



myminapp

Manual	
Date:	2024-02-08
Programm version:	0.9.2
Author:	berryunit

Contents

Contents.....	2
Quickstart	3
1. Basics	5
1.1 Application structure	5
1.2 Terms and explanations.....	6
1.3 Details.....	6
2. Application.....	7
2.1 Application instances	7
2.2 Application definition.....	8
2.3 Application server	10
2.4 Web frontend.....	11
2.4.1 Operation.....	11
2.4.2 Examples	12
2.4.3 Other.....	19
3. Commands.....	20
4. Devices	21
5. Utilities	23
5.1 Helper	23
5.2 Logger	23
5.3 Monitor.....	23
5.4 Scheduler.....	24
5.5 Storage.....	25
6. Create your own classes	26
6.1 First steps.....	27
6.2 Further steps	30
7. Backup.....	31
Appendix	32
A. Licenses	32
B. Installing with pip.....	34
C. Creating certificates	35
D. Recording data for statistical purposes.....	39
E. Example scenario	42
F. Program development	43
G. Credits	46

Quickstart

'myminapp' is a minimal but complete Python application that exemplifies how small DIY home projects can be programmed with little effort.

It offers the following features out-of-the-box:

- Complete, transparent application structure with commands, logging and data storage
- Simple usage via Python shell and integration into scripts and programs
- Application server and web frontend for HTTP(S) requests
- Time-controlled automation
- Multilingual message texts
- Various commands that can be directly useful or serve as patterns
- Easy expandability for the implementation of individual tasks

A computer running Python version 3.10 or higher is required. A USB SSD is recommended for SBCs such as Raspberry Pi, as SD cards are not suitable for continuous operation.

To install myminapp and execute the 'helloworld' command in different ways, proceed as follows.

Step 1 - Install myminapp

Download the release file 'myminapp-version.zip' from '<https://github.com/berryunit/myminapp>' and unzip it into a directory with read and write access.

The following assumes that the file was unpacked under Linux by user 'u1' to '/home/u1/myminapp'. It is also assumed that Python version 3 can be called up on the system with `python3`.

Step 2 - Execute command 'helloworld' via Python shell

1. Start the Python shell in the 'home/u1' directory (above myminapp). Enter:

```
python3
```

2. Enter the following within the Python shell:

```
from myminapp.app import App
app1 = App(1)
app1.perform_command({'cmd':'helloworld', 'value':'Hello World'})
app1.close()
```

3. Exit the Python shell with:

```
exit()
```

Step 3 - Execute requests via application server

Start the application server in the 'home/u1' directory (above myminapp). Enter:

```
python3 -m myminapp.appserver
```

3.1 - Execute command 'helloworld' via CURL

Open another terminal and enter the following (without line break):

```
curl --get --data-urlencode '{"cmd": "helloworld", "value": "Hello World"}'  
http://localhost:8081
```

3.2 - Execute command 'helloworld' via browser URL line

Open a browser and enter it in the URL line:

```
http://localhost:8081/{"cmd": "helloworld", "value": "Hello World"}
```

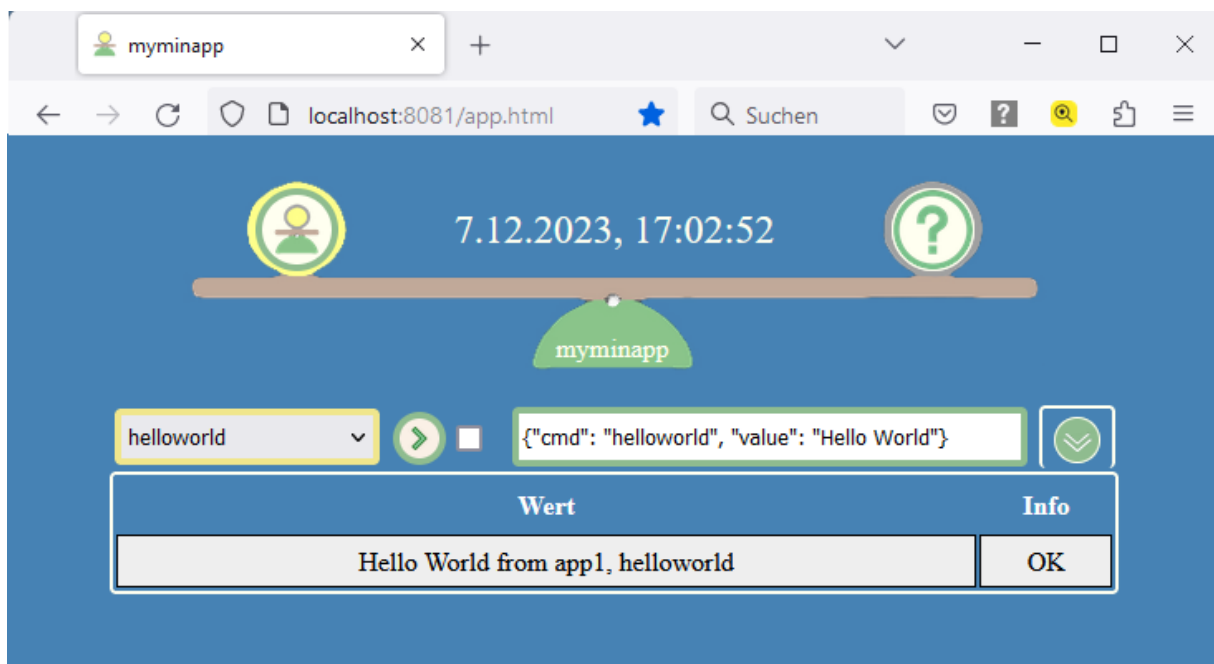
3.3 - Execute 'helloworld' command via web frontend

1. Open a browser and enter the URL in the URL line:

```
http://localhost:8081/app.html
```

2. Select the command 'helloworld' and press the command execution button.

Example image (Mozilla Firefox, myminapp language setting 'de'):



Terminate the application server by entering the following in the terminal:

```
Ctrl+C
```

Step 4 - Execute command 'helloworld' via test class

In the 'home/u1' directory (above myminapp), enter:

```
python3 -m myminapp.test.command.test_helloworld
```

1. Basics

This chapter contains basic information that should be known.

1.1 Application structure

Directory	File	Description
myminapp/		Application directory
	app.py	Module with the 'App' class to be instantiated
	appdef.py	Definition/configuration of the application
	appserver.py	Application server for HTTP(S) requests
/lib/command/	*.py	Supplied and own command classes
/lib/device/*	*.py	Supplied and own device classes
/lib/text/	message.py	Multilingual message texts
/lib/util/	helper.py	Class 'Helper' for auxiliary tasks
/lib/util/	logger.py	Class 'Logger' as standard logger wrapper
/lib/util/	monitor.py	Class 'Monitor' for application instance monitoring
/lib/util/	scheduler.py	Class 'Scheduler' for automation by schedule
/lib/util/	storage.py	Class 'Storage' for data storage access
/web/	*.*	Web frontend files
/data/cert/*	*.*	Certificate files (optional)
/data/log/	*.log	Log files per application instance (app1.log, app2.log, ...)
/data/storage/	*.db	Data files per application instance (app1.db, app2.db, ...)
/data/temp/*	*.*	Temporary, instance-related files
/doc/	*.*	Documentation
/test/command/	test_*.py	Supplied and own command test classes
/test/util/	test_*.py	Utility test classes

1.2 Terms and explanations

Term	Explanation
Module	Python module in the form of a file with name in lower case and extension '.py'. Module name identical, but without extension. Examples: 'helloworld.py', 'test_helloworld.py'; 'helloworld', 'test_helloworld'.
Class	Python class that is defined within a module file (here only one class per module file). Class name corresponds to module name, but without underscore and starting with a capital letter. If there are several name elements, a so-called CamelCase. Examples: 'HelloWorld', 'TestHelloWorld'.
App	Class 'App' of the 'app' module. The application is started with an instance of this class (app1, app2, ...). The 'appdef' module is used for definition/configuration. HTTP(S) requests are processed via the 'appserver' module.
Command	Command for executing a specific task via an instance of the 'App' class. Is implemented by a supplied or custom command class that is derived from the 'Cmd' superclass.
Device	To be understood in the sense of 'device', 'component', 'element' that can be used by command classes. Is implemented by a supplied or custom device class that represents a smart home device, a camera, a diagram generator, an e-mail dispatch or another physical or virtual unit. Device classes form the equipment for command classes.
Utility	Supporting class such as 'Helper'.
Logging	Message output via the 'Logger' class, which in turn uses the Python standard logger. Logging takes place under the name of the application instance. If the log output is specified in files, the output is to '/data/log/' (app1.log, app2.log, ...).
Storage	Data storage for optional storage of command outputs. Access is via the 'Storage' class. The data storage is implemented by the version of SQLite included in Python. An SQLite database is automatically created under '/data/storage/' (app1.db, app2.db, ...) for each application instance on initial access.
Test	<p>There is a test class for each of the utility classes and all supplied command classes, which is derived from 'unittest.TestCase'. If a command makes use of device classes, these are also tested implicitly if the corresponding device class specifications are not commented out in the application definition. Command classes are dynamically instantiated via the 'TestUtil' class.</p> <p>The names of the test modules/test classes correspond to the names of the modules/classes to be tested, with 'test_' or 'Test' prefixed. Examples: 'test_helloworld', 'TestHelloWorld'.</p> <p>For each individual command class, a corresponding test class should be implemented according to the given pattern. This means: Derivation of 'unittest.TestCase', implementation of the public method 'test_command' and the private method '__perform_command'.</p>

1.3 Details

As myminapp is an example application, the source code is an essential component. This manual therefore only provides basic information on the use of myminapp; details can and should be taken from the source code.

In addition, peripheral details can be found in the appendix of this documentation, for example on licenses, certificate import and conceptual aspects of development.

2. Application

The 'Quickstart' chapter describes how myminapp can be installed from the release file. Multiple installations are supported: The myminapp application directory can simply be copied under additional directories ('myminapp_dev/myminapp', 'myminapp_test/myminapp' etc.). When using the application server, simply ensure that different port numbers are used for the individual installations.

Alternatively, myminapp can also be installed with the Package Installer for Python (pip). See chapter 'Installing with pip' in the appendix.

2.1 Application instances

For each installation, the application is executed via one or more instances of the 'App' class, which is coded in the module file 'app.py'. The principle is as follows:

- Instantiation with transfer of an instance number (1 for the first instance, 2 for the second, etc.)
- Call the 'perform_command' method once or several times to execute a command
- Call the 'close' method to close resources and terminate the instance

The example from the 'Quickstart' chapter is used to illustrate this:

```
from myminapp.app import App
app1 = App(1)
app1.perform_command({'cmd':'helloworld', 'value':'Hello World!'})
app1.close()
```

Essentially, an instance performs the following tasks:

Method `__init__`:

- Set the instance name ('app1' for instance number 1, 'app2' for instance number 2 etc.)
- Initialize instance variables
- Open logger and monitor for this instance
- Start the schedulers assigned to this instance

Method `'perform_command'`:

- Assign the command to be executed
- Instantiate the command, if not yet in the cache, and add it to the cache
- Set the input parameters, add the instance name and execute the command
- Receive and process the result
- Process warnings and errors
- Execute logging and monitoring
- Optionally output error-free results to the instance's data storage
- Return the result

Method `'close'`:

- Close the schedulers
- Close the commands in the cache
- Close the logger and the monitor
- Exit the instance

Because logging and data storage are instance-based, instances can be used largely independently of each other. For example, one instance could be used for automation and a second instance for tasks as required, without overlapping logging, data storage or temporary data. In this case, all instances use the same application definition.

2.2 Application definition

The module file 'appdef.py' is used to define/configure the myminapp application.

The module file essentially comprises the following sections:

- Home and data directory, logging and language
- Fields
- Commands
- Devices
- Command presets
- Command scheduler sets

The individual sections are briefly explained below. Details can be found in the module file itself.

Home and data directory, logging and language:

The variables 'HOME' and 'DATA_HOME' are automatically set dynamically when the 'App' class is instantiated. They should not be changed.

Logging is specified via the following variables: 'LOG_STDOUT', 'LOG_FILE', 'LOG_FILE_MAX_BYTES', 'LOG_FILE_BACKUP_COUNT' and 'LOG_LEVEL'.

The variable 'LANG' is used to set the language. Currently 'en' and 'de' are supported.

Fields:

These are data fields and prefix fields that apply across all application layers and are used specifically for commands and data storage.

Data fields are defined in the variable 'FIELD_DEFSETS'. The data fields already defined match the supplied commands, so they should not be changed. Custom fields can be added according to the pattern.

An example of a data field is 'value', which is used for the 'helloworld' command:

```
FIELD_DEFSETS = {
    ...,
    "value": {
        "format": "str", "storageformat": "TEXT",
        "en": {"label": "Value", "desc": "Variable for various purposes"},
        "de": {"label": "Wert", "desc": "Variable für diverse Zwecke"}
    },
    ...,
}
```

The variable 'PREFIX_FIELD_DEFSETS' defines fields that form the prefix for each entity in the data storage. These definitions should not be changed.

Commands:

The variable 'COMMAND_DEFSETS' is used to specify the supplied and custom commands. The specification for command 'helloworld' is cited as an example:

```
COMMAND_DEFSETS = {
    ...,
    "helloworld": {
        "module": "myminapp.lib.command.helloworld", "class": "HelloWorld",
        "input": ["value"],
        "output": ["value", "info"],
        "storage": True, "logging": True
    },
    ...,
}
```

The explicit module name and the class name must be entered in the first line under the name of the command. This data is used to dynamically instantiate commands.

The input fields are to be specified as a list in 'input', the output fields in 'output'. The 'info' field is generally required for a uniform output format across all commands.

'storage' can be used to determine whether error-free command results should be written to the data storage, i.e. to the entity with the name of the command (in this case 'helloworld'). 'logging' is used to specify whether a log entry should be written for the command call regardless of the log level.

Devices:

The variables 'NAME_DEVICE_MAP' and 'DEVICE_CLASS_CONNECTION_SPEC' are used to specify devices.

Device-specific data is assigned to a name in 'NAME_DEVICE_MAP'. The name must be unique, should consist of capital letters and should not be longer than about 10 characters; underscores can be used if necessary. An example of this is a device called 'TV', which is assigned a Fritz switch socket, which is represented by the device class 'Fritz':

```
NAME_DEVICE_MAP = {
    ...,
    "TV": {"devclass": "Fritz", "devcat": "P", "devid": "12345 1234567"},
    ...,
}
```

In 'DEVICE_CLASS_CONNECTION_SPEC', connection data is assigned to a device class specified in 'NAME_DEVICE_MAP'. The following example refers to the device class 'Fritz', which connects to a Fritzbox:

```
DEVICE_CLASS_CONNECTION_SPEC = {
    ...,
    "Fritz": {"conntype": "hub", "address": "192.168.178.1", "port": 49000,
              "user": "fritzuser", "password": "..."},
    ...,
}
```

Command presets:

The 'COMMAND_PRESETS' variable supports convenient command execution. This is used in the web frontend, for example, when a command is selected there, as shown in the 'Quickstart' chapter for the 'helloworld' command. The preset defined for this is cited as an example:

```
COMMAND_PRESETS = {
    ...,
    "helloworld": {"cmd": "helloworld", "value": "Hello World"},
    ...,
}
```

Command scheduler sets:

For automation purposes, the variable 'COMMAND_SCHEDSETS' can be used to specify the time-controlled execution of commands by referencing command presets that were previously created in 'COMMAND_PRESETS'. As a pro forma example, 'helloworld' is also used here, even if the automation of this command is unlikely to make much sense:

```
COMMAND_SCHEDSETS = {
    ...,
    "schedule_1": { "cmdpreset": "helloworld", "days": "1,2",
                    "time": "12:59:01", "appnum": 1},
    "schedule_2": {"cmdpreset": "helloworld", "days": "3,4,5,6,7",
                    "intervalunit": 'h', "appnum": 1},
    "schedule_3": { "cmdpreset": "helloworld", "days": "*",
                    "time": "10:00:00-03:00:00", "intervaldiff": 30, "appnum": 1},
    ...,
}
```

For an explanation of this specifications, see the information on the scheduler in the 'Utilities' chapter.

2.3 Application server

The myminapp application can optionally be executed via the application server, which is coded in the module file 'appserver.py'. The application server processes GET requests via HTTP(S). For examples, please refer to the 'Quickstart' chapter.

The 'appserver' module has a main method with optional call parameters:

Parameters	Explanation
-h, --help	Displays a help text for calling the server.
-appnum	Instance number with which the application is instantiated. The default is 1.
-host	Host on which the server can be reached. The default is 127.0.0.1 alias 'localhost'. If the server is to be accessible from outside the host, the explicit IP address (or the host name) must be specified, for example 192.168.178.26.
-port	Port under which the server can be reached. If the server is to be used for several application instances in parallel, the port number must be varied each time the server is started, for example 8081 for instance 1, 8082 for instance 2 etc.
-cert	Only for HTTPS: Certificate file, for example './myminapp/data/cert/cert.pem'.
-key	Only for HTTPS: Key file, for example './myminapp/data/cert/key.pem'.

Example call with specification of host and port:

```
python3 -m myminapp.appserver -host 192.168.178.26 -port 8082
```

The application server essentially performs the following tasks:

- Instantiate the 'App' class, taking into account the optional call parameters
- Receive and check HTTP(S) GET requests
- Recognize requests from the web frontend by the prefix 'web='
- Convert query data from type 'JSON' to Python type 'dict'



- Call the 'perform_command' method on the application instance
- Process result data, with special preparation for requests from the web frontend
- Convert the result data from Python type 'dict' to type 'JSON'
- Send the reply
- Close the application instance when entering Ctrl+C in the terminal

2.4 Web frontend

The web frontend requires a running application server. If the server is running locally and was started with default values, the following must be entered in the browser: `http://localhost:8081/app.html`.

If the server is running on a different host, the explicit IP address or host name must be specified, for example: `http://192.168.178.26:8081/app.html`. In this case, the browser will indicate that the connection is not secure. In a home network, this warning can normally be tolerated and the connection can still be established.

However, if the warning is to be avoided, communication must be organized via HTTPS by using appropriate certificate files. Self-signed certificate files (for host IP 192.168.178.26) and a self-created CA certificate are available under '/data/cert/' for this purpose. Further information on this can be found in the appendix in the chapter 'Creating certificates'.

It should be noted at this point that the myminapp application is only suitable for use in the home network, as it does not have the security features required for direct public access. However, if a user has configured a suitable router to access his home network from outside in a secure manner, the myminapp application can of course also be accessed in this way.

Note: The example screens shown below were created with the German language setting (LANG = 'de' in 'appdef.py').

2.4.1 Operation

The use of the web frontend is deliberately kept simple and is particularly convenient if command presets have been defined.

The main operating elements are marked with the numbers 1 to 7 in the following illustration and explained below.

The screenshot shows the myminapp web frontend interface. At the top, there is a status bar with a user icon (1), the date and time '9.12.2023, 12:05:09', and a help icon (2). Below this is a green 'myminapp' logo. The main interface features a dropdown menu (3) set to 'state_2', a green play button (4), and a command input field (5) containing '{"cmd": "state", "devname": "SOLAR, DESKTOP, FRIDGE, TV, WASHER"}'. To the right of the input field is a green checkmark button (6). Below these elements is a table (7) displaying the status of various devices.

Name	Kategorie	Zustand	Leistung W	Energie kWh	°C	Info
SOLAR	P	ON	35.4	687.783	5.0	OK
DESKTOP	P	ON	28.2	68.756	20.5	OK
FRIDGE	P	ON	3.0	84.058	24.0	OK
TV	P	ON	2.4	42.824	18.5	OK
WASHER	P	ON	0.2	9.333	22.0	OK

About the controls:

- 1 = Button to open an info box
- 2 = Button to open the manual in a new browser tab
- 3 = Selection of a command preset. The command is transferred to the input line (5)
- 4 = Button for executing the command, checkbox for repeated execution every 15 seconds
- 5 = Input line
- 6 = Download the data output
- 7 = Data output

2.4.2 Examples

The use of the web frontend is shown below using various commands as examples. An example scenario is assumed that matches the supplied commands, devices and specifications.

Specification of the command 'state' in 'COMMAND_DEFSETS':

```
"state": {  
  "module": "myminapp.lib.command.state", "class": "State",  
  "input": ["devname", "devcat"],  
  "output": ["devname", "devcat", "state", "power", "energy", "temp", "info"],  
  "storage": True, "logging": False  
},
```

Presets involved in the execution of the 'state' command in 'COMMAND_PRESETS':

```
"state_1": {"cmd": "state", "devname": "EM"},  
"state_2": {"cmd": "state", "devname": "SOLAR, DESKTOP, FRIDGE, TV, WASHER"},
```

These presets can be used to query the status data of the energy meter and various switchable sockets.

Participating scheduling entries in 'COMMAND_SCHEDSETS':

```
"schedule_1": {"cmdpreset": "state_1", "days": "*", "intervalunit": 'm', "appnum": 1},  
"schedule_2": {"cmdpreset": "state_2", "days": "*", "intervalunit": 'm', "appnum": 1},
```

These scheduling entries are used to execute the command specified with 'state_1' and 'state_2' at the start and end of a 5-minute section (see also the 'Scheduler' section).

As the option 'storage' True has been set for the command 'state', the application instance ensures that the output data of the command is stored in the instance-related data storage. The name of the entity/database table always corresponds to the name of the command, in this case 'state'.

In principle, the application instance can run 24/7 so that data is continuously collected for statistical analysis.

Specification of the 'epstats' command:

```
"epstats": {  
  "module": "myminapp.lib.command.epstats", "class": "EPStats",  
  "input": ["type", "entity", "units", "from", "to", "devname", "devcat"],  
  "output": ["fields", "entries", "charts", "info"],  
  "storage": False, "logging": False  
},
```

Presets involved in the execution of the 'epstats' command:

```
"epstats_1_power": {"cmd": "epstats", "type": "power", "entity": "state",  
                    "units": 5, "from": "4h", "to": "*",  
                    "devname": "EM"},  
  
"epstats_2_power": {"cmd": "epstats", "type": "power", "entity": "state",  
                    "units": 4, "from": "4h", "to": "*",  
                    "devname": "SOLAR, DESKTOP, FRIDGE, TV, WASHER"},
```

These presets can be used to determine statistical data for the energy meter and various switch sockets from the database table/entity 'state'.

A brief explanation of some parameter values (see source code in 'epstats.py' for details):

- **type** - 'power' for average power in watts, 'energy' for consumption in kilowatt hours, or 'powerplus' (only with "devcat": "M") for average feed-in power in watts, for example with excess solar energy
- **units** - Number of time units for grouping the data output: 1=year; 2=year, month; 3=year, month, day; 4=year, month, day, hour; 5=year, month, day, hour, minute
- **from** - Complete or abbreviated timestamp as a number ('20231210150101', '20231210'). Alternatively, number of days, hours or minutes ('5d', '4h', '30m') in the past starting from 'to', if a timestamp or asterisk is specified there
- **to** - Complete or abbreviated timestamp as for 'from', or asterisk (*) for the current time. Alternatively, number of days, hours or minutes ('5d', '4h', '30m') into the future starting from 'from' if a timestamp is specified there
- **devname** - Device name or CSV name list

The following excerpt shows the data output after selecting 'epstats_1_power'. This assumes that the 'state' command was previously executed automatically via scheduling.



The screenshot shows the myminapp interface with a user profile icon, the date and time '9.12.2023, 11:46:46', and a help icon. Below the header is a dropdown menu set to 'epstats_1_power' and a search bar containing the command: {"cmd": "epstats", "type": "power", "entity": "state", "units": 5, "from": "4h", "to": "*", "devname": "EM"}. The main part of the interface is a table with 11 columns: Jahr, Monat, Tag, Stunde, 5 Minuten, Name, Kategorie, Anzahl, Min., Max., and Leistung W. The table displays 20 rows of data for the year 2023, month 12, day 9, showing power consumption in watts over 5-minute intervals.

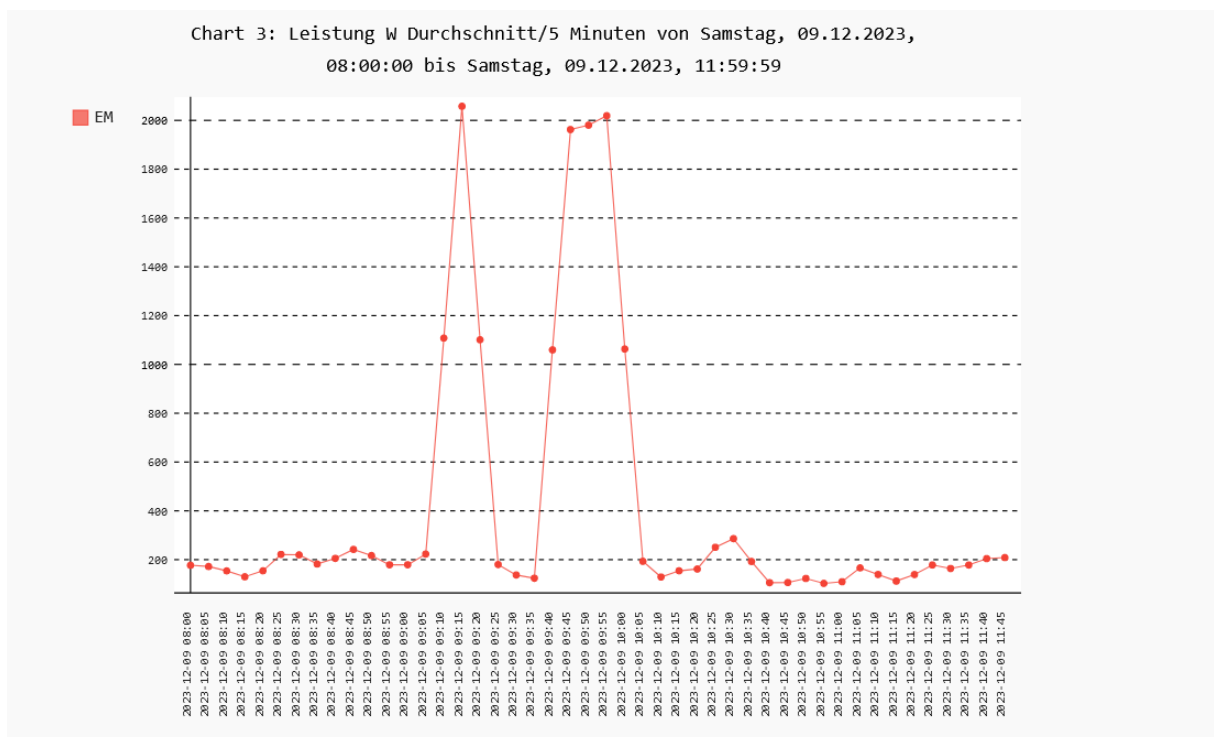
Jahr	Monat	Tag	Stunde	5 Minuten	Name	Kategorie	Anzahl	Min.	Max.	Leistung W
2023	12	9	8	0	EM	M	2	175.9	178.4	177.2
2023	12	9	8	5	EM	M	2	167.6	176.8	172.2
2023	12	9	8	10	EM	M	2	139.8	168.2	154.0
2023	12	9	8	15	EM	M	2	120.1	139.8	130.0
2023	12	9	8	20	EM	M	2	119.8	188.7	154.3
2023	12	9	8	25	EM	M	2	189.1	253.2	221.2
2023	12	9	8	30	EM	M	2	190.7	248.9	219.8
2023	12	9	8	35	EM	M	2	174.0	190.8	182.4
2023	12	9	8	40	EM	M	2	168.2	241.7	205.0
2023	12	9	8	45	EM	M	2	241.8	242.1	242.0
2023	12	9	8	50	EM	M	2	192.5	241.7	217.1
2023	12	9	8	55	EM	M	2	164.6	192.8	178.7
2023	12	9	9	0	EM	M	2	165.4	192.3	178.9
2023	12	9	9	5	EM	M	2	181.1	264.7	222.9
2023	12	9	9	10	EM	M	2	185.1	2030.9	1108.0

According to the command preset, statistical data was determined over 4 hours back in time starting from the current time, which is visible at the top of the image ("from": "4h", "to": "*"). These are the hours 11, 10, 9 and 8 of December 9, 2023.

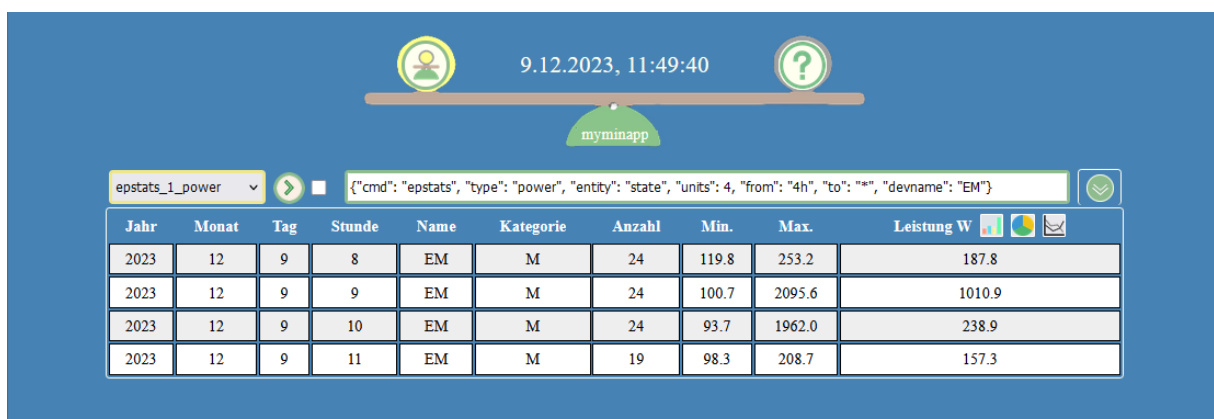
The data output was grouped according to the 5 time units year, month, day, hour and minute ("units": 5).

The data was previously recorded continuously in 5-minute sections, which is also taken into account here in the data output. The value 2 in the 'Number' column confirms that 2 data records were written per 5 minutes, one data record at the beginning and one at the end of the respective time period.

A special feature is the ability to display and interact with SVG chart files using buttons. The 'epstats' command provides the data for this and the application server prepares the data output accordingly. The buttons for bar, pie and line charts are visible in the header in the 'Power W' column. The following illustration shows the line chart for the data output shown above in another browser tab.



A change from "units": 5 to "units": 4 in the input line causes grouping by hours:



The next example shows statistical data determined with preset 'epstats_2_power'. In this example, the data was also determined for the last 4 hours and the grouping is based on hours.

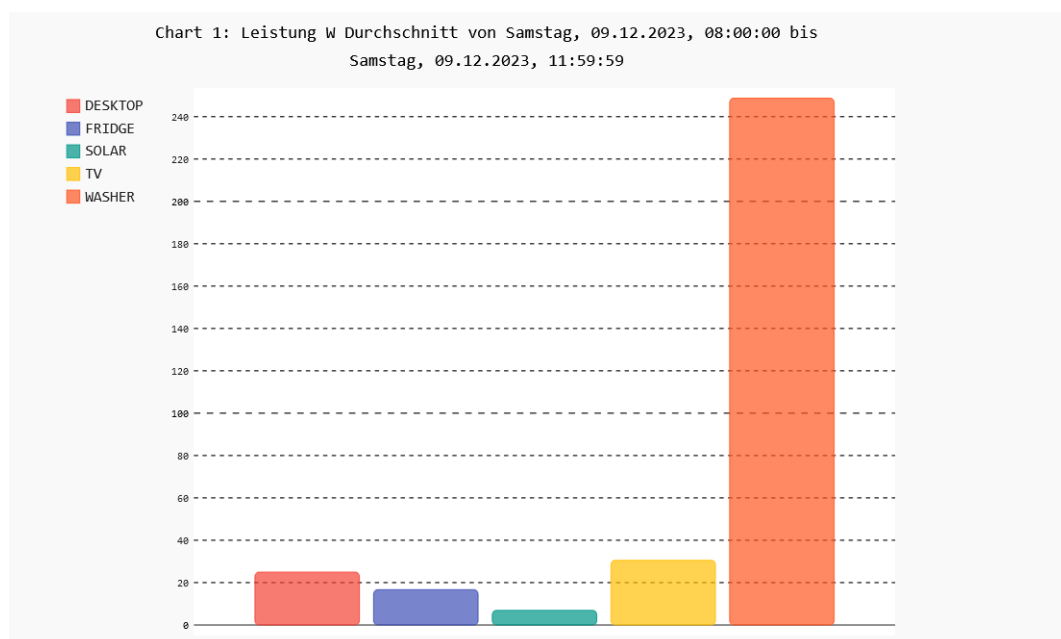
In contrast to the previous example, in which only the one device 'EM' was taken into account, the devices 'DESKTOP', 'FRIDGE', 'SOLAR', 'TV' and 'WASHER' are taken into account here. The power data of the 'WASHER' device is particularly noticeable in hour 9. This correlates with the total consumption measured via the 'EM' device in the previous example. In addition, bar, pie and line charts are shown for the relevant period.



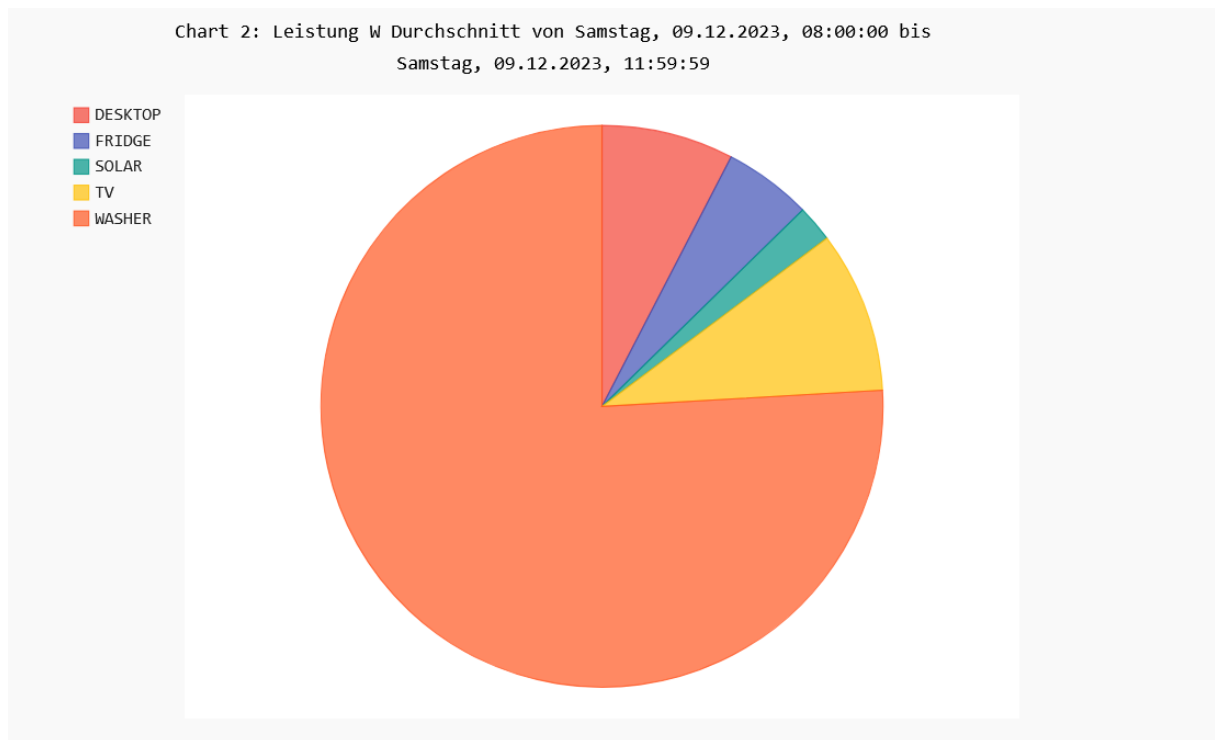
The screenshot shows the myminapp interface with a user profile icon, a clock showing 9.12.2023, 11:56:15, and a help icon. Below the header, there is a dropdown menu set to 'epstats_2_power' and a command input field containing: `{ "cmd": "epstats", "type": "power", "entity": "state", "units": 4, "from": "4h", "to": "", "devname": "SOLAR, DESKTOP, FRIDGE, TV, WASHER" }`. The main part of the interface is a table with the following columns: Jahr, Monat, Tag, Stunde, Name, Kategorie, Anzahl, Min., Max., and Leistung W. The table displays data for the year 2023, month 12, day 9, from hour 8 to hour 10. The 'WASHER' device shows a significant peak in power consumption at hour 9.

Jahr	Monat	Tag	Stunde	Name	Kategorie	Anzahl	Min.	Max.	Leistung W
2023	12	9	8	DESKTOP	P	24	2.1	2.2	2.2
2023	12	9	8	FRIDGE	P	24	2.9	57.4	10.1
2023	12	9	8	SOLAR	P	24	0.1	0.2	0.2
2023	12	9	8	TV	P	24	2.4	71.2	33.6
2023	12	9	8	WASHER	P	24	0.2	0.2	0.2
2023	12	9	9	DESKTOP	P	24	2.1	39.0	32.1
2023	12	9	9	FRIDGE	P	24	2.8	63.1	20.4
2023	12	9	9	SOLAR	P	24	0.1	3.6	1.0
2023	12	9	9	TV	P	24	2.4	70.5	39.0
2023	12	9	9	WASHER	P	24	0.2	1912.0	868.4
2023	12	9	10	DESKTOP	P	24	30.6	50.6	34.8
2023	12	9	10	FRIDGE	P	24	2.9	62.8	14.4
2023	12	9	10	SOLAR	P	24	3.6	11.5	7.6
2023	12	9	10	TV	P	24	2.4	23.7	6.0
2023	12	9	10	WASHER	P	24	0.2	1833.8	115.8

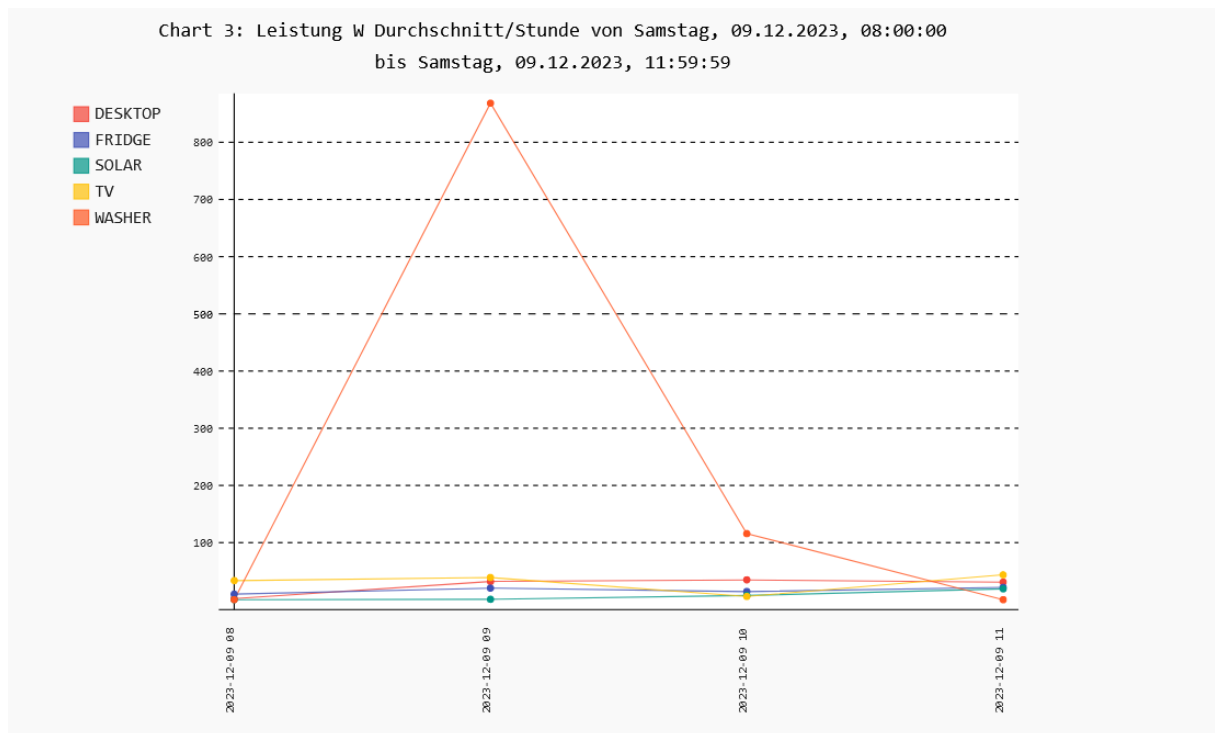
Bar chart:



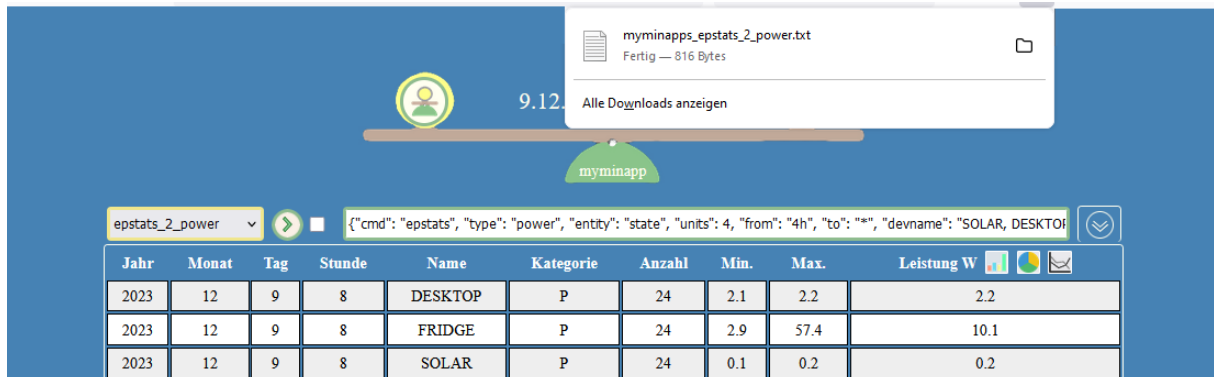
Pie chart:



Line chart:

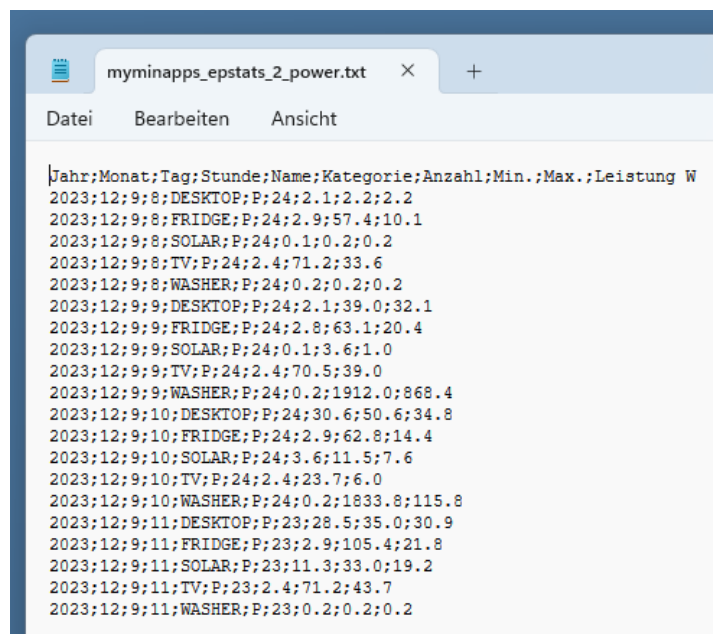


To download a data output, click on the button directly above the table on the right shown in the following image. The download takes place as a CSV file with a semicolon separator to the destination specified by the browser settings for downloads.



The screenshot shows the myminapp interface. At the top, there's a user profile icon and the date '9.12.'. Below it, a green bar with the 'myminapp' logo. A dropdown menu is open, showing 'epstats_2_power' and a button to download the file. Below the dropdown is a table with columns: Jahr, Monat, Tag, Stunde, Name, Kategorie, Anzahl, Min., Max., and Leistung W. The table contains three rows of data for December 9, 2023, at 8:00.

Jahr	Monat	Tag	Stunde	Name	Kategorie	Anzahl	Min.	Max.	Leistung W
2023	12	9	8	DESKTOP	P	24	2.1	2.2	2.2
2023	12	9	8	FRIDGE	P	24	2.9	57.4	10.1
2023	12	9	8	SOLAR	P	24	0.1	0.2	0.2



The screenshot shows a text editor window titled 'myminapps_epstats_2_power.txt'. The content is a CSV file with the following data:

```
Jahr;Monat;Tag;Stunde;Name;Kategorie;Anzahl;Min.;Max.;Leistung W
2023;12;9;8;DESKTOP;P;24;2.1;2.2;2.2
2023;12;9;8;FRIDGE;P;24;2.9;57.4;10.1
2023;12;9;8;SOLAR;P;24;0.1;0.2;0.2
2023;12;9;8;TV;P;24;2.4;71.2;33.6
2023;12;9;8;WASHER;P;24;0.2;0.2;0.2
2023;12;9;9;DESKTOP;P;24;2.1;39.0;32.1
2023;12;9;9;FRIDGE;P;24;2.8;63.1;20.4
2023;12;9;9;SOLAR;P;24;0.1;3.6;1.0
2023;12;9;9;TV;P;24;2.4;70.5;39.0
2023;12;9;9;WASHER;P;24;0.2;1912.0;868.4
2023;12;9;10;DESKTOP;P;24;30.6;50.6;34.8
2023;12;9;10;FRIDGE;P;24;2.9;62.8;14.4
2023;12;9;10;SOLAR;P;24;3.6;11.5;7.6
2023;12;9;10;TV;P;24;2.4;23.7;6.0
2023;12;9;10;WASHER;P;24;0.2;1833.8;115.8
2023;12;9;11;DESKTOP;P;23;28.5;35.0;30.9
2023;12;9;11;FRIDGE;P;23;2.9;105.4;21.8
2023;12;9;11;SOLAR;P;23;11.3;33.0;19.2
2023;12;9;11;TV;P;23;2.4;71.2;43.7
2023;12;9;11;WASHER;P;23;0.2;0.2;0.2
```

A few more examples are shown below without commentary.

Preset 'state_2': Current status of various devices

"state_2": {"cmd": "state", "devname": "SOLAR, DESKTOP, FRIDGE, TV, WASHER"},



The screenshot shows the myminapp interface. At the top, there's a user profile icon and the date '9.12.2023, 12:05:09'. Below it, a green bar with the 'myminapp' logo. A dropdown menu is open, showing 'state_2' and a button to download the file. Below the dropdown is a table with columns: Name, Kategorie, Zustand, Leistung W, Energie kWh, °C, and Info. The table contains five rows of data for various devices.

Name	Kategorie	Zustand	Leistung W	Energie kWh	°C	Info
SOLAR	P	ON	35.4	687.783	5.0	OK
DESKTOP	P	ON	28.2	68.756	20.5	OK
FRIDGE	P	ON	3.0	84.058	24.0	OK
TV	P	ON	2.4	42.824	18.5	OK
WASHER	P	ON	0.2	9.333	22.0	OK

Preset 'psdischarge_1': Conditional discharge of a power station

```
"psdischarge_1": { "cmd": "psdischarge", "value": "minlevel=42"},
```

Wert	Datei	Objekt	Info
OFF (level=6)	app1-numcam757.bmp		OK

Preset 'pscharge_1': Conditional charging of a power station

```
"pscharge_1": { "cmd": "pscharge", "value": "cmax=50, smin=100"},
```

Wert	Info
cmax=50,ccur=-87,smin=100,scur=38,p1=OFF,p2=OFF,p3=OFF	OK

Preset 'plug_OFF': Switching off a switched socket outlet

```
"plug_OFF": {"cmd": "setting", "devname": "PLUG", "value": "OFF"},
```

Name	Kategorie	Wert	Info
PLUG	P	OFF	OK

Free selection based on preset 'select_1'

```
"select_1": {"cmd": "selection", "value": "SELECT * FROM state WHERE id = 1;"},
```

Adjustments have been made to the selected preset in the command line:

```
{\"cmd\": \"selection\", \"value\": \"SELECT * FROM state WHERE t1 = 2023 AND t2 = 12 AND t3 = 9 AND t4 = 11;\"}
```

In principle, any selections can be prepared, adapted if necessary and executed using all database tables/entities in the database of the application instance.

3. Commands

Each command is briefly described below under its module name. Details can be found in the respective module file `/lib/command/<module name>.py` and in the application definition.

To create your own command classes, see chapter 'Create your own classes'.

Command	Explanation
cmd	Super class for all supplied and own commands.
epstats	Creates statistical data on energy and power on the basis of data that has been automatically determined by command 'state', for example. Uses the 'Chart' device class to generate bar charts, pie charts and line charts.
helloworld	'Hello World' implementation. The source code can serve as a simple template for your own command classes.
monitorview	Display of the current monitoring data of an application instance. Uses the 'Monitor' utility.
pscharge	Automatic charging of a power station via disused laptop power supplies. Uses the device classes 'DTSU' and 'Fritz' to obtain current power data, as well as the device class 'Bosch' for switching the power supply units via switching sockets.
psdischarge	Automatic discharging of a Powerstation for night-time base load coverage. Uses the device class 'Numcam757' to detect the level of a power station from the display and the device class 'Relay5V' to switch a small USB consumer on the power station to activate the display lighting and switch the current flow from the power station to the inverter.
selection	Free data selection from any entities in the data storage.
setting	Cross-product switching of switchable sockets and relays and setting of inverter limits. Uses the device classes 'Fritz', 'Bosch', 'Nohub', 'Relay5V' and 'OpenDTU'.
state	Cross-product determination of status values of switchable sockets, relays, energy meters and inverters. Uses the device classes 'Fritz', 'Bosch', 'Nohub', 'Relay5V', 'DTSU' and 'OpenDTU'. Can be used automatically to create the basis for statistical data (see 'epstats' command).
textdoc	Generates simple Python docstring documentation in the form of text files that are saved under <code>/data/temp/textdoc/myminapp</code> .

The following commands do not use a device, therefore have no external dependencies and can be used directly after installation:

- helloworld
- monitorview
- textdoc

The following command is applicable as soon as a database table/entity exists in the data storage:

- selection

4. Devices

Each device is briefly described here under its module name. Details can be found in the respective module file `/lib/device/<subdirectory>/<module name>.py` and in the application definition. If external dependencies exist, this is indicated in each case.

To create your own device classes, see chapter 'Create your own classes'.

Device	Explanation
camera.numcam757	<p>Determines the fill level of a Powerstation 'Anker 757' using a simple small HD camera positioned in front of the display of the Powerstation. Returns the fill level as a number (0 to 100) if the image is successfully captured, otherwise -1. The image can optionally be saved as a bitmap file, for example as with command 'psdischarge' under <code>/data/temp/camera/app<n>-numcam757.bmp</code>.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• External package 'pygame' (pip install pygame)• Simple noname HD camera• Powerstation 'Anker 757'
graphics.chart	<p>Creates bar charts, pie charts and line charts as SVG files (Scalable Vector Graphics), which are saved under <code>/data/temp/chart/</code>.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• External package 'pygal' (pip install pygal)
inverter.opendtu	<p>Communicates via HTTP with OpenDTU hardware to receive status data of a 'Hoymiles' inverter or to set an inverter limit.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• External package 'requests' (pip install requests)• Hardware as described under https://github.com/tbnobody/OpenDTU• Hoymiles' inverters, for example 'HM 300' or 'HM 600'
meter.dtsu	<p>Communicates via Modbus RTU with an energy meter that is integrated at a central location in the house's power grid, typically in the distribution cabinet. The current total power in watts (plus and minus) and the total consumption in kilowatt hours are read.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• External module 'minimalmodbus' (pip install minimalmodbus)• Energy meter 'Chint DTSU666'
plug.bosch	<p>Communicates via HTTPS with a hub of the type 'Bosch Smart Home Controller II' to read status data from switchable sockets and switch them.</p> <p>Dependencies:</p> <ul style="list-style-type: none">• External package 'requests' (pip install requests)• 'Bosch Smart Home Controller II'• At least one 'Bosch Smart Plug Compact' switchable socket



plug.fritz	<p>Communicates via HTTP or TR-064 with a 'FRITZ!BOX' hub to read status data from switchable sockets and switch them.</p> <p>Dependencies:</p> <ul style="list-style-type: none"> • External package 'fritzconnection' (pip install fritzconnection) • 'FRITZ!BOX' that supports 'FRITZ!DECT' switchable sockets • At least one 'FRITZ!DECT' type switchable socket ('200' or '210')
plug.nohub	<p>Communicates directly with 'Shelly' switchable sockets via HTTP without a hub in order to read status data from them and switch them.</p> <p>Dependencies:</p> <ul style="list-style-type: none"> • External package 'requests' (pip install requests) • At least one 'Shelly Plus Plug S' switchable socket
relay.relay5v	<p>Communicates via GPIO (General Purpose Input Output) with a relay to read switching states and trigger switching operations.</p> <p>Dependencies:</p> <ul style="list-style-type: none"> • External package 'RPi.GPIO' • 'Raspberry Pi' SBC (Single Board Computer), for example 'Raspberry Pi 4B' • 5 Volt relay, for example 'Elegoo DC 5V'

5. Utilities

In the following, each supplied utility is briefly described under its class name. The respective module file can be found under `/lib/util/<module name>.py`.

5.1 Helper

The 'Helper' class offers various methods that can be used from all application layers. In particular, the following features should be mentioned:

- Obtaining message texts via method 'mtext'
- Obtaining the message level via method 'mlevel'
- Obtaining an exception trace back line via method 'tpline'
- Conversion from and to JSON via the 'dict2json' and 'json2dict' methods
- Various time and calendar-related methods
- Other methods

5.2 Logger

The 'Logger' class is used for message output and in turn uses the Python standard logger (Python module 'logging'). The log levels 'ERROR', 'WARNING', 'INFO' and 'DEBUG' are supported. The specification is made in the application definition. The class is only used by the 'App' and 'TestLogger' classes.

Logging takes place under the name of the application instance. If the log output is specified in files, the output is to `'/data/log/'` (app1.log, app2.log, ...).

The module file `'/lib/text/message.py'` contains message texts with a numeric code in English and German.

The message levels 'ERROR', 'WARNING', 'INFO' and 'DEBUG' are supported. A list of message codes is declared for each of the 'ERROR', 'WARNING' and 'DEBUG' levels, so that a specific message can be classified simply by adding its code to one of the lists. For messages with codes that do not appear in any of the three lists, level INFO applies.

During logging, the message levels are automatically assigned via the message code.

5.3 Monitor

The 'Monitor' class is used to write status data of the respective application instance to an SQLite database, which is located in the working memory and supports access from multiple threads. Monitoring is particularly useful when processing scheduling records.

See also the information on the 'monitorview' command in the 'Commands' chapter above and the following chapter 'Scheduler'.



5.4 Scheduler

The scheduling records specified in the application definition are processed via the 'Scheduler' class. The class is only used by the 'App' and 'TestScheduler' classes.

When an application instance is started, an instance of the 'Scheduler' class and a concurrent thread are created for each assigned scheduling record. From this, the 'get_action_number' method is called every 3 seconds; if the action number is greater than 0, the scheduled action is executed.

The parameters for the specification of scheduling records are described below.

Parameters	Explanation
cmdpreset	Name of a command specified in 'COMMAND_PRESET' that is called by the scheduler.
days	Days on which the execution is to take place. The following details are supported: <ul style="list-style-type: none">'*' (asterisk) - all days'0, 1, ..., 7' - CSV list of days (0 or 7 for Sunday, 1 for Monday etc.)
time	Time or time frame for execution. The following formats are supported: <ul style="list-style-type: none">Time for one-time call: 'HH:MM:SS', for example '10:00:00' for 10 o'clock. Is not permitted in combination with parameters 'intervaldiff' and 'intervalunit'Time frame for multiple calls: 'HH:MM:SS-HH:MM:SS', for example '10:00:00-15:00:00' for 10 am to 3 pm. If the second time is less than the first, the following day applies. Requires the specification of the parameter 'intervaldiff' or 'intervalunit' <p>If the 'time' parameter is not specified, the entire day is assumed as the time frame (00:00:00-23:59:59), which makes it necessary to specify the 'intervaldiff' or 'intervalunit' parameter.</p>
intervaldiff	Execution interval in seconds, for example 60 for 60 seconds. Specifications from 5 to 43200 (5 seconds to 12 hours) are permitted. The first execution begins immediately after the application instance is started.
	This parameter is not permitted in combination with a one-time specification for the 'time' parameter.
intervalunit	Execution interval by time unit. This parameter is particularly suitable for determining status data at regular intervals (e.g. with the 'state' command) in order to evaluate it statistically later (e.g. with the 'epstats' command).
	The following information is supported: <ul style="list-style-type: none">'m' - executed at the beginning and end of a 5-minute section'h' - executed every hour at the beginning and end of a full hour'd' - executed daily at the beginning and end of a full day <p>The first execution takes place after the start of the application instance at the beginning of the respective time unit. For example, if the instance was started at 10:12:00, the first execution would take place at 'm' around 10:15:00, at 'h' around 11:00:00 and at 'd' around 00:00:00 (of the following day).</p> <p>This parameter is not permitted in combination with a one-time specification for the 'time' parameter.</p>



appnum	Instance number. This is used to assign the scheduling entry to an application instance, for example 1 for application instance 1 ('app1').
--------	---

Examples of scheduling entries in 'COMMAND_SCHEDSETS':

```
"schedule_1": {"cmdpreset": "state_1", "days": "*", "intervalunit": 'm', "appnum": 1},
"schedule_2": {"cmdpreset": "state_2", "days": "*", "intervalunit": 'h', "appnum": 1},
"schedule_3": {"cmdpreset": "state_3", "days": "*", "intervalunit": 'd', "appnum": 1},

"schedule_4": {"cmdpreset": "state_4", "days": "1,3,5", "time": "18:00:00-03:00:00",
               "intervaldiff": 30, "appnum": 1},

"schedule_5": {"cmdpreset": "helloworld", "days": "7", "time": "15:01:01", "appnum": 1}
```

Note: The continuous, time-related collection of data for subsequent statistical analysis may seem trivial at first glance. On closer inspection, however, there are various aspects that need to be taken into account. This topic is therefore dealt with separately in the appendix under 'Recording data for statistical purposes'.

5.5 Storage

The data storage is implemented by the version of SQLite contained in the Python standard library.

The 'Storage' class is primarily used by the 'App' class (and the 'TestStorage' class). A database is automatically created under '/data/storage/' (app1.db, app2.db etc.) for each application instance when it is first accessed.

In order to automatically save valid command outputs, the value True must be entered for the 'storage' parameter when specifying a command in 'COMMAND_DEFSETS'. A database table (entity) is then automatically created with the name of the command and the data fields specified in the 'output' parameter.

Each entity also receives the set of prefix fields specified in 'PREFIX_FIELD_DEFSETS'. These fields include the primary key ('id') as well as fields for the simple temporal assignment of each entry of an entity ('t0', 't1', ..., 't6').

The 'Storage' class can also be used by command or device classes, as is the case with the 'epstats' and 'selection' commands, for example.



6. Create your own classes

The myminapp application can easily be extended with your own command and device classes using an IDE such as Visual Studio Code. Here are some tips on how to do this.

For command classes:

All command classes are derived from the 'Cmd' superclass in the 'cmd' module. It has the following methods:

- **__init__(self, name:str):** Is called by the derived classes with their class name
- **exec(self, input:dict):** Is overridden by the derived classes to execute the respective command
- **close(self):** Is overridden by the derived classes to close resources
- **add_command_output(self, code:int, text:str, trace:str, dataset:dict, datasets:list):** Used for standardized command output. Is called once or several times by the derived classes in order to provide the respective result in the form of a list (datasets)

The source code of the classes 'HelloWorld' in 'helloworld.py' and 'TestHelloWorld' in 'test_helloworld.py' can serve as an example for deriving your own command class. The very simple source code and the supplementary comments should provide all the essentials.

For device classes:

As device classes each represent a physical or virtual unit in the broadest sense, the code for the implementation can be very individual. Therefore, no superclass has been provided.

When creating them, however, it should generally be noted that device classes are only used by command classes. The corresponding structure, which is given by the existing classes, should not be broken. In addition, to maintain transparency, it is desirable to proceed uniformly with regard to initializing and closing device classes. The following methods should therefore always be implemented:

- **__init__(self, ...):** Implicit call during instantiation, with or without parameters as required
- **close(self):** Close resources if required, otherwise formally

'DTSU' in 'meter.dtsu.py' is recommended as a simple example of a device class.

General:

The source code documentation ('Docstrings') and the separation between public and private methods should also be standardized according to the pattern of the supplied classes in order to maintain the transparency of the myminapp application.

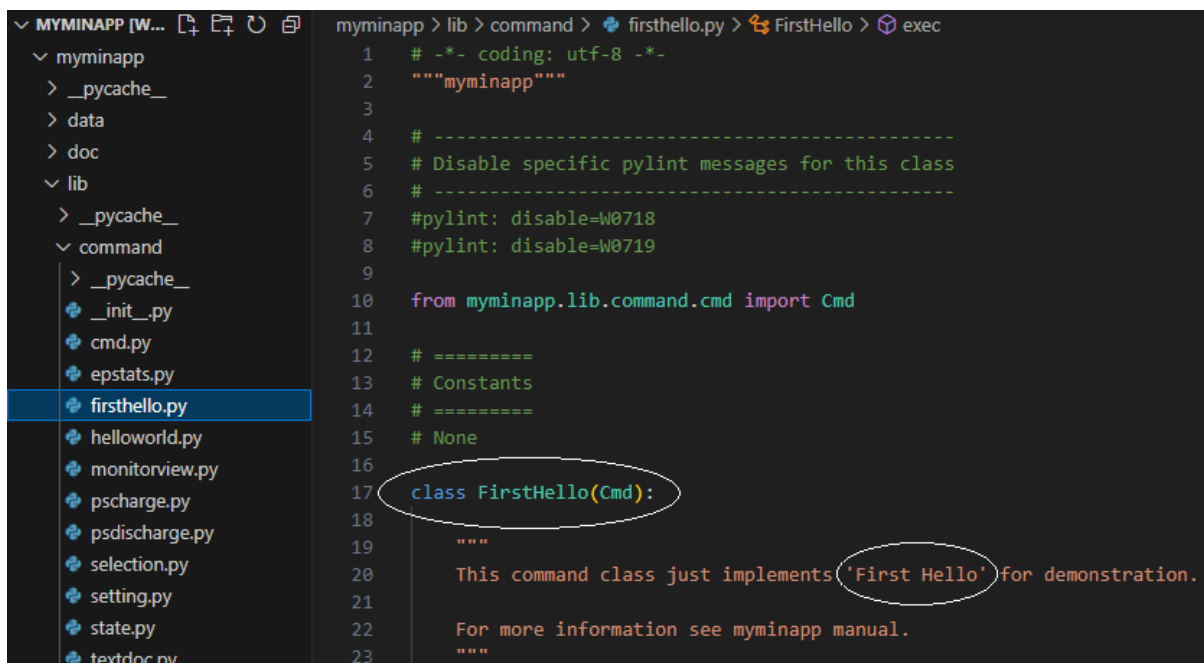
Finally, the required specifications for the new custom command or device class must be added to the application definition.

6.1 First steps

The following is a step-by-step description of how to create your own first command, called 'firsthello'.

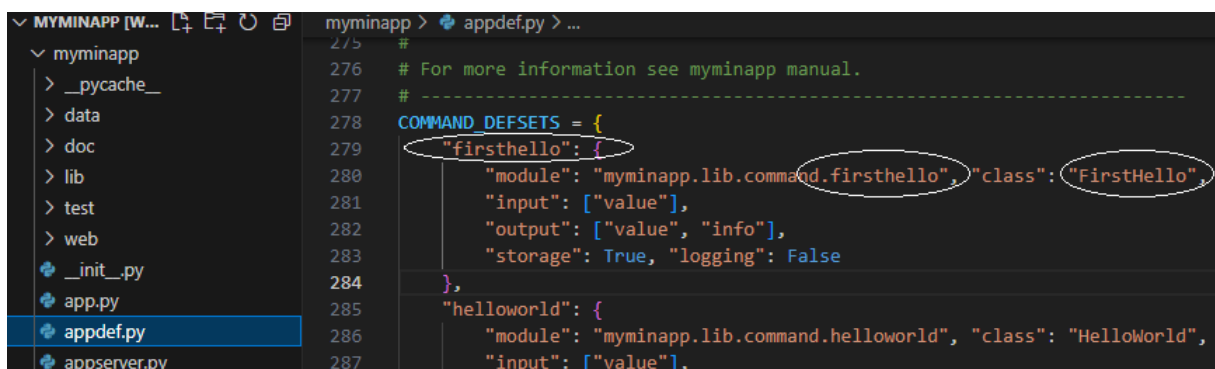
Step sequence 1:

- Open the IDE of your choice (e.g. Visual Studio Code) and open the 'myminapp' directory. Navigate to the 'lib/command' area and copy the file 'helloworld.py' to 'firsthello.py'.
- In the source code of the copied file, change the class name from 'HelloWorld' to 'FirstHello' (case-sensitive) and adjust the docstring text accordingly.



```
1 # -*- coding: utf-8 -*-
2 """myminapp"""
3
4 # -----
5 # Disable specific pylint messages for this class
6 # -----
7 #pylint: disable=W0718
8 #pylint: disable=W0719
9
10 from myminapp.lib.command.cmd import Cmd
11
12 # =====
13 # Constants
14 # =====
15 # None
16
17 class FirstHello(Cmd):
18
19 """
20 This command class just implements 'First Hello' for demonstration.
21
22 For more information see myminapp manual.
23 """
```

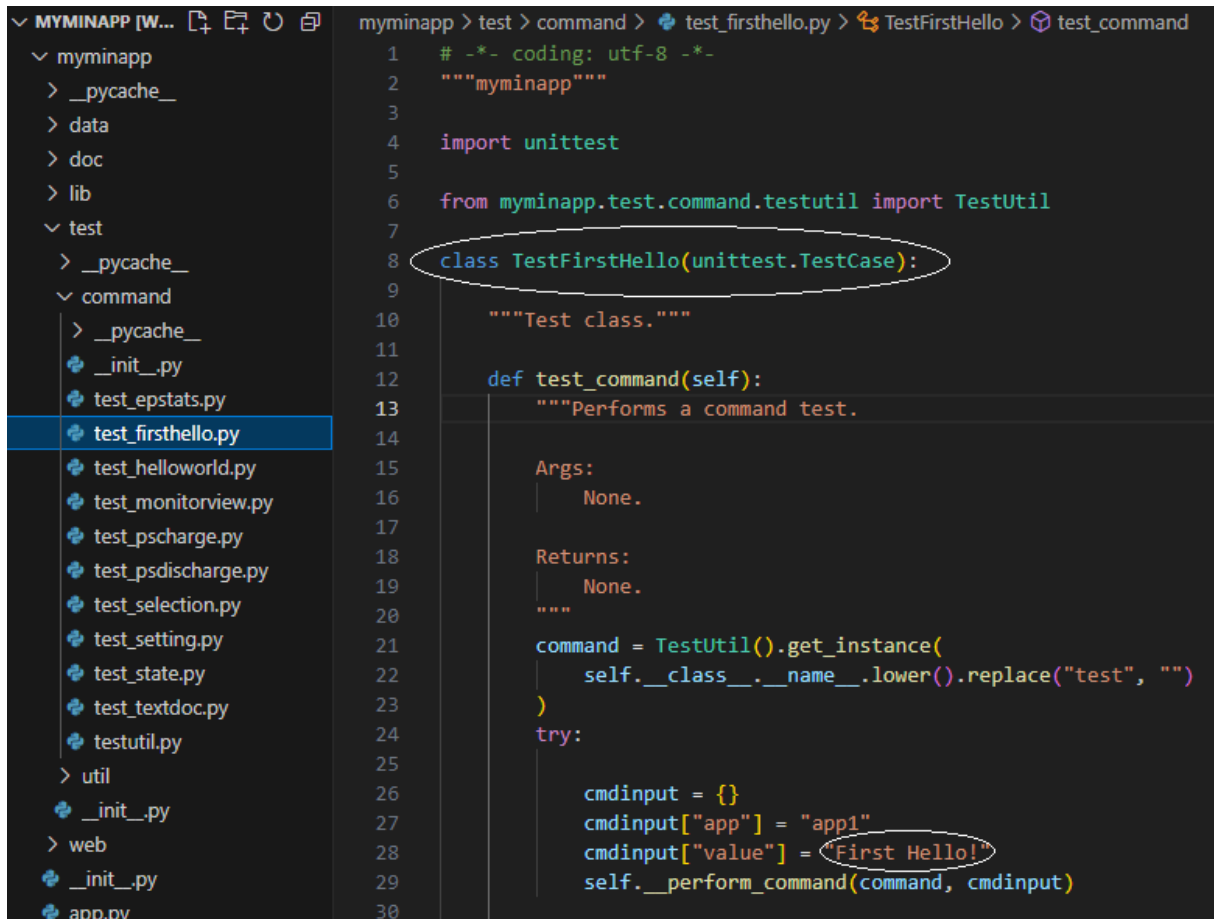
- Navigate to the file 'appdef.py', copy the command specification for 'helloworld' in the source code under 'COMMAND_DEFSSETS' and paste it directly above.
- In the copied area, change the command name 'helloworld' to 'firsthello' and also adapt the module and class names accordingly. Pay particular attention to commas, capitalization, indentation, etc.



```
275 #
276 # For more information see myminapp manual.
277 # -----
278 COMMAND_DEFSSETS = {
279     "firsthello": {
280         "module": "myminapp.lib.command.firsthello", "class": "FirstHello",
281         "input": ["value"],
282         "output": ["value", "info"],
283         "storage": True, "logging": False
284     },
285     "helloworld": {
286         "module": "myminapp.lib.command.helloworld", "class": "HelloWorld",
287         "input": ["value"],
```

Step sequence 2:

- Navigate to the 'test/command' area and copy the file 'test_helloworld.py' to 'test_firsthello.py'.
- In the source code of the copied file, change the class name from 'TestHelloWorld' to 'TestFirstHello' (case-sensitive) and adjust the input parameter 'value'.



```
1  # -*- coding: utf-8 -*-
2  """myminapp"""
3
4  import unittest
5
6  from myminapp.test.command.testutil import TestUtil
7
8  class TestFirstHello(unittest.TestCase):
9
10     """Test class."""
11
12     def test_command(self):
13         """Performs a command test.
14
15         Args:
16             None.
17
18         Returns:
19             None.
20         """
21         command = TestUtil().get_instance(
22             self.__class__.__name__.lower().replace("test", "")
23         )
24         try:
25
26             cmdinput = {}
27             cmdinput["app"] = "app1"
28             cmdinput["value"] = "First Hello!"
29             self.__perform_command(command, cmdinput)
30
```

Once all changes have been saved in the IDE, open a terminal above the 'myminapp' directory and enter:

```
python3 -m myminapp.test.command.test_firsthello
```

The result should then look something like this:

```
...
Result firsthello: [{'code': 0, 'text': 'OK', 'trace': '', 'data': {'value': 'First Hello!
from app1, firsthello', 'info': 'OK'}}]
.
-----
Ran 1 test in 0.006s

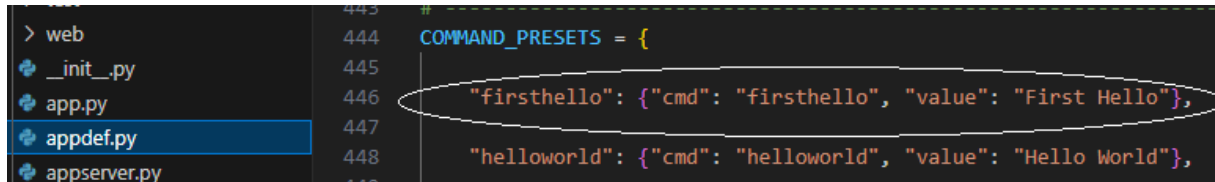
OK
...
```

This means that the first own command has already been successfully implemented and tested!

Step sequence 3:

The new 'firsthello' command can now also be called in the other ways described in the 'Quickstart' chapter for 'helloworld'. To make it directly selectable in the web frontend, a suitable entry must be added to 'appdef.py' under 'COMMAND_PRESETS':

```
"firsthello": {"cmd": "firsthello", "value": "First Hello"},
```

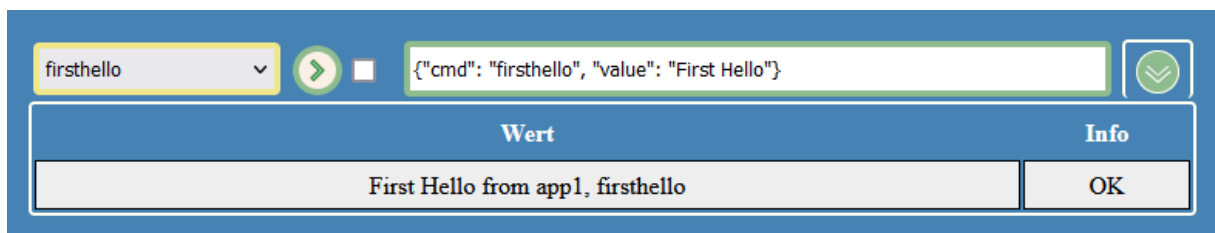


Now start the application server with `python3 -m myminapp.appserver`

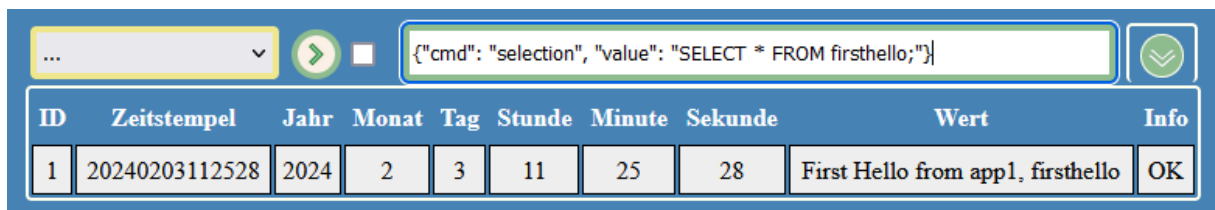
Note: The following images were generated with the language setting `LANG = 'de'` in 'appdef.py'.

```
2024-02-03 11:18:20,294 | INFO | app1 | 900 | Logger 'app1' erstellt mit Handler(n): ['<StreamHandler <stdout> (INFO)>', '<Rotating
FileHandler /home/ui1/dev/myminapp/myminapp/data/log/app1.log (INFO)>']
2024-02-03 11:18:20,296 | INFO | app1 | 100 | Applikationsinstanz 'app1' wurde mit Prozess-ID 22613 gestartet.
2024-02-03 11:18:20,299 | INFO | app1 | 150 | Appserver host: 127.0.0.1, port: 8081, pid: 22613, appnum: 1. Beenden mit Strg+C.
```

Then call up the web frontend in the browser with `http://localhost:8081/app.html` and select and execute the new command:



As the "storage" option was set to True in the command specification for 'firsthello', there is now already an entry in the data storage entity of the command. To select the entry, the supplied command 'selection' is used here as an example and entered directly in the command line:



Close the application server with `Ctrl+C` in the terminal window:

```
2024-02-03 11:25:28,720 | INFO | app1 | 110 | Command 'firsthello' dem Cache hinzugefügt.
2024-02-03 11:37:49,503 | INFO | app1 | 110 | Command 'selection' dem Cache hinzugefügt.
^C2024-02-03 11:44:42,556 | INFO | app1 | 151 | Appserver hat Stop-Anfrage erhalten...
2024-02-03 11:44:42,556 | INFO | app1 | 113 | SchlieÙe Commands im Cache...
2024-02-03 11:44:42,557 | INFO | app1 | 114 | Command 'firsthello' geschlossen.
2024-02-03 11:44:42,557 | INFO | app1 | 114 | Command 'selection' geschlossen.
2024-02-03 11:44:42,557 | INFO | app1 | 117 | Commands geschlossen.
2024-02-03 11:44:42,558 | INFO | app1 | 122 | SchlieÙe Monitor...
2024-02-03 11:44:42,558 | INFO | app1 | 124 | Monitor geschlossen.
2024-02-03 11:44:42,558 | INFO | app1 | 901 | Logger 'app1' wird geschlossen.
```

6.2 Further steps

Automation

If required, 'firsthello' could also be called by schedule by adding a schedule entry in 'appdef.py' under 'COMMAND_SCHEDULESETS'. To call the command daily with an interval of 120 seconds, for example, this could be added:

```
"schedule_hello": {"cmdpreset": "firsthello", "days": "*", "intervaldiff": 120, "appnum": 1},
```

Note: The application server must be restarted after making changes to 'appdef.py'.

Implementation of individual tasks

In order to realize individual tasks, own command classes can be designed practically arbitrarily, provided that this is done according to the pattern described above.

For minimal tasks, a private method such as '__get_result' in 'FirstHello' may be sufficient, or further private methods can be programmed.

Functions that require either packages not included in Python or virtual/physical devices (external dependencies), as well as complex or extensive functions, should generally be represented in a separate device class, which is then used by the command class.

For your own device classes, you can also follow the pattern described above for command classes: copy an existing class, carry out the necessary renaming and then code as required. Furthermore, appropriate additions must be made for device classes in 'appdef.py' under 'NAME_DEVICE_MAP' and 'DEVICE_CLASS_CONNECTION_SPEC'.

Have fun experimenting and implementing your own ideas!



7. Backup

As is generally known, relevant data should be backed up at appropriate intervals. This can be done by simply copying the complete myminapp directory.

Depending on the configuration and use of the myminapp application, there may be locks on files during operation, particularly in the '/data/storage' area. It is then recommended to briefly close the relevant application instances, perform the copy and then restart the application instances.



Appendix

A. Licenses

1. Python

See <https://docs.python.org/3/license.html>

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.12.0 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.12.0 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All Rights Reserved" are retained in Python 3.12.0 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.12.0 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.12.0.
4. PSF is making Python 3.12.0 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.12.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.0, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.12.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

2. myminapp

myminapp

Copyright (c) 2024 - berryunit

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.



THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3. additional libraries/packages/modules

Python 3 version 10 or higher is required to install the 'myminapp' application and run it with parts of the basic equipment supplied. No other libraries/packages/modules are required.

However, supplied device classes require additional Python elements, which is pointed out both in the source code and in this documentation. If the user uses such device classes, it is his responsibility to observe the relevant license terms.



B. Installing with pip

As an example application, myminapp should be installed from the release file in a user directory as described in the 'Quickstart' chapter. This is the easiest way to access the source code, and parallel installations are simple to handle.

Alternatively, myminapp can also be installed from the Python Package Index (PyPI). The installation should be carried out with user rights. To do this, the user (in this case 'u1') has to enter this:

```
pip install myminapp
```

This would install myminapp to the following destination, for example:

```
/home/u1/.local/lib/Pythonversionsnumber/site-packages/myminapp
```

To display the installation, enter:

```
pip show myminapp
```

To uninstall, enter:

```
pip uninstall myminapp
```

C. Creating certificates

Certificates are required for communication via HTTPS. Self-signed certificates can be used for this in the home network, including a CA certificate, which is normally provided by a certification authority. This can be imported into browsers so that they classify the HTTPS connection to the application server as trustworthy and therefore do not issue any warnings or prevent access.

The myminapp application is already supplied with the following self-created certificate files:

- /data/cert/ca/mycacert.crt - CA certificate file
- /data/cert/cert.pem - Certificate file for the application server (192.168.178.26)
- /data/cert/key.pem - Associated key file

The following is an example of how these certificate files are created under Windows using 'Open SSL' can be created yourself.

Step 1: Download and install OpenSSL

The tool 'Win64 OpenSSL v????' Light from '<https://slproweb.com/products/Win32OpenSSL.html>' and install it. Version v3.2.0 was used for the example. After installation, the 'Win64 OpenSSL Command Prompt' is available. Open it and use 'cd' to change to a suitable directory, for example to 'c:/mycerts'. The directories 'ca' and 'ca/private' are required for the example.

The further steps are to be carried out via the 'Win64 OpenSSL Command Prompt' (enter commands without line breaks).

Step 2: Create CA certificate

2.1 Generate private CA key

```
openssl genpkey -algorithm RSA -aes128 -out ./ca/private/mycakey.pem -outform PEM -pkeyopt  
rsa_keygen_bits:2048
```

Result: ./ca/private/mycakey.pem

2.2 Create CA certificate (also called 'root certificate')

```
openssl req -x509 -noenc -sha256 -days 3650 -key ./ca/private/mycakey.pem -out  
./ca/mycacert.crt  
...  
Country Name (2 letter code) [AU]:DE  
State or Province Name (full name) [Some-State]:SH  
Locality Name (eg, city) []:  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ME  
Organizational Unit Name (eg, section) []:  
Common Name (e.g. server FQDN or YOUR name) []:  
Email Address []:
```

Result: ./ca/mycacert.crt

Step 3: Create certificate for application server

3.1 Generate private key

```
openssl genpkey -algorithm RSA -out ./key.pem -outform PEM -pkeyopt rsa_keygen_bits:2048
```

Result: ./key.pem



3.2 Generate certificate signing request (CSR)

This combines data from the private key with data from the 'certificate applicant'. The result is called a certificate request and is used to create the actual certificate.

```
openssl req -new -key ./key.pem -out ./cert.csr
...
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:SH
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ME
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Result: ./cert.csr

3.3 Create extension configuration file 'cert.ext' with Editor

This is used to specify the domains that are actually to be entered as 'Common Name'. Apparently, the specification directly in Common Name has been a candidate for deprecation for a long time, therefore the extension file can generally be used. The pattern for the relevant section is:

```
[alt_names]
IP.1 = 127.0.0.1
IP.2 = x.x.x.x
...
DNS.1 = localhost
DNS.2 = abc.my.domain
DNS.3 = *.my.domain
...
```

Sample content (as used for the supplied certificate files):

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[alt_names]
IP.1 = 192.168.178.26
```

3.4 Create certificate

A certificate is now created and signed on the basis of the CSR and the CA. In addition, a 'mycacert.srl' file is created under './ca'. A 'serial number' is generated in this file, which can be uniquely assigned to the created certificate. This is necessary so that the CA can always generate a new unique number for certificates created in the future.

```
openssl x509 -req -in ./cert.csr -CA ./ca/mycacert.crt -CAkey ./ca/private/mycakey.pem -
CAcreateserial -out ./cert.crt -days 3650 -sha256 -extfile ./cert.ext
...
Certificate request self-signature ok
...
```

Result 1: ./ca/mycacert.srl
Result 2: ./cert.crt



3.5 Create PEM variant from .crt certificate

```
openssl x509 -in ./cert.crt -out ./cert.pem -outform PEM
```

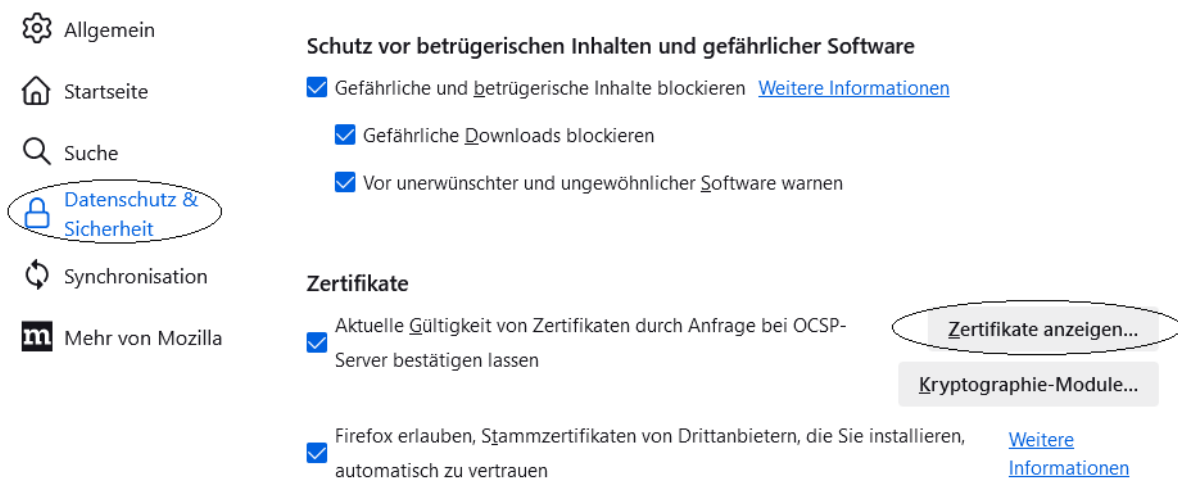
Result: ./cert.pem

All the required files are now available. The supplied files in the '/data/cert/' directory must be replaced with the corresponding files you have created yourself.

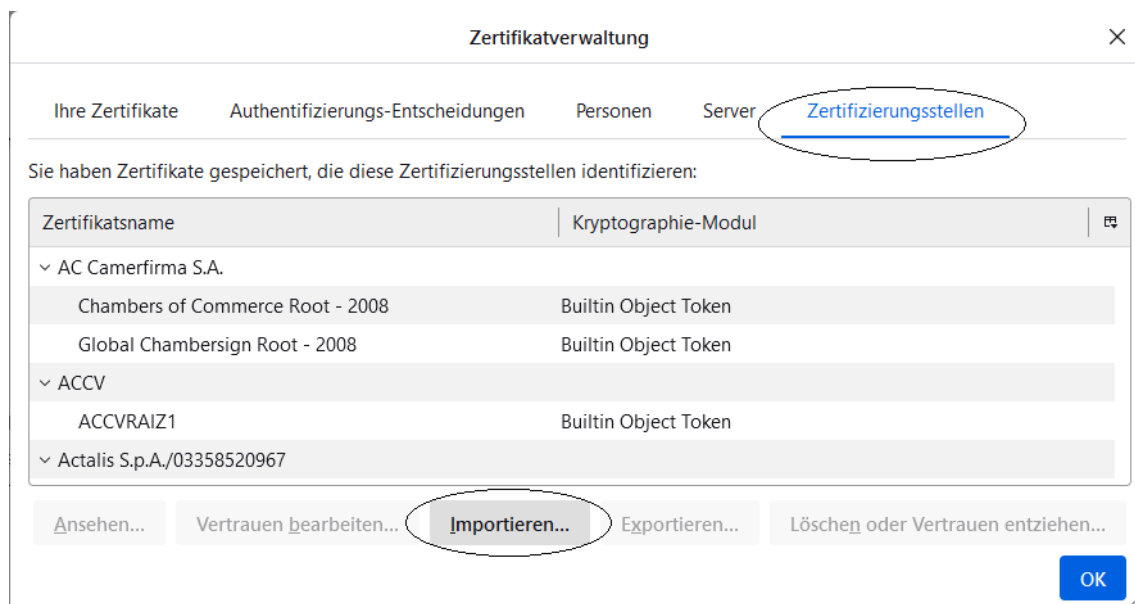
Step 4: Import CA certificate into browser

The process is illustrated using Mozilla Firefox as an example. The process is similar for other browsers.

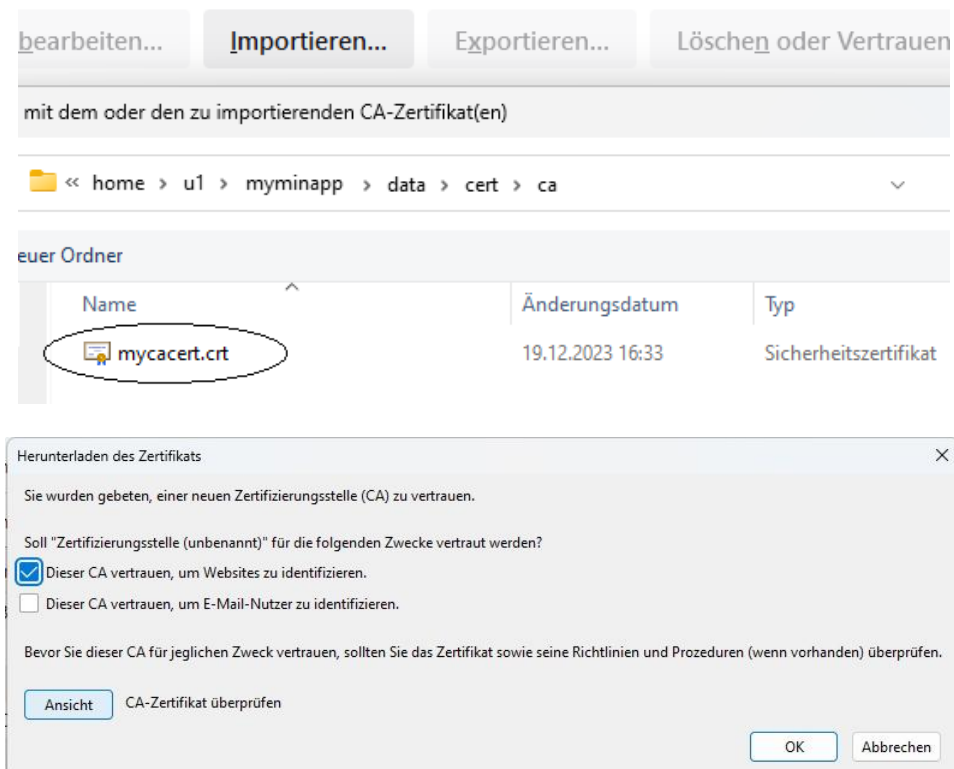
4.1 Navigate to the relevant area via the 'Settings' menu item



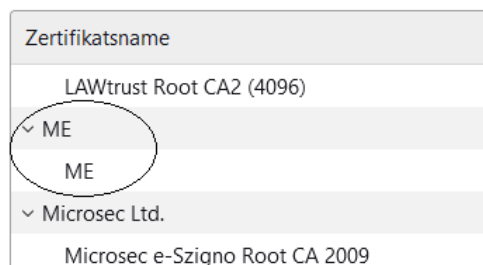
4.2 Press the 'Import' button in the certificate management under 'Certification authorities'



4.3 Importing the 'mycacert.crt' CA certificate file



The imported CA certificate is displayed under the name of the organization, in this case 'ME':



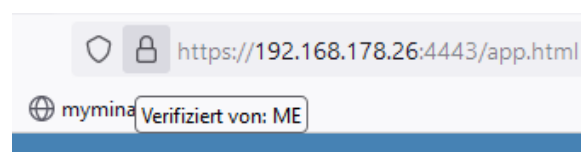
Step 5: Start the application server with the appropriate arguments

The two certificate files 'cert.pem' and 'key.pem' should be copied into the myminapp directory to '/data/cert/' and replace the supplied files there.

Calling the application server can then look like this, for example (without line breaks):

```
python3 -m myminapp.appserver -host 192.168.178.26 -port 4443 -cert  
./myminapp/data/cert/cert.pem -key ./myminapp/data/cert/key.pem
```

The HTTPS connection can now be established in the browser without warnings:



D. Recording data for statistical purposes

Various aspects must be taken into account when continuously recording data over time for subsequent statistical analysis. This is explained below using the recording and statistical evaluation of status data.

The aim in this context was to be able to display consumption and power data in relation to various time units in a practical and reliable manner. The utilities 'Scheduler' and 'Storage' as well as the commands 'state' and 'epstats' are used for this purpose.

1. Time units

Year, month, day, hour and minute have been defined as time units. These time units can be directly assigned to a timestamp such as '2023-12-09 11:05' and its individual elements. Seconds or even milliseconds would not be useful here, as the informative value of the statistics would not be improved in practice; in addition, the update rate of physical units such as switching sockets etc. is often slower.

2. Access

To make accessing stored data via time units quick and easy, an entry prefix has been defined for all entities in the data storage, which is placed in front of each entry and contains both the timestamp and its individual elements. Pro forma, the time unit second was also taken into account here. The prefix consists of 8 fields of type integer:

id - Unique ID (primary key)
t0 - Timestamp, here pro forma including second
t1 - Year
t2 - Month
t3 - Day
t4 - Hour
t5 - Minute
t6 - Second (pro forma)

The formulation of a query can now very easily include a specific year, a specific month, etc. This is possible both in terms of time periods, for example for the first 5 days of December 2023 ($t1 = 2023$ AND $t2 = 12$ AND $t3 > 0$ AND $t3 < 6$), as well as in terms of time units, for example across days for hour 10 in the month of December 2023 ($t1 = 2023$ AND $t2 = 12$ AND $t4 = 10$). It should be noted that hours are counted from 0 to 23 and minutes from 0 to 59.

3. Writing

Another important aspect is the time at which status data is written, taking into account the statistical significance per time unit.

Principle A

If, for example, a certain status were written exactly once an hour, this would be perfectly acceptable at first glance. For example, 3 consumption meter readings, each written at the 30th minute over hours 10 to 12, would be displayed as follows:

Hour 10: 250 kWh
Hour 11: 280 kWh
Hour 12: 307 kWh



However, if the consumption per time unit is to be displayed, this must be determined by the statistical evaluation by forming the difference to the respective preceding entry (307 - 280 = 27 kWh and 280 - 250 = 30 kWh). This causes two problems:

- The difference generally covers two points in time in different time units. This makes the allocation of consumption values in relation to a time unit systematically imprecise.
- If gaps occur in the continuous recording of status data, the difference is no longer related to two directly consecutive time units. This results in misleading statistical values that can hardly be displayed plausibly in a bar or line chart, for example.

Principle B

A much more robust principle is to write the status data at the beginning and end of each time unit. The difference between the two values is therefore always related to the respective time unit, and gaps in the continuous recording do not lead to misleading values, but are also recognizable as recording gaps in the statistical display.

Ideally, the previous example could look something like this:

Hour 10: 240 and 268 kWh (difference 28 kWh)

Hour 11: 268 and 290 kWh (difference 22 kWh)

Hour 12: 290 and 320 kWh (difference 30 kWh)

'Ideal case' here means that the To value of an entry corresponds to the From value of the subsequent entry (in the example 268 and 290 kWh). This should also apply in most cases if the first and last seconds of the time unit are measured or written.

However, it cannot be ruled out that smaller deviations may result, especially if there is a high energy flow at the transition to the next time unit. This could then look as follows:

Hour 10: 240.00 and **268.00** kWh (difference 28.00 kWh)

Hour 11: **268.05** and 290.00 kWh (difference **21.05** kWh)

Hour 12: 290.00 and 320.00 kWh (difference 30.00 kWh)

Such uncertainties are therefore to be expected, but only have a very minor effect on the statistical evaluation. They are practically irrelevant for larger time units (day, month, year).

However, it is important that the two times of writing are consistently mapped by the time values in the entry prefix. In the specific context, the application instance transfers the timestamp set by the scheduler for an action for writing. The data is therefore always saved with the correct time values, even if the execution of an action should exceptionally delay writing at the end of a time unit until the start of the next time unit.

4. Change frequency and relevance

Other aspects are the frequency and relevance of changes to status data. If, for example, data were saved per minute or even per second, even though the data has not changed at all or only slightly, this would not only unnecessarily increase the storage volume, but would also lead to many irrelevant entries in the statistical display.

Against this background, a 5-minute interval was defined as the smallest meaningful time unit in the specific context. Therefore, the 'm' entry in the 'intervalunit' parameter in the scheduling entry specifies not one, but 5 minutes: 00:00 to 04:59, 05:00 to 09:59 etc. Accordingly, the scheduler signals an action at the start and end of a 5-minute interval.



The 5-minute sections are also taken into account in the statistical analysis by means of an appropriately formulated query. The following query variants are intended to illustrate this.

The framework of the queries, which are executed here using the command 'selection' as an example and select data from the database table/entity 'state', looks as follows:

```
{"cmd": "selection", "value": "<SELECT ...> FROM state <where clause> <group by clause>;"}
```

A where clause restricts the selection to hour 15 of December 10, 2023 and the device 'DESKTOP' and is the same for all query variants:

```
WHERE t1=2023 AND t2=12 AND t3=10 AND t4=15 AND devname='DESKTOP'
```

The following are selected first: minute 't5', second 't6', a conversion from 't5' to the relevant 5-minute section 'm5', and the power in watts 'power':

```
SELECT t5, t6, (t5/5)*5 AS 'm5', ROUND(power, 2) AS 'power'

t5; t6; m5; power
0; 1; 0; 29.0
4; 53; 0; 29.3
5; 2; 5; 29.3
9; 54; 5; 29.0
10; 1; 10; 29.0
14; 53; 10; 30.2
...
```

The result shows that writing took place at the beginning and end of a 5-minute section, in this case at minute 0, second 1 and minute 4, second 53 etc. The start and end can vary by a few seconds, as the scheduler is called every 3 seconds and slight latencies can occur; however, the times always remain within the time period.

The following selection reduces the result to the 5-minute section 'm5' and the power 'power', but a grouping does not yet take place:

```
SELECT (t5/5)*5 AS 'm5', ROUND(power, 2) AS 'power'

m5; power
0; 29.0
0; 29.3
5; 29.3
5; 29.0
10; 29.0
10; 30.2
...
```

Finally, grouped by 'm5' with the average of 'power' is selected. In this form, the result is useful for displaying the average power per 5 minutes in statistics tables and charts.

```
SELECT (t5/5)*5 AS 'm5', ROUND(AVG(power), 2) AS 'power' ... GROUP BY 1

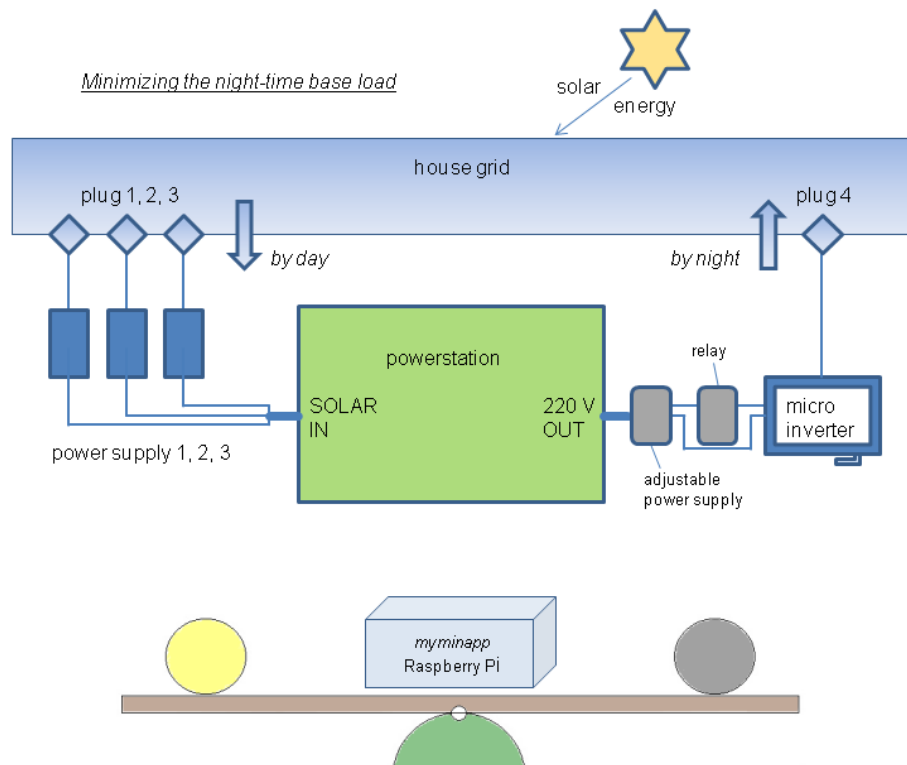
m5; power
0; 29.15
5; 29.15
10; 29.6
...
```

E. Example scenario

The following diagram shows an example of an automation scenario that was implemented using the myminapp application and some of the commands and devices supplied.

During the day, excess power from a standard balcony power station is supplied to a power station via up to three disused laptop power supplies (see 'pscharge' command).

At night, available power from the power station is fed into the house grid, thus minimizing the night-time base load (see 'psdischarge' command).



F. Program development

This chapter provides information on conceptual aspects of the development of 'myminapp'.

1. Objective

The 'myminapp' project is to be understood as a collection of examples in the form of a complete and functional Python application. It should make it possible to realize smaller DIY projects in the home sector with little effort.

The application should be easy to use from the terminal, be able to be integrated into other programs and support requests via HTTP(S) in JSON format.

Actions should be organized and executed in the form of commands. The commands should be encapsulated as classes and derived from a single superclass so that users can easily add their own commands for individual tasks. The term 'command' is used in the following.

The application should be further equipped in the form of classes, each of which represents a physical or virtual unit and can be used by the commands. The term 'device' is used for 'unit' in the following.

2. Programming language

Python was chosen as the programming language for the following reasons:

- Python is a powerful, widely used and very popular programming language that is easy to learn and use
- It can be used for the simplest to the most demanding tasks across all platforms
- The reservoir of generally available information, tutorials, examples and additional installable libraries/packages is practically inexhaustible
- As a source code/bytecode interpreter language, Python is particularly flexible when it comes to the dynamic integration of resources, for example
- From a security point of view, it should be noted that Python programs are generally available in open source code; however, this is not critical in the area of application addressed here
- The Python Software Foundation (PSF) license is a permissive open source license (see <https://docs.python.org/3/license.html>)

3. Guidelines

The following guidelines aim to achieve an application that is as simply structured, stable and flexible as possible, easy to use, readable and expandable.

3.1 General guidelines for the implementation of the application

- No dependence on a specific development environment or platform
- No dependence on specific frameworks
- No dependency on external Python packages for basic features
- As few dependencies as possible on external Python packages for special features
- Organization of the application in a simple, transparent structure
- Application definition/configuration in just one file with Python data types
- Messages in German and English in just one file with Python data types
- Organization of the program code in classes, each in an exclusive file
- Basically object-oriented code, but which may also have procedural parts
- Module, class, method and variable names as well as code comments in English

- Docstring documentation of the modules, classes and methods in English
- Simple command base class for deriving your own commands
- Dynamic instantiation of the command classes and dynamic call of their methods
- Internal data format for command inputs and outputs: Python type 'dictionary'
- External data format for command inputs and outputs: JSON
- Communication with the data storage according to the pattern of an interface
- Implementation of the data storage with SQLite (included in Python)
- Test classes according to a standardized pattern for each command and each utility
- Documentation of the application in a manual in PDF format in German and English

3.2 Guidelines aimed at safety aspects

- The application is intended for home use and for use in the local network. Authentication, roles and rights management are therefore omitted in favor of ease of use
- Communication via HTTPS should be optionally supported in order to comply with current browser standards when using the web frontend
- Manipulation of the application via the communication channels provided or the commands supplied should be virtually impossible
- The application definition/configuration should only be possible by directly editing the relevant file

3.3 Guidelines for coding

- The code should essentially be object-oriented and comply with Python standards
- The quality of the code should be good, but very high requirements cannot and should not be met here
- The code should be practical in terms of the objective of the application; for example, it should not be too finely granulated in order to keep the application structure clear
- Each class should be coded in a separate file (a separate module) in order to keep the structure transparent and support good maintainability
- Within the classes, a distinction should be made between public and private methods ('def a_public_method()' and 'def __a_private_method()')
- Methods, including comment lines, should typically not exceed 50 lines and in exceptional cases a maximum of 200 lines

3.4 Guidelines for exception handling

- In order to keep the application as simple as possible in line with the objective, separate exception derivations should be omitted
- If a final action is to take place within a method in every case, the relevant code must be enclosed in a try-final block (for example, to safely close a database connection before the method is exited)
- Within a method of utility and equipment classes (Utility, Device), try-except blocks should be avoided as far as possible, unless an exception must be handled directly (e.g. for retries without leaving the method)
- For plausibility checks and in similar cases, raise exception can be used
- Commands should transfer an exception that occurs during an action with an error code, error text and trace text to the result. This enables the execution of a series of actions within a command
- Exceptions should only be handled at the highest level of the application if possible
- The exceptions handled at the highest level should always be logged with error text and trace text



- The application server that processes HTTP(S) GET requests should return HTTP code 400 (file not found) if the basic check of a request fails, and HTTP code 500 (internal server error) with additional error text if a command generally fails
- The web frontend should receive the error text in the result of the response for exceptions within the command execution, not the trace text

4. Basic equipment

The application should be delivered in a basic configuration that contains not only the basic features but also commands and devices that have already been implemented. These should be both functional and able to serve as samples.

Dependencies beyond the Python standard library should be limited to devices as far as possible.

5. Development environment

An IDE well suited for Python such as Microsoft Visual Studio Code should be used, for example under Windows with the Windows Subsystem for Linux (WSL). Version control should be carried out with GIT.

In line with the objective of the application, no framework is used.

6. Distribution, installation, publication

For distribution, the application is packed into a release file 'myminapp-<version number>'.zip.

The release file must be unpacked for installation. The application can then be used directly with Python 3.10 or higher, as the basic features do not require any external dependencies. It is also important that several installations can exist independently in parallel by simply unpacking the release file under additional directories or copying an existing installation.

This kind of installation is the easiest way to access the source code, and copies are easy to handle. Alternatively, myminapp should also be installable using the Package Installer for Python (pip).

The application is to be published via GitHub (and Python Package Index (PyPI)).

7. Development process

Whether the application will be further developed by the author or a team is left open at this point.



G. Credits

Many thanks to...

- the many experts and committed people who share their knowledge and experience on the topics of renewable energies and smart homes
- the professional and amateur software developers who share their knowledge and experience via tutorials, videos, blogs and other means, here specifically on programming with Python
- the Python Software Foundation specifically
- the open source community in general
- Microsoft and the involved developers for Visual Studio Code and GitHub
- DeepL for the support with the translation into English