is an example of a plugin using OAuth2 to retreive data from a web service.

It has an underscore before its name because otherwise there would be a name clash with the underlying Python module that is used to obtain the Fitbit data (sigh).

requirements

You should own a Fitbit device. Only the Fitbit Aria scale has been tested.

Visit the page `/plugin/fitbit/page/setup.html` to register your instance of Igor with the *fitbit.com* cloud API.

Manual registration

If the *setup.html* page does not work you can do manual setup. You should register your Igor as an application at https://dev.fitbit.com/apps/new. You will probably have to register yourself as a Fitbit developer before you can do this. There is one parameter that you need to specify on the registration page that is crucial you get right: the *Callback URL*. This is where an Igor user is redirected to after giving permission and must be correct. If your Igor runs on `igor.local` port `9333` the URL you must use is `http://igor.local:9333/plugin/fitbit/auth2`.

The fitbit registration of your application gives you a `client_id_` and `client_secret` that you enter into the plugin data and that identify your Igor to Fitbit.

per-user requirements

For each Fitbit user that is also an Igor user visit the *setup.html* page and create the per-user entry. Then add the *action* (see below) to pull data for that user into Igor.

Manual user setup

For each Fitbit user *yournamehere* you first create empty entries `sensors/fitbit/yournamehere` and `identities/yournamehere/plugindata/fitbit`.

You now take the following steps to give Igor permission to fetch your data:

- You use a normal browser to visit http://igor.local:9333/plugin/fitbit/auth1?user=yournamehere.
- Your browser will be redirected to the Fitbit website, where you are asked for permission to give Igor access to your health data. You log in with your Fitbit username and password and give permission.
- Your browser will be redirected back to Igor, to the *Callback URL* specified previously, with some secret data.
- Igor will contact the Fitbit website with the secret data to obtain your access tokens.
- These access tokens are stored in `identities/yournamehere/plugindata/fitbit`.

If all this worked Igor can now get the health data for *yournamehere*. You can add the *action* to fetch the data regularly.

plugin entrypoints

- `/plugin/fitbit?user=yournamehere` Obtain health data for user *yournamehere*, refreshing the access tokens as needed. Optional parameters:
  - `methods` a comma-separated list of methods to call (default: `get_bodyweight`). See Python Fitbit documentation for the details.
  - All other keyword arguments are passed to each of the methods in turn.
- `/plugin/fitbit/auth1?user=yournamehere` Start the authentication process for user *yournamehere* to enable Igor to obtain the health data.
- `/plugin/fitbit/auth2?code=...&state=yournamehere` Second step in the authentication process, called through browser redirection by Fitbit.
- `/plugin/fitbit/settings` and `/plugin/fitbit/userSettings` are internal calls to implement the actions in *setup.html*.

schema

- `plugindata/fitbit` Identity of this Igor for the Fitbit cloud service:
  - `client_id` the identity of your application (Igor)
  - `client_secret` the password of your application (Igor)
  - `system` Should be `en_GB` for imperial values, and (for example) `nl_NL` for metric.
- `identities/_yournamehere_/plugindata/fitbit` Fitbit access data for person *yournamehere*:
  - `token` Fitbit token. Most important fields are `refresh_token` and `access_token`.
  - `methods`, `resource`, `period` and other keyword arguments can be specified, these will be used as defaults when `/plugin/fitbit` is called for this user.
- `sensors/fitbit/_yournamehere_` Fitbit measurements for person *yournamehere*:
  - `body-weight` Weight data time series:
    - `value` The value of this measurement
    - `dateTime` For which date the value is.
  - ...

actions

One for each user for which Igor needs to fetch Fitbit health data:

```
<action>
        <interval>36000</interval>
        <url>/plugin/fitbit?user=yournamehere</url>
        <representing>sensors/fitbit</representing>
</action>
```

5.27.9 home - some actions that make sense in a home setting

This plugin is mainly there to show *"it can be done"*. It is a plugin with only a database fragment, without any Python or shell script implementation.

It contains a couple of actions that make sense in a home setting.

actions

- *checkNight* updates the variable `environment/night` to reflect whether it is day or night. As included night starts at 23:00 and ends at 08:00.
- *updatePeople* updates the `people/_name_/home` boolean whenever a device owned by person `_name_` shows up or disappears in `environment/devices`.
- *countPeopleAtHome* updates `environment/people/count` to reflect how many people are currently considered to be at home.

5.27.10 homey - integration with the Homey device

The Athom Homey is a device with a somewhat similar goal to Igor: connecting all sorts of IoT devices in the home (and external services as well).

Homey and Igor take different approaches, but offer ample opportunity for integration: both offer ReST interfaces and both have a plugin model that allows control of external devices.

At the moment the Igor homey plugin does one thing: it allows reading of RFID tags using the Homey RFID reader.

Further documentation is to be provided.

5.27.11 iirfid - read RFID tags

This plugin reads RFID tags (mifare tags, specifically) and stores all presented tags in `sensors/rfid`, sequentially. Tags that are known (by hardware address) are also stored in `sensors/tags`.

The intention of this separation is that a *tag* is a more high-level concept (such as "Jacks keyring") that is used to trigger further actions. Tags using other methods than RFID (for example barcodes or QR-codes) would be merged here with the RFID tags, so the concept of a *tag* becomes *a thing that can trigger actions when presented to a suitable reader*.

requirements

The main requirement is an RFID-reader that connects to a computer using an Arduino and a USB interface. It actually requires a very specific RFID-reader of which we happen to have two lying around at CWI (constructed by Interactive Institute for a previous project), which no-one else probably has, so this plugin is more an example than anything else.

To use this plugin as the basis for support of another RFID reader: modify the script `iirfid/scripts/rfidread.py` to support the reader you have.

schema

Some of the data is unique to the iirfid plugin, some other data (with *tags* or *rfid* in the name in stead of *iirfid*) is shared with other rfid reader plugins.

- `sensors/iirfidrfid`: Collects hardware unique IDs (in order of the time the tags were presented) in sub-elements:
  - `rfidtag`: 4 colon-separated hex bytes.
- `sensors/tags`: Collects "known" tags in order presented in sub-elements:
  - `tag`: human-readable name.
- `plugindata/iirfid/host`: if non-empty, host to which the RFID reader is connected (must be reachable with *ssh* without password). String.
- `plugindata/iirfid/serial`: Serial device to which RFID reader is connected. String.
- `plugindata/iirfid/baud`: Baud rate. Integer.
- `plugindata/rfid/tag`: Maps mifare tag unique IDs to human-readable names:
  - `id`: 4 colon-separated hex bytes.
  - `name`: human readable name.

actions

- *start*: starts the rfid reader asynchronously, telling it to POST each tag presented to this Igor, `sensors/rfid/rfidtag`.
- *cleanup*: remove all entries in `sensors/rfid` and `sensors/tags` except the last one.

internal actions

- When a new raw tag is entered into `sensors/iirfidrfid` look it up in `plugindata/rfid` and post the user-friendly name in `sensors/tags`.

5.27.12 iotsa433Device - plugin for iotsa 433MHz home automation bridge

This plugin interfaces to a Iotsa 433MHz home automation sender/receiver.

The rest of the information in this readme is wrong:-)

schema

- `devices/iotsaDevice/target`: data to be copied to the device.
- `devices/iotsaDevice/current`: data read from the device.

actions

- Whenever `devices/iotsaDevice/target` changes this change is forwarded to the device.
- Whenever `/action/pull-iotsaDevice` is called the device data is copied to `current`.

Notes

- Usually you will install this plugin with a different name, and specifically a name that matches the `.local` hostname of the device.
- You will probably have to edit `pluginData/iotsaDevice` manually to set the API endpoint and protocol. An installation user interface (like for other plugins) is still missing.
- If you have a multifunction iotsa device, with multiple API endpoints, it is better to install multiple *iotsaDevice* plugins, one for each API endpoint, and edit the `pluginData` manually to set correct hostname and endpoint.

5.27.13 iotsaDevice - basic plugin for iotsa device

This plugin interfaces to a Iotsa device. Iotsa devices are small internet-enabled appliances, see https://github.com/cwi-dis/iotsa for examples, the source code for the server, as well as schematics and 3D-models of the hardware needed.

schema

- `devices/iotsaDevice/target`: data to be copied to the device.
- `devices/iotsaDevice/current`: data read from the device.

actions

- Whenever `devices/iotsaDevice/target` changes this change is forwarded to the device.
- Whenever `/action/pull-iotsaDevice` is called the device data is copied to `current`.

Notes

- Usually you will install this plugin with a different name, and specifically a name that matches the `.local` hostname of the device.
- You will probably have to edit `pluginData/iotsaDevice` manually to set the API endpoint and protocol. An installation user interface (like for other plugins) is still missing.
- If you have a multifunction iotsa device, with multiple API endpoints, it is better to install multiple *iotsaDevice* plugins, one for each API endpoint, and edit the `pluginData` manually to set correct hostname and endpoint.

5.27.14 iotsaDiscovery - discover and initialize iotsa devices

This plugin is primarily a set of user interfaces to discover and initialize new iotsa-based devices (see https://github.com/cwi-dis/iotsa for examples).

The main UI page allows discovering uninitialized iotsa devices (which setup a private WiFi network with a network name that follows a known pattern) and initialized iotsa devices on the WiFi network (which advertise their existence through mDNS/Bonjour/Rendezvous).

You can then select a device and examine what sort of device it is, and what services it provides. You can also change the configuration of the device.

Specifically, iotsaDiscovery helps you to install a key and certificate into the device (so HTTPS works without warning messages about unknown certificates). And it helps you to install the Igor Issuer shared secret key, so that the device knows it can trust Igor capabilities.

5.27.15 kaku - control KlikAanKlikUit devices

Turns lights (or other devices) on and off using a KlikAanKlikUit device. Should be fairly easy to adapt for other vendors, as long as a computer-connectable controller exists.

requirements

- A KAKU USB controller (and some KAKU outlets)
- The KAKU helper program from `../../helpers/kaku`, which in turn requires *pyusb*.

schema

- `environment/switch`: Contains boolean elements stating which switches should be on and off.
- `plugindata/kaku/host`: the hostname or IP address to wich the USB controller is connected, and on which the helper program should be installed (string, default localhost). Must be reachable with *ssh* without password.
- `plugindata/kaku/switch`: Entry mapping switch names to ids. Can occur multiple times:
  - `name`: human readable name (string).

- `id`: KAKU switch number (integer).

actions

Whenever the value of any element in `environment/lights` changes the element name is looked up on `plugindata/kaku/switch` and the corresponding switch is turned on or off.

5.27.16 Kodi - control Kodi media player

This plugin controls a Kodi entertainment system. It allows controlling playback of media items and such, and reports status information on when Kodi is currently playing. Uses the Kodi JSONRPC interface over HTTP.

There is no support for

schema

- `environment/mediaPlayback`: Information on what audio/video/TV/Music is currently being playing
- `devices/kodi/current`: current status of Kodi, what it is currently doing
- `devices/kodi/target`: commands for Kodi, what it should be doing
- `plugindata/kodi/url`: URL to access Kodi JSONRPC end point

requirements

Use of this plugin requires access to a Kodi media player.

actions

- Periodically retreive Kodi current status to `devices/kodi/current`
- Transmit commands to Kodi whenever `devices/kodi/target` changes
- Update `environment/mediaPlayback` from `devices/kodi/current`

5.27.17 lan - test availability of services

The lan plugin, which is badly named, tests availability of internet services. It works just as well for services on the LAN as for services out on the internet.

When the plugin runs, usually from a timed action in `actions`, it accepts a number of parameters as URL query parameters. It attempts to open a tcp-connection to a service on a given host/port combination, and returns either `ok` or an error message. The plugin also sets the status of the service in `status/services/_name_/alive` and others based on whether this is successful or not.

The following parameters are accepted:

- `name`: name of the service (string).
- `ip`: hostname or ip-address to contact (string). Default is to use the value of *name*.
- `port`: port to contact (integer), default 80.
- `timeout`: how many seconds to wait for the connection to be established before giving up (integer), default 5.

- `url`: alternative to *ip/port/timeout*, try to connect to a http[s]-based service.
- `service`: where status report will be posted. Default is `services/%s`, where the `name` is filled in for the `%s`.

schema

For each service *name* there will be an entry `/data/status/services/_name_` which will be filled with with the following entries by the corresponding action:

- `alive`: a boolean telling whether the service is alive (and working correctly)
- `lastSuccess`: a timestamp telling when the service was most recently contacted successfully
- `lastFailure`: a timestamp telling when the service was most recently not contacted successfully
- `lastActivity`: a timestamp telling when the plugin was run most recently
- `errorMessage`: if this exists it is a string explaining in human-friendly language what is wrong with the service.

user interface

There is a user interface at */plugin/lan/page/index.html* that allows inspecting/adding/deleting lan plugin watchers. It also has some suggested actions to add, to test internet health and Igor health.

5.27.18 lcd - show messages on an LCD display

This plugin is normally called in an *action* that is triggered whenever a new `environment/messages/message` appears. It displays that message on an external LCD display.

See the say plugin for an alternative way to present messages to the user (by speaking them).

requirements

To use the plugin as-is (as opposed to using it as sample code for other ways to display messages) requires a command line tool `lcdecho`. This tool is available as part of the [iotsa Display Server], *URL to be provided later*.

actions

One action, to display the content of any element inside `environment/messages` whenever it changes.

5.27.19 neoclock - use a NeoClock for showing status information

A NeoClock is a clock with 60 LEDs that shows the time, but additionally can show status information and alerts through a REST-like web service. It is built around an esp8266-based microprocessor board with NeoPixel LEDs.

Instructions for building (hardware and software) can be found at https://github.com/cwi-dis/iotsaNeoClock.

This plugin uses the NeoClock to show three types of status information:

- *Igor status*: if Igor does not update the NeoClock status for 5 minutes (for example because Igor has crashed) the inner LED circle will light up orange.

- *System status*: if any messages have been deposited in `environment/systemHealth` the inner LED circle will light up red.
- *Rain forecast*: the outer circle of LEDs will show rain intensity for the coming hour (if the buienradar plugin is also installed).

schema

- `environment/systemHealth/messages`: if non-empty the neoClock will show a warning status
- `sensors/buienradar`: if this exists it will be used to fill the rain forecast status
- `plugindata/neoclock/host`: host name for the neoclock
- `plugindata/neoclock/protocol`: protocol for accessing the neoclock (`http`, `https` or `coap`)

requirements

Use of this plugin requires a [iotsa NeoClock], *URL to be provided later*.

actions

Two actions:

- Update neoclock *temporalStatus* (the outer ring) whenever `sensors/buienradar` changes.
- Update neoclock *status* (the inner ring) every minute, with color depending on the content of `environment/systemHealth/messages`, a timeout of 5 minutes and a timeout color of orange.

5.27.20 netatmo - get environmental data from a NetAtmo weather station

This plugin gets data from a NetAtmo weather station through their cloud API.

Raw data is stored in `sensors/netatmo` with one element per weather station (using the names the user assigned to the stations).

Some parsed weather data is stored in `environment/weather`, in a slightly ad-hoc way: inside temperature is taken from any station that has a wifi interface, outside temperature from any station with an RF interface.

The plugin could also be adapted to get data from other NetAtmo sensors.

requirements

- a NetAtmo weather station

You will need to register your Igor with the NetAtmo cloud service, to tell NetAtmo that it is okay that Igor gets your (possibly privacy-sensitive) weather data.

The *setup.html* page should help you doing this.

Manual registration

If the *setup.html* page does not work for you you can set registration up manually.

schema

- `plugindata/netatmo/authentication` Authentication parameters:
  - `clientId` String of hexadecimal digits identifying your Igor, from *dev.netatmo.com*.
  - `clientSecret` Longer string of alfanumeric characters identifying your Igor, from *dev.netatmo.com*.
  - `username` Your personal username, from *netatmo.com*.
  - `password` Your personal password, from *netatmo.com*.
- `sensors/netatmo/_stationname_` Data for one weather station (base station or extension)
- `environment/weather/tempInside` Current indoor temperature
- `environment/weather/tempOutside` Current outdoor temperature

5.27.21 passiveSensor - get sensor readings via REST

This plugin periodically polls a REST endpoint and saves the resulting readings in the Igor database.

As distributed it will read sensor data from BLE sensors, using the same REST service as the BLE plugin, but using different endpoints. For each sensor the most recent reading is remembered.

Currently the BLE server supports the following sensors:

- *nearable*: devices supporting the Estimote Nearable protocol.
- *ibeacon*: devices supporting the iBeacon protocol originally developed by Apple but now supported by many beacon vendors.
- *cwi_sensortag*: Texas Instrument CC2650 SensorTag devices running the CWI-DIS firmware (link to be provided later).

The intention is to install this plugin not under its standard name, but under the name of the sensor type (such as *ibeacon* or *nearable*). Sensors are identified by UUID, or BLE Mac address for *cwi_sensortag*.

requirements

Only tested on a Raspberry PI with an external Bluetooth LE USB dongle and the Bluez bluetooth stack.

Requires the *bleServer*, see `../../../helpers`.

schema

- `sensors/passiveSensor/lastActivity`: Time of the last reply of the REST service.
- `sensors/passiveSensor/_name_Device`: Occurs once per *name*-type sensor. Contains information such as *uuid* or *address* and all data the sensor advertises (temperature, accelerometer, rss, etc).
- `status/devices/passiveSensor`: Device status updated on every action.
- `plugindata/passiveSensor`:
  - `protocol`: Usually *http* or *https*.
  - `host`: where the REST service runs, usually *localhost*.
  - `port`: which port the REST service runs on.
  - `endpoint`: Rest of the REST URL after the initial /.

actions

One action:

- Pull data from the REST server into `sensors/passiveSensor` every 2 seconds.

5.27.22 philips - control a Philips TV set

This plugin controls a Philips TV set, allowing the TV to be turned on or off, channels to be switched, etc.

It is a proof of concept.

requirements

- Turning the TV on or off requires an infrared transmitter with a USB interface. The schematics and software for the specific transmitter used will be made available later.
- Switching channels, controlling volume and such require a Philips TV with the JointSpace features, which allow control over the TV through a REST API. This is supported on most Philips TVs produced between 2009 and 2014, see the JointSpace sourceforge page for details.
- Scripts *tvcmd.py* to control the IR transmitter and *philips.py* to use the REST api.

schema

- `devices/tv/power`: boolean, true if TV is on (or should be on).
- `devices/tv/volume`: integer, TV sound volume.
- many more...

actions

Three actions:

- Whenever `devices/tv/power` changes turn on or off the TV through the infrared transmitter.
- Whenever any other element in `devices/tv` changes forward that change to the TV through the REST interface.
- Every 60 seconds obtain the status of the TV and update the elements in `devices/tv`.

5.27.23 plant - control the position of an internet-connected plant

This plugin interfaces to a Iotsa MotorServer board. This board is a esp8266-based small REST-like web server that controls a stepper motor, thereby allowing the physicl position of something in the house to be controlled over the internet.

The MotorServer repository https://github.com/cwi-dis/iotsaMotorServer contains the source code for the server, as well as schematics and 3D-models of the hardware needed.

schema

- `environment/energy/electricity`: float, current electricity consumption in kW. Stored here by a plugin like smartmeter_iotsa.
- `devices/plant/target`: float, wanted cable length in millimeters.
- `devices/plant/pos`: float (read-only), current cable length in millimeters.
- `devices/plant/speed`: float (read-only), current motor speed.
- `devices/plant/inrange`: integer (read-only), nonzero if the cable is at an ultimate position, as indicated by the end stop microswitches.

actions

- every minute the current position (and speed, etc) of the plant is retrieved and stored in `devices/plant`.
- Whenever `environment/energy/electricity` changes and if `environment/night` is false `devices/plant/target` is updated. This action contains the formula to convert kW (unit of electricity use) to cm (unit of plant height).
- Whenever `devices/plant/target` changes this change is forwarded to the MotorServer.

5.27.24 say - speak messages aloud

This plugin uses a text-to-speech program to speak out any messages appearing in `environment/messages`. The TTS utility can be run on the Igor machine or on a machine that is accessible via *ssh* (without a password).

requirements

The plugin uses a command line utility `say`, as available on OSX. If `say` is not available it will try (in order) `espeak`, `flite` and `festival`, which are available on most Linux distributions.

schema

- `plugindata/say/voice`: string, the voice to use (default is to use the default voice). Not supported on Linux.
- `plugindata/say/remoteHost`: string, host on which to run the script (using *ssh*), default is the host Igor runs on. Only supported for OSX remote hosts.

actions

When any element inside `environment/messages` is modified (usually these are `message` elements) the contents are spoken.

5.27.25 smartmeter_iotsa - read current household energy use

This plugin interfaces to a *iotsa p1 reader* https://github.com/cwi-dis/iotsaSmartMeter. It consists of an esp8266 that interfaces to the standardised dutch smart meter p1 port, see DSMR v2.2.

It reads all information including current electricity and gas use, electricity delivered back to the net, total consumption, etc.

requirements

- A dutch smart energy meter.
- A iotsa p1 reader.

schema

- `sensors/smartMeter`: raw data read from the smart meter:
  - `timestamp`: Time of reading, as provided by the smart meter itself (isotime).
  - `meter_id_electricity` and `meter_id_gas`: unique IDs of the meters (string).
  - `current_kw`: Current electricity use in kW (float).
  - `total_kwh_tariff_1`: Total electricity use under normal tariff over meter lifetime (kWh, float).
  - `total_kwh_tariff_2`: Total electricity use under reduced (night) tariff over meter lifetime (kWh, float).
  - all of those are also available with `_returned` appended for delivery back to the net.
  - `current_tariff`: integer, 1 or 2 depending on day or night.
  - `total_power_failures`: total number of power failures over device lifetime (integer).
  - `total_power_failures_`: total number of long power failures over device lifetime (integer).
  - `total_gas_m3`: Total gas consumption over meter lifetime (float).
  - `unkown`: Other readings. This element has a `tag` attribute with the DSMR identifier and the data is a string with the unparsed DSMR content.
- `plugindata/smartmeter_iotsa/host`: string, host name or IP address of the iotsa p1 reader.
- `plugindata/smartmeter_iotsa/protocol`: protocol to access the iotsa p1 reader (`http`, `https` or `coap`).

internal actions

- Once every minute the smart meter is read and the data deposited into `sensors/smartMeter`.
- Whenever the data in `sensors/smartMeter` is changed `environment/energy/electricity` is updated.

5.27.26 systemHealth - look after health of services and other infrastructure

This plugin checks services, sensors and devices for error messages and aggregates these. It is triggered by the *systemHealth* action. It works together with the `/systemHealth.html` template document and the *neoclock* plugin (which can show that any important exceptional condition currently exists).

The sequence of steps that happens is as follows:

- A plugin such as *lan* detects that a service is unavailable.
- *systemHealth* registers this in `environment/systemHealth`.
- *neoclock* detects this and shows a moderately visible warning to the end user.
- The end user uses a web browser to visit `/systemHealth.html` to see what the problem is.

- The end user now has two options to make the visible warning go away:
  - Fix the problem.
  - Use one of the *ignore* buttons. The error will subsequently still be recorded but it will not trigger the *neoclock* visible warning for an hour or a day or so.

Each service (device, sensor) has two fields that are important. As an example, for service *internet*:

- `service/internet/errorMessage`: If this element exists and is non-empty it means the *internet* service has an error condition (and that that condition is serious enough that we want to inform the user about it). The message will be copied into `environment/systemHealth/messages/internet`.
- `service/internet/ignoreErrorUntil`: If this value (timestamp) is set and is in the past it will be deleted. If it is in the future, the `errorMessage` field will not be copied into `environment/systemHealth`, effectively ignoring the error condition for some period of time.

actions

The *systemHealth* action triggers this plugin.

5.27.27 timemachine - Time Machine backup status

This plugin check when the most recent backup has been made with Apple Time Machine. It uses the `tmutil` utility, so the igor user must have the right access privileges to run that.

The following parameters are accepted:

- `name`: name of the service (string). This is used as the name of the element in `services`. Default `backup`.
- `remoteHost`: hostname or ip-address where `tmutil` is run, using `ssh`. Default is to run locally.

The status of the backup (including any error messages) is stored in `/data/status/services/_name_`.

User Interface

There is a user interface page *index.html* that allows adding of Time Machine service watchers and deleting them again.

5.27.28 user - manage igor users

This plugin is primarily a set of user interfaces to allow you to add users, delete users and change passwords.

For adding and deleting users (and for changing passwords for anyone but yourself) you should be logged in as *admin*.

5.27.29 watchdog - reboot linux system in case of problems

Reboots the host system in case of problems.

A plugin that opens the Linux watchdog device `/dev/watchdog`. The parameter `timeout` specifies after how many seconds the watchdog will fire and reboot the system, unless the watchdog plugin is accessed again before that timeout. Can be used to make the Igor machine reboot when Igor hangs, or when anomalous conditions are detected (and there is reason to believe a reboot will resolve these issues:-).

CHAPTER 6

Command line utilities

Igor comes with a number of command line utilities to access and control the database. Most of these use a common configuration file `~/.igor/igor.cfg` and a number of environment variables to set defaults for arguments.

6.1 igorVar

igorVar is the main program used to access the database from the command line and from shell scripts. It allows getting and setting of database variables in various different formats (text, xml, json, python).

Access Igor home automation service and other http databases

```
usage: igorVar [-h] [--configFile FILE] [--config SECTION] [-u URL]
               [--verbose] [--bearer TOKEN] [--access TOKEN]
               [--credentials USER:PASS] [--noverify] [--certificate CERTFILE]
               [--noSystemRootCertificates] [-e] [-v VARIANT] [-M MIMETYPE]
               [--text] [--json] [--xml] [--python] [--pretty] [--delete]
               [--create] [--put MIMETYPE] [--post MIMETYPE] [--data DATA]
               [--checkdata] [--checknonempty] [-0]
               var
```

6.1.1 Positional Arguments

var

Variable to get (or put, or post). The variable is specified in XPath syntax. Relative names are relative to the toplevel `/data` element in the database. Absolute names are also allowed (so `/data/environment` is equivalent to `environment`).

Access to non-database portions of the REST API is allowed, so getting `/action/save` will have the side-effect of saving the database.

Full XPath syntax is allowed, so something like `actions/action[name='save']` will retrieve the definition of the *save* action.

For some XPath expressions, such as expressions with a toplevel XPath function, it may be necessary to pass the `--eval` switch.

6.1.2 Named Arguments

--configFile Get default arguments from ini-style config FILE (default: ~/.igor/igor.cfg)

--config Get default arguments from config file section [SECTION] (default: igor).

-u, --url Base URL of the server (default: "http://igor.local:9333/data")

Default: "http://igor.local:9333/data"

--verbose Print what is happening

--bearer Add Authorization: Bearer TOKEN header line

Default: ""

--access Add access_token=TOKEN query argument

Default: ""

--credentials Add Authorization: Basic header line with given credentials

Default: ""

--noverify Disable verification of https signatures

--certificate Verify https certificates from given file

Default: ""

--noSystemRootCertificates Do not use system root certificates, use REQUESTS_CA_BUNDLE or what requests package has

-e, --eval Evaluate XPath expression in stead of retrieving variable (by changing /data to /evaluate in URL)

-v, --variant Variant of data to get (or put, post)

-M, --mimetype Get result as given mimetype

--text Get result as plain text

--json Get result as JSON

--xml Get result as XML

--python Get result as Python (converted from JSON)

--pretty Pretty-print result (only for Python, currently)

--delete Delete variable

--create Create or clear a variable

--put PUT data of type MIMETYPE, from –data or stdin

--post POST data of type MIMETYPE, from –data or stdin

--data POST or PUT DATA, in stead of reading from stdin

--checkdata Check that data is valid XML or JSON

--checknonempty Check that data is valid XML or JSON data, and fail silently on empty data

-0, --allow-empty Allow empty data from stdin

Argument defaults can also be specified in environment variables like IGORSERVER_URL (for –url), etc.

Some examples may help understanding how *igorVar* can be used to access and modify your database. The following command will create a variable (in the `sandbox` section of the database, which is specifically there for experimenting), assuming it does not exist yet. It will print the full XPath of the variable created.

```
igorVar --put text/plain --data "Hello world" sandbox/helloworld
```

Running that command multiple times will overwrite the variable. Running the following command is completely equivalent:

```
echo "Hello World" | igorVar --put text/plain sandbox/helloworld
```

You can get the plaintext value of variable with either of the following commands (which are completely equivalent):

```
igorVar --mimetype text/plain sandbox/helloworld
igorVar --mimetype text/plain /data/sandbox/helloworld
```

The default mimetype, however, is `application/xml` because *igorVar* is primarily meant for use in scripts, not interactively by a human.

To delete a variable, if it exists, use

```
igorVar --delete sandbox/helloworld
```

Post is similar to put, but always create a new entry. So if you have no *helloworld* variable yet the following command line is equivalent to the first example. But if you run it three times you will have three variables, named `sandbox/helloworld[1]`, `sandbox/helloworld[2]` and `sandbox/helloworld[3]`:

```
igorVar --post text/plain --data "Hello world one" sandbox/helloworld
igorVar --post text/plain --data "Hello world two" sandbox/helloworld
igorVar --post text/plain --data "Hello world three" sandbox/helloworld
```

To retrieve one of these you *must* supply the index. Or use `--variant multi` but that only works when you want the value in JSON or XML:

```
igorVar --mimetype text/plain 'sandbox/helloworld[2]'
igorVar --mimetype application/json --variant multi sandbox/helloworld
```

Note that this `[2]` construct is an XPath expression. Read up on XPath, it is a very powerful but simple language to select variables in your database. Also note that many XPath operators are also magical to the shell, so put your variable name in single or double quotes when in doubt.

6.2 igorCA

igorCA is the command line interface to using Igor as a Certificate Authority. Under the hood it is implemented using the `openssl` command line tool. It is intended to serve as CA for the `.local` domain, to enable secure communication between local devices and Igor (and other local devices).

When used as a command line tool *igorCA* can also comunicate to another *igorCA* operating as a plugin in another Igor, thereby making it possible to run the CA only on a single machine in the local network, even if multiple Igor instances are used. This mode of operation is enabled by using the *–remote* argument to *igorCA*.

Igor Certificate and Key utility

```
usage: igorCA [-h] [--configFile FILE] [--config SECTION] [-u URL] [--verbose]
              [--bearer TOKEN] [--access TOKEN] [--credentials USER:PASS]
              [--noverify] [--certificate CERTFILE]
              [--noSystemRootCertificates] [-s BITS] [-r] [-d DIR]
              action [arguments [arguments ...]]
```

6.2.1 Positional Arguments

action Action to perform: help, initialize, ...

Default: "help"

arguments Arguments to the action

6.2.2 Named Arguments

--configFile Get default arguments from ini-style config FILE (default: ~/.igor/igor.cfg)

--config Get default arguments from config file section [SECTION] (default: igor).

-u, --url Base URL of the server (default: "http://igor.local:9333/data")

Default: "http://igor.local:9333/data"

--verbose Print what is happening

--bearer Add Authorization: Bearer TOKEN header line

Default: ""

--access Add access_token=TOKEN query argument

Default: ""

--credentials Add Authorization: Basic header line with given credentials

Default: ""

--noverify Disable verification of https signatures

--certificate Verify https certificates from given file

Default: ""

--noSystemRootCertificates Do not use system root certificates, use REQUESTS_CA_BUNDLE or what requests package has

-s, --keysize Override key size (default: 2048)

-r, --remote Use CA on remote Igor (default is on the local filesystem)

-d, --database (local only) Database and scripts are stored in DIR (default: ~/.igor, environment IGORSERVER_DIR)

Available commands: csrtemplate Return template config file for for openSSL CSR request current Print current certificate serial for given Common Name dn Return CA distinghuished name as a JSON structure gen Generate a a server key and certificate for a named service and sign it with the intermediate Igor CA key. genCRL Generate CRL in static/crl.pem getCRL Output the CRL (Certificate Revocation List) getRoot Returns the signing certificate chain (for installation in browser or operating system) help Show list of available commands initialize create CA infrastructure, root key and certificate and intermediate key and certificate list Return list of certificates signed (optional arguments restricts to single Common Name). revoke Revoke a certificate. Argument is the number of the certificate to revoke (see list). revokecn Revoke certificate for a given Common Name, if one exists self Create a server key and certificate

for Igor itself, and sign it with the intermediate Igor CA key selfCSR Create secret key and CSR for Igor itself. Outputs CSR. sign Sign a Certificate Signing Request and return the certificate. status Returns nothing if CA status is ok, otherwise error message

6.2.3 igorCA actions

`initialize` ***root-issuer intermediate-issuer*** Create CA infrastructure, root key and certificate and intermediate key and certificate.

The *root-issuer* argument is the name of this Certificate Authority, according to the X.509 standard. See <https://tools.ietf.org/rfc/rfc5280.txt> section 4.1.2.4 for exact details of the fields allowed, but some fields are required: *C* for country name, *ST* for state and *O* for organization. Adding *CN* for common name is probably a good idea, this will allow people on the local network to recognize the certificate issuer.

The fields are introduced by slashes, so the following is an example of what you could use as *root-issuer* after replacing the various fields with the correct values for your situation:

```
'/CN=root.ca.igor.local/C=NL/ST=Netherlands/O=Jack Jansen'
```

The *intermediate-issuer* is the same, but for the intermediate certificate. `CN=intermediate.ca.igor.local` is suggested.

`getRoot` Returns the signing certificate chain (for installation in browser or operating system).

`status` Returns nothing if CA status is ok, otherwise error message

`csrtemplate` Return template config file for openSSL CSR (Certificate Signing Request)

`dn` Return CA distinghuished name as a JSON structure

`gen` ***prefix name-or-ip*** **[...]** Generate a a server key and certificate for a named service and sign it with the intermediate Igor CA key. The *prefix* is used to generate the filenames where the results of this action are stored:

- *prefix*`.key` will contain the secret key for the service
- *prefix*`.crt` will contain the certificate for the service
- *prefix*`.csr` is a temporary file containing the CSR
- *prefix*`.csrConfig` is a temporary containing the openSSL configuration used to create the CSR

`sign` ***csrfile certfile*** Read a Certificate Signing Request from *csrfile* and sign it with the CA keys. Save the resulting certificate to `certfile`.

`selfCSR` ***name-or-ip*** **[...]** Create secret key and CSR (Certificate Signing Request) for Igor itself. Pass all DNS names (or IP addresses) for this Igor host as arguments. Outputs CSR.

`self` ***name-or-ip*** **[...]** Create a secret key and certificate for Igor itself, and sign it with the intermediate Igor CA key.

`revoke` ***number*** Revoke a certificate. Argument is the number of the certificate to revoke (can be obtained through the *list* action or by inspecting the certificate). Regenerates CRL as well.

`genCRL` Generate CRL (Certificate Revokation List) in `static/crl.pem` so it can be retrieved by other Igors.

`getCRL` Output the CRL (Certificate Revocation List), for example for use in browsers or in the operating system certificate support.

`list` Return list of certificates signed and certificates signed and subsequently revoked.

6.3 igorControl

igorControl allows some control over a running Igor, through the REST interface. All functions it allows can also be accessed through *igorVar* but *igorControl* provides a more convenient interface.

Control Igor home automation service

```
usage: igorControl [-h] [--configFile FILE] [--config SECTION] [-u URL]
                   [--verbose] [--bearer TOKEN] [--access TOKEN]
                   [--credentials USER:PASS] [--noverify]
                   [--certificate CERTFILE] [--noSystemRootCertificates]
                   action [NAME=VALUE [NAME=VALUE ...]]
```

6.3.1 Positional Arguments

action Action to perform: help, save, stop, restart, command, ...

NAME=VALUE Arguments to the action

6.3.2 Named Arguments

--configFile Get default arguments from ini-style config FILE (default: ~/.igor/igor.cfg)

--config Get default arguments from config file section [SECTION] (default: igor).

-u, --url Base URL of the server (default: "http://igor.local:9333/data")

Default: "http://igor.local:9333/data"

--verbose Print what is happening

--bearer Add Authorization: Bearer TOKEN header line

Default: ""

--access Add access_token=TOKEN query argument

Default: ""

--credentials Add Authorization: Basic header line with given credentials

Default: ""

--noverify Disable verification of https signatures

--certificate Verify https certificates from given file

Default: ""

--noSystemRootCertificates Do not use system root certificates, use REQUESTS_CA_BUNDLE or what requests package has

Argument defaults can also be specified in environment variables like IGORSERVER_URL (for –url), etc.

6.3.3 igorControl actions

version Show Igor version.

save Saves the database to the filesystem.

stop Gracefully stop Igor.

restart Attempt to gracefully stop and restart Igor.

log Show current igor log file.

dump Show internal run queues, action handlers and events.

fail Raises a Python exception (intended for testing only).

flush Wait until all currently queued urlCaller events have been completed (intended for testing only).

6.4 igorSetup

igorSetup is the utility to initialize an Igor installation on the current machine and control it from the command line. Unlike the other command line utilities this utility uses normal Unix/Linux filesystem access and process control, and it can therefore only be used on the machine that also runs Igor (and by a user that has the right Unix permissions).

```
usage:
Usage: igorSetup [options] command [command-args]

Initialize or modify igor database.
Note: use only on host where igorServer is hosted, and not when it is running.
```

6.4.1 Positional Arguments

action	Action to perform: help, initialize, ...
	Default: "help"
arguments	Arguments to the action

6.4.2 Named Arguments

-d, --database	Database and scripts are stored in DIR (default: ~/.igor, environment IGORSERVER_DIR)
-r, --run	Run any needed shell commands (default is to print them only)

6.4.3 igorSetup actions

`initialize` create empty igor database.

`runatboot` make igorServer run at system boot (Linux or OSX, requires sudo permission).

`runatlogin` make igorServer run at user login (OSX only).

`start` start service (using normal OSX or Linux commands).

`stop` stop service (using normal OSX or Linux commands).

`add` ***pathname*** **`[...]`** add plugin (copy) from given *pathname*. Only use this command while Igor is not running. Note that it is potentially dangerous to install an Igor plugin, especially if it comes from an unknown source: an Igor plugin currently has complete access to the Igor internals, and can therefore access any data or modify it, and probably also read or write files on your Igor host.

addstd ***name*[=*srcname*]*** **[...]** add standard plugin *srcname* (linked) with given *name*. Only use this command while Igor is not running. Using the *plugin.html* Igor interface is easier.

remove ***name*** **[...]** remove plugin *name*. Using the *plugin.html* Igor interface is easier.

list show all installed plugins. Using the *plugin.html* Igor interface is easier.

liststd list all available standard plugins. Using the *plugin.html* Igor interface is easier.

certificate ***hostname*** **[...]** create https certificate for Igor using Igor as CA (using the igorCA module). Only use this command while Igor is not running.

certificateSelfSigned ***subject hostname*** **[...]** create self-signed https certificate for Igor (deprecated, use `certificate` command in stead). Only use this command while Igor is not running.

edit stop, edit the database (using the `$EDITOR` program) and restart the service.

rebuild stop, rebuild and restart the service (must be run in source directory).

rebuildedit stop, edit database, rebuild and start the service (must be run in source directory).

6.5 Configuration file

igorVar, *igorCA* and *igorControl* all read default values for named arguments from a configuration file `~/.igor/igor.cfg`, section `[igor]` (but these are overridable through the `--configFile` and `--config` arguments).

The following *igor.cfg* file causes `igorVar` to access an Igor on machine *downstairs.local* and `igorVar --config upstairs` to access an Igor an machine *upstairs.local* with HTTPS certification turned off:

```
[igor]
url = https://downstairs.local:9333/data/
[upstairs]
url = https://upstairs.local:9333/data/
noverify = 1
```

6.6 Environment variables

igorVar, *igorCA* and *igorControl* can also get their default values for named arguments from environment variables. These environment variables start with `IGORSERVER_` followed by the upper-cased argument name. As an example, `IGORSERVER_URL` can be used to provide a default for the `--url` argument.

Values passed on the command line have the highest priority, then values in environment variables, then values read from the configuration file.

CHAPTER 7

Python modules

Igor comes with a number of Python extension modules to access and control the database. These modules actually also implement the functionality of the corresponding Igor command line tools.

7.1 igorVar

igorVar is the main module used to access the database from Python programs. It allows getting and setting of database variables in various different formats (text, xml, json, python).

exception igorVar.**IgorError**
Exception raised by this module when an error occurs

class igorVar.**IgorServer**(*url*, *bearer_token=None*, *access_token=None*, *credentials=None*, *certificate=None*, *noverify=False*, *printmessages=False*)
Main object used to access an Igor server.

The object is instantiated with parameters that specify how to contact Igor. After that it provides an interface similar to `requests` to allow REST operations on the Igor database.

Parameters

- **url** (*str*) – URL of Igor, including the /data/ bit. For example `https://igor.local:9333/data/`.
- **bearer_token** (*str*) – An Igor external capability that has been supplied to your program and that governs which access rights your program has. Passed to Igor in the http `Authorization: Bearer` header.
- **access_token** (*str*) – An alternative (and possibly less safe) way to pass an external capability to Igor, through an `access_token` query parameter in the URL.
- **credentials** (*str*) – A *username:password* string used to authenticate your program to Igor and thereby govern which access rights you have. Passed to Igor in the http `Authorization: Basic` header.

- **certificate** (*str*) – If the certificate of the Igor to contact is not trusted system-wide you can supply it here.
- **noverify** (*bool*) – If you want to use https but bypass certificate verification you can pass `True` here.
- **printmessages** (*bool*) – Be a bit more verbose (on stderr).

_action (*method*, *item*, *variant*, *format=None*, *data=None*, *datatype=None*, *query=None*)
Low-level REST interface to the database, can be used to do GET, PUT, POST, DELETE and other calls under program control.

Parameters

- **method** (*str*) – the REST method to call.
- **item** (*str*) – the XPath expression defining the item to operate on.
- **variant** (*str*) – Same as for *get()*.
- **format** (*str*) – Same as for *get()*.
- **data** (*str*) – new value for the item, same as for *put()*.
- **datatype** (*str*) – mimetype of the *data* argument, same as for *put()*.
- **query** (*dict*) – Same as for *get()*.

Returns The REST call result.

Raises *IgorError* – in case of both communication errors and http errors returned by Igor.

delete (*item*, *variant=None*, *format=None*, *query=None*)
Delete an item in the database.

Parameters

- **item** (*str*) – the XPath expression defining the item to delete.
- **variant** (*str*) – Same as for *get()* but not generally useful.
- **format** (*str*) – Same as for *get()* but not generally useful.
- **query** (*dict*) – Same as for *get()* but not generally useful.

Returns An empty string in case of success (or in case of the item not existing)

Raises *IgorError* – in case of both communication errors and http errors returned by Igor.

get (*item*, *variant=None*, *format=None*, *query=None*)
Get a value (or values) from the database.

Relative *item* names are relative to the toplevel `/data` element in the database. Absolute names are also allowed (so `/data/environment` is equivalent to `environment`).

Access to non-database portions of the REST API is allowed, so getting `/action/save` will have the side-effect of saving the database.

Full XPath syntax is allowed, so something like `actions/action[name='save']` will retrieve the definition of the *save* action. For XPath expressions matching multiple elements you must specify *variant='multi'*.

Parameters

- **item** (*str*) – The XPath expression defining the value(s) to get.
- **variant** (*str*) – An optional modifier to specify which data you want returned:

- `multi` allows the query to match multiple elements and return all of them (otherwise this is an error)
- `raw` also returns attributes and access-control information (for XML only)
- `multiraw` combines those two
- `ref` returns an XPath reference in stead of the value(s) found

- **format** (*str*) – The mimetype you want returned. Supported are:
  - `text/plain` plaintext without any structuring information
  - `application/xml` an XML string (default)
  - `application/json` a JSON string
- **query** (*dict*) – An optional http query to pass in the request. Useful mainly when accessing non-database entrypoints in Igor, such as `/action` or `/plugin` entries.

Returns The value of the item, as a string in the format specified by *format*.

Raises *IgorError* – in case of both communication errors and http errors returned by Igor.

post(*item*, *data*, *datatype*, *variant=None*, *format=None*, *query=None*)
Create an item in the database.

Even if *item* refers to an existing location in the database a new item is created, after all items with the same name.

Parameters

- **item** (*str*) – the XPath expression defining the item to delete.
- **data** (*str*) – new value for the item.
- **datatype** (*str*) – mimetype of the *data* argument:
  - `text/plain` plaintext without any structuring information (default)
  - `application/xml` an XML string
  - `application/json` a JSON string
- **variant** (*str*) – Same as for *get()* but not generally useful.
- **format** (*str*) – Same as for *get()* but not generally useful.
- **query** (*dict*) – Same as for *get()* but not generally useful.

Returns The XPath of the element created.

Raises *IgorError* – in case of both communication errors and http errors returned by Igor.

put(*item*, *data*, *datatype*, *variant=None*, *format=None*, *query=None*)
Replace or create an item in the database.

If *item* refers to a non-existing location in the database the item is created, if the item already exists it is replaced. It is an error to refer to multiple existing items.

Parameters

- **item** (*str*) – the XPath expression defining the item to delete.
- **data** (*str*) – new value for the item.
- **datatype** (*str*) – mimetype of the *data* argument:
  - `text/plain` plaintext without any structuring information (default)

- `application/xml` an XML string
- `application/json` a JSON string

- **variant** (*str*) – Same as for *get()* but not generally useful.
- **format** (*str*) – Same as for *get()* but not generally useful.
- **query** (*dict*) – Same as for *get()* but not generally useful.

Returns The XPath of the element created (or modified).

Raises *IgorError* – in case of both communication errors and http errors returned by Igor.

7.2 igorCA

igorCA is the Python interface to using Igor as a Certificate Authority. It can run all needed `openssl` commands locally, but will usually be used to communicate with another *igorCA* embedded in Igor through the REST interface.

class igorCA.**IgorCA**(*argv0*, *igorServer=None*, *keysize=None*, *database=None*)

Interface to Certificate Authority for Igor.

Parameters

- **argv0** (*str*) – program name (for error messages and such)
- **igorServer** (*str*) – optional URL for Igor server to use as CA (default: use local CA through openSSL commands)
- **keySize** (*int*) – default keysize (default default: 2048 bits)
- **database** (*str*) – for local CA: the location of the Igor database (default: ~/.igor)

do_csrtemplate()

Return template config file for for openSSL CSR request

do_dn()

Return CA distinghuished name as a JSON structure

do_genCSR(*keyFile*, *csrFile*, *csrConfigFile*, *allNames=[]*, *keysize=None*)

Create key and CSR for a service. Returns CSR.

do_getRoot()

Returns the signing certificate chain (for installation in browser or operating system)

do_list(*cn=None*)

Return list of certificates signed.

do_revoke(*number*)

Revoke a certificate. Argument is the number of the certificate to revoke (see list).

do_signCSR(*csr*)

Sign a CSR. Returns certificate.

do_status()

Returns nothing if CA status is ok, otherwise error message

7.3 igorServlet

This module allows you to easily create a REST microservice that supports Igor capability-based access control. You create an *IgorServlet* object passing all the parameters needed to listen for requests and check capabilities.

You supply callback methods for the REST endpoints and start the service.

Then, as requests come in, capabilities are checked, the REST parameters are decoded and passed as arguments to your callback methods, the return value of your callback is optionally JSON-encoded and sent back to the caller.

Intended use is that the requests come from the Igor server, and the microservice implements a device or sensor (or group of sensors), or is used to check the status of a service.

class igorServlet.**IgorServlet**(*port=8080*, *name='igorServlet'*, *nossl=False*, *capabilities=None*, *noCapabilities=None*, *database='.'*, *sslname='igor'*, *nolog=False*, *audience=None*, *issuer=None*, *issuerSharedKey=None*, ***kwargs*)

Class implementing a REST server for use with Igor.

Objects of this class implement a simple REST server, which can optionally be run in a separate thread. After creating the server object (either before the service is running or while it is running) the calling application can use `addEndpoint()` to create endpoints with callbacks for the various http(s) verbs.

The service understands Igor access control (external capabilities and issuer shared secret keys) and takes care of implementing the access control policy set by the issuer. Callbacks are only made when the REST call carries a capability that allows the specific operation.

The service is implemented using *Flask* and *gevent.pywsgi*.

The service can by used by calling `run()`, which will run forever in the current thread and not return. Alternatively you can call `start()` which will run the service in a separate thread until `stop()` is called.

Parameters

- **port** (*int*) – the TCP port the service will listen to.
- **name** (*str*) – name of the service.
- **nossl** (*bool*) – set to `True` to serve http (default: serve https).
- **capabilities** (*bool*) – set to `True` to enable using Igor capability-based access control.
- **noCapabilities** (*bool*) – set to `True` to disable using Igor capability-based access control. The default for those two arguments is currently to *not* use capabilities but this is expected to change in a future release.
- **database** (*str*) – the directory containing the SSL key *sslname*`.key` and certificate *sslname*`.crt`. Default is `'.'`, but `argumentParser()` will return `database='~/.igor'`. This is because certificates contain only a host name, not a port or protocol, hence if `IgorServlet` is running on the same machine as Igor they must share a certificate.
- **sslname** (*str*) – The name used in the key and certificate filename. Default is `igor` for reasons explained above.
- **nolog** (*bool*) – Set to `True` to disable *gevent.pywsgi* apache-style logging of incoming requests to stdout
- **audience** (*str*) – The `<aud>` value trusted in the capabilities. Usually either the hostname or the base URL of this service.
- **issuer** (*str*) – The `<iss>` value trusted in the capabilities. Usually the URL for the Igor running the issuer with `/issuer` as endpoint. Can be set after creation.
- **issuerSharedKey** (*str*) – The secret symmetric key shared between the audience (this program) and the issuer. Can be set after creation.

Note that while it is possible to specify `capabilities=True` and `nossl=True` at the same time this is not a good idea: the capabilities are encrypted but have a known structure, and therefore the *issuerSharedKey* would be open to a brute-force attack.

addEndpoint(*path*, *mimetype='application/json'*, *get=None*, *put=None*, *post=None*, *delete=None*)
Add REST endpoint.

Use this call to add an endpoint to this service and supply the corresponding callback methods.

When a REST request is made to this endpoint the first things that happens (if capability support has been enabled) is that a capability is carried in the request and that it is valid (using *audience*, *issuer* and *issuerSecretKey*). Then it is checked that the capability gives the right to execute this operation.

Arguments to the REST call (and passed to the callback method) can be supplied in three different ways:

- if the request carries a URL query the values are supplied to the callback as named parameters.
- otherwise, if the request contains JSON data this should be an object, and the items in the object are passed as named parameters.
- otherwise, if the request contains data that is not JSON this data is passed (as a string) as the `data` argument.

Parameters

- **path** (*str*) – The new endpoint, starting with `/`.
- **mimetype** (*str*) – How the return value of the callback should be encoded. Currently `application/json` and `text/plain` are supported.
- **get** – Callback for GET calls.
- **put** – Callback for PUT calls.
- **post** – Callback for POST calls.
- **delete** – Callback for DELETE calls.

static argumentParser(*parser=None*, *description=None*, *port=None*, *name=None*)
Static method to create ArgumentParser with common arguments for IgorServlet.

Programs using IgorServlet as their REST interface will likely share a number of command line arguments (to allow setting of port, protocol, capability support, etc). This method returns such a parser, which will return (from `parse_args()`) a namespace with arguments suitable for IgorServlet.

Parameters

- **parser** (*argparse.ArgumentParser*) – optional parser (by default one is created)
- **description** (*str*) – description for the parser constructor (if one is created)
- **port** (*int*) – default for the `--port` argument (the port the service listens to)
- **name** (*str*) – name of the service (default taken from `sys.argv[0]`)

Returns A pre-populated `argparse.ArgumentParser`

hasIssuer()
Return `True` if this IgorServlet has an issuer and shared key

run()
Run the REST service.

This will start serving and never return, until `stop()` is called (in a callback or from another thread).

setIssuer(*issuer*, *issuerSharedKey*)
Set URL of issuer trusted by this service and shared symmetric key.

If the issuer and shared key are not known yet during IgorServlet creation they can be set later using this method, or changed.

Parameters

- **issuer** (*str*) – The <iss> value trusted in the capabilities. Usually the URL for the Igor running the issuer with /issuer as endpoint.
- **issuerSharedKey** (*str*) – The secret symmetric key shared between the audience (this program) and the issuer.

stop()
Stop the REST service.

This will stop the service and join() the thread that was running it.

7.3.1 argumentParser arguments

If you use *igorServlet.IgorServlet.argumentParser()* to create your *argparse* parser your program will have the following arguments (aside from any you add yourself):

Mini-server for use with Igor

```
usage: igorServlet [-h] [--database DIR] [--name NAME] [--sslname NAME]
                   [--port PORT] [--nossl] [--nolog] [--noCapabilities]
                   [--capabilities] [--audience URL] [--issuer URL]
                   [--sharedKey B64STRING]
```

Named Arguments

--database	Config and logfiles in DIR (default: /home/docs/.igor, environment IGORSERVER_DIR)
	Default: "/home/docs/.igor"
--name	Program name for use in log and config filenames in database dir(default: __main__)
	Default: "__main__"
--sslname	Program name to look up SSL certificate and key in database dir (default: igor)
	Default: "igor"
--port	Port to serve on (default: 8080)
	Default: 8080
--nossl	Do no use https (ssl) on the service, even if certificates are available
--nolog	Disable http server logging to stdout
--noCapabilities	Disable access control via capabilities (allowing all access)
--capabilities	Enable access control via capabilities
--audience	Audience string for this servlet (for checking capabilities and their signature)
--issuer	Issuer string for this servlet (for checking capabilities and their signature)

--sharedKey Secret key shared with issuer (for checking capabilities and their signature)

CHAPTER 8

Rest entry points

The Igor HTTP or HTTPS server has a number of REST entrypoints. In the descriptions below, * can be a sequence of non-special characters (letters, digits, hyphens, etc), ** can also include special characters such as / and others (for pathnames and XPath expressions) and square brackets ([and]) denote optional parts.

Per entrypoint (URL path) the allowed methods are listed. If an entry point accepts query parameters (the bit after the ? in the URL) this is also explained.

8.1 /data/**

Accesses data elements. GET, PUT, POST and DELETE allowed. The argument is technically an XPath expression that should resolve to a single element (but see below). Data can be provided with mimetypes `application/xml`, `application/json` or `text/plain`.

Query parameters:

- `.VARIANT` can be used to get data slightly different:
  - `multi` to get allow getting multiple elements (normally multiple elements matching the expression is considered an error, except for `text/plain` when the values are concatenated)
  - `raw` will also include namespaced elements and attributes. These are used for internal administration and usually not returned.
  - `multiraw` combines `multi` and `raw` behaviour.
  - `ref` to get an XPath for the resulting objects.
- `.METHOD` convenience parameter for debugging with the browser. a `GET` request with `.METHOD=DELETE` will do a `DELETE` operation.

8.2 /evaluate/**

Evaluate an XPath expression that can return any XPath expression value. `GET` only.

8.3 /*

Get static or template web pages or other files. `GET` only. Returns static data from the `static` directory or an interpolated Jinja2 template from the `template` directory. Parameters are passed to the template.

8.4 /action/*

Trigger a named action. `GET` only.

8.5 /action/*/*

Trigger a named plugin action. `GET` only. The first field is the plugin name. The second field is the action name, which is looked up in `/data/plugindata/pluginname`.

8.6 /plugin/*[/*]

Run a plugin. `GET` only. First field is the plugin name, optional second field is the method of the plugin to call (default *index*). Query parameters are passed to the Python method as named arguments. Plugin data from `/data/plugindata/_name_` is made available to the plugin factory function, and if a *user* query parameter is present Igor will add a *userData* argument containing the data from `/data/identities/_user_/plugindata/_name_` (as a Python dictionary).

8.7 /plugin/*/page/*.html

Retrieve a plugin UI (user interface) page. The template page is looked up inside the plugin directory and rendered through Jinja2. Query parameters are passed to the template. Plugin data is available as above.

8.8 /plugin/*/script/*

Run a plugin script. `GET` only. First argument is the plugin name, second argument is the script name. The script is obtained from file `/plugins/_name_/scripts/_scriptname_.sh`.

query parameters are passed to the script as environment variables, with `igor_` prepended to the query parameter name. Per-plugin data and optional per-user data (as for `/plugin`) is combined into an `igor_pluginData` environment variable with has all the data in JSON encoded form. The plugin script has access to the plugin capability set, valid for a single call to the database (through `igorVar`) via a one-time-password mechanism.

8.9 /login

Helper for logging in and out with a browser session.

8.10 /internal/*[/*]

Helper for running internal commands such as *save*, *restart* or *updateStatus*.

CHAPTER 9

Igor database schema

While there is very little in Igor that is hard-coded here is the schema that is generally used by Igor databases.

The (preliminary) schema for access control can be found in the separate document *capabilities.md*.

The toplevel element is called `data`. Usually you can refer to the children of the toplevel element using a relative path such as `environment/location`, sometimes (within expressions triggered by events, for example) you must refer by absolute path, then you should use `/data/environment/location`.

There is no formal schema, but in the descriptions of the leaf elements below we specify the type expected:

- *integer*, *float*: the usual. Encoded as strings in XML, should be converted to native values in JSON.
- *string*: UTF-8 unicode string. Represented with the usual XML escaping in XML.
- *timestamp*: *integer* value representing seconds since the Unix epoch (1-Jan-1970 00:00 UTC).
- *isotime*: String representing local date and time in ISO-8601 human-readable form, for example `2017-04-14T14:25:09`.
- *boolean*: because XPath 1.0 has no concept of booleans and to facilitate round-tripping to JSON an empty value should be used for *false* and the string `true` for *true*.

There is an escape mechanism to use when a variable name would be illegal as an XML tag name. The following two XML code fragments are identical:

```
<abcd>efg</abcd>
```

```
<_e _e="abcd">efg</_e>
```

but in the latter `abcd` can be any string. Round-tripping to JSON of this construct is transparent.

In various elements XPath expressions can be used. These follow XPath 1.0, with a number of extensions:

- `$originalContext` is available in *action* XPaths and refers to the element that triggered the action.
- `igor_dateTime(number)` converts a *timestamp* to an *isotime*. When called without argument it returns the current date and time.

- `igor_timestamp(isotime)` converts an *isotime* to a *timestamp*. When called without argument it returns the current time.
- `igor_year_from_dateTime(isotime)` and similar functions from XPath 2.0 are available with the `igor_` prefix.
- `igor_date_equal(isotime, isotime)`, `igor_time_equal(isotime, isotime)`, `igor_dateTime_equal(isotime, isotime)` and the usual variations are available for comparing dates, times and date-time combinations.
- `igor_ifelse(expr1, expr2)` returns *expr1* if it is true, otherwise *expr2*.
- `igor_ifthenelse(expr1, expr2, expr3)` If *expr1* is true returns *expr2*, otherwise *expr3*.

9.1 environment

Stores high level information about the environment (household) in which Igor is operating such as GPS location.

9.1.1 environment/night

Nonzero if it is considered night. Set by standard action *checkNight*, used by various plugins to refrain from actions like turning on loud devices.

9.1.2 environment/location

GPS location of the home. Used by plugins like *buienradar* to get rain forecasts for the correct place. Generally initialized by the user (but could be done using a GPS plugin).

- `lat`: lattitude (float).
- `lon`: longitude (float).

9.1.3 environment/energy

Current energy consumption information. Indirectly set by plugins like *smartmeter*.

- `electricity`: current electricity consumption in kWh (float).

9.1.4 environment/weather

High-level information about temperature and such. Indirectly set by plugins like *netatmo*.

- `tempInside`: temperature inside in degrees Celcius (float).
- `tempOutside`: temperature outside in degrees Celcius (float).

9.1.5 environment/people

Information about people that is not considered privacy-sensitive:

- `count`: number of people on the premises

9.1.6 environment/messages

Informational messages produced by various plugins (or by external agents with a `POST` through the REST inferface). New messages will be picked up by plugins like *lcd* or *say* to present them to the user. Standard action *cleanup* will remove them after a while.

- `message`: a text string to be shown or spoken to the user (string).

9.1.7 environment/devices

A set of boolean values indicating that devices of which the end user is aware (such as mobile phones or keyring transponders) are in the house and active. Names should be user-friendly. The intention is that this category is used specifically for devices that are portable, and that are carried around by people (or dogs, or cars, or bicycles). These values are set by plugins like *dhcp*, after converting MAC-addresses to user-friendly names based on information in *plugindata*. Values here are used by rules from *actions* to populate *people*.

Example value:

- `laptopJack`: true if Jack's laptop is in the house (boolean).

9.1.8 environments/lights

(*Unused, currently*) A set of booleans indicating the state of lights (or actually any electric device controllable through a smart outlet or something similar). These may be write-only, as reading the state of light switches is often impossible. The names should be human-readable, and changes in the values are picked up by plugins like *kaku* to turn on or off those lights (after converting the names to switch IDs through information in *plugindata*).

Example value:

- `diningRoomTable`: boolean, set to true or false to turn this light on or off.

9.1.9 environment/systemHealth

High-level information about how the technical infrastructure of the household (such as the internet connection, and Igor itself) is functioning. The *systemHealth* plugin maintains this, from low-level information in `/data/status`. *systemHealth* creates entries in `environment/systemHealth/messages` with descriptive names and user-readable text. These entries are removed when the anomalous condition no longer exists. For example:

```
<systemHealth>
        <messages>
                <internet>It seems the internet connection is down.</internet>
        </messages>
</systemHealth>
```

The user can silence anomalous conditions he or she knows about (and does not want to be bothered with) for a period of time by setting fields in `/data/status`.

9.2 status

Status information on everything Igor knows about, such as whether services and devices are functioning, when they were last accessed correctly and any error messages produced. Updated by `/internal/updateStatus`, governed by the `representing` variable in actions and such. Interpreted by the *systemHealth* plugin, among others.

Each entry has a number of fields:

- `alive` (boolean) true if last attempt to access the device or service was successful.
- `errorMessage` (string, optional) human-readable error message in case last attempt was unsuccessful.
- `lastActivity` (timestamp) time of the last attempt to access the device or service.
- `lastSuccess` (timestamp, optional) time of the most recent successful access.
- `lastFailure` (timestamp, optional) time of the most recent unsuccessful access.
- `ignoreErrorsUntil` (timestamp, optional) "silencing" timestamp, the *systemHealth* plugin will not complain about errors in this entry until the given time.

Entries are grouped by their type:

- `status/igor` Igor components, insofar they can be tested separately:
  - `status/igor/start` Igor main server startup. This entry has a few more fields beside the ones listed above:
    - `status/igor/start/url`: Base URL to use with `igorVar --url` (string).
    - `status/igor/start/host`: Host name on which this Igor instance runs (string).
    - `status/igor/start/port`: Port on which this Igor listens (integer).
    - `status/igor/start/version`: Igor version (string).
    - `status/igor/start/count`: How often this Igor instance has been (re)started.
  - `status/igor/core` the Igor main server loop
  - `status/igor/save` saving the Igor database to disk
  - `status/igor/web` the external HTTP interface to Igor
- `status/sensors` Sensors, or sensor categories (for sensors such as *ble* where a single plugin handles multiple sensors. Entries are named for the sensor or category.
- `status/devices` Devices (actuators and appliances). Entries are named for the individual device.
- `status/services` Services external to Igor, for which only status information is kept. Some examples:
  - `status/services/internet` whether the internet connection works. Determined by the *lan plugin* by trying to access *google.com*.
  - `status/services/backup` whether Time Machine backups are made. Determined by the *timemachine plugin*.

9.3 sensors

Stores low level information from devices that are generally considered read-only such as temperature sensors. See the descriptions of the individual plugins for details:

- `sensors/dhcp`: DHCP leases. See *dhcp plugin readme*.
- `sensors/ble`: Visible Bluetooth LE devices. See *ble plugin readme*.
- `sensors/rfid`: RFID tags recently presented. See *iirfid plugin readme* and *homey plugin readme*.
- `sensors/tags`: Named RFID tags recently presented. See *iirfid plugin readme* and *homey plugin readme*.
- `sensors/netAtmo`: Weather data. See *netatmo plugin readme*.

- `sensors/smartMeter`: Energy consumption. See *smartMeter plugin readme*.
- `sensors/fitbit`: Health data. See *fitbit plugin readme*.
- `sensors/buienradar`: Expected rainfall data. See *buienradar plugin readme* and *neoclock plugin readme*.

9.4 devices

Stores low level information for devices that are write-only (actuators, such as motors to lower blinds) or read-write (applicances such as television sets). For writable devices such as actuators there are usually rules in *actions* that take care of changing the state of the actuator when values in this section are changed.

See the descriptions of the individual plugins for details:

- `devices/tv`: Television set, information like power status, current channel, etc. See *philips plugin readme*.
- `devices/plant`: Current position of the movable plant, see *plant plugin readme*.
- `devices/lcd`: Adding a new `devices/lcd/message` will result in this message being displayed. See *lcd plugin readme*.

9.5 people

Stores high level information about actual people, such as whether they are home or not. Names in the *people* section match names in the *identities* section.

The intention about the separation between *people* and *environment/people* is that the latter is available to everyone who has access to the database, while the former is only accessible to users who have logged in (assuming capability-based access control is enabled). The data in *identities/username* is even more protected and only available to that specific user. So, *identities/username* can contain contain private information (such as hardware address of mobile phone), *people/username* can contain semi-private information (such as whether *username* is at home or not) and *identities/people* non-private information (such as the number of people currently at home).

As an example:

- `people/jack/home`: Boolean that indicates whether a use "jack" is considered to be in the house (as determined by rules that trigger on his devices).

9.6 identities

Stores identifying information about people, such as the identity of their cellphone or login information for cloud-based health data storage.

As an example:

- `identities/jack/encryptedPassword` The encrypted Igor password for user *jack*. Verified by the `/login` entry point, after which the user identity is stored in the session, or when supplied through the HTTP `Authorization: Basic` header.
- `identities/jack/plugindata`: Per-user data for plugins. For example:
  - `identities/jack/plugindata/fitbit`: Information that allows the *fitbit plugin* to obtain health information for user "Jack".
- `identities/jack/device`: Name of a device that user "Jack" tends to carry with him (string).

If capability support is enabled, identity entries will also carry the set of capabilities for that user, but these are inaccessible during normal operation.

A special user *admin* will carry a set of *master capabilities*.

9.7 actions

Stores triggers and actions that operate on the database. *(this name is hardcoded in the Igor implementation)* Action elements can also be present inside `plugindata` children.

Actions can be triggered by external access, timers, conditions in the database or a combination of those:

- Actions that are named, for example `save`, can be triggered by external means (by accessing `http://igor.local:9333/action/save`). Multiple actions can have the same name, and external access will trigger all of them.
- Actions can have an interval and will then be triggered periodically.
- Actions can have an XPath expression and will the be triggered whenever any database element matching this expression is modified. As an example, the `save` action above has an expression of `/data/identities//*` resulting the in the database being saved whenever anything in the *identities* section is changed.

When an action is triggered a number of conditions is tested to see whether the action actually needs to be run or not:

- It is possible to specify that an action should not be run before a certain time.
- It is possible to specify that an action should not be run more often than a given minimum time interval.
- It is possible to specify a database condition (as an XPath expression) to determine whether the action should be run or not.

If the condition is met then the action is run. This takes the form of an HTTP operation on a URL, possible with data to provide to the operation. The url and data fields support the use of XPath expressions inside curly braces `{` and `}` (Attribute Value Templates, AVTs, such as used in XForms and SMIL, for example).

If the action is triggered by an XPath expression then the XPath expressions within the action (such as in an AVT or condition) are run in a context with the triggering element as the current node. The triggering element is also available as the `$originalContext` variable, so if the expression changes the context you can still refer to the triggering node.

Here is a description of the available elements:

- `actions/action/name`: Name of the action (string). Action will trigger when `/actions/name` is accessed.
- `actions/action/xpath`: XPath expression that must deliver a *node* or *nodeset* (string). Action will trigger if any of these nodes is modified.
- `actions/action/multiple`: A boolean that signals what should happen if multiple elements are changed (and match the xpath expression) by the same operation. When false (the default) the action triggers once, with a random element as the context. When true the action will trigger for each element in the nodeset.
- `actions/action/aggregate`: A boolean to indicate that multiple triggers of this action can be aggregated into a single call. Note that this is completely different from `multiple`, it can be used to forestall scheduling an action if the identical action is already waiting to be executed.
- `actions/action/interval`: Interval in seconds (integer). Action will trigger at least once every *interval* seconds.
- `actions/action/minInterval`: Minimum interval in seconds (integer). Action will trigger at most once every *minInterval* seconds.

- `actions/action/notBefore`: Earliest time this action will trigger again (timestamp). This field is set by Igor whenever the action is triggered, using data from *minInterval*, and it is actually the way the *minInterval* functionality is implemented.
- `actions/action/condition`: XPath expression that is evaluated whenever a trigger has happened and that must return *true* for the action to be executed.
- `actions/action/url`: The URL to which a request should be made (string). AVTs can be used in this field. This is the only required field.
- `actions/action/method`: The method used to access the url, default GET (string).
- `actions/action/data`: For POST and PUT methods, the data to supply to the operation (string). Can use AVTs.
- `actions/action/mimetype`: The MIME type of *data* (string), default `text/plain`.
- `actions/action/representing`: The entity on whose behalf this action runs (string), for reporting in *systemHealth*.
- `actions/action/creator`: The plugin that created this action (string), for showing the action in the plugin UI.

If capabilities are enabled each action can carry a set of capabilities and the *actions* element itself can also carry a set (that will be inherited by each action).

9.7.1 Standard actions

There are a number of standard actions, which are used by Igor itself or used to fill some of the standard elements in the database. Multiple actions with the same name can exist, and all of them will fire (so you can add actions to do additional things if these events happen). These actions (by *name*) are:

- *start*: fired when Igor is started (automatically by Igor).
- *save*: saves the in-memory copy of the database to the external file. Called periodically, and whenever a part of the database that is somehow considered important is changed.
- *cleanup*: deletes old elements in `environment/messages` and such.
- *updateActions*: updates the internal action datastructure whenever elements are added (not changed) in `actions`.
- *checkNight*: maintains the value of `environment/night`.
- *updatePeople*: updates people availability in `people` when device availability in `environment/devices` changes.
- *countPeopleAtHome*: updates `environment/people/count` when people availability changes.

9.8 sandbox

Does nothing special, specifically meant to play with `igorVar` and REST access and such.

9.9 eventSources

Contains references to Server-Sent event (SSE) sources, and where in the database the resulting values should be stored. Each event source will create a listener thread that opens a connection to the source and updates the database

as events come in. Event types and event IDs are currently ignored, only the SSE *data* field is used.

- `eventSources/eventSource/src`: URL of the SSE endpoint to connect to (string).
- `eventSources/eventSource/srcMethod`: Method used to access the *src* url (string), default GET.
- `eventSources/eventSource/dst`: URL where the SSE *data* should be sent to (string).
- `eventSources/eventSource/dstMethod`: Method used to access the *dst* url (string), default PUT.
- `eventSources/eventSource/mimetype`: How the *data* field should be interpreted (and how it is forwarded to *dst*). Default `application/json`.

9.10 plugindata

Contains per-plugin configuration data, such as the mapping of hardware network addresses (MAC addresses) to device names. Can also contain *action* elements (for actions that are specific to the plugin implementation). These actions will run with all the access control rights of the plugin itself.

See the descriptions of the individual plugins for details on per-plugin data.

CHAPTER 10

Access Control schema

Currently data pertaining to access control is stored in the main database, with an XML namespace of `http://jackjansen.nl/igor/authentication` (usually encoded with the `xmlns:au` prefix).

This data is hidden from normal Igor access (unless query parameter `.VARIANT=raw` is used). In principle this should be safe, because an external call cannot modify the capabilities and there is no secret information contained in the capability data. There is however an issue that a call with the right permissions can accidentally delete a capability be replacing a subtree (with `PUT`).

10.1 Capability structure

A capability is stored in an `au:capability` element.

- `comment` textual description, to keep us sane during development.
- `cid` unique ID of this capability.
- `child` one entry for each child (delegated) capability of this capability.
- `parent` parent of this capability.
- `delegate` boolean, if `true` this capability can be delegated. If the value is the string `external` this capability can be the parent of any capability as long as that new capability has an `aud` field.
- `obj` an XPath referencing a single element (or a nonexisting element with a single existing parent element) to which this capability refers. Rights on that object and its descendants are governed by a number of other fields:
  - `get` Together with `obj` defines on which elements this capability grants `GET` rights:
    - empty (or non-existent): none.
    - `self` the element itself only.
    - `descendant-or-self` the whole subtree rooted at the element (the element itself, its children, its grandchildren, etc).
    - `descendant` the whole subtree rooted at the element except the element itself.

        * `child` direct children of the element.
        * More values may be added later.
    - `put` Together with `obj` defines on which elements this capability grants `PUT` rights. Values as for `get`.
    - `post` Together with `obj` defines on which elements this capability grants `POST` rights. Values as for `get`.
    - `delete` Together with `obj` defines on which elements this capability grants `DELETE` rights. Values as for `get`.

The `obj` field will usually be an absolute XPath (starting with `/data`) but there are a number of other values used for non-database accesses (REST and other):

- `/action` is the virtual object tree of actions (the REST `/action` entrypoints)
- `/internal` is the virtual object tree of internal actions (the REST `/internal` entrypoints)
- `/plugin` is the virtual object tree of plugins (the REST `/plugin` entrypoint)
- `/filesystem` is the right to do operations that modify the filesystem. Checking this capability is currently only implemented for installing plugins.

Capabilities that have an external representation may have a few extra fields:

- `iss` Issuer of this capability. Usually the URL of `/issuer` on this igor.
- `aud` Audience of this capability. Required for capabilities that Igor will encode as a JWT (Json Web Token, https://en.wikipedia.org/wiki/JSON_Web_Token) in an outgoing `Authentication: Bearer` header.
- `sub` Subject of this capability. Required for capabilities that Igor receives as JWT in an incoming `Authentication: Bearer` header.
- For outgoing capabilities there may be other fields that are meaningful to the *audience* of the capability.

External capabilities are protected using a symmetric key that is shared between Issuer and Audience (for outgoing capabilities) or Issuer and Subject (for incoming keys). This key is used to sign the JWT.

10.2 Database schema additions

10.2.1 /internal/accessControl

Not really part of the database, but this is the entry point to manage (list, delegate, revoke) capabilities.

10.2.2 /data/au:access

Required for further schema requirements.

10.2.3 /data/au:access/au:defaultCapabilities

Capabilities that will be used for any action, user or request that has no `Authentication: Bearer` header. For users and actions this set of capabilities is also valid if they have their own set. In other words: their own set augments the set of capabilities, it does not replace it.

These capabilities should be here:

- get(descendant-or-self), /data/environment
- get(descendant-or-self), /data/status

- get(descendant-or-self), /data/services/igor
- get(child), /static
- get(child), /internal/accessControl
- get(descendant-or-self)/put(descendant)/post(descendant)/delete(descendant), /data/sandbox

10.2.4 /data/au:access/au:exportedCapabilities

Stores each `au:capability` for which an external representation has been created. Mainly so that each capability has an "owner".

10.2.5 /data/au:access/au:revokedCapabilities

Stores all external capabilities that have been revoced. For each such capability there is a `au:revokedCapability` with at least a field `cid` that holds the capability ID. Optionally there is an `nva` field, Not Valid After, copied from the original capability, that indicates when this revoked capability can be cleaned up because the original is no longer valid.

10.2.6 /data/au:access/au:unusedCapabilities

This is an optional area to store capabilities that are valid but currently not used, and that have no owner. For Igor development, really.

10.2.7 /data/au:access/au:sharedKeys

Empty placeholder for the secret shared key data in the *shadow.xml* database (see below).

10.2.8 /data/identities

Capabilities carried by all users that are logged in. Contains at least:

- get(descendent-or-self), /data/people

10.2.9 /data/identities/admin

User that holds the master capabilities, capabilities with fairly unlimited access from which more limited capabilities are descended (through delegation).

There are at least the following capabilities, of which most other capabilities are descended (through delegation and narrowing the scope):

- get(descendant-or-self)+put(descendant)+post(descendant)+delete(descendant), /data
- get(descendant), /action
- get(descendant), /plugin
- get(descendant), /pluginscript
- get(descendant), /internal
- an empty capability (no rights, no object) with `cid=root` and no parent. This is the root of the capability tree.

10.2.10 /data/identities/*user*

Capabilities this user will carry when logged in. Contains at least:

- get(descendent-or-self)+put(descendent)+post(descendent)+delete(descendent), /data/identities/*user*
- put(descendent)+post(descendent)+delete(descendent), /data/people/*user*

10.2.11 /data/actions

Capabilities that are carried by all actions. Contains at least:

- get(descendant), /plugin
- get(child), /action

10.2.12 /data/actions/action

Capabilities this action will carry when executing.

10.2.13 /data/plugindata/*pluginname*/au:capability

Capabilities this plugin will carry when executing. Also available to the template pages and scripts for this plugin.

10.3 Shadow database

The main igor database `.igor/database.xml` does not contain any secret information, so that access control secrets cannot be leaked accidentally through the REST interface. Therefore, all secret information is kept in a separate database `.igor/shadow.xml` which has in principle the same structure as the main database, but only contains secret information.

In practice, the shadow database contains only the shared secret keys:

10.3.1 /data/au:access/au:sharedKeys

Stores symmetric keys shared between Igor and a single external party. These keys are used to sign outgoing capabilities (and check incoming capabilities). Each key is stored in an `au:sharedKey` element with the following fields:

- `iss` Issuer.
- `aud` (optional) Audience.
- `sub` (optional) Subject.
- `externalKey` Symmteric key to use.

Keys are looked up either by the combination of *iss* and *aud* (for outgoing keys) or *iss* and *sub* (for incoming keys).

10.4 Implementation details

This section lists some of the ideas that came up when designing the capability structure. They may not be true anymore, but the text is kept here because it is not currently stored anywhere else.

10.4.1 Capability consistency checks

Capabilities need to be checked for consistency, and for adherence to the schema.

The following checks are done as a first order check, and ensure the base infrastructure for the schema is in place:

- `/data/au:access` exists.
- `/data/au:access/au:defaultCapabilities` exists.
- `/data/au:access/au:exportedCapabilities` exists.
- `/data/au:access/au:revokedCapabilities` exists.
- `/data/au:access/au:unusedCapabilities` exists.
- `/data/au:access/au:sharedKeys` exists.
- `/data/identities` exists.
- `/data/identities/admin` exists.
- `/data/identities/admin/au:capability[cid='root']` exists.
- `/data/actions` exists.

As a second check we test that the default set of capabilities (as per the schema above) exist and are in their correct location.

As a third check we enumerate all capabilities and check the following assertions. These ensure that the tree of all capabilities is consistent:

- Each capability must have a `cid`.
- This `cid` must be unique.
- Each capability (except `cid=root`) must have an existing `parent`, if not the capability is given `parent=root`.
- Each capability must have its `cid` listed in the parent `child` fields. If not it is added.
- Each `child` of each capability must exist. If not the `child` is removed.

As a fourth check we check that every capability is in an expected location. In other words, the DOM parent of the capability is one of:

- Any of the containers in the first check, or
- `/data/identities/*`
- `/data/plugindata/*`
- `/data/actions/action`

Capabilities that fail this check are moved into `/data/au:access/au:unusedCapabilities`.

10.4.2 Actions on adding a new user

To be refined, but at least:

- Create `/data/people` entry.
- Create `/data/identities` entry,
  - Fill with capabilities mentioned above
  - Create password

The API will need at least *name* and *password*. Because of access control policies it is implied that only the *admin* user can call this API (or any agent that the *admin* user has granted the corresponding capabilities to).

10.4.3 Actions on deleting a user

- Move any non-standard capabilities (really: any capability with `aud` not the current Igor) to a safe place (probably the *admin* user).
- Delete `/data/people` and `/data/identities` entries.

The API will need just the user name, and the same access control rules as for adding users will apply.

10.4.4 Actions on adding a new device

- Create SSL key with `igorCA` (or `iotsa/extras/make-igor-signed-cert.sh` or via `/plugin/ca`) and copy the key and certificate to the device.
- Create a shared secret key with new device as audience, via `/capabilities.html` or `/internal/accessControl`, and copy the secret key to the device.
- Create an initial "allow all API actions" capability for the device (TBD) and store it in some users' space (current user? admin user?)
  - Igor should automatically pick up the correct secret key and encode the capability with it, when talking to the device.
- If the device is also a *sensor*, i.e. if it can also trigger actions in Igor, all of the *sensor* actions must also be done.

10.4.5 Actions on adding a new sensor

- Create a shared secret key with the new sensor as subject, via `/capabilities.html` or `/internal/accessControl`, and copy the secret key to the device.
- Create a capability (with the sensor as subject and audience Igor) for each action the sensor should be able to trigger.
- Export these capabilities (Igor will pick up the correct secret key based on the subject) and copy them to the sensor.

10.4.6 Generalized API for adding a device or sensor

Data to be supplied to this action:

- Name of the device/sensor.
- Boolean *isSensor*.

- Boolean *isDevice*.
- Hostname or IP address of the sensor (defaults to name with *.local* appended).
  - if *isDevice* this will be used as the *audience* of the first shared key.
  - If *isSensor* this will be used as the *subject* of the second shared key.
- if *isDevice*: Partial URL of the API of this device (such as `/api`). Will be the *object* of the device access capability stored in the users' *identities* entry.
- if *isSensor*: List of (*name*, *verb*, *object*) this sensor will contact (or empty for non-sensor devices). If non-empty the sensor shared key (audience Igor, subject the sensor) will be used to sign these.

Data returned:

- if *isDevice*:
  - SSL key
  - SSL certificate
  - shared device key
- if *isSensor*:
  - List of (*name*, *verb*, *url*, signed capability).

Data saved in Igor database:

- Shared keys (in the hidden area)
- if *isDevice*: Capability for accessing the device
- Entry in either `/data/devices` or `/data/sensors`.

It needs to be worked out what the access control rights are that are needed for this API. It seems as though no special rights are needed for devices, and for sensors the caller needs to have capabilities (with `delegate=true`) for each of the verb/object combinations.

It also needs to be worked out whether the user (or other agent) that calls this API gets permissions to the `/data/devices` or `/data/sensors` areas.

10.4.7 Deleting a device or sensor

- The entries in `/data/devices` and `/data/sensors` should be deleted.
- The shared keys should be deleted.
- The SSL certificate should be revoked.

10.4.8 Other actions on agent changes

To be determined what is needed when adding/removing/changing plugins and actions.

Also to be determined whether anything needs to be done when certificates expire or are revoked.

Also to be determined what to do when secret keys are deleted and re-added.

CHAPTER 11

Plugin implementation

The description here is probably not complete enough yet. Examine some of the standard plugins to see how things work. Some good examples plugins:

- *say* is a simple output-only plugin that uses shell commands to drive a speech synthesizer.
- *lan* uses Python sockets and *requests* to check whether services are available.
- *iotsaDiscovery* is a plugin with an elaborate user interface and a good example of the Jinja HTML templates working together with the Python plugin code.
- *fitbit* obtains data for one or more users from an external cloud service, and uses the three-way *OAuth2* handshake to authenticate to that service.

11.1 Plugin Structure

A plugin generally contains either a Python module or a set of scripts (or both) to communicate with an external device or service. In addition a plugin can contain user-interface pages to allow the user to configure or operate the plugin (or the device it controls).

11.1.1 igorplugin.py

A plugin can be implemented in Python. Then it must define a class (or factory function)

```
igorPlugin(igor, pluginName, pluginData)
```

which is called whenever any method of the plugin is to be called. This function should return an object on which the individual methods are looked up. The `igorPlugin` function is called every time a plugin method needs to be called, but it can of course return a singleton object. See the *watchdog* plugin for an example. *igor* is a pointer to the global Igor object (see below), *PluginName* is the name under which the plugin has been installed, and *PluginData* is filled from `/data/plugindata/_pluginname_`.

Each operation will call a method on the object. There will always be an argument `token` which will be the current full set of capabilities (or `None` if Igor is running without capability support) and which the plugin will have to pass to most Igor API calls. The full set of capabilities consists of the capabilities carried by the caller of this plugin *plus* the capabilities of the plugin itself. There is another argument `callerToken` which contains only the capabilities of the caller. The intention of these two sets of capabilities is that the plugin code will use the *token* set when accessing private data and *callerToken* when accessing data that is passed in by the user (or when it wants to ensure the calling user actually has the right to access the data).

Accessing `/plugin/pluginname` will call the `index()` method.

Accessing `/plugin/pluginname/methodname` will call `methodname()`.

The methods are called with `**kwargs` encoding the plugin arguments, and if there is a `user` argument there will be an additional argument `userData` which is filled from `/data/identities/_user_/plugindata/_pluginname_`.

Methods starting with an underscore _ are not callable through the web interface but can be called by the plugin template pages or by other plugins.

The *igor* object has a number of attributes that allow access to various aspects of Igor:

- `igor.database` is the XML database (implemented in `igor.xmlDatabase.DBImpl`)
- `igor.databaseAccessor` is a higher level, more REST-like interface to the database.
- `igor.internal` gives access to the set of commands implemented by `igor.__main__.IgorInternal`.
- `igor.app` is the web application (from `igor.webApp.WebApp`).
- `igor.plugins` is the plugin manager.

11.1.2 scripts

A plugin can be (partially) implemented with shell scripts. Accessing `/pluginscript/pluginname/scriptname` will try to run `pluginname/scripts/scriptname.sh`.

Scripts get an environment variable `IGORSERVER_URL` set correctly so they can use the *igorVar* command easily.

Each argument is passed to the script (in Python notation) with `igor_` prepended to the name. `igor_pluginName` contains the name under which the plugin is installed.

The per-plugin data from `/data/plugindata/_pluginname_` and (if the *user* argument is present) the per-user per-plugin data from `/data/identities/_user_/plugindata/_pluginname_` is encoded as a Python dictionary and passed in the `igor_pluginData` environment variable.

11.1.3 database-fragment.xml

Many plugins require plugin-specific data in the database. Often there are one or more of the following items:

- plugin-specific actions that are needed to actually fire the plugin,
- plugin settings, for example to tell which host a specific device is connected to,
- boilerplate entries for where the plugin will store its data.

Usually these entries are explained in the plugin readme file, in the *schema* section.

Usually there is a file `database-fragment.xml` that show the entries needed. Basically this file is the minimal set of elements that should be in the database for the plugin to function.

This database fragment is overlayed onto the database when installing the plugin. Every occurrence of the exact string `{plugin}` is replaced by the name of the plugin before installing into the database.

> *Note*: this looks somewhat like a Jinja construct but it is not, for the current release. Simple text substitution is used.

The fragment overlay installation may be delayed until the next time the Igor server is restarted.

It may be necessary to do some hand editing of the database after installing, because you may have to modify some elements (such as hostname fields) and you may need to duplicate some (with modifications) for example if you want the *lan* plugin to test different services.

11.1.4 *.html

A plugin can contain a user interface through HTML pages. These are accessed with URLs `/plugin/_pluginname_/page/_pagename_.html`. Actually, these are Jinja2 templates. Within the template you have access to the following variables:

- *pluginName* is the name under which the plugin has been installed.
- *pluginObject* the plugin object (if the plugin has an `igorplugin.py` Python module).
- *callerToken* is the capability of the current user (the user visiting the page).
- *token* is the capability of the current user (the user visiting the page) plus the capability owned by the plugin itself.
- *pluginData* is the internal data of the plugin (from `/data/plugindata`).
- *igor* is the toplevel Igor object.
- *user* is the current user (if logged in).
- *userData* is the per-plugin data for the current user (if logged in).
- all url parameters.

> *Note*: the availability of the *igor* object means that a plugin has rather unlimited power, and can probably run any command and access any file that the userID under which Igor is executing can access. This is a security issue, and you should never install plugins from sources you do not trust. This will be addressed in a future release.

In general, the template should provide forms and such to allow the user to change settings, and then call methods in the plugin proper to implement those changes (because the plugin will run with a *token* that allows read/write access to the plugin data). If methods are intended to be called solely from templates and never directly through the REST interface you should start the methodname with an underscore.

The plugin decides whether to use *token* or *callerToken* to access data depending on whether it is accessing data on behalf of itself (then it uses *token*) or on behalf of the person visiting the web page (in which case it uses *callerToken*).

Plugins can access other plugins through the `igor.plugins` object.

CHAPTER 12

Implementation details

12.1 ~/.igor directory structure

The `~/.igor` directory can contain the following files and subdirectories:

- `database.xml` The main XML database. See *Igor database schema* and *Access Control schema* for the format.
- `database.xml.YYYYMMDDHHMMSS` Backups of the database (created automatically).
- `shadow.xml` Database for secrets. Structurally identical to main database but only contains issuer shared secret keys.
- `plugins` directory with installed plugins. Many will be symlinks into `std-plugins` directory.
- `std-plugins` symlink to standard plugins directory in the igor package (will be re-created on every startup).
- `igor.log` the *httpd-style* log of all Igor activity.
- `igor.log.*` older logfiles.
- `igor.crt` and `igor.key` if Igor is run in *https* mode these are the certificate and key used. `igor.crt` is also used by *igorVar* and *igorControl* to check the server identity.
- `ca` certificate authority data such as signing keys and certificates.
- `igorSessions.db` may be available to store igor user sessions.
- `igor.cfg` configuration file for *igorVar*, *igorControl* and *igorCA* (not used by *igorServer* or *igorSetup*). See *Configuration file* for details.

12.2 Internal APIs

This section still needs to be written. For now you have to look at the source code. Here is a quick breakdown of the object structure to get you started:

The toplevel singleton object is of class `IgorServer`, declared in __main__.py. It is usually called `igor` in plugins and such, and many objects have a `self.igor` backlink to this object.

The igor object has the following attributes that are considered part of its public (internal) interface:

- `pathnames` is an object storing all relevant pathnames.
- `internal` (class `__main__.IgorInternal`) implements the methods of the `/internal` REST endpoint.
- `access` (class `access.__init__.Access`) implements capabilities, access control, external shared secret keys and storage of all these.
- `database` is the low-level XML database (class `xmlDatabase.DBImpl`) which allows fairly unrestricted access to all data, including the underlying DOM tree.
- `databaseAccessor` is a higher level, more secure API to the database (class `webApp.XmlDatabaseAccess`).
- `urlCaller` is used to make REST calls, both internally within Igor and to external services (class `callUrl.UrlCaller`)
- `plugins` is the plugin manager (class `pluginHandler.IgorPlugins`)
- `actionHandler` implements actions, when they are triggered and what they do (class `actions.ActionCollection`)

12.3 Access Control Implementation

(version of 12-Nov-2017)

Note that this description is seriously outdated.

The Igor access control implementation falls apart into two distinct areas:

- *mechanism*, how Igor implements that it is possible to check that a certain operation can proceed on a certain object in the current circumstances, and
- *policy*, the actual checking.

This document is about the *mechanism*, the *policy* is the subject of Pauline's research.

12.3.1 Mechanism API

Three types of objects are involved in the access checking mechanism. The whole implementation is in the module `igor.access`. This module will eventually contain the policy implementation as well (to replace the current strawman policy).

The naming of API elements (specifically the use of *token*) reflects our current thinking about policy direction but is by no means limiting: a token could just as easily contain an *identity* and *token checking* would then be implemented as *ACL checking*.

But, of course, the API may change to accomodate the policy.

12.3.2 Object Structure

The main object is a singleton object `igor.access.singleton` of class `Access`. This object is used by all other modules to obtain tokes and token checkers. It is a singleton because the database is a singleton too.

Tokens are represented by instances of the `AccessToken` class (or classes with the same API). An `AccessToken` can represent an actual token (supplied by an incoming HTTP request or picked up by an Igor *action* for outgoing or internal requests), but there are also special implementations for *"No token supplied"* and *"Igor itself"*. The latter is used, for example, when Igor updates its *last boot time* variable, and has no external representation.

Tokens are checked by instances of the `AccessChecker` class. Whenever *any* operation on *any* object is attempted an access checker for that object is instantiated. It is passed the *token* accompanying the operation, and decides whether the operation is allowed.

12.3.3 Integration

Integration with the rest of Igor is very simple. All database access methods require a *token* parameter, and before returning any XML element (or the value from any XML element, or before modifying or deleting the XML element) they obtain an *AccessChecker* for the object. The operation only proceeds if the *AccessChecker* allows it.

> As of this writing access checking is not fully implemented for XPath functions yet, so it is theoretically possible to obtain data from an element by not accessing the element directly but by passing it through an XML function. This will be addressed later.

The higher level API calls also all have a *token* parameter, and usually simply pass the token on to the lower layers.

At the top level of incoming HTTP requests the token is obtained from the HTTP headers (or something similar).

At the top level of *action* firing the token is obtained from the action description in the database (possibly indirectly).

> There is a bit of *policy* here: it may turn out we want to carry the original token that caused the action to fire, or maybe a token representing the union of the two tokens.

Plugins are similar to actions, they can also carry their own token.

12.3.4 Access Interface

The `Access` object has four methods:

- `checkerForElement(element)` returns an `AccessChecker` instance for the given XML element. The intention is that this checker can be cached (for example as a hidden pointer on the XML element implementation) as long as it is deleted when the access policies for the element change.
- `tokenForRequest(headers)` returns an `AccessToken` for an incoming HTTP request.
- `tokenForIgor()` returns a special token for internal Igor operations.
- `tokenForPlugin(name)` returns a token for the plugin with the given name. *(this API is expected to change)*
- `tokenForAction(element)` returns the token for the action whose XML element is passed in.

12.3.5 AccessToken interface

The `AccessToken` object has one method:

- `addToHeaders(headers)` called when a token should be carried on an outgoing HTTP request. If the token has a valid externl representation it adds that representation to the `headers` dictionary. *(this API is expected to change)*

12.3.6 AccessChecker interface

The `AccessChecker` object has one method:

- `allowed(operation, token)` return `True` if `token` (which is an `AccessToken`) has the right to execute `operation`. Currently `operation` is a string with the following possible values:
  - `'get'` (read the element)
  - `'put'` (modify the element)
  - `'post'` (to create children elements)
  - `'delete'` (remove the element)
  - `'run'` (run the action or plugin)

CHAPTER 13

Getting started

First read the *Introduction* to get a feeling for what Igor does, how it does it and whether this fits your needs. Also check the *available standard plugins* to check for which devices and services plugins are available, and that these match the things you want to use Igor for.

Then read the documentation on *installing the software* and install the Igor software. Next you should check *Initial setup* which will tell you how to initialize your database, with the various options you want (mainly to do with security and access control).

Then *Administration* and *Command line utilities* will show you how to use Igor in normal operation.

To interface between Igor and other applications or services you should check the *Command line utilities* documentation for interfacing to shell scripts, *Python modules* for interfacing to Python programs and *Rest entry points* for interfacing to REST services.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

Symbols

A

D

G

H

I

P

R

S