
FlexMeasures Documentation

Release 0.14.1

Seita B.V.

Jun 26, 2023

CONTENTS

1	A quick glance at usage	3
2	Use cases	5
3	A possible road to start using FlexMeasures in your operation	7
4	Where to start reading?	9
5	Developer support	11
	Python Module Index	309
	HTTP Routing Table	311
	Index	313

When we use a lot of renewable energy, flexibility is becoming crucial and valuable, e.g. for demand response. FlexMeasures is the intelligent & developer-friendly EMS to support real-time energy flexibility apps, rapidly and scalable.

The problem it helps to solve is:

What are the best times to run flexible assets, like batteries or heat pumps?

In a nutshell, FlexMeasures turns data into optimized schedules for flexible assets. Why? Planning ahead allows flexible assets to serve the whole system with their flexibility, e.g. by shifting energy consumption to other times. For the asset owners, this creates CO₂ savings but also monetary value (e.g. through self-consumption, dynamic tariffs and grid incentives).



However, developing apps & services around energy flexibility is expensive work. FlexMeasures is designed to be developer-friendly, which helps you to go to market quickly, while keeping the costs of software development at bay. FlexMeasures supports:

- Real-time data integration & intelligence
- Model data well — units, time resolution & uncertainty (of forecasts)
- Faster app-building (API/UI/CLI, plugin & multi-tenancy support)

More on this in [Developer support](#). FlexMeasures proudly is an incubation project at the [Linux Energy Foundation](#). Also, read more on where FlexMeasures is useful in [Use cases](#).

A QUICK GLANCE AT USAGE

A tiny, but complete example: Let's install FlexMeasures from scratch. Then, using only the terminal (FlexMeasures of course also has APIs for all of this), load hourly prices and optimize a 12h-schedule for a battery that is half full at the beginning. Finally, look at our new schedule.

```
$ pip install flexmeasures # FlexMeasures can also be run via Docker
$ docker pull postgres; docker run --name pg-docker -e POSTGRES_PASSWORD=docker -e
→POSTGRES_DB=flexmeasures-db -d -p 5433:5432 postgres:latest
$ export SQLALCHEMY_DATABASE_URI="postgresql://postgres:docker@127.0.0.1:5433/
→flexmeasures-db" && export SECRET_KEY=notsecret
$ flexmeasures db upgrade # create tables
$ flexmeasures add toy-account --kind battery # setup account incl. a user, battery (ID
→1) and market (ID 2)
$ flexmeasures add beliefs --sensor-id 2 --source toy-user prices-tomorrow.csv --
→timezone utc # load prices, also possible per API
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --start ${TOMORROW}T07:00+01:00 --duration PT12H \
  --soc-at-start 50% --roundtrip-efficiency 90% # this is also possible per API
$ flexmeasures show beliefs --sensor-id 1 --start ${TOMORROW}T07:00:00+01:00 --duration
→PT12H # also visible per UI, of course
```

We discuss this in more depth at *Toy example: Scheduling a battery, from scratch*.

USE CASES

Here are a few relevant areas in which FlexMeasures can help you:

- E-mobility (smart EV (Electric Vehicle) charging, V2G (Vehicle to Grid), V2H (Vehicle to Home))
- Heating (heat pump control)
- Industry (best running times for processes with buffering capacity)

You decide what to optimize for — prices, CO₂, peaks.

It becomes even more interesting to use FlexMeasures in *integrated scenarios* with increased complexity. For example, in modern domestic/office settings that combine solar panels, electric heating and EV charging, in industry settings that optimize for self-consumption of local solar panels, or when consumers can engage with multiple markets simultaneously.

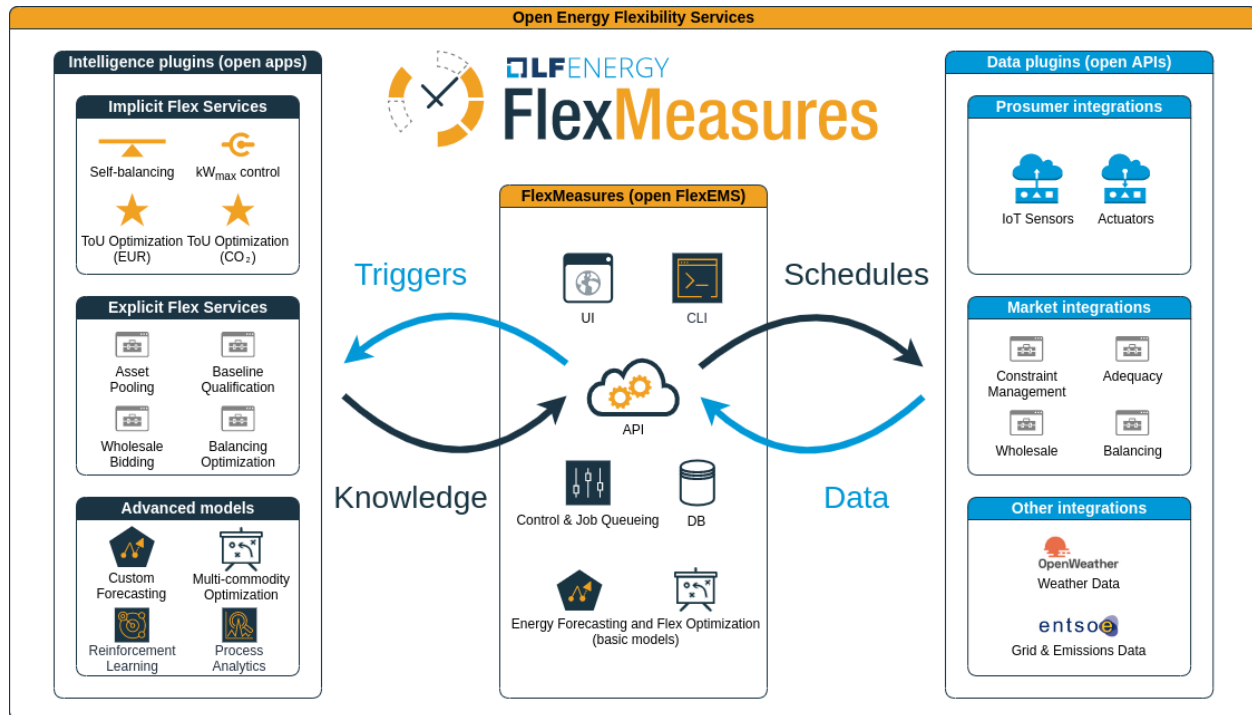
In these cases, our goal is that FlexMeasures helps you to achieve “*value stacking*”, which is often required to achieve a positive business case. Multiple sources of value can combine with multiple types of assets.

As possible users, we see energy service companies (ESCOs) who want to build real-time apps & services around energy flexibility for their customers, or medium/large industrials who are looking for support in their internal digital tooling.

However, even small companies and hobby projects might find FlexMeasures useful! We are constantly improving the ease of use.

FlexMeasures can be used as your EMS, but it can also integrate with existing systems as a smart backend, or as an add-on to deal with energy flexibility specifically.

The image below shows how FlexMeasures, with the help of plugins fitted for a given use case, turns data into optimized schedules:



A POSSIBLE ROAD TO START USING FLEXMEASURES IN YOUR OPERATION

We make FlexMeasures, so that software developers are as productive with energy optimization as possible. Because we are developers ourselves, we know that it takes a couple smaller steps to engage with new technology.

Your journey, from dipping your toes in the water towards being a happy FlexMeasures power user, could look like this:

1. Quickstart — Find an optimized schedule for your flexible asset, like a battery, with standard FlexMeasures tooling. This is basically what the from-scratch tutorial above does. All you need are 10 minutes and a CSV file with prices to optimise against.
2. Automate — get the prices from an open API, for instance [ENTSO-E](#) (using a plugin like [flexmeasures-entsoe](#)), and run the scheduler regularly in a cron job.
3. Integrate — Load the schedules via FlexMeasures' API, so you can directly control your assets and/or show them within your own frontend.
4. Customize — Load other data (e.g. your solar production or weather forecasts via [flexmeasures-openweathermap](#)). Adapt the algorithms, e.g. do your own forecasting or tweak the standard scheduling algorithm so it optimizes what you care about. Or write a plugin for accessing a new kind of market. The opportunities are endless!

WHERE TO START READING?

You (the reader) might be a user connecting with a FlexMeasures server or working on hosting FlexMeasures. Maybe you are planning to develop a plugin or even core functionality. In [Getting started](#), we have some helpful tips how to dive into this documentation!

DEVELOPER SUPPORT

FlexMeasures is designed to help with three basic needs of developers in the energy flexibility domain:

5.1 I need help with integrating real-time data and continuously computing new data

FlexMeasures is designed to make decisions based on data in an automated way. Data pipelining and dedicated machine learning tooling is crucial.

- API/CLI functionality to read in time series data
- Extensions for integrating 3rd party data, e.g. from [ENTSO-E](#) or [OpenWeatherMap](#)
- Forecasting for the upcoming hours
- Schedule optimization for flexible assets

5.2 It's hard to correctly model data with different sources, resolutions, horizons and even uncertainties

Much developer time is spent correcting data and treating it correctly, so that you know you are computing on the right knowledge.

FlexMeasures is built on the [timely-beliefs framework](#), so we model this real-world aspect accurately:

- Expected data properties are explicit (e.g. unit, time resolution)
- Incoming data is converted to fitting unit and time resolution automatically
- FlexMeasures also stores who thought that something happened (or that it will happen), and when they thought so
- Uncertainty can be modelled (useful for forecasting)

5.3 I want to build new features quickly, not spend days solving basic problems

Building customer-facing apps & services is where developers make impact. We make their work easy.

- FlexMeasures has well-documented API endpoints and CLI commands to interact with its model and data
- You can extend it easily with your own logic by writing plugins
- A backend UI shows you your assets in maps and your data in plots. There is also support for plots to be available per API, for integration in your own frontend
- Multi-tenancy — model multiple accounts on one server. Data is only seen/editable by authorized users in the right account

For more on FlexMeasures services, read *In-built smart functionality*. Or head right over to *Getting started*.

Using FlexMeasures benefits operators as well as asset owners, by allowing for automation, insight, autonomy and profit sharing. For more on benefits, consult *Benefits*.

FlexMeasures is compliant with the [Universal Smart Energy Framework \(USEF\)](#). Therefore, this documentation uses USEF terminology, e.g. for role definitions. In this context, the intended users of FlexMeasures are a Supplier (energy company) and its Prosumers (asset owners who have energy contracts with that Supplier). The platform operator of FlexMeasures can be an Aggregator.

5.3.1 Getting started

FlexMeasures is useful from different perspectives. The documentation is quite vast, so we give you some pointers here what to read first, based on your perspective.

Using FlexMeasures

You are connecting to a running FlexMeasures server, e.g. for sending data, getting schedules or administrate users and assets.

First, you'll need an account from the party running the server. Also, you probably want to:

- Look at the UI, e.g. pages for *Dashboard* and *Administration*.
- Read the *API Introduction*.
- Learn how to interact with the API in *Posting data*.

Hosting FlexMeasures

You want to run your own FlexMeasures instance, to offer services or for trying it out. You'll want to:

- Have a first playful scheduling session, following *Toy example: Scheduling a battery, from scratch*.
- Get real with the tutorial on *Installation & First steps*.
- Discover the power of *CLI Commands*.
- Understand how to *How to deploy FlexMeasures*.

Plugin developers

You want to extend the functionality of FlexMeasures, e.g. a custom integration or a custom algorithm:

- Read the docs on *Writing Plugins*.
- See how some existing plugins are made [flexmeasures-entsoe](#) or [flexmeasures-openweathermap](#)
- Of course, some of the developers resources (see below) might be helpful to you, as well.

Warning: Please read [note_on_datamodel_transition](#).

Core developers

You want to help develop FlexMeasures, e.g. to fix a bug. We provide a getting-started guide to becoming a developer at *Developing for FlexMeasures*.

5.3.2 Get in touch

We want you to succeed in using, hosting or extending FlexMeasures. For all your questions and ideas, you can join the FlexMeasures community in the following ways:

- View the code and/or create a ticket on [GitHub](#)
- Join the [#flexmeasures](#) Slack channel over at <https://lfenergy.slack.com>
- Write to us at flexmeasures@lists.lfenergy.org (you can join this mailing list [here](#))
- Follow [@flexmeasures](#) on Twitter

We'd love to hear from you!

5.3.3 FlexMeasures Changelog

v0.14.1 | June 26, 2023

Bugfixes

- Relax constraint validation of *StorageScheduler* to accommodate violations caused by floating point precision [see [PR #731](#)]
- Avoid saving any NaN (not a number) values to the database, when calling `flexmeasures add report` [see [PR #735](#)]
- Fix browser console error when loading asset or sensor page with only a single data point [see [PR #732](#)]
- Fix showing multiple sensors with bare 3-letter currency code as their units (e.g. EUR) in one chart [see [PR #738](#)]
- Fix defaults for the `--start-offset` and `--end-offset` options to `flexmeasures add report`, which weren't being interpreted in the local timezone of the reporting sensor [see [PR #744](#)]
- Relax constraint for overlaying plot traces for sensors with various resolutions, making it possible to show e.g. two price sensors in one chart, where one of them records hourly prices and the other records quarter-hourly prices [see [PR #743](#)]

- Resolve bug where different page loads would potentially influence the time axis of each other's charts, by avoiding mutation of shared field definitions [see [PR #746](#)]

v0.14.0 | June 15, 2023

Note: Read more on these features on [the FlexMeasures blog](#).

New features

- Allow setting a storage efficiency using the new `storage-efficiency` field when calling `/sensors/<id>/schedules/trigger` (POST) through the API (within the `flex-model` field), or when calling `flexmeasures add schedule for-storage` through the CLI [see [PR #679](#)]
- Allow setting multiple SoC (state of charge) maxima and minima constraints for the *StorageScheduler*, using the new `soc-minima` and `soc-maxima` fields when calling `/sensors/<id>/schedules/trigger` (POST) through the API (within the `flex-model` field) [see [PR #680](#)]
- New CLI command `flexmeasures add report` to calculate a custom report from sensor data and save the results to the database, with the option to export them to a CSV or Excel file [see [PR #659](#)]
- New CLI commands `flexmeasures show reporters` and `flexmeasures show schedulers` to list available reporters and schedulers, respectively, including any defined in registered plugins [see [PR #686](#) and [PR #708](#)]
- Allow creating public assets through the CLI, which are available to all users [see [PR #727](#)]

Bugfixes

- Fix charts not always loading over https in secured scenarios [see [PR #716](#)]

Infrastructure / Support

- Introduction of the classes *Reporter*, *PandasReporter* and *AggregatorReporter* to help customize your own reporter functions (experimental) [see [PR #641](#) and [PR #712](#)]
- The setting `FLEXMEASURES_PLUGINS` can be set as environment variable now (as a comma-separated list) [see [PR #660](#)]
- Packaging was modernized to stop calling `setup.py` directly [see [PR #671](#)]
- Remove API versions 1.0, 1.1, 1.2, 1.3 and 2.0, while making sure that sunset endpoints keep returning HTTP status 410 (Gone) responses [see [PR #667](#) and [PR #717](#)]
- Support Pandas 2 [see [PR #673](#)]
- Add code documentation from package structure and docstrings to official docs [see [PR #698](#)]

Warning: The setting `FLEXMEASURES_PLUGIN_PATHS` has been deprecated since v0.7. It has now been sunset. Please replace it with `FLEXMEASURES_PLUGINS`.

v0.13.3 | June 10, 2023

Bugfixes

- Fix forwarding arguments in deprecated util function [see [PR #719](#)]

v0.13.2 | June 9, 2023

Bugfixes

- Fix failing to save results of scheduling and reporting on subsequent calls for the same time period [see [PR #709](#)]

v0.13.1 | May 12, 2023

Bugfixes

- *@deprecated* not returning the output of the decorated function [see [PR #678](#)]

v0.13.0 | May 1, 2023

Warning: Sunset notice for API versions 1.0, 1.1, 1.2, 1.3 and 2.0: after upgrading to `flexmeasures==0.13`, users of these API versions may receive HTTP status 410 (Gone) responses. See the [documentation for deprecation and sunset](#). The relevant endpoints have been deprecated since `flexmeasures==0.12`.

Warning: The API endpoint (`[POST] /sensors/{id}/schedules/trigger`) to make new schedules sunsets the deprecated (since v0.12) storage flexibility parameters (they move to the `flex-model` parameter group), as well as the parameters describing other sensors (they move to `flex-context`).

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

Note: Read more on these features on [the FlexMeasures blog](#).

New features

- Keyboard control over replay [see [PR #562](#)]
- Overlay charts (e.g. power profiles) on the asset page using the `sensors_to_show` attribute, and distinguish plots by source (different trace), sensor (different color) and source type (different stroke dash) [see [PR #534](#)]
- The `FLEXMEASURES_MAX_PLANNING_HORIZON` config setting can also be set as an integer number of planning steps rather than just as a fixed duration, which makes it possible to schedule further ahead in coarser time steps [see [PR #583](#)]
- Different text styles for CLI output for errors, warnings or success messages. [see [PR #609](#)]
- Added API endpoints and webpages `/accounts` and `/accounts/<id>` to list accounts and show an overview of the assets, users and account roles of an account [see [PR #605](#)]
- Avoid redundantly recomputing jobs that are triggered without a relevant state change. `FLEXMEASURES_JOB_CACHE_TTL` config setting defines the time in which the jobs with the same arguments are not being recomputed. [see [PR #616](#)]

Bugfixes

- Fix copy button on tutorials and other documentation, so that only commands are copied and no output or comments [see [PR #636](#)]
- `GET /api/v3_0/assets/public` should ask for token authentication and not forward to login page [see [PR #649](#)]

Infrastructure / Support

- Support blackout tests for sunset API versions [see [PR #651](#)]
- Sunset API versions 1.0, 1.1, 1.2, 1.3 and 2.0 [see [PR #650](#)]
- Sunset several API fields for `/sensors/<id>/schedules/trigger` (POST) that have moved into the `flex-model` or `flex-context` fields [see [PR #580](#)]
- Fix broken `make show-data-model` command [see [PR #638](#)]
- Bash script for a clean database to run toy-tutorial by using `make clean-db db_name=database_name` command [see [PR #640](#)]

v0.12.3 | February 28, 2023

Bugfixes

- Fix premature deserialization of `flex-context` field for `/sensors/<id>/schedules/trigger` (POST) [see [PR #593](#)]

v0.12.2 | February 4, 2023

Bugfixes

- Fix CLI command `flexmeasures schedule for-storage` without `--as-job` flag [see [PR #589](#)]

v0.12.1 | January 12, 2023

Bugfixes

- Fix validation of (deprecated) API parameter `roundtrip-efficiency` [see [PR #582](#)]

v0.12.0 | January 4, 2023

Warning: After upgrading to `flexmeasures==0.12`, users of API versions 1.0, 1.1, 1.2, 1.3 and 2.0 will receive "Deprecation" and "Sunset" response headers, and warnings are logged for FlexMeasures hosts whenever users call API endpoints in these deprecated API versions. The relevant endpoints are planned to become unresponsive in `flexmeasures==0.13`.

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

Note: Read more on these features on [the FlexMeasures blog](#).

New features

- Hit the replay button to visually replay what happened, available on the sensor and asset pages [see [PR #463](#) and [PR #560](#)]
- Ability to provide your own custom scheduling function [see [PR #505](#)]
- Visually distinguish forecasts/schedules (dashed lines) from measurements (solid lines), and expand the tooltip with timing info regarding the forecast/schedule horizon or measurement lag [see [PR #503](#)]
- The asset page also allows to show sensor data from other assets that belong to the same account [see [PR #500](#)]
- The CLI command `flexmeasures monitor latest-login` supports to check if (bot) users who are expected to contact FlexMeasures regularly (e.g. to send data) fail to do so [see [PR #541](#)]
- The CLI command `flexmeasures show beliefs` supports showing beliefs data in a custom resolution and/or timezone, and also saving the shown beliefs data to a CSV file [see [PR #519](#)]
- Improved import of time series data from CSV file: 1) drop duplicate records with warning, 2) allow configuring which column contains explicit recording times for each data point (use case: import forecasts) [see [PR #501](#)], 3) localize timezone naive data, 4) support reading in datetime and timedelta values, 5) remove rows with NaN values, and 6) filter by values in specific columns [see [PR #521](#)]
- Filter data by source in the API endpoint `/sensors/data` (GET) [see [PR #543](#)]

- Allow posting null values to `/sensors/data` (POST) to correctly space time series that include missing values (the missing values are not stored) [see [PR #549](#)]
- Allow setting a custom planning horizon when calling `/sensors/<id>/schedules/trigger` (POST), using the new `duration` field [see [PR #568](#)]
- New resampling functionality for instantaneous sensor data: 1) `flexmeasures show beliefs` can now handle showing (and saving) instantaneous sensor data and non-instantaneous sensor data together, and 2) the API endpoint `/sensors/data` (GET) now allows fetching instantaneous sensor data in a custom frequency, by using the “resolution” field [see [PR #542](#)]

Bugfixes

- The CLI command `flexmeasures show beliefs` now supports plotting time series data that includes NaN values, and provides better support for plotting multiple sensors that do not share the same unit [see [PR #516](#) and [PR #539](#)]
- Fixed JSON wrapping of return message for `/sensors/data` (GET) [see [PR #543](#)]
- Consistent CLI/UI support for asset lat/lng positions up to 7 decimal places (previously the UI rounded to 4 decimal places, whereas the CLI allowed more than 4) [see [PR #522](#)]
- Stop trimming the planning window in response to price availability, which is a problem when SoC targets occur outside of the available price window, by making a simplistic assumption about future prices [see [PR #538](#)]
- Faster loading of initial charts and calendar date selection [see [PR #533](#)]

Infrastructure / Support

- Reduce size of Docker image (from 2GB to 1.4GB) [see [PR #512](#)]
- Allow extra requirements to be freshly installed when running `docker-compose up` [see [PR #528](#)]
- Remove bokeh dependency and obsolete UI views [see [PR #476](#)]
- Fix `flexmeasures db-ops dump` and `flexmeasures db-ops restore` not working in docker containers [see [PR #530](#)] and incorrectly reporting a success when `pg_dump` and `pg_restore` are not installed [see [PR #526](#)]
- Plugins can save BeliefsSeries, too, instead of just BeliefsDataFrames [see [PR #523](#)]
- Improve documentation and code w.r.t. storage flexibility modelling — prepare for handling other schedulers & merge battery and car charging schedulers [see [PR #511](#), [PR #537](#) and [PR #566](#)]
- Revised strategy for removing unchanged beliefs when saving data: retain the oldest measurement (ex-post belief), too [see [PR #518](#)]
- Scheduling test for maximizing self-consumption, and improved time series db queries for fixed tariffs (and other long-term constants) [see [PR #532](#)]
- Clean up table formatting for `flexmeasures show` CLI commands [see [PR #540](#)]
- Add "Deprecation" and "Sunset" response headers for API users of deprecated API versions, and log warnings for FlexMeasures hosts when users still use them [see [PR #554](#) and [PR #565](#)]
- Explain how to avoid potential SMTPRecipientsRefused errors when using FlexMeasures in combination with a mail server [see [PR #558](#)]
- Set a limit to the allowed planning window for API users, using the `FLEXMEASURES_MAX_PLANNING_HORIZON` setting [see [PR #568](#)]

Warning: The API endpoint (`[POST] /sensors/{id}/schedules/trigger`) to make new schedules will (in v0.13) sunset the storage flexibility parameters (they move to the `flex-model` parameter group), as well as the parameters describing other sensors (they move to `flex-context`).

Warning: The CLI command `flexmeasures monitor tasks` has been deprecated (it's being renamed to `flexmeasures monitor last-run`). The old name will be sunset in version 0.13.

Warning: The CLI command `flexmeasures add schedule` has been renamed to `flexmeasures add schedule for-storage`. The old name will be sunset in version 0.13.

v0.11.3 | November 2, 2022

Bugfixes

- Fix scheduling with imperfect efficiencies, which resulted in exceeding the device's lower SoC limit. [see [PR #520](#)]
- Fix scheduler for Charge Points when taking into account inflexible devices [see [PR #517](#)]
- Prevent rounding asset lat/long positions to 4 decimal places when editing an asset in the UI [see [PR #522](#)]

v0.11.2 | September 6, 2022

Bugfixes

- Fix regression for sensors recording non-instantaneous values [see [PR #498](#)]
- Fix broken auth check for creating assets with CLI [see [PR #497](#)]

v0.11.1 | September 5, 2022

Bugfixes

- Do not fail asset page if none of the sensors has any data [see [PR #493](#)]
- Do not fail asset page if one of the shown sensors records instantaneous values [see [PR #491](#)]

v0.11.0 | August 28, 2022

New features

- The asset page now shows the most relevant sensor data for the asset [see [PR #449](#)]
- Individual sensor charts show available annotations [see [PR #428](#)]
- New API options to further customize the optimization context for scheduling, including the ability to use different prices for consumption and production (feed-in) [see [PR #451](#)]

- Admins can group assets by account on dashboard & assets page [see [PR #461](#)]
- Collapsible side-panel (hover/swipe) used for date selection on sensor charts, and various styling improvements [see [PR #447](#) and [PR #448](#)]
- Add CLI command `flexmeasures jobs show-queues` [see [PR #455](#)]
- Switched from 12-hour AM/PM to 24-hour clock notation for time series chart axis labels [see [PR #446](#)]
- Get data in a given resolution [see [PR #458](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Do not fail asset page if entity addresses cannot be built [see [PR #457](#)]
- Asynchronous reloading of a chart's dataset relies on that chart already having been embedded [see [PR #472](#)]
- Time scale axes in sensor data charts now match the requested date range, rather than stopping at the edge of the available data [see [PR #449](#)]
- The docker-based tutorial now works with UI on all platforms (port 5000 did not expose on MacOS) [see [PR #465](#)]
- Fix interpretation of scheduling results in toy tutorial [see [PR #466](#) and [PR #475](#)]
- Avoid formatting `datetime.timedelta` durations as nominal ISO durations [see [PR #459](#)]
- Account admins cannot add assets to other accounts any more; and they are shown a button for asset creation in UI [see [PR #488](#)]

Infrastructure / Support

- Docker compose stack now with Redis worker queue [see [PR #455](#)]
- Allow access tokens to be passed as env vars as well [see [PR #443](#)]
- Queue workers can get initialised without a custom name and name collisions are handled [see [PR #455](#)]
- New API endpoint to get public assets [see [PR #461](#)]
- Allow editing an asset's JSON attributes through the UI [see [PR #474](#)]
- Allow a custom message when monitoring latest run of tasks [see [PR #489](#)]

v0.10.1 | August 12, 2022

Bugfixes

- Fix some UI styling regressions in e.g. color contrast and hover effects [see [PR #441](#)]

v0.10.0 | May 8, 2022

New features

- New design for FlexMeasures' UI back office [see [PR #425](#)]
- Improve legibility of chart axes [see [PR #413](#)]
- API provides health readiness check at `/api/v3_0/health/ready` [see [PR #416](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Fix small problems in support for the admin-reader role & role-based authorization [see [PR #422](#)]

Infrastructure / Support

- Dockerfile to run FlexMeasures in container; also docker-compose file [see [PR #416](#)]
- Unit conversion prefers shorter units in general [see [PR #415](#)]
- Shorter CI builds in Github Actions by caching Python environment [see [PR #361](#)]
- Allow to filter data by source using a tuple instead of a list [see [PR #421](#)]

v0.9.4 | April 28, 2022

Bugfixes

- Support checking validity of custom units (i.e. non-SI, non-currency units) [see [PR #424](#)]

v0.9.3 | April 15, 2022

Bugfixes

- Let registered plugins use CLI authorization [see [PR #411](#)]

v0.9.2 | April 10, 2022

Bugfixes

- Prefer unit conversions to short stock units [see [PR #412](#)]
- Fix filter for selecting one deterministic belief per event, which was duplicating index levels [see [PR #414](#)]

v0.9.1 | March 31, 2022

Bugfixes

- Fix auth bug not masking locations of inaccessible assets on map [see [PR #409](#)]
- Fix CLI auth check [see [PR #407](#)]
- Fix resampling of sensor data for scheduling [see [PR #406](#)]

v0.9.0 | March 25, 2022

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

New features

- Three new CLI commands for cleaning up your database: delete 1) unchanged beliefs, 2) NaN values or 3) a sensor and all of its time series data [see [PR #328](#)]
- Add CLI option to pass a data unit when reading in time series data from CSV, so data can automatically be converted to the sensor unit [see [PR #341](#)]
- Add CLI option to specify custom strings that should be interpreted as NaN values when reading in time series data from CSV [see [PR #357](#)]
- Add CLI commands `flexmeasures add sensor`, `flexmeasures add asset-type`, `flexmeasures add beliefs` (which were experimental features before) [see [PR #337](#)]
- Add CLI commands for showing organisational structure [see [PR #339](#)]
- Add CLI command for showing time series data [see [PR #379](#)]
- Add CLI command for attaching annotations to assets: `flexmeasures add holidays` adds public holidays [see [PR #343](#)]
- Add CLI command for resampling existing sensor data to new resolution [see [PR #360](#)]
- Add CLI command to delete an asset, with its sensors and data. [see [PR #395](#)]
- Add CLI command to edit/add an attribute on an asset or sensor. [see [PR #380](#)]
- Add CLI command to add a toy account for tutorials and trying things [see [PR #368](#)]
- Add CLI command to create a charging schedule [see [PR #372](#)]
- Support for percent (%) and permille (‰) sensor units [see [PR #359](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

Infrastructure / Support

- Plugins can import common FlexMeasures classes (like Asset and Sensor) from a central place, using `from flexmeasures import Asset, Sensor` [see [PR #354](#)]
- Adapt CLI command for entering some initial structure (`flexmeasures add structure`) to new datamodel [see [PR #349](#)]
- Align documentation requirements with pip-tools [see [PR #384](#)]
- Beginning API v3.0 - more REST-like, supporting assets, users and sensor data [see [PR #390](#) and [PR #392](#)]

v0.8.0 | January 24, 2022

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

Warning: In case you use FlexMeasures for simulations using `FLEXMEASURES_MODE = "play"`, allowing to overwrite data is now set separately using `FLEXMEASURES_ALLOW_DATA_OVERWRITE`. Add `FLEXMEASURES_ALLOW_DATA_OVERWRITE = True` to your config settings to keep the old behaviour.

Note: v0.8.0 is doing much of the work we need to do to move to the new data model (see `note_on_datamodel_transition`). We hope to keep the migration steps for users very limited. One thing you'll notice is that we are copying over existing data to the new model (which will be kept in sync) with the `db upgrade` command (see warning above), which can take a few minutes.

New features

- Bar charts of sensor data for individual sensors, that can be navigated using a calendar [see [PR #99](#) and [PR #290](#)]
- Charts with sensor data can be requested in one of the supported [[vega-lite themes](#)] (incl. a dark theme) [see [PR #221](#)]
- Mobile friendly (responsive) charts of sensor data, and such charts can be requested with a custom width and height [see [PR #313](#)]
- Schedulers take into account round-trip efficiency if set [see [PR #291](#)]
- Schedulers take into account min/max state of charge if set [see [PR #325](#)]
- Fallback policies for charging schedules of batteries and Charge Points, in cases where the solver is presented with an infeasible problem [see [PR #267](#) and [PR #270](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Deprecations

- The Portfolio and Analytics views are deprecated [see [PR #321](#)]

Bugfixes

- Fix recording time of schedules triggered by UDI events [see [PR #300](#)]
- Set bar width of bar charts based on sensor resolution [see [PR #310](#)]
- Fix bug in sensor data charts where data from multiple sources would be stacked, which incorrectly suggested that the data should be summed, whereas the data represents alternative beliefs [see [PR #228](#)]

Infrastructure / Support

- Account-based authorization, incl. new decorators for endpoints [see [PR #210](#)]
- Central authorization policy which lets database models codify who can do what (permission-based) and relieve API endpoints from this [see [PR #234](#)]
- Improve data specification for forecasting models using timely-beliefs data [see [PR #154](#)]
- Properly attribute Mapbox and OpenStreetMap [see [PR #292](#)]
- Allow plugins to register their custom config settings, so that FlexMeasures can check whether they are set up correctly [see [PR #230](#) and [PR #237](#)]
- Add sensor method to obtain just its latest state (excl. forecasts) [see [PR #235](#)]
- Migrate attributes of assets, markets and weather sensors to our new sensor model [see [PR #254](#) and [project 9](#)]
- Migrate all time series data to our new sensor data model based on the [timely beliefs](#) lib [see [PR #286](#) and [project 9](#)]
- Support the new asset model (which describes the organisational structure, rather than sensors and data) in UI and API. Until the transition to our new data model is completed, the new API for assets is at `/api/dev/generic_assets`. [see [PR #251](#) and [PR #290](#)]
- Internal search methods return most recent beliefs by default, also for charts, which can make them load a lot faster [see [PR #307](#) and [PR #312](#)]
- Support unit conversion for posting sensor data [see [PR #283](#) and [PR #293](#)]
- Improve the core device scheduler to support dealing with asymmetric efficiency losses of individual devices, and with asymmetric up and down prices for deviating from previous commitments (such as a different feed-in tariff) [see [PR #291](#)]
- Stop automatically triggering forecasting jobs when API calls save nothing new to the database, thereby saving redundant computation [see [PR #303](#)]

v0.7.1 | November 8, 2021

Bugfixes

- Fix device messages, which were mixing up older and more recent schedules [see [PR #231](#)]

v0.7.0 | October 26, 2021

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

Warning: The config setting `FLEXMEASURES_PLUGIN_PATHS` has been renamed to `FLEXMEASURES_PLUGINS`. The old name still works but is deprecated.

New features

- Set a logo for the top left corner with the new `FLEXMEASURES_MENU_LOGO_PATH` setting [see [PR #184](#)]
- Add an extra style-sheet which applies to all pages with the new `FLEXMEASURES_EXTRA_CSS_PATH` setting [see [PR #185](#)]
- Data sources can be further distinguished by what model (and version) they ran [see [PR #215](#)]
- Enable plugins to automate tests with app context [see [PR #220](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Fix users resetting their own password [see [PR #195](#)]
- Fix scheduling for heterogeneous settings, for instance, involving sensors with different time zones and/or resolutions [see [PR #207](#)]
- Fix `sensors/<id>/chart` view [see [PR #223](#)]

Infrastructure / Support

- FlexMeasures plugins can be Python packages now. We provide [a cookie-cutter template](#) for this approach. [see [PR #182](#)]
- Set default timezone for new users using the `FLEXMEASURES_TIMEZONE` config setting [see [PR #190](#)]
- To avoid databases from filling up with irrelevant information, only beliefs data representing *changed beliefs are saved*, and *unchanged beliefs are dropped* [see [PR #194](#)]
- Monitored CLI tasks can get better names for identification [see [PR #193](#)]
- Less custom logfile location, document logging for devs [see [PR #196](#)]

- Keep forecasting and scheduling jobs in the queues for only up to one day [see [PR #198](#)]

v0.6.1 | October 23, 2021

New features

Bugfixes

- Fix (dev) CLI command for adding a `GenericAssetType` [see [PR #173](#)]
- Fix (dev) CLI command for adding a `Sensor` [see [PR #176](#)]
- Fix missing conversion of data source names and ids to `DataSource` objects [see [PR #178](#)]
- Fix `GetDeviceMessage` to ensure chronological ordering of values [see [PR #216](#)]

Infrastructure / Support

v0.6.0 | September 3, 2021

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`). In case you are using experimental developer features and have previously set up sensors, be sure to check out the upgrade instructions in [PR #157](#). Furthermore, if you want to create custom user/account relationships while upgrading (otherwise the upgrade script creates accounts based on email domains), check out the upgrade instructions in [PR #159](#). If you want to use both of these custom upgrade features, do the upgrade in two steps. First, as described in [PR 157](#) and upgrading up to revision `b6d49ed7cceb`, then as described in [PR 159](#) for the rest.

Warning: The config setting `FLEXMEASURES_LISTED_VIEWS` has been renamed to `FLEXMEASURES_MENU_LISTED_VIEWS`.

Warning: Plugins now need to set their version on their module rather than on their blueprint. See the [documentation for writing plugins](#).

New features

- Multi-tenancy: Supporting multiple customers per FlexMeasures server, by introducing the *Account* concept. Accounts have users and assets associated. [see [PR #159](#) and [PR #163](#)]
- In the UI, the root view (“/”), the platform name and the visible menu items can now be more tightly controlled (per account roles of the current user) [see also [PR #163](#)]
- Analytics view offers grouping of all assets by location [see [PR #148](#)]
- Add (experimental) endpoint to post sensor data for any sensor. Also supports our ongoing integration with data internally represented using the [timely beliefs](#) lib [see [PR #147](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

Infrastructure / Support

- Add possibility to send errors to Sentry [see [PR #143](#)]
- Add CLI task to monitor if tasks ran successfully and recently enough [see [PR #146](#)]
- Document how to use a custom favicon in plugins [see [PR #152](#)]
- Allow plugins to register multiple Flask blueprints [see [PR #171](#)]
- Continue experimental integration with [timely beliefs](#) lib: link multiple sensors to a single asset [see [PR #157](#)]
- The experimental parts of the data model can now be visualised, as well, via *make show-data-model* (add the `-dev` option in Makefile) [also in [PR #157](#)]

v0.5.0 | June 7, 2021

Warning: If you retrieve weather forecasts through FlexMeasures: we had to switch to OpenWeatherMap, as Dark Sky is closing. This requires an update to config variables — the new setting is called `OPENWEATHERMAP_API_KEY`.

New features

- Allow plugins to overwrite UI routes and customise the teaser on the login form [see [PR #106](#)]
- Allow plugins to customise the copyright notice and credits in the UI footer [see [PR #123](#)]
- Display loaded plugins in footer and support plugin versioning [see [PR #139](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Fix last login date display in user list [see [PR #133](#)]
- Choose better forecasting horizons when weather data is posted [see [PR #131](#)]

Infrastructure / Support

- Add tutorials on how to add and read data from FlexMeasures via its API [see [PR #130](#)]
- For weather forecasts, switch from Dark Sky (closed from Aug 1, 2021) to OpenWeatherMap API [see [PR #113](#)]
- Entity address improvements: add new id-based *fml* scheme, better documentation and more validation support of entity addresses [see [PR #81](#)]
- Re-use the database between automated tests, if possible. This shaves 2/3rd off of the time it takes for the FlexMeasures test suite to run [see [PR #115](#)]
- Make assets use MW as their default unit and enforce that in CLI, as well (API already did) [see [PR #108](#)]
- Let CLI package and plugins use Marshmallow Field definitions [see [PR #125](#)]

- add `time_utils.get_recent_clock_time_window()` function [see [PR #135](#)]

v0.4.1 | May 7, 2021

Bugfixes

- Fix regression when editing assets in the UI [see [PR #122](#)]
- Fixed a regression that stopped asset, market and sensor selection from working [see [PR #117](#)]
- Prevent logging out user when clearing the session [see [PR #112](#)]
- Prevent user type data source to be created without setting a user [see [PR #111](#)]

v0.4.0 | April 29, 2021

Warning: Upgrading to this version requires running `flexmeasures db upgrade` (you can create a backup first with `flexmeasures db-ops dump`).

New features

- Allow for views and CLI functions to come from plugins [see also [PR #91](#)]
- Configure the UI menu with `FLEXMEASURES_LISTED_VIEWS` [see [PR #91](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Asset edit form displayed wrong error message. Also enabled the asset edit form to display the invalid user input back to the user [see [PR #93](#)]

Infrastructure / Support

- Updated dependencies, including Flask-Security-Too [see [PR #82](#)]
- Improved documentation after user feedback [see [PR #97](#)]
- Begin experimental integration with [timely beliefs](#) lib: Sensor data as `TimedBeliefs` [see [PR #79](#) and [PR #99](#)]
- Add sensors with CLI command currently meant for developers only [see [PR #83](#)]
- Add data (beliefs about sensor events) with CLI command currently meant for developers only [see [PR #85](#) and [PR #103](#)]

v0.3.1 | April 9, 2021

Bugfixes

- PostMeterData endpoint was broken in API v2.0 [see [PR #95](#)]

v0.3.0 | April 2, 2021

New features

- FlexMeasures can be installed with `pip` and its CLI commands can be run with `flexmeasures` [see [PR #54](#)]
- Optionally setting recording time when posting data [see [PR #41](#)]
- Add assets and weather sensors with CLI commands [see [PR #74](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Show screenshots in documentation and add some missing content [see [PR #60](#)]
- Documentation listed 2.0 API endpoints twice [see [PR #59](#)]
- Better xrange and title if only schedules are plotted [see [PR #67](#)]
- User page did not list number of assets correctly [see [PR #64](#)]
- Missing *postPrognosis* endpoint for >1.0 API blueprints [part of [PR #41](#)]

Infrastructure / Support

- Added concept pages to documentation [see [PR #65](#)]
- Dump and restore postgres database as CLI commands [see [PR #68](#)]
- Improved installation tutorial as part of [[PR #54](#)]
- Moved developer docs from Readmes into the main documentation [see [PR #73](#)]
- Ensured unique sensor ids for all sensors [see [PR #70](#) and (fix) [PR #77](#)]

v0.2.3 | February 27, 2021

New features

- Power charts available via the API [see [PR #39](#)]
- User management via the API [see [PR #25](#)]
- Better visibility of asset icons on maps [see [PR #30](#)]

Note: Read more on these features on [the FlexMeasures blog](#).

Bugfixes

- Fix maps on new asset page (update MapBox lib) [see [PR #27](#)]
- Some asset links were broken [see [PR #20](#)]
- Password reset link on account page was broken [see [PR #23](#)]

Infrastructure / Support

- CI via Github Actions [see [PR #1](#)]
- Integration with [timely beliefs](#) lib: Sensors [see [PR #13](#)]
- Apache 2.0 license [see [PR #16](#)]
- Load js & css from CDN [see [PR #21](#)]
- Start using marshmallow for input validation, also introducing `HTTP status 422 (Unprocessable Entity)` in the API [see [PR #25](#)]
- Replace `solarpy` with `pvl` (due to license conflict) [see [PR #16](#)]
- Stop supporting the creation of new users on asset creation (to reduce complexity) [see [PR #36](#)]

5.3.4 Benefits

Automation

FlexMeasures provides decision-making support so that the platform operator can schedule flexibility activations. It forecasts the state of assets and proposes the best flexibility activations (shifting or curtailment) for future periods. This is done with modern forecasting and scheduling intelligence.

Insight

Both platform operator and asset owners can monitor the assets - past and current states as well as forecasts are displayed numerically in plots and tables. Activations of flexibility which were ordered in the past can be reviewed. Proposed and scheduled flexibility activations show their expected effects (on imbalance as well as on financial returns).

Autonomy

The companies connected to FlexMeasures only give up as much control as necessary. The asset owners still control the main behaviour of their assets. The owners allow the platform operator to schedule flexibility activations within limits they can set.

Also the platform operator stays in charge: They can choose to approve all proposed flexibility activations manually or to let FlexMeasures automatically schedule them. As FlexMeasures is open source, they can choose to host it themselves or let a third party (like Seita BV) do that.

Profit sharing

The platform operator (as EScO or Aggregator) and asset owners can share the profit made from flexibility activations between them. FlexMeasures plans on providing basic accounting for this.

Note: Read more on flexibility opportunities and activations, as well as profit sharing on [Benefits from energy flexibility](#)

5.3.5 Benefits from energy flexibility

FlexMeasures was created so that the value of energy flexibility can be realised. This will make energy cheaper to use, and can also reduce CO₂ emissions. Here, we define a few terms around this idea, which come up in other parts of this documentation.

- *Flexibility opportunities and activation*
 - *Opportunities*
 - *Activation*
- *An example: the balancing market*
- *Types of flexibility*
 - *Curtailment*
 - *Shifting*
- *Profits of flexibility activation*
 - *Computing value*
 - *Accounting / Sharing value*

Flexibility opportunities and activation

Opportunities

In an energy system with flexible energy assets present (e.g. batteries, heating/cooling), there are opportunities to profit from the availability and activation of their flexibility.

Energy flexibility can come from the ability to store energy (“storage”), to delay (or advance) planned consumption or production (“shifting”), the ability to lower production (“curtailment”), or the ability to increase or decrease consumption (“demand response”) — see [Types of flexibility](#) for a deeper discussion.

Under a given incentive, this flexibility represents an opportunity to profit by scheduling consumption or production differently than originally planned. Within FlexMeasures, flexibility is represented as the difference between a suggested schedule and a given baseline. By default, a baseline is determined by our own forecasts.

Opportunities are expressed with respect to given economical and ecological incentives. For example, a suggested schedule may represent an opportunity to save X EUR and Y tonnes of CO₂.

Activation

The activation of flexibility usually happens in a context of incentives. Often, that context is a market. We recommend the [USEF white paper on the flexibility value chain](#) for an excellent introduction of who can benefit from energy flexibility and how it can be delivered. The high-level takeaways are these:

- the value of flexibility flows back to Prosumers along a chain of roles involved in the activation of their flexibility: the **Flexibility Value Chain**.
- a portfolio of flexible assets (and even individual assets) may provide services in multiple contexts in the same period: **value stacking**.
- **Explicit demand-side flexibility** services involve Aggregators, while **implicit demand-side flexibility** services involve Energy Service Companies (ESCOs).
- Many remuneration components for flexibility services requires the determination of a baseline according to some **baseline methodology**.
- Both availability and activation of flexibility have value.

The overall value (from availability and activation of flexibility), and how this value is shared amongst stakeholders in the various roles in the Flexibility Value Chain, can be accounted for by the platform operator. We talk more about this in *Profits of flexibility activation*.

An example: the balancing market

An example of a market on which flexibility can be activated is the balancing market, which is meant to bring the grid frequency back to a target level within a matter of minutes. Consider the aforementioned differences between suggested schedules and a given baseline. In the context of the balancing market, differences indicating an increase in production or a decrease in consumption on activation both result in an increasing grid frequency (back towards the target frequency).

The balancing market pays for such services, and they are often referred to as “*up-regulation*”. It works the other way around, too: differences indicating a decrease in production or an increase in consumption both result in a decreasing grid frequency (“*down-regulation*”).

Types of flexibility

The FlexMeasures platform distinguishes between different types of flexibility. We explain them here in more detail, together with examples.

Curtailment

Curtailment happens when an asset temporarily lowers or stops its production or consumption. A defining feature of curtailment is that total production or consumption decreases when this this flexibility is activated.

- A typical example of curtailing production is when a wind turbine adjusts the pitch angle of its blades to decrease the generator torque.
- An example of curtailing consumption is load shedding of energy intensive industries.

Curtailment offers may specify some freedom in terms of how much energy can be curtailed. In these cases, the user can select the energy volume (in MWh) to be ordered, within constraints set by the relevant Prosumer. The net effect of a curtailment action is also measured in terms of an energy volume (see the flexibility metrics in the portfolio page).

Note that the volume ordered is not necessarily equal to the volume curtailed: the ordered volume relates only to the selected time window, while the curtailed volume may include volumes outside of the selected time window. For

example, an asset that runs an all-or-nothing consumption process of 2 hours can be ordered to curtail consumption for 1 hour, but will in effect stop the entire process. In this case, the curtailed volume will be higher than the ordered volume, and the platform will take into account the total expected curtailment in its calculations.

Shifting

Shifting happens when an asset delays or advances its energy production or consumption. A defining feature of shifting is that total production or consumption remains the same when this flexibility is activated.

- An example of delaying consumption is when a charging station postpones the charging process of an electric vehicle.
- An example of advancing consumption is when a cooling unit starts to cool before the upper temperature bound was reached (pre-cooling).

Shifting offers may specify some freedom in terms of how much energy can be shifted. In these cases, the user can select the energy volume (in MWh) to be ordered, within constraints set by the relevant Prosumer. This energy volume represents how much energy is shifting into or out of the selected time window. The net effect of a shifting action is measured in terms of an energy-time volume (see the flexibility metrics in the portfolio page). This volume is a multiplication of the energy volume being shifted and the duration of that shift.

Profits of flexibility activation

The realised value from activating flexibility has to be computed and accounted for. Both of these activities depend on the context in which FlexMeasures is being used, and we expect that it will often have to be implemented in a custom manner (much as the actual scheduling optimisation).

Todo: Making it possible to configure custom scheduling and value accounting is on the roadmap for FlexMeasures.

Computing value

The computation of the value is what drives the scheduling optimisation. This value is usually monetary, and in that case there should be some form of market configured. This can be a constant or time-of-use tariff, or a real market. However, there are other possibilities, for instance if the optimisation goal is to minimise CO₂ emissions. Then, the realised value is avoided CO₂, which nowadays has an assumed value, e.g. in [the EU ETS carbon market](#).

Accounting / Sharing value

The realisation of payments is outside of the scope of FlexMeasures, but it can provide the accounting to enable them (as was said above, this is usually a part of the optimisation problem formulation).

However, next to fuelling algorithmic optimisation, the way that the value of energy flexibility is shared among the stakeholders will also be an important driver for project participation. Accounting plays an important role here.

There are different roles in a modern smart energy system (e.g. “Prosumer”, “DSO”, Aggregator”, “ESCo”), and they all enjoy the benefits of flexibility in different ways (see for example [this resource](#) for more details).

In our opinion, the only way to successful implementation of energy flexibility is if profits are shared between these stakeholders. This assumes contractual relationships. Use cases which FlexMeasures can support well are the following relationships:

- between Aggregator and Prosumer, where the Aggregator sells the balancing power to a third party and shares the profits with the Prosumer according to some contracted method for profit sharing. In this case the stated costs and revenues for the Prosumer may be after deducting the Aggregator fee (which typically include price components per flex activation and price components per unit of time, but may include arbitrarily complex price components).
- between ESCo and Prosumer, where the ESCo advises the Prosumer to optimise against e.g. dynamic prices. Likewise, stated numbers may be after deducting the ESCo fee.

FlexMeasures can take these intricacies into account if a custom optimisation algorithm is plugged in to model them.

Alternatively, we can assume that all profit from activating flexibility goes to the Prosumer, or simply report the profits before sharing (and before deducting any service fees).

5.3.6 In-built smart functionality

The main purpose of the FlexMeasures platform is to serve as a basis to rapidly build energy flexibility services. Much software architecture and wiring groundwork is already included for this purpose, like an API, support for plotting and multi-tenancy and extensibility.

That said, several smart features come with FlexMeasures. Once the sensor structure and data is in place, they should be usable without much coding.

Todo: We'll write more tutorials on this.

Monitoring

The FlexMeasures platform continuously reads in meter data from your assets. To assist your maintenance, it can alert you to situations which need your attention:

- Breaches of thresholds (protect devices)
- Data gaps & strange outliers (assure data quality)
- Idle processes / leaks (minimise waste)

Todo: These features are [work in progress](#). Most of our customers already do this by themselves in a straightforward manner.

Forecasting

The FlexMeasures platform continuously creates forecasts for the rest of day.

All relevant data should be forecasted:

- Energy assets
- Weather data
- Market prices

Scheduling

The FlexMeasures platform optimises schedules for your flexible assets. This is where energy flexibility is valorised!

Examples are:

- Charging schedules of batteries
- Heat pumps management
- Buffering of machinery

The goals can be maximal cost savings, maximal usage of solar power or stable energy supply for the most crucial consumers.

5.3.7 Algorithms

- *Forecasting*
- *Scheduling*
 - *Storage devices*
- *Possible future work on algorithms*
 - *More configurable forecasting*
 - *Other optimisation goals for scheduling*
 - *Scheduling of other flexible asset types*
 - *Broker algorithm*
 - *Trading algorithm*

Forecasting

Forecasting algorithms are used by FlexMeasures to assess the likelihood of future consumption/production and prices. Weather forecasting is included in the platform, but is usually not the result of an internal algorithm (weather forecast services are being used by import scripts, e.g. with [this tool](#)).

FlexMeasures uses linear regression and falls back to naive forecasting of the last known value if errors happen. What might be even more important than the type of algorithm is the features handed to the model — lagged values (e.g. value of the same time yesterday) and regressors (e.g. wind speed prediction to forecast wind power production).

The performance of our algorithms is indicated by the mean absolute error (MAE) and the weighted absolute percentage error (WAPE). Power profiles on an asset level often include zero values, such that the mean absolute percentage error (MAPE), a common statistical measure of forecasting accuracy, is undefined. For such profiles, it is more useful to report the WAPE, which is also known as the volume weighted MAPE. The MAE of a power profile gives an indication of the size of the uncertainty in consumption and production. This allows the user to compare an asset's predictability to its flexibility, i.e. to the size of possible flexibility activations.

Example benchmarks per asset type are listed in the table below for various assets and forecasting horizons. FlexMeasures updates the benchmarks automatically for the data currently selected by the user. Amongst other factors, accuracy is influenced by:

- The chosen metric (see below)
- Resolution of the forecast

- Horizon of the forecast
- Asset type
- Location / Weather conditions
- Level of aggregation

Accuracies in the table are reported as 1 minus WAPE, which can be interpreted as follows:

- 100% accuracy denotes that all values are correct.
- 50% accuracy denotes that, on average, the values are wrong by half of the reference value.
- 0% accuracy denotes that, on average, the values are wrong by exactly the reference value (i.e. zeros or twice the reference value).
- negative accuracy denotes that, on average, the values are off-the-chart wrong (by more than the reference value itself).

Asset Average power per asset	Building 204 W	Charge Points 75 W	Solar 140 W	Wind (offshore) 518 W	Day-ahead market
1 - WAPE (1 hour ahead)	93.4 %	87.6 %	95.2 %	81.6 %	88.0 %
1 - WAPE (6 hours ahead)	92.6 %	73.0 %	83.7 %	73.8 %	81.9 %
1 - WAPE (24 hours ahead)	92.4 %	65.2 %	46.1 %	60.1 %	81.4 %
1 - WAPE (48 hours ahead)	92.1 %	63.7 %	43.3 %	56.9 %	72.3 %

Defaults:

- The application uses an ordinary least squares auto-regressive model with external variables.
- Lagged outcome variables are selected based on the periodicity of the asset (e.g. daily and/or weekly).
- Common external variables are weather forecasts of temperature, wind speed and irradiation.
- Timeseries data with frequent zero values are transformed using a customised Box-Cox transformation.
- To avoid over-fitting, cross-validation is used.
- Before fitting, explicit annotations of expert knowledge to the model (like the definition of asset-specific seasonality and special time events) are possible.
- The model is currently fit each day for each asset and for each horizon.

Improvements:

- Most assets have yearly seasonality (e.g. wind, solar) and therefore forecasts would benefit from ≥ 2 years of history.

Scheduling

Given price conditions or other conditions of relevance, a scheduling algorithm is used by the Aggregator (in case of explicit DR) or by the Energy Service Company (in case of implicit DR) to form a recommended schedule for the Prosumer's flexible assets.

Storage devices

So far, FlexMeasures provides algorithms for storage — for batteries (e.g. home batteries or EVs) and car charging stations. We thus cover the asset types “battery”, “one-way_evse” and “two-way_evse”.

These algorithms schedule the storage assets based directly on the latest beliefs regarding market prices, within the specified time window. They are mixed integer linear programs, which are configured in FlexMeasures and then handed to a dedicated solver.

For all scheduling algorithms, a starting state of charge (SOC) as well as a set of SOC targets can be given. If no SOC is available, we set the starting SOC to 0.

Also, per default we incentivise the algorithms to prefer scheduling charging now rather than later, and discharging later rather than now. We achieve this by adding a tiny artificial price slope. We penalise the future with at most 1 per thousand times the price spread. This behaviour can be turned off with the *prefer_charging_sooner* parameter set to *False*.

Note: For the resulting consumption schedule, consumption is defined as positive values.

Possible future work on algorithms

Enabling more algorithmic expression in FlexMeasures is crucial. This are a few ideas for future work. Some of them are excellent topics for Bachelor or Master theses. so get in touch if that is of interest to you.

More configurable forecasting

On the roadmap for FlexMeasures is to make features easier to configure, especially regressors. Furthermore, we plan to add more types of forecasting algorithms, like random forest or even LSTM.

Other optimisation goals for scheduling

Next to market prices, optimisation goals like reduced CO₂ emissions are sometimes required. There are multiple ways to measure this, e.g. against the CO₂ mix in the grid, or the use of fossil fuels.

Scheduling of other flexible asset types

Next to storage, there are other interesting flexible assets which can require specific implementations. For shifting, there are heat pumps and other buffers. For curtailment, there are wind turbines and solar panels.

Note: See *Types of flexibility* for more info on shifting and curtailment.

Broker algorithm

A broker algorithm is used by the Aggregator to analyse flexibility in the Supplier's portfolio of assets, and to suggest the most valuable flexibility activations to take for each time slot. The differences to single-asset scheduling are that these activations are based on a helicopter perspective (the Aggregator optimises a portfolio, not a single asset) and that the flexibility offers are presented to the Supplier in the form of an order book.

Trading algorithm

A trading algorithm is used to assist the Supplier with its decision-making across time slots, based on the order books made by the broker (see above). The algorithm suggests which offers should be accepted next, and the Supplier may automate its decision-making by letting the algorithm place orders on its behalf.

A default approach would be a myopic greedy strategy — order all flexibility opportunities with a positive expected value in the first available timeslot, then those in the second available timeslot, and so on.

5.3.8 Security aspects

Data

There are two types of data on FlexMeasures servers - files (e.g. source code, images) and data in a database (e.g. user data and time series for energy consumption/generation or weather).

- Files are stored on EBS volumes on Amazon Web Services. These are shared with other customers of Amazon, but protected from them by Linux's chroot system – each user can see only the files in their own section of the disk.
- Database data is stored in PostgresDB instances which are not shared with other Amazon customers. They are password-protected.
- Finally, The application communicates all data with HTTPS, the Hypertext Transfer Protocol encrypted by Transport Layer Security. This is used even if the application is accessed via `http://`.

Authentication

Authentication is the system by which users tell the FlexMeasures platform that they are who they claim they are. This involves a username/password combination ("credentials") or an access token.

- No user passwords are stored in clear text on any server - the FlexMeasures platform only stores the hashed passwords (encrypted with the [bcrypt hashing algorithm](#)). If an attacker steals these password hashes, they cannot compute the passwords from them in a practical amount of time.
- Access tokens are used so that the sending of usernames and passwords is limited (even if they are encrypted via https, see above) when dealing with the part of the FlexMeasures platform which sees the most traffic: the API functionality. Tokens thus have use cases for some scenarios, where developers want to treat authentication information with a little less care than credentials should be treated with, e.g. sharing among computers. However, they also expire fast, which is a common industry practice (by making them short-lived and requiring refresh, FlexMeasures limits the time an attacker can abuse a stolen token). At the moment, the access tokens on FlexMeasures platform expire after six hours. Access tokens are encrypted and validated with the [sha256_crypt algorithm](#), and the functionality to expire tokens is realised by storing the seconds since January 1, 2011 in the token. The maximum age of access tokens in FlexMeasures can be altered by setting the env variable `SECURITY_TOKEN_MAX_AGE` to the number of seconds after which tokens should expire.

Note: Authentication (and authorization, see below) affects the FlexMeasures API and UI. The CLI (command line interface) can only be used if the user is already on the server and can execute `flexmeasures` commands, thus we can safely assume they are admins.

Authorization

Authorization is the system by which the FlexMeasures platform decides whether an authenticated user can access data. Data about users and assets. Or metering data, forecasts and schedules.

For instance, a user is authorized to update his or her personal data, like the surname. Other users should not be authorized to do that. We can also authorize users to do something because they belong to a certain account. An example for this is to read the meter data of the account's assets. Any regular user should *only* be able to read data that their account should be able to see.

Note: Each user belongs to exactly one account.

In a nutshell, the way FlexMeasures implements authorization works as follows: The data models codify under which conditions a user can have certain permissions to work with their data. Permissions allow distinct ways of access like reading, writing or deleting. The API endpoints are where we know what needs to happen to what data, so there we make sure that the user has the necessary permissions.

We already discussed certain conditions under which a user has access to data — being a certain user or belonging to a specific account. Furthermore, authorization conditions can also be implemented via *roles*:

- **Account roles** are often used for authorization. We support several roles which are mentioned in the USEF framework but more roles are possible (e.g. defined by custom-made services, see below). For example, a user might be authorized to write sensor data if they belong to an account with the “MDC” account role (“MDC” being short for meter data company).
- **User roles** give a user personal authorizations. For instance, we have a few *admins* who can perform all actions, and *admin-readers* who can read everything. Other roles have only an effect within the user's account, e.g. there could be an “HR” role which allows to edit user data like surnames within the account.
- Roles cannot be edited via the UI at the moment. They are decided when a user or account is created in the CLI (for adding roles later, we use the database for now). Editing roles in UI and CLI is future work.

Note: Custom energy flexibility services developed on top of FlexMeasures also need to implement authorization. More on this in [Custom authorization](#). Here is an example for a custom authorization concept: services can use account roles to achieve their custom authorization. E.g. if several services run on one FlexMeasures server, each service could define a “MyService-subscriber” account role, to make sure that only users of such accounts can use the endpoints.

5.3.9 Toy example: Scheduling a battery, from scratch

Let's walk through an example from scratch! We'll ...

- install FlexMeasures
- create an account with a battery asset
- load hourly prices
- optimize a 12h-schedule for a battery that is half full

What do you need? Your own computer, with one of two situations: either you have [Docker](#) or your computer supports Python 3.8+, pip and PostgresDB. The former might be easier, see the installation step below. But you choose.

Below are the flexmeasures CLI commands we'll run, and which we'll explain step by step. There are some other crucial steps for installation and setup, so this becomes a complete example from scratch, but this is the meat:

```
# setup an account with a user, battery (ID 1) and market (ID 2)
$ flexmeasures add toy-account --kind battery
# load prices to optimise the schedule against
$ flexmeasures add beliefs --sensor-id 2 --source toy-user prices-tomorrow.csv --
  ↪timezone Europe/Amsterdam
# make the schedule
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --start ${TOMORROW}T07:00+01:00 --duration PT12H \
  --soc-at-start 50% --roundtrip-efficiency 90%
```

Okay, let's get started!

Note: You can copy the commands by hovering on the top right corner of code examples. You'll copy only the commands, not the output!

Install Flexmeasures and the database

Docker

If [docker](#) is running on your system, you're good to go. Otherwise, see [here](#).

We start by installing the FlexMeasures platform, and then use Docker to run a postgres database and tell FlexMeasures to create all tables.

```
$ docker pull lfenergy/flexmeasures:latest
$ docker pull postgres
$ docker network create flexmeasures_network
$ docker run --rm --name flexmeasures-tutorial-db -e POSTGRES_PASSWORD=fm-db-passwd -e_
  ↪POSTGRES_DB=flexmeasures-db -d --network=flexmeasures_network postgres:latest
$ docker run --rm --name flexmeasures-tutorial-fm --env SQLALCHEMY_DATABASE_
  ↪URI=postgresql://postgres:fm-db-passwd@flexmeasures-tutorial-db:5432/flexmeasures-db --
  ↪env SECRET_KEY=notsecret --env FLASK_ENV=development --env LOGGING_LEVEL=INFO -d --
  ↪network=flexmeasures_network -p 5000:5000 lfenergy/flexmeasures
$ docker exec flexmeasures-tutorial-fm bash -c "flexmeasures db upgrade"
```

Note: A tip on Linux/macOS — You might have the `docker` command, but need *sudo* rights to execute it. `alias docker='sudo docker'` enables you to still run this tutorial.

Now - what's *very important* to remember is this: The rest of this tutorial will happen *inside* the `flexmeasures-tutorial-fm` container! This is how you hop inside the container and run a terminal there:

```
$ docker exec -it flexmeasures-tutorial-fm bash
```

To leave the container session, hold CTRL-D or type “exit”.

To stop the containers, you can type

```
$ docker stop flexmeasures-tutorial-db
$ docker stop flexmeasures-tutorial-fm
```

To start the containers again, do this (note that re-running the *docker run* commands above *deletes and re-creates* all data!):

```
$ docker start flexmeasures-tutorial-db
$ docker start flexmeasures-tutorial-fm
```

Note: For newer versions of MacOS, port 5000 is in use by default by Control Center. You can turn this off by going to System Preferences > Sharing and untick the “Airplay Receiver” box. If you don’t want to do this for some reason, you can change the host port in the `docker run` command to some other port, for example 5001. To do this, change `-p 5000:5000` in the command to `-p 5001:5000`. If you do this, remember that you will have to go to `localhost:5001` in your browser when you want to inspect the FlexMeasures UI.

Note: Got docker-compose? You could run this tutorial with 5 containers :) — Go to *Seeing it work: Running the toy tutorial*.

On your PC

This example is from scratch, so we’ll assume you have nothing prepared but a (Unix) computer with Python (3.8+) and two well-known developer tools, `pip` and `postgres`.

We’ll create a database for FlexMeasures:

```
$ sudo -i -u postgres
$ createdb -U postgres flexmeasures-db
$ createuser --pwprompt -U postgres flexmeasures-user      # enter your password, we'll
→ use "fm-db-passwd"
$ exit
```

Then, we can install FlexMeasures itself, set some variables and tell FlexMeasures to create all tables:

```
$ pip install flexmeasures
$ export SQLALCHEMY_DATABASE_URI="postgresql://flexmeasures-user:fm-db-
→ passwd@localhost:5432/flexmeasures-db" SECRET_KEY=notsecret LOGGING_LEVEL="INFO"
→ DEBUG=0
$ export FLASK_ENV="development"
$ flexmeasures db upgrade
```

Note: When installing with `pip`, on some platforms problems might come up (e.g. macOS, Windows). One reason is that FlexMeasures requires some libraries with lots of C code support (e.g. Numpy). One way out is to use Docker, which uses a prepared Linux image, so it'll definitely work.

In case you want to re-run the tutorial, then it's recommended to delete the old database and create a fresh one. Run the following command to create a clean database with a new user, where it is optional. If you don't provide the user, then the default `postgres` user will be used to create the database.

```
$ make clean-db db_name=flexmeasures-db [db_user=flexmeasures]
```

Add some structural data

The data we need for our example is both structural (e.g. a company account, a user, an asset) and numeric (we want market prices to optimize against).

Let's create the structural data first.

FlexMeasures offers a command to create a toy account with a battery:

```
$ flexmeasures add toy-account --kind battery
```

```
Toy account Toy Account with user toy-user@flexmeasures.io created successfully. You
↪ might want to run `flexmeasures show account --id 1`
The sensor recording battery power is <Sensor 1: discharging, unit: MW res.: 0:15:00>.
The sensor recording day-ahead prices is <Sensor 2: day-ahead prices, unit: EUR/MWh res.
↪: 1:00:00>.
The sensor recording solar forecasts is <Sensor 3: production, unit: MW res.: 0:15:00>.
```

And with that, we're done with the structural data for this tutorial!

If you want, you can inspect what you created:

```
$ flexmeasures show account --id 1
```

```
=====
Account Toy Account (ID: 1)
=====
```

Account has no roles.

All users:

Id	Name	Email	Last Login	Roles
1	toy-user	toy-user@flexmeasures.io		account-admin

All assets:

ID	Name	Type	Location
1	toy-battery	battery	(52.374, 4.88969)
3	toy-solar	solar	(52.374, 4.88969)

(continues on next page)

(continued from previous page)

```
$ flexmeasures show asset --id 1
```

```
=====
Asset toy-battery (ID: 1)
=====
```

Type	Location	Attributes
battery	(52.374, 4.88969)	capacity_in_mw: 0.5 min_soc_in_mwh: 0.05 max_soc_in_mwh: 0.45 sensors_to_show: [2, [3, 1]]

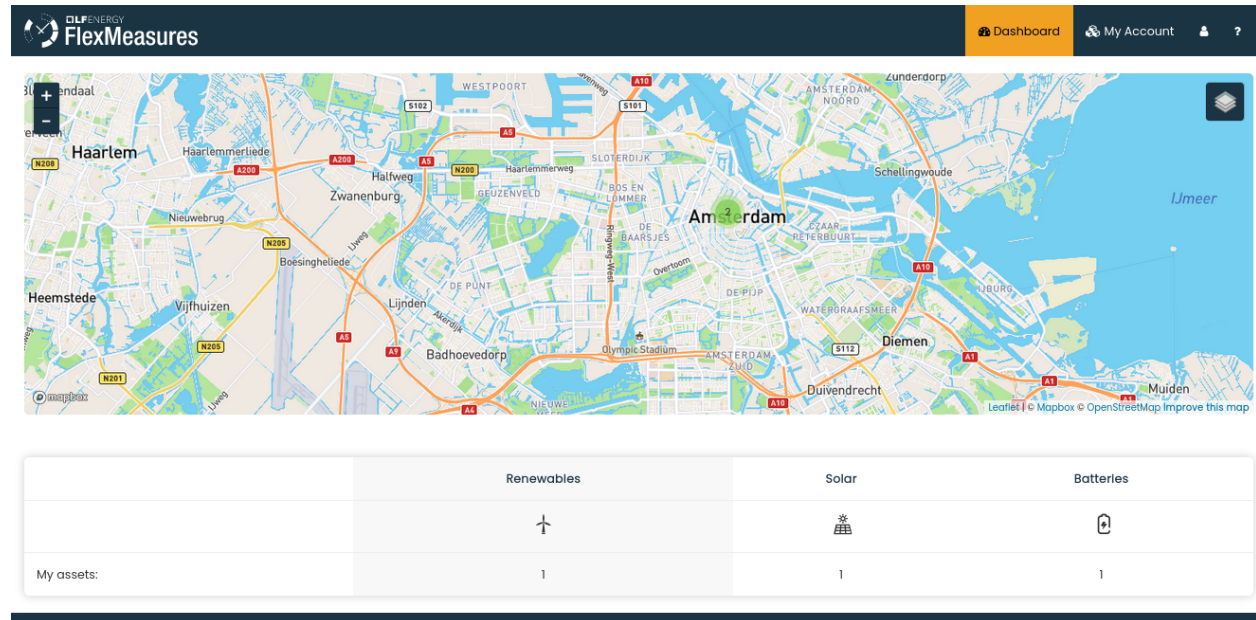
All sensors in asset:

ID	Name	Unit	Resolution	Timezone	Attributes
1	discharging	MW	15 minutes	Europe/Amsterdam	

Yes, that is quite a large battery :)

Note: Obviously, you can use the `flexmeasures` command to create your own, custom account and assets. See *CLI Commands*. And to create, edit or read asset data via the API, see *Version 3.0*.

We can also look at the battery asset in the UI of FlexMeasures (in Docker, the FlexMeasures web server already runs, on your PC you can start it with `flexmeasures run`). Visit <http://localhost:5000/> (username is “toy-user@flexmeasures.io”, password is “toy-password”):



Note: You won't see the map tiles, as we have not configured the `MAPBOX_ACCESS_TOKEN`. If you have one, you can configure it via `flexmeasures.cfg` (for Docker, see *Configuration and customization*).

Add some price data

Now to add price data. First, we'll create the csv file with prices (EUR/MWh, see the setup for sensor 2 above) for tomorrow.

```
$ TOMORROW=$(date --date="next day" '+%Y-%m-%d')
$ echo "Hour,Price"
$ ${TOMORROW}T00:00:00,10
$ ${TOMORROW}T01:00:00,11
$ ${TOMORROW}T02:00:00,12
$ ${TOMORROW}T03:00:00,15
$ ${TOMORROW}T04:00:00,18
$ ${TOMORROW}T05:00:00,17
$ ${TOMORROW}T06:00:00,10.5
$ ${TOMORROW}T07:00:00,9
$ ${TOMORROW}T08:00:00,9.5
$ ${TOMORROW}T09:00:00,9
$ ${TOMORROW}T10:00:00,8.5
$ ${TOMORROW}T11:00:00,10
$ ${TOMORROW}T12:00:00,8
$ ${TOMORROW}T13:00:00,5
$ ${TOMORROW}T14:00:00,4
$ ${TOMORROW}T15:00:00,4
$ ${TOMORROW}T16:00:00,5.5
$ ${TOMORROW}T17:00:00,8
$ ${TOMORROW}T18:00:00,12
$ ${TOMORROW}T19:00:00,13
$ ${TOMORROW}T20:00:00,14
$ ${TOMORROW}T21:00:00,12.5
$ ${TOMORROW}T22:00:00,10
$ ${TOMORROW}T23:00:00,7" > prices-tomorrow.csv
```

This is time series data, in FlexMeasures we call “beliefs”. Beliefs can also be sent to FlexMeasures via API or imported from open data hubs like [ENTSO-E](#) or [OpenWeatherMap](#). However, in this tutorial we'll show how you can read data in from a CSV file. Sometimes that's just what you need :)

```
$ flexmeasures add beliefs --sensor-id 2 --source toy-user prices-tomorrow.csv --
→timezone Europe/Amsterdam
Successfully created beliefs
```

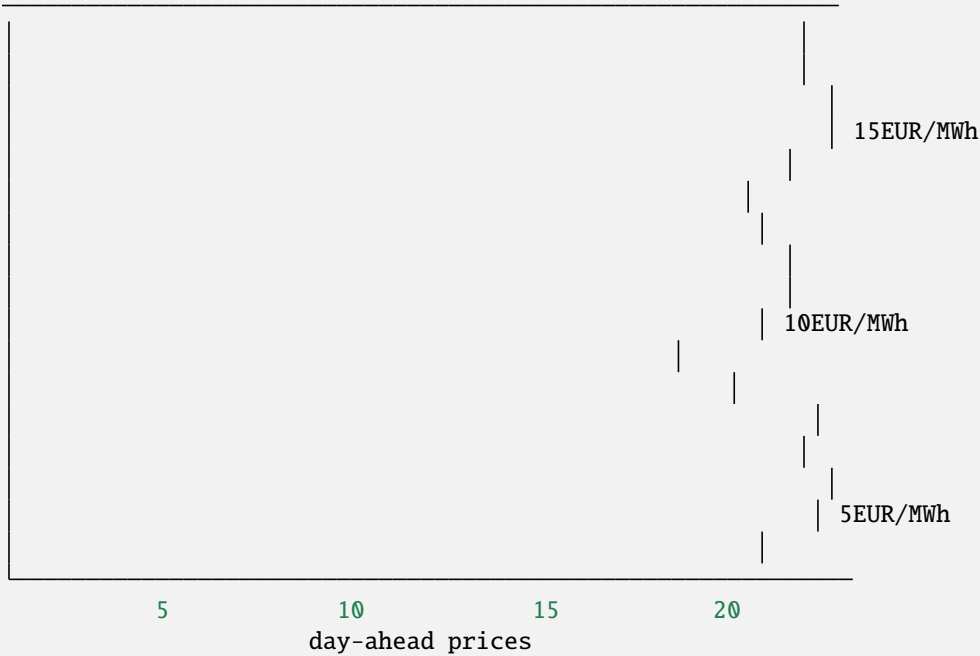
In FlexMeasures, all beliefs have a data source. Here, we use the username of the user we created earlier. We could also pass a user ID, or the name of a new data source we want to use for CLI scripts.

Note: Attention: We created and imported prices where the times have no time zone component! That happens a lot. FlexMeasures can localize them for you to a given timezone. Here, we localized the data to the timezone of the price sensor - Europe/Amsterdam - so the start time for the first price is `2022-03-03 00:00:00+01:00` (midnight in Amsterdam).

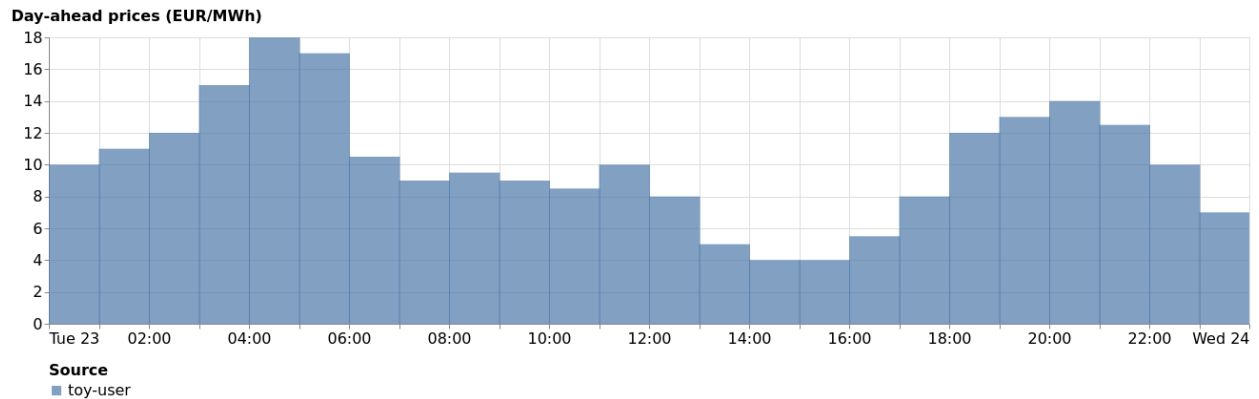
Let's look at the price data we just loaded:

```
$ flexmeasures show beliefs --sensor-id 2 --start ${TOMORROW}T00:00:00+01:00 --duration PT24H
```

Beliefs for Sensor 'day-ahead prices' (ID 2).
Data spans a day and starts at 2022-03-03 00:00:00+01:00.
The time resolution (x-axis) is an hour.



Again, we can also view these prices in the [FlexMeasures UI](#):



Note: Technically, these prices for tomorrow may be forecasts (depending on whether you are running through this tutorial before or after the day-ahead market's gate closure). You can also use FlexMeasures to compute forecasts yourself. See [Forecasting & scheduling](#).

Make a schedule

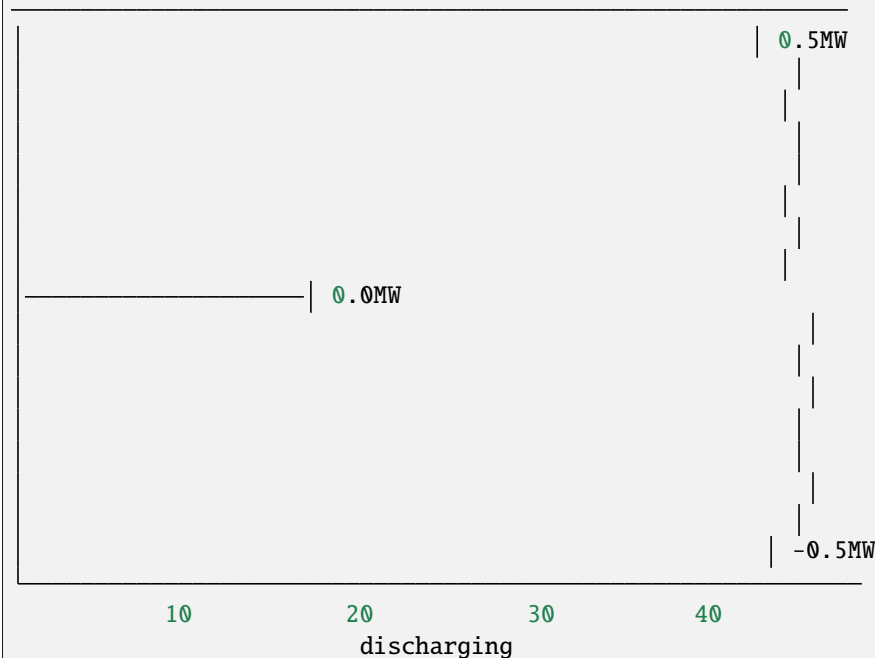
Finally, we can create the schedule, which is the main benefit of FlexMeasures (smart real-time control).

We'll ask FlexMeasures for a schedule for our discharging sensor (ID 1). We also need to specify what to optimise against. Here we pass the Id of our market price sensor (3). To keep it short, we'll only ask for a 12-hour window starting at 7am. Finally, the scheduler should know what the state of charge of the battery is when the schedule starts (50%) and what its roundtrip efficiency is (90%).

```
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --start ${TOMORROW}T07:00+01:00 --duration PT12H \
  --soc-at-start 50% --roundtrip-efficiency 90%
New schedule is stored.
```

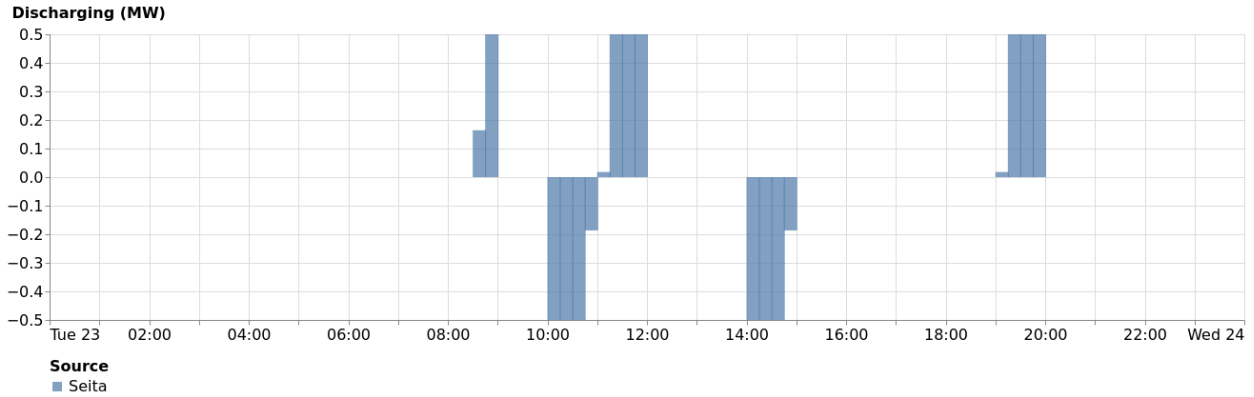
Great. Let's see what we made:

```
$ flexmeasures show beliefs --sensor-id 1 --start ${TOMORROW}T07:00:00+01:00 --duration PT12H
Beliefs for Sensor 'discharging' (ID 1).
Data spans 12 hours and starts at 2022-03-04 07:00:00+01:00.
The time resolution (x-axis) is 15 minutes.
```



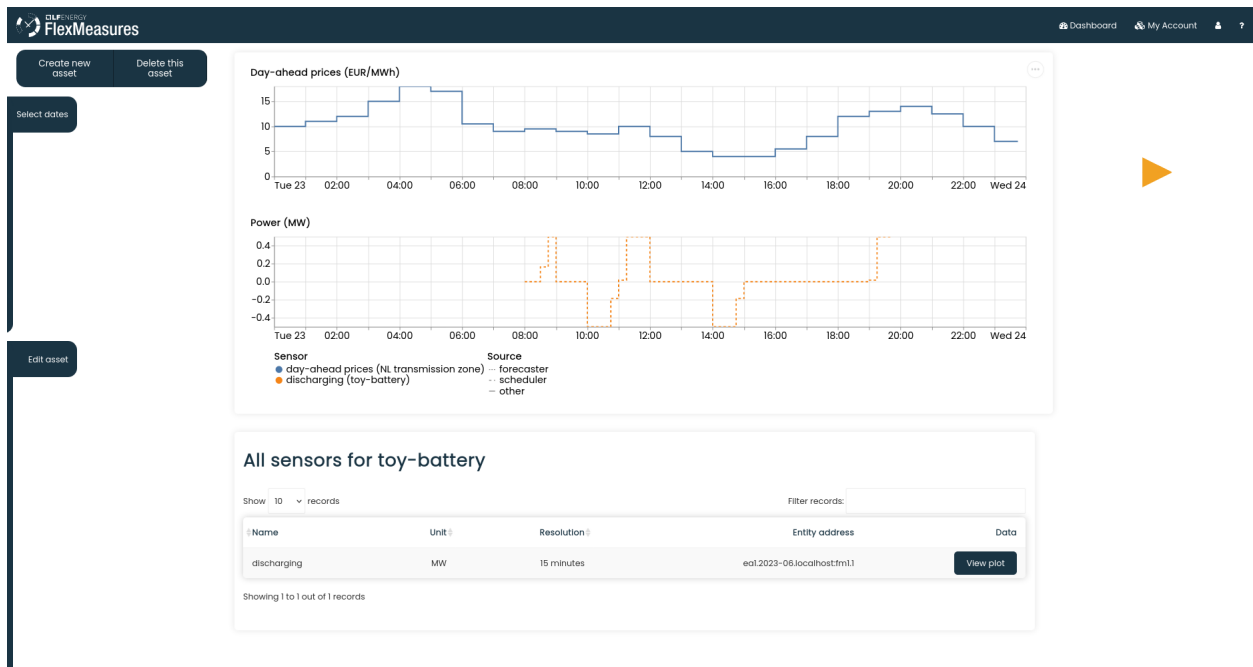
Here, negative values denote output from the grid, so that's when the battery gets charged.

We can also look at the charging schedule in the [FlexMeasures UI](#) (reachable via the asset page for the battery):



Recall that we only asked for a 12 hour schedule here. We started our schedule *after* the high price peak (at 4am) and it also had to end *before* the second price peak fully realised (at 8pm). Our scheduler didn't have many opportunities to optimize, but it found some. For instance, it does buy at the lowest price (at 2pm) and sells it off at the highest price within the given 12 hours (at 6pm).

The [asset page for the battery](#) shows both prices and the schedule.



Note: The flexmeasures add schedule for-storage command also accepts state-of-charge targets, so the schedule can be more sophisticated. But that is not the point of this tutorial. See [flexmeasures add schedule](#)

```
for-storage --help.
```

Take into account solar production

So far we haven't taken into account any other devices that consume or produce electricity. We'll now add solar production forecasts and reschedule, to see the effect of solar on the available headroom for the battery.

First, we'll create a new csv file with solar forecasts (MW, see the setup for sensor 3 above) for tomorrow.

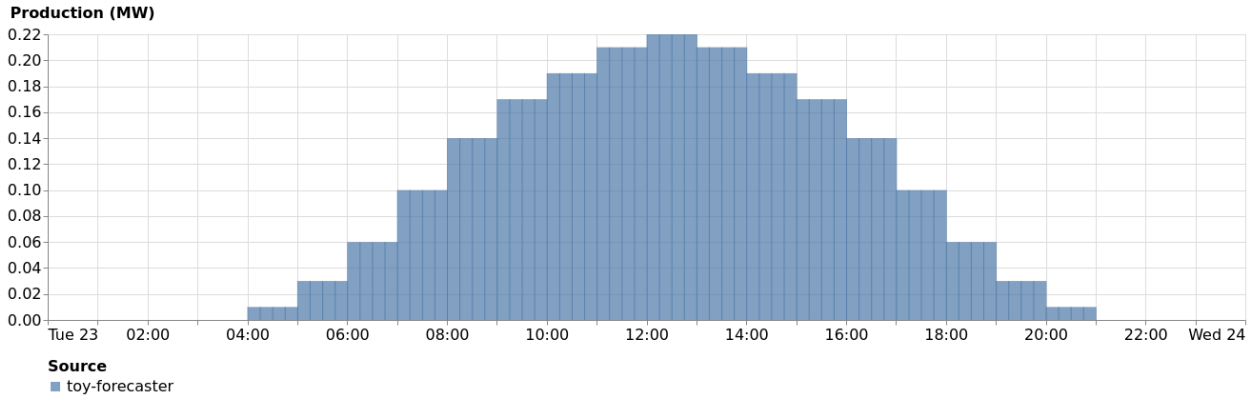
```
$ TOMORROW=$(date --date="next day" '+%Y-%m-%d')
$ echo "Hour,Price
$ ${TOMORROW}T00:00:00,0.0
$ ${TOMORROW}T01:00:00,0.0
$ ${TOMORROW}T02:00:00,0.0
$ ${TOMORROW}T03:00:00,0.0
$ ${TOMORROW}T04:00:00,0.01
$ ${TOMORROW}T05:00:00,0.03
$ ${TOMORROW}T06:00:00,0.06
$ ${TOMORROW}T07:00:00,0.1
$ ${TOMORROW}T08:00:00,0.14
$ ${TOMORROW}T09:00:00,0.17
$ ${TOMORROW}T10:00:00,0.19
$ ${TOMORROW}T11:00:00,0.21
$ ${TOMORROW}T12:00:00,0.22
$ ${TOMORROW}T13:00:00,0.21
$ ${TOMORROW}T14:00:00,0.19
$ ${TOMORROW}T15:00:00,0.17
$ ${TOMORROW}T16:00:00,0.14
$ ${TOMORROW}T17:00:00,0.1
$ ${TOMORROW}T18:00:00,0.06
$ ${TOMORROW}T19:00:00,0.03
$ ${TOMORROW}T20:00:00,0.01
$ ${TOMORROW}T21:00:00,0.0
$ ${TOMORROW}T22:00:00,0.0
$ ${TOMORROW}T23:00:00,0.0" > solar-tomorrow.csv
```

Then, we read in the created CSV file as beliefs data. This time, different to above, we want to use a new data source (not the user) — it represents whoever is making these solar production forecasts. We create that data source first, so we can tell *flexmeasures add beliefs* to use it. Setting the data source type to “forecaster” helps FlexMeasures to visualize distinguish its data from e.g. schedules and measurements.

Note: The `flexmeasures add source` command also allows to set a model and version, so sources can be distinguished in more detail. But that is not the point of this tutorial. See `flexmeasures add source --help`.

```
$ flexmeasures add source --name "toy-forecaster" --type forecaster
Added source <Data source 4 (toy-forecaster)>
$ flexmeasures add beliefs --sensor-id 3 --source 4 solar-tomorrow.csv --timezone Europe/
↪ Amsterdam
Successfully created beliefs
```

The one-hour CSV data is automatically resampled to the 15-minute resolution of the sensor that is recording solar production. We can see solar production in the [FlexMeasures UI](#) :

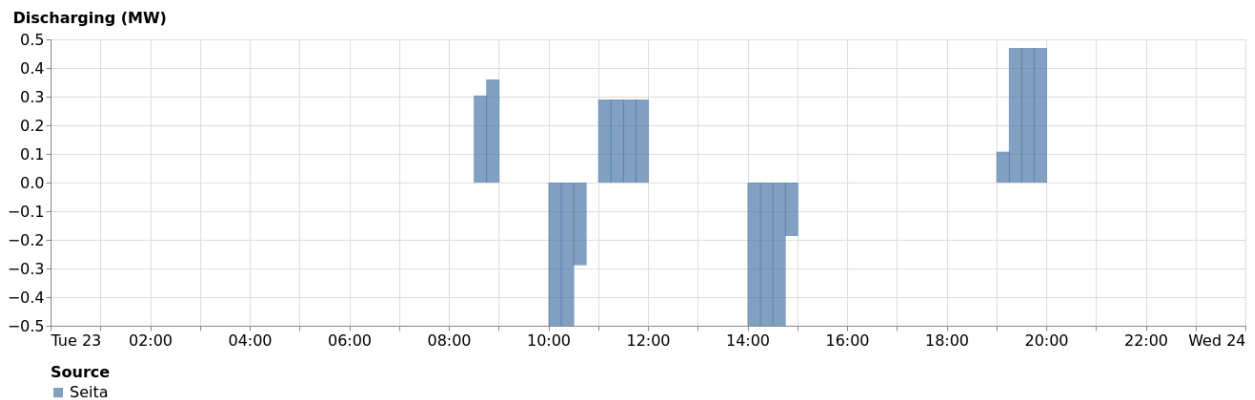


Note: The `flexmeasures add beliefs` command has many options to make sure the read-in data is correctly interpreted (unit, timezone, delimiter, etc). But that is not the point of this tutorial. See `flexmeasures add beliefs --help`.

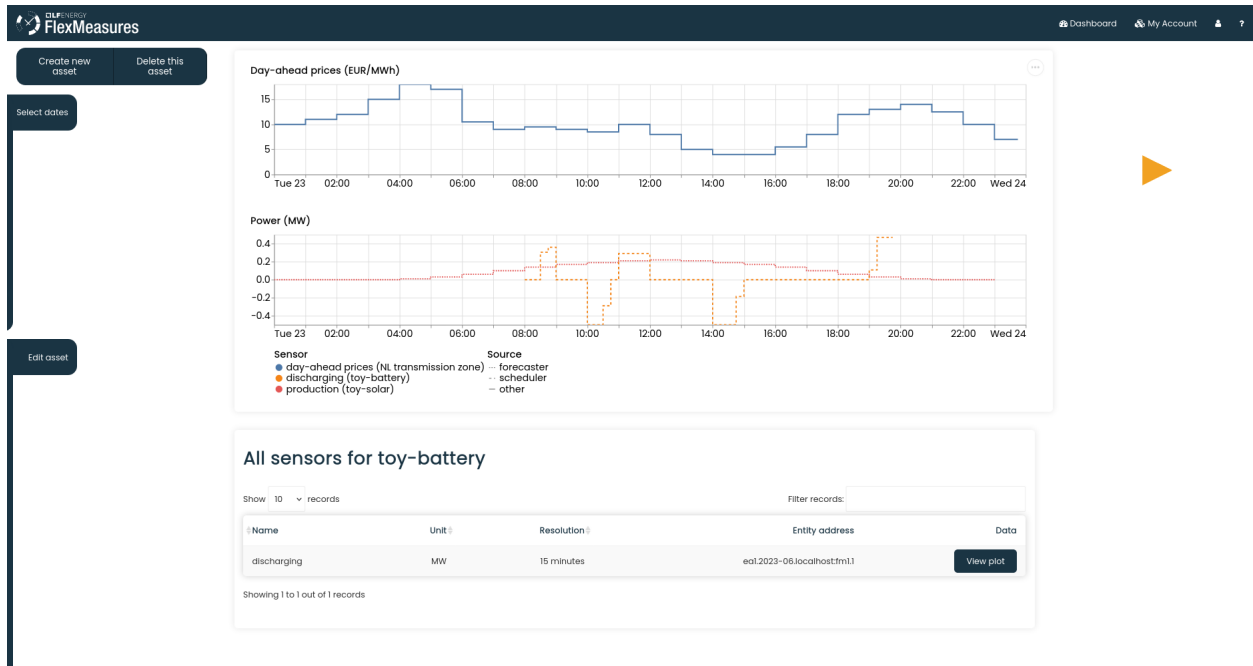
Now, we'll reschedule the battery while taking into account the solar production. This will have an effect on the available headroom for the battery.

```
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --inflexible-device-sensor 3 \
  --start ${TOMORROW}T07:00+01:00 --duration PT12H \
  --soc-at-start 50% --roundtrip-efficiency 90%
New schedule is stored.
```

We can see the updated scheduling in the [FlexMeasures UI](#) :



The [asset page for the battery](#) now shows the solar data, too.



5.3.10 Installation & First steps

Getting FlexMeasures to run

This section walks you through installing FlexMeasures on your own PC and running it continuously. We'll cover getting started by making a secret key, connecting a database and creating one user & one asset.

Note: Maybe these starting points are also interesting for you:

- For an example to see FlexMeasures in action with the least effort, see *Toy example: Scheduling a battery, from scratch*.
- You can run FlexMeasures via Docker, see docker and *Running a complete stack with docker-compose*.
- Are you not hosting FlexMeasures, but want to learn how to interact with it? Start with *Posting data*.

Install FlexMeasures

Install dependencies and the flexmeasures platform itself:

```
$ pip install flexmeasures
```

Note: With newer Python versions and Windows, some smaller dependencies (e.g. tables or rq-win) might cause issues as support is often slower. You might overcome this with a little research, by [installing from wheels](#) or [from the repo](#), respectively.

Make a secret key for sessions and password salts

Set a secret key which is used to sign user sessions and re-salt their passwords. The quickest way is with an environment variable, like this:

```
$ export SECRET_KEY=something-secret
```

(on Windows, use `set` instead of `export`)

This suffices for a quick start.

If you want to consistently use FlexMeasures, we recommend you add this setting to your config file at `~/flexmeasures.cfg` and use a truly random string. Here is a Pythonic way to generate a good secret key:

```
$ python -c "import secrets; print(secrets.token_urlsafe())"
```

Configure environment

Set an environment variable to indicate in which environment you are operating (one out of development|testing|staging|production). We'll go with development here:

```
$ export FLASK_ENV=development
```

(on Windows, use `set` instead of `export`)

or:

```
$ echo "FLASK_ENV=development" >> .env
```

Note: The default is `production`, which will not work well on localhost due to SSL issues.

Preparing the time series database

- Make sure you have a Postgres (Version 9+) database for FlexMeasures to use. See [Postgres database](#) (section “Getting ready to use”) for instructions on this.
- Tell flexmeasures about it:

```
$ export SQLALCHEMY_DATABASE_URI="postgresql://<user>:<password>@<host-  
↪address>[:<port>]/<db>"
```

If you install this on localhost, `host-address` is `127.0.0.1` and the port can be left out. (on Windows, use `set` instead of `export`)

- Create the Postgres DB structure for FlexMeasures:

```
$ flexmeasures db upgrade
```

This suffices for a quick start.

Note: For a more permanent configuration, you can create your FlexMeasures configuration file at `~/flexmeasures.cfg` and add this:

```
SQLALCHEMY_DATABASE_URI = "postgresql://<user>:<password>@<host-address>[:<port>]/<db>"
```

Adding data

Add an account & user

FlexMeasures is a tenant-based platform — multiple clients can enjoy its services on one server. Let's create a tenant account first:

```
$ flexmeasures add account --name "Some company"
```

This command will tell us the ID of this account. Let's assume it was 2.

FlexMeasures is also a web-based platform, so we need to create a user to authenticate:

```
$ flexmeasures add user --username <your-username> --email <your-email-address> --  
↪account-id 2 --roles=admin
```

- This will ask you to set a password for the user.
- Giving the first user the admin role is probably what you want.

Add structure

Populate the database with some standard asset types, user roles etc.:

```
$ flexmeasures add initial-structure
```

Add your first asset

There are three ways to add assets:

First, you can use the `flexmeasures CLI Commands`:

```
$ flexmeasures add asset --name "my basement battery pack" --asset-type-id 3 --latitude_  
↪65 --longitude 123.76 --account-id 2
```

For the asset type ID, I consult `flexmeasures show asset-types`.

For the account ID, I looked at the output of `flexmeasures add account` (the command we issued above) — I could also have consulted `flexmeasures show accounts`.

The second way to add an asset is the UI — head over to <https://localhost:5000/assets> (after you started FlexMeasures, see step “Run FlexMeasures” further down) and add a new asset there in a web form.

Finally, you can also use the `POST /api/v2_0/assets` endpoint in the FlexMeasures API to create an asset.

Add your first sensor

Usually, we are here because we want to measure something with respect to our assets. Each assets can have sensors for that, so let's add a power sensor to our new battery asset, using the flexmeasures *CLI Commands*:

```
$ flexmeasures add sensor --name power --unit MW --event-resolution 5 --timezone Europe/
↪Amsterdam --asset-id 1 --attributes '{"capacity_in_mw": 7}'
```

The asset ID I got from the last CLI command, or I could consult `flexmeasures show account --account-id <my-account-id>`.

Add time series data (beliefs)

There are three ways to add data:

First, you can load in data from a file (CSV or Excel) via the flexmeasures *CLI Commands*:

```
$ flexmeasures add beliefs --file my-data.csv --skiprows 2 --delimiter ";" --source_
↪OurLegacyDatabase --sensor-id 1
```

This assumes you have a file `my-data.csv` with measurements, which was exported from some legacy database, and that the data is about our sensor with ID 1. This command has many options, so do use its `--help` function.

Second, you can use the `POST /api/v3_0/sensors/data` endpoint in the FlexMeasures API to send meter data.

Finally, you can tell FlexMeasures to create forecasts for your meter data with the `flexmeasures add forecasts` command, here is an example:

```
$ flexmeasures add forecasts --from-date 2020-03-08 --to-date 2020-04-08 --asset-type_
↪Asset --asset my-solar-panel
```

Note: You can also use the API to send forecast data.

Run FlexMeasures

Running the web service

It's finally time to start running FlexMeasures:

```
$ flexmeasures run
```

(This might print some warnings, see the next section where we go into more detail)

Note: In a production context, you shouldn't run a script - hand the app object to a WSGI process, as your platform of choice describes. Often, that requires a WSGI script. We provide an example WSGI script in *Continuous integration*. You can also take a look at FlexMeasures' Dockerfile to get an idea how to run FlexMeasures with gunicorn.

You can visit `http://localhost:5000` now to see if the app's UI works. When you see the dashboard, the map will not work. For that, you'll need to get your `MAPBOX_ACCESS_TOKEN` and add it to your config file.

Other settings, for full functionality

Set mail settings

For FlexMeasures to be able to send email to users (e.g. for resetting passwords), you need an email account which can do that (e.g. GMail). Set the `MAIL_*` settings in your configuration, see [Mail](#).

Install an LP solver

For planning balancing actions, the FlexMeasures platform uses a linear program solver. Currently that is the Cbc solver. See [FLEXMEASURES_LP_SOLVER](#) if you want to change to a different solver.

Installing Cbc can be done on Unix via:

```
$ apt-get install coinor-cbc
```

(also available in different popular package managers).

We provide a script for installing from source (without requiring `sudo` rights) in the `ci` folder.

More information (e.g. for installing on Windows) on [the Cbc website](#).

Install and configure Redis

To let FlexMeasures queue forecasting and scheduling jobs, install a [Redis](#) server (or rent one) and configure access to it within FlexMeasures' config file (see above). You can find the necessary settings in [Redis](#).

Then, start workers in a console (or some other method to keep a long-running process going):

```
$ flexmeasures jobs run-worker --queue forecasting
$ flexmeasures jobs run-worker --queue scheduling
```

Where to go from here?

If your data structure is good, you should think about (continually) adding measurement data. This tutorial mentioned how to add data, but [Posting data](#) goes deeper with examples and terms & definitions.

Then, you probably want to use FlexMeasures to generate forecasts and schedules! For this, read further in [Forecasting & scheduling](#).

5.3.11 Posting data

The platform FlexMeasures strives on the data you feed it. Let's demonstrate how you can get data into FlexMeasures using the API. This is where FlexMeasures gets connected to your system as a smart backend and helps you build smart energy services.

We will show how to use the API endpoints for POSTing data. You can call these at regular intervals (through scheduled scripts in your system, for example), so that FlexMeasures always has recent data to work with. Of course, these endpoints can also be used to load historic data into FlexMeasures, so that the forecasting models have access to enough data history.

Note: For the purposes of forecasting and scheduling, it is often advisable to use a less fine-grained resolution than most metering services keep. For example, while such services might measure every ten seconds, FlexMeasures will usually do its job no less effective if you feed it data with a resolution of five minutes. This will also make the data integration much easier. Keep in mind that many data sources like weather forecasting or markets can have data resolutions of an hour, anyway.

Table of contents

- [*Prerequisites*](#)
- [*Posting sensor data*](#)
- [*Posting power data*](#)
- [*Observations vs forecasts*](#)
- [*Posting flexibility states*](#)

Prerequisites

- FlexMeasures needs some structural meta data for data to be understood. For example, for adding weather data we need to define a weather sensor, and what kind of weather sensors there are. You also need a user account. If you host FlexMeasures yourself, you need to add this info first. Head over to [*Getting started*](#), where these steps are covered, study our [*CLI Commands*](#) or look into plugins which do this like [*flexmeasures-entsoe*](#) or [*flexmeasures-openweathermap*](#).
- You should be familiar with where to find your API endpoints (see [*Main endpoint and API versions*](#)) and how to authenticate against the API (see [*Authentication*](#)).

Note: For deeper explanations of the data and the meta fields we'll send here, You can always read the [*API Introduction*](#), to the FlexMeasures API, e.g. [*Signs of power values*](#), [*Frequency and resolution*](#), [*Setting the recording time*](#) and [*Units*](#).

Note: To address assets and sensors, these tutorials assume entity addresses valid in the namespace `fm1`. See [*API Introduction*](#) for more explanations.

Posting sensor data

Sensor data (both observations and forecasts) can be posted to `POST /sensors/data`. This endpoint represents the basic method of getting time series data into FlexMeasures via API. It is agnostic to the type of sensor and can be used to POST data for both physical and economical events that have happened in the past or will happen in the future. Some examples:

- readings from electricity and gas meters
- readings from temperature and pressure sensors
- state of charge of a battery
- estimated availability of parking spots
- price forecasts

The exact URL will depend on your domain name, and will look approximately like this:

```
[POST] https://company.flexmeasures.io/api/<version>/sensors/data
```

This example “PostSensorDataRequest” message posts prices for hourly intervals between midnight and midnight the next day for the Korean Power Exchange (KPX) day-ahead auction, registered under sensor 16. The `prior` indicates that the prices were published at 3pm on December 31st 2014 (i.e. the clearing time of the KPX day-ahead market, which is at 3 PM on the previous day — see below for a deeper explanation).

```
{
  "type": "PostSensorDataRequest",
  "sensor": "ea1.2021-01.io.flexmeasures.company:fm1.16",
  "values": [
    52.37,
    51.14,
    49.09,
    48.35,
    48.47,
    49.98,
    58.7,
    67.76,
    69.21,
    70.26,
    70.46,
    70,
    70.7,
    70.41,
    70,
    64.53,
    65.92,
    69.72,
    70.51,
    75.49,
    70.35,
    70.01,
    66.98,
    58.61
  ],
  "start": "2015-01-01T00:00:00+09:00",
  "duration": "PT24H",
  "prior": "2014-12-31T15:00:00+09:00",
  "unit": "KRW/kWh"
}
```

Note how the resolution of the data comes out at 60 minutes when you divide the duration by the number of data points. If this resolution does not match the sensor’s resolution, FlexMeasures will try to upsample the data to make the match or, if that is not possible, complain. Likewise, if the data unit does not match the sensor’s unit, FlexMeasures will attempt to convert the data or, if that is not possible, complain.

Posting power data

For power data, USEF specifies separate message types for observations and forecasts. Correspondingly, we allow the following message types to be used with the [POST] /sensors/data endpoint (see *Posting sensor data*):

```
{
  "type": "PostMeterDataRequest"
}
```

```
{
  "type": "PostPrognosisRequest"
}
```

For these message types, FlexMeasures validates whether the data unit is suitable for communicating power data. Additionally, we validate whether meter data lies in the past, and prognoses lie in the future.

Single value, single sensor

A single average power value for a 15-minute time interval for a single sensor, posted 5 minutes after realisation.

```
{
  "type": "PostSensorDataRequest",
  "sensor": "ea1.2021-01.io.flexmeasures.company:fm1.1",
  "value": 220,
  "start": "2015-01-01T00:00:00+00:00",
  "duration": "PT0H15M",
  "horizon": "-PT5M",
  "unit": "MW"
}
```

Multiple values, single sensor

Multiple values (indicating a univariate timeseries) for 15-minute time intervals for a single sensor, posted 5 minutes after each realisation.

```
{
  "type": "PostSensorDataRequest",
  "sensor": "ea1.2021-01.io.flexmeasures.company:fm1.1",
  "values": [
    220,
    210,
    200
  ],
  "start": "2015-01-01T00:00:00+00:00",
  "duration": "PT0H45M",
  "horizon": "-PT5M",
  "unit": "MW"
}
```

Observations vs forecasts

To correctly tell FlexMeasures when a meter reading or forecast was known is crucial, as it determines which data is being used to compute schedules or to make other forecasts.

Usually, the time of posting is assumed to be the time when the data was known. But you can also explicitly tell FlexMeasures what these times are. This either works with one fixed time (for the whole set of data being sent) or with a horizon (which applies to each data point separately).

E.g. to post a forecast rather than an observation after the fact, simply set the `prior` to the moment at which the forecasts were made, e.g. at “2015-01-01T16:30:00+09:00”. Assuming your data starts at 5.00pm, this denotes that the data are forecasts, made half an hour before realisation.

Alternatively, to indicate that each individual observation was made directly after the end of its 15-minute interval (i.e. at 3.15pm, 3.30pm and so on), set a `horizon` to “PT0H” instead of a `prior`.

Finally, delays in reading out sensor data can be simulated by setting the `horizon` field to a negative value. For example, a horizon of “-PT1H” would denote that each temperature reading was observed one hour after the fact (i.e. at 4.15pm, 4.30pm and so on).

See *Setting the recording time* for more information regarding the `prior` and `horizon` fields.

A good example for the use of the `prior` field are markets, which have clearing times. For example, at the KPX day-ahead auction this is every day at 3pm. This point in time (i.e. when contracts are signed) determines the difference between an ex-post observation and an ex-ante forecast.

Another example for the `prior` field is running simulations with FlexMeasures. It gives you control over the timing so that you could run a month in the past as if it happened right now.

Posting flexibility states

There is one more crucial kind of data that FlexMeasures needs to know about: What are the current states of flexible devices? For example, a battery has a certain state of charge, which is relevant to describe the flexibility that the battery currently has. In our terminology, this is called the “flex model” and you can read more at *Describing flexibility*.

Owners of such devices can post the flex model along with triggering the creation of a new schedule, to `[POST] /schedules/trigger`. The URL might look like this:

```
https://company.flexmeasures.io/api/<version>/sensors/10/schedules/trigger
```

The following example triggers a schedule for a power sensor (with ID 10) of a battery asset, asking to take into account the battery’s current state of charge. From this, FlexMeasures derives the energy flexibility this battery has in the next 48 hours and computes an optimal charging schedule. The endpoint also allows to limit the flexibility range and also to set target values.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh"
  }
}
```

Note: At the moment, FlexMeasures only supports flexibility models suitable for batteries and car chargers here (asset types “battery”, “one-way_evse” or “two-way_evse”). This will be expanded to other flexible assets as needed.

Note: Flexibility states are persisted on sensor attributes. To record a more complete history of the state of charge, set up a separate sensor and post data to it using [POST] `/sensors/data` (see *Posting sensor data*).

In *How scheduling jobs are queued*, we'll cover what happens when FlexMeasures is triggered to create a new schedule, and how those schedules can be retrieved via the API, so they can be used to steer assets.

5.3.12 Forecasting & scheduling

Once FlexMeasures contains data (see *Posting data*), you can enjoy its forecasting and scheduling services. Let's take a look at how FlexMeasures users can access information from these services, and how you (if you are hosting FlexMeasures yourself) can set up the data science queues for this.

Table of contents

- *Maintaining the queues*
- *How forecasting jobs are queued*
- *How scheduling jobs are queued*
- *Getting power forecasts (prognoses)*
- *Getting schedules (control signals)*

If you want to learn more about the actual algorithms used in the background, head over to *Algorithms*.

Note: FlexMeasures comes with in-built scheduling algorithms. You can use your own algorithm, as well, see plugin-customization.

Maintaining the queues

Note: If you are not hosting FlexMeasures yourself, skip right ahead to *How forecasting jobs are queued* or *Getting power forecasts (prognoses)*.

Here we assume you have access to a Redis server and configured it (see *Redis*).

Start to run one worker for each kind of job (in a separate terminal):

```
$ flexmeasures jobs run-worker --queue forecasting
$ flexmeasures jobs run-worker --queue scheduling
```

You can also clear the job queues:

```
$ flexmeasures jobs clear-queue --queue forecasting
$ flexmeasures jobs clear-queue --queue scheduling
```

When the main FlexMeasures process runs (e.g. by `flexmeasures run`), the queues of forecasting and scheduling jobs can be visited at `http://localhost:5000/tasks/forecasting` and `http://localhost:5000/tasks/schedules`, respectively (by admins).

When forecasts and schedules have been generated, they should be visible at `http://localhost:5000/analytics`.

Note: You can run workers who process jobs on different computers than the main server process. This can be a great architectural choice. Just keep in mind to use the same databases (postgres/redis) and to stick to the same FlexMeasures version on both.

How forecasting jobs are queued

A forecasting job is an order to create forecasts based on measurements. A job can be about forecasting one point in time or about forecasting a range of points.

In FlexMeasures, the usual way of creating forecasting jobs would be right in the moment when new power, weather or price data arrives through the API (see [Posting data](#)). So technically, you don't have to do anything to keep fresh forecasts.

The decision which horizons to forecast is currently also taken by FlexMeasures. For power data, FlexMeasures makes this decision depending on the asset resolution. For instance, a resolution of 15 minutes leads to forecast horizons of 1, 6, 24 and 48 hours. For price data, FlexMeasures chooses to forecast prices forward 24 and 48 hours. These are decent defaults, and fixing them has the advantage that schedulers (see below) will know what to expect. However, horizons will probably become more configurable in the near future of FlexMeasures.

You can also add forecasting jobs directly via the CLI. We explain this practice in the next section.

Historical forecasts

There might be reasons to add forecasts of past time ranges. For instance, for visualisation of past system behaviour and to check how well the forecasting models have been doing on a longer stretch of data.

If you host FlexMeasures yourself, we provide a CLI task for adding forecasts for whole historic periods. This is an example call:

Here we request 6-hour forecasts to be made for two sensors, for a period of two days:

```
$ flexmeasures add forecasts --sensor-id 2 --sensor-id 3 \  
  --from-date 2015-02-01 --to-date 2015-08-31 \  
  --horizon 6 --as-job
```

This is half a year of data, so it will take a while.

It can be good advice to dispatch this work in smaller chunks. Alternatively, note the `--as-job` parameter. If you use it, the forecasting jobs will be queued and picked up by worker processes (see above). You could run several workers (e.g. one per CPU) to get this work load done faster.

Run `flexmeasures add forecasts --help` for more information.

How scheduling jobs are queued

In FlexMeasures, a scheduling job is an order to plan optimised actions for flexible devices. It usually involves a linear program that combines a state of energy flexibility with forecasted data to draw up a consumption or production plan ahead of time.

There are two ways to queue a scheduling job:

First, we can add a scheduling job to the queue via the API. We already learned about the `[POST] /schedules/trigger` endpoint in *Posting flexibility states*, where we saw how to post a flexibility state (in this case, the state of charge of a battery at a certain point in time).

Here, we extend that (storage) example with an additional target value, representing a desired future state of charge.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh"
    "soc-targets": [
      {
        "value": 25,
        "datetime": "2015-06-02T16:00:00+00:00"
      }
    ]
  }
}
```

We now have described the state of charge at 10am to be 12.1. In addition, we requested that it should be 25 at 4pm. For instance, this could mean that a car should be charged at 90% at that time.

If FlexMeasures receives this message, a scheduling job will be made and put into the queue. In turn, the scheduling job creates a proposed schedule. We'll look a bit deeper into those further down in *Getting schedules (control signals)*.

Note: Even without a target state of charge, FlexMeasures will create a scheduling job. The flexible device can then be used with more freedom to reach the system objective (e.g. buy power when it is cheap, store it, and sell back when it's expensive).

A second way to add scheduling jobs is via the CLI, so this is available for people who host FlexMeasures themselves:

```
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --start 2022-07-05T07:00+01:00 --duration PT12H \
  --soc-at-start 50% --roundtrip-efficiency 90% --as-job
```

Here, the `--as-job` parameter makes the difference for queueing — without it, the schedule is computed right away.

Run `flexmeasures add schedule for-storage --help` for more information.

Getting power forecasts (prognoses)

Prognoses (the USEF term used for power forecasts) are used by FlexMeasures to determine the best control signals to valorise on balancing opportunities.

You can access forecasts via the FlexMeasures API at [\[GET\] /sensors/data](#). Getting them might be useful if you want to use prognoses in your own system, or to check their accuracy against meter data, i.e. the realised power measurements. The FlexMeasures UI also lists forecast accuracy, and visualises prognoses and meter data next to each other.

A prognosis can be requested at a URL looking like this:

```
https://company.flexmeasures.io/api/<version>/sensors/data
```

This example requests a prognosis for 24 hours, with a rolling horizon of 6 hours before realisation.

```
{
  "type": "GetPrognosisRequest",
  "sensor": "ea1.2021-01.io.flexmeasures.company:fm1.1",
  "start": "2015-01-01T00:00:00+00:00",
  "duration": "PT24H",
  "horizon": "PT6H",
  "resolution": "PT15M",
  "unit": "MW"
}
```

Getting schedules (control signals)

We saw above how FlexMeasures can create optimised schedules with control signals for flexible devices (see *Posting flexibility states*). You can access the schedules via the [\[GET\] /schedules/<uuid>](#) endpoint. The URL then looks like this:

```
https://company.flexmeasures.io/api/<version>/sensors/<id>/schedules/<uuid>
```

Here, the schedule's Universally Unique Identifier (UUID) should be filled in that is returned in the [\[POST\] /schedules/trigger](#) response. Schedules can be queried by their UUID for up to 1 week after they were triggered (ask your host if you need to keep them around longer). Afterwards, the exact schedule can still be retrieved through the [\[GET\] /sensors/data](#), using precise filter values for `start`, `prior` and `source`.

The following example response indicates that FlexMeasures planned ahead 45 minutes for the requested battery power sensor. The list of consecutive power values represents the target consumption of the battery (negative values for production). Each value represents the average power over a 15 minute time interval.

```
{
  "values": [
    2.15,
    3,
    2
  ],
  "start": "2015-06-02T10:00:00+00:00",
  "duration": "PT45M",
  "unit": "MW"
}
```

How to interpret these control signals?

One way of reaching the target consumption in this example is to let the battery start to consume with 2.15 MW at 10am, increase its consumption to 3 MW at 10.15am and decrease its consumption to 2 MW at 10.30am.

However, because the targets values represent averages over 15-minute time intervals, the battery still has some degrees of freedom. For example, the battery might start to consume with 2.1 MW at 10.00am and increase its consumption to 2.25 at 10.10am, increase its consumption to 5 MW at 10.15am and decrease its consumption to 2 MW at 10.20am. That should result in the same average values for each quarter-hour.

5.3.13 Building custom UIs

FlexMeasures provides its own UI (see [Dashboard](#)), but it is a back office platform first. Most energy service companies already have their own user-facing system. We therefore made it possible to incorporate information from FlexMeasures in custom UIs.

This tutorial will show how the FlexMeasures API can be used from JavaScript to extract information and display it in a browser (using HTML). We'll extract information about users, assets and even whole plots!

Table of contents

- [Get an authentication token](#)
- [Load user information](#)
- [Load asset information](#)
- [Embedding charts](#)

Note: We'll use standard JavaScript for this tutorial, in particular the [fetch](#) functionality, which many browsers support out-of-the-box these days. You might want to use more high-level frameworks like jQuery, Angular, React or VueJS for your frontend, of course.

Get an authentication token

FlexMeasures provides the [POST] `/api/requestAuthToken` endpoint, as discussed in [Authentication](#). Here is a JavaScript function to call it:

```
var flexmeasures_domain = "http://localhost:5000";

function getAuthToken(){
    return fetch(flexmeasures_domain + '/api/requestAuthToken',
        {
            method: "POST",
            mode: "cors",
            headers:
            {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({"email": email, "password": password})
        }
    )
    .then(function(response) { return response.json(); })
}
```

(continues on next page)

(continued from previous page)

```

    .then(console.log("Got auth token from FlexMeasures server ..."));
}

```

It only expects you to set email and password somewhere (you could also pass them to the function, your call). In addition, we expect here that `flexmeasures_domain` is set to the FlexMeasures server you interact with, for example `"https://company.flexmeasures.io"`.

We'll see how to make use of the `getAuthToken` function right away, keep on reading.

Load user information

Let's say we are interested in a particular user's meta data. For instance, which email address do they have and which timezone are they operating in?

Given we have set a variable called `userId`, here is some code to find out and display that information in a simple HTML table:

```

<h1>User info</h1>
<p>
  Email address: <span id="user_email"></span>
</p>
<p>
  Time zone: <span id="user_timezone"></span>
</p>

```

```

function loadUserInfo(userId, authToken) {
  fetch(flexmeasures_domain + '/api/v2_0/user/' + userId,
    {
      method: "GET",
      mode: "cors",
      headers:
        {
          "Content-Type": "application/json",
          "Authorization": authToken
        },
    }
  )
  .then(console.log("Got user data from FlexMeasures server ..."))
  .then(function(response) { return response.json(); })
  .then(function(userInfo) {
    document.querySelector('#user_email').innerHTML = userInfo.email;
    document.querySelector('#user_timezone').innerHTML = userInfo.timezone;
  })
}

document.onreadystatechange = () => {
  if (document.readyState === 'complete') {
    getAuthToken()
    .then(function(response) {
      var authToken = response.auth_token;
      loadUserInfo(userId, authToken);
    })
  }
}

```

(continues on next page)

(continued from previous page)

```
}
}
```

The result looks like this in your browser:

User info

Email address: demo@seita.nl

Time zone: Europe/Amsterdam

From FlexMeasures, we are using the [\[GET\] /user](#) endpoint, which loads information about one user. Browse its documentation to learn about other information you could get.

Load asset information

Similarly, we can load asset information. Say we have a variable `accountId` and we want to show which assets FlexMeasures administrates for that account.

For the example below, we've used the Id of the account from our toy tutorial, see [toy tutorial](#).

```
<style>
  #assetTable th, #assetTable td {
    border-right: 1px solid gray;
    padding-left: 5px;
    padding-right: 5px;
  }
</style>
```

```
<table id="assetTable">
  <thead>
    <tr>
      <th>Asset name</th>
      <th>Id</th>
      <th>Latitude</th>
      <th>Longitude</th>
    </tr>
  </thead>
  <tbody></tbody>
</table>
```

```
function loadAssets(accountId, authToken) {
  var params = new URLSearchParams();
  params.append("account_id", accountId);
  fetch(flexmeasures_domain + '/api/v3_0/assets?' + params.toString(),
    {
      method: "GET",
```

(continues on next page)

(continued from previous page)

```

        mode: "cors",
        headers:
            {
                "Content-Type": "application/json",
                "Authorization": authToken
            },
    }
)
.then(console.log("Got asset data from FlexMeasures server ..."))
.then(function(response) { return response.json(); })
.then(function(rows) {
    rows.forEach(row => {
        const tbody = document.querySelector('#assetTable tbody');
        const tr = document.createElement('tr');
        tr.innerHTML = `<td>${row.name}</td><td>${row.id}</td><td>${row.latitude}</td>
↪<td>${row.longitude}</td>`;
        tbody.appendChild(tr);
    });
})
}

document.onreadystatechange = () => {
    if (document.readyState === 'complete') {
        getAuthToken()
        .then(function(response) {
            var authToken = response.auth_token;
            loadAssets(accountId, authToken);
        })
    }
}

```

The result looks like this in your browser:

Asset name	Id	Latitude	Longitude
toy-solar	11	52.374	4.88969
toy-building	12	52.374	4.88969
toy-battery	13	52.374	4.88969

From FlexMeasures, we are using the [\[GET\] /assets](#) endpoint, which loads a list of assets. Note how, unlike the user endpoint above, we are passing a query parameter to the API (`account_id`). We are only displaying a subset of the information which is available about assets. Browse the endpoint documentation to learn other information you could get.

For a listing of public assets, replace `/api/v3_0/assets` with `/api/v3_0/assets/public`.

Embedding charts

Creating charts from data can consume lots of development time. FlexMeasures can help here by delivering ready-made charts. In this tutorial, we'll embed a chart with electricity prices.

First, we define a div tag for the chart and a basic layout (full width). We also load the visualization libraries we need (more about that below), and set up a custom formatter we use in FlexMeasures charts.

```
<script src="https://d3js.org/d3.v6.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/vega@5.22.1"></script>
<script src="https://cdn.jsdelivr.net/npm/vega-lite@5.2.0"></script>
<script src="https://cdn.jsdelivr.net/npm/vega-embed@6.20.8"></script>
<script>
    vega.expressionFunction('quantityWithUnitFormat', function(datum, params) {
        return d3.format(params[0])(datum) + " " + params[1];
    });
</script>

<div id="sensor-chart" style="width: 100%;"></div>
```

Now we define a JavaScript function to ask the FlexMeasures API for a chart and then embed it:

```
function embedChart(params, authToken, sensorId, divId){
    fetch(
        flexmeasures_domain + '/api/dev/sensor/' + sensorId + '/chart?include_data=true&
        ↪ ' + params.toString(),
        {
            method: "GET",
            mode: "cors",
            headers:
                {
                    "Content-Type": "application/json",
                    "Authorization": authToken
                }
        }
    )
    .then(function(response) {return response.json();})
    .then(function(data) {vegaEmbed(divId, data)})
}
```

This function allows us to request a chart (actually, a JSON specification of a chart that can be interpreted by vega-lite), and then embed it within a div tag of our choice.

From FlexMeasures, we are using the `GET /api/dev/sensor/(id)/chart/` endpoint. Browse the endpoint documentation to learn more about it.

Note: Endpoints in the developer API are still under development and are subject to change in new releases.

Here are some common parameter choices for our JavaScript function:

```
var params = new URLSearchParams();
params.append("width", 400); // an integer number of pixels; without it, the chart will
↪ be scaled to the full width of the container (note that we set the div width to 100%)
```

(continues on next page)

(continued from previous page)

```

params.append("height", 400); // an integer number of pixels; without it, a FlexMeasures_
↪ default is used
params.append("event_starts_after", '2022-10-01T00:00+01'); // only fetch events from_
↪ midnight October 1st
params.append("event_ends_before", '2022-10-08T00:00+01'); // only fetch events until_
↪ midnight October 8th
params.append("beliefs_before", '2022-10-03T00:00+01'); // only fetch beliefs prior to_
↪ October 3rd (time travel)

```

As FlexMeasures uses the [Vega-Lite Grammar of Interactive Graphics](#) internally, we also need to import this library to render the chart (see the `script` tags above). It's crucial to note that FlexMeasures is not transferring images across HTTP here, just information needed to render them.

Note: It's best to match the visualization library versions you use in your frontend to those used by FlexMeasures. These are set by the `FLEXMEASURES_JS_VERSIONS` config (see [Configuration](#)) with defaults kept in `flexmeasures/utils/config_defaults`.

Now let's call this function when the HTML page is opened, to embed our chart:

```

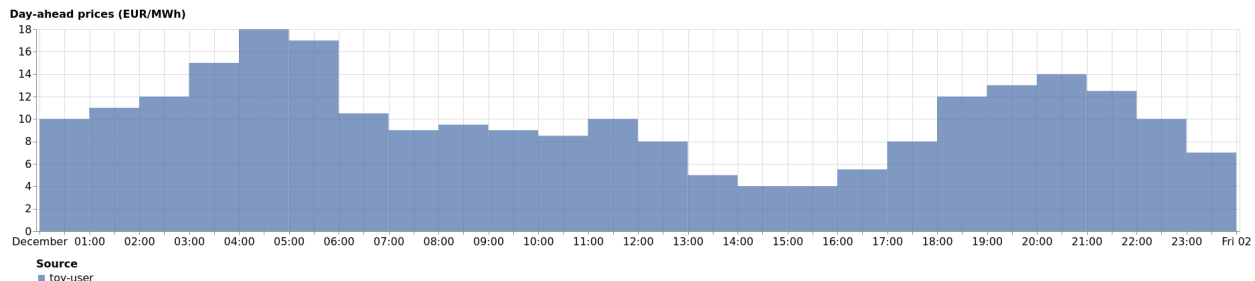
document.onreadystatechange = () => {
  if (document.readyState === 'complete') {
    getAuthToken()
      .then(function(response) {
        var authToken = response.auth_token;

        var params = new URLSearchParams();
        params.append("event_starts_after", '2022-01-01T00:00+01');
        embedChart(params, authToken, 3, '#sensor-chart');
      })
  }
}

```

The parameters we pass in describe what we want to see: all data for sensor 3 since 2022. If you followed our [toy tutorial](#) on a fresh FlexMeasures installation, sensor 3 contains market prices (authenticate with the toy-user to gain access).

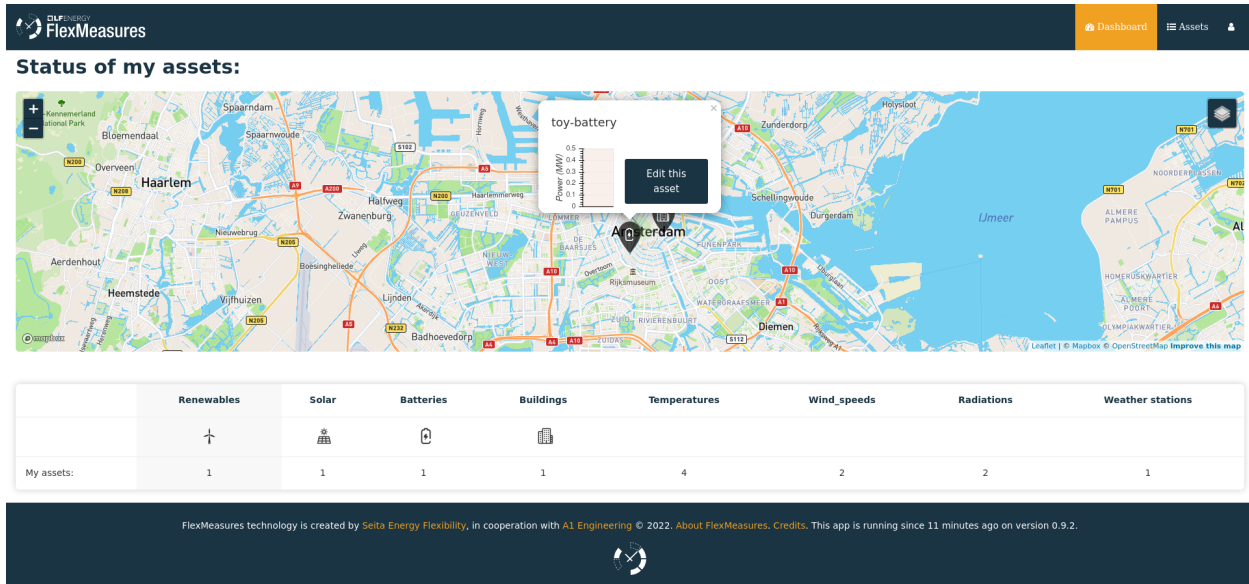
The result looks like this in your browser:



5.3.14 Dashboard

The dashboard shows where the user's assets are located and how many different asset types are connected to the platform. The view serves to quickly identify the status of assets, such as whether there are upcoming opportunities to valorise on flexibility activations. In particular, the page contains:

- *Interactive map of assets*
- *Summary of asset types*
- *Grouping by accounts*



Interactive map of assets

The map shows all of the user's assets with icons for each asset type. Clicking on an asset allows the user to see its current state (e.g. latest measurement of wind power production) and to navigate to the analytics page to see more details, for instance forecasts.

Summary of asset types

The summary below the map lists all asset types that the user has hooked up to the platform and how many of each there are. Clicking on the asset type name leads to the analytics page, where data is shown aggregated for that asset type.

Grouping by accounts

Note: This is a feature for user with role `admin` or `admin-reader`.

By default, the map is layered by asset type. However, on the bottom right admins can also switch to grouping by accounts. Then, map layers will contain the assets owned by accounts, and you can easily see who you're serving with what.

5.3.15 Assets & data

The asset page allows to see data from the asset's sensors, and also to edit attributes of the asset, like its location. Other attributes are stored as a JSON string, which can be edited here as well. This is meant for meta information that may be used to customize views or functionality, e.g. by plugins. This includes the possibility to specify which sensors the asset page should show. For instance, here we include a price sensor from a public asset, by setting `{"sensor_to_show": [3, 2]}` (sensor 3 on top, followed by sensor 2 below).



Note: It is possible to overlay data for multiple sensors, by setting the `sensors_to_show` attribute to a nested list. For example, `{"sensor_to_show": [3, [2, 4]]}` would show the data for sensor 4 laid over the data for sensor 2.

Note: While it is possible to show an arbitrary number of sensors this way, we recommend showing only the most crucial ones for faster loading, less page scrolling, and generally, a quick grasp of what the asset is up to.

Note: Asset attributes can be edited through the CLI as well, with the CLI command `flexmeasures edit attribute`.

5.3.16 Administration

The administrator can see assets and users here.

Assets

Listing all assets:




FlexMeasures

DashboardAssetsUsers?

Asset overview

Show 10 records

Filter records:

Name	Location	Asset id	Account	Sensors
 toy-solar	LAT: 52.3740 LONG: 4.8897	1	Docker Toy Account	0
 toy-building	LAT: 52.3740 LONG: 4.8897	2	Docker Toy Account	0
 toy-battery	LAT: 52.3740 LONG: 4.8897	3	Docker Toy Account	1

Showing 1 to 3 out of 3 records

FlexMeasures technology is created by [Seita Energy Flexibility](#), in cooperation with [A1 Engineering](#) © 2022. [About FlexMeasures](#). [Credits](#). This app is running since 18 minutes ago on version 0.11.0.dev21.

Users

Listing all users:

FlexMeasures

DashboardAssetsTasks?

All active users

Show 10 records

Filter records:

☐ Include inactive

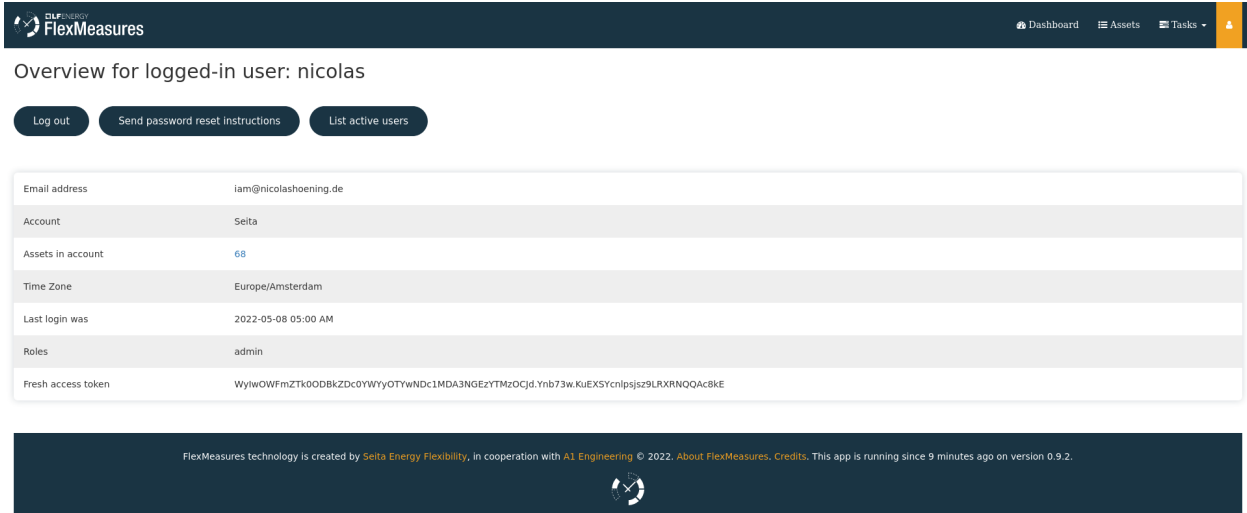
Username	Email	Roles	Account	Timezone	Last Login	Active
nicolas	iam@nicolashoening.de	admin	Seita	Europe/Amsterdam	an hour ago	True
ki_yeol	shinky@ynu.ac.kr	admin	A1	Asia/Seoul	Oct 06 2020	True
Summer	summer@seita.nl		Some company	UTC	Jan 13	True
toy-user	toy-user@flexmeasures.io	account-admin	Toy Account	Europe/Amsterdam	Mar 30	True
mohammudullah	mohammudullah@seita.nl	admin-reader	Toy Account	Europe/Amsterdam	Apr 23	True

Showing 11 to 15 out of 15 records

Previous12Next

FlexMeasures technology is created by [Seita Energy Flexibility](#), in cooperation with [A1 Engineering](#) © 2022. [About FlexMeasures](#). [Credits](#). This app is running since 10 minutes ago on version 0.9.2.

Viewing one user:



The screenshot shows the FlexMeasures web interface. At the top, there's a navigation bar with 'FlexMeasures' logo and links for 'Dashboard', 'Assets', and 'Tasks'. Below the navigation bar, it says 'Overview for logged-in user: nicolas'. There are three buttons: 'Log out', 'Send password reset instructions', and 'List active users'. Below these buttons is a table with user details:

Email address	iam@nicolashoenig.de
Account	Seita
Assets in account	68
Time Zone	Europe/Amsterdam
Last login was	2022-05-08 05:00 AM
Roles	admin
Fresh access token	WylwOWFmZTk0ODBkZDc0YWYyOTYwNDc1MDA3NGEzYTMzOCJd.Ynb73w.KuEXSYcnlpsjsz9LXRNRQQAcbkE

At the bottom of the interface, there's a footer with the text: 'FlexMeasures technology is created by Seita Energy Flexibility, in cooperation with A1 Engineering. © 2022. About FlexMeasures. Credits. This app is running since 9 minutes ago on version 0.9.2.' There is also a small circular logo with a refresh icon.

5.3.17 API Introduction

This document details the Application Programming Interface (API) of the FlexMeasures web service. The API supports user automation for flexibility valorisation in the energy sector, both in a live setting and for the purpose of simulating scenarios. The web service adheres to the concepts and terminology used in the Universal Smart Energy Framework (USEF).

All requests and responses to and from the web service should be valid JSON messages. For deeper explanations on how to construct messages, see *Notation*.

Main endpoint and API versions

All versions of the API are released on:

```
https://<flexmeasures-root-url>/api
```

So if you are running FlexMeasures on your computer, it would be:

```
https://localhost:5000/api
```

Let's assume we are running a server for a client at:

```
https://company.flexmeasures.io/api
```

where *company* is a client of ours. All their accounts' data lives on that server.

We assume in this document that the FlexMeasures instance you want to connect to is hosted at <https://company.flexmeasures.io>.

Let's see what the `/api` endpoint returns:

```
>>> import requests
>>> res = requests.get("https://company.flexmeasures.io/api")
```

(continues on next page)

(continued from previous page)

```
>>> res.json()
{'flexmeasures_version': '0.9.0',
 'message': 'For these API versions endpoints are available. An authentication token can
↳be requested at: /api/requestAuthToken. For a list of services, see https://
↳flexmeasures.readthedocs.io',
 'status': 200,
 'versions': ['v3_0']}
}
```

So this tells us which API versions exist. For instance, we know that the latest API version is available at:

```
https://company.flexmeasures.io/api/v3_0
```

Also, we can see that a list of endpoints is available on <https://flexmeasures.readthedocs.io> for each of these versions.

Note: Sunset API versions are still documented there, simply select an older version.

Authentication

Service usage is only possible with a user access token specified in the request header, for example:

```
{
  "Authorization": "<token>"
}
```

A fresh “<token>” can be generated on the user’s profile after logging in:

```
https://company.flexmeasures.io/logged-in-user
```

or through a POST request to the following endpoint:

```
https://company.flexmeasures.io/api/requestAuthToken
```

using the following JSON message for the POST request data:

```
{
  "email": "<user email>",
  "password": "<user password>"
}
```

which gives a response like this if the credentials are correct:

```
{
  "auth_token": "<authentication token>",
  "user_id": "<ID of the user>"
}
```

Note: Each access token has a limited lifetime, see [Authentication](#).

Deprecation and sunset

When an API feature becomes obsolete, we deprecate it. Deprecation of major features doesn't happen a lot, but when it does, it happens in multiple stages, during which we support clients and hosts in adapting. For more information on our multi-stage deprecation approach and available options for FlexMeasures hosts, see *Deprecation and sunset for hosts*.

Clients

Professional API users should monitor API responses for the "Deprecation" and "Sunset" response headers [see [draft-ietf-httpapi-deprecation-header-02](#) and [RFC 8594](#), respectively], so system administrators can be warned when using API endpoints that are flagged for deprecation and/or are likely to become unresponsive in the future.

The deprecation header field shows an [IMF-fixdate](#) indicating when the API endpoint was deprecated. The sunset header field shows an [IMF-fixdate](#) indicating when the API endpoint is likely to become unresponsive.

More information about a deprecation, sunset, and possibly recommended replacements, can be found under the "Link" response header. Relevant relations are:

- "deprecation"
- "successor-version"
- "latest-version"
- "alternate"
- "sunset"

Here is a client-side code example in Python (this merely prints out the deprecation header, sunset header and relevant links, and should be revised to make use of the client's monitoring tools):

```
def check_deprecation_and_sunset(self, url, response):
    """Print deprecation and sunset headers, along with info links.

    Reference
    -----
    https://flexmeasures.readthedocs.io/en/latest/api/introduction.html#deprecation-and-
    ↪ sunset
    """
    # Go through the response headers in their given order
    for header, content in response.headers:
        if header == "Deprecation":
            print(f"Your request to {url} returned a deprecation warning. Deprecation:
            ↪ {content}")
        elif header == "Sunset":
            print(f"Your request to {url} returned a sunset warning. Sunset: {content}")
        elif header == "Link" and ('rel="deprecation";' in content or 'rel="sunset";' in
            ↪ content):
            print(f"Further info is available: {content}")
```

Hosts

FlexMeasures versions go through the following stages for deprecating major features (such as API versions):

- *Stage 1: Deprecation*: status 200 (OK) with *relevant headers*, plus a toggle to 410 (Gone) for blackout tests
- *Stage 2: Preliminary sunset*: status 410 (Gone), plus a toggle to 200 (OK) for sunset rollbacks
- *Stage 3: Definitive sunset*: status 410 (Gone)

Let's go over these stages in more detail.

Stage 1: Deprecation

When upgrading to a FlexMeasures version that deprecates an API version (e.g. `flexmeasures==0.12` deprecates API version 2), clients will receive "Deprecation" and "Sunset" response headers [see [draft-ietf-httpapi-deprecation-header-02](#) and [RFC 8594](#), respectively].

Hosts should not expect every client to monitor response headers and proactively upgrade to newer API versions. Please make sure that your users have upgraded before you upgrade to a FlexMeasures version that sunsets an API version. You can do this by checking your server logs for warnings about users who are still calling deprecated endpoints.

In addition, we recommend running blackout tests during the deprecation notice phase. You (and your users) can learn which systems need attention and how to deal with them. Be sure to announce these beforehand. Here is an example of how to run a blackout test: If a sunset happens in version 0.13, and you are hosting a version which includes the deprecation notice (e.g. 0.12), FlexMeasures will simulate the sunset if you set the config setting `FLEXMEASURES_API_SUNSET_ACTIVE = True` (see [Sunset Configuration](#)). During such a blackout test, clients will receive HTTP status 410 (Gone) responses when calling corresponding endpoints.

What is a blackout test

A blackout test is a planned, timeboxed event when a host will turn off a certain API or some of the API capabilities. The test is meant to help developers understand the impact the retirement will have on the applications and users. [Source: Platform of Trust](#)

Stage 2: Preliminary sunset

When upgrading to a FlexMeasures version that sunsets an API version (e.g. `flexmeasures==0.13` sunsets API version 2), clients will receive HTTP status 410 (Gone) responses when calling corresponding endpoints.

In case you have users that haven't upgraded yet, and would still like to upgrade FlexMeasures (to the version that officially sunsets the API version), you can. For a little while after sunset (usually one more minor version), we will continue to support a "sunset rollback". To enable this, just set the config setting `FLEXMEASURES_API_SUNSET_ACTIVE = False` and consider announcing some more blackout tests to your users, during which you can set this setting to `True` to reactivate the sunset.

Stage 3: Definitive sunset

After upgrading to one of the next FlexMeasures versions (e.g. `flexmeasures==0.14`), clients that call sunset endpoints will receive HTTP status 410 (Gone) responses.

5.3.18 Notation

This page helps you to construct messages to the FlexMeasures API. Please consult the endpoint documentation first. Here we dive into topics useful across endpoints.

Singular vs plural keys

Throughout this document, keys are written in singular if a single value is listed, and written in plural if multiple values are listed, for example:

```
{
  "keyToValue": "this is a single value",
  "keyToValues": ["this is a value", "and this is a second value"]
}
```

The API, however, does not distinguish between singular and plural key notation.

Sensors and entity addresses

In many API endpoints, sensors are identified by their ID, e.g. `/sensors/45`. However, all sensors can also be identified with an entity address following the EA1 addressing scheme prescribed by USEF[1], which is mostly taken from IETF RFC 3720 [2].

This is the complete structure of an EA1 address:

```
{
  "sensor": "ea1.{date code}.{reversed domain name}:{locally unique string}"
}
```

Here is a full example for an entity address of a sensor in FlexMeasures:

```
{
  "sensor": "ea1.2021-02.io.flexmeasures.company:fm1.73"
}
```

where FlexMeasures runs at *company.flexmeasures.io* (which the current domain owner started using in February 2021), and the locally unique string uses the *fm1* scheme (see below) to identify sensor ID 73.

Assets are listed at:

```
https://company.flexmeasures.io/assets
```

The full entity addresses of all of the asset's sensors can be obtained on the asset's page, e.g. for asset 81:

```
https://company.flexmeasures.io/assets/81
```

Entity address structure

Some deeper explanations about an entity address:

- “ea1” is a constant, indicating this is a type 1 USEF entity address
- The date code “must be a date during which the naming authority owned the domain name used in this format, and should be the first month in which the domain name was owned by this naming authority at 00:01 GMT of the first day of the month.
- The reversed domain name is taken from the naming authority (person or organization) creating this entity address
- The locally unique string can be used for local purposes, and FlexMeasures uses it to identify the resource. Fields in the locally unique string are separated by colons, see for other examples IETF RFC 3721, page 6 [3]. While [2] says it’s possible to use dashes, dots or colons as separators, we might use dashes and dots in latitude/longitude coordinates of sensors, so we settle on colons.

[1] <https://www.usef.energy/app/uploads/2020/01/USEF-Flex-Trading-Protocol-Specifications-1.01.pdf>

[2] <https://tools.ietf.org/html/rfc3720>

[3] <https://tools.ietf.org/html/rfc3721>

Types of sensor identification used in FlexMeasures

FlexMeasures expects the locally unique string string to contain information in a certain structure. We distinguish type `fm0` and type `fm1` FlexMeasures entity addresses.

The `fm1` scheme is the latest version. It uses the fact that all FlexMeasures sensors have unique IDs.

```
ea1.2021-01.io.flexmeasures:fm1.42  
ea1.2021-01.io.flexmeasures:fm1.<sensor_id>
```

The `fm0` scheme is the original scheme. It identified different types of sensors (such as grid connections, weather sensors and markets) in different ways. The `fm0` scheme has been deprecated and is no longer supported officially.

Timeseries

Timestamps and durations are consistent with the ISO 8601 standard. The frequency of the data is implicit (from duration and number of values), while the resolution of the data is explicit, see *Frequency and resolution*.

All timestamps in requests to the API must be timezone-aware. For instance, in the below example, the timezone indication “Z” indicates a zero offset from UTC.

We use the following shorthand for sending sequential, equidistant values within a time interval:

```
{  
  "values": [  
    10,  
    5,  
    8  
  ],  
  "start": "2016-05-01T13:00:00Z",  
  "duration": "PT45M"  
}
```

Technically, this is equal to:

```
{
  "timeseries": [
    {
      "value": 10,
      "start": "2016-05-01T13:00:00Z",
      "duration": "PT15M"
    },
    {
      "value": 5,
      "start": "2016-05-01T13:15:00Z",
      "duration": "PT15M"
    },
    {
      "value": 8,
      "start": "2016-05-01T13:30:00Z",
      "duration": "PT15M"
    }
  ]
}
```

This intuitive convention allows us to reduce communication by sending univariate timeseries as arrays.

In all current versions of the FlexMeasures API, only equidistant timeseries data is expected to be communicated. Therefore:

- only the array notation should be used (first notation from above),
- “start” should be a timestamp on the hour or a multiple of the sensor resolution thereafter (e.g. “16:10” works if the resolution is 5 minutes), and
- “duration” should also be a multiple of the sensor resolution.

Describing flexibility

FlexMeasures computes schedules for energy systems that consist of multiple devices that consume and/or produce electricity. We model a device as an asset with a power sensor, and compute schedules only for flexible devices, while taking into account inflexible devices.

To compute a schedule, FlexMeasures first needs to assess the flexibility state of the system. This is described by the *flex model* (information about the state and possible actions of the flexible device) and the *flex-context* (information about the system as a whole, in order to assess the value of activating flexibility).

This information goes beyond the usual time series recorded by an asset’s sensors. It’s being sent through the API when triggering schedule computation. Some parts of it can be persisted on the asset & sensor model as attributes (that’s design work in progress).

We distinguish the information with two groups:

Flex model

The flexibility model describes to the scheduler what the flexible asset's state is, and what constraints or preferences should be taken into account. Which type of flexibility model is relevant to a scheduler usually relates to the type of device.

Usually, not the whole flexibility model is needed. FlexMeasures can infer missing values in the flex model, and even get them (as default) from the sensor's attributes. This means that API and CLI users don't have to send the whole flex model every time.

Here are the three types of flexibility models you can expect to be built-in:

- 1) For **storage devices** (e.g. batteries, and EV batteries connected to charge points), the schedule deals with the state of charge (SOC).

The possible flexibility parameters are:

- `soc-at-start` (defaults to 0)
- `soc-unit` (kWh or MWh)
- `soc-min` (defaults to 0)
- `soc-max` (defaults to max soc target)
- `soc-minima` (defaults to NaN values)
- `soc-maxima` (defaults to NaN values)
- `soc-targets` (defaults to NaN values)
- `roundtrip-efficiency` (defaults to 100%)
- `storage-efficiency` (defaults to 100%)¹
- `prefer-charging-sooner` (defaults to True, also signals a preference to discharge later)

For some examples, see the [\[POST\] /sensors/\(id\)/schedules/trigger](#) endpoint docs.

- 2) For **shiftable processes**

Todo: A simple algorithm exists, needs integration into FlexMeasures and asset type clarified.

- 3) For **buffer devices** (e.g. thermal energy storage systems connected to heat pumps), use the same flexibility parameters described above for storage devices. Here are some tips to model a buffer with these parameters:

- Describe the thermal energy content in kWh or MWh.
- Set `soc-minima` to the accumulative usage forecast.
- Set `roundtrip-efficiency` to the square of the conversion efficiency.²

In addition, folks who write their own custom scheduler (see [Plugin Customizations](#)) might also require their custom flexibility model. That's no problem, FlexMeasures will let the scheduler decide which flexibility model is relevant and how it should be validated.

¹ The storage efficiency (e.g. 95% or 0.95) to use for the schedule is applied over each time step equal to the sensor resolution. For example, a storage efficiency of 95 percent per (absolute) day, for scheduling a 1-hour resolution sensor, should be passed as a storage efficiency of $0.95^{1/24} = 0.997865$.

² Setting a roundtrip efficiency of higher than 1 is not supported. We plan to implement a separate field for COP (coefficient of performance) values.

Note: We also aim to model situations with more than one flexible asset, with different types of flexibility. This is ongoing architecture design work, and therefore happens in development settings, until we are happy with the outcomes. Thoughts welcome :)

Flex context

With the flexibility context, we aim to describe the system in which the flexible assets operates:

- **inflexible-device-sensors** — power sensors that are relevant, but not flexible, such as a sensor recording rooftop solar power connected behind the main meter, whose production falls under the same contract as the flexible device(s) being scheduled
- **consumption-price-sensor** — the sensor which defines costs/revenues of consuming energy
- **production-price-sensor** — the sensor which defines cost/revenues of producing energy

These should be independent on the asset type and consequently also do not depend on which scheduling algorithm is being used.

Tracking the recording time of beliefs

For all its time series data, FlexMeasures keeps track of the time they were recorded. Data can be defined and filtered accordingly, which allows you to get a snapshot of what was known at a certain point in time.

Note: FlexMeasures uses the [timely-beliefs data model](#) for modelling such facts about time series data, and accordingly we use the term “belief” in this documentation. In that model, the recording time is referred to as “belief time”.

Querying by recording time

Some GET endpoints have two optional timing fields to allow such filtering.

The **prior** field (a timestamp) can be used to select beliefs recorded before some moment in time. It can be used to “time-travel” to see the state of information at some moment in the past.

In addition, the **horizon** field (a duration) can be used to select beliefs recorded before some moment in time, *relative to each event*. For example, to filter out meter readings communicated within a day (denoted by a negative horizon) or forecasts created at least a day beforehand (denoted by a positive horizon).

The two timing fields follow the ISO 8601 standard and are interpreted as follows:

- **prior**: recorded prior to <timestamp>.
- **horizon**: recorded at least <duration> before the fact (indicated by a positive horizon), or at most <duration> after the fact (indicated by a negative horizon).

For example (note that you can use both fields together):

```
{
  "horizon": "PT6H",
  "prior": "2020-08-01T17:00:00Z"
}
```

These fields denote that the data should have been recorded at least 6 hours before the fact (i.e. forecasts) and prior to 5 PM on August 1st 2020 (UTC).

Note: In addition to these two timing filters, beliefs can be filtered by their source (see [Sources](#)).

Setting the recording time

Some POST endpoints have two optional fields to allow setting the time at which beliefs are recorded in an explicit manner. This is useful to keep an accurate history of what was known at what time, especially for prognoses. If not used, FlexMeasures will infer the belief time from the arrival time of the message.

The “prior” field (a timestamp) can be used to set a single time at which the entire time series (e.g. a prognosed series) was recorded. Alternatively, the “horizon” field (a duration) can be used to set the recording times relative to each (prognosed) event. In case both fields are set, the earliest possible recording time is determined and recorded for each (prognosed) event.

The two timing fields follow the ISO 8601 standard and are interpreted as follows:

```
{
  "values": [
    10,
    5,
    8
  ],
  "start": "2016-05-01T13:00:00Z",
  "duration": "PT45M",
  "prior": "2016-05-01T07:45:00Z",
}
```

This message implies that the entire prognosis was recorded at 7:45 AM UTC, i.e. 6 hours before the end of the entire time interval.

```
{
  "values": [
    10,
    5,
    8
  ],
  "start": "2016-05-01T13:00:00Z",
  "duration": "PT45M",
  "horizon": "PT6H"
}
```

This message implies that all prognosed values were recorded 6 hours in advance. That is, the value for 1:00-1:15 PM was made at 7:15 AM, the value for 1:15-1:30 PM was made at 7:30 AM, and the value for 1:30-1:45 PM was made at 7:45 AM.

Negative horizons may also be stated (breaking with the ISO 8601 standard) to indicate a belief about something that has already happened (i.e. after the fact, or simply *ex post*). For example, the following message implies that all prognosed values were made 10 minutes after the fact:

```
{
  "values": [
```

(continues on next page)

(continued from previous page)

```

    10,
    5,
    8
],
"start": "2016-05-01T13:00:00Z",
"duration": "PT45M",
"horizon": "-PT10M"
}

```

Note that, for a horizon indicating a belief 10 minutes after the *start* of each 15-minute interval, the “horizon” would have been “PT5M”. This denotes that the prognosed interval has 5 minutes left to be concluded.

Frequency and resolution

FlexMeasures handles two types of time series, which can be distinguished by defining the following timing properties for events recorded by sensors:

- Frequency: how far apart events occur (a constant duration between event starts)
- Resolution: how long an event lasts (a constant duration between the start and end of an event)

Note: FlexMeasures runs on Pandas, and follows Pandas terminology accordingly. The term frequency as used by Pandas is the reciprocal of the [SI quantity for frequency](#).

1. The first type of time series describes non-instantaneous events such as average hourly wind speed. For this case, it is commonly assumed that `frequency == resolution`. That is, events follow each other sequentially and without delay.
2. The second type of time series describes instantaneous events (zero resolution) such as temperature at a given time. For this case, we have `frequency != resolution`.

Specifying a frequency and resolution is redundant for POST requests that contain both “values” and a “duration” — FlexMeasures computes the frequency by dividing the duration by the number of values, and, for sensors that record non-instantaneous events, assumes the resolution of the data is equal to the frequency.

When POSTing data, FlexMeasures checks this inferred resolution against the required resolution of the sensors that are posted to. If these can’t be matched (through upsampling), an error will occur.

GET requests (such as `/sensors/data`) return data with a frequency either equal to the resolution that the sensor is configured for (for non-instantaneous sensors), or a default frequency befitting (in our opinion) the requested time interval. A “resolution” may be specified explicitly to obtain the data in downsampled form, which can be very beneficial for download speed. For non-instantaneous sensors, the specified resolution needs to be a multiple of the sensor’s resolution, e.g. hourly or daily values if the sensor’s resolution is 15 minutes. For instantaneous sensors, the specified resolution is interpreted as a request for data in a specific frequency. The resolution of the underlying data will remain zero (and the returned message will say so).

Sources

Requests for data may filter by source. FlexMeasures keeps track of the data source (the data's author, for example, a user, forecaster or scheduler belonging to a given organisation) of time series data. For example, to obtain data originating from data source 42, include the following:

```
{  
  "source": 42,  
}
```

Data source IDs can be found by hovering over data in charts.

Note: Older API version (< 3) accepted user IDs (integers), account roles (strings) and lists thereof, instead of data source IDs (integers).

Units

From API version 3 onwards, we are much more flexible with sent units. A valid unit for timeseries data is any unit that is convertible to the configured sensor unit registered in FlexMeasures. So, for example, you can send timeseries data with “W” unit to a “kW” sensor. And if you wish to do so, you can even send a timeseries with “kWh” unit to a “kW” sensor. In this case, FlexMeasures will convert the data using the resolution of the timeseries.

For API versions 1 and 2, the unit sent needs to be an exact match with the sensor unit, and only “MW” is allowed for power sensors.

Signs of power values

USEF recommends to use positive power values to indicate consumption and negative values to indicate production, i.e. to take the perspective of the Prosumer. If an asset has been configured as a pure producer or pure consumer, the web service will help avoid mistakes by checking the sign of posted power values.

5.3.19 Version 3.0

Summary

Resource	Operation	Description
Asset	<i>GET /api/v3_0/assets</i>	Download asset list
	<i>POST /api/v3_0/assets</i>	Create a new asset
	<i>DELETE /api/v3_0/assets/(id)</i>	Delete an asset
	<i>GET /api/v3_0/assets/(id)</i>	Get an asset
	<i>PATCH /api/v3_0/assets/(id)</i>	Update an asset
	<i>GET /api/v3_0/assets/public</i>	Return all public assets.
Chart	<i>GET /api/v3_0/assets/(id)/chart/</i>	Download a chart with time series
	<i>GET /api/v3_0/assets/(id)/chart_data/</i>	Download time series for use in charts
Data	<i>GET /api/v3_0/sensors/data</i>	Download sensor data
	<i>POST /api/v3_0/sensors/data</i>	Upload sensor data
Health	<i>GET /api/v3_0/health/ready</i>	Get readiness status
Public	<i>GET /api/</i>	List available API versions
	<i>POST /api/requestAuthToken</i>	Obtain an authentication token
	<i>GET /api/v3_0</i>	Obtain a service listing for this version
Schedule	<i>GET /api/v3_0/sensors/(id)/schedules/(uuid)</i>	Download schedule from the platform
	<i>POST /api/v3_0/sensors/(id)/schedules/trigger</i>	Trigger scheduling job
Sensor	<i>GET /api/v3_0/sensors</i>	Download sensor list
User	<i>GET /api/v3_0/users</i>	Download user list
	<i>GET /api/v3_0/users/(id)</i>	Get a user
	<i>PATCH /api/v3_0/users/(id)</i>	Patch data for an existing user
	<i>PATCH /api/v3_0/users/(id)/password-reset</i>	Password reset

API Details

GET /api/

Public endpoint to list API versions.

POST /api/requestAuthToken

API endpoint to get a fresh authentication access token. Be aware that this fresh token has a limited lifetime (which depends on the current system setting SECURITY_TOKEN_MAX_AGE).

Pass the *email* parameter to identify the user. Pass the *password* parameter to authenticate the user (if not already authenticated in current session)

GET /api/v3_0

API endpoint to get a service listing for this version.

Response Headers

- *Content-Type* – application/json

Status Codes

- 200 OK – PROCESSED

GET /api/v3_0/assets

List all assets owned by a certain account.

This endpoint returns all accessible assets for the account of the user. The *account_id* query parameter can be used to list assets from a different account.

Example response

An example of one asset being returned:

```
[
  {
    "id": 1,
    "name": "Test battery",
    "latitude": 10,
    "longitude": 100,
    "account_id": 2,
    "generic_asset_type_id": 1
  }
]
```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- [200 OK](#) – PROCESSED
- [400 Bad Request](#) – INVALID_REQUEST
- [401 Unauthorized](#) – UNAUTHORIZED
- [403 Forbidden](#) – INVALID_SENDER
- [422 Unprocessable Entity](#) – UNPROCESSABLE_ENTITY

POST /api/v3_0/assets

Create new asset.

This endpoint creates a new asset.

Example request

```
{
  "name": "Test battery",
  "generic_asset_type_id": 2,
  "account_id": 2,
  "latitude": 40,
  "longitude": 170.3,
}
```

The newly posted asset is returned in the response.

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 201 Created – CREATED
- 400 Bad Request – INVALID_REQUEST
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

DELETE /api/v3_0/assets/(id)

Delete an asset given its identifier.

This endpoint deletes an existing asset, as well as all sensors and measurements recorded for it.

Request Headers

- Authorization – The authentication token
- Content-Type – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 204 No Content – DELETED
- 400 Bad Request – INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v3_0/assets/(id)

Fetch a given asset.

This endpoint gets an asset.

Example response

```
{
  "generic_asset_type_id": 2,
  "name": "Test battery",
  "id": 1,
  "latitude": 10,
  "longitude": 100,
  "account_id": 1,
}
```

Request Headers

- Authorization – The authentication token
- Content-Type – application/json

Response Headers

- `Content-Type` – `application/json`

Status Codes

- `200 OK` – `PROCESSED`
- `400 Bad Request` – `INVALID_REQUEST`, `REQUIRED_INFO_MISSING`, `UNEXPECTED_PARAMS`
- `401 Unauthorized` – `UNAUTHORIZED`
- `403 Forbidden` – `INVALID_SENDER`
- `422 Unprocessable Entity` – `UNPROCESSABLE_ENTITY`

PATCH `/api/v3_0/assets/{id}`

Update an asset given its identifier.

This endpoint sets data for an existing asset. Any subset of asset fields can be sent.

The following fields are not allowed to be updated: - `id` - `account_id`

Example request

```
{
  "latitude": 11.1,
  "longitude": 99.9,
}
```

Example response

The whole asset is returned in the response:

```
{
  "generic_asset_type_id": 2,
  "id": 1,
  "latitude": 11.1,
  "longitude": 99.9,
  "name": "Test battery",
  "account_id": 2,
}
```

Request Headers

- `Authorization` – The authentication token
- `Content-Type` – `application/json`

Response Headers

- `Content-Type` – `application/json`

Status Codes

- `200 OK` – `UPDATED`
- `400 Bad Request` – `INVALID_REQUEST`, `REQUIRED_INFO_MISSING`, `UNEXPECTED_PARAMS`
- `401 Unauthorized` – `UNAUTHORIZED`
- `403 Forbidden` – `INVALID_SENDER`

- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v3_0/assets/<id>/chart/

GET from /assets/<id>/chart

GET /api/v3_0/assets/<id>/chart_data/

GET from /assets/<id>/chart_data

Data for use in charts (in case you have the chart specs already).

GET /api/v3_0/assets/public

Return all public assets.

This endpoint returns all public assets.

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST
- 401 Unauthorized – UNAUTHORIZED
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v3_0/health/ready

Get readiness status

Example response:

```
{
  'database_sql': True
}
```

GET /api/v3_0/sensors

API endpoint to list all sensors of an account.

This endpoint returns all accessible sensors. Accessible sensors are sensors in the same account as the current user. Only admins can use this endpoint to fetch sensors from a different account (by using the *account_id* query parameter).

Example response

An example of one sensor being returned:

```
[
  {
    "entity_address": "ea1.2021-01.io.flexmeasures.company:fm1.42",
    "event_resolution": 15,
    "generic_asset_id": 1,
    "name": "Gas demand",
    "timezone": "Europe/Amsterdam",
```

(continues on next page)

(continued from previous page)

```
    "unit": "m³/h"
  }
]
```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET `/api/v3_0/sensors/{id}/schedules/
uuid`

Get a schedule from FlexMeasures.

Optional fields

- “duration” (6 hours by default; can be increased to plan further into the future)

Example response

This message contains a schedule indicating to consume at various power rates from 10am UTC onwards for a duration of 45 minutes.

```
{
  "values": [
    2.15,
    3,
    2
  ],
  "start": "2015-06-02T10:00:00+00:00",
  "duration": "PT45M",
  "unit": "MW"
}
```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_TIMEZONE, INVALID_DOMAIN, INVALID_UNIT, UNKNOWN_SCHEDULE, UNRECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

POST /api/v3_0/sensors/{id}/schedules/trigger

Trigger FlexMeasures to create a schedule.

Trigger FlexMeasures to create a schedule for this sensor. The assumption is that this sensor is the power sensor on a flexible asset.

In this request, you can describe:

- the schedule's main features (when does it start, what unit should it report, prior to what time can we assume knowledge)
- the flexibility model for the sensor (state and constraint variables, e.g. current state of charge of a battery, or connection capacity)
- the flexibility context which the sensor operates in (other sensors under the same EMS which are relevant, e.g. prices)

For details on flexibility model and context, see *Describing flexibility*. Below, we'll also list some examples.

Note: This endpoint does not support to schedule an EMS with multiple flexible sensors at once. This will happen in another endpoint. See <https://github.com/FlexMeasures/flexmeasures/issues/485>. Until then, it is possible to call this endpoint for one flexible endpoint at a time (considering already scheduled sensors as inflexible).

The length of the schedule can be set explicitly through the 'duration' field. Otherwise, it is set by the config setting *FLEXMEASURES_PLANNING_HORIZON*, which defaults to 48 hours. If the flex-model contains targets that lie beyond the planning horizon, the length of the schedule is extended to accommodate them. Finally, the schedule length is limited by *max_planning_horizon_config*, which defaults to 2520 steps of the sensor's resolution. Targets that exceed the max planning horizon are not accepted.

The appropriate algorithm is chosen by FlexMeasures (based on asset type). It's also possible to use custom schedulers and custom flexibility models, see *Plugin Customizations*.

If you have ideas for algorithms that should be part of FlexMeasures, let us know: <https://flexmeasures.io/get-in-touch/>

Example request A

This message triggers a schedule for a storage asset, starting at 10.00am, at which the state of charge (soc) is 12.1 kWh.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh"
  }
}
```

Example request B

This message triggers a 24-hour schedule for a storage asset, starting at 10.00am, at which the state of charge (soc) is 12.1 kWh, with a target state of charge of 25 kWh at 4.00pm. The global minimum and maximum soc are set to 10 and 25 kWh, respectively. To guarantee a minimum SOC in the period prior to 4.00pm, local minima constraints are imposed (via soc-minima) at 2.00pm and 3.00pm, for 15kWh and 20kWh, respectively. Roundtrip efficiency for use in scheduling is set to 98%. Storage efficiency is set to 99.99%, denoting the state of charge left after each time step equal to the sensor's resolution. Aggregate consumption (of all devices within this EMS) should be priced by sensor 9, and aggregate production should be priced by sensor 10, where the aggregate power flow in the EMS is described by the sum over sensors 13, 14 and 15 (plus the flexible sensor being optimized, of course). Note that, if forecasts for sensors 13, 14 and 15 are not available, a schedule cannot be computed.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "duration": "PT24H",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh",
    "soc-targets": [
      {
        "value": 25,
        "datetime": "2015-06-02T16:00:00+00:00"
      },
    ],
    "soc-minima": [
      {
        "value": 15,
        "datetime": "2015-06-02T14:00:00+00:00"
      },
      {
        "value": 20,
        "datetime": "2015-06-02T15:00:00+00:00"
      }
    ],
    "soc-min": 10,
    "soc-max": 25,
    "roundtrip-efficiency": 0.98,
    "storage-efficiency": 0.9999,
  },
  "flex-context": {
    "consumption-price-sensor": 9,
    "production-price-sensor": 10,
    "inflexible-device-sensors": [13, 14, 15]
  }
}
```

Example response

This message indicates that the scheduling request has been processed without any error. A scheduling job has been created with some Universally Unique Identifier (UUID), which will be picked up by a worker. The given UUID may be used to obtain the resulting schedule: see `/sensors/<id>/schedules/<uuid>`.

```
{
  "status": "PROCESSED",
```

(continues on next page)

(continued from previous page)

```

    "schedule": "364bfd06-c1fa-430b-8d25-8f5a547651fb",
    "message": "Request has been processed."
  }

```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_DATA
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v3_0/sensors/data

Get sensor data from FlexMeasures.

Example request

```

{
  "sensor": "ea1.2021-01.io.flexmeasures:fm1.1",
  "start": "2021-06-07T00:00:00+02:00",
  "duration": "PT1H",
  "resolution": "PT15M",
  "unit": "m³/h"
}

```

The unit has to be convertible from the sensor's unit.

Optional fields

- “resolution” (see *Frequency and resolution*)
- “horizon” (see *Tracking the recording time of beliefs*)
- “prior” (see *Tracking the recording time of beliefs*)
- “source” (see *Sources*)

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

POST /api/v3_0/sensors/data

Post sensor data to FlexMeasures.

Example request

```
{
  "sensor": "ea1.2021-01.io.flexmeasures:fm1.1",
  "values": [-11.28, -11.28, -11.28, -11.28],
  "start": "2021-06-07T00:00:00+02:00",
  "duration": "PT1H",
  "unit": "m³/h"
}
```

The above request posts four values for a duration of one hour, where the first event start is at the given start time, and subsequent events start in 15 minute intervals throughout the one hour duration.

The sensor is the one with ID=1. The unit has to be convertible to the sensor's unit. The resolution of the data has to match the sensor's required resolution, but FlexMeasures will attempt to upsample lower resolutions. The list of values may include null values.

Request Headers

- Authorization – The authentication token
- Content-Type – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v3_0/users

API endpoint to list all users of an account.

This endpoint returns all accessible users. By default, only active users are returned. The *include_inactive* query parameter can be used to also fetch inactive users. Accessible users are users in the same account as the current user. Only admins can use this endpoint to fetch users from a different account (by using the *account_id* query parameter).

Example response

An example of one user being returned:

```
[
  {
    'active': True,
    'email': 'test_prosumer@seita.nl',
    'account_id': 13,
    'flexmeasures_roles': [1, 3],
    'id': 1,
    'timezone': 'Europe/Amsterdam',
    'username': 'Test Prosumer User'
  }
]
```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- [200 OK](#) – PROCESSED
- [400 Bad Request](#) – INVALID_REQUEST
- [401 Unauthorized](#) – UNAUTHORIZED
- [403 Forbidden](#) – INVALID_SENDER
- [422 Unprocessable Entity](#) – UNPROCESSABLE_ENTITY

GET /api/v3_0/users/(id)

API endpoint to get a user.

This endpoint gets a user. Only admins or the members of the same account can use this endpoint.

Example response

```
{
  'account_id': 1,
  'active': True,
  'email': 'test_prosumer@seita.nl',
  'flexmeasures_roles': [1, 3],
  'id': 1,
  'timezone': 'Europe/Amsterdam',
  'username': 'Test Prosumer User'
}
```

Request Headers

- [Authorization](#) – The authentication token
- [Content-Type](#) – application/json

Response Headers

- [Content-Type](#) – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

PATCH /api/v3_0/users/(id)

API endpoint to patch user data.

This endpoint sets data for an existing user. It has to be used by the user themselves, admins or account-admins (of the same account). Any subset of user fields can be sent. If the user is not an (account-)admin, they can only edit a few of their own fields.

The following fields are not allowed to be updated at all:

- id
- account_id

Example request

```
{
  "active": false,
}
```

Example response

The following user fields are returned:

```
{
  'account_id': 1,
  'active': True,
  'email': 'test_prosumer@seita.nl',
  'flexmeasures_roles': [1, 3],
  'id': 1,
  'timezone': 'Europe/Amsterdam',
  'username': 'Test Prosumer User'
}
```

Request Headers

- Authorization – The authentication token
- Content-Type – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – UPDATED
- 400 Bad Request – INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

PATCH /api/v3_0/users/(id)/password-reset

API endpoint to reset the user's current password, cookies and auth tokens, and to email a password reset link to the user.

Reset the user's password, and send them instructions on how to reset the password. This endpoint is useful from a security standpoint, in case of worries the password might be compromised. It sets the current password to something random, invalidates cookies and auth tokens, and also sends an email for resetting the password to the user.

Users can reset their own passwords. Only admins can use this endpoint to reset passwords of other users.

Request Headers

- **Authorization** – The authentication token
- **Content-Type** – application/json

Response Headers

- **Content-Type** – application/json

Status Codes

- 200 OK – PROCESSED
- 400 Bad Request – INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

5.3.20 Developer API

These endpoints are still under development and are subject to change in new releases.

Summary

Resource	Operation	Description
Chart	<i>GET /api/dev/asset/(id)/</i>	Download asset attributes for use in charts
	<i>GET /api/dev/sensor/(id)/</i>	Download sensor attributes for use in charts
	<i>GET /api/dev/sensor/(id)/chart/</i>	Download a chart with time series
	<i>GET /api/dev/sensor/(id)/chart_annotations/</i>	Download annotations for use in charts
	<i>GET /api/dev/sensor/(id)/chart_data/</i>	Download time series for use in charts

API Details

GET `/api/dev/asset/(id)/`

GET from `/asset/<id>`

GET `/api/dev/sensor/(id)/`

GET from `/sensor/<id>`

GET `/api/dev/sensor/(id)/chart/`

GET from `/sensor/<id>/chart`

Optional fields

- “event_starts_after” (see the [timely-beliefs documentation](#))
- “event_ends_before” (see the [timely-beliefs documentation](#))
- “beliefs_after” (see the [timely-beliefs documentation](#))
- “beliefs_before” (see the [timely-beliefs documentation](#))
- “include_data” (if true, chart specs include the data; if false, use the [GET /api/dev/sensor/\(id\)/chart_data/](#) endpoint to fetch data)
- “width” (an integer number of pixels; without it, the chart will be scaled to the full width of the container (hint: use `<div style="width: 100%;">` to set a div width to 100%))
- “height” (an integer number of pixels; without it, FlexMeasures sets a default, currently 300)

GET `/api/dev/sensor/(id)/chart_annotations/`

GET from `/sensor/<id>/chart_annotations`

Annotations for use in charts (in case you have the chart specs already).

GET `/api/dev/sensor/(id)/chart_data/`

GET from `/sensor/<id>/chart_data`

Data for use in charts (in case you have the chart specs already).

Optional fields

- “event_starts_after” (see the [timely-beliefs documentation](#))
- “event_ends_before” (see the [timely-beliefs documentation](#))
- “beliefs_after” (see the [timely-beliefs documentation](#))
- “beliefs_before” (see the [timely-beliefs documentation](#))
- “resolution” (see [resolutions](#))
- “most_recent_beliefs_only” (if true, returns the most recent belief for each event; if false, returns each belief for each event; defaults to true)

5.3.21 API change log

Note: The FlexMeasures API follows its own versioning scheme. This is also reflected in the URL, allowing developers to upgrade at their own pace.

v3.0-10 | 2023-06-12

- Introduced the `storage-efficiency` field to the `flex-model` field for `/sensors/<id>/schedules/trigger` (POST).

v3.0-9 | 2023-04-26

- Added missing documentation for the public endpoints for authentication and listing active API versions.
- Added REST endpoint for listing available services for a specific API version: `/api/v3_0` (GET). This functionality is similar to the `getService` endpoint for older API versions, but now also returns the full URL for each available service.

v3.0-8 | 2023-03-23

- Added REST endpoint for listing accounts and their account roles: `/accounts` (GET)
- Added REST endpoint for showing an account and its account roles: `/accounts/<id>` (GET)

v3.0-7 | 2023-02-28

- Fix premature deserialization of `flex-context` field for `/sensors/<id>/schedules/trigger` (POST).

v3.0-6 | 2023-02-01

- Sunset all fields that were moved to `flex-model` and `flex-context` fields to `/sensors/<id>/schedules/trigger` (POST). See v3.0-5.

v3.0-5 | 2023-01-04

- Introduced `flex-model` and `flex-context` fields to `/sensors/<id>/schedules/trigger` (POST). They are dictionaries and group pre-existing fields:
 - `soc-at-start` -> send in `flex-model` instead
 - `soc-min` -> send in `flex-model` instead
 - `soc-max` -> send in `flex-model` instead
 - `soc-targets` -> send in `flex-model` instead
 - `soc-unit` -> send in `flex-model` instead
 - `roundtrip-efficiency` -> send in `flex-model` instead
 - `prefer-charging-sooner` -> send in `flex-model` instead
 - `consumption-price-sensor` -> send in `flex-context` instead

- production-price-sensor -> send in flex-context instead
- inflexible-device-sensors -> send in flex-context instead
- Introduced the duration field to `/sensors/<id>/schedules/trigger` (POST) for setting a planning horizon explicitly.
- Allow posting soc-targets to `/sensors/<id>/schedules/trigger` (POST) that exceed the default planning horizon, and ignore posted targets that exceed the max planning horizon.
- Added a subsection on deprecating and sunseting to the Introduction section.
- Added a subsection on describing flexibility to the Notation section.

v3.0-4 | 2022-12-08

- Allow posting null values to `/sensors/data` (POST) to correctly space time series that include missing values (the missing values are not stored).
- Introduced the source field to `/sensors/data` (GET) to obtain data for a given source (ID).
- Fixed the JSON wrapping of the return message for `/sensors/data` (GET).
- Changed the Notation section:
 - Rewrote the section on filtering by source (ID) with a deprecation notice on filtering by account role and user ID.

v3.0-3 | 2022-08-28

- Introduced consumption_price_sensor, production_price_sensor and inflexible_device_sensors fields to `/sensors/<id>/schedules/trigger` (POST).

v3.0-2 | 2022-07-08

- Introduced the “resolution” field to `/sensors/data` (GET) to obtain data in a given resolution.

v3.0-1 | 2022-05-08

- Added REST endpoint for checking application health (readiness to accept requests): `/health/ready` (GET).

v3.0-0 | 2022-03-25

- Added REST endpoint for listing sensors: `/sensors` (GET).
- Added REST endpoints for managing sensor data: `/sensors/data` (GET, POST).
- Added REST endpoints for managing assets: `/assets` (GET, POST) and `/assets/<id>` (GET, PATCH, DELETE).
- Added REST endpoints for triggering and getting schedules: `/sensors/<id>/schedules/<uuid>` (GET) and `/sensors/<id>/schedules/trigger` (POST).
- **[Breaking change]** Switched to plural resource names for REST endpoints: `/users/<id>` (GET, PATCH) and `/users/<id>/password-reset` (PATCH).
- **[Breaking change]** Deprecated the following endpoints (NB replacement endpoints mentioned below no longer require the message “type” field):

- *getConnection* -> use */sensors* (GET) instead
 - *getDeviceMessage* -> use */sensors/<id>/schedules/<uuid>* (GET) instead, where *<id>* is the sensor id from the “event” field and *<uuid>* is the value of the “schedule” field returned by */sensors/<id>/schedules/trigger* (POST)
 - *getMeterData* -> use */sensors/data* (GET) instead, replacing the “connection” field with “sensor”
 - *getPrognosis* -> use */sensors/data* (GET) instead, replacing the “connection” field with “sensor”
 - *getService* -> use */api/v3_0* (GET) instead (since v3.0-9), or consult the public API documentation instead, at <https://flexmeasures.readthedocs.io>
 - *postMeterData* -> use */sensors/data* (POST) instead, replacing the “connection” field with “sensor”
 - *postPriceData* -> use */sensors/data* (POST) instead, replacing the “market” field with “sensor”
 - *postPrognosis* -> use */sensors/data* (POST) instead, replacing the “connection” field with “sensor”
 - *postUdiEvent* -> use */sensors/<id>/schedules/trigger* (POST) instead, where *<id>* is the sensor id from the “event” field, and rename the following fields:
 - * “datetime” -> “start”
 - * “value” -> “soc-at-start”
 - * “unit” -> “soc-unit”
 - * “targets” -> “soc-targets”
 - * “soc_min” -> “soc-min”
 - * “soc_max” -> “soc-max”
 - * “roundtrip_efficiency” -> “roundtrip-efficiency”
 - *postWeatherData* -> use */sensors/data* (POST) instead
 - *restoreData*
- Changed the Introduction section:
 - Rewrote the section on service listing for API versions to refer to the public documentation.
 - Rewrote the section on entity addresses to refer to *sensors* instead of *connections*.
 - Rewrote the sections on roles and sources into a combined section that refers to account roles rather than USEF roles.
 - Deprecated the section on group notation.

v2.0-7 | 2022-05-05

API v2.0 is removed.

v2.0-6 | 2022-04-26

API v2.0 is sunset.

v2.0-5 | 2022-02-13

API v2.0 is deprecated.

v2.0-4 | 2022-01-04

- Updated entity addresses in documentation, according to the fm1 scheme.
- Changed the Introduction section:
 - Rewrote the subsection on entity addresses to refer users to where they can find the entity addresses of their sensors.
 - Rewrote the subsection on sensor identification (formerly known as asset identification) to place the fm1 scheme front and center.
- Fixed the categorisation of the *postMeterData*, *postPrognosis*, *postPriceData* and *postWeatherData* endpoints from the User category to the Data category.

v2.0-3 | 2021-06-07

- Updated all entity addresses in documentation according to the fm0 scheme, preserving backwards compatibility.
- Introduced the fm1 scheme for entity addresses for connections, markets, weather sensors and sensors.

v2.0-2 | 2021-04-02

- **[Breaking change]** Switched the interpretation of horizons to rolling horizons.
- **[Breaking change]** Deprecated the use of ISO 8601 repeating time intervals to denote rolling horizons.
- Introduced the “prior” field for *postMeterData*, *postPrognosis*, *postPriceData* and *postWeatherData* endpoints.
- Changed the Introduction section:
 - Rewrote the subsection on prognoses to explain the horizon and prior fields.
- Changed the Simulation section:
 - Rewrote relevant examples using horizon and prior fields.

v2.0-1 | 2021-02-19

- Added REST endpoints for managing users: */users/* (GET), */user/<id>* (GET, PATCH) and */user/<id>/password-reset* (PATCH).

v2.0-0 | 2020-11-14

- Added REST endpoints for managing assets: `/assets/` (GET, POST) and `/asset/<id>` (GET, PATCH, DELETE).

v1.3-14 | 2022-05-05

API v1.3 is removed.

v1.3-13 | 2022-04-26

API v1.3 is sunset.

v1.3-12 | 2022-02-13

API v1.3 is deprecated.

v1.3-11 | 2022-01-05

Affects all versions since v1.3.

- Changed and extended the `postUdiEvent` endpoint:
 - The recording time of new schedules triggered by calling the endpoint is now the time at which the endpoint was called, rather than the datetime of the sent state of charge (SOC).
 - Introduced the “prior” field for the purpose of communicating an alternative recording time, thereby keeping support for simulations.
 - Introduced an optional “roundtrip_efficiency” field, for use in scheduling.

v1.3-10 | 2021-11-08

Affects all versions since v1.3.

- Fixed the `getDeviceMessage` endpoint for cases in which there are multiple schedules available, by returning only the most recent one.

v1.3-9 | 2021-04-21

Affects all versions since v1.0.

- Fixed regression by partially reverting the breaking change of v1.3-8: Re-instantiated automatic inference of horizons for Post requests for API versions below v2.0, but changed to inference policy: now inferring the data was recorded **right after each event** took place (leading to a zero horizon for each data point) rather than **after the last event** took place (which led to a different horizon for each data point); the latter had been the inference policy before v1.3-8.

v1.3-8 | 2020-04-02

Affects all versions since v1.0.

- **[Breaking change]**, partially reverted in v1.3-9] Deprecated the automatic inference of horizons for *postMeterData*, *postPrognosis*, *postPriceData* and *postWeatherData* endpoints for API versions below v2.0.

v1.3-7 | 2020-12-16

Affects all versions since v1.0.

- Separated the dual purpose of the “horizon” field in the *getMeterData* and *getPrognosis* endpoints by introducing the “prior” field:
 - The “horizon” field in GET endpoints is now always interpreted as a rolling horizon, regardless of whether it is stated as an ISO 8601 repeating time interval.
 - The *getMeterData* and *getPrognosis* endpoints now accept an optional “prior” field to select only data recorded before a certain ISO 8601 timestamp (replacing the unintuitive usage of the horizon field for specifying a latest time of belief).

v1.3-6 | 2020-12-11

Affects all versions since v1.0.

- The *getMeterData* and *getPrognosis* endpoints now return the INVALID_SOURCE status 400 response in case the optional “source” field is used and no relevant sources can be found.

v1.3-5 | 2020-10-29

Affects all versions since v1.0.

- Endpoints to POST meter data will now check incoming data to see if the required asset’s resolution is being used — upsampling is done if possible. These endpoints can now return the REQUIRED_INFO_MISSING status 400 response.
- Endpoints to GET meter data will return data in the asset’s resolution — downsampling to the “resolution” field is done if possible.
- As they need to determine the asset, all of the mentioned POST and GET endpoints can now return the UNRECOGNIZED_ASSET status 400 response.

v1.3-4 | 2020-06-18

- Improved support for use cases of the *getDeviceMessage* endpoint in which a longer duration, between posting UDI events and retrieving device messages based on those UDI events, is required; the default *time to live* of UDI event identifiers is prolonged from 500 seconds to 7 days, and can be set as a config variable (*FLEXMEASURES_PLANNING_TTL*)

v1.3-3 | 2020-06-07

- Changed backend support (API specifications unaffected) for scheduling charging stations to scheduling Electric Vehicle Supply Equipment (EVSE), in accordance with the Open Charge Point Interface (OCPI).

v1.3-2 | 2020-03-11

- Fixed example entity addresses in simulation section

v1.3-1 | 2020-02-08

- Backend change: the default planning horizon can now be set in FlexMeasures's configuration (*FLEXMEASURES_PLANNING_HORIZON*)

v1.3-0 | 2020-01-28

- Introduced new event type “soc-with-targets” to support scheduling charging stations (see extra example for the *postUdiEvent* endpoint)
- The *postUdiEvent* endpoint now triggers scheduling jobs to be set up (rather than scheduling directly triggered by the *getDeviceMessage* endpoint)
- The *getDeviceMessage* now queries the job queue and database for an available schedule

v1.2-6 | 2022-05-05

API v1.2 is removed.

v1.2-5 | 2022-04-26

API v1.2 is sunset.

v1.2-4 | 2022-02-13

API v1.2 is deprecated.

v1.2-3 | 2020-01-28

- Updated endpoint descriptions with additional possible status 400 responses:
 - `INVALID_DOMAIN` for invalid entity addresses
 - `UNKNOWN_PRICES` for infeasible schedules due to missing prices

v1.2-2 | 2018-10-08

- Added a list of registered types of weather sensors to the Simulation section and *postWeatherData* endpoint
- Changed example for the *postPriceData* endpoint to reflect Korean situation

v1.2-1 | 2018-09-24

- Added a local table of contents to the Simulation section
- Added a description of the *postPriceData* endpoint in the Simulation section
- Added a description of the *postWeatherData* endpoint in the Simulation section
- Revised the subsection about posting power data in the Simulation section
- Revised the entity address for UDI events to include the type of the event

```
i.e.
{
  "type": "PostUdiEventRequest",
  "event": "ea1.2021-01.io.flexmeasures.company:7:10:203:soc",
}

rather than the erroneously double-keyed:

{
  "type": "PostUdiEventRequest",
  "event": "ea1.2021-01.io.flexmeasures.company:7:10:203",
  "type": "soc"
}
```

v1.2-0 | 2018-09-08

- Added a description of the *postUdiEvent* endpoint in the Prosumer and Simulation sections
- Added a description of the *getDeviceMessage* endpoint in the Prosumer and Simulation sections

v1.1-8 | 2022-05-05

API v1.1 is removed.

v1.1-7 | 2022-04-26

API v1.1 is sunset.

v1.1-6 | 2022-02-13

API v1.1 is deprecated.

v1.1-5 | 2020-06-18

- Fixed the *getConnection* endpoint where the returned list of connection names had been unnecessarily nested

v1.1-4 | 2020-03-11

- Added support for posting daily and weekly prices for the *postPriceData* endpoint

v1.1-3 | 2018-09-08

- Added the Simulation section:
 - Added information about setting up a new simulation
 - Added examples for calling the *postMeterData* endpoint
 - Added example for calling the *getPrognosis* endpoint

v1.1-2 | 2018-08-15

- Added the *postPrognosis* endpoint
- Added the *postPriceData* endpoint
- Added a description of the *postPrognosis* endpoint in the Aggregator section
- Added a description of the *postPriceData* endpoint in the Aggregator and Supplier sections
- Added the *restoreData* endpoint for servers in play mode

v1.1-1 | 2018-08-06

- Added the *getConnection* endpoint
- Added the *postWeatherData* endpoint
- Changed the Introduction section:
 - Added information about the sign of power values (production is negative)
 - Updated information about horizons (now anchored to the end of each time interval rather than to the start)
- Added an optional horizon to the *postMeterData* endpoint

v1.1-0 | 2018-07-15

- Added the *getPrognosis* endpoint
- Changed the *getMeterData* endpoint to accept an optional resolution, source, and horizon
- Changed the Introduction section:
 - Added information about timeseries resolutions
 - Added information about sources
 - Updated information about horizons
- Added a description of the *getPrognosis* endpoint in the Supplier section

v1.0-4 | 2022-05-05

API v1.0 is removed.

v1.0-3 | 2022-04-26

API v1.0 is sunset.

v1.0-2 | 2022-02-13

API v1.0 is deprecated.

v1.0-1 | 2018-07-10

- Moved specifications to be part of the platform's Sphinx documentation:
 - Each API service is now documented in the docstring of its respective endpoint
 - Added sections listing all endpoints per version
 - Documentation includes specifications of **all** supported API versions (supported versions have a registered Flask blueprint)

v1.0-0 | 2018-07-10

- Started change log
- Added Introduction section with notes regarding:
 - Authentication
 - Relevant roles for the API
 - Key notation
 - The addressing scheme for assets
 - Connection group notation
 - Timeseries notation
 - Prognosis notation

- Units of timeseries data
- Added a description of the *getService* endpoint in the Introduction section
- Added a description of the *postMeterData* endpoint in the MDC section
- Added a description of the *getMeterData* endpoint in the Prosumer section

5.3.22 CLI Commands

FlexMeasures comes with a command-line utility, which helps to manage data. Below, we list all available commands. Each command has more extensive documentation if you call it with `--help`.

We keep track of changes to these commands in *FlexMeasures CLI Changelog*. You can also get the current overview over the commands you have available by:

```
flexmeasures --help
flexmeasures [command] --help
```

This also shows admin commands made available through Flask and installed extensions (such as [Flask-Security](#) and [Flask-Migrate](#)), of which some are referred to in this documentation.

add - Add data

<code>flexmeasures add initial-structure</code>	Initialize structural data like users, roles and asset types.
<code>flexmeasures add account-role</code>	Create a FlexMeasures tenant account role.
<code>flexmeasures add account</code>	Create a FlexMeasures tenant account.
<code>flexmeasures add user</code>	Create a FlexMeasures user.
<code>flexmeasures add asset-type</code>	Create a new asset type.
<code>flexmeasures add asset</code>	Create a new asset.
<code>flexmeasures add sensor</code>	Add a new sensor.
<code>flexmeasures add beliefs</code>	Load beliefs from file.
<code>flexmeasures add source</code>	Add a new data source.
<code>flexmeasures add forecasts</code>	Create forecasts.
<code>flexmeasures add schedule for-storage</code>	Create a charging schedule for a storage asset.
<code>flexmeasures add holidays</code>	Add holiday annotations to accounts and/or assets.
<code>flexmeasures add annotation</code>	Add annotation to accounts, assets and/or sensors.
<code>flexmeasures add toy-account</code>	Create a toy account, for tutorials and trying things.
<code>flexmeasures add report</code>	Create a report.

show - Show data

<code>flexmeasures show accounts</code>	List accounts.
<code>flexmeasures show account</code>	Show an account, its users and assets.
<code>flexmeasures show asset-types</code>	List available asset types.
<code>flexmeasures show asset</code>	Show an asset and its sensors.
<code>flexmeasures show roles</code>	List available account- and user roles.
<code>flexmeasures show data-sources</code>	List available data sources.
<code>flexmeasures show beliefs</code>	Plot time series data.
<code>flexmeasures show reporters</code>	List available reporters.
<code>flexmeasures show schedulers</code>	List available schedulers.

edit - Edit data

<code>flexmeasures edit attribute</code>	Edit (or add) an asset attribute or sensor attribute.
<code>flexmeasures edit resample-data</code>	Assign a new event resolution to an existing sensor and resample its data accordingly.

delete - Delete data

<code>flexmeasures delete structure</code>	Delete all structural (non time-series) data, like assets (types), roles and users.
<code>flexmeasures delete account-role</code>	Delete a tenant account role.
<code>flexmeasures delete account</code>	Delete a tenant account & also their users (with assets and power measurements).
<code>flexmeasures delete user</code>	Delete a user & also their assets and power measurements.
<code>flexmeasures delete asset</code>	Delete an asset & also its sensors and data.
<code>flexmeasures delete sensor</code>	Delete a sensor and all beliefs about it.
<code>flexmeasures delete measurements</code>	Delete measurements (with horizon ≤ 0).
<code>flexmeasures delete prognoses</code>	Delete forecasts and schedules (forecasts > 0).
<code>flexmeasures delete unchanged-beliefs</code>	Delete unchanged beliefs.
<code>flexmeasures delete nan-beliefs</code>	Delete NaN beliefs.

jobs - Job queueing

<code>flexmeasures jobs run-worker</code>	Start a worker process for forecasting and/or scheduling jobs.
<code>flexmeasures jobs show queues</code>	List job queues.
<code>flexmeasures jobs clear-queue</code>	Clear a job queue.

db-ops - Operations on the whole database

<code>flexmeasures db-ops dump</code>	Create a dump of all current data (using <i>pg_dump</i>).
<code>flexmeasures db-ops load</code>	Load backed-up contents (see <i>db-ops save</i>), run <i>reset</i> first.
<code>flexmeasures db-ops reset</code>	Reset database data and re-create tables from data model.
<code>flexmeasures db-ops restore</code>	Restore the dump file, see <i>db-ops dump</i> (run <i>reset</i> first).
<code>flexmeasures db-ops save</code>	Backup db content to files.

5.3.23 FlexMeasures CLI Changelog

since v0.14.1 | June XX, 2023

- Avoid saving any NaN values to the database, when calling `flexmeasures add report`.
- Fix defaults for the `--start-offset` and `--end-offset` options to `flexmeasures add report`, which weren't being interpreted in the local timezone of the reporting sensor.

since v0.14.0 | June 15, 2023

- Allow setting a storage efficiency using the new `--storage-efficiency` option to the `flexmeasures add schedule for-storage` CLI command.
- Add CLI command `flexmeasures add report` to calculate a custom report from sensor data and save the results to the database, with the option to export them to a CSV or Excel file.
- Add CLI command `flexmeasures show reporters` to list available reporters, including any defined in registered plugins.
- Add CLI command `flexmeasures show schedulers` to list available schedulers, including any defined in registered plugins.
- Make `--account-id` optional in `flexmeasures add asset` to support creating public assets, which are available to all users.

since v0.13.0 | May 1, 2023

- Add `flexmeasures add source` CLI command for adding a new data source.
- Add `--inflexible-device-sensor` option to `flexmeasures add schedule`.

since v0.12.0 | January 04, 2023

- Add `--resolution`, `--timezone` and `--to-file` options to `flexmeasures show beliefs`, to show beliefs data in a custom resolution and/or timezone, and also to save shown beliefs data to a CSV file.
- Add options to `flexmeasures add beliefs` to 1) read CSV data with timezone naive datetimes (use `--timezone` to localize the data), 2) read CSV data with datetime/timedelta units (use `--unit datetime` or `--unit timedelta`, 3) remove rows with NaN values, and 4) add filter to read-in data by matching values in specific columns (use `--filter-column` and `--filter-value` together).
- Fix `flexmeasures db-ops dump` and `flexmeasures db-ops restore` incorrectly reporting a success when `pg_dump` and `pg_restore` are not installed.
- Add `flexmeasures monitor last-seen`.
- Rename `flexmeasures monitor tasks` to `flexmeasures monitor last-run`.
- Rename `flexmeasures add schedule` to `flexmeasures add schedule for-storage` (in expectation of more scheduling commands, based on in-built flex models).

since v0.11.0 | August 28, 2022

- Add `flexmeasures jobs show-queues` to show contents of computation job queues.
- `--name` parameter in `flexmeasures jobs run-worker` is now optional.
- Add `--custom-message` param to `flexmeasures monitor` tasks.
- Rename `-optimization-context-id` to `--consumption-price-sensor` in `flexmeasures add schedule`, and added `--production-price-sensor`.

since v0.9.0 | March 25, 2022

- Add CLI commands for showing data `flexmeasures show accounts`, `flexmeasures show account`, `flexmeasures show roles`, `flexmeasures show asset-types`, `flexmeasures show asset`, `flexmeasures show data-sources`, and `flexmeasures show beliefs`.
- Add `flexmeasures db-ops resample-data` CLI command to resample sensor data to a different resolution.
- Add `flexmeasures edit attribute` CLI command to edit/add an attribute on an asset or sensor.
- Add `flexmeasures add toy-account` for tutorials and trying things.
- Add `flexmeasures add schedule` to create a new schedule for a given power sensor.
- Add `flexmeasures delete asset` to delete an asset (including its sensors and data).
- Rename `flexmeasures add structure` to `flexmeasures add initial-structure`.

since v0.8.0 | January 26, 2022

- Add `flexmeasures add sensor`, `flexmeasures add asset-type`, `flexmeasures add beliefs`. These were previously experimental features (under the *dev-add* command group).
- `flexmeasures add asset` now directly creates an asset in the new data model.
- Add `flexmeasures delete sensor`, `flexmeasures delete nan-beliefs` and `flexmeasures delete unchanged-beliefs`.

since v0.6.0 | April 2, 2021

- Add `flexmeasures add account`, `flexmeasures delete account`, and the `--account-id` param to `flexmeasures add user`.

since v0.4.0 | April 2, 2021

- Add the `dev-add` command group for experimental features around the upcoming data model refactoring.

since v0.3.0 | April 2, 2021

- Refactor CLI into the main groups add, delete, jobs and db-ops
- Add `flexmeasures add asset`, `flexmeasures add user` and `flexmeasures add weather-sensor`
- Split the `populate-db` command into `flexmeasures add structure` and `flexmeasures add forecasts`

5.3.24 Running via Docker

FlexMeasures can be run via [docker](#).

[Docker](#) is great to save developers from installation trouble, but also for running FlexMeasures inside modern cloud environments in a scalable manner.

Note: We also support running all needed parts of a FlexMeasures EMS setup via [docker-compose](#), which is helpful for developers and might inform hosting efforts. See [Running a complete stack with docker-compose](#).

Warning: For now, the use case is local development. Using in production is a goal for later. Follow [our progress](#).

The *flexmeasures* image

Getting the image

You can use versions we host at Docker Hub, e.g.:

```
docker pull lfenergy/flexmeasures:latest
```

You can also build the FlexMeasures image yourself, from source:

```
docker build -t flexmeasures/my-version .
```

The tag is your choice.

Running

Running the image (as a container) might work like this (remember to get the image first, see above):

```
docker run --env SQLALCHEMY_DATABASE_URI=postgresql://user:pass@localhost:5432/dbname --  
→env SECRET_KEY=blabla --env FLASK_ENV=development -d --net=host lfenergy/flexmeasures
```

Note: Don't know what your image is called (its "tag")? We used `lfenergy/flexmeasures` here, as that should be the name when pulling it from Docker Hub. You can run `docker images` to see which images you have.

The two minimal environment variables to run the container successfully are the database URI and the secret key, see [Configuration](#). `FLASK_ENV=development` is needed if you do not have an SSL certificate set up (the default mode is `production`, and in that mode FlexMeasures requires https for security reasons). If you see too much output, you can also set `LOGGING_LEVEL=INFO`.

In this example, we connect to a postgres database running on our local computer, so we use the host network. In the docker-compose section below, we use a Docker container for the database, as well.

Browsing `http://localhost:5000` should work now and ask you to log in.

Of course, you might not have created a user. You can use `docker exec -it <flexmeasures-container-name> bash` to go inside the container and use the *CLI Commands* to create everything you need.

Configuration and customization

Using *Configuration* by file is usually what you want to do. It's easier than adding environment variables to `docker run`. Also, not all settings can be given via environment variables. A good example is the `MAPBOX_ACCESS_TOKEN`, so you can load maps on the dashboard.

To load a configuration file into the container when starting up, we make use of the *instance folder*. You can put a configuration file called `flexmeasures.cfg` into a local folder called `flexmeasures-instance` and then mount that folder into the container, like this:

```
docker run -v $(pwd)/flexmeasures-instance:/app/instance:ro -d --net=host lfenergy/
↪ flexmeasures
```

Warning: The location of the instance folder depends on how we serve FlexMeasures. The above works with gunicorn. See the compose file for an alternative (for the FlexMeasures CLI), and you can also read the above link about the instance folder.

Note: This is also a way to add your custom logic (as described in *Writing Plugins*) to the container. We'll document that shortly. Plugins which should be installed (e.g. by `pip`) are a bit more difficult to support (you'd need to add `pip install` before the actual entry point). Ideas welcome.

5.3.25 Postgres database

This document describes how to get the postgres database ready to use and maintain it (do migrations / changes to the structure).

Note: This is about a stable database, useful for longer development work or production. A super quick way to get a postgres database running with Docker is described in *Toy example: Scheduling a battery, from scratch*. In *Running a complete stack with docker-compose* we use both postgres and redis.

We also spend a few words on coding with database transactions in mind.

Table of contents

- *Getting ready to use*
 - *Install*
 - *Make sure postgres represents datetimes in UTC timezone*
 - *Create “flexmeasures” and “flexmeasures_test” databases and users*

- *Add Postgres Extensions to your database(s)*
- *Configure FlexMeasures app for that database*
- *Get structure (and some data) into place*
- *Visualize the data model*
- *Maintenance*
 - *Make first migration*
 - *Make another migration*
 - *Get database structure updated*
 - *Working with the migration history*
 - *Check out database status*
- *Transaction management*

Getting ready to use

Notes:

- We assume `flexmeasures` for your database and username here. You can use anything you like, of course.
- The name `flexmeasures_test` for the test database is good to keep this way, as automated tests are looking for that database / user / password.

Install

We believe FlexMeasures works with Postgres above version 9 and we ourselves have run it with versions up to 14.

On Unix:

```
$ sudo apt-get install postgresql-12 # replace 12 with the version available in your
↪packages
$ pip install psycopg2-binary
```

On Windows:

- Download postgres here: <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>
- Install and remember your postgres user password
- Add the lib and bin directories to your Windows path: <http://bobbyong.com/blog/installing-postgresql-on-windoes/>
- `conda install psycopg2`

Make sure postgres represents datetimes in UTC timezone

(Otherwise, pandas can get confused with daylight saving time.)

Luckily, many web hosters already have `timezone= 'UTC'` set correctly by default, but local postgres installations often use `timezone='localtime'`.

In any case, check both your local installation and the server, like this:

Find the `postgres.conf` file. Mine is at `/etc/postgresql/9.6/main/postgresql.conf`. You can also type `SHOW config_file;` in a postgres console session (as superuser) to find the config file.

Find the `timezone` setting and set it to `'UTC'`.

Then restart the postgres server.

```
$ sudo service postgresql restart
```

Create “flexmeasures” and “flexmeasures_test” databases and users

From the terminal:

Open a console (use your Windows key and type `cmd`). Proceed to create a database as the postgres superuser (using your postgres user password):

```
$ sudo -i -u postgres
$ createdb -U postgres flexmeasures
$ createdb -U postgres flexmeasures_test
$ createuser --pwprompt -U postgres flexmeasures      # enter your password
$ createuser --pwprompt -U postgres flexmeasures_test # enter "flexmeasures_test" as
↪password
$ exit
```

Or, from within Postgres console:

```
CREATE USER flexmeasures WITH UNENCRYPTED PASSWORD 'this-is-your-secret-choice';
CREATE DATABASE flexmeasures WITH OWNER = flexmeasures;
CREATE USER flexmeasures_test WITH UNENCRYPTED PASSWORD 'flexmeasures_test';
CREATE DATABASE flexmeasures_test WITH OWNER = flexmeasures_test;
```

Finally, test if you can log in as the flexmeasures user:

```
$ psql -U flexmeasures --password -h 127.0.0.1 -d flexmeasures
```

```
\q
```


Add Postgres Extensions to your database(s)

To find the nearest sensors, FlexMeasures needs some extra Postgres support. Add the following extensions while logged in as the postgres superuser:

```
$ sudo -u postgres psql
```

```
\connect flexmeasures
CREATE EXTENSION cube;
CREATE EXTENSION earthdistance;
```

If you have it, connect to the `flexmeasures_test` database and repeat creating these extensions there. Then `exit`.

Configure FlexMeasures app for that database

Write:

```
SQLALCHEMY_DATABASE_URI = "postgresql://flexmeasures:<password>@127.0.0.1/flexmeasures"
```

into the config file you are using, e.g. `~/flexmeasures.cfg`

Get structure (and some data) into place

You need data to enjoy the benefits of FlexMeasures or to develop features for it. In this section, there are some ways to get started.

Import from another database

Here is a short recipe to import data from a FlexMeasures database (e.g. a demo database) into your local system.

On the to-be-exported database:

```
$ flexmeasures db-ops dump
```

Note: Only the data gets dumped here.

Then, we create the structure in our database anew, based on the data model given by the local codebase:

```
$ flexmeasures db-ops reset
```

Then we import the data dump we made earlier:

```
$ flexmeasures db-ops restore <DATABASE DUMP FILENAME>
```

A potential `alembic_version` error should not prevent other data tables from being restored. You can also choose to import a complete db dump into a freshly created database, of course.

Note: To make sure passwords will be decrypted correctly when you authenticate, set the same `SECURITY_PASSWORD_SALT` value in your config as the one that was in use when the dumped passwords were encrypted!

Create data manually

First, you can get the database structure with:

```
$ flexmeasures db upgrade
```

Note: If you develop code (and might want to make changes to the data model), you should also check out the maintenance section about database migrations.

You can create users with the `add user` command. Check it out:

```
$ flexmeasures add user --help
```

You can create some pre-determined asset types and data sources with this command:

```
$ flexmeasures add initial-structure
```

You can also create assets in the FlexMeasures UI.

On the command line, you can add many things. Check what data you can add yourself:

```
$ flexmeasures add --help
```

For instance, you can create forecasts for your existing metered data with this command:

```
$ flexmeasures add forecasts --help
```

Check out its `--help` content to learn more. You can set which assets and which time window you want to forecast. Of course, making forecasts takes a while for a larger dataset. You can also simply queue a job with this command (and run a worker to process the [Redis Queues](#)).

Just to note, there are also commands to get rid of data. Check:

```
$ flexmeasures delete --help
```

Check out the [CLI Commands](#) documentation for more details.

Visualize the data model

You can visualise the data model like this:

```
$ make show-data-model
```

This will generate a picture based on the model code. You can also generate picture based on the actual database, see inside the Makefile.

Maintenance

Maintenance is supported with the alembic tool. It reacts automatically to almost all changes in the SQLAlchemy code. With alembic, multiple databases, such as development, staging and production databases can be kept in sync.

Make first migration

Run these commands from the repository root directory (read below comments first):

```
$ flexmeasures db init
$ flexmeasures db migrate
$ flexmeasures db upgrade
```

The first command (`flexmeasures db init`) is only needed here once, it initialises the alembic migration tool. The second command generates the SQL for your current db model and the third actually gives you the db structure.

With every migration, you get a new migration step in `migrations/versions`. Be sure to add that to `git`, as future calls to `flexmeasures db upgrade` will need those steps, and they might happen on another computer.

Hint: You can edit these migrations steps, if you want.

Make another migration

Just to be clear that the `db init` command is needed only at the beginning - you usually do, if your model changed:

```
$ flexmeasures db migrate --message "Please explain what you did, it helps for later"
$ flexmeasures db upgrade
```

Get database structure updated

The goal is that on any other computer, you can always execute

```
$ flexmeasures db upgrade
```

to have the database structure up-to-date with all migrations.

Working with the migration history

The history of migrations is at your fingertips:

```
$ flexmeasures db current
$ flexmeasures db history
```

You can move back and forth through the history:

```
$ flexmeasures db downgrade
$ flexmeasures db upgrade
```

Both of these accept a specific revision id parameter, as well.

Check out database status

Log in into the database:

```
$ psql -U flexmeasures --password -h 127.0.0.1 -d flexmeasures
```

with the password from `flexmeasures/development_config.py`. Check which tables are there:

```
\dt
```

To log out:

```
\q
```

Transaction management

It is really useful (and therefore an industry standard) to bundle certain database actions within a transaction. Transactions are atomic - either the actions in them all run or the transaction gets rolled back. This keeps the database in a sane state and really helps having expectations during debugging.

Please see the package `flexmeasures.data.transactional` for details on how a FlexMeasures developer should make use of this concept. If you are writing a script or a view, you will find there the necessary structural help to bundle your work in a transaction.

5.3.26 How to deploy FlexMeasures

Here you can learn how to get FlexMeasures onto a server.

Note: FlexMeasures can be deployed via Docker. Read more at [Running via Docker](#). You need other components (e.g. postgres and redis) which are not handled here. See [Running a complete stack with docker-compose](#) for inspiration.

Table of contents

- [WSGI configuration](#)
- [Install the linear solver on the server](#)

WSGI configuration

On your own computer, `flexmeasures run` is a nice way to start FlexMeasures. On a production web server, you want it done the WSGI way. Here is an example how to serve FlexMeasures as WSGI app:

```
# This file contains the WSGI configuration required to serve up your
# web application.
# It works by setting the variable 'application' to a WSGI handler of some description.
# The crucial part are the last two lines. We add some ideas for possible other logic.

import os
project_home = u'/path/to/your/code/flexmeasures'
```

(continues on next page)

(continued from previous page)

```
# use this if you want to load your own ``.env`` file.
from dotenv import load_dotenv
load_dotenv(os.path.join(project_home, '.env'))
# use this if you run from source
if project_home not in sys.path:
    sys.path = [project_home] + sys.path
# adapt PATH to find our LP solver if it is installed from source
os.environ["PATH"] = os.environ.get("PATH") + ":/home/seita/Cbc-2.9/bin"

# create flask app - the name "application" has to be passed to the WSGI server
from flexmeasures.app import create as create_app
application = create_app()
```

The web server is told about the WSGI script, but also about the object which represents the application. For instance, if this script is called `wsgi.py`, then the relevant argument to the gunicorn server is `wsgi:application`.

Keep in mind that FlexMeasures is based on [Flask](#), so almost all knowledge on the web on how to deploy a Flask app also helps with deploying FlexMeasures.

Install the linear solver on the server

To compute schedules, FlexMeasures uses the [Cbc](#) mixed integer linear optimization solver. It is used through [Pyomo](#), so in principle supporting a [different solver](#) would be possible.

Cbc needs to be present on the server where FlexMeasures runs, under the `cbc` command.

You can install it on Debian like this:

```
$ apt-get install coinor-cbc
```

If you can't use the package manager on your host, the solver has to be installed from source. We provide an example script in `ci/install-cbc-from-source.sh` to do that, where you can also pass a directory for the installation.

In case you want to install a later version, adapt the version in the script.

5.3.27 Redis Queues

Requirements

The hard computation work (e.g. forecasting, scheduling) should happen outside of web requests (asynchronously), in job queues accessed by worker processes.

This queueing relies on a Redis server, which has to be installed locally, or used on a separate host. In the latter case, configure [Redis](#) details in your FlexMeasures config file.

Here we assume you have access to a Redis server and configured it (see [Redis](#)). The FlexMeasures unit tests use `fakeredis` to simulate this task queueing, with no configuration required.

Note: See also [Running a complete stack with docker-compose](#) for usage of Redis via Docker and a more hands-on tutorial on the queues.

Run workers

Here is how to run one worker for each kind of job (in separate terminals):

```
$ flexmeasures jobs run-worker --name our-only-worker --queue forecasting|scheduling
```

Running multiple workers in parallel might be a great idea.

```
$ flexmeasures jobs run-worker --name forecaster --queue forecasting
$ flexmeasures jobs run-worker --name scheduler --queue scheduling
```

You can also clear the job queues:

```
$ flexmeasures jobs clear-queue --queue forecasting
$ flexmeasures jobs clear-queue --queue scheduling
```

When the main FlexMeasures process runs (e.g. by `flexmeasures run`), the queues of forecasting and scheduling jobs can be visited at `http://localhost:5000/tasks/forecasting` and `http://localhost:5000/tasks/schedules`, respectively (by admins).

Inspect the queue and jobs

The first option to inspect the state of the forecasting queue should be via the formidable [RQ dashboard](#). If you have admin rights, you can access it at `your-flexmeasures-url/rq/`, so for instance `http://localhost:5000/rq/`. You can also start RQ dashboard yourself (but you need to know the redis server credentials):

```
$ pip install rq-dashboard
$ rq-dashboard --redis-host my.ip.addr.ess --redis-password secret --redis-database 0
```

RQ dashboard shows you ongoing and failed jobs, and you can see the error messages of the latter, which is very useful.

Finally, you can also inspect the queue and jobs via a console (see the nice [RQ documentation](#)), which is more powerful. Here is an example of inspecting the finished jobs and their results:

```
from redis import Redis
from rq import Queue
from rq.job import Job
from rq.registry import FinishedJobRegistry

r = Redis("my.ip.addr.ess", port=6379, password="secret", db=2)
q = Queue("forecasting", connection=r)
finished = FinishedJobRegistry(queue=q)

finished_job_ids = finished.get_job_ids()
print("%d jobs finished successfully." % len(finished_job_ids))

job1 = Job.fetch(finished_job_ids[0], connection=r)
print("Result of job %s: %s" % (job1.id, job1.result))
```

Redis queues on Windows

On Unix, the `rq` system is automatically set up as part of FlexMeasures's main setup (the `rq` dependency).

However, `rq` is [not functional on Windows](#) without the Windows Subsystem for Linux.

On these versions of Windows, FlexMeasures's queuing system uses an extension of Redis Queue called `rq-win`. This is also an automatically installed dependency of FlexMeasures.

However, the Redis server needs to be set up separately. Redis itself does not work on Windows, so it might be easiest to commission a Redis server in the cloud (e.g. on [kamatera.com](#)).

If you want to install Redis on Windows itself, it can be set up on a virtual machine as follows:

- [Install Vagrant on Windows](#) and [VirtualBox](#)
- Download the [vagrant-redis](#) vagrant configuration
- Extract `vagrant-redis.zip` in any folder, e.g. in `c:\vagrant-redis`
- Set `config.vm.box = "hashicorp/precise64"` in the Vagrantfile, and remove the line with `config.vm.box_url`
- Run `vagrant up` in Command Prompt
- In case `vagrant up` fails because VT-x is not available, [enable it](#) in your bios [if you can](#) (more debugging tips [here](#) if needed)

5.3.28 Error monitoring

When you run a FlexMeasures server, you want to stay on top of things going wrong. We added two ways of doing that:

- You can connect to Sentry, so that all errors will be sent to your Sentry account. Add the token you got from Sentry in the config setting `SENTRY_SDN` and you're up and running!
- Another source of crucial errors are things that did not even happen! For instance, a (bot) user who is supposed to send data regularly, fails to connect with FlexMeasures. Or, a task to import prices from a day-ahead market, which you depend on later for scheduling, fails silently.

Let's look at how to monitor for things not happening in more detail:

Monitoring the time users were last seen

The CLI task `flexmeasures monitor last-seen` lets you be alerted if a user has contacted your FlexMeasures instance longer ago than you expect. This is most useful for bot users (a.k.a. scripts).

Here is an example for illustration:

```
$ flexmeasures monitor last-seen --account-role SubscriberToServiceXYZ --user-role bot --
↪maximum-minutes-since-last-seen 100
```

As you see, users are filtered by roles. You might need to add roles before this works as you want.

Todo: Adding roles and assigning them to users and/or accounts is not supported by the CLI or UI yet (besides `flexmeasures add account-role`). This is [work in progress](#). Right now, it requires you to add roles on the database level.

Monitoring task runs

The CLI task `flexmeasures monitor latest-run` lets you be alerted when tasks have not successfully run at least so-and-so many minutes ago. The alerts will come in via Sentry, but you can also send them to email addresses with the config setting `FLEXMEASURES_MONITORING_MAIL_RECIPIENTS`.

For illustration, here is one example of how we monitor the latest run times of tasks on a server — the below is run in a cron script every hour and checks if every listed task ran 60, 6 or 1440 minutes ago, respectively:

```
$ flexmeasures monitor latest-run --task get_weather_forecasts 60 --task get_recent_
meter_data 6 --task import_epex_prices 1440
```

The first task (`get_weather_forecasts`) is actually supported within FlexMeasures, while the other two sit in plugins we wrote.

This task status monitoring is enabled by decorating the functions behind these tasks with:

```
@task_with_status_report
def my_function():
    ...
```

Then, FlexMeasures will log if this task ran, and if it succeeded or failed. The result is in the table `latest_task_runs`, and that's where the `flexmeasures monitor latest-run` will look.

Note: The decorator should be placed right before the function (after all other decorators).

Per default the function name is used as task name. If the number of tasks accumulate (e.g. by using multiple plugins that each define a task or two), it is useful to come up with more dedicated names. You can add a custom name as argument to the decorator:

```
@task_with_status_report("pluginA_myFunction")
def my_function():
    ...
```

5.3.29 Modes

FlexMeasures can be run in specific modes (see the `FLEXMEASURES_MODE` config setting). This is useful for certain special situations. Two are supported out of the box and we document here how FlexMeasures behaves differently in these modes.

Demo

In this mode, the server is assumed to be used as a demonstration tool. Most of the following adaptations therefore happen in the UI.

- [Data] Demo data is often from an older source, and it's a hassle to change the year to the current year. FlexMeasures allows to set `FLEXMEASURES_DEMO_YEAR` and when in demo mode, the current year will be translated to that year in the background.
- [UI] Logged-in users can view queues on the demo server (usually only admins can do that)
- [UI] Demo servers often display login credentials, so visitors can try out functionality. Use the `FLEXMEASURES_PUBLIC_DEMO_CREDENTIALS` config setting to do this.

- [UI] The dashboard shows all non-empty asset groups, instead of only the ones for the current user.
- [UI] The analytics page mocks confidence intervals around power, price and weather data, so that the demo data doesn't need to have them.
- [UI] The portfolio page mocks flexibility numbers and a mocked control action.

Play

In this mode, the server is assumed to be used to run simulations.

Big features

- [API] The inferred recording time of incoming data is immediately after the event took place, rather than the actual time at which the server received the data.
- [API] Posting price or weather data does not trigger forecasting jobs.
- [API] The `restoreData` endpoint is registered, enabling database resets through the API.

Note: A former feature of play mode is now a separate config setting. To allow overwriting existing data when saving data to the database, use `FLEXMEASURES_ALLOW_DATA_OVERWRITE`.

Small features

- [API] Posted UDI events are not enforced to be consecutive.
- [API] Names in `GetConnectionResponse` are the connections' unique database names rather than their display names (this feature is planned to be deprecated).
- [UI] The dashboard plot showing the latest power value is not enforced to lie in the past (in case of simulating future values).

5.3.30 Writing Plugins

You can extend FlexMeasures with functionality like UI pages, API endpoints, CLI functions and custom scheduling algorithms. This is eventually how energy flexibility services are built on top of FlexMeasures!

In a nutshell, a FlexMeasures plugin adds functionality via one or more [Flask Blueprints](#).

How to make FlexMeasures load your plugin

Use the config setting `FLEXMEASURES_PLUGINS` to list your plugin(s).

A setting in this list can:

1. point to a plugin folder containing an `__init__.py` file
2. be the name of an installed module (i.e. in a Python console `import <module_name>` would work)

Each plugin defines at least one Blueprint object. These will be registered with the Flask app, so their functionality (e.g. routes) becomes available.

We'll discuss an example below.

In that example, we use the first option from above to tell FlexMeasures about the plugin. It is the simplest way to start playing around.

The second option (the plugin being an importable Python package) allows for more professional software development. For instance, it is more straightforward in that case to add code hygiene, version management and dependencies (your plugin can depend on a specific FlexMeasures version and other plugins can depend on yours).

To hit the ground running with that approach, we provide a [CookieCutter template](#). It also includes a few Blueprint examples and best practices.

Continue reading the [Plugin showcase](#) or possibilities to do [Plugin Customizations](#).

5.3.31 Plugin showcase

Here is a showcase file which constitutes a FlexMeasures plugin called `our_client`.

- We demonstrate adding a view, which can be rendered using the FlexMeasures base templates.
- We also showcase a CLI function which has access to the FlexMeasures `app` object. It can be called via `flexmeasures our-client test`.

We first create the file `<some_folder>/our_client/__init__.py`. This means that `our_client` is the plugin folder and becomes the plugin name.

With the `__init__.py` below, plus the custom Jinja2 template, `our_client` is a complete plugin.

```
__version__ = "2.0"

from flask import Blueprint, render_template, abort

from flask_security import login_required
from flexmeasures.ui.utils.view_utils import render_flexmeasures_template

our_client_bp = Blueprint('our-client', __name__,
                          template_folder='templates')

# Showcase: Adding a view

@our_client_bp.route('/')
@our_client_bp.route('/my-page')
@login_required
def my_page():
    msg = "I am a FlexMeasures plugin !"
    # Note that we render via the in-built FlexMeasures way
    return render_flexmeasures_template(
        "my_page.html",
        message=msg,
    )

# Showcase: Adding a CLI command

import click
from flask import current_app
from flask.cli import with_appcontext
```

(continues on next page)

(continued from previous page)

```

our_client_bp.cli.help = "Our client commands"

@our_client_bp.cli.command("test")
@with_appcontext
def our_client_test():
    print(f"I am a CLI command, part of FlexMeasures: {current_app}")

```

Note: You can overwrite FlexMeasures routing in your plugin. In our example above, we are using the root route `/`. FlexMeasures registers plugin routes before its own, so in this case visiting the root URL of your app will display this plugged-in view (the same you'd see at */my-page*).

Note: The `__version__` attribute on our module is being displayed in the standard FlexMeasures UI footer, where we show loaded plugins. Of course, it can also be useful for your own maintenance.

The template would live at `<some_folder>/our_client/templates/my_page.html`, which works just as other FlexMeasures templates (they are Jinja2 templates):

```

{% extends "base.html" %}

{% set active_page = "my-page" %}

{% block title %} Our client dashboard {% endblock %}

{% block divs %}

    <!-- This is where your custom content goes... -->

    {{ message }}

{% endblock %}

```

Note: Plugin views can also be added to the FlexMeasures UI menu — just name them in the config setting `FLEXMEASURES_MENU_LISTED_VIEWS`. In this example, add `my-page`. This also will make the `active_page` setting in the above template useful (highlights the current page in the menu).

Starting the template with `{% extends "base.html" %}` integrates your page content into the FlexMeasures UI structure. You can also extend a different base template. For instance, we find it handy to extend `base.html` with a custom base template, to extend the footer, as shown below:

```

{% extends "base.html" %}

{% block copyright_notice %}

Created by <a href="https://seita.nl/">Seita Energy Flexibility</a>,
in cooperation with <a href="https://ourclient.nl/">Our Client</a>
&copy;

```

(continues on next page)

(continued from previous page)

```
<script>var CurrentYear = new Date().getFullYear(); document.write(CurrentYear)
</script>.

{% endblock copyright_notice %}
```

We'd name this file `our_client_base.html`. Then, we'd extend our page template from `our_client_base.html`, instead of `base.html`.

Using other code files in your non-package plugin

Say you want to include other Python files in your plugin, importing them in your `__init__.py` file. With this file-only version of loading the plugin (if your plugin isn't imported as a package), this is a bit tricky.

But it can be achieved if you put the plugin path on the import path. Do it like this in your `__init__.py`:

```
import os
import sys

HERE = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, HERE)

from my_other_file import my_function
```

Notes on writing tests for your plugin

Good software practice is to write automatable tests. We encourage you to also do this in your plugin. We do, and our CookieCutter template for plugins (see above) has simple examples how that can work for the different use cases (i.e. UI, API, CLI).

However, there are two caveats to look into:

- Your tests need a FlexMeasures app context. FlexMeasure's app creation function provides a way to inject a list of plugins directly. The following could be used for instance in your app fixture within the top-level `conftest.py` if you are using `pytest`:

```
from flexmeasures.app import create as create_flexmeasures_app
from .. import __name__

test_app = create_flexmeasures_app(env="testing", plugins=[f"../{__name__}"])
```

- Test frameworks collect tests from your code and therefore might import your modules. This can interfere with the registration of routes on your Blueprint objects during plugin registration. Therefore, we recommend reloading your route modules, after the Blueprint is defined and before you import them. For example:

```
my_plugin_ui_bp: Blueprint = Blueprint(
    "MyPlugin-UI",
    __name__,
    template_folder="my_plugin/ui/templates",
    static_folder="my_plugin/ui/static",
    url_prefix="/MyPlugin",
)
# Now, before we import this dashboard module, in which the "/dashboard" route is_
```

(continues on next page)

(continued from previous page)

```

↪ attached to my_plugin_ui_bp,
# we make sure it's being imported now, *after* the Blueprint's creation.
importlib.reload(sys.modules["my_plugin.my_plugin.ui.views.dashboard"])
from my_plugin.ui.views import dashboard

```

The packaging path depends on your plugin's package setup, of course.

5.3.32 Plugin Customizations

Adding your own scheduling algorithm

FlexMeasures comes with in-built scheduling algorithms for often-used use cases. However, you can use your own algorithm, as well.

The idea is that you'd still use FlexMeasures' API to post flexibility states and trigger new schedules to be computed (see *Posting flexibility states*), but in the background your custom scheduling algorithm is being used.

Let's walk through an example!

First, we need to write a class (inhering from the Base Scheduler) with a *schedule* function which accepts arguments just like the in-built schedulers (their code is [here](#)). The following minimal example gives you an idea of some meta information you can add for labeling your data, as well as the inputs and outputs of such a scheduling function:

```

from datetime import datetime, timedelta
import pandas as pd
from pandas.tseries.frequencies import to_offset
from flexmeasures import Scheduler, Sensor

class DummyScheduler(Scheduler):

    __author__ = "My Company"
    __version__ = "2"

    def compute(
        self,
        *args,
        **kwargs
    ):
        """
        Just a dummy scheduler that always plans to consume at maximum capacity.
        (Schedulers return positive values for consumption, and negative values for ↪
        production)
        """
        return pd.Series(
            self.sensor.get_attribute("capacity_in_mw"),
            index=pd.date_range(self.start, self.end, freq=self.resolution, inclusive=
            ↪ "left"),
        )

    def deserialize_config(self):
        """Do not care about any flex config sent in."""
        self.config_deserialized = True

```

Note: It's possible to add arguments that describe the asset flexibility model and the flexibility (EMS) context in more detail. For example, for storage assets we support various state-of-charge parameters. For details on flexibility model and context, see *Describing flexibility* and the [POST] `/sensors/{id}/schedules/trigger` endpoint.

Finally, make your scheduler be the one that FlexMeasures will use for certain sensors:

```
from flexmeasures import Sensor

scheduler_specs = {
    "module": "flexmeasures.data.tests.dummy_scheduler", # or a file path, see note_
    ↪ below
    "class": "DummyScheduler",
}

my_sensor = Sensor.query.filter(Sensor.name == "My power sensor on a flexible asset").
    ↪ one_or_none()
my_sensor.attributes["custom-scheduler"] = scheduler_specs
```

From now on, all schedules (see *Forecasting & scheduling*) which are requested for this sensor should get computed by your custom function! For later lookup, the data will be linked to a new data source with the name “My Opinion”.

Note: To describe the module, we used an importable module here (actually a custom scheduling function we use to test this). You can also provide a full file path to the module, e.g. “/path/to/my_file.py”.

Todo: We're planning to use a similar approach to allow for custom forecasting algorithms, as well.

Deploying your plugin via Docker

You can extend the FlexMeasures Docker image with your plugin's logic.

Imagine your plugin package (with an `__init__.py` file, one of the setups we discussed in *Plugin showcase*) is called `flexmeasures_testplugin`. Then, this is a minimal possible Dockerfile — containers based on this will serve FlexMeasures (see the original Dockerfile in the FlexMeasures repository) with the plugin logic, like endpoints:

```
FROM lfenergy/flexmeasures

COPY flexmeasures_testplugin/ /app/flexmeasures_testplugin
ENV FLEXMEASURES_PLUGINS="/app/flexmeasures_testplugin"
```

You can of course also add multiple plugins this way.

If you also want to install your requirements, you could for instance add these layers:

```
COPY requirements/app.in /app/requirements/flexmeasures_testplugin.txt
RUN pip3 install --no-cache-dir -r requirements/flexmeasures_testplugin.txt
```

Note: No need to install flexmeasures here, as the Docker image we are based on already installed FlexMeasures from code. If you `pip3-install` your plugin here (assuming it's on Pypi), check if it recognizes that FlexMeasures installation

as it should.

Adding your own style sheets

You can style your plugin's pages in a distinct way by adding your own style-sheet. This happens by overwriting FlexMeasures styles block. Add to your plugin's base template (see above):

```
{% block styles %}
    {{ super() }}
    <!-- Our client styles -->
    <link rel="stylesheet" href="{% url_for('our_client_bp.static', filename='css/style.
→css') %}">
{% endblock %}
```

This will find `css/styles.css` if you add that folder and file to your Blueprint's static folder.

Note: This styling will only apply to the pages defined in your plugin (to pages based on your own base template). To apply a styling to all other pages which are served by FlexMeasures, consider using the config setting `FLEXMEASURES_EXTRA_CSS_PATH`.

Adding config settings

FlexMeasures can automatically check for you if any custom config settings, which your plugin is using, are present. This can be very useful in maintaining installations of FlexMeasures with plugins. Config settings can be registered by setting the (optional) `__settings__` attribute on your plugin module:

```
__settings__ = {
    "MY_PLUGIN_URL": {
        "description": "URL used by my plugin for x.",
        "level": "error",
    },
    "MY_PLUGIN_TOKEN": {
        "description": "Token used by my plugin for y.",
        "level": "warning",
        "message_if_missing": "Without this token, my plugin will not do y.",
        "parse_as": str,
    },
    "MY_PLUGIN_COLOR": {
        "description": "Color used to override the default plugin color.",
        "level": "info",
    },
}
```

Alternatively, use `from my_plugin import __settings__` in your plugin module, and create `__settings__.py` with:

```
MY_PLUGIN_URL = {
    "description": "URL used by my plugin for x.",
    "level": "error",
}
```

(continues on next page)

(continued from previous page)

```

MY_PLUGIN_TOKEN = {
    "description": "Token used by my plugin for y.",
    "level": "warning",
    "message_if_missing": "Without this token, my plugin will not do y.",
    "parse_as": str,
}
MY_PLUGIN_COLOR = {
    "description": "Color used to override the default plugin color.",
    "level": "info",
}

```

Finally, you might want to override some FlexMeasures configuration settings from within your plugin. Some examples for possible settings are named on this page, e.g. the custom style (see above) or custom logo (see below). There is a *record_once* function on Blueprints which can help with this. An example:

```

@our_client_bp.record_once
def record_logo_path(setup_state):
    setup_state.app.config[
        "FLEXMEASURES_MENU_LOGO_PATH"
    ] = "/path/to/my/logo.svg"

```

Using a custom favicon icon

The favicon might be an important part of your customisation. You probably want your logo to be used.

First, your blueprint needs to know about a folder with static content (this is fairly common — it's also where you'd put your own CSS or JavaScript files):

```

our_client_bp = Blueprint(
    "our_client",
    "our_client",
    static_folder="our_client/ui/static",
)

```

Put your icon file in that folder. The exact path may depend on how you set your plugin directories up, but this is how a blueprint living in its own directory could work.

Then, overwrite the `/favicon.ico` route which FlexMeasures uses to get the favicon from:

```

from flask import send_from_directory

@our_client_bp.route("/favicon.ico")
def favicon():
    return send_from_directory(
        our_client_bp.static_folder,
        "img/favicon.png",
        mimetype="image/png",
    )

```

Here we assume your favicon is a PNG file. You can also use a classic *.ico* file, then your mime type probably works best as `image/x-icon`.

Validating arguments in your CLI commands with marshmallow

Arguments to CLI commands can be validated using `marshmallow`. FlexMeasures is using this functionality (via the `MarshmallowClickMixin` class) and also defines some custom field schemas. We demonstrate this here, and also show how you can add your own custom field schema:

```
from datetime import datetime

import click
from flexmeasures.data.schemas import AwareDateTimeField
from flexmeasures.data.schemas.utils import MarshmallowClickMixin
from marshmallow import fields

class CLIStrField(fields.Str, MarshmallowClickMixin):
    """
    String field validator, made usable for CLI functions.
    You could also define your own validations here.
    """

@click.command("meet")
@click.option(
    "--where",
    required=True,
    type=CLIStrField(),
    help="(Required) Where we meet",
)
@click.option(
    "--when",
    required=False,
    type=AwareDateTimeField(format="iso"), # FlexMeasures already made this field_
    ↪suitable for CLI functions
    help="[Optional] When we meet (expects timezone-aware ISO 8601 datetime format)",
)
def schedule_meeting(
    where: str,
    when: datetime | None = None,
):
    print(f"Okay, see you {where} on {when}.")
```

Customising the login page teaser

FlexMeasures shows an image carousel next to its login form (see `ui/templates/admin/login_user.html`).

You can overwrite this content by adding your own login template and defining the teaser block yourself, e.g.:

```
{% extends "admin/login_user.html" %}

{% block teaser %}

    <h1>Welcome to my plugin!</h1>

{% endblock %}
```

Place this template file in the template folder of your plugin blueprint (see above). Your template must have a different filename than “login_user”, so FlexMeasures will find it properly!

Finally, add this config setting to your FlexMeasures config file (using the template filename you chose, obviously):

```
SECURITY_LOGIN_USER_TEMPLATE = "my_user_login.html"
```

5.3.33 Developing for FlexMeasures

This page instructs developers who work on FlexMeasures how to set up the development environment. Furthermore, we discuss several guidelines and best practices.

Table of contents

- *Getting started*
- *Logfile*
- *Tests*
- *Versioning*
- *Auto-applying formatting and code style suggestions*
- *Using Visual Studio, including spell checking*
- *A hint about using notebooks*
- *A hint for Unix developers*

Warning: Are you implementing code based on FlexMeasures, please read `note_on_datamodel_transition`.

Getting started

Virtual environment

Using a virtual environment is best practice for Python developers. We also strongly recommend using a dedicated one for your work on FlexMeasures, as our make target (see below) will use `pip-sync` to install dependencies, which could interfere with some libraries you already have installed.

- Make a virtual environment: `python3.10 -m venv flexmeasures-venv` or use a different tool like `mkvirtualenv` or `virtualenvwrapper`. You can also use an [Anaconda distribution](#) as base with `conda create -n flexmeasures-venv python=3.10`.
- Activate it, e.g.: `source flexmeasures-venv/bin/activate`

Download FlexMeasures

Clone the [FlexMeasures repository](#) from GitHub.

```
$ git clone https://github.com/FlexMeasures/flexmeasures.git
```

Dependencies

Go into the `flexmeasures` folder and install all dependencies including the ones needed for development:

```
$ cd flexmeasures
$ make install-for-dev
```

Install the LP solver. On Unix the Cbc LP solver can be installed with:

```
$ apt-get install coinor-cbc
```

Configuration

Most configuration happens in a config file, see [Configuration](#) on where it can live and all supported settings.

For now, we let it live in your home directory and we add the first required setting: a secret key:

```
echo "SECRET_KEY=\"`python3 -c 'import secrets; print(secrets.token_hex(24))'`\"" >> ~/.
flexmeasures.cfg
```

Also, we add some env settings in an `.env` file. Create that file in the `flexmeasures` directory (from where you'll run `flexmeasures`) and enter:

```
FLASK_ENV="development"
LOGGING_LEVEL="INFO"
```

The development mode makes sure we don't need SSL to connect, among other things.

Database

See [Postgres database](#) for tips on how to install and upgrade databases (postgres and redis).

Loading data

If you have a SQL Dump file, you can load that:

```
$ psql -U {user_name} -h {host_name} -d {database_name} -f {file_path}
```

One other possibility is to add a toy account (which owns some assets and a battery):

```
$ flexmeasures add toy-account
```

Run locally

Now, to start the web application, you can run:

```
$ flexmeasures run
```

Or:

```
$ python run-local.py
```

And access the server at <http://localhost:5000>

If you added a toy account, you could log in with *toy-user@flexmeasures.io*, password *toy-password*.

Otherwise, you need to add some other user first. Here is how we add an admin:

```
$ flexmeasures add account --name MyCompany
$ flexmeasures add user --username admin --account-id 1 --email admin@mycompany.io --
↪ roles admin
```

(The account-id you need in the 2nd command is printed by the 1st)

Logfile

FlexMeasures logs to a file called `flexmeasures.log`. You'll find this in the application's context folder, e.g. where you called `flexmeasures run`.

A rolling log file handler is used, so if `flexmeasures.log` gets to a few megabytes in size, it is copied to *flexmeasures.log.1* and the original file starts over empty again.

The default logging level is `WARNING`. To see more, you can update this with the config setting `LOGGING_LEVEL`, e.g. to `INFO` or `DEBUG`

Tests

You can run automated tests with:

```
$ make test
```

which behind the curtains installs dependencies and calls `pytest`.

However, a test database (postgres) is needed to run these tests. If you have postgres, here is the short version on how to add the test database:

```
$ make clean-db db_name=flexmeasures_test db_user=flexmeasures_test
$ # the password for the db user is "flexmeasures_test"
```

Note: The section *Postgres database* has more details on using postgres for FlexMeasures.

Alternatively, if you don't feel like installing postgres for the time being, here is a docker command to provide a test database:

```
$ docker run --rm --name flexmeasures-test-db -e POSTGRES_PASSWORD=flexmeasures_test -e
↳ POSTGRES_DB=flexmeasures_test -e POSTGRES_USER=flexmeasures_test -p 5432:5432 -v ./ci/
↳ load-psql-extensions.sql:/docker-entrypoint-initdb.d/load-psql-extensions.sql -d
↳ postgres:latest
```

Warning: This assumes that the port 5432 is not being used on your machine (for instance by an existing postgres database service).

If you want the tests to create a coverage report (printed on the terminal), you can run the `pytest` command like this:

```
$ pytest --cov=flexmeasures --cov-config .coveragerc
```

You can add `--cov-report=html`, after which a file called `htmlcov/index.html` is generated. Or, after a test run with coverage turned on as shown above, you can still generate it in another form:

```
$ python3 -m coverage [html|lcov|json]
```

Versioning

We use `setuptools_scm` for versioning, which bases the FlexMeasures version on the latest git tag and the commits since then.

So as a developer, it's crucial to use git tags for versions only.

We use semantic versioning, and we always include the patch version, not only max and min, so that `setuptools_scm` makes the correct guess about the next minor version. Thus, we should use `2.0.0` instead of `2.0`.

See `to_pypi.sh` for more commentary on the development versions.

Our API has its own version, which moves much slower. This is important to explicitly support outside apps who were coded against older versions.

Auto-applying formatting and code style suggestions

We use `Black` to format our Python code and `Flake8` to enforce the PEP8 style guide and linting. We also run `mypy` on many files to do some static type checking.

We do this so real problems are found faster and the discussion about formatting is limited. All of these can be installed by using `pip`, but we recommend using them as a pre-commit hook. To activate that behaviour, do:

```
$ pip install pre-commit
$ pre-commit install
```

in your virtual environment.

Now each git commit will first run `flake8`, then `black` and finally `mypy` over the files affected by the commit (`pre-commit` will install these tools into its own structure on the first run).

This is also what happens automatically server-side when code is committed to a branch (via GitHub Actions), but having those tests locally as well will help you spot these issues faster.

If `flake8`, `black` or `mypy` propose changes to any file, the commit is aborted (saying that it “failed”). The changes proposed by `black` are implemented automatically (you can review them with `git diff`). Some of them might even resolve the `flake8` warnings :)

Using Visual Studio, including spell checking

Are you using Visual Studio Code? Then the code you just cloned also contains the editor configuration (part of) our team is using (see `.vscode`)!

We recommend installing the `flake8` and `spellright` extensions.

For `spellright`, the FlexMeasures repository contains the project dictionary. Here are steps to link main dictionaries, which usually work on a Linux system:

```
$ mkdir $HOME/.config/Code/Dictionaries
$ ln -s /usr/share/hunspell/* ~/.config/Code/Dictionaries
```

Consult the extension's Readme for other systems.

A hint about using notebooks

If you edit notebooks, make sure results do not end up in git:

```
$ conda install -c conda-forge nbstripout
$ nbstripout --install
```

(on Windows, maybe you need to look closer at <https://github.com/kynan/nbstripout>)

A hint for Unix developers

I added this to my `~/.bashrc`, so I only need to type `fm` to get started and have the ssh agent set up, as well as up-to-date code and dependencies in place.

```
addssh(){
    eval `ssh-agent -s`
    ssh-add ~/.ssh/id_github
}
fm(){
    addssh
    cd ~/workspace/flexmeasures
    git pull # do not use if any production-like app runs from the git code
    workon flexmeasures-venv # this depends on how you created your virtual environment
    make install-for-dev
}
```

Note: All paths depend on your local environment, of course.

5.3.34 Configuration

The following configurations are used by FlexMeasures.

Required settings (e.g. postgres db) are marked with a double star (**). To enable easier quickstart tutorials, continuous integration use cases and basic usage of FlexMeasures within other projects, these required settings, as well as a few others, can be set by environment variables — this is also noted per setting. Recommended settings (e.g. mail, redis) are marked by one star (*).

Note: FlexMeasures is best configured via a config file. The config file for FlexMeasures can be placed in one of two locations:

- in the user’s home directory (e.g. `~/flexmeasures.cfg` on Unix). In this case, note the dot at the beginning of the filename!
- in the app’s instance directory (e.g. `/path/to/your/flexmeasures/code/instance/flexmeasures.cfg`). The path to that instance directory is shown to you by running `flexmeasures` (e.g. `flexmeasures run`) with required settings missing or otherwise by running `flexmeasures shell`.

Basic functionality

LOGGING_LEVEL

Level above which log messages are added to the log file. See the `logging` package in the Python standard library.

Default: `logging.WARNING`

Note: This setting is also recognized as environment variable.

FLEXMEASURES_MODE

The mode in which FlexMeasures is being run, e.g. “demo” or “play”. This is used to turn on certain extra behaviours, see [Modes](#) for details.

Default: `""`

FLEXMEASURES_ALLOW_DATA_OVERWRITE

Whether to allow overwriting existing data when saving data to the database.

Default: `False`

FLEXMEASURES_LP_SOLVER

The command to run the scheduling solver. This is the executable command which FlexMeasures calls via the [pyomo library](#). Other values might be `cplex` or `glpk`. Consult [their documentation](#) to learn more.

Default: `"cbc"`

FLEXMEASURES_HOSTS_AND_AUTH_START

Configuration used for entity addressing. This contains the domain on which FlexMeasures runs and the first month when the domain was under the current owner's administration.

Default: `{"flexmeasures.io": "2021-01"}`

FLEXMEASURES_PLUGINS

A list of plugins you want FlexMeasures to load (e.g. for custom views or CLI functions). This can be a Python list (e.g. `["plugin1", "plugin2"]`) or a comma-separated string (e.g. `"plugin1, plugin2"`).

Two types of entries are possible here:

- File paths (absolute or relative) to plugins. Each such path needs to point to a folder, which should contain an `__init__.py` file where the Blueprint is defined.
- Names of installed Python modules.

Added functionality in plugins needs to be based on Flask Blueprints. See [Writing Plugins](#) for more information and examples.

Default: `[]`

Note: This setting is also recognized as environment variable (since v0.14, which is also the version required to pass this setting as a string).

FLEXMEASURES_DB_BACKUP_PATH

Relative path to the folder where database backups are stored if that feature is being used.

Default: `"migrations/dumps"`

FLEXMEASURES_PROFILE_REQUESTS

Whether to turn on a feature which times requests made through FlexMeasures. Interesting for developers.

Default: `False`

UI

FLEXMEASURES_PLATFORM_NAME

Name being used in headings and in the menu bar.

For more fine-grained control, this can also be a list, where it's possible to set the platform name for certain account roles (as a tuple of view name and list of applicable account roles). In this case, the list is searched from left to right, and the first fitting name is used.

For example, `(["MyMDCApp", ["MDC"]], ["MyApp"])` would show the name "MyMDCApp" for users connected to accounts with the account role "MDC", while all others would see the name "/MyApp".

Note: This fine-grained control requires FlexMeasures version 0.6.0

Default: "FlexMeasures"

FLEXMEASURES_MENU_LOGO_PATH

A URL path to identify an image being used as logo in the upper left corner (replacing some generic text made from platform name and the page title). The path can be a complete URL or a relative from the app root.

Default: ""

FLEXMEASURES_EXTRA_CSS_PATH

A URL path to identify a CSS style-sheet to be added to the base template. The path can be a complete URL or a relative from the app root.

Note: You can also add extra styles for plugins with the usual Blueprint method. That is more elegant but only applies to the Blueprint's views.

Default: ""

FLEXMEASURES_ROOT_VIEW

Root view (reachable at "/"). For example `"/dashboard"`.

For more fine-grained control, this can also be a list, where it's possible to set the root view for certain account roles (as a tuple of view name and list of applicable account roles). In this case, the list is searched from left to right, and the first fitting view is shown.

For example, `(["metering-dashboard", ["MDC", "Prosumer"]], ["default-dashboard"])` would route to `"/metering-dashboard"` for users connected to accounts with account roles "MDC" or "Prosumer", while all others would be routed to `"/default-dashboard"`.

If this setting is empty or not applicable for the current user, the "/" view will be shown (FlexMeasures' default dashboard or a plugin view which was registered at "/").

Default []

Note: This setting was introduced in FlexMeasures version 0.6.0

FLEXMEASURES_MENU_LISTED_VIEWS

A list of the view names which are listed in the menu.

Note: This setting only lists the names of views, rather than making sure the views exist.

For more fine-grained control, the entries can also be tuples of view names and list of applicable account roles. For example, the entry (`"details": ["MDC", "Prosumer"]`) would add the `"/details"` link to the menu only for users who are connected to accounts with roles `"MDC"` or `"Prosumer"`. For clarity: the title of the menu item would read `"Details"`, see also the `FLEXMEASURES_LISTED_VIEW_TITLES` setting below.

Note: This fine-grained control requires FlexMeasures version 0.6.0

Default: `["dashboard", "analytics", "portfolio", "assets", "users"]`

FLEXMEASURES_MENU_LISTED_VIEW_ICONS

A dictionary containing a Font Awesome icon name for each view name listed in the menu. For example, `{"freezer-view": "snowflake-o"}` puts a snowflake icon () next to your freezer-view menu item.

Default: `{}`

Note: This setting was introduced in FlexMeasures version 0.6.0

FLEXMEASURES_MENU_LISTED_VIEW_TITLES

A dictionary containing a string title for each view name listed in the menu. For example, `{"freezer-view": "Your freezer"}` lists the freezer-view in the menu as `"Your freezer"`.

Default: `{}`

Note: This setting was introduced in FlexMeasures version 0.6.0

FLEXMEASURES_HIDE_NAN_IN_UI

Whether to hide the word “nan” if any value in metrics tables is NaN.

Default: False

RQ_DASHBOARD_POLL_INTERVAL

Interval in which viewing the queues dashboard refreshes itself, in milliseconds.

Default: 3000 (3 seconds)

FLEXMEASURES_ASSET_TYPE_GROUPS

How to group asset types together, e.g. in a dashboard.

Default: {"renewables": ["solar", "wind"], "EVSE": ["one-way_evse", "two-way_evse"]}

FLEXMEASURES_JS_VERSIONS

Default: {"vega": "5.22.1", "vegaembed": "6.20.8", "vegalite": "5.2.0"}

Timing

FLEXMEASURES_TIMEZONE

Timezone in which the platform operates. This is useful when datetimes are being localized.

Default: "Asia/Seoul"

FLEXMEASURES_JOB_TTL

Time to live for jobs (e.g. forecasting, scheduling) in their respective queue.

A job that is passed this time to live might get cleaned out by Redis' memory manager.

Default: timedelta(days=1)

FLEXMEASURES_PLANNING_TTL

Time to live for schedule UUIDs of successful scheduling jobs. Set a negative timedelta to persist forever.

Default: timedelta(days=7)

FLEXMEASURES_JOB_CACHE_TTL

Time to live for the job caching keys in seconds. The default value of 1h responds to the reality that within an hour, there is not much change, other than the input arguments, that justifies recomputing the schedules.

In an hour, we will have more accurate forecasts available and the situation of the power grid might have changed (imbalance prices, distribution level congestion, activation of FCR or aFRR reserves, ...).

Set a negative value to persist forever.

Warning: Keep in mind that unless a proper clean up mechanism is set up, the number of caching keys will grow with time if the TTL is set to a negative value.

Default: 3600

FLEXMEASURES_DEFAULT_DATASOURCE

The default DataSource of the resulting data from *DataGeneration* classes.

Default: "FlexMeasures"

FLEXMEASURES_PLANNING_HORIZON

The default horizon for making schedules. API users can set a custom duration if they need to.

Default: `timedelta(days=2)`

FLEXMEASURES_MAX_PLANNING_HORIZON

The maximum horizon for making schedules. API users are not able to request longer schedules. Can be set to a specific `datetime.timedelta` or to an integer number of planning steps, where the duration of a planning step is equal to the resolution of the applicable power sensor. Set to `None` to forgo this limitation altogether.

Default: 2520 (e.g. 7 days for a 4-minute resolution sensor, 105 days for a 1-hour resolution sensor)

Access Tokens

MAPBOX_ACCESS_TOKEN

Token for accessing the MapBox API (for displaying maps on the dashboard and asset pages). You can learn how to obtain one [here](#)

Default: `None`

Note: This setting is also recognized as environment variable.

SENTRY_SDN

Set tokenized URL, so errors will be sent to Sentry when `app.env` is not in *debug* or *testing* mode. E.g.: `https://<examplePublicKey>@o<something>.ingest.sentry.io/<project-Id>`

Default: None

Note: This setting is also recognized as environment variable.

SQLAlchemy

This is only a selection of the most important settings. See [the Flask-SQLAlchemy Docs](#) for all possibilities.

SQLALCHEMY_DATABASE_URI (**)

Connection string to the postgres database, format: `postgresql://<user>:<password>@<host-address>[:<port>]/<db>`

Default: None

Note: This setting is also recognized as environment variable.

SQLALCHEMY_ENGINE_OPTIONS

Configuration of the SQLAlchemy engine.

Default:

```
{
  "pool_recycle": 299,
  "pool_pre_ping": True,
  "connect_args": {"options": "-c timezone=utc"},
}
```

SQLALCHEMY_TEST_DATABASE_URI

When running tests (`make test`, which runs `pytest`), the default database URI is set in `utils.config_defaults.TestingConfig`. You can use this setting to overwrite that URI and point the tests to an (empty) database of your choice.

Note: This setting is only supported as an environment variable, not in a config file, and only during testing.

Security

This is only a selection of the most important settings. See [the Flask-Security Docs](#) as well as the [Flask-CORS docs](#) for all possibilities.

SECRET_KEY (**)

Used to sign user sessions and also as extra salt (a.k.a. pepper) for password salting if SECURITY_PASSWORD_SALT is not set. This is actually part of Flask - but is also used by Flask-Security to sign all tokens.

It is critical this is set to a strong value. For python3 consider using: `secrets.token_urlsafe()` You can also set this in a file (which some Flask tutorials advise).

Note: Leave this setting set to `None` to get more instructions when you attempt to run FlexMeasures.

Default: `None`

SECURITY_PASSWORD_SALT

Extra password salt (a.k.a. pepper)

Default: `None` (falls back to SECRET_KEY)

SECURITY_TOKEN_AUTHENTICATION_HEADER

Name of the header which carries the auth bearer token in API requests.

Default: `Authorization`

SECURITY_TOKEN_MAX_AGE

Maximal age of security tokens in seconds.

Default: `60 * 60 * 6` (six hours)

SECURITY_TRACKABLE

Whether to track user statistics. Turning this on requires certain user fields. We do not use this feature, but we do track number of logins.

Default: `False`

CORS_ORIGINS

Allowed cross-origins. Set to "*" to allow all. For development (e.g. JavaScript on localhost) you might use "null" in this list.

Default: []

CORS_RESOURCES:

FlexMeasures resources which get cors protection. This can be a regex, a list of them or a dictionary with all possible options.

Default: [r"/api/*"]

CORS_SUPPORTS_CREDENTIALS

Allows users to make authenticated requests. If true, injects the Access-Control-Allow-Credentials header in responses. This allows cookies and credentials to be submitted across domains.

Note: This option cannot be used in conjunction with a "*" origin.

Default: True

Mail

For FlexMeasures to be able to send email to users (e.g. for resetting passwords), you need an email account which can do that (e.g. GMail).

This is only a selection of the most important settings. See [the Flask-Mail Docs](#) for others.

Note: The mail settings are also recognized as environment variables.

MAIL_SERVER (*)

Email name server domain.

Default: "localhost"

MAIL_PORT (*)

SMTP port of the mail server.

Default: 25

MAIL_USE_TLS

Whether to use TLS.

Default: False

MAIL_USE_SSL

Whether to use SSL.

Default: False

MAIL_USERNAME (*)

Login name of the mail system user.

Default: None

MAIL_DEFAULT_SENDER (*)

Tuple of shown name of sender and their email address.

Note: Some recipient mail servers will refuse emails for which the shown email address (set under MAIL_DEFAULT_SENDER) differs from the sender's real email address (registered to MAIL_USERNAME). Match them to avoid SMTPRecipientsRefused errors.

Default:

```
(  
    "FlexMeasures",  
    "no-reply@example.com",  
)
```

MAIL_PASSWORD

Password of mail system user.

Default: None

Monitoring

Monitoring potential problems in FlexMeasure's operations.

SENTRY_DSN

Set tokenized URL, so errors will be sent to Sentry when `app.env` is not in *debug* or *testing* mode. E.g.: `https://<examplePublicKey>@o<something>.ingest.sentry.io/<project-Id>`

Default: None

FLEXMEASURES_SENTRY_CONFIG

A dictionary with values to configure reporting to Sentry. Some options are taken care of by FlexMeasures (e.g. environment and release), but not all. See *here* <<https://docs.sentry.io/platforms/python/configuration/options/>>_ for a complete list.

Default: {}

FLEXMEASURES_TASK_CHECK_AUTH_TOKEN

Token which external services can use to check on the status of recurring tasks within FlexMeasures.

Default: None

FLEXMEASURES_MONITORING_MAIL_RECIPIENTS

E-mail addresses to send monitoring alerts to from the CLI task `flexmeasures monitor tasks`. For example `["fred@one.com", "wilma@two.com"]`

Default: []

Redis

FlexMeasures uses the Redis database to support our forecasting and scheduling job queues.

Note: The redis settings are also recognized as environment variables.

FLEXMEASURES_REDIS_URL (*)

URL of redis server.

Default: "localhost"

FLEXMEASURES_REDIS_PORT (*)

Port of redis server.

Default: 6379

FLEXMEASURES_REDIS_DB_NR (*)

Number of the redis database to use (Redis per default has 16 databases, numbered 0-15)

Default: 0

FLEXMEASURES_REDIS_PASSWORD (*)

Password of the redis server.

Default: None

Demonstrations

FLEXMEASURES_PUBLIC_DEMO_CREDENTIALS

When `FLEXMEASURES_MODE=demo`, this can hold login credentials (demo user email and password, e.g. ("demo" at seita.nl", "flexdemo")), so anyone can log in and try out the platform.

Default: None

FLEXMEASURES_DEMO_YEAR

When `FLEXMEASURES_MODE=demo`, this setting can be used to make the FlexMeasures platform select data from a specific year (e.g. 2015), so that old imported data can be demoed as if it were current.

Default: None

Sunset

FLEXMEASURES_API_SUNSET_ACTIVE

Allow control over the effect of sunsetting API versions. Specifically, if `True`, the endpoints of sunset API versions will return `HTTP status 410 (Gone)` status codes. If `False`, these endpoints will either return `HTTP status 410 (Gone)` status codes, or work like before (including Deprecation and Sunset headers in their response), depending on whether the installed FlexMeasures version still contains the endpoint implementations.

Default: `False`

FLEXMEASURES_API_SUNSET_DATE

Allow to override the default sunset date for your clients.

Default: None (defaults are set internally for each sunset API version, e.g. "2023-05-01" for v2.0)

FLEXMEASURES_API_SUNSET_LINK

Allow to override the default sunset link for your clients.

Default: None (defaults are set internally for each sunset API version, e.g. "https://flexmeasures.readthedocs.io/en/v0.13.0/api/v2_0.html" for v2.0)

5.3.35 Developing on the API

The FlexMeasures API is the main way that third-parties can automate their interaction with FlexMeasures, so it's highly important.

This is a small guide for creating new versions of the API and its docs.

Warning: This guide was written for API versions below v3.0 and is currently out of date.

Todo: A guide for endpoint design, e.g. using Marshmallow schemas and common validators.

Table of contents

- *Introducing a new API version*
 - *Set up new module with routes*
 - *Set up a new blueprint*
 - *New or updated endpoint implementations*
 - *Testing*
 - *UI Crud*
 - *Documentation*

Introducing a new API version

Larger changes to the API, other than fixes and refactoring, should be done by creating a new API version. In the guide we're assuming the new version is v1.1.

Whether we need a new API version or not, doesn't have a clear set of rules yet. Certainly backward-incompatible changes should require one, but as you'll see, there is also certain overhead in creating a new version, so a careful trade-off is advised.

Note: For the rest of this guide we'll assume your new API version is v1.1.

Set up new module with routes

In `flexmeasures/api` create a new module (folder with `__init__.py`). Copy over the `routes.py` from the previous API version. By default we import all routes from the previous version:

```
from flexmeasures.api.v1 import routes as v1_routes, implementations as v1_
    ↪ implementations
```

Set the service listing for this version (or overwrite completely if needed):

```
v1_1_service_listing = copy.deepcopy(v1_routes.v1_service_listing)
v1_1_service_listing["version"] = "1.1"
```

Then update and redecorate each API endpoint as follows:

```
@flexmeasures_api.route("/getService", methods=["GET"])
@as_response_type("GetServiceResponse")
@append_doc_of(v1_routes.get_service)
def get_service():
    return v1_implementations.get_service_response(v1_1_service_listing)
```

Set up a new blueprint

In the new module's `flexmeasures/api/v1_1/__init__.py`, copy the contents of `flexmeasures/api/v1/__init__.py` (previous API version). Change all references to the version name in the new file (for example: `flexmeasures_api_v1` should become `flexmeasures_api_v1_1`).

In `flexmeasures/api/__init__.py` update the version listing in `get_versions()` and register a blueprint for the new api version by adding:

```
from flexmeasures.api.v1_1 import register_at as v1_1_register_at
v1_1_register_at(app)
```

New or updated endpoint implementations

Write functionality of new or updated endpoints in:

```
flexmeasures/api/v1_1/implementations.py
```

Utility functions that are commonly shared between endpoint implementations of different versions should go in:

```
flexmeasures/api/common/utils
```

where we distinguish between response decorators, request validators and other utils.

Testing

If you changed an endpoint in the new version, write a test for it. Usually, there is no need to copy the tests for unchanged endpoints, if not a major API version is being released.

Test the entire api or just your new version:

```
$ pytest -k api
$ pytest -k v1_1
```

UI Crud

In `ui/crud`, we support FlexMeasures' in-built UI with Flask endpoints, which then talk to our internal API. The routes used there point to an API version. You should consider updating them to point to your new version.

Documentation

In `documentation/api` start a new specification `v1_1.rst` with contents like this:

```
.. _v1_1:

Version 1.1
=====

Summary
-----

.. qrefflask:: flexmeasures.app:create()
   :blueprints: flexmeasures_api, flexmeasures_api_v1_1
   :order: path
   :include-empty-docstring:

API Details
-----

.. autoflask:: flexmeasures.app:create()
   :blueprints: flexmeasures_api, flexmeasures_api_v1_1
   :order: path
   :include-empty-docstring:
```

If you are ready to publish the new specifications, enter your changes in `documentation/api/change_log.rst` and update the api toctree in `documentation/index.rst` to include the new version in the table of contents.

You're not done. Several sections in the API documentation list endpoints as examples. If you want other developers to use your new API version, make sure those examples reference the latest endpoints. Remember that [Sphinx autoflask](#) likes to prefix the names of endpoints with the blueprint's name, for example:

```
.. autoflask:: flexmeasures.app:create()
   :endpoints: flexmeasures_api_v1_1.post_meter_data
```

5.3.36 Continuous integration

Automate deployment via Github actions and Git

At FlexMeasures headquarters, we implemented a specific workflow to automate our deployment. It uses the Github action workflow (see the `.github/workflows` directory), which pushes to a remote upstream repository. We use this workflow to build and deploy the project to our staging server.

Documenting this might be useful for self-hosters, as well. The GitHub Actions workflows are triggered by commits being pushed to the repository, but it can also inspire your custom deployment script.

We'll refer to Github Actions as our “CI environment” and our staging server as the “deployment server”.

- In `lint-and-test.yml`, we set up the app, then run the tests and linters. If testing succeeds and if the commit was on the `main` branch, `deploy.yml` deploys the code from the CI environment to the deployment server.
- Of course, the CI environment needs to properly authenticate at the deployment server.
- With the hooks functionality of Git, a post-receive script can then (re-)start the FlexMeasures app on the deployment server.

Let's review these three steps in detail:

Using git to deploy code (remote upstream)

We support deployment of the FlexMeasures project on a staging server via Git checkout.

The deployment uses git's ability to push code to a remote upstream repository. This repository needs to be installed on your staging server.

We trigger this deployment in `deploy.yml` and it's being done in `DEPLOY.sh`. There, we add the remote and then push the current branch to it.

We thus need to tell the deployment environment two things:

- Add the setting `STAGING_REMOTE_REPO` as an environment variable on the CI environment (e.g. `deploy.yml` expects it in the Github repository secrets). An example value is `seita@ssh.our-server.com:/home/seita/flexmeasures-staging/flexmeasures.git`. So in this case, `ssh.our-server.com` is the deployment server, which we'll also use below. `seita` needs to become your ssh username on that server and the rest is the path to where you want to check out the repo.
- Make sure the env variable `BRANCH_NAME` is set, e.g. to “`main`”, so that the CI environment knows what exact code to push to your deployment server.

Authenticate at the deployment server (with an ssh key)

For CI environment and deployment server to interact securely, we of course need to put in place some authentication measures.

First, they need to know each other. Let the deployment server know it's okay to talk to the CI environment, by adding an entry to `~/.ssh/known_hosts`. Similarly, you might need to let the CI environment know it's okay to talk to the deployment server (e.g. in our Github Actions config, `deploy.yml` expects this entry in the Github repository secrets as `KNOWN_DEPLOYMENT_HOSTS`).

You can create these entries with `ssh-keyscan -t rsa <your host>`, where host might be `github.com` or `ssh.our-server.com` (see above).

Second, the CI environment needs to authenticate at the deployment server using an SSH key pair.

Use `ssh-keygen` to create one, using no password.

- Add the private part of this ssh key pair to the CI environment, so that the deployment server can accept the pushed code. (e.g. as `~/.ssh/id_rsa`). In `deploy.yml`, we expect it as the secret `SSH_DEPLOYMENT_KEY`, which adds the key for us.
- Finally, the public part of the key pair should be in `~/.ssh/authorized_keys` on your deployment server.

(Re-)start FlexMeasures on the deployment server (install Post-Receive Hook)

Only pushing the code will not actually deploy the updated FlexMeasures into a usable web app on the deployment server. For this, we need to trigger a script.

Log on to the deployment server (via SSH) and install a script to (re-)start FlexMeasures as a Git Post Receive Hook in the remote repo where we deployed the code (see above). This hook will be triggered whenever a push is received from the deployment environment.

The example script below can be a Post Receive Hook (save as `hooks/post-receive` in your remote origin repo and update paths). It will force a checkout of the main branch into our working directory, update dependencies, upgrade the database structure and finally touch the `wsgi.py` file.

Note: Note that we are not installing FlexMeasures itself (that would require `make install-flexmeasures`, which essentially is `python setup.py develop`), as that is not needed for our base requirement here: to run this checked-out code with a web server that uses a WSGI file to define the app. Running CLI commands will not work without installation. Also, installing FlexMeasures requires a version, which is gotten from the git status (via `setuptools_scm`). We are working on a checked-out copy of the git code here without git meta information, so installing would fail anyways.

The last step, touching a `wsgi.py` file, is often used as a way to soft-restart the running application — here you need to adapt to your circumstances.

```
#!/bin/bash

PATH_TO_GIT_WORK_TREE=/path/to/where/you/want/to/checkout/code/to
ACTIVATE_VENV="command-to-activate-your-venv"
PATH_TO_WSGI=/path/to/wsgi/script/for/the/app

echo "CHECKING OUT CODE TO GIT WORK TREE ($PATH_TO_GIT_WORK_TREE) ..."
GIT_WORK_TREE=$PATH_TO_GIT_WORK_TREE git checkout -f

cd $PATH_TO_GIT_WORK_TREE
PATH=$PATH_TO_VENV/bin:$PATH

echo "INSTALLING DEPENDENCIES ..."
make install-deps

echo "UPGRADING DATABASE STRUCTURE ..."
make upgrade-db

echo "RESTARTING APPLICATION ..."
touch $PATH_TO_WSGI
```

A WSGI file can do various things, as well, but the simplest form is shown below.

```
from flexmeasures.app import create as create_app

application = create_app()
```

The web server is told about the WSGI script, but also about the object which represents the application. For instance, if this script is called `wsgi.py`, then the relevant argument to the gunicorn server is `wsgi:application`.

5.3.37 Custom authorization

Our *Authorization* section describes general authorization handling in FlexMeasures.

If you are creating your own API endpoints for a custom energy flexibility service (on top of FlexMeasures), you should also get your authorization right. It's recommended to get familiar with the decorators we provide. Here are some pointers, but feel free to read more in the `flexmeasures.auth` package.

In short, we recommend to use the `@permission_required_for_context` decorator (more explanation below).

FlexMeasures also supports role-based decorators, e.g. `@account_roles_required`. These authorization decorators are more straightforward to use than the `@permission_required_for_context` decorator. However, they are a bit crude as they do not distinguish on what the context is, nor do they qualify on the required permission (e.g. read versus write).¹

Finally, all decorators available through [Flask-Security-Too](#) can be used, e.g. `@auth_required` (that's technically only checking authentication) or `@permissions_required`.

Permission-based authorization

Via permissions, it's possible to define authorization access to data, distinguishing between create, read, update and delete access. It's a finer model than simply allowing per role.

The data models codify under which conditions a user can have certain permissions to work with their data. You, as the endpoint author, need to make sure this is checked. Here is an example (taken from the decorator docstring):

```
@app.route("/resource/<resource_id>", methods=["GET"])
@use_kwargs(
    {"the_resource": ResourceIdField(data_key="resource_id")},
    location="path",
)
@permission_required_for_context("read", arg_name="the_resource")
@as_json
def view(resource_id: int, resource: Resource):
    return dict(name=resource.name)
```

As you see, there is some sorcery with `@use_kwargs` going on before we check the permissions. That decorator is relaying to a [Marshmallow](#) field definition. Here, `ResourceIdField` is a definition which de-serializes an ID (passed in as a request parameter) into a `Resource` instance. This instance can then be asked if the current user may read it. That last part is what `@permission_required_for_context` is doing. You can find these Marshmallow fields in `flexmeasures.api.common.schemas`.

¹ Some authorization features are not possible for endpoints decorated in this way. For instance, we have an `admin-reader` role who should be able to read but not write everything — with only role-based decorators we can not allow this user to read (as we don't know what permission the endpoint requires).

Account roles

Another way to implement custom authorization is to define custom account roles. E.g. if several services run on one FlexMeasures server, each service could define a “MyService-subscriber” account role.

To make sure that only users of such accounts can use the endpoints:

```
@flexmeasures_ui.route("/bananas")
@account_roles_required("MyService-subscriber")
def bananas_view:
    pass
```

Note: This endpoint decorator lists required roles, so the authenticated user’s account needs to have each role. You can also use the `@account_roles_accepted` decorator. Then the user’s account only needs to have at least one of the roles.

User roles

There are also decorators to check user roles. Here is an example:

```
@flexmeasures_ui.route("/bananas")
@roles_required("account-admin")
def bananas_view:
    pass
```

Note: You can also use the `@roles_accepted` decorator.

5.3.38 Running a complete stack with docker-compose

To install FlexMeasures, plus the libraries and databases it depends on, on your computer is some work, and can have unexpected hurdles, e.g. depending on the operating system. A nice alternative is to let that happen within Docker. The whole stack can be run via [Docker compose](#), saving the developer much time.

For this, we assume you are in the directory (in the [FlexMeasures git repository](#)) housing `docker-compose.yml`.

Note: The minimum Docker version is 17.09 and for docker-compose we tested successfully at version 1.25. You can check your versions with `docker[-compose] --version`.

Note: The command might also be `docker compose` (no dash), for instance if you are using [Docker Desktop](#).

Build the compose stack

Run this:

```
$ docker-compose build
```

This pulls the images you need, and re-builds the FlexMeasures ones from code. If you change code, re-running this will re-build that image.

This compose script can also serve as an inspiration for using FlexMeasures in modern cloud environments (like Kubernetes). For instance, you might want to not build the FlexMeasures image from code, but simply pull the image from DockerHub.

If you wanted, you could stop building from source, and directly use the official flexmeasures image for the server and worker container (set `image: lfenergy/flexmeasures` in the file `docker-compose.yml`).

Run the compose stack

Start the stack like this:

```
$ docker-compose up
```

Warning: This might fail if ports 5000 (Flask) or 6379 (Redis) are in use on your system. Stop these processes before you continue.

Check `docker ps` or `docker-compose ps` to see if your containers are running:

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                                            CREATED        STATUS        PORTS                               NAMES
beb9bf567303   flexmeasures_server                "bash -c 'flexmeasur..."                     44 seconds ago Up 38s        0.0.0.0:5000->5000/tcp              flexmeasures-server-1
e36cd54a7fd5   flexmeasures_worker                "flexmeasures jobs r..."                     44 seconds ago Up 5s         5000/tcp                            flexmeasures-worker-1
c9985de27f68   postgres                            "docker-entrypoint.s..."                     45 seconds ago Up 40s        5432/tcp                            flexmeasures-test-db-1
03582d37230e   postgres                            "docker-entrypoint.s..."                     45 seconds ago Up 40s        5432/tcp                            flexmeasures-dev-db-1
792ec3d86e71   redis                               "docker-entrypoint.s..."                     45 seconds ago Up 40s        0.0.0.0:6379->6379/tcp              flexmeasures-queue-db-1
```

The FlexMeasures server container has a health check implemented, which is reflected in this output and you can see which ports are available on your machine to interact.

You can use the terminal or `docker-compose logs` to look at output. `docker inspect <container>` and `docker exec -it <container> bash` can be quite useful to dive into details. We'll see the latter more in this tutorial.

Configuration

You can pass in your own configuration (e.g. for MapBox access token, or db URI, see below) like we described in [Configuration and customization](#) — put a file `flexmeasures.cfg` into a local folder called `flexmeasures-instance` (the volume should be already mapped).

In case your configuration loads FlexMeasures plugins that have additional dependencies, you can add a `requirements.txt` file to the same local folder. The dependencies listed in that file will be freshly installed each time you run `docker-compose up`.

Data

The postgres database is a test database with toy data filled in when the `flexmeasures` container starts. You could also connect it to some other database (on your PC, in the cloud), by setting a different `SQLALCHEMY_DATABASE_URI` in the config.

Seeing it work: Running the toy tutorial

A good way to see if these containers work well together, and maybe to inspire how to use them for your own purposes, is the *Toy example: Scheduling a battery, from scratch*.

The `flexmeasures-server` container already creates the toy account when it starts (see its initial command). We'll now walk through the rest of the toy tutorial, with one twist at the end, when we create the battery schedule.

Let's go into the `flexmeasures-worker` container:

```
$ docker exec -it flexmeasures-worker-1 bash
```

There, we add the price data, as described in [Add some price data](#). Create the prices and add them to the FlexMeasures DB in the container's bash session.

Next, we put a scheduling job in the worker's queue. This only works because we have the Redis container running — the toy tutorial doesn't have it. The difference is that we're adding `--as-job`:

```
$ flexmeasures add schedule for-storage --sensor-id 1 --consumption-price-sensor 2 \
  --start ${TOMORROW}T07:00+01:00 --duration PT12H --soc-at-start 50% \
  --roundtrip-efficiency 90% --as-job
```

We should now see in the output of `docker logs flexmeasures-worker-1` something like the following:

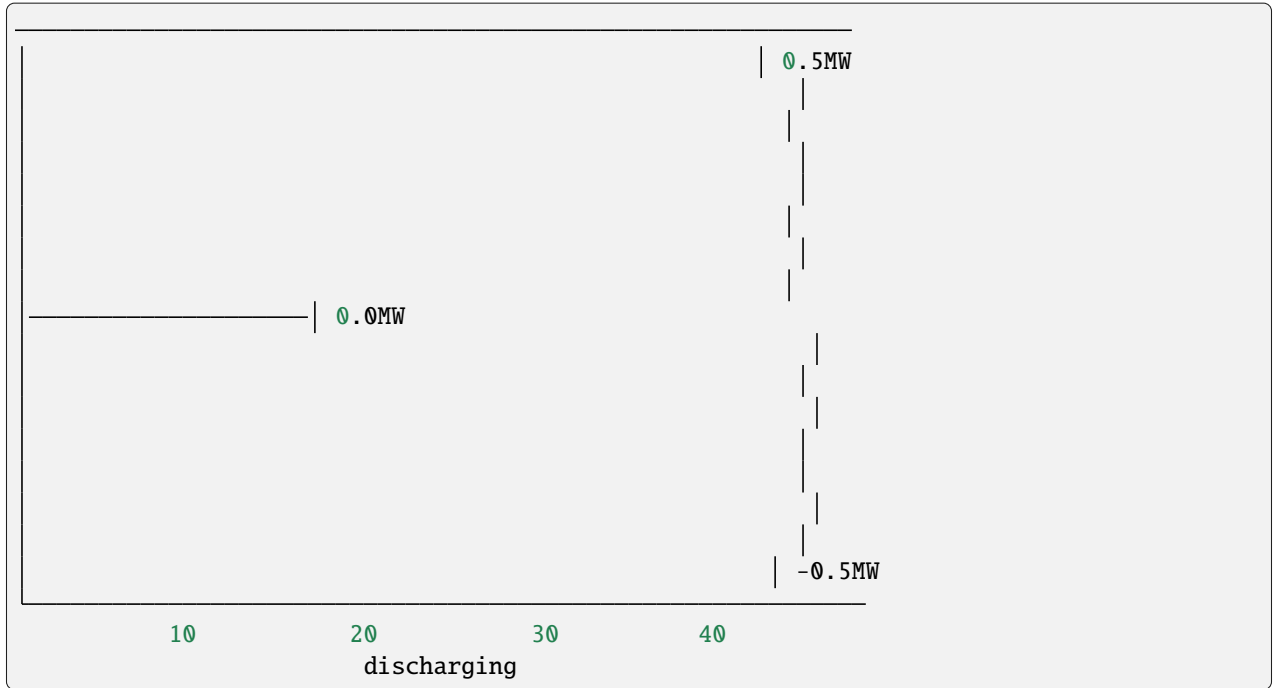
```
Running Scheduling Job d3e10f6d-31d2-46c6-8308-01ede48f8fdd: <Sensor 2: charging, unit:
↪MW res.: 0:15:00>, from 2022-07-06 07:00:00+01:00 to 2022-07-06 19:00:00+01:00
```

So the job had been queued in Redis, was then picked up by the worker process, and the result should be in our SQL database container. Let's check!

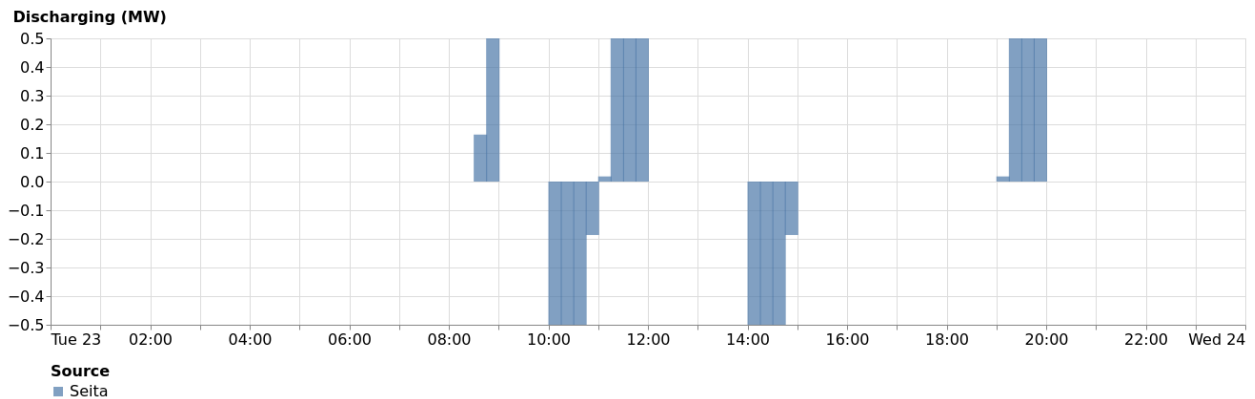
We'll not go into the server container this time, but simply send a command:

```
$ TOMORROW=$(date --date="next day" '+%Y-%m-%d')
$ docker exec -it flexmeasures-server-1 bash -c "flexmeasures show beliefs --sensor-id 1
↪--start ${TOMORROW}T07:00:00+01:00 --duration PT12H"
```

The charging/discharging schedule should be there:



Like in the original toy tutorial, we can also check in the server container’s web UI (username is “toy-user@flexmeasures.io”, password is “toy-password”):



Scripting with the Docker stack

A very important aspect of this stack is if it can be put to interesting use. For this, developers need to be able to script things — like we just did with the toy tutorial.

Note that instead of starting a console in the containers, we can also send commands to them right away. For instance, we sent the complete `flexmeasures show beliefs` command and then viewed the output on our own machine. Likewise, we send the `pytest` command to run the unit tests (see below).

Used this way, and in combination with the powerful list of *CLI Commands*, this FlexMeasures Docker stack is scriptable for interesting applications and simulations!

Running tests

You can run tests in the flexmeasures docker container, using the database service `test-db` in the compose file (per default, we are using the `dev-db` database service).

After you've started the compose stack with `docker-compose up`, run:

```
$ docker exec -it -e SQLALCHEMY_TEST_DATABASE_URI="postgresql://fm-test-db-user:fm-test-
db-pass@test-db:5432/fm-test-db" flexmeasures-server-1 pytest
```

This rounds up the dev experience offered by running FlexMeasures in Docker. Now you can develop FlexMeasures and also run your tests. If you develop plugins, you could extend the command being used, e.g. `bash -c "cd /path/to/my/plugin && pytest"`.

<code>flexmeasures.api</code>	FlexMeasures API routes and implementations.
<code>flexmeasures.app</code>	Starting point of the Flask application.
<code>flexmeasures.auth</code>	Authentication and authorization policies and helpers.
<code>flexmeasures.cli</code>	CLI functions for FlexMeasures hosts.
<code>flexmeasures.data</code>	Models & schemata, as well as business logic (queries & services).
<code>flexmeasures.ui</code>	Backoffice user interface & charting support.
<code>flexmeasures.utils</code>	Utilities for the FlexMeasures project.

5.3.39 flexmeasures.api

Modules

<code>flexmeasures.api.common</code>	Functionality common to all API versions.
<code>flexmeasures.api.dev</code>	Endpoints under development.
<code>flexmeasures.api.play</code>	Endpoints to support "play" mode, data restoration
<code>flexmeasures.api.sunset</code>	A place to keep all routes to endpoints that previously existed and are now sunset.
<code>flexmeasures.api.v3_0</code>	FlexMeasures API v3

flexmeasures.api.common

Modules

<code>flexmeasures.api.common.implementations</code>
<code>flexmeasures.api.common.responses</code>
<code>flexmeasures.api.common.routes</code>
<code>flexmeasures.api.common.schemas</code>
<code>flexmeasures.api.common.utils</code>

flexmeasures.api.common.implementations

Functions

`flexmeasures.api.common.implementations.get_task_run()`

Get latest task runs. This endpoint returns output conforming to the task monitoring tool (bobbydams/py-pinger)

`flexmeasures.api.common.implementations.ping()`

`flexmeasures.api.common.implementations.post_task_run()`

Post that a task has been (attempted to) run. Form fields to send in: name: str, status: bool [defaults to True], datetime: datetime [defaults to now]

flexmeasures.api.common.responses

Functions

`flexmeasures.api.common.responses.already_received_and_successfully_processed(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.conflicting_resolutions(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.deprecated_api_version(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.incomplete_event(requested_event_id, requested_event_type, message) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_datetime(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_domain(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_flex_config(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_horizon(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_market() → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_message_type(message_type: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_method(request_method) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_period(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_ptu_duration(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_replacement(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_resolution_str(message: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_role(requested_access_role: str) → Tuple[dict, int]`

`flexmeasures.api.common.responses.invalid_sender(required_permissions: List[str] | None = None) → Tuple[dict, int]`

Signify that the sender is invalid to perform the request. Fits well with 403 errors. Optionally tell the user which permissions they should have.

```

flexmeasures.api.common.responses.invalid_source(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.invalid_timezone(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.invalid_unit(quantity: str | None, units: Sequence[str] | Tuple[str] |
None) → Tuple[dict, int]

flexmeasures.api.common.responses.is_response_tuple(value) → bool
    Check if an object qualifies as a ResponseTuple
flexmeasures.api.common.responses.no_backup(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.no_message_type(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.outdated_event_id(requested_event_id, existing_event_id) →
    Tuple[dict, int]

flexmeasures.api.common.responses.pluralize(usef_role_name: str) → str
    Adding a trailing 's' works well for USEF roles.
flexmeasures.api.common.responses.power_value_too_big(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.power_value_too_small(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.ptus_incomplete(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.request_processed(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.required_info_missing(fields: Sequence[str], message: str = "") →
    Tuple[dict, int]

flexmeasures.api.common.responses.unapplicable_resolution(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unknown_prices(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unknown_schedule(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_asset(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_backup(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_connection_group(message: str) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_event(requested_event_id, requested_event_type) →
    Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_event_type(requested_event_type) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_market(requested_market) → Tuple[dict, int]
flexmeasures.api.common.responses.unrecognized_sensor(lat: float | None = None, lng: float | None =
    None) → Tuple[dict, int]

```

Classes

class flexmeasures.api.common.responses.BaseMessage(*base_message=""*)

Set a base message to which extra info can be added by calling the wrapped function with additional string arguments. This is a decorator implemented as a class.

__init__(*base_message=""*)

flexmeasures.api.common.routes

Functions

flexmeasures.api.common.routes.get_ping()

flexmeasures.api.common.routes.get_task_run()

flexmeasures.api.common.routes.post_task_run()

flexmeasures.api.common.schemas

Modules

<i>flexmeasures.api.common.schemas. generic_assets</i>
<i>flexmeasures.api.common.schemas. sensor_data</i>
<i>flexmeasures.api.common.schemas.sensors</i>
<i>flexmeasures.api.common.schemas.users</i>

flexmeasures.api.common.schemas.generic_assets

Classes

class flexmeasures.api.common.schemas.generic_assets.AssetIdField(*, *strict: bool = False*,
***kwargs*)

Field that represents a generic asset ID. It de-serializes from the asset id to an asset instance.

_deserialize(*asset_id: int, attr, obj, **kwargs*) → *GenericAsset*

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in *data* to be deserialized.
- **data** – The raw input data passed to the *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

Changed in version 3.0.0: Added `**kwargs` to signature.

_serialize(*asset*: [GenericAsset](#), *attr*, *data*, ***kwargs*) → `int`

Return a string if *self.as_string*=*True*, otherwise return this field's *num_type*.

flexmeasures.api.common.schemas.sensor_data**Functions**

`flexmeasures.api.common.schemas.sensor_data.select_schema_to_ensure_list_of_floats`(*values*: *list*[*float*] | *float*, *_*) → *fields.List* | *Single-Value-Field*

Allows both a single float and a list of floats. Always returns a list of floats.

Meant to improve user experience by not needing to make a list out of a single item, such that:

```
{
  "values": [3.7]
}
```

can be written as:

```
{
  "values": 3.7
}
```

Either will be de-serialized to [3.7].

Note that serialization always results in a list of floats. This ensures that we are not requiring the same flexibility from users who are retrieving data.

Classes

```
class flexmeasures.api.common.schemas.sensor_data.GetSensorDataSchema(*, only:
    types.StrSequenceOrSet |
    None = None, exclude:
    types.StrSequenceOrSet
    = (), many: bool = False,
    context: dict | None =
    None, load_only:
    types.StrSequenceOrSet
    = (), dump_only:
    types.StrSequenceOrSet
    = (), partial: bool |
    types.StrSequenceOrSet
    = False, unknown: str |
    None = None)
```

dump_bdf(*sensor_data_description*: dict, ***kwargs*) → dict

Turn the de-serialized and validated data description into a response.

Specifically, this function: - queries data according to the given description - converts to a single deterministic belief per event - ensures the response respects the requested time frame - converts values to the requested unit - converts values to the requested resolution

```
class flexmeasures.api.common.schemas.sensor_data.PostSensorDataSchema(*, only:
    types.StrSequenceOrSet
    | None = None, exclude:
    types.StrSequenceOrSet
    = (), many: bool =
    False, context: dict |
    None = None,
    load_only:
    types.StrSequenceOrSet
    = (), dump_only:
    types.StrSequenceOrSet
    = (), partial: bool |
    types.StrSequenceOrSet
    = False, unknown: str |
    None = None)
```

This schema includes data, so it can be used for POST requests or GET responses.

TODO: For the GET use case, look at `api/common/validators.py::get_data_downsampling_allowed`
(sets a resolution parameter which we can pass to the data collection function).

check_resolution_compatibility_of_sensor_data(*data*, ***kwargs*)

Ensure event frequency is compatible with the sensor's event resolution.

For a sensor recording instantaneous values, any event frequency is compatible. For a sensor recording non-instantaneous values, the event frequency must fit the sensor's event resolution. Currently, only upsampling is supported (e.g. converting hourly events to 15-minute events).

static load_bdf(*sensor_data*: dict) → BeliefsDataFrame

Turn the de-serialized and validated data into a BeliefsDataFrame.

static possibly_convert_units(*data*)

Convert values if needed, to fit the sensor's unit. Marshmallow runs this after validation.

static possibly_upsample_values(*data*)

Upsample the data if needed, to fit to the sensor's resolution. Marshmallow runs this after validation.

post_load_sequence(*data*: *dict*, ***kwargs*) → BeliefsDataFrame

If needed, upsample and convert units, then deserialize to a BeliefsDataFrame.

```
class flexmeasures.api.common.schemas.sensor_data.SensorDataDescriptionSchema(*, only:
    types.StrSequenceOrSet
    | None =
    None, exclude:
    types.StrSequenceOrSet
    = (), many:
    bool = False,
    context: dict |
    None = None,
    load_only:
    types.StrSequenceOrSet
    = (),
    dump_only:
    types.StrSequenceOrSet
    = (), partial:
    bool |
    types.StrSequenceOrSet
    = False,
    unknown: str |
    None = None)
```

Schema describing sensor data (specifically, the sensor and the timing of the data).

check_schema_unit_against_sensor_unit(*data*, ***kwargs*)

Allows units compatible with that of the sensor. For example, a sensor with W units allows data to be posted with units: - W, kW, MW, etc. (i.e. units with different prefixes) - J/s, Nm/s, etc. (i.e. units that can be converted using some multiplier) - Wh, kWh, etc. (i.e. units that represent a stock delta, which knowing the duration can be converted to a flow) For compatible units, the SensorDataSchema converts values to the sensor's unit.

```
class flexmeasures.api.common.schemas.sensor_data.SingleValueField(*, allow_nan: bool = False,
    as_string: bool = False,
    **kwargs)
```

Field that both de-serializes and serializes a single value to a list of floats (length 1).

_deserialize(*value*, *attr*, *obj*, ***kwargs*) → list[float]

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in *data* to be deserialized.
- **data** – The raw input data passed to the *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added *attr* and *data* parameters.

Changed in version 3.0.0: Added ***kwargs* to signature.

`_serialize(value, attr, data, **kwargs) → list[float]`

Return a string if `self.as_string=True`, otherwise return this field's `num_type`.

flexmeasures.api.common.schemas.sensors

Classes

class flexmeasures.api.common.schemas.sensors.SensorField(entity_type: str, fm_scheme: str, *args, **kwargs)

Field that de-serializes to a Sensor, and serializes a Sensor, Asset, Market or WeatherSensor into an entity address (string).

`__init__(entity_type: str, fm_scheme: str, *args, **kwargs)`

Parameters

- **entity_type** – “sensor”, “connection”, “market” or “weather_sensor”
- **fm_scheme** – “fm0” or “fm1”

`_deserialize(value, attr, obj, **kwargs) → Sensor`

De-serialize to a Sensor.

`_serialize(value: Sensor, attr, data, **kwargs)`

Serialize to an entity address.

class flexmeasures.api.common.schemas.sensors.SensorIdField(*, strict: bool = False, **kwargs)

Field that represents a sensor ID. It de-serializes from the sensor id to a sensor instance.

`_deserialize(sensor_id: int, attr, obj, **kwargs) → Sensor`

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in `data` to be deserialized.
- **data** – The raw input data passed to the `Schema.load`.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

Changed in version 3.0.0: Added `**kwargs` to signature.

`_serialize(sensor: Sensor, attr, data, **kwargs) → int`

Return a string if `self.as_string=True`, otherwise return this field's `num_type`.

Exceptions

```
exception flexmeasures.api.common.schemas.sensors.EntityAddressValidationError(message: str
| list | dict,
field_name:
str =
'_schema',
data:
Mapping[str,
Any] | Iter-
able[Mapping[str,
Any]] | None
= None,
valid_data:
list[dict[str,
Any]] |
dict[str, Any]
| None =
None,
**kwargs)
```

flexmeasures.api.common.schemas.users

Classes

```
class flexmeasures.api.common.schemas.users.AccountIdField(*, strict: bool = False, **kwargs)
```

Field that represents an account ID. It deserializes from the account id to an account instance.

```
_deserialize(account_id: str, attr, obj, **kwargs) → Account
```

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in *data* to be deserialized.
- **data** – The raw input data passed to the *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added *attr* and *data* parameters.

Changed in version 3.0.0: Added ***kwargs* to signature.

```
_serialize(account: Account, attr, data, **kwargs) → int
```

Return a string if *self.as_string=True*, otherwise return this field's *num_type*.

```
classmethod load_current()
```

Use this with the *load_default* arg to *__init__* if you want the current user's account by default.

class flexmeasures.api.common.schemas.users.UserIdField(*args, **kwargs)

Field that represents a user ID. It deserializes from the user id to a user instance.

__init__(*args, **kwargs)

_deserialize(user_id: *int*, attr, obj, **kwargs) → *User*

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in *data* to be deserialized.
- **data** – The raw input data passed to the *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added **attr** and **data** parameters.

Changed in version 3.0.0: Added ****kwargs** to signature.

_serialize(user: *User*, attr, data, **kwargs) → *int*

Return a string if *self.as_string=True*, otherwise return this field's *num_type*.

flexmeasures.api.common.utils

Modules

flexmeasures.api.common.utils.api_utils

flexmeasures.api.common.utils.args_parsing

flexmeasures.api.common.utils.decorators

flexmeasures.api.common.utils.deprecation_utils

flexmeasures.api.common.utils.migration_utils

flexmeasures.api.common.utils.validators

This module is part of our data model migration (see <https://github.com/SeitaBV/flexmeasures/projects/9>).

flexmeasures.api.common.utils.api_utils**Functions**

`flexmeasures.api.common.utils.api_utils.append_doc_of(fun)`

`flexmeasures.api.common.utils.api_utils.asset_replace_name_with_id(connections_as_name: List[str]) → List[str]`

Look up the owner and id given the asset name and construct a type 1 USEF entity address.

`flexmeasures.api.common.utils.api_utils.catch_timed_belief_replacements(error: IntegrityError)`
 Catch IntegrityErrors due to a UniqueViolation on the TimedBelief primary key.

Return a more informative message.

`flexmeasures.api.common.utils.api_utils.contains_empty_items(groups: List[List[str]])`

Return True if any of the items in the groups is empty.

`flexmeasures.api.common.utils.api_utils.determine_belief_timing(event_values: list, start: datetime, resolution: timedelta, horizon: timedelta, prior: datetime, sensor: Sensor) → Tuple[List[datetime], List[timedelta]]`

Determine event starts from start, resolution and len(event_values), and belief horizons from horizon, prior, or both, taking into account the sensor's knowledge horizon function.

In case both horizon and prior is set, we take the greatest belief horizon, which represents the earliest belief time.

`flexmeasures.api.common.utils.api_utils.enqueue_forecasting_jobs(forecasting_jobs: list[Job] | None = None)`

Enqueue forecasting jobs.

Parameters

forecasting_jobs – list of forecasting Jobs for redis queues.

`flexmeasures.api.common.utils.api_utils.get_form_from_request(_request) → dict | None`

`flexmeasures.api.common.utils.api_utils.get_sensor_by_generic_asset_type_and_location(generic_asset_type_name: str, latitude: float = 0, longitude: float = 0) → Sensor | Tuple[dict, int]`

Search a sensor by generic asset type and location. Can create a sensor if needed (depends on API mode) and then inform the requesting user which one to use.

```
flexmeasures.api.common.utils.api_utils.groups_to_dict(connection_groups: List[str], value_groups:
List[List[str]], generic_asset_type_name:
str, plural_name: str | None = None,
groups_name='groups') → dict
```

Put the connections and values in a dictionary and simplify if groups have identical values and/or if there is only one group.

Examples:

```
>> connection_groups = [[1]] >> value_groups = [[300, 300, 300]] >> response_dict =
groups_to_dict(connection_groups, value_groups, "connection") >> print(response_dict) << {
    "connection": 1, "values": [300, 300, 300]
}

>> connection_groups = [[1], [2]] >> value_groups = [[300, 300, 300], [300, 300, 300]]
>> response_dict = groups_to_dict(connection_groups, value_groups, "connection") >>
print(response_dict) << {
    "connections": [1, 2], "values": [300, 300, 300]
}

>> connection_groups = [[1], [2]] >> value_groups = [[300, 300, 300], [400, 400, 400]]
>> response_dict = groups_to_dict(connection_groups, value_groups, "connection") >>
print(response_dict) << {
    "groups": [
        {
            "connection": 1, "values": [300, 300, 300]
        }, {
            "connection": 2, "values": [400, 400, 400]
        }
    ]
}
```

```
flexmeasures.api.common.utils.api_utils.list_access(service_listing, service_name)
```

For a given USEF service name (API endpoint) in a service listing, return the list of USEF roles that are allowed to access the service.

```
flexmeasures.api.common.utils.api_utils.message_replace_name_with_ea(message_with_connections_as_asset_names:
dict) → dict
```

For each connection in the message specified by a name, replace that name with the correct entity address. TODO: Deprecated. This function is now only used in tests of deprecated API versions and should go (also `asset_replace_name_with_id`)

```
flexmeasures.api.common.utils.api_utils.parse_as_list(connection: str | float | Sequence[str | float],
of_type: type | None = None) → Sequence[str
| float | None]
```

Return a list of connections (or values), even if it's just one connection (or value)


```
flexmeasures.api.common.utils.api_utils.save_and_enqueue(data: BeliefsDataFrame |
                                                         List[BeliefsDataFrame], forecasting_jobs:
                                                         list[Job] | None = None,
                                                         save_changed_beliefs_only: bool = True)
                                                         → ResponseTuple
```

```
flexmeasures.api.common.utils.api_utils.save_to_db(timed_values: BeliefsDataFrame | List[Power |
                                                                 Price | Weather], forecasting_jobs: List[Job] = [],
                                                                 save_changed_beliefs_only: bool = True) →
                                                                 Tuple[dict, int]
```

Put the timed values into the database and enqueue forecasting jobs.

Data can only be replaced on servers in play mode.

TODO: remove this legacy function in its entirety (announced v0.8.0)

Parameters

- **timed_values** – BeliefsDataFrame or a list of Power, Price or Weather values to be saved
- **forecasting_jobs** – list of forecasting Jobs for redis queues.
- **save_changed_beliefs_only** – if True, beliefs that are already stored in the database with an earlier belief time are dropped.

Returns

ResponseTuple

```
flexmeasures.api.common.utils.api_utils.unique_ever_seen(iterable: Sequence, selector: Sequence)
```

Return unique iterable elements with corresponding lists of selector elements, preserving order.

```
>>> a, b = unique_ever_seen([[10, 20], [10, 20], [20, 40]], [1, 2, 3])
>>> print(a)
[[10, 20], [20, 40]]
>>> print(b)
[[1, 2], 3]
```

```
flexmeasures.api.common.utils.api_utils.upsample_values(value_groups: List[List[float]] | List[float],
                                                         from_resolution: timedelta, to_resolution:
                                                         timedelta) → List[List[float]] | List[float]
```

Upsample the values (in value groups) to a smaller resolution. from_resolution has to be a multiple of to_resolution

flexmeasures.api.common.utils.args_parsing

Functions

```
flexmeasures.api.common.utils.args_parsing.handle_error(error, req, schema, *, error_status_code,
                                                         error_headers)
```

Replacing webargs’s error parser, so we can throw custom Exceptions.

```
flexmeasures.api.common.utils.args_parsing.load_data(request, schema)
```

We allow parameters to come from either GET args or POST JSON, as validators can be attached to either.

```
flexmeasures.api.common.utils.args_parsing.validation_error_handler(error: FMValidationError)
```

Handles errors during parsing. Aborts the current HTTP request and responds with a 422 error. FMValidationError attributes “result” and “status” are packaged in the response.

flexmeasures.api.common.utils.decorators

Functions

`flexmeasures.api.common.utils.decorators.as_response_type(response_type)`

Decorator which adds a “type” parameter to the data of the flask response. Example:

```
@app.route('/postMeterData') @as_response_type("PostMeterDataResponse") @as_json def
post_meter_data() -> dict:

    return {"message": "Meter data posted"}
```

The response.json will be:

```
{
    "message": "Meter data posted", "type": "PostMeterDataResponse"
}
```

Parameters

response_type – The response type.

`flexmeasures.api.common.utils.decorators.split_response(response: Response) → tuple[dict, int, dict]`

Split Flask Response object into json data, status code and headers.

flexmeasures.api.common.utils.deprecation_utils

Functions

`flexmeasures.api.common.utils.deprecation_utils.deprecate_blueprint(blueprint: Blueprint, deprecation_date: pd.Timestamp | str | None = None, deprecation_link: str | None = None, sunset_date: pd.Timestamp | str | None = None, sunset_link: str | None = None, **kwargs)`

Deprecates every route on a blueprint by adding the “Deprecation” header with a deprecation date.

Also logs a warning when a deprecated endpoint is called.

```
>>> from flask import Flask, Blueprint
>>> app = Flask('some_app')
>>> deprecated_bp = Blueprint('API version 1', 'v1_bp')
>>> app.register_blueprint(deprecated_bp, url_prefix='/v1')
>>> deprecate_blueprint(
    deprecated_bp,
    deprecation_date="2022-12-14",
    deprecation_link="https://flexmeasures.readthedocs.io/some-deprecation-
↪notice",
    sunset_date="2023-02-01",
    sunset_link="https://flexmeasures.readthedocs.io/some-sunset-notice",
)
```

Parameters

- **blueprint** – The blueprint to be deprecated
- **deprecation_date** – date indicating when the API endpoint was deprecated, used for the “Deprecation” header if no date is given, defaults to “true” see <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header#section-2-1>
- **deprecation_link** – url providing more information about the deprecation
- **sunset_date** – date indicating when the API endpoint is likely to become unresponsive
- **sunset_link** – url providing more information about the sunset

References

- Deprecation header: <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header>
- Sunset header: <https://www.rfc-editor.org/rfc/rfc8594>

```
flexmeasures.api.common.utils.deprecation_utils.deprecate_fields(fields: str | list[str],
                                                                deprecation_date:
                                                                pd.Timestamp | str | None =
                                                                None, deprecation_link: str |
                                                                None = None, sunset_date:
                                                                pd.Timestamp | str | None =
                                                                None, sunset_link: str | None =
                                                                None)
```

Deprecates a field (or fields) on a route by adding the “Deprecation” header with a deprecation date.

Also logs a warning when a deprecated field is used.

```
>>> from flask_classful import route
>>> @route("/item/", methods=["POST"])
    @use_kwargs(
        {
            "color": ColorField,
            "length": LengthField,
        }
    )
    def post_item(color, length):
        deprecate_field(
            "color",
            deprecation_date="2022-12-14",
            deprecation_link="https://flexmeasures.readthedocs.io/some-deprecation-
↪notice",
            sunset_date="2023-02-01",
            sunset_link="https://flexmeasures.readthedocs.io/some-sunset-notice",
        )
```

Parameters

- **fields** – The fields (as a list of strings) to be deprecated
- **deprecation_date** – date indicating when the field was deprecated, used for the “Deprecation” header if no date is given, defaults to “true” see <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header#section-2-1>

- **deprecation_link** – url providing more information about the deprecation
- **sunset_date** – date indicating when the field is likely to become unresponsive
- **sunset_link** – url providing more information about the sunset

References

- Deprecation header: <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header>
- Sunset header: <https://www.rfc-editor.org/rfc/rfc8594>

`flexmeasures.api.common.utils.deprecation_utils.override_from_config(setting: Any, config_setting_name: str) → Any`

Override setting by config setting, unless the latter is None or is missing.

`flexmeasures.api.common.utils.deprecation_utils.sunset_blueprint(blueprint, api_version_being_sunset: str, sunset_link: str, api_version_upgrade_to: str = '3.0', rollback_possible: bool = True, **kwargs)`

Sunset every route on a blueprint by returning 410 (Gone) responses, if sunset is active.

Whether the sunset is active can be toggled using the config setting “FLEXMEASURES_API_SUNSET_ACTIVE”. If the sunset is inactive, this function will not affect any requests in this blueprint. If the endpoint implementations have been removed, set `rollback_possible=False`.

Errors will be logged by `utils.error_utils.error_handling_router`.

`flexmeasures.api.common.utils.migration_utils`

This module is part of our data model migration (see <https://github.com/SeitaBV/flexmeasures/projects/9>). It will become obsolete when we deprecate the fm0 scheme for entity addresses.

Functions

`flexmeasures.api.common.utils.migration_utils.get_sensor_by_unique_name(sensor_name: str, generic_asset_type_names: List[str] | None = None) → Sensor | Tuple[dict, int]`

Search a sensor by unique name, returning a `ResponseTuple` if it is not found.

Optionally specify a list of generic asset type names to filter on. This function should be used only for sensors that correspond to the old Market class.

flexmeasures.api.common.utils.validators

Functions

`flexmeasures.api.common.utils.validators.assets_required(generic_asset_type_name: str, plural_name: str | None = None, groups_name='groups')`

Decorator which specifies that a GET or POST request must specify one or more assets. It parses relevant form data and sets the “generic_asset_name_groups” keyword param. Example:

```
@app.route('/postMeterData') @assets_required("connection", plural_name="connections") def
post_meter_data(generic_asset_name_groups):

    return 'Meter data posted'
```

Given this example, the message must specify one or more assets as “connections”. If that is the case, then the assets are passed to the function as `generic_asset_name_groups`.

Connections can be listed in one of the following ways: - value of ‘connection’ key (for a single asset) - values of ‘connections’ key (for multiple assets that have the same timeseries data) - values of the ‘connection’ and/or ‘connections’ keys listed under the ‘groups’ key

(for multiple assets with different timeseries data)

`flexmeasures.api.common.utils.validators.get_data_downsampling_allowed(entity_type: str, fm_scheme: str)`

Decorator which allows downsampling of data which a GET request returns. It checks for a form parameter “resolution”. If that is given and is a multiple of the sensor’s event_resolution, downsampling is performed on the data. This is done by setting the “resolution” keyword parameter, which is obeyed by `collect_time_series_data` and used in resampling.

The original resolution of the data is the event_resolution of the sensor. Therefore, the decorator should follow after the `assets_required` decorator.

Example:

```
@app.route('/getMeterData') @assets_required("connection") @get_data_downsampling_allowed("connection")
def get_meter_data(generic_asset_name_groups, resolution):

    return data
```

`flexmeasures.api.common.utils.validators.include_current_user_source_id(source_ids: List[int]) → List[int]`

Includes the source id of the current user.

`flexmeasures.api.common.utils.validators.optional_duration_accepted(default_duration: timedelta)`

Decorator which specifies that a GET or POST request accepts an optional duration. It parses relevant form data and sets the “duration” keyword param.

Example:

```
@app.route('/getDeviceMessage') @optional_duration_accepted(timedelta(hours=6)) def
get_device_message(duration):

    return 'Here is your message'
```

The message may specify a duration to overwrite the default duration of 6 hours.

```
flexmeasures.api.common.utils.validators.optional_horizon_accepted(ex_post: bool = False,
                                                                    infer_missing: bool = True,
                                                                    infer_missing_play: bool =
                                                                    False,
                                                                    accept_repeating_interval:
                                                                    bool = False)
```

Decorator which specifies that a GET or POST request accepts an optional horizon. The horizon should be in accordance with the ISO 8601 standard. It parses relevant form data and sets the “horizon” keyword param (a timedelta).

Interpretation for GET requests: - Denotes “at least <horizon> before the fact (positive horizon),
or at most <horizon> after the fact (negative horizon)”

- This results in the filter `belief_horizon_window = (horizon, None)`

Interpretation for POST requests: - Denotes “at <horizon> before the fact (positive horizon),
or at <horizon> after the fact (negative horizon)”

- this results in the assignment `belief_horizon = horizon`

For example:

```
@app.route('/postMeterData') @optional_horizon_accepted() def post_meter_data(horizon):
    return 'Meter data posted'
```

Parameters

- **ex_post** – if True, only non-positive horizons are allowed.
- **infer_missing** – if True, servers assume that the `belief_horizon` of posted values is 0 hours. This setting is meant to be used for POST requests.
- **infer_missing_play** – if True, servers in play mode assume that the `belief_horizon` of posted values is 0 hours. This setting is meant to be used for POST requests.
- **accept_repeating_interval** – if True, the “rolling” keyword param is also set (this was used for POST requests before v2.0)

```
flexmeasures.api.common.utils.validators.optional_prior_accepted(ex_post: bool = False,
                                                                    infer_missing: bool = True,
                                                                    infer_missing_play: bool =
                                                                    False)
```

Decorator which specifies that a GET or POST request accepts an optional prior. It parses relevant form data and sets the “prior” keyword param.

Interpretation for GET requests: - Denotes “at least before <prior>” - This results in the filter `belief_time_window = (None, prior)`

Interpretation for POST requests: - Denotes “recorded <prior> to some datetime, - this results in the assignment `belief_time = prior`

Parameters

- **ex_post** – if True, only ex-post datetimes are allowed.
- **infer_missing** – if True, servers assume that the `belief_time` of posted values is server time. This setting is meant to be used for POST requests.

- **infer_missing_play** – if True, servers in play mode assume that the belief_time of posted values is server time. This setting is meant to be used for POST requests.

```
flexmeasures.api.common.utils.validators.optional_user_sources_accepted(default_source: int |
                                                                    str | list[int | str] |
                                                                    None = None)
```

Decorator which specifies that a GET or POST request accepts an optional source or list of data sources. It parses relevant form data and sets the “user_source_ids” keyword parameter.

Data originating from the requesting user is included by default. That is, user_source_ids always includes the source id of the requesting user.

Each source should either be a known USEF role name or a user id. We’ll parse them as a list of source ids.

Case 1: If a request states one or more data sources, then we’ll only query those, in addition to the user’s own data. Default sources specified in the decorator (see example below) are ignored.

Case 2: If a request does not state any data sources, a list of default sources will be used.

Case 2A: Default sources can be specified in the decorator (see example below).

Case 2B: If no default sources are specified in the decorator, all sources are included.

Example:

```
@app.route('/getMeterData')                @optional_sources_accepted("MDC")                def
get_meter_data(user_source_ids):
    return 'Meter data posted'
```

The source ids then include the user’s own id, and ids of other users that are registered as a Meter Data Company.

If the message specifies:

```
{
  "sources": ["Prosumer", "ESCo"]
}
```

The source ids then include the user’s own id, and ids of other users whose organisation account is registered as a Prosumer and/or Energy Service Company.

```
flexmeasures.api.common.utils.validators.parse_duration(duration_str: str, start: datetime | None =
                                                         None) → timedelta | Duration | None
```

Parses the ‘duration’ string into a Duration object. If needed, try deriving the timedelta from the actual time span (e.g. in case duration is 1 year). If the string is not a valid ISO 8601 time interval, return None.

TODO: Deprecate for DurationField.

```
flexmeasures.api.common.utils.validators.parse_horizon(horizon_str: str) → Tuple[timedelta | None,
                                                                                    bool]
```

Validates whether a horizon string represents a valid ISO 8601 (repeating) time interval.

Examples:

```
horizon = "PT6H" horizon = "R/PT6H" horizon = "-PT10M"
```

Returns horizon as timedelta and a boolean indicating whether the repetitive indicator “R/” was used. If horizon_str could not be parsed with various methods, then horizon will be None

```
flexmeasures.api.common.utils.validators.parse_isodate_str(start: str) → datetime | None
```

Validates whether the string ‘start’ is a valid ISO 8601 datetime.

`flexmeasures.api.common.utils.validators.period_required(fn)`

Decorator which specifies that a GET or POST request must specify a time period (by start and duration). It parses relevant form data and sets the “start” and “duration” keyword params. Example:

```
@app.route('/postMeterData') @period_required def post_meter_data(period):
    return 'Meter data posted'
```

The message must specify a ‘start’ and a ‘duration’ in accordance with the ISO 8601 standard. This decorator should not be used together with `optional_duration_accepted`.

`flexmeasures.api.common.utils.validators.post_data_checked_for_required_resolution(entity_type: str, fn_scheme: str)`

Decorator which checks that a POST request receives time series data with the event resolutions required by the sensor. It sets the “resolution” keyword argument. If the resolution in the data is a multiple of the sensor resolution, values are upsampled to the sensor resolution. Finally, this decorator also checks if all sensors have the same `event_resolution` and complains otherwise.

The resolution of the data is inferred from the duration and the number of values. Therefore, the decorator should follow after the `values_required`, `period_required` and `assets_required` decorators. Example:

```
@app.route('/postMeterData') @values_required @period_required @as-
sets_required("connection") @post_data_checked_for_required_resolution("connection") def
post_meter_data(value_groups, start, duration, generic_asset_name_groups, resolution)
    return 'Meter data posted'
```

`flexmeasures.api.common.utils.validators.type_accepted(message_type: str)`

Decorator which specifies that a GET or POST request must specify the specified message type. Example:

```
@app.route('/postMeterData') @type_accepted('PostMeterDataRequest') def post_meter_data():
    return 'Meter data posted'
```

The message must specify ‘PostMeterDataRequest’ as its ‘type’.

Parameters

message_type – The message type.

`flexmeasures.api.common.utils.validators.unit_required(fn)`

Decorator which specifies that a GET or POST request must specify a unit. It parses relevant form data and sets the “unit” keyword param. Example:

```
@app.route('/postMeterData') @unit_required def post_meter_data(unit):
    return 'Meter data posted'
```

The message must specify a ‘unit’.

`flexmeasures.api.common.utils.validators.units_accepted(quantity: str, *units: str)`

Decorator which specifies that a GET or POST request must specify one of the specified physical units. First parameter specifies the physical or economical quantity. It parses relevant form data and sets the “unit” keyword param. Example:

```
@app.route('/postMeterData') @units_accepted("power", 'MW', 'MWh') def
post_meter_data(unit):
    return 'Meter data posted'
```

The message must either specify ‘MW’ or ‘MWh’ as the unit.

Parameters

- **quantity** – The physical or economic quantity
- **units** – The possible units.

`flexmeasures.api.common.utils.validators.valid_sensor_units(sensor: str) → List[str]`

Returns the accepted units for this sensor.

`flexmeasures.api.common.utils.validators.validate_user_sources(sources: int | str | List[int | str]) → List[int]`

Return a list of user-based data source ids, given: - one or more user ids - one or more account role names

`flexmeasures.api.common.utils.validators.values_required(fn)`

Decorator which specifies that a GET or POST request must specify one or more values. It parses relevant form data and sets the “value_groups” keyword param. Example:

```
@app.route('/postMeterData') @values_required def post_meter_data(value_groups):
    return 'Meter data posted'
```

The message must specify one or more values. If that is the case, then the values are passed to the function as value_groups.

Functionality common to all API versions.

Functions

`flexmeasures.api.common.register_at(app: Flask)`

This can be used to register this blueprint together with other api-related things

flexmeasures.api.dev**Modules**

flexmeasures.api.dev.sensors

flexmeasures.api.dev.sensors**Functions**

`flexmeasures.api.dev.sensors.get_sensor_or_abort(id: int) → Sensor`

Util function to help the GET requests. Will be obsolete..

Classes

`class flexmeasures.api.dev.sensors.AssetAPI`

This view exposes asset attributes through API endpoints under development. These endpoints are not yet part of our official API, but support the FlexMeasures UI.

`get(id: int, asset: GenericAsset)`

GET from /asset/<id>

`class flexmeasures.api.dev.sensors.SensorAPI`

This view exposes sensor attributes through API endpoints under development. These endpoints are not yet part of our official API, but support the FlexMeasures UI.

`get(id: int, sensor: Sensor)`

GET from /sensor/<id>

`get_chart(id: int, sensor: Sensor, **kwargs)`

GET from /sensor/<id>/chart

Optional fields

- “event_starts_after” (see the [timely-beliefs documentation](#))
- “event_ends_before” (see the [timely-beliefs documentation](#))
- “beliefs_after” (see the [timely-beliefs documentation](#))
- “beliefs_before” (see the [timely-beliefs documentation](#))
- “include_data” (if true, chart specs include the data; if false, use the [GET /api/dev/sensor/\(id\)/chart_data/](#) endpoint to fetch data)
- “width” (an integer number of pixels; without it, the chart will be scaled to the full width of the container (hint: use `<div style="width: 100%;">` to set a div width to 100%))
- “height” (an integer number of pixels; without it, FlexMeasures sets a default, currently 300)

`get_chart_annotations(id: int, sensor: Sensor, **kwargs)`

GET from /sensor/<id>/chart_annotations

Annotations for use in charts (in case you have the chart specs already).

`get_chart_data(id: int, sensor: Sensor, **kwargs)`

GET from /sensor/<id>/chart_data

Data for use in charts (in case you have the chart specs already).

Optional fields

- “event_starts_after” (see the [timely-beliefs documentation](#))
- “event_ends_before” (see the [timely-beliefs documentation](#))
- “beliefs_after” (see the [timely-beliefs documentation](#))
- “beliefs_before” (see the [timely-beliefs documentation](#))
- “resolution” (see [resolutions](#))
- “most_recent_beliefs_only” (if true, returns the most recent belief for each event; if false, returns each belief for each event; defaults to true)

Endpoints under development. Use at your own risk.

Functions

`flexmeasures.api.dev.register_at(app: Flask)`

This can be used to register FlaskViews.

flexmeasures.api.play

Modules

`flexmeasures.api.play.implementations`

`flexmeasures.api.play.routes`

flexmeasures.api.play.implementations

Functions

`flexmeasures.api.play.implementations.restore_data_response()`

flexmeasures.api.play.routes

Functions

`flexmeasures.api.play.routes.restore_data()`

API endpoint to restore the database to one of the saved backups.

Example request

This message restores the database to a backup named demo_v0.

```
{
  "backup": "demo_v0"
}
```

Example response

This message indicates that the backup has been restored without any error.

```
{
  "message": "Request has been processed. Database restored to demo_v0.",
  "status": "PROCESSED"
}
```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

NO_BACKUP, UNRECOGNIZED_BACKUP

Status 401

UNAUTHORIZED

Status 405

INVALID_METHOD

Endpoints to support “play” mode, data restoration

Functions

`flexmeasures.api.play.register_at(app: Flask)`

This can be used to register this blueprint together with other api-related things

flexmeasures.api.sunset

Modules

`flexmeasures.api.sunset.routes`

flexmeasures.api.sunset.routes

Functions

`flexmeasures.api.sunset.routes.implementation_gone()`

A place to keep all routes to endpoints that previously existed and are now sunset.

Functions

`flexmeasures.api.sunset.register_at(app: Flask)`

This can be used to register this blueprint together with other api-related things

flexmeasures.api.v3_0

Modules

flexmeasures.api.v3_0.accounts

flexmeasures.api.v3_0.assets

flexmeasures.api.v3_0.health

flexmeasures.api.v3_0.public

flexmeasures.api.v3_0.sensors

flexmeasures.api.v3_0.users

flexmeasures.api.v3_0.accounts

Classes

class flexmeasures.api.v3_0.accounts.**AccountAPI**

get(*id*: *int*, *account*: *Account*)

API endpoint to get an account.

This endpoint retrieves an account, given its id. Only admins or the user themselves can use this endpoint.

Example response

```
{
  'id': 1,
  'name': 'Test Account'
  'account_roles': [1, 3],
}
```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401

UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

index()

API endpoint to list all accounts accessible to the current user.

This endpoint returns all accessible accounts. Accessible accounts are your own account, or all accounts for admins. When the super-account concept (GH#203) lands, then users in such accounts see all managed accounts.

Example response

An example of one account being returned:

```
[
  {
    'id': 1,
    'name': 'Test Account'
    'account_roles': [1, 3],
  }
]
```

Reqheader Authorization
The authentication token

Reqheader Content-Type
application/json

Resheader Content-Type
application/json

Status 200
PROCESSED

Status 400
INVALID_REQUEST

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

flexmeasures.api.v3_0.assets**Classes****class** flexmeasures.api.v3_0.assets.AssetAPI

This API view exposes generic assets. Under development until it replaces the original Asset API.

delete(*id*: int, *asset*: GenericAsset)

Delete an asset given its identifier.

This endpoint deletes an existing asset, as well as all sensors and measurements recorded for it.

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 204

DELETED

Status 400

INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 422

UNPROCESSABLE_ENTITY

fetch_one(*id*, *asset*)

Fetch a given asset.

This endpoint gets an asset.

Example response

```
{
  "generic_asset_type_id": 2,
  "name": "Test battery",
  "id": 1,
  "latitude": 10,
  "longitude": 100,
  "account_id": 1,
}
```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200
PROCESSED

Status 400
INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

get_chart(*id*: *int*, *asset*: [GenericAsset](#), ***kwargs*)

GET from /assets/<id>/chart

get_chart_data(*id*: *int*, *asset*: [GenericAsset](#), ***kwargs*)

GET from /assets/<id>/chart_data

Data for use in charts (in case you have the chart specs already).

index(*account*: [Account](#))

List all assets owned by a certain account.

This endpoint returns all accessible assets for the account of the user. The *account_id* query parameter can be used to list assets from a different account.

Example response

An example of one asset being returned:

```
[
  {
    "id": 1,
    "name": "Test battery",
    "latitude": 10,
    "longitude": 100,
    "account_id": 2,
    "generic_asset_type_id": 1
  }
]
```

Reqheader Authorization
The authentication token

Reqheader Content-Type
application/json

Resheader Content-Type
application/json

Status 200
PROCESSED

Status 400
INVALID_REQUEST

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

patch(*asset_data*: *dict*, *id*: *int*, *db_asset*: *GenericAsset*)

Update an asset given its identifier.

This endpoint sets data for an existing asset. Any subset of asset fields can be sent.

The following fields are not allowed to be updated: - id - account_id

Example request

```
{
  "latitude": 11.1,
  "longitude": 99.9,
}
```

Example response

The whole asset is returned in the response:

```
{
  "generic_asset_type_id": 2,
  "id": 1,
  "latitude": 11.1,
  "longitude": 99.9,
  "name": "Test battery",
  "account_id": 2,
}
```

Reqheader Authorization
The authentication token

Reqheader Content-Type
application/json

Resheader Content-Type
application/json

Status 200
UPDATED

Status 400
INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

post(*asset_data: dict*)

Create new asset.

This endpoint creates a new asset.

Example request

```
{
  "name": "Test battery",
  "generic_asset_type_id": 2,
  "account_id": 2,
  "latitude": 40,
  "longitude": 170.3,
}
```

The newly posted asset is returned in the response.

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 201

CREATED

Status 400

INVALID_REQUEST

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 422

UNPROCESSABLE_ENTITY

public()

Return all public assets.

This endpoint returns all public assets.

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST

Status 401

UNAUTHORIZED

Status 422
UNPROCESSABLE_ENTITY

flexmeasures.api.v3_0.health

Classes

class flexmeasures.api.v3_0.health.**HealthAPI**

is_ready()

Get readiness status

Example response:

```
{
  'database_sql': True
}
```

flexmeasures.api.v3_0.public

Functions

flexmeasures.api.v3_0.public.**quickref_directive**(*content*)

Adapted from sphinxcontrib/autohttp/flask_base.py:quickref_directive.

Classes

class flexmeasures.api.v3_0.public.**ServicesAPI**

index()

API endpoint to get a service listing for this version.

Resheader Content-Type
application/json

Status 200
PROCESSED

flexmeasures.api.v3_0.sensors

Classes

class flexmeasures.api.v3_0.sensors.**SensorAPI**

get_data(*response: dict*)

Get sensor data from FlexMeasures.

Example request

```
{
  "sensor": "ea1.2021-01.io.flexmeasures:fm1.1",
  "start": "2021-06-07T00:00:00+02:00",
  "duration": "PT1H",
  "resolution": "PT15M",
  "unit": "m³/h"
}
```

The unit has to be convertible from the sensor's unit.

Optional fields

- “resolution” (see *Frequency and resolution*)
- “horizon” (see *Tracking the recording time of beliefs*)
- “prior” (see *Tracking the recording time of beliefs*)
- “source” (see *Sources*)

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 422

UNPROCESSABLE_ENTITY

get_schedule(*sensor*: [Sensor](#), *job_id*: *str*, *duration*: *timedelta*, ***kwargs*)

Get a schedule from FlexMeasures.

Optional fields

- “duration” (6 hours by default; can be increased to plan further into the future)

Example response

This message contains a schedule indicating to consume at various power rates from 10am UTC onwards for a duration of 45 minutes.

```
{
  "values": [
    2.15,
    3,
```

(continues on next page)

(continued from previous page)

```

    2
  ],
  "start": "2015-06-02T10:00:00+00:00",
  "duration": "PT45M",
  "unit": "MW"
}

```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400INVALID_TIMEZONE, INVALID_DOMAIN, INVALID_UNIT, UN-
KNOWN_SCHEDULE, UNRECOGNIZED_CONNECTION_GROUP**Status 401**

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 405

INVALID_METHOD

Status 422

UNPROCESSABLE_ENTITY

index(*account*: [Account](#))

API endpoint to list all sensors of an account.

This endpoint returns all accessible sensors. Accessible sensors are sensors in the same account as the current user. Only admins can use this endpoint to fetch sensors from a different account (by using the *account_id* query parameter).

Example response

An example of one sensor being returned:

```

[
  {
    "entity_address": "ea1.2021-01.io.flexmeasures.company:fm1.42",
    "event_resolution": 15,
    "generic_asset_id": 1,
    "name": "Gas demand",
    "timezone": "Europe/Amsterdam",
    "unit": "m³/h"
  }
]

```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 422

UNPROCESSABLE_ENTITY

post_data(*bdf: BeliefsDataFrame*)

Post sensor data to FlexMeasures.

Example request

```
{
  "sensor": "ea1.2021-01.io.flexmeasures:fm1.1",
  "values": [-11.28, -11.28, -11.28, -11.28],
  "start": "2021-06-07T00:00:00+02:00",
  "duration": "PT1H",
  "unit": "m³/h"
}
```

The above request posts four values for a duration of one hour, where the first event start is at the given start time, and subsequent events start in 15 minute intervals throughout the one hour duration.

The sensor is the one with ID=1. The unit has to be convertible to the sensor's unit. The resolution of the data has to match the sensor's required resolution, but FlexMeasures will attempt to upsample lower resolutions. The list of values may include null values.

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST

Status 401

UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

trigger_schedule(*sensor*: [Sensor](#), *start_of_schedule*: *datetime*, *duration*: *timedelta*, *belief_time*: *datetime* | *None* = *None*, *flex_model*: *dict* | *None* = *None*, *flex_context*: *dict* | *None* = *None*, ***kwargs*)

Trigger FlexMeasures to create a schedule.

Trigger FlexMeasures to create a schedule for this sensor. The assumption is that this sensor is the power sensor on a flexible asset.

In this request, you can describe:

- the schedule's main features (when does it start, what unit should it report, prior to what time can we assume knowledge)
- the flexibility model for the sensor (state and constraint variables, e.g. current state of charge of a battery, or connection capacity)
- the flexibility context which the sensor operates in (other sensors under the same EMS which are relevant, e.g. prices)

For details on flexibility model and context, see [Describing flexibility](#). Below, we'll also list some examples.

Note: This endpoint does not support to schedule an EMS with multiple flexible sensors at once. This will happen in another endpoint. See <https://github.com/FlexMeasures/flexmeasures/issues/485>. Until then, it is possible to call this endpoint for one flexible endpoint at a time (considering already scheduled sensors as inflexible).

The length of the schedule can be set explicitly through the 'duration' field. Otherwise, it is set by the config setting `FLEXMEASURES_PLANNING_HORIZON`, which defaults to 48 hours. If the flex-model contains targets that lie beyond the planning horizon, the length of the schedule is extended to accommodate them. Finally, the schedule length is limited by `max_planning_horizon_config`, which defaults to 2520 steps of the sensor's resolution. Targets that exceed the max planning horizon are not accepted.

The appropriate algorithm is chosen by FlexMeasures (based on asset type). It's also possible to use custom schedulers and custom flexibility models, see [Plugin Customizations](#).

If you have ideas for algorithms that should be part of FlexMeasures, let us know: <https://flexmeasures.io/get-in-touch/>

Example request A

This message triggers a schedule for a storage asset, starting at 10.00am, at which the state of charge (soc) is 12.1 kWh.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh"
  }
}
```

Example request B

This message triggers a 24-hour schedule for a storage asset, starting at 10.00am, at which the state of charge (soc) is 12.1 kWh, with a target state of charge of 25 kWh at 4.00pm. The global minimum and maximum soc are set to 10 and 25 kWh, respectively. To guarantee a minimum SOC in the period prior to 4.00pm, local minima constraints are imposed (via soc-minima) at 2.00pm and 3.00pm, for 15kWh and 20kWh, respectively. Roundtrip efficiency for use in scheduling is set to 98%. Storage efficiency is set to 99.99%, denoting the state of charge left after each time step equal to the sensor's resolution. Aggregate consumption (of all devices within this EMS) should be priced by sensor 9, and aggregate production should be priced by sensor 10, where the aggregate power flow in the EMS is described by the sum over sensors 13, 14 and 15 (plus the flexible sensor being optimized, of course). Note that, if forecasts for sensors 13, 14 and 15 are not available, a schedule cannot be computed.

```
{
  "start": "2015-06-02T10:00:00+00:00",
  "duration": "PT24H",
  "flex-model": {
    "soc-at-start": 12.1,
    "soc-unit": "kWh",
    "soc-targets": [
      {
        "value": 25,
        "datetime": "2015-06-02T16:00:00+00:00"
      },
    ],
    "soc-minima": [
      {
        "value": 15,
        "datetime": "2015-06-02T14:00:00+00:00"
      },
      {
        "value": 20,
        "datetime": "2015-06-02T15:00:00+00:00"
      }
    ],
    "soc-min": 10,
    "soc-max": 25,
    "roundtrip-efficiency": 0.98,
    "storage-efficiency": 0.9999,
  },
  "flex-context": {
    "consumption-price-sensor": 9,
    "production-price-sensor": 10,
    "inflexible-device-sensors": [13, 14, 15]
  }
}
```

Example response

This message indicates that the scheduling request has been processed without any error. A scheduling job has been created with some Universally Unique Identifier (UUID), which will be picked up by a worker. The given UUID may be used to obtain the resulting schedule: see `/sensors/<id>/schedules/<uuid>`.

```
{
  "status": "PROCESSED",
  "schedule": "364bfd06-c1fa-430b-8d25-8f5a547651fb",
}
```

(continues on next page)

(continued from previous page)

```

    "message": "Request has been processed."
}

```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_DATA

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 405

INVALID_METHOD

Status 422

UNPROCESSABLE_ENTITY

flexmeasures.api.v3_0.users**Classes**

```
class flexmeasures.api.v3_0.users.UserAPI
```

```
    get(id: int, user: User)
```

API endpoint to get a user.

This endpoint gets a user. Only admins or the members of the same account can use this endpoint.

Example response

```

{
    'account_id': 1,
    'active': True,
    'email': 'test_prosumer@seita.nl',
    'flexmeasures_roles': [1, 3],
    'id': 1,
    'timezone': 'Europe/Amsterdam',
    'username': 'Test Prosumer User'
}

```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401

UNAUTHORIZED

Status 403

INVALID_SENDER

Status 422

UNPROCESSABLE_ENTITY

index(*account*: [Account](#), *include_inactive*: *bool* = *False*)

API endpoint to list all users of an account.

This endpoint returns all accessible users. By default, only active users are returned. The *include_inactive* query parameter can be used to also fetch inactive users. Accessible users are users in the same account as the current user. Only admins can use this endpoint to fetch users from a different account (by using the *account_id* query parameter).

Example response

An example of one user being returned:

```
[
  {
    'active': True,
    'email': 'test_prosumer@seita.nl',
    'account_id': 13,
    'flexmeasures_roles': [1, 3],
    'id': 1,
    'timezone': 'Europe/Amsterdam',
    'username': 'Test Prosumer User'
  }
]
```

Reqheader Authorization

The authentication token

Reqheader Content-Type

application/json

Resheader Content-Type

application/json

Status 200

PROCESSED

Status 400

INVALID_REQUEST

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

patch(*id*: int, *user*: User, ***user_data*)

API endpoint to patch user data.

This endpoint sets data for an existing user. It has to be used by the user themselves, admins or account-admins (of the same account). Any subset of user fields can be sent. If the user is not an (account-)admin, they can only edit a few of their own fields.

The following fields are not allowed to be updated at all:

- id
- account_id

Example request

```
{
  "active": false,
}
```

Example response

The following user fields are returned:

```
{
  'account_id': 1,
  'active': True,
  'email': 'test_prosumer@seita.nl',
  'flexmeasures_roles': [1, 3],
  'id': 1,
  'timezone': 'Europe/Amsterdam',
  'username': 'Test Prosumer User'
}
```

Reqheader Authorization
The authentication token

Reqheader Content-Type
application/json

Resheader Content-Type
application/json

Status 200
UPDATED

Status 400
INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

reset_user_password(*id: int, user: User*)

API endpoint to reset the user's current password, cookies and auth tokens, and to email a password reset link to the user.

Reset the user's password, and send them instructions on how to reset the password. This endpoint is useful from a security standpoint, in case of worries the password might be compromised. It sets the current password to something random, invalidates cookies and auth tokens, and also sends an email for resetting the password to the user.

Users can reset their own passwords. Only admins can use this endpoint to reset passwords of other users.

Reqheader Authorization
The authentication token

Reqheader Content-Type
application/json

Resheader Content-Type
application/json

Status 200
PROCESSED

Status 400
INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEXPECTED_PARAMS

Status 401
UNAUTHORIZED

Status 403
INVALID_SENDER

Status 422
UNPROCESSABLE_ENTITY

FlexMeasures API v3

Functions

`flexmeasures.api.v3_0.register_at(app: Flask)`

This can be used to register this blueprint together with other api-related things

FlexMeasures API routes and implementations.

Functions

`flexmeasures.api.get_versions()` → `dict`

Public endpoint to list API versions.

`flexmeasures.api.register_at(app: Flask)`

This can be used to register this blueprint together with other api-related things

`flexmeasures.api.request_auth_token()`

API endpoint to get a fresh authentication access token. Be aware that this fresh token has a limited lifetime (which depends on the current system setting `SECURITY_TOKEN_MAX_AGE`).

Pass the *email* parameter to identify the user. Pass the *password* parameter to authenticate the user (if not already authenticated in current session)

5.3.40 flexmeasures.app

Starting point of the Flask application.

Functions

`flexmeasures.app.create(env: str | None = None, path_to_config: str | None = None, plugins: list[str] | None = None) → Flask`

Create a Flask app and configure it.

Set the environment by setting `FLASK_ENV` as environment variable (also possible in `.env`). Or, overwrite any `FLASK_ENV` setting by passing an `env` in directly (useful for testing for instance).

A path to a config file can be passed in (otherwise a config file will be searched in the home or instance directories).

Also, a list of plugins can be set. Usually this works as a config setting, but this is useful for automated testing.

5.3.41 flexmeasures.auth

Modules

<code>flexmeasures.auth.decorators</code>	Auth decorators for endpoints
<code>flexmeasures.auth.error_handling</code>	Auth error handling.
<code>flexmeasures.auth.policy</code>	Tooling & docs for implementing our auth policy

flexmeasures.auth.decorators

Auth decorators for endpoints

Functions

`flexmeasures.auth.decorators.account_roles_accepted(*account_roles)`

Decorator which specifies that a user's account must have at least one of the specified roles (or must be an admin).

Example:

```
@app.route('/postMeterData')    @account_roles_accepted('Prosumer',    'MDC')    def
post_meter_data():

    return 'Meter data posted'
```

The current user's account must have either the *Prosumer* role or *MDC* role in order to use the service.

Parameters

account_roles – The possible roles.

`flexmeasures.auth.decorators.account_roles_required(*account_roles)`

Decorator which specifies that a user's account must have all the specified roles. Example:

```
@app.route('/dashboard')
@account_roles_required('Prosumer', 'App-subscriber')
def dashboard():
    return 'Dashboard'
```

The current user's account must have both the *Prosumer* role and *App-subscriber* role in order to view the page.

Parameters

roles – The required roles.

`flexmeasures.auth.decorators.permission_required_for_context(permission: str, arg_pos: int | None = None, arg_name: str | None = None, arg_loader: Callable | None = None)`

This decorator can be used to make sure that the current user has the necessary permission to access the context. The context needs to be an `AuthModelMixin` and is found ... - by loading it via the `arg_loader` callable; - otherwise:

- by the keyword argument `arg_name`;
- and/or by a position in the non-keyword arguments (`arg_pos`).

If nothing is passed, the context lookup defaults to `arg_pos=0`.

Using both `arg_name` and `arg_pos` arguments is useful when `Marshmallow` de-serializes to a dict and you are using `use_args`. In this case, the context lookup applies first `arg_pos`, then `arg_name`.

The permission needs to be a known permission and is checked with principal descriptions from the context's access control list (see `AuthModelMixin.__acl__`).

Usually, you'd place a `marshmallow` field further up in the decorator chain, e.g.:

```
@app.route("/resource/<resource_id>", methods=["GET"]) @use_kwargs(
    {"the_resource": ResourceIdField(data_key="resource_id")}, location="path",
) @permission_required_for_context("read",    arg_name="the_resource")    @as_json    def
view(resource_id: int, the_resource: Resource):

    return dict(name=the_resource.name)
```

Where `ResourceIdField._deserialize()` turns the id parameter into a Resource context (if possible).

This decorator raises a 403 response if there is no principal for the required permission. It raises a 401 response if the user is not authenticated at all.

`flexmeasures.auth.decorators.roles_accepted(*roles)`

As in Flask-Security, but also accept admin

`flexmeasures.auth.decorators.roles_required(*roles)`

As in Flask-Security, but wave through if user is admin

flexmeasures.auth.error_handling

Auth error handling.

Beware: There is a historical confusion of naming between authentication and authorization.

Names of Responses have to be kept as they were called in original W3 protocols. See explanation below.

Functions

`flexmeasures.auth.error_handling.unauthenticated_handler`(*mechanisms: list | None = None, headers: dict | None = None*)

Handler for authentication problems. :param mechanisms: a list of which authentication mechanisms were tried. :param headers: a dict of headers to return. We respond with json if the request doesn't say otherwise. Also, other FlexMeasures packages can define that they want to wrap JSON responses and/or render HTML error pages (for non-JSON requests) in custom ways — by registering `unauthenticated_handler_api` & `unauthenticated_handler_html`, respectively.

`flexmeasures.auth.error_handling.unauthenticated_handler_e(e)`

Swallow error. Useful for classical Flask error handler registration.

`flexmeasures.auth.error_handling.unauthorized_handler`(*func: Callable | None = None, params: list | None = None*)

Handler for authorization problems. :param func: the Flask-Security-Too decorator, if relevant, and params are its parameters.

We respond with json if the request doesn't say otherwise. Also, other FlexMeasures packages can define that they want to wrap JSON responses and/or render HTML error pages (for non-JSON requests) in custom ways — by registering `unauthorized_handler_api` & `unauthorized_handler_html`, respectively.

`flexmeasures.auth.error_handling.unauthorized_handler_e(e)`

Swallow error. Useful for classical Flask error handler registration.

flexmeasures.auth.policy

Tooling & docs for implementing our auth policy

Functions

`flexmeasures.auth.policy.check_access(context: AuthModelMixin, permission: str)`

Check if current user can access this auth context if this permission is required, either with admin rights or principal(s).

Raises 401 or 403 otherwise.

`flexmeasures.auth.policy.check_account_membership(user, principal: str) → bool`

`flexmeasures.auth.policy.check_account_role(user, principal: str) → bool`

`flexmeasures.auth.policy.check_user_identity(user, principal: str) → bool`

`flexmeasures.auth.policy.check_user_role(user, principal: str) → bool`

`flexmeasures.auth.policy.user_has_admin_access(user, permission: str) → bool`

`flexmeasures.auth.policy.user_matches_principals(user, principals: str | Tuple[str] | List[str] | Tuple[str]]) → bool`

Tests if the user matches all passed principals. Returns False if no principals are passed.

Classes

`class flexmeasures.auth.policy.AuthModelMixin`

Authentication and authorization policies and helpers.

Functions

`flexmeasures.auth.register_at(app: Flask)`

5.3.42 flexmeasures.cli

Modules

<code>flexmeasures.cli.data_add</code>	CLI commands for populating the database
<code>flexmeasures.cli.data_delete</code>	CLI commands for removing data
<code>flexmeasures.cli.data_edit</code>	CLI commands for editing data
<code>flexmeasures.cli.data_show</code>	CLI commands for listing database contents and classes
<code>flexmeasures.cli.db_ops</code>	CLI commands for saving, resetting, etc of the database
<code>flexmeasures.cli.jobs</code>	CLI commands for controlling jobs
<code>flexmeasures.cli.monitor</code>	CLI commands for monitoring functionality.
<code>flexmeasures.cli.utils</code>	Utils for FlexMeasures CLI

flexmeasures.cli.data_add

CLI commands for populating the database

Functions

flexmeasures.cli.data_add.**check_errors**(errors: *dict[str, list[str]]*)

flexmeasures.cli.data_add.**check_timezone**(timezone)

flexmeasures.cli.data_add.**parse_source**(source)

flexmeasures.cli.data_delete

CLI commands for removing data

flexmeasures.cli.data_edit

CLI commands for editing data

Functions

flexmeasures.cli.data_edit.**parse_attribute_value**(attribute_null_value: *bool*, attribute_float_value: *float | None = None*, attribute_bool_value: *bool | None = None*, attribute_str_value: *str | None = None*, attribute_int_value: *int | None = None*) → *float | int | bool | str | None*

Parse attribute value.

flexmeasures.cli.data_edit.**single_true**(iterable) → *bool*

flexmeasures.cli.data_show

CLI commands for listing database contents and classes

Functions

flexmeasures.cli.data_show.**list_items**(item_type)

Show available items of a specific type.

flexmeasures.cli.db_ops

CLI commands for saving, resetting, etc of the database

flexmeasures.cli.jobs

CLI commands for controlling jobs

Functions

`flexmeasures.cli.jobs.handle_worker_exception(job, exc_type, exc_value, traceback)`

Just a fallback, usually we would use the per-queue handler.

`flexmeasures.cli.jobs.parse_queue_list(queue_names_str: str) → list[rq.queue.Queue]`

Parse a | separated string of queue names against the app.queues dict.

The app.queues dict is expected to have queue names as keys, and rq.Queue objects as values.

Parameters

queue_names_str – a string with queue names separated by the | character

Returns

a list of Queue objects.

flexmeasures.cli.monitor

CLI commands for monitoring functionality.

Functions

`flexmeasures.cli.monitor.send_lastseen_monitoring_alert(users: list[User], last_seen_delta: timedelta, alerted_users: bool, account_role: str | None = None, user_role: str | None = None)`

Tell monitoring recipients and Sentry about user(s) we haven't seen in a while.

`flexmeasures.cli.monitor.send_task_monitoring_alert(task_name: str, msg: str, latest_run: LatestTaskRun | None = None, custom_msg: str | None = None)`

Send any monitoring message per Sentry and per email. Also log an error.

flexmeasures.cli.utils

Utils for FlexMeasures CLI

Functions

```
flexmeasures.cli.utils.get_timerange_from_flag(last_hour: bool = False, last_day: bool = False,
                                              last_7_days: bool = False, last_month: bool = False,
                                              last_year: bool = False, timezone:
                                              ~pytz.tzinfo.BaseTzInfo = <DstTzInfo 'Asia/Seoul'
                                              LMT+8:28:00 STD>) → tuple[datetime.datetime,
                                              datetime.datetime]
```

This function returns a time range [start,end] of the last-X period. See input parameters for more details.

Parameters

- **last_hour** (*bool*) – flag to get the time range of the last finished hour.
- **last_day** (*bool*) – flag to get the time range for yesterday.
- **last_7_days** (*bool*) – flag to get the time range of the previous 7 days.
- **last_month** (*bool*) – flag to get the time range of last calendar month
- **last_year** (*bool*) – flag to get the last completed calendar year
- **timezone** – timezone object to represent

Returns

start:datetime, end:datetime

Classes

```
class flexmeasures.cli.utils.DeprecatedDefaultGroup(*args, **kwargs)
```

Invokes a default subcommand, *and* shows a deprecation message.

Also adds the *invoked_default* boolean attribute to the context. A group callback can use this information to figure out if it's being executed directly (invoking the default subcommand) or because the execution flow passes onwards to a subcommand. By default it's None, but it can be the name of the default subcommand to execute.

```
import click
from flexmeasures.cli.utils import DeprecatedDefaultGroup

@click.group(cls=DeprecatedDefaultGroup, default="bar", deprecation_message=
↳ "renamed to `foo bar`.")
def foo(ctx):
    if ctx.invoked_default:
        click.echo("foo")

@foo.command()
def bar():
    click.echo("bar")
```

```
$ flexmeasures foo
DeprecationWarning: renamed to `foo bar`.
foo
bar
$ flexmeasures foo bar
bar
```

```
__init__(*args, **kwargs)
```

```
get_command(ctx, cmd_name)
```

Given a context and a command name, this returns a `Command` object if it exists or returns *None*.

```
class flexmeasures.cli.utils.MsgStyle
```

Stores the text styles for the different events

Styles options are the attributes of the `click.style` which can be found [here](<https://click.palletsprojects.com/en/8.1.x/api/#click.style>).

CLI functions for FlexMeasures hosts.

Functions

```
flexmeasures.cli.is_running() → bool
```

True if we are running one of the custom FlexMeasures CLI commands.

We use this in combination with authorization logic, e.g. we assume that only sysadmins run commands there, but also we consider forecasting & scheduling jobs to be in that realm, as well.

This tooling might not live forever, as we could evolve into a more sophisticated auth model for these cases. For instance, these jobs are queued by the system, but caused by user actions (sending data), and then they are run by the system.

See also: the `run_as_cli` test fixture, which uses the (non-public) `PRETEND_RUNNING_AS_CLI` env setting.

```
flexmeasures.cli.register_at(app: Flask)
```

5.3.43 flexmeasures.data

Modules

<code>flexmeasures.data.config</code>	Database configuration utils
<code>flexmeasures.data.models</code>	Data models for FlexMeasures
<code>flexmeasures.data.queries</code>	Data query functions
<code>flexmeasures.data.schemas</code>	Data schemas (Marshmallow)
<code>flexmeasures.data.scripts</code>	Useful scripts
<code>flexmeasures.data.services</code>	Business logic
<code>flexmeasures.data.transactional</code>	These, and only these, functions should help you with treating your own code in the context of one database transaction.
<code>flexmeasures.data.utils</code>	Utils around the data models and db sessions

flexmeasures.data.config

Database configuration utils

Functions

`flexmeasures.data.config.commit_and_start_new_session(app: Flask)`

Use this when a script wants to save a state before continuing. Not tested well, just a starting point - not recommended anyway for any logic used by views or tasks. Maybe `session.flush` can help you there.

`flexmeasures.data.config.configure_db_for(app: Flask)`

Call this to configure the database and the tools we use on it for the Flask app. This should only be called once in the app's lifetime.

`flexmeasures.data.config.init_db()`

Initialise the database object

flexmeasures.data.models

Modules

<code>flexmeasures.data.models.annotations</code>	
<code>flexmeasures.data.models.assets</code>	
<code>flexmeasures.data.models.charts</code>	
<code>flexmeasures.data.models.data_sources</code>	
<code>flexmeasures.data.models.forecasting</code>	
<code>flexmeasures.data.models.generic_assets</code>	
<code>flexmeasures.data.models.legacy_migration_utils</code>	This module is part of our data model migration (see https://github.com/SeitaBV/flexmeasures/projects/9).
<code>flexmeasures.data.models.markets</code>	
<code>flexmeasures.data.models.parsing_utils</code>	
<code>flexmeasures.data.models.planning</code>	
<code>flexmeasures.data.models.reporting</code>	
<code>flexmeasures.data.models.task_runs</code>	
<code>flexmeasures.data.models.time_series</code>	
<code>flexmeasures.data.models.user</code>	
<code>flexmeasures.data.models.validation_utils</code>	
<code>flexmeasures.data.models.weather</code>	

`flexmeasures.data.models.annotations`

Functions

`flexmeasures.data.models.annotations.get_or_create_annotation(annotation: Annotation)` → *Annotation*

Add annotation to db session if it doesn't exist in the session already.

Return the old annotation object if it exists (and expunge the new one). Otherwise, return the new one.

`flexmeasures.data.models.annotations.to_annotation_frame(annotations: list[flexmeasures.data.models.annotations.Annotation])` → `DataFrame`

Transform a list of annotations into a DataFrame.

We don't use a `BeliefsDataFrame` here, because they are designed for quantitative data only.

Classes

class flexmeasures.data.models.annotations.**AccountAnnotationRelationship**(**kwargs)

Links annotations to accounts.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class flexmeasures.data.models.annotations.**Annotation**(**kwargs)

An annotation is a nominal value that applies to a specific time or time span.

Examples of annotation types:

- user annotation: `annotation.type == "label"` and `annotation.source.type == "user"`
- unresolved alert: `annotation.type == "alert"`
- resolved alert: `annotation.type == "label"` and `annotation.source.type == "alerting script"`
- organisation holiday: `annotation.type == "holiday"` and `annotation.source.type == "user"`
- public holiday: `annotation.type == "holiday"` and `annotation.source.name == "workalendar"`

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

classmethod **add**(df: DataFrame, annotation_type: str, expunge_session: bool = False, allow_overwrite: bool = False, bulk_save_objects: bool = False, commit_transaction: bool = False) → list[Annotation]

Add a data frame describing annotations to the database and return the Annotation objects.

Parameters

- **df** – Data frame describing annotations. Expects the following columns (or multi-index levels): - start - end or duration - content - belief_time - source
- **annotation_type** – One of the possible Enum values for `annotation.type`
- **expunge_session** – if True, all non-flushed instances are removed from the session before adding annotations. Expunging can resolve problems you might encounter with states of objects in your session. When using this option, you might want to flush newly-created objects which are not annotations (e.g. a sensor or data source object).
- **allow_overwrite** – if True, new objects are merged if False, objects are added to the session or bulk saved
- **bulk_save_objects** – if True, objects are bulk saved with `session.bulk_save_objects()`, which is quite fast but has several caveats, see: https://docs.sqlalchemy.org/orm/persistence_techniques.html#bulk-operations-caveats if False, objects are added to the session with `session.add_all()`
- **commit_transaction** – if True, the session is committed if False, you can still add other data to the session and commit it all within an atomic transaction

```
class flexmeasures.data.models.annotations.GenericAssetAnnotationRelationship(**kwargs)
```

Links annotations to generic assets.

```
    __init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

```
class flexmeasures.data.models.annotations.SensorAnnotationRelationship(**kwargs)
```

Links annotations to sensors.

```
    __init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

flexmeasures.data.models.assets

Functions

```
flexmeasures.data.models.assets.assets_share_location(assets: List[Asset]) → bool
```

Return True if all assets in this list are located on the same spot. TODO: In the future, we might soften this to compare if assets are in the same “housing” or “site”.

Classes

```
class flexmeasures.data.models.assets.Asset(**kwargs)
```

Each asset is an energy- consuming or producing hardware.

This model is now considered legacy. See GenericAsset and Sensor.

```
    __init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

```
    property entity_address: str
```

Entity address under the latest fm scheme for entity addresses.

```
    property entity_address_fm0: str
```

Entity address under the fm0 scheme for entity addresses.

```
    event_resolution: timedelta
```

```
    get_attribute(attribute: str)
```

Looks for the attribute on the corresponding Sensor.

This should be used by all code to read these attributes, over accessing them directly on this class, as this table is in the process to be replaced by the Sensor table.

id

property is_pure_consumer: `bool`

Return True if this asset is consuming but not producing.

property is_pure_producer: `bool`

Return True if this asset is producing but not consuming.

knowledge_horizon_fnc: `str`

knowledge_horizon_par: `dict`

latest_state(*event_ends_before: datetime | None = None*) \rightarrow *Power*

Search the most recent event for this sensor, optionally before some datetime.

name: `str`

property power_unit: `float`

Return the ‘unit’ property of the generic asset, just with a more insightful name.

timezone: `str`

unit: `str`

class flexmeasures.data.models.assets.**AssetType**(***kwargs*)

Describing asset types for our purposes

This model is now considered legacy. See GenericAssetType.

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

property preconditions: `Dict[str, bool]`

Assumptions about the time series data set, such as normality and stationarity For now, this is usable input for Prophet (see init), but it might evolve or go away.

property weather_correlations: `List[str]`

Known correlations of weather sensor type and asset type.

class flexmeasures.data.models.assets.**Power**(*use_legacy_kwargs=True, **kwargs*)

All measurements of power data are stored in one slim table. Negative values indicate consumption.

This model is now considered legacy. See TimedBelief.

__init__(*use_legacy_kwargs=True, **kwargs*)

data_source_id

datetime

horizon

classmethod make_query(***kwargs*) \rightarrow *Query*

Construct the database query.

value

flexmeasures.data.models.charts

Modules

```
flexmeasures.data.models.charts.  
belief_charts  
flexmeasures.data.models.charts.defaults
```

flexmeasures.data.models.charts.belief_charts

Functions

```
flexmeasures.data.models.charts.belief_charts.bar_chart(sensor: Sensor, event_starts_after:  
datetime | None = None,  
event_ends_before: datetime | None =  
None, **override_chart_specs: dict)
```

```
flexmeasures.data.models.charts.belief_charts.chart_for_multiple_sensors(sensors_to_show:  
list['Sensor'],  
list['Sensor']],  
event_starts_after:  
datetime | None =  
None,  
event_ends_before:  
datetime | None =  
None, **over-  
ride_chart_specs:  
dict)
```

```
flexmeasures.data.models.charts.belief_charts.create_circle_layer(sensors: list['Sensor'],  
event_start_field_definition:  
dict,  
event_value_field_definition:  
dict, shared_tooltip: list)
```

```
flexmeasures.data.models.charts.belief_charts.create_line_layer(sensors: list['Sensor'],  
event_start_field_definition: dict,  
event_value_field_definition:  
dict)
```

```
flexmeasures.data.models.charts.belief_charts.create_rect_layer(event_start_field_definition: dict,  
event_value_field_definition:  
dict, shared_tooltip: list)
```

```
flexmeasures.data.models.charts.belief_charts.determine_shared_sensor_type(sensors:  
list['Sensor']) →  
str
```

```
flexmeasures.data.models.charts.belief_charts.determine_shared_unit(sensors: list['Sensor']) →  
str
```

flexmeasures.data.models.charts.defaults

Functions

`flexmeasures.data.models.charts.defaults.apply_chart_defaults(fn)`

`flexmeasures.data.models.charts.defaults.merge_vega_lite_specs(child: dict, parent: dict) → dict`

Merge nested dictionaries, with child inheriting values from parent.

Child values are updated with parent values if they exist. In case a field is a string and that field is updated with some dict, the string is moved inside the dict under a field defined in `vega_lite_field_mapping`. For example, 'title' becomes 'text' and 'mark' becomes 'type'.

Functions

`flexmeasures.data.models.charts.chart_type_to_chart_specs(chart_type: str, **kwargs) → dict`

Create chart specs of a given chart type, using FlexMeasures defaults for settings like width and height.

Parameters

chart_type – Name of a variable defining chart specs or a function returning chart specs. The chart specs can be a dictionary or an Altair chart specification. - In case of a dictionary, the creator needs to ensure that the dictionary contains valid specs - In case of an Altair chart specification, Altair validates for you

Returns

A dictionary containing a vega-lite chart specification

flexmeasures.data.models.data_sources

Classes

`class flexmeasures.data.models.data_sources.DataGeneratorMixin`

`classmethod get_data_source_info() → dict`

Create and return the data source info, from which a data source lookup/creation is possible.

See for instance `get_data_source_for_job()`.

`class flexmeasures.data.models.data_sources.DataSource(name=None, type=None, user=None, **kwargs)`

Each data source is a data-providing entity.

`__init__(name=None, type=None, user=None, **kwargs)`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

property description

Extended description

For example:

```
>>> DataSource("Seita", type="forecaster", model="naive", version="1.2").
      ↪description
<<< "Seita's naive model v1.2.0"
```

id

property label

Human-readable label (preferably not starting with a capital letter, so it can be used in a sentence).

name: `str`

flexmeasures.data.models.forecasting

Modules

```
flexmeasures.data.models.forecasting.
exceptions
flexmeasures.data.models.forecasting.
model_spec_factory
flexmeasures.data.models.forecasting.
model_specs
flexmeasures.data.models.forecasting.utils
```

flexmeasures.data.models.forecasting.exceptions

Exceptions

exception `flexmeasures.data.models.forecasting.exceptions.InvalidHorizonException`

exception `flexmeasures.data.models.forecasting.exceptions.NotEnoughDataException`

flexmeasures.data.models.forecasting.model_spec_factory

Functions

`flexmeasures.data.models.forecasting.model_spec_factory.configure_regressors_for_nearest_weather_sensor`

For Assets, we use weather data as regressors. Here, we configure them.

```
flexmeasures.data.models.forecasting.model_spec_factory.create_initial_model_specs(sensor:
    ~flexmeasures.data.models.time_series.TimeSeries,
    forecast_start:
    ~datetime.datetime,
    forecast_end:
    ~datetime.datetime,
    forecast_horizon:
    ~datetime.timedelta,
    ex_post_horizon:
    ~datetime.timedelta
    | None
    = None,
    transform_to_normal:
    bool =
    True,
    use_regressors:
    bool =
    True,
    use_periodicity:
    bool =
    True,
    custom_model_params:
    dict |
    None =
    None,
    time_series_class:
    type |
    None =
    <class
    'flexmeasures.data.models.time_series.TimeSeries'
    →
    ModelSpecs
```

Generic model specs for all asset types (also for markets and weather sensors) and horizons. Fills in training, testing periods, lags. Specifies input and regressor data. Does not fill in which model to actually use. TODO: check if enough data is available both for lagged variables and regressors TODO: refactor assets and markets to store a list of pandas offset or timedelta instead of booleans for

seasonality, because e.g. although solar and building assets both have daily seasonality, only the former is insensitive to daylight savings. Therefore: solar periodicity is 24 hours, while building periodicity is 1 calendar day.

```
flexmeasures.data.models.forecasting.model_spec_factory.get_normalization_transformation_from_sensor_at
```

Transform data to be normal, using the BoxCox transformation. Lambda parameter is chosen according to the asset type.

Classes

```
class flexmeasures.data.models.forecasting.model_spec_factory.TBSeriesSpecs(search_params:
    dict, name: str,
    time_series_class:
    type | None =
    <class 'flexmeasures.data.models.time_series.TimeSeries'>,
    search_fnc: str =
    'search',
    original_tz:
    ~datetime.tzinfo |
    None = <UTC>,
    feature_transformation:
    ~timetomodel.transforming.ReversibleTransformation | None = None,
    post_load_processing:
    ~timetomodel.transforming.Transformation | None = None,
    resampling_config:
    ~typing.Dict[str, ~typing.Any] |
    None = None,
    interpolation_config:
    ~typing.Dict[str, ~typing.Any] |
    None = None)
```

Compatibility for using `timetomodel.SeriesSpecs` with `timely_beliefs.BeliefsDataFrames`.

This implements `_load_series` such that `<time_series_class>.search` is called, with the parameters in `search_params`. The search function is expected to return a `BeliefsDataFrame`.

```
__init__(search_params: dict, name: str, time_series_class: type | None = <class
'flexmeasures.data.models.time_series.TimedBelief'>, search_fnc: str = 'search', original_tz:
~datetime.tzinfo | None = <UTC>, feature_transformation:
~timetomodel.transforming.ReversibleTransformation | None = None, post_load_processing:
~timetomodel.transforming.Transformation | None = None, resampling_config: ~typing.Dict[str,
~typing.Any] | None = None, interpolation_config: ~typing.Dict[str, ~typing.Any] | None = None)
```

_load_series() → Series

Subclasses overwrite this function to get the raw data. This method is responsible to call any post_load_processing at the right place.

check_data(df: DataFrame)

Raise error if data is empty or contains nan values. Here, other than in load_series, we can show the query, which is quite helpful.

flexmeasures.data.models.forecasting.model_specs

Modules

```
flexmeasures.data.models.forecasting.
model_specs.linear_regression
flexmeasures.data.models.forecasting.
model_specs.naive
```

flexmeasures.data.models.forecasting.model_specs.linear_regression

Functions

flexmeasures.data.models.forecasting.model_specs.linear_regression.**ols_specs_configurator**(**kwargs)

Create and customize initial specs with OLS. See model_spec_factory for param docs.

flexmeasures.data.models.forecasting.model_specs.naive

Functions

flexmeasures.data.models.forecasting.model_specs.naive.**naive_specs_configurator**(**kwargs)

Create and customize initial specs with OLS. See model_spec_factory for param docs.

Classes

class flexmeasures.data.models.forecasting.model_specs.naive.**Naive**(*args, **kwargs)

Naive prediction model for a single input feature that simply throws back the given feature. Under the hood, it uses linear regression by ordinary least squares, trained with points (0,0) and (1,1).

__init__(*args, **kwargs)

flexmeasures.data.models.forecasting.utils

Functions

`flexmeasures.data.models.forecasting.utils.check_data_availability`(*old_sensor_model*,
old_time_series_data_model,
forecast_start: *datetime*,
forecast_end: *datetime*,
query_window:
Tuple[datetime, datetime],
horizon: *timedelta*)

Check if enough data is available in the database in the first place, for training window and lagged variables. Otherwise, suggest new forecast period. TODO: we could also check regressor data, if we get regressor specs passed in here.

`flexmeasures.data.models.forecasting.utils.create_lags`(*n_lags*: *int*, *sensor*: *Sensor*, *horizon*:
timedelta, *resolution*: *timedelta*,
use_periodicity: *bool*) → *List[timedelta]*

List the lags for this asset type, using horizon and resolution information.

`flexmeasures.data.models.forecasting.utils.get_query_window`(*training_start*: *datetime*, *forecast_end*:
datetime, *lags*: *List[timedelta]*) →
Tuple[datetime, datetime]

Derive query window from start and end date, as well as lags (if any). This makes sure we have enough data for lagging and forecasting.

`flexmeasures.data.models.forecasting.utils.set_training_and_testing_dates`(*forecast_start*:
datetime, *training_and_testing_period*:
timedelta |
Tuple[datetime,
datetime]) →
Tuple[datetime,
datetime]

If needed (if *training_and_testing_period* is a *timedelta*), derive *training_start* and *testing_end* from *forecast_start*, otherwise simply return *training_and_testing_period*.

```
|-----forecast_horizon/belief_horizon-----| | |-----resolution-----| belief_time event_start
event_end

|--resolution--|--resolution--|--resolution--|--resolution--|--resolution--|--resolution--|

|-----forecast_horizon-----| |||| belief_time event_start |||||

|-----forecast_horizon-----| |||| belief_time event_start ||||

|-----forecast_horizon-----| |||
belief_time event_start |||

|-----max_lag-----|-----training_and_testing_period-----|-----forecast_period-----|
query_start training_start || testing_end/forecast_start | forecast_end

|-----min_lag-----| | |-----forecast_horizon-----| |

| belief_time event_start |||
|| |-----forecast_horizon-----| |
```



```

|| belief_time event_start ||
|| |-----forecast_horizon-----|
|| | belief_time event_start |
|-----query_window-----|

```

Functions

```

flexmeasures.data.models.forecasting.lookup_model_specs_configurator(model_search_term: str =
    'linear-OLS') →
    Callable[[...],
    Tuple[ModelSpecs, str,
    str]]

```

This function maps a model-identifying search term to a model configurator function, which can make model meta data. Why use a string? It might be stored on RQ jobs. It might also leave more freedom, we can then map multiple terms to the same model or vice versa (e.g. when different versions exist).

Model meta data in this context means a tuple of:

- `timetomodel.ModelSpecs`. To fill in those specs, a configurator should accept: - `old_sensor`: `Union[Asset, Market, WeatherSensor]`, - `start`: `datetime`, # Start of forecast period - `end`: `datetime`, # End of forecast period - `horizon`: `timedelta`, # Duration between time of forecasting and time which is forecast - `ex_post_horizon`: `timedelta` = `None`, - `custom_model_params`: `dict` = `None`, # overwrite forecasting params, useful for testing or experimentation
- a `model_identifier` (useful in case the `model_search_term` was generic, e.g. “latest”)
- **a fallback_model_search_term: a string which the forecasting machinery can use to choose a different model (using this mapping again) in case of failure.**

So to implement a model, write such a function and decide here which search term(s) map(s) to it.

flexmeasures.data.models.generic_assets

Functions

```

flexmeasures.data.models.generic_assets.assets_share_location(assets: List[GenericAsset]) →
    bool

```

Return True if all assets in this list are located on the same spot. TODO: In the future, we might soften this to compare if assets are in the same “housing” or “site”.

```

flexmeasures.data.models.generic_assets.create_generic_asset(generic_asset_type: str, **kwargs)
    → GenericAsset

```

Create a `GenericAsset` and assigns it an id.

Parameters

- **generic_asset_type** – “asset”, “market” or “weather_sensor”
- **kwargs** – should have values for keys “name”, and: - “asset_type_name” or “asset_type” when `generic_asset_type` is “asset” - “market_type_name” or “market_type” when `generic_asset_type` is “market” - “weather_sensor_type_name” or “weather_sensor_type” when `generic_asset_type` is “weather_sensor” - alternatively, “sensor_type” is also fine

Returns

the created `GenericAsset`

```
flexmeasures.data.models.generic_assets.get_center_location_of_assets(user: User | None) →  
                                         Tuple[float, float]
```

Find the center position between all generic assets of the user’s account.

Classes

```
class flexmeasures.data.models.generic_assets.GenericAsset(**kwargs)
```

An asset is something that has economic value.

Examples of tangible assets: a house, a ship, a weather station. Examples of intangible assets: a market, a country, a copyright.

```
__init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

```
add_annotations(df: DataFrame, annotation_type: str, commit_transaction: bool = False)
```

Add a data frame describing annotations to the database, and assign the annotations to this asset.

```
property asset_type: GenericAssetType
```

This property prepares for dropping the “generic” prefix later

```
chart(chart_type: str = 'chart_for_multiple_sensors', event_starts_after: datetime | None = None,  
      event_ends_before: datetime | None = None, beliefs_after: datetime | None = None, beliefs_before:  
      datetime | None = None, source: DataSource | List[DataSource] | int | List[int] | str | List[str] | None =  
      None, include_data: bool = False, dataset_name: str | None = None, **kwargs) → dict
```

Create a vega-lite chart showing sensor data.

Parameters

- **chart_type** – currently only “bar_chart” # todo: where can we properly list the available chart types?
- **event_starts_after** – only return beliefs about events that start after this datetime (inclusive)
- **event_ends_before** – only return beliefs about events that end before this datetime (inclusive)
- **beliefs_after** – only return beliefs formed after this datetime (inclusive)
- **beliefs_before** – only return beliefs formed before this datetime (inclusive)
- **source** – search only beliefs by this source (pass the DataSource, or its name or id) or list of sources
- **include_data** – if True, include data in the chart, or if False, exclude data
- **dataset_name** – optionally name the dataset used in the chart (the default name is sensor_<id>)

Returns

JSON string defining vega-lite chart specs

count_annotations(*annotation_starts_after: datetime | None = None, annotations_after: datetime | None = None, annotation_ends_before: datetime | None = None, annotations_before: datetime | None = None, source: DataSource | List[DataSource] | int | List[int] | str | List[str] | None = None, annotation_type: str | None = None*) → int

Count the number of annotations assigned to this asset.

classmethod get_timerange(*sensors: List['Sensor']*) → Dict[str, datetime]

Time range for which sensor data exists.

Parameters

sensors – sensors to check

Returns

dictionary with start and end, for example: {

```
'start': datetime.datetime(2020, 12, 3, 14, 0, tzinfo=pytz.utc), 'end': date-
time.datetime(2020, 12, 3, 14, 30, tzinfo=pytz.utc)
```

}

great_circle_distance(***kwargs*)

Query great circle distance (unclear if in km or in miles).

Can be called with an object that has latitude and longitude properties, for example:

```
great_circle_distance(object=asset)
```

Can also be called with latitude and longitude parameters, for example:

```
great_circle_distance(latitude=32, longitude=54) great_circle_distance(lat=32, lng=54)
```

Requires the following Postgres extensions: earthdistance and cube.

property has_energy_sensors: bool

True if at least one energy sensor is attached

property has_power_sensors: bool

True if at least one power sensor is attached

search_annotations(*annotations_after: datetime | None = None, annotations_before: datetime | None = None, source: DataSource | List[DataSource] | int | List[int] | str | List[str] | None = None, annotation_type: str | None = None, include_account_annotations: bool = False, as_frame: bool = False*) → List[Annotation] | DataFrame

Return annotations assigned to this asset, and optionally, also those assigned to the asset's account.

Parameters

- **annotations_after** – only return annotations that end after this datetime (exclusive)
- **annotations_before** – only return annotations that start before this datetime (exclusive)

search_beliefs(*sensors: List['Sensor'] | None = None, event_starts_after: datetime | None = None, event_ends_before: datetime | None = None, beliefs_after: datetime | None = None, beliefs_before: datetime | None = None, horizons_at_least: timedelta | None = None, horizons_at_most: timedelta | None = None, source: DataSource | List[DataSource] | int | List[int] | str | List[str] | None = None, most_recent_beliefs_only: bool = True, most_recent_events_only: bool = False, as_json: bool = False*) → BeliefsDataFrame | str

Search all beliefs about events for all sensors of this asset

If you don't set any filters, you get the most recent beliefs about all events.

Parameters

- **sensors** – only return beliefs about events registered by these sensors
- **event_starts_after** – only return beliefs about events that start after this datetime (inclusive)
- **event_ends_before** – only return beliefs about events that end before this datetime (inclusive)
- **beliefs_after** – only return beliefs formed after this datetime (inclusive)
- **beliefs_before** – only return beliefs formed before this datetime (inclusive)
- **horizons_at_least** – only return beliefs with a belief horizon equal or greater than this timedelta (for example, use timedelta(0) to get ante knowledge time beliefs)
- **horizons_at_most** – only return beliefs with a belief horizon equal or less than this timedelta (for example, use timedelta(0) to get post knowledge time beliefs)
- **source** – search only beliefs by this source (pass the DataSource, or its name or id) or list of sources
- **most_recent_events_only** – only return (post knowledge time) beliefs for the most recent event (maximum event start)
- **as_json** – return beliefs in JSON format (e.g. for use in charts) rather than as BeliefsDataFrame

Returns

dictionary of BeliefsDataFrames or JSON string (if as_json is True)

property sensors_to_show: `list['Sensor' | list['Sensor']]`

Sensors to show, as defined by the sensors_to_show attribute.

Sensors to show are defined as a list of sensor ids, which is set by the “sensors_to_show” field of the asset’s “attributes” column. Valid sensors either belong to the asset itself, to other assets in the same account, or to public assets. In case the field is missing, defaults to two of the asset’s sensors.

Sensor ids can be nested to denote that sensors should be ‘shown together’, for example, layered rather than vertically concatenated. How to interpret ‘shown together’ is technically left up to the function returning chart specs, as are any restrictions regarding what sensors can be shown together, such as: - whether they should share the same unit - whether they should share the same name - whether they should belong to different assets

For example, this denotes showing sensors 42 and 44 together:

```
sensors_to_show = [40, 35, 41, [42, 44], 43, 45]
```

property timerange: `Dict[str, datetime]`

Time range for which sensor data exists.

Returns

dictionary with start and end, for example: {

```
    'start': datetime.datetime(2020, 12, 3, 14, 0, tzinfo=pytz.utc), 'end': datetime.datetime(2020, 12, 3, 14, 30, tzinfo=pytz.utc)
```

```
}
```

property timerange_of_sensors_to_show: `Dict[str, datetime]`

Time range for which sensor data exists, for sensors to show.

Returns

dictionary with start and end, for example: {

```

        'start': datetime.datetime(2020, 12, 3, 14, 0, tzinfo=pytz.utc), 'end': date-
        time.datetime(2020, 12, 3, 14, 30, tzinfo=pytz.utc)
    }

```

property timezone: `str`

Timezone relevant to the asset.

If a timezone is not given as an attribute of the asset, it is taken from one of its sensors.

class `flexmeasures.data.models.generic_assets.GenericAssetType(**kwargs)`

An asset type defines what type an asset belongs to.

Examples of asset types: WeatherStation, Market, CP, EVSE, WindTurbine, SolarPanel, Building.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

flexmeasures.data.models.legacy_migration_utils

This module is part of our data model migration (see <https://github.com/SeitaBV/flexmeasures/projects/9>). It will become obsolete when Assets, Markets and WeatherSensors can no longer be initialized.

Functions

`flexmeasures.data.models.legacy_migration_utils.copy_old_sensor_attributes`(*old_sensor*,
old_sensor_type_attributes:
List[str],
old_sensor_attributes:
List[str],
old_sensor_type:
AssetType |
MarketType |
WeatherSen-
sorType = None)
→ dict

Parameters

- **old_sensor** – an Asset, Market or WeatherSensor instance
- **old_sensor_type_attributes** – names of attributes of the old sensor's type that should be copied
- **old_sensor_attributes** – names of attributes of the old sensor that should be copied
- **old_sensor_type** – the old sensor's type

Returns

dictionary containing an “attributes” dictionary with attribute names and values

```
flexmeasures.data.models.legacy_migration_utils.get_old_model_type(kwargs: dict,  
                                                                    old_sensor_type_class:  
                                                                    Type[AssetType |  
                                                                    MarketType |  
                                                                    WeatherSensorType],  
                                                                    old_sensor_type_name_key:  
                                                                    str, old_sensor_type_key:  
                                                                    str) → AssetType |  
                                                                    MarketType |  
                                                                    WeatherSensorType
```

Parameters

- **kwargs** – keyword arguments used to initialize a new Asset, Market or WeatherSensor
- **old_sensor_type_class** – AssetType, MarketType, or WeatherSensorType
- **old_sensor_type_name_key** – “asset_type_name”, “market_type_name”, or “weather_sensor_type_name”
- **old_sensor_type_key** – “asset_type”, “market_type”, or “sensor_type” (instead of “weather_sensor_type”), i.e. the name of the class attribute for the db.relationship to the type’s class

Returns

the old sensor’s type

flexmeasures.data.models.markets

Classes

```
class flexmeasures.data.models.markets.Market(**kwargs)
```

Each market is a pricing service.

This model is now considered legacy. See GenericAsset and Sensor.

```
__init__(**kwargs)
```

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

```
property entity_address: str
```

Entity address under the latest fm scheme for entity addresses.

```
property entity_address_fm0: str
```

Entity address under the fm0 scheme for entity addresses.

```
event_resolution: timedelta
```

```
get_attribute(attribute: str)
```

Looks for the attribute on the corresponding Sensor.

This should be used by all code to read these attributes, over accessing them directly on this class, as this table is in the process to be replaced by the Sensor table.

id

knowledge_horizon_fnc: `str`

knowledge_horizon_par: `dict`

name: `str`

property price_unit: `str`

Return the ‘unit’ property of the generic asset, just with a more insightful name.

timezone: `str`

unit: `str`

class flexmeasures.data.models.markets.**MarketType**(**kwargs)

Describing market types for our purposes. This model is now considered legacy. See GenericAssetType.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

property preconditions: `Dict[str, bool]`

Assumptions about the time series data set, such as normality and stationarity For now, this is usable input for Prophet (see init), but it might evolve or go away.

class flexmeasures.data.models.markets.**Price**(use_legacy_kwargs=True, **kwargs)

All prices are stored in one slim table.

This model is now considered legacy. See TimedBelief.

__init__(use_legacy_kwargs=True, **kwargs)

data_source_id

datetime

horizon

classmethod make_query(**kwargs) → Query

Construct the database query.

value

flexmeasures.data.models.parsing_utils

Functions

flexmeasures.data.models.parsing_utils.**parse_source_arg**(source: `DataSource` | `int` | `str` | `Sequence[DataSource]` | `Sequence[int]` | `Sequence[str]` | `None`) → `list[DataSource]` | `None`

Parse the “source” argument by looking up DataSources corresponding to any given ids or names.

Passes None as is (i.e. no source argument is given). Accepts ids and names as list or tuples, always converting them to a list.

flexmeasures.data.models.planning

Modules

flexmeasures.data.models.planning.battery

*flexmeasures.data.models.planning.
charging_station*

*flexmeasures.data.models.planning.
exceptions*

*flexmeasures.data.models.planning.
linear_optimization*

flexmeasures.data.models.planning.storage

flexmeasures.data.models.planning.utils

flexmeasures.data.models.planning.battery

Functions

flexmeasures.data.models.planning.battery.schedule_battery(*args, **kwargs)

flexmeasures.data.models.planning.charging_station

Functions

flexmeasures.data.models.planning.charging_station.schedule_charging_station(*args,
**kwargs)

flexmeasures.data.models.planning.exceptions

Exceptions

exception flexmeasures.data.models.planning.exceptions.MissingAttributeException

exception flexmeasures.data.models.planning.exceptions.UnknownForecastException

exception flexmeasures.data.models.planning.exceptions.UnknownMarketException

exception flexmeasures.data.models.planning.exceptions.UnknownPricesException

exception flexmeasures.data.models.planning.exceptions.WrongTypeAttributeException

flexmeasures.data.models.planning.linear_optimization

Functions

`flexmeasures.data.models.planning.linear_optimization.device_scheduler`(*device_constraints*: *List[DataFrame]*, *ems_constraints*: *DataFrame*, *commitment_quantities*: *List[Series]*, *commitment_downwards_deviation_price*: *List[Series]* | *List[float]*, *commitment_upwards_deviation_price*: *List[Series]* | *List[float]*, *initial_stock*: *float = 0*)
 → *Tuple[List[Series], float, SolverResults]*

This generic device scheduler is able to handle an EMS with multiple devices, with various types of constraints on the EMS level and on the device level, and with multiple market commitments on the EMS level. A typical example is a house with many devices. The commitments are assumed to be with regard to the flow of energy to the device (positive for consumption, negative for production). The solver minimises the costs of deviating from the commitments.

Device constraints are on a device level. Handled constraints (listed by column name):

max: maximum stock assuming an initial stock of zero (e.g. in MWh or boxes) min: minimum stock assuming an initial stock of zero equal: exact amount of stock (we do this by clamping min and max) efficiency: amount of stock left at the next datetime (the rest is lost) derivative max: maximum flow (e.g. in MW or boxes/h) derivative min: minimum flow derivative equals: exact amount of flow (we do this by clamping derivative min and derivative max) derivative down efficiency: conversion efficiency of flow out of a device (flow out : stock decrease) derivative up efficiency: conversion efficiency of flow into a device (stock increase : flow in)

EMS constraints are on an EMS level. Handled constraints (listed by column name):

derivative max: maximum flow derivative min: minimum flow

Commitments are on an EMS level. Parameter explanations:

commitment_quantities: amounts of flow specified in commitments (both previously ordered and newly requested)

- e.g. in MW or boxes/h

commitment_downwards_deviation_price: penalty for downwards deviations of the flow

- e.g. in EUR/MW or EUR/(boxes/h)
- either a single value (same value for each flow value) or a Series (different value for each flow value)

commitment_upwards_deviation_price: penalty for upwards deviations of the flow

All Series and DataFrames should have the same resolution.

For now, we pass in the various constraints and prices as separate variables, from which we make a MultiIndex DataFrame. Later we could pass in a MultiIndex DataFrame directly.

flexmeasures.data.models.planning.storage

Functions

`flexmeasures.data.models.planning.storage.add_storage_constraints`(*start: datetime, end: datetime, resolution: timedelta, soc_at_start: float, soc_targets: list[dict[str, datetime | float]] | pd.Series | None, soc_maxima: list[dict[str, datetime | float]] | pd.Series | None, soc_minima: list[dict[str, datetime | float]] | pd.Series | None, soc_max: float, soc_min: float*) → `pd.DataFrame`

Collect all constraints for a given storage device in a DataFrame that the device_scheduler can interpret.

Parameters

- **start** – Start of the schedule.
- **end** – End of the schedule.
- **resolution** – Timedelta used to resample the forecasts to the resolution of the schedule.
- **soc_at_start** – State of charge at the start time.
- **soc_targets** – Exact targets for the state of charge at each time.
- **soc_maxima** – Maximum state of charge at each time.
- **soc_minima** – Minimum state of charge at each time.
- **soc_max** – Maximum state of charge at all times.
- **soc_min** – Minimum state of charge at all times.

Returns

Constraints (StorageScheduler.COLUMNS) for a storage device, at each time step (index). See device_scheduler for possible column names.

`flexmeasures.data.models.planning.storage.build_device_soc_targets`(*targets: list[dict[str, datetime | float]] | pd.Series, soc_at_start: float, start_of_schedule: datetime, end_of_schedule: datetime, resolution: timedelta*) → `pd.Series`

`flexmeasures.data.models.planning.storage.build_device_soc_values`(*soc_values: list[dict[str, datetime | float]] | pd.Series, soc_at_start: float, start_of_schedule: datetime, end_of_schedule: datetime, resolution: timedelta*) → `pd.Series`

Utility function to create a Pandas series from SOC values we got from the flex-model.

Should set NaN anywhere where there is no target.

SOC values should be indexed by their due date. For example, for quarter-hourly targets between 5 and 6 AM: >>> df = pd.Series(data=[1, 2, 2.5, 3], index=pd.date_range(datetime(2010,1,1,5), datetime(2010,1,1,6), freq=timedelta(minutes=15), inclusive="right")) >>> print(df)

```
2010-01-01 05:15:00 1.0 2010-01-01 05:30:00 2.0 2010-01-01 05:45:00 2.5 2010-01-01 06:00:00 3.0
Freq: 15T, dtype: float64
```

TODO: this function could become the deserialization method of a new SOCValueSchema (targets, plural), which wraps SOCValueSchema.

flexmeasures.data.models.planning.storage.**create_constraint_violations_message**(*constraint_violations: list*) → str

Create a human-readable message with the constraint_violations.

Parameters

constraint_violations – list with the constraint violations

Returns

human-readable message

flexmeasures.data.models.planning.storage.**get_pattern_match_word**(*word: str*) → str

Get a regex pattern to match a word

The conditions to delimit a word are:

- start of line
- whitespace
- end of line
- word boundary
- arithmetic operations

Returns

regex expression

flexmeasures.data.models.planning.storage.**prepend_serie**(*serie: Series, value*) → Series

Prepend a value to a time series series

Parameters

- **serie** – serie containing the timed values
- **value** – value to place in the first position

flexmeasures.data.models.planning.storage.**sanitize_expression**(*expression: str, columns: list*) → tuple[str, list]

Wrap column in commas to accept arbitrary column names (e.g. with spaces).

Parameters

- **expression** – expression to sanitize
- **columns** – list with the name of the columns of the input data for the expression.

Returns

sanitized expression and columns (variables) used in the expression

```
flexmeasures.data.models.planning.storage.validate_constraint(constraints_df: pd.DataFrame,  
                                                             lhs_expression: str, inequality: str,  
                                                             rhs_expression: str,  
                                                             round_to_decimals: int | None =  
                                                             6) → list[dict]
```

Validate the feasibility of a given set of constraints.

Parameters

- **constraints_df** – DataFrame with the constraints
- **lhs_expression** – left-hand side of the inequality expression following pd.eval format. No need to use the syntax *column* to reference column, just use the column name.
- **inequality** – inequality operator, one of ('<=', '<', '>=', '>', '==', '!=').
- **rhs_expression** – right-hand side of the inequality expression following pd.eval format. No need to use the syntax *column* to reference column, just use the column name.
- **round_to_decimals** – Number of decimals to round off to before validating constraints.

Returns

List of constraint violations, specifying their time, constraint and violation.

```
flexmeasures.data.models.planning.storage.validate_storage_constraints(constraints:  
                                                                       DataFrame,  
                                                                       soc_at_start: float,  
                                                                       soc_min: float,  
                                                                       soc_max: float,  
                                                                       resolution: timedelta)  
→ list[dict]
```

Check that the storage constraints are fulfilled, e.g min <= equals <= max.

A. Global validation

A.1) min >= soc_min A.2) max <= soc_max

B. Validation in the same time frame

B.1) min <= max B.2) min <= equals B.3) equals <= max

C. Validation in different time frames

C.1) equals(t) - equals(t-1) <= derivative_max(t) C.2) derivative_min(t) <= equals(t) - equals(t-1) C.3) min(t) - max(t-1) <= derivative_max(t) C.4) max(t) - min(t-1) >= derivative_min(t) C.5) equals(t) - max(t-1) <= derivative_max(t) C.6) derivative_min(t) <= equals(t) - min(t-1)

Parameters

- **constraints** – dataframe containing the constraints of a storage device
- **soc_at_start** – State of charge at the start time.
- **soc_min** – Minimum state of charge at all times.
- **soc_max** – Maximum state of charge at all times.
- **resolution** – Constant duration between the start of each time step.

Returns

List of constraint violations, specifying their time, constraint and violation.

Classes

```
class flexmeasures.data.models.planning.storage.StorageScheduler(sensor, start, end, resolution,
                                                                belief_time: datetime | None =
                                                                None, round_to_decimals: int |
                                                                None = 6, flex_model: dict |
                                                                None = None, flex_context: dict
                                                                | None = None)
```

compute(skip_validation: *bool* = *False*) → *pd.Series* | *None*

Schedule a battery or Charge Point based directly on the latest beliefs regarding market prices within the specified time window. For the resulting consumption schedule, consumption is defined as positive values.

Parameters

skip_validation – If True, skip validation of constraints specified in the data.

Returns

The computed schedule.

compute_schedule() → *pd.Series* | *None*

Schedule a battery or Charge Point based directly on the latest beliefs regarding market prices within the specified time window. For the resulting consumption schedule, consumption is defined as positive values.

Deprecated method in v0.14. As an alternative, use `StorageScheduler.compute()`.

deserialize_flex_config()

Deserialize storage flex model and the flex context against schemas. Before that, we fill in values from wider context, if possible. Mostly, we allow several fields to come from sensor attributes. TODO: this work could maybe go to the schema as a pre-load hook (if we pass in the sensor to schema initialization)

Note: Before we apply the flex config schemas, we need to use the flex config identifiers with hyphens, (this is how they are represented to outside, e.g. by the API), after deserialization we use internal schema names (with underscores).

ensure_soc_min_max()

Make sure we have min and max SOC. If not passed directly, then get default from sensor or targets.

persist_flex_model()

Store new soc info as GenericAsset attributes

possibly_extend_end()

Extend schedule period in case a target exceeds its end.

The schedule's duration is possibly limited by the server config setting 'FLEXMEASURES_MAX_PLANNING_HORIZON'.

todo: when `deserialize_flex_config` becomes a single schema for the whole scheduler, this function would become a class method with a `@post_load` decorator.

flexmeasures.data.models.planning.utils

Functions

`flexmeasures.data.models.planning.utils.add_tiny_price_slope(prices: DataFrame, col_name: str = 'event_value', d: float = 0.001) → DataFrame`

Add tiny price slope to `col_name` to represent e.g. inflation as a simple linear price increase. This is meant to break ties, when multiple time slots have equal prices, in favour of acting sooner. We penalise the future with at most `d` times the price spread (1 per thousand by default).

`flexmeasures.data.models.planning.utils.fallback_charging_policy(sensor: Sensor, device_constraints: DataFrame, start: datetime, end: datetime, resolution: timedelta) → Series`

This fallback charging policy is to just start charging or discharging, or do neither, depending on the first target state of charge and the capabilities of the Charge Point. Note that this ignores any cause of the infeasibility and, while probably a decent policy for Charge Points, should not be considered a robust policy for other asset types.

`flexmeasures.data.models.planning.utils.get_market(sensor: Sensor) → Sensor`

Get market sensor from the sensor's attributes.

`flexmeasures.data.models.planning.utils.get_power_values(query_window: Tuple[datetime, datetime], resolution: timedelta, beliefs_before: datetime | None, sensor: Sensor) → ndarray`

Get measurements or forecasts of an inflexible device represented by a power sensor.

If the requested schedule lies in the future, the returned data will consist of (the most recent) forecasts (if any exist). If the requested schedule lies in the past, the returned data will consist of (the most recent) measurements (if any exist). The latter amounts to answering “What if we could have scheduled under perfect foresight?”.

Parameters

- **query_window** – datetime window within which events occur (equal to the scheduling window)
- **resolution** – timedelta used to resample the forecasts to the resolution of the schedule
- **beliefs_before** – datetime used to indicate we are interested in the state of knowledge at that time
- **sensor** – power sensor representing an energy flow out of the device

Returns

power measurements or forecasts (consumption is positive, production is negative)

`flexmeasures.data.models.planning.utils.get_prices(query_window: Tuple[datetime, datetime], resolution: timedelta, beliefs_before: datetime | None, price_sensor: Sensor | None = None, sensor: Sensor | None = None, allow_trimmed_query_window: bool = True) → Tuple[DataFrame, Tuple[datetime, datetime]]`

Check for known prices or price forecasts.

If so allowed, the query window is trimmed according to the available data. If not allowed, prices are extended to the edges of the query window: - The first available price serves as a naive backcast. - The last available price serves as a naive forecast.

```
flexmeasures.data.models.planning.utils.idle_after_reaching_target(schedule: Series, target:
                                                                    Series, initial_state: float =
                                                                    0) → Series
```

Stop planned (dis)charging after target is reached (or constraint is met).

```
flexmeasures.data.models.planning.utils.initialize_df(columns: List[str], start: datetime, end:
                                                       datetime, resolution: timedelta, inclusive: str
                                                       = 'left') → DataFrame
```

```
flexmeasures.data.models.planning.utils.initialize_index(start: date | datetime | str, end: date |
                                                         datetime | str, resolution: timedelta | str,
                                                         inclusive: str = 'left') → DatetimeIndex
```

```
flexmeasures.data.models.planning.utils.initialize_series(data: Series | List[float] | ndarray | float
                                                         | None, start: datetime, end: datetime,
                                                         resolution: timedelta, inclusive: str =
                                                         'left') → Series
```

Classes

```
class flexmeasures.data.models.planning.Scheduler(sensor, start, end, resolution, belief_time: datetime
                                                    | None = None, round_to_decimals: int | None = 6,
                                                    flex_model: dict | None = None, flex_context: dict |
                                                    None = None)
```

Superclass for all FlexMeasures Schedulers.

A scheduler currently computes the schedule for one flexible asset. TODO: extend to multiple flexible assets.

The scheduler knows the power sensor of the flexible asset. It also knows the basic timing parameter of the schedule (start, end, resolution), including the point in time when knowledge can be assumed to be available (belief_time).

Furthermore, the scheduler needs to have knowledge about the asset's flexibility model (under what constraints can the schedule be optimized?) and the system's flexibility context (which other sensors are relevant, e.g. prices). These two flexibility configurations are usually fed in from outside, so the scheduler should check them. The `deserialize_flex_config` function can be used for that.

```
__init__(sensor, start, end, resolution, belief_time: datetime | None = None, round_to_decimals: int | None
         = 6, flex_model: dict | None = None, flex_context: dict | None = None)
```

Initialize a new Scheduler.

TODO: We might adapt the class design, so that a Scheduler object is initialized with configuration parameters,

and can then be used multiple times (via `compute()`) to compute schedules of different kinds, e.g.

If we started later (put in a later start), what would the schedule be? If we could change set points less often (put in a coarser resolution), what would the schedule be? If we knew what was going to happen (put in a later belief_time), what would the schedule have been?

For now, we don't see the best separation between config and state parameters (esp. within flex models) E.g. `start` and `flex_model[soc_at_start]` are intertwined.

compute() → Series | None

Overwrite with the actual computation of your schedule.

compute_schedule() → Series | None

Overwrite with the actual computation of your schedule.

Deprecated method in v0.14. As an alternative, use Scheduler.compute().

deserialize_config()

Check all configurations we have, throwing either ValidationErrors or ValueErrors. Other code can decide if/how to handle those.

deserialize_flex_config()

Check if the flex model and flex context are valid. Should be overwritten.

Ideas: - Apply a schema to check validity (see in-built flex model schemas) - Check for inconsistencies between settings (can also happen in Marshmallow) - fill in missing values from the scheduler's knowledge (e.g. sensor attributes)

Raises ValidationErrors or ValueErrors.

deserialize_timing_config()

Check if the timing of the schedule is valid. Raises ValueErrors.

classmethod get_data_source_info() → dict

Create and return the data source info, from which a data source lookup/creation is possible. See for instance get_data_source_for_job().

persist_flex_model()

If useful, (parts of) the flex model can be persisted here, e.g. as asset attributes, sensor attributes or as sensor data (beliefs).

flexmeasures.data.models.reporting

Modules

```
flexmeasures.data.models.reporting.  
aggregator  
flexmeasures.data.models.reporting.  
pandas_reporter
```

flexmeasures.data.models.reporting.aggregator

Classes

class flexmeasures.data.models.reporting.aggregator.**AggregatorReporter**(*sensor*: Sensor,
reporter_config_raw:
dict | None = None)

This reporter applies an aggregation function to multiple sensors

_compute(*start: datetime, end: datetime, input_resolution: timedelta | None = None, belief_time: datetime | None = None*) → `tb.BeliefsDataFrame`

This method merges all the `BeliefDataFrames` into a single one, dropping all indexes but `event_start`, and applies an aggregation function over the columns.

deserialize_config()

Validate the report config against a Marshmallow Schema. Ideas: - Override this method - Call superclass method to apply validation and common variables deserialization (see `PandasReporter`) - (Partially) extract the relevant `reporter_config` parameters into class attributes.

Raises `ValidationErrors` or `ValueErrors`.

flexmeasures.data.models.reporting.pandas_reporter

Classes

class `flexmeasures.data.models.reporting.pandas_reporter.PandasReporter`(*sensor: Sensor, reporter_config_raw: dict | None = None*)

This reporter applies a series of pandas methods on

_apply_transformations()

Convert the series using the given list of transformation specs, which is called in the order given.

Each transformation specs should include a ‘method’ key specifying a method name of a Pandas DataFrame.

Optionally, ‘args’ and ‘kwargs’ keys can be specified to pass on arguments or keyword arguments to the given method.

All data exchange is made through the dictionary `self.data`. The superclass `Reporter` already fetches `BeliefsDataFrames` of the sensors and saves them in the `self.data` dictionary fields `sensor_<sensor_id>`. In case you need to perform complex operations on dataframes, you can split the operations in several steps and saving the intermediate results using the parameters `df_input` and `df_output` for the input and output dataframes, respectively.

Example:

The example below converts from hourly meter readings in kWh to electricity demand in kW.

```
transformations = [
    {"method": "diff"}, {"method": "shift", "kwargs": {"periods": -1}}, {"method": "head", "args": [-1]},
]
```

_compute(*start: datetime, end: datetime, input_resolution: timedelta | None = None, belief_time: datetime | None = None*) → `tb.BeliefsDataFrame`

This method applies the transformations and outputs the dataframe defined in `final_df_output` field of the `report_config`.

_process_pandas_args(*args: list, method: str*) → `list`

This method applies the function `get_object_or_literal` to all the arguments to detect where to replace a string “@<object-name>” with the actual object stored in `self.data[“<object-name>”]`.

_process_pandas_kwargs(*kwargs: dict, method: str*) → `dict`

This method applies the function `get_object_or_literal` to all the keyword arguments to detect where to replace a string “@<object-name>” with the actual object stored in `self.data[“<object-name>”]`.

deserialize_config()

Validate the report config against a Marshmallow Schema. Ideas: - Override this method - Call superclass method to apply validation and common variables deserialization (see PandasReporter) - (Partially) extract the relevant reporter_config parameters into class attributes.

Raises ValidationErrors or ValueErrors.

get_object_or_literal(value: *Any*, method: *str*) → *Any*

This method allows using the dataframes as inputs of the Pandas methods that are run in the transformations. Make sure that they have been created before accessed.

This works by putting the symbol @ in front of the name of the dataframe that we want to reference. For instance, to reference the dataframe test_df, which lives in self.data, we would do @test_df.

This functionality is disabled for methods *eval* and *query* to avoid interfering their internal behaviour given that they also use @ to allow using local variables.

Example: >>> self.get_object_or_literal(["@df_wind", "@df_solar"], "sum") [<BeliefsDataFrame for Wind Turbine sensor>, <BeliefsDataFrame for Solar Panel sensor>]

Classes

class flexmeasures.data.models.reporting.**Reporter**(sensor: *Sensor*, reporter_config_raw: *dict* | *None* = *None*)

Superclass for all FlexMeasures Reporters.

__init__(sensor: *Sensor*, reporter_config_raw: *dict* | *None* = *None*) → *None*

Initialize a new Reporter.

Attributes: :param sensor: sensor where the output of the reporter will be saved to. :param reporter_config_raw: dictionary with the serialized configuration of the reporter.

_compute(start: *datetime*, end: *datetime*, input_resolution: *timedelta* | *None* = *None*, belief_time: *datetime* | *None* = *None*) → BeliefsDataFrame

Overwrite with the actual computation of your report.

Returns BeliefsDataFrame

report as a BeliefsDataFrame.

compute(start: *datetime*, end: *datetime*, input_resolution: *timedelta* | *None* = *None*, belief_time: *datetime* | *None* = *None*, ***kwargs*) → tb.BeliefsDataFrame

This method triggers the creation of a new report.

The same object can generate multiple reports with different start, end, input_resolution and belief_time values.

In the future, this function will parse arbitrary input arguments defined in a schema.

deserialize_config()

Validate the report config against a Marshmallow Schema. Ideas: - Override this method - Call superclass method to apply validation and common variables deserialization (see PandasReporter) - (Partially) extract the relevant reporter_config parameters into class attributes.

Raises ValidationErrors or ValueErrors.

fetch_data(start: *datetime*, end: *datetime*, input_resolution: *timedelta* | *None* = *None*, belief_time: *datetime* | *None* = *None*)

Fetches the time_beliefs from the database

flexmeasures.data.models.task_runs

Classes

class flexmeasures.data.models.task_runs.LatestTaskRun(**kwargs)

” Log the (latest) running of a task. This is intended to be used for live monitoring. For a full analysis, there are log files.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

static record_run(task_name: str, status: bool)

Record the latest task run (overwriting previous ones). If the row is not yet in the table, create it first. Does not commit.

flexmeasures.data.models.time_series

Classes

class flexmeasures.data.models.time_series.Sensor(name, generic_asset=None, generic_asset_id=None, attributes=None, **kwargs)

A sensor measures events.

__init__(name, generic_asset=None, generic_asset_id=None, attributes=None, **kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

chart(chart_type: str = 'bar_chart', event_starts_after: datetime_type | None = None, event_ends_before: datetime_type | None = None, beliefs_after: datetime_type | None = None, beliefs_before: datetime_type | None = None, source: DataSource | list[DataSource] | int | list[int] | str | list[str] | None = None, most_recent_beliefs_only: bool = True, include_data: bool = False, include_sensor_annotations: bool = False, include_asset_annotations: bool = False, include_account_annotations: bool = False, dataset_name: str | None = None, **kwargs) → dict

Create a vega-lite chart showing sensor data.

Parameters

- **chart_type** – currently only “bar_chart” # todo: where can we properly list the available chart types?
- **event_starts_after** – only return beliefs about events that start after this datetime (inclusive)
- **event_ends_before** – only return beliefs about events that end before this datetime (inclusive)
- **beliefs_after** – only return beliefs formed after this datetime (inclusive)

- **beliefs_before** – only return beliefs formed before this datetime (inclusive)
- **source** – search only beliefs by this source (pass the DataSource, or its name or id) or list of sources
- **most_recent_beliefs_only** – only return the most recent beliefs for each event from each source (minimum belief horizon)
- **include_data** – if True, include data in the chart, or if False, exclude data
- **include_sensor_annotations** – if True and include_data is True, include sensor annotations in the chart, or if False, exclude these
- **include_asset_annotations** – if True and include_data is True, include asset annotations in the chart, or if False, exclude them
- **include_account_annotations** – if True and include_data is True, include account annotations in the chart, or if False, exclude them
- **dataset_name** – optionally name the dataset used in the chart (the default name is sensor_<id>)

Returns

JSON string defining vega-lite chart specs

check_required_attributes(attributes: list[str] | tuple[str, Type] | tuple[Type, ...]))

Raises if any attribute in the list of attributes is missing, or has the wrong type.

Parameters

attributes – List of either an attribute name or a tuple of an attribute name and its allowed type (the allowed type may also be a tuple of several allowed types)

event_resolution: `timedelta`

classmethod find_closest(generic_asset_type_name: str, sensor_name: str, n: int = 1, **kwargs) → 'Sensor' | list['Sensor'] | None

Returns the closest n sensors within a given asset type (as a list if n > 1). Parses latitude and longitude values stated in kwargs.

Can be called with an object that has latitude and longitude properties, for example:

```
sensor = Sensor.find_closest("weather_station", "wind speed", object=generic_asset)
```

Can also be called with latitude and longitude parameters, for example:

```
sensor = Sensor.find_closest("weather_station", "temperature", latitude=32, longitude=54)
sensor = Sensor.find_closest("weather_station", "temperature", lat=32, lng=54)
```

Finally, pass in an account_id parameter if you want to query an account other than your own. This only works for admins. Public assets are always queried.

get_attribute(attribute: str, default: Any | None = None) → Any

Looks for the attribute on the Sensor. If not found, looks for the attribute on the Sensor's GenericAsset. If not found, returns the default.

id

property is_strictly_non_negative: `bool`

Return True if this sensor strictly records non-negative values.

property is_strictly_non_positive: `bool`

Return True if this sensor strictly records non-positive values.

knowledge_horizon_fnc: `str`

knowledge_horizon_par: `dict`

latest_state(*source*: `DataSource` | *list*[`DataSource`] | *int* | *list*[*int*] | *str* | *list*[*str*] | *None* = *None*) → `tb.BeliefsDataFrame`

Search the most recent event for this sensor, and return the most recent ex-post belief.

Parameters

source – search only beliefs by this source (pass the `DataSource`, or its name or id) or list of sources

make_hashable() → `tuple`

Returns a tuple with the properties subject to change. In principle all properties (except ID) of a given sensor could be changed, but not all changes are relevant to warrant reanalysis (e.g. scheduling or forecasting).

property_measures_energy: `bool`

True if this sensor's unit is measuring energy

property_measures_power: `bool`

True if this sensor's unit is measuring power

name: `str`

search_annotations(*annotation_starts_after*: `datetime_type` | *None* = *None*, *annotations_after*: `datetime_type` | *None* = *None*, *annotation_ends_before*: `datetime_type` | *None* = *None*, *annotations_before*: `datetime_type` | *None* = *None*, *source*: `DataSource` | *list*[`DataSource`] | *int* | *list*[*int*] | *str* | *list*[*str*] | *None* = *None*, *include_asset_annotations*: `bool` = *False*, *include_account_annotations*: `bool` = *False*, *as_frame*: `bool` = *False*) → *list*[`Annotation`] | `pd.DataFrame`

Return annotations assigned to this sensor, and optionally, also those assigned to the sensor's asset and the asset's account.

Parameters

- **annotations_after** – only return annotations that end after this datetime (exclusive)
- **annotations_before** – only return annotations that start before this datetime (exclusive)

search_beliefs(*event_starts_after*: `datetime_type` | *None* = *None*, *event_ends_before*: `datetime_type` | *None* = *None*, *beliefs_after*: `datetime_type` | *None* = *None*, *beliefs_before*: `datetime_type` | *None* = *None*, *horizons_at_least*: `timedelta` | *None* = *None*, *horizons_at_most*: `timedelta` | *None* = *None*, *source*: `DataSource` | *list*[`DataSource`] | *int* | *list*[*int*] | *str* | *list*[*str*] | *None* = *None*, *most_recent_beliefs_only*: `bool` = *True*, *most_recent_events_only*: `bool` = *False*, *most_recent_only*: `bool` = *None*, *one_deterministic_belief_per_event*: `bool` = *False*, *one_deterministic_belief_per_event_per_source*: `bool` = *False*, *resolution*: `str` | `timedelta` = *None*, *as_json*: `bool` = *False*) → `tb.BeliefsDataFrame` | `str`

Search all beliefs about events for this sensor.

If you don't set any filters, you get the most recent beliefs about all events.

Parameters

- **event_starts_after** – only return beliefs about events that start after this datetime (inclusive)
- **event_ends_before** – only return beliefs about events that end before this datetime (inclusive)
- **beliefs_after** – only return beliefs formed after this datetime (inclusive)

- **beliefs_before** – only return beliefs formed before this datetime (inclusive)
- **horizons_at_least** – only return beliefs with a belief horizon equal or greater than this timedelta (for example, use timedelta(0) to get ante knowledge time beliefs)
- **horizons_at_most** – only return beliefs with a belief horizon equal or less than this timedelta (for example, use timedelta(0) to get post knowledge time beliefs)
- **source** – search only beliefs by this source (pass the DataSource, or its name or id) or list of sources
- **most_recent_beliefs_only** – only return the most recent beliefs for each event from each source (minimum belief horizon)
- **most_recent_events_only** – only return (post knowledge time) beliefs for the most recent event (maximum event start)
- **one_deterministic_belief_per_event** – only return a single value per event (no probabilistic distribution and only 1 source)
- **one_deterministic_belief_per_event_per_source** – only return a single value per event per source (no probabilistic distribution)
- **as_json** – return beliefs in JSON format (e.g. for use in charts) rather than as BeliefsDataFrame

Returns

BeliefsDataFrame or JSON string (if as_json is True)

property timerange: `dict[str, datetime.datetime]`

Time range for which sensor data exists.

Returns

dictionary with start and end, for example: {

```
    'start': datetime.datetime(2020, 12, 3, 14, 0, tzinfo=pytz.utc), 'end': datetime.datetime(2020, 12, 3, 14, 30, tzinfo=pytz.utc)
```

}

timezone: `str`

unit: `str`

class flexmeasures.data.models.time_series.TimedBelief(sensor, source, **kwargs)

A timed belief holds a precisely timed record of a belief about an event.

It also records the source of the belief, and the sensor that the event pertains to.

__init__(sensor, source, **kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

classmethod add(bdf: BeliefsDataFrame, expunge_session: bool = False, allow_overwrite: bool = False, bulk_save_objects: bool = False, commit_transaction: bool = False)

Add a BeliefsDataFrame as timed beliefs in the database.

Parameters

- **bdf** – the BeliefsDataFrame to be persisted
- **expunge_session** – if True, all non-flushed instances are removed from the session before adding beliefs. Expunging can resolve problems you might encounter with states of objects in your session. When using this option, you might want to flush newly-created objects which are not beliefs (e.g. a sensor or data source object).
- **allow_overwrite** – if True, new objects are merged if False, objects are added to the session or bulk saved
- **bulk_save_objects** – if True, objects are bulk saved with `session.bulk_save_objects()`, which is quite fast but has several caveats, see: https://docs.sqlalchemy.org/orm/persistence_techniques.html#bulk-operations-caveats if False, objects are added to the session with `session.add_all()`
- **commit_transaction** – if True, the session is committed if False, you can still add other data to the session and commit it all within an atomic transaction

belief_horizon: `timedelta`

cumulative_probability: `float`

event_start: `datetime`

event_value: `float`

classmethod search(*sensors: Sensor | int | str | list[Sensor | int | str]*, *sensor: Sensor = None*, *event_starts_after: datetime_type | None = None*, *event_ends_before: datetime_type | None = None*, *beliefs_after: datetime_type | None = None*, *beliefs_before: datetime_type | None = None*, *horizons_at_least: timedelta | None = None*, *horizons_at_most: timedelta | None = None*, *source: DataSource | list[DataSource] | int | list[int] | str | list[str] | None = None*, *user_source_ids: int | list[int] | None = None*, *source_types: list[str] | None = None*, *exclude_source_types: list[str] | None = None*, *most_recent_beliefs_only: bool = True*, *most_recent_events_only: bool = False*, *most_recent_only: bool = None*, *one_deterministic_belief_per_event: bool = False*, *one_deterministic_belief_per_event_per_source: bool = False*, *resolution: str | timedelta = None*, *sum_multiple: bool = True*) → `tb.BeliefsDataFrame | dict[str, tb.BeliefsDataFrame]`

Search all beliefs about events for the given sensors.

If you don't set any filters, you get the most recent beliefs about all events.

Parameters

- **sensors** – search only these sensors, identified by their instance or id (both unique) or name (non-unique)
- **event_starts_after** – only return beliefs about events that start after this datetime (inclusive)
- **event_ends_before** – only return beliefs about events that end before this datetime (inclusive)
- **beliefs_after** – only return beliefs formed after this datetime (inclusive)
- **beliefs_before** – only return beliefs formed before this datetime (inclusive)
- **horizons_at_least** – only return beliefs with a belief horizon equal or greater than this timedelta (for example, use `timedelta(0)` to get ante knowledge time beliefs)
- **horizons_at_most** – only return beliefs with a belief horizon equal or less than this timedelta (for example, use `timedelta(0)` to get post knowledge time beliefs)

- **source** – search only beliefs by this source (pass the DataSource, or its name or id) or list of sources
 - **user_source_ids** – Optional list of user source ids to query only specific user sources
 - **source_types** – Optional list of source type names to query only specific source types *
 - **exclude_source_types** – Optional list of source type names to exclude specific source types *
 - **most_recent_beliefs_only** – only return the most recent beliefs for each event from each source (minimum belief horizon)
 - **most_recent_events_only** – only return (post knowledge time) beliefs for the most recent event (maximum event start)
 - **one_deterministic_belief_per_event** – only return a single value per event (no probabilistic distribution and only 1 source)
 - **one_deterministic_belief_per_event_per_source** – only return a single value per event per source (no probabilistic distribution)
 - **resolution** – Optional timedelta or pandas freqstr used to resample the results **
 - **sum_multiple** – if True, sum over multiple sensors; otherwise, return a dictionary with sensor names as key, each holding a BeliefsDataFrame as its value
- If user_source_ids is specified, the “user” source type is automatically included (and not excluded). Somewhat redundant, though still allowed, is to set both source_types and exclude_source_types.

**** Note that:**

- timely-beliefs converts string resolutions to datetime.timedelta objects (see <https://github.com/SeitaBV/timely-beliefs/issues/13>).
- for sensors recording non-instantaneous data: updates both the event frequency and the event resolution
- for sensors recording instantaneous data: updates only the event frequency (and event resolution remains 0)

sensor_id

source_id

class flexmeasures.data.models.time_series.TimedValue

A mixin of all tables that store time series data, either forecasts or measurements. Represents one row.

Note: This will be deprecated in favour of Timely-Beliefs - based code (see Sensor/TimedBelief)

classmethod **make_query**(old_sensor_names: tuple[str], query_window: tuple[datetime_type | None, datetime_type | None], belief_horizon_window: tuple[timedelta | None, timedelta | None] = (None, None), belief_time_window: tuple[datetime_type | None, datetime_type | None] = (None, None), belief_time: datetime_type | None = None, user_source_ids: int | list[int] | None = None, source_types: list[str] | None = None, exclude_source_types: list[str] | None = None, session: Session = None) → Query

Can be extended with the make_query function in subclasses. We identify the assets by their name, which assumes a unique string field can be used. The query window consists of two optional datetimes (start and end). The horizon window expects first the shorter horizon (e.g. 6H) and then the longer horizon (e.g. 24H). The session can be supplied, but if None, the implementation should find a session itself.

Parameters

- **user_source_ids** – Optional list of user source ids to query only specific user sources
 - **source_types** – Optional list of source type names to query only specific source types *
 - **exclude_source_types** – Optional list of source type names to exclude specific source types *
- If user_source_ids is specified, the “user” source type is automatically included (and not excluded). Somewhat redundant, though still allowed, is to set both source_types and exclude_source_types.

todo: add examples # todo: switch to using timely_beliefs queries, which are more powerful

```
classmethod search(old_sensor_names: str | list[str], event_starts_after: datetime_type | None = None,
                    event_ends_before: datetime_type | None = None, horizons_at_least: timedelta | None
                    = None, horizons_at_most: timedelta | None = None, beliefs_after: datetime_type |
                    None = None, beliefs_before: datetime_type | None = None, user_source_ids: int |
                    list[int] | None = None, source_types: list[str] | None = None, exclude_source_types:
                    list[str] | None = None, resolution: str | timedelta = None, sum_multiple: bool =
                    True) → tb.BeliefsDataFrame | dict[str, tb.BeliefsDataFrame]
```

Basically a convenience wrapper for services.collect_time_series_data, where time series data collection is implemented.

flexmeasures.data.models.user

Functions

`flexmeasures.data.models.user.is_user(o) → bool`

True if object is or proxies a User, False otherwise.

Takes into account that object can be of LocalProxy type, and uses get_current_object to get the underlying (User) object.

`flexmeasures.data.models.user.remember_last_seen(user)`

Update the last_seen field

`flexmeasures.data.models.user.remember_login(the_app, user)`

We do not use the tracking feature of flask_security, but this basic meta data are quite handy to know

Classes

`class flexmeasures.data.models.user.Account(**kwargs)`

Account of a tenant on the server. Bundles Users as well as GenericAssets.

`__init__(**kwargs)`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

has_role(*role*: *str* | *AccountRole*) → *bool*

Returns *True* if the account has the specified role.

Parameters

role – An account role name or *AccountRole* instance

search_annotations(*annotation_starts_after*: *datetime* | *None* = *None*, *annotations_after*: *datetime* | *None* = *None*, *annotation_ends_before*: *datetime* | *None* = *None*, *annotations_before*: *datetime* | *None* = *None*, *source*: *DataSource* | *List*[*DataSource*] | *int* | *List*[*int*] | *str* | *List*[*str*] | *None* = *None*, *as_frame*: *bool* = *False*) → *List*[*Annotation*] | *pd.DataFrame*

Return annotations assigned to this account.

Parameters

- **annotations_after** – only return annotations that end after this datetime (exclusive)
- **annotations_before** – only return annotations that start before this datetime (exclusive)

class flexmeasures.data.models.user.**AccountRole**(***kwargs*)

__init__(***kwargs*)

A simple constructor that allows initialization from *kwargs*.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class flexmeasures.data.models.user.**Role**(***kwargs*)

__init__(***kwargs*)

A simple constructor that allows initialization from *kwargs*.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class flexmeasures.data.models.user.**RolesAccounts**(***kwargs*)

__init__(***kwargs*)

A simple constructor that allows initialization from *kwargs*.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class flexmeasures.data.models.user.**RolesUsers**(***kwargs*)

__init__(***kwargs*)

A simple constructor that allows initialization from *kwargs*.

Sets attributes on the constructed instance using the names and values in *kwargs*.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class flexmeasures.data.models.user.**User**(***kwargs*)

We use the flask security *UserMixin*, which does include functionality, but not the fields (those are in *flask_security/models::FsUserMixin*). We went with a pick&choose approach. This gives us more freedom, e.g. to choose our own table name or add logic around the activation status. If we add new FS functionality (e.g. 2FA), the fields needed for that need to be added here.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

has_role(role: str | Role) → bool

Returns *True* if the user identifies with the specified role.

Overwritten from flask_security.core.UserMixin.

Parameters

role – A role name or *Role* instance

property is_authenticated: bool

We are overloading this, so it also considers being active. Inactive users can by definition not be authenticated.

roles

The roles attribute is being used by Flask-Security in the @roles_required decorator (among others). With this little overload fix, it will only return the user's roles if they are authenticated. We do this to prevent that if a user is logged in while the admin deactivates them, their session would still work. In effect, we strip unauthenticated users from their roles. To read roles of an unauthenticated user (e.g. being inactive), use the *flexmeasures_roles* attribute. If our auth model has moved to an improved way, e.g. requiring modern tokens, we should consider relaxing this. Note: This needed to become a hybrid property when moving to Flask-Security 3.4

flexmeasures.data.models.validation_utils

Functions

flexmeasures.data.models.validation_utils.**check_required_attributes**(sensor: Sensor, attributes: List[str | Tuple[str, Type | Tuple[Type, ...]]])

Raises if any attribute in the list of attributes is missing on the Sensor, or has the wrong type.

Parameters

- **sensor** – Sensor object to check for attributes
- **attributes** – List of either an attribute name or a tuple of an attribute name and its allowed type (the allowed type may also be a tuple of several allowed types)

Exceptions

exception flexmeasures.data.models.validation_utils.**MissingAttributeException**

exception flexmeasures.data.models.validation_utils.**WrongTypeAttributeException**

flexmeasures.data.models.weather

Classes

class flexmeasures.data.models.weather.**Weather**(*use_legacy_kwargs=True, **kwargs*)

All weather measurements are stored in one slim table.

This model is now considered legacy. See TimedBelief.

__init__(*use_legacy_kwargs=True, **kwargs*)

data_source_id

datetime

horizon

classmethod **make_query**(***kwargs*) → Query

Construct the database query.

value

class flexmeasures.data.models.weather.**WeatherSensor**(***kwargs*)

A weather sensor has a location on Earth and measures weather values of a certain weather sensor type, such as temperature, wind speed and irradiance.

This model is now considered legacy. See GenericAsset and Sensor.

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

property **entity_address**: **str**

Entity address under the latest fm scheme for entity addresses.

property **entity_address_fm0**: **str**

Entity address under the fm0 scheme for entity addresses.

event_resolution: **timedelta**

get_attribute(*attribute: str*)

Looks for the attribute on the corresponding Sensor.

This should be used by all code to read these attributes, over accessing them directly on this class, as this table is in the process to be replaced by the Sensor table.

great_circle_distance(***kwargs*)

Query great circle distance (unclear if in km or in miles).

Can be called with an object that has latitude and longitude properties, for example:

```
great_circle_distance(object=asset)
```

Can also be called with latitude and longitude parameters, for example:

```
great_circle_distance(latitude=32, longitude=54) great_circle_distance(lat=32, lng=54)
```

```

id
knowledge_horizon_fnc: str
knowledge_horizon_par: dict
name: str
timezone: str
unit: str
property weather_unit: float

```

Return the ‘unit’ property of the generic asset, just with a more insightful name.

class flexmeasures.data.models.weather.WeatherSensorType(**kwargs)

This model is now considered legacy. See GenericAssetType.

__init__(**kwargs)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

Data models for FlexMeasures

Exceptions

exception flexmeasures.data.models.ModelException

flexmeasures.data.queries

Modules

flexmeasures.data.queries.analytics

flexmeasures.data.queries.annotations

flexmeasures.data.queries.data_sources

flexmeasures.data.queries.generic_assets

flexmeasures.data.queries.portfolio

flexmeasures.data.queries.sensors

flexmeasures.data.queries.utils

flexmeasures.data.queries.analytics

Functions

```
flexmeasures.data.queries.analytics.get_power_data(resource: str | Resource,  
                                                    show_consumption_as_positive: bool,  
                                                    showing_individual_traces_for: str, metrics: dict,  
                                                    query_window: Tuple[datetime, datetime],  
                                                    resolution: str, forecast_horizon: timedelta) →  
                                                    Tuple[DataFrame, DataFrame, DataFrame, dict]
```

Get power data and metrics.

Return power observations, power forecasts and power schedules (each might be an empty DataFrame) and a dict with the following metrics: - expected value - mean absolute error - mean absolute percentage error - weighted absolute percentage error

Todo: Power schedules ignore horizon.

```
flexmeasures.data.queries.analytics.get_prices_data(metrics: dict, market_sensor: Sensor,  
                                                    query_window: Tuple[datetime, datetime],  
                                                    resolution: str, forecast_horizon: timedelta) →  
                                                    Tuple[DataFrame, DataFrame, dict]
```

Get price data and metrics.

Return price observations, price forecasts (either might be an empty DataFrame) and a dict with the following metrics: - expected value - mean absolute error - mean absolute percentage error - weighted absolute percentage error

```
flexmeasures.data.queries.analytics.get_revenues_costs_data(power_data: DataFrame,  
                                                            prices_data: DataFrame,  
                                                            power_forecast_data: DataFrame,  
                                                            prices_forecast_data: DataFrame,  
                                                            metrics: Dict[str, float], unit_factor:  
                                                            float, resolution: str,  
                                                            showing_individual_traces: bool) →  
                                                            Tuple[DataFrame, DataFrame, dict]
```

Compute revenues/costs data. These data are purely derivative from power and prices. For forecasts we use the WAPE metrics. Then we calculate metrics on this construct. The unit factor is used when multiplying quantities and prices, e.g. when multiplying quantities in kWh with prices in EUR/MWh, use a unit factor of 0.001.

Return revenue/cost observations, revenue/cost forecasts (either might be an empty DataFrame) and a dict with the following metrics: - expected value - mean absolute error - mean absolute percentage error - weighted absolute percentage error

```
flexmeasures.data.queries.analytics.get_weather_data(assets: List[Asset], metrics: dict, sensor_type:  
WeatherSensorType, query_window:  
Tuple[datetime, datetime], resolution: str,  
forecast_horizon: timedelta) →  
Tuple[DataFrame, DataFrame, str, Sensor, dict]
```

Get most recent weather data and forecast weather data for the requested forecast horizon.

Return weather observations, weather forecasts (either might be an empty DataFrame), the name of the sensor type, the weather sensor and a dict with the following metrics: - expected value - mean absolute error - mean absolute percentage error - weighted absolute percentage error

flexmeasures.data.queries.annotations

Functions

`flexmeasures.data.queries.annotations.query_asset_annotations(asset_id: int, annotations_after: datetime | None = None, annotations_before: datetime | None = None, sources: list[DataSource] | None = None, annotation_type: str | None = None)` → *Query*

Match annotations assigned to the given asset.

flexmeasures.data.queries.data_sources

Functions

`flexmeasures.data.queries.data_sources.get_or_create_source(source: User | str, source_type: str | None = None, model: str | None = None, flush: bool = True)` → *DataSource*

`flexmeasures.data.queries.data_sources.get_source_or_none(source: int | str, source_type: str | None = None)` → *DataSource* | *None*

Parameters

- **source** – source id
- **source_type** – optionally, filter by source type

flexmeasures.data.queries.generic_assets

Functions

`flexmeasures.data.queries.generic_assets.get_asset_group_queries(group_by_type: bool = True, group_by_account: bool = False, group_by_location: bool = False, custom_aggregate_type_groups: Dict[str, List[str]] | None = None)` → *Dict*[*str*, *Query*]

An asset group is defined by Asset queries, which this function can generate. Each query has a name (for the asset group it represents). These queries still need an executive call, like `all()`, `count()` or `first()`. This function limits the assets to be queried to the current user’s account, if the user is not an admin. Note: Make sure the current user has the “read” permission on their account (on `GenericAsset.__class__`? See <https://github.com/FlexMeasures/flexmeasures/issues/200>) or is an admin. :param group_by_type: If `True`, groups will be made for assets with the same type. We prefer pluralised group names here. Defaults to `True`. :param group_by_account: If `True`, groups will be made for assets within the same account. This makes sense for admins, as they can query across accounts. :param group_by_location: If `True`, groups will be made for assets at the same location. Naming of the location currently supports charge points (for EVSEs). :param custom_aggregate_type_groups: dict of asset type groupings (mapping group names to names of asset types). See also the setting `FLEXMEASURES_ASSET_TYPE_GROUPS`.

```
flexmeasures.data.queries.generic_assets.get_location_queries(account_id: int | None = None) → Dict[str, Query]
```

Make queries for grouping assets by location.

We group EVSE assets by location (if they share a location, they belong to the same Charge Point) Like `get_asset_group_queries`, the values in the returned dict still need an executive call, like `all()`, `count()` or `first()`. Note that this function will still load and inspect assets to do its job.

The Charge Points are named on the basis of the first EVSE in their list, using either the whole EVSE name or that part that comes before a " - " delimiter. For example: If:

```
evse_name = "Seoul Hilton - charger 1"
```

Then:

```
charge_point_name = "Seoul Hilton (Charge Point)"
```

A Charge Point is a special case. If all assets on a location are of type EVSE, we can call the location a "Charge Point".

Parameters

account_id – Pass in an account ID if you want to query an account other than your own. This only works for admins. Public assets are always queried.

```
flexmeasures.data.queries.generic_assets.group_assets_by_location(asset_list: List[GenericAsset]) → List[List[GenericAsset]]
```

```
flexmeasures.data.queries.generic_assets.query_assets_by_type(type_names: List[str] | str, account_id: int | None = None, query: Query | None = None) → Query
```

Return a query which looks for GenericAssets by their type.

Parameters

- **type_names** – Pass in a list of type names or only one type name.
- **account_id** – Pass in an account ID if you want to query an account other than your own. This only works for admins. Public assets are always queried.
- **query** – Pass in an existing Query object if you have one.

flexmeasures.data.queries.portfolio

Functions

```
flexmeasures.data.queries.portfolio.get_power_data(resource_dict: Dict[str, Resource]) → Tuple[Dict[str, DataFrame], Dict[str, DataFrame], Dict[str, float], Dict[str, float], Dict[str, float], Dict[str, float]]
```

Get power data, separating demand and supply, as time series per resource and as totals (summed over time) per resource and per asset.

Getting sensor data of a Resource leads to database queries (unless results are already cached).

Returns

a tuple comprising: - a dictionary of resource names (as keys) and a DataFrame with aggregated time series of supply (as values) - a dictionary of resource names (as keys) and a DataFrame

with aggregated time series of demand (as values) - a dictionary of resource names (as keys) and their total supply summed over time (as values) - a dictionary of resource names (as keys) and their total demand summed over time (as values) - a dictionary of asset names (as keys) and their total supply summed over time (as values) - a dictionary of asset names (as keys) and their total demand summed over time (as values)

```
flexmeasures.data.queries.portfolio.get_price_data(resource_dict: Dict[str, Resource]) →
                                                    Tuple[Dict[str, BeliefsDataFrame], Dict[str, float]]
```

```
flexmeasures.data.queries.portfolio.get_structure(assets: List[Asset]) → Tuple[Dict[str, AssetType],
                                                    List[Market], Dict[str, Resource]]
```

Get asset portfolio structured as Resources, based on AssetTypes present in a list of Assets.

Initializing Resources leads to some database queries.

Parameters

assets – a list of Assets

Returns

a tuple comprising: - a dictionary of resource names (as keys) and the asset type represented by these resources (as values) - a list of (unique) Markets that are relevant to these resources - a dictionary of resource names (as keys) and Resources (as values)

flexmeasures.data.queries.sensors

Functions

```
flexmeasures.data.queries.sensors.query_sensor_by_name_and_generic_asset_type_name(sensor_name:
                                                                                      str |
                                                                                      None =
                                                                                      None,
                                                                                      generic_asset_type_names:
                                                                                      List[str]
                                                                                      | None
                                                                                      = None,
                                                                                      ac-
                                                                                      count_id:
                                                                                      int |
                                                                                      None =
                                                                                      None)
→
Query
```

Match a sensor by its own name and that of its generic asset type.

Parameters

- **sensor_name** – should match (if None, no match is needed)
- **generic_asset_type_names** – should match at least one of these (if None, no match is needed)
- **account_id** – Pass in an account ID if you want to query an account other than your own. This only works for admins. Public assets are always queried.

```
flexmeasures.data.queries.sensors.query_sensors_by_proximity(latitude: float, longitude: float,  
                                                             generic_asset_type_name: str |  
                                                             None, sensor_name: str | None,  
                                                             account_id: int | None = None) →  
                                                             Query
```

Order them by proximity of their asset's location to the target.

flexmeasures.data.queries.utils

Functions

```
flexmeasures.data.queries.utils.create_beliefs_query(cls: Type[TimedValue], session: Session,  
                                                    old_sensor_class: Model, old_sensor_names:  
                                                    Tuple[str], start: datetime | None, end:  
                                                    datetime | None) → Query
```

```
flexmeasures.data.queries.utils.get_belief_timing_criteria(cls: Type[TimedValue], asset_class:  
                                                         Model, belief_horizon_window:  
                                                         Tuple[timedelta | None, timedelta |  
                                                         None], belief_time_window:  
                                                         Tuple[datetime | None, datetime |  
                                                         None]) → List[BinaryExpression]
```

Get filter criteria for the desired windows with relevant belief times and belief horizons.

todo: interpret belief horizons with respect to knowledge time rather than event end. - a positive horizon denotes a before-the-fact belief (ex-ante w.r.t. knowledge time) - a negative horizon denotes an after-the-fact belief (ex-post w.r.t. knowledge time)

Parameters

- **belief_horizon_window** – short belief horizon and long belief horizon, each an optional timedelta Interpretation: - a positive short horizon denotes “at least <horizon> before the fact” (min ex-ante) - a positive long horizon denotes “at most <horizon> before the fact” (max ex-ante) - a negative short horizon denotes “at most <horizon> after the fact” (max ex-post) - a negative long horizon denotes “at least <horizon> after the fact” (min ex-post)
- **belief_time_window** – earliest belief time and latest belief time, each an optional datetime

Examples (assuming the knowledge time of each event coincides with the end of the event):

```
# Query beliefs formed between 1 and 7 days before each individual event belief_horizon_window =  
(timedelta(days=1), timedelta(days=7))  
  
# Query beliefs formed at least 2 hours before each individual event belief_horizon_window =  
(timedelta(hours=2), None)  
  
# Query beliefs formed at most 2 hours after each individual event belief_horizon_window = (-  
timedelta(hours=2), None)  
  
# Query beliefs formed at least after each individual event belief_horizon_window = (None,  
timedelta(hours=0))  
  
# Query beliefs formed from May 2nd to May 13th (left inclusive, right exclusive) be-  
lief_time_window = (datetime(2020, 5, 2), datetime(2020, 5, 13))  
  
# Query beliefs formed from May 14th onwards belief_time_window = (datetime(2020, 5, 14), None)  
  
# Query beliefs formed before May 13th belief_time_window = (None, datetime(2020, 5, 13))
```

```
flexmeasures.data.queries.utils.get_source_criteria(cls: Type[TimedValue] | Type[TimedBelief],
                                                    user_source_ids: int | List[int], source_types:
                                                    List[str], exclude_source_types: List[str]) →
                                                    List[BinaryExpression]
```

```
flexmeasures.data.queries.utils.multiply_dataframe_with_deterministic_beliefs(df1:
                                                                               DataFrame,
                                                                               df2:
                                                                               DataFrame,
                                                                               multiplication_factor:
                                                                               float = 1,
                                                                               result_source:
                                                                               str | None =
                                                                               None) →
                                                                               DataFrame
```

Create new DataFrame where the event_value columns of df1 and df2 are multiplied.

If df1 and df2 have belief_horizon columns, the belief_horizon column of the new DataFrame is determined as the minimum of the input horizons. The source columns of df1 and df2 are not used. A source column for the new DataFrame can be set by passing a result_source (string).

The index of the resulting DataFrame contains the outer join of the indices of df1 and df2. Event values are np.nan for rows that are not in both DataFrames.

Parameters

- **df1** – DataFrame with “event_value” column and optional “belief_horizon” and “source” columns
- **df2** – DataFrame with “event_value” column and optional “belief_horizon” and “source” columns
- **multiplication_factor** – extra scalar to determine the event_value of the resulting DataFrame
- **result_source** – string label for the source of the resulting DataFrame

Returns

DataFrame with “event_value” column, an additional “belief_horizon” column if both df1 and df2 contain this column, and an additional “source” column if result_source is set.

```
flexmeasures.data.queries.utils.potentially_limit_assets_query_to_account(query: Query,
                                                                           account_id: int |
                                                                           None = None) →
                                                                           Query
```

Filter out all assets that are not in the current user’s account. For admins and CLI users, no assets are filtered out, unless an account_id is set.

Parameters

account_id – if set, all assets that are not in the given account will be filtered out (only works for admins and CLI users). For querying public assets in particular, don’t use this function.

```
flexmeasures.data.queries.utils.simplify_index(bdf: BeliefsDataFrame, index_levels_to_columns:
                                                List[str] | None = None) → DataFrame
```

Drops indices other than event_start. Optionally, salvage index levels as new columns.

Because information stored in the index levels is potentially lost*, we cannot guarantee a complete description of beliefs in the BeliefsDataFrame. Therefore, we type the result as a regular pandas DataFrame.

- The index levels are dropped (by overwriting the multi-level index with just the “event_start” index level). Only for the columns named in `index_levels_to_columns`, the relevant information is kept around.

`flexmeasures.data.queries.utils.source_type_criterion(source_types: List[str]) → BinaryExpression`

Criterion to collect only data from sources that are of the given type.

`flexmeasures.data.queries.utils.source_type_exclusion_criterion(source_types: List[str]) → BinaryExpression`

Criterion to exclude sources that are of the given type.

`flexmeasures.data.queries.utils.user_source_criterion(cls: Type[TimedValue] | Type[TimedBelief],
user_source_ids: int | List[int]) → BinaryExpression`

Criterion to search only through user data from the specified user sources.

We distinguish user sources (sources with `source.type == “user”`) from other sources (`source.type != “user”`). Data with a user source originates from a registered user. Data with e.g. a script source originates from a script.

This criterion doesn’t affect the query over non-user type sources. It does so by ignoring user sources that are not in the given list of `source_ids`.

Data query functions

flexmeasures.data.schemas

Modules

flexmeasures.data.schemas.account

flexmeasures.data.schemas.assets

flexmeasures.data.schemas.generic_assets

flexmeasures.data.schemas.reporting

flexmeasures.data.schemas.scheduling

flexmeasures.data.schemas.sensors

flexmeasures.data.schemas.sources

flexmeasures.data.schemas.times

flexmeasures.data.schemas.units

flexmeasures.data.schemas.users

flexmeasures.data.schemas.utils

flexmeasures.data.schemas.account

Classes

class flexmeasures.data.schemas.account.**AccountIdField**(*, *strict*: *bool* = *False*, **kwargs)

Field that deserializes to an Account and serializes back to an integer.

_deserialize(value, attr, obj, **kwargs) → *Account*

Turn an account id into an Account.

_serialize(account, attr, data, **kwargs)

Turn an Account into a source id.

class flexmeasures.data.schemas.account.**AccountRoleSchema**(*args, **kwargs)

AccountRole schema, with validations.

class **Meta**

model

alias of *AccountRole*

opts: **SchemaOpts** = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>

class flexmeasures.data.schemas.account.**AccountSchema**(*args, **kwargs)

Account schema, with validations.

class **Meta**

model

alias of *Account*

opts: **SchemaOpts** = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>

flexmeasures.data.schemas.assets

Classes

class flexmeasures.data.schemas.assets.**AssetSchema**(*args, **kwargs)

Asset schema, with validations.

TODO: deprecate, as it is based on legacy data model. Move some attributes to SensorSchema.

class **Meta**

model

alias of *Asset*

opts: **SchemaOpts** = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>

class flexmeasures.data.schemas.assets.**LatitudeField**(*args, **kwargs)

Field that deserializes to a latitude float with max 7 decimal places.

__init__(*args, **kwargs)

class flexmeasures.data.schemas.assets.**LatitudeLongitudeValidator**(*, *error*: *str* | *None* = *None*)

Validator which succeeds if the value passed has at most 7 decimal places.

```
__init__(*, error: str | None = None)
```

```
class flexmeasures.data.schemas.assets.LatitudeValidator(*, error: str | None = None, allow_none: bool = False)
```

Validator which succeeds if the value passed is in the range [-90, 90].

```
__init__(*, error: str | None = None, allow_none: bool = False)
```

```
class flexmeasures.data.schemas.assets.LongitudeField(*args, **kwargs)
```

Field that deserializes to a longitude float with max 7 decimal places.

```
__init__(*args, **kwargs)
```

```
class flexmeasures.data.schemas.assets.LongitudeValidator(*, error: str | None = None, allow_none: bool = False)
```

Validator which succeeds if the value passed is in the range [-180, 180].

```
__init__(*, error: str | None = None, allow_none: bool = False)
```

flexmeasures.data.schemas.generic_assets

Classes

```
class flexmeasures.data.schemas.generic_assets.GenericAssetIdField(*args, **kwargs)
```

Field that deserializes to a GenericAsset and serializes back to an integer.

```
_deserialize(value, attr, obj, **kwargs) → GenericAsset
```

Turn a generic asset id into a GenericAsset.

```
_serialize(asset, attr, data, **kwargs)
```

Turn a GenericAsset into a generic asset id.

```
class flexmeasures.data.schemas.generic_assets.GenericAssetSchema(*args, **kwargs)
```

GenericAsset schema, with validations.

```
class Meta
```

```
    model
```

alias of [GenericAsset](#)

```
opts: SchemaOpts = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>
```

```
class flexmeasures.data.schemas.generic_assets.GenericAssetTypeSchema(*args, **kwargs)
```

GenericAssetType schema, with validations.

```
class Meta
```

```
    model
```

alias of [GenericAssetType](#)

```
opts: SchemaOpts = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>
```

```
class flexmeasures.data.schemas.generic_assets.JSON(*, load_default: typing.Any =
    <marshmallow.missing>, missing: typing.Any =
    <marshmallow.missing>, dump_default:
    typing.Any = <marshmallow.missing>, default:
    typing.Any = <marshmallow.missing>,
    data_key: str | None = None, attribute: str |
    None = None, validate: None |
    (typing.Callable[[typing.Any], typing.Any] |
    typing.Iterable[typing.Callable[[typing.Any],
    typing.Any]]) = None, required: bool = False,
    allow_none: bool | None = None, load_only:
    bool = False, dump_only: bool = False,
    error_messages: dict[str, str] | None = None,
    metadata: typing.Mapping[str, typing.Any] |
    None = None, **additional_metadata)
```

_deserialize(value, attr, data, **kwargs) → dict

Deserialize value. Concrete Field classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** – The attribute/key in *data* to be deserialized.
- **data** – The raw input data passed to the *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – In case of formatting or validation failure.

Returns

The deserialized value.

Changed in version 2.0.0: Added **attr** and **data** parameters.

Changed in version 3.0.0: Added ****kwargs** to signature.

_serialize(value, attr, data, **kwargs) → str

Serializes **value** to a basic Python datatype. Noop by default. Concrete Field classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj, **kwargs):
        if not value:
            return ''
        return str(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.
- **kwargs** (*dict*) – Field-specific keyword arguments.

Returns

The serialized value

flexmeasures.data.schemas.reporting**Modules**

<code>flexmeasures.data.schemas.reporting.</code>
<code>aggregation</code>
<code>flexmeasures.data.schemas.reporting.</code>
<code>pandas_reporter</code>

flexmeasures.data.schemas.reporting.aggregation**Classes**

```
class flexmeasures.data.schemas.reporting.aggregation.AggregatorSchema(*, only:
    types.StrSequenceOrSet
    | None = None, exclude:
    types.StrSequenceOrSet
    = (), many: bool =
    False, context: dict |
    None = None,
    load_only:
    types.StrSequenceOrSet
    = (), dump_only:
    types.StrSequenceOrSet
    = (), partial: bool |
    types.StrSequenceOrSet
    = False, unknown: str |
    None = None)
```

Schema for the reporter_config of the AggregatorReporter

Example: .. code-block:: json

```
{
  "beliefs_search_configs": [
    {
      "sensor": 1, "source": 1, "alias": "pv"
    }, {
      "sensor": 1, "source": 2, "alias": "consumption"
    }
  ], "method": "sum", "weights": {
    "pv": 1.0, "consumption": -1.0
  }
}
```


flexmeasures.data.schemas.reporting.pandas_reporter

Classes

```
class flexmeasures.data.schemas.reporting.pandas_reporter.PandasMethodCall(*, only:
    types.StrSequenceOrSet
    | None = None,
    exclude:
    types.StrSequenceOrSet
    = (), many: bool =
    False, context:
    dict | None =
    None, load_only:
    types.StrSequenceOrSet
    = (), dump_only:
    types.StrSequenceOrSet
    = (), partial: bool |
    types.StrSequenceOrSet
    = False, unknown:
    str | None = None)
```

```
class flexmeasures.data.schemas.reporting.pandas_reporter.PandasReporterConfigSchema(*,
                                                                                       only:
                                                                                       types.StrSequenceOrSet
                                                                                       |
                                                                                       None
                                                                                       =
                                                                                       None,
                                                                                       ex-
                                                                                       clude:
                                                                                       types.StrSequenceOrSet
                                                                                       = (),
                                                                                       many:
                                                                                       bool
                                                                                       =
                                                                                       False,
                                                                                       con-
                                                                                       text:
                                                                                       dict |
                                                                                       None
                                                                                       =
                                                                                       None,
                                                                                       load_only:
                                                                                       types.StrSequenceOrSet
                                                                                       = (),
                                                                                       dump_only:
                                                                                       types.StrSequenceOrSet
                                                                                       = (),
                                                                                       par-
                                                                                       tial:
                                                                                       bool |
                                                                                       types.StrSequenceOrSet
                                                                                       =
                                                                                       False,
                                                                                       un-
                                                                                       known:
                                                                                       str |
                                                                                       None
                                                                                       =
                                                                                       None)
```

This schema lists fields that can be used to describe sensors in the optimised portfolio

Example:

```
{
  "input_sensors":
    [[{"sensor": 1, "alias": "df1"}]
  ], "transformations": [
    {
      "df_input": "df1", "df_output": "df2", "method": "copy"
    }, {
      "df_input": "df2", "df_output": "df2", "method": "sum"
    }, {
```

```

        "method": "sum", "kwargs": {"axis": 0}
    }
], "final_df_output": "df2"

```

validate_chaining(data, **kwargs)

This validator ensures that we are always given an input and that the final_df_output is computed.

Classes

```

class flexmeasures.data.schemas.reporting.BeliefsSearchConfigSchema(*, only:
    types.StrSequenceOrSet |
    None = None, exclude:
    types.StrSequenceOrSet =
    (), many: bool = False,
    context: dict | None =
    None, load_only:
    types.StrSequenceOrSet =
    (), dump_only:
    types.StrSequenceOrSet =
    (), partial: bool |
    types.StrSequenceOrSet =
    False, unknown: str | None
    = None)

```

This schema implements the required fields to perform a TimedBeliefs search using the method flexmeasures.data.models.time_series:Sensor.search_beliefs

```

class flexmeasures.data.schemas.reporting.ReporterConfigSchema(*, only: types.StrSequenceOrSet |
    None = None, exclude:
    types.StrSequenceOrSet = (),
    many: bool = False, context: dict |
    None = None, load_only:
    types.StrSequenceOrSet = (),
    dump_only:
    types.StrSequenceOrSet = (),
    partial: bool |
    types.StrSequenceOrSet = False,
    unknown: str | None = None)

```

This schema is used to validate Reporter class configurations (reporter_config). Inherit from this to extend this schema with your own parameters.

flexmeasures.data.schemas.scheduling

Modules

```

flexmeasures.data.schemas.scheduling.
storage

```

flexmeasures.data.schemas.scheduling.storage

Classes

class flexmeasures.data.schemas.scheduling.storage.**EfficiencyField**(*args, **kwargs)

Field that deserializes to a Quantity with % units. Must be greater than 0% and less than or equal to 100%.

Examples:

```
>>> ef = EfficiencyField()
>>> ef.deserialize(0.9)
<Quantity(90.0, 'percent')>
>>> ef.deserialize("90%")
<Quantity(90.0, 'percent')>
>>> ef.deserialize("0%")
Traceback (most recent call last):
...
marshmallow.exceptions.ValidationError: ['Must be greater than 0 and less than or_
↳ equal to 1.']
```

__init__(*args, **kwargs)

class flexmeasures.data.schemas.scheduling.storage.**SOCValueSchema**(*args, **kwargs)

A point in time with a target value.

__init__(*args, **kwargs)

class flexmeasures.data.schemas.scheduling.storage.**StorageFlexModelSchema**(start: *datetime*,
sensor: *Sensor*,
*args, **kwargs)

This schema lists fields we require when scheduling storage assets. Some fields are not required, as they might live on the Sensor.attributes. You can use StorageScheduler.deserialize_flex_config to get that filled in.

__init__(start: *datetime*, sensor: *Sensor*, *args, **kwargs)

Pass the schedule's start, so we can use it to validate soc-target datetimes.

post_load_sequence(data: *dict*, **kwargs) → *dict*

Perform some checks and corrections after we loaded.

Classes

class flexmeasures.data.schemas.scheduling.**FlexContextSchema**(**only*: *types.StrSequenceOrSet* |
None = *None*, *exclude*:
types.StrSequenceOrSet = (), *many*:
bool = *False*, *context*: *dict* | *None* =
None, *load_only*:
types.StrSequenceOrSet = (),
dump_only: *types.StrSequenceOrSet*
= (), *partial*: *bool* |
types.StrSequenceOrSet = *False*,
unknown: *str* | *None* = *None*)

This schema lists fields that can be used to describe sensors in the optimised portfolio

flexmeasures.data.schemas.sensors

Classes

```
class flexmeasures.data.schemas.sensors.SensorIdField(*args, **kwargs)
```

Field that deserializes to a Sensor and serializes back to an integer.

```
_deserialize(value: int, attr, obj, **kwargs) → Sensor
```

Turn a sensor id into a Sensor.

```
_serialize(sensor: Sensor, attr, data, **kwargs) → int
```

Turn a Sensor into a sensor id.

```
class flexmeasures.data.schemas.sensors.SensorSchema(*args, **kwargs)
```

Sensor schema, with validations.

```
class Meta
```

```
model
```

alias of `Sensor`

```
opts: SchemaOpts = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>
```

```
class flexmeasures.data.schemas.sensors.SensorSchemaMixin(*, only: types.StrSequenceOrSet | None
= None, exclude:
types.StrSequenceOrSet = (), many: bool
= False, context: dict | None = None,
load_only: types.StrSequenceOrSet = (),
dump_only: types.StrSequenceOrSet =
(), partial: bool |
types.StrSequenceOrSet = False,
unknown: str | None = None)
```

Base sensor schema.

Here we include all fields which are implemented by `timely_beliefs.SensorDBMixin`. All classes inheriting from `timely_beliefs.Sensor` don't need to repeat these. In a while, this schema can represent our unified `Sensor` class.

When subclassing, also subclass from `ma.SQLAlchemySchema` and add your own DB model class, e.g.:

```
class Meta:
```

```
model = Asset
```

flexmeasures.data.schemas.sources

Classes

```
class flexmeasures.data.schemas.sources.DataSourceIdField(*, strict: bool = False, **kwargs)
```

Field that deserializes to a DataSource and serializes back to an integer.

```
_deserialize(value, attr, obj, **kwargs) → DataSource
```

Turn a source id into a DataSource.

```
_serialize(source, attr, data, **kwargs)
```

Turn a DataSource into a source id.

flexmeasures.data.schemas.times

Classes

class flexmeasures.data.schemas.times.**AwareDateTimeField**(*args, **kwargs)

Field that de-serializes to a timezone aware datetime and serializes back to a string.

_deserialize(value: *str*, attr, obj, **kwargs) → *datetime*

Work-around until this PR lands: <https://github.com/marshmallow-code/marshmallow/pull/1787>

class flexmeasures.data.schemas.times.**DurationField**(*args, **kwargs)

Field that deserializes to a ISO8601 Duration and serializes back to a string.

_deserialize(value, attr, obj, **kwargs) → *timedelta* | *isodate.Duration*

Use the isodate library to turn an ISO8601 string into a timedelta. For some non-obvious cases, it will become an *isodate.Duration*, see `ground_from` for more. This method throws a *ValidationError* if the string is not ISO norm.

_serialize(value, attr, data, **kwargs)

An implementation of `_serialize`. It is not guaranteed to return the same string as was input, if `ground_from` has been used!

static ground_from(duration: *timedelta* | *isodate.Duration*, start: *datetime* | *None*) → *timedelta*

For some valid duration strings (such as “P1M”, a month), converting to a *datetime.timedelta* is not possible (no obvious number of days). In this case, `_deserialize` returned an *isodate.Duration*. We can derive the *timedelta* by grounding to an actual time span, for which we require a timezone-aware start *datetime*.

class flexmeasures.data.schemas.times.**PlanningDurationField**(*args, **kwargs)

classmethod load_default()

Use this with the `load_default` arg to `__init__` if you want the default FlexMeasures planning horizon.

Exceptions

exception flexmeasures.data.schemas.times.**DurationValidationError**(message: *str* | *list* | *dict*,
field_name: *str* = `'_schema'`,
data: *Mapping*[*str*, *Any*] |
Iterable[*Mapping*[*str*, *Any*]] |
None = *None*, valid_data:
list[*dict*[*str*, *Any*]] | *dict*[*str*,
Any] | *None* = *None*,
**kwargs)

flexmeasures.data.schemas.units

Classes

class flexmeasures.data.schemas.units.**QuantityField**(to_unit: *str*, *args, **kwargs)

Marshmallow/Click field for validating quantities against a unit registry.

The FlexMeasures unit registry is based on the pint library.

For example:

```
>>> percentage_field = QuantityField("%", validate=validate.Range(min=0, max=1))
>>> percentage_field.deserialize("2.5%")
<Quantity(2.5, 'percent')>
>>> percentage_field.deserialize(0.025)
<Quantity(2.5, 'percent')>
>>> power_field = QuantityField("kW", validate=validate.Range(max=ur.Quantity(
↳ "1 kW"))))
>>> power_field.deserialize("120 W")
<Quantity(0.12, 'kilowatt')>
```

```
__init__(to_unit: str, *args, **kwargs)
```

```
_deserialize(value, attr, obj, **kwargs) → Quantity
```

Turn a quantity describing string into a Quantity.

```
_serialize(value, attr, data, **kwargs)
```

Turn a Quantity into a string in scientific format.

```
class flexmeasures.data.schemas.units.QuantityValidator(*, error: str | None = None)
```

Validator which succeeds if the value passed to it is a valid quantity.

```
__init__(*, error: str | None = None)
```

flexmeasures.data.schemas.users

Classes

```
class flexmeasures.data.schemas.users.UserSchema(*args, **kwargs)
```

This schema lists fields we support through this API (e.g. no password).

```
class Meta
```

```
model
```

alias of *User*

```
opts: SchemaOpts = <flask_marshmallow.sqla.SQLAlchemySchemaOpts object>
```

flexmeasures.data.schemas.utils

Functions

```
flexmeasures.data.schemas.utils.with_appcontext_if_needed()
```

Execute within the script's application context, in case there is one.

An exception is *flexmeasures run*, which has a click context at the time the decorator is called, but no longer has a click context at the time the decorated function is called, which, typically, is a request to the running FlexMeasures server.

Classes

class flexmeasures.data.schemas.utils.**MarshmallowClickMixin**(*args, **kwargs)

__init__(*args, **kwargs)

convert(value, param, ctx, **kwargs)

 Convert the value to the correct type. This is not called if the value is None (the missing value).

 This must accept string values from the command line, as well as values that are already the correct type. It may also convert other compatible types.

 The param and ctx arguments may be None in certain situations, such as when converting prompt input.

 If the value cannot be converted, call fail() with a descriptive message.

Parameters

- **value** – The value to convert.
- **param** – The parameter that is using this type to convert its value. May be None.
- **ctx** – The current context that arrived at this value. May be None.

get_metavar(param)

 Returns the metavar default for this param if it provides one.

name: **str**

 the descriptive name of this type

Exceptions

exception flexmeasures.data.schemas.utils.**FMValidationError**(message: *str* | *list* | *dict*, field_name: *str* = *'_schema'*, data: *Mapping*[*str*, *Any*] | *Iterable*[*Mapping*[*str*, *Any*]] | *None* = *None*, valid_data: *list*[*dict*[*str*, *Any*]] | *dict*[*str*, *Any*] | *None* = *None*, **kwargs)

Custom validation error class. It differs from the classic validation error by having two attributes, according to the USEF 2015 reference implementation. Subclasses of this error might adjust the *status* attribute accordingly.

Data schemas (Marshmallow)

flexmeasures.data.scripts

Modules

<code>flexmeasures.data.scripts.data_gen</code>	Populate the database with data we know or read in.
<code>flexmeasures.data.scripts.visualize_data_model</code>	

flexmeasures.data.scripts.data_gen

Populate the database with data we know or read in.

Functions

`flexmeasures.data.scripts.data_gen.add_default_account_roles(db: SQLAlchemy)`

Add a few useful account roles, inspired by USEF.

`flexmeasures.data.scripts.data_gen.add_default_asset_types(db: SQLAlchemy) → Dict[str, GenericAssetType]`

Add a few useful asset types.

`flexmeasures.data.scripts.data_gen.add_default_data_sources(db: SQLAlchemy)`

`flexmeasures.data.scripts.data_gen.add_default_user_roles(db: SQLAlchemy)`

Add a few useful user roles.

`flexmeasures.data.scripts.data_gen.add_transmission_zone_asset(country_code: str, db: SQLAlchemy) → GenericAsset`

Ensure a GenericAsset exists to model a transmission zone for a country.

`flexmeasures.data.scripts.data_gen.get_affected_classes(structure: bool = True, data: bool = False) → List`

`flexmeasures.data.scripts.data_gen.reset_db(db: SQLAlchemy)`

`flexmeasures.data.scripts.data_gen.save_tables(db: SQLAlchemy, backup_name: str = "", structure: bool = True, data: bool = False, backup_path: str = 'migrations/dumps')`

flexmeasures.data.scripts.visualize_data_model

Functions

`flexmeasures.data.scripts.visualize_data_model.check_sqlalchemy_schemadisplay_installation()`

Make sure the library which translates the model into a graph structure is installed with the right version.

`flexmeasures.data.scripts.visualize_data_model.create_schema_pic(*args, **kwargs)`

`flexmeasures.data.scripts.visualize_data_model.create_uml_pic(*args, **kwargs)`

`flexmeasures.data.scripts.visualize_data_model.show_image(*args, **kwargs)`

`flexmeasures.data.scripts.visualize_data_model.uses_dot(func)`

Decorator to make sure that if dot/graphviz (for drawing the graph) is not installed there is a proper message.

Useful scripts

flexmeasures.data.services

Modules

<code>flexmeasures.data.services.accounts</code>	
<code>flexmeasures.data.services.annotations</code>	
<code>flexmeasures.data.services.asset_grouping</code>	Convenience functions and class for accessing generic assets in groups.
<code>flexmeasures.data.services.data_sources</code>	
<code>flexmeasures.data.services.forecasting</code>	Logic around scheduling (jobs)
<code>flexmeasures.data.services.resources</code>	Generic services for accessing asset data.
<code>flexmeasures.data.services.scheduling</code>	Logic around scheduling (jobs)
<code>flexmeasures.data.services.sensors</code>	
<code>flexmeasures.data.services.time_series</code>	
<code>flexmeasures.data.services.timerange</code>	
<code>flexmeasures.data.services.users</code>	
<code>flexmeasures.data.services.utils</code>	

flexmeasures.data.services.accounts

Functions

`flexmeasures.data.services.accounts.get_account_roles(account_id: int) → list[flexmeasures.data.models.user.AccountRole]`

`flexmeasures.data.services.accounts.get_accounts(role_name: str | None = None) → list[Account]`
Return a list of Account objects. The role_name parameter allows to filter by role.

`flexmeasures.data.services.accounts.get_number_of_assets_in_account(account_id: int) → int`
Get the number of assets in an account.

flexmeasures.data.services.annotations

Functions

`flexmeasures.data.services.annotations.prepare_annotations_for_chart(df: DataFrame, event_starts_after: datetime | None = None, event_ends_before: datetime | None = None, max_line_length: int = 60) → DataFrame`

Prepare a DataFrame with annotations for use in a chart.

- Clips annotations outside the requested time window.
- Wraps on whitespace with a given max line length
- Stacks annotations for the same event

`flexmeasures.data.services.annotations.stack_annotations(x: DataFrame) → DataFrame`

Select earliest start, and include all annotations as a list.

The list of strings results in a multi-line text encoding in the chart.

flexmeasures.data.services.asset_grouping

Convenience functions and class for accessing generic assets in groups. For example, group by asset type or by location.

Functions

`flexmeasures.data.services.asset_grouping.get_asset_group_queries(group_by_type: bool = True, group_by_account: bool = False, group_by_location: bool = False, custom_aggregate_type_groups: Dict[str, List[str]] | None = None) → Dict[str, Query]`

An asset group is defined by Asset queries, which this function can generate.

Each query has a name (for the asset group it represents). These queries still need an executive call, like `all()`, `count()` or `first()`.

This function limits the assets to be queried to the current user’s account, if the user is not an admin.

Note: Make sure the current user has the “read” permission on their account (on `GenericAsset.__class__`?? See <https://github.com/FlexMeasures/flexmeasures/issues/200>) or is an admin.

Parameters

- **group_by_type** – If True, groups will be made for assets with the same type. We prefer pluralised group names here. Defaults to True.
- **group_by_account** – If True, groups will be made for assets within the same account. This makes sense for admins, as they can query across accounts.
- **group_by_location** – If True, groups will be made for assets at the same location. Naming of the location currently supports charge points (for EVSEs).
- **custom_aggregate_type_groups** – dict of asset type groupings (mapping group names to names of asset types). See also the setting `FLEXMEASURES_ASSET_TYPE_GROUPS`.

Classes

class flexmeasures.data.services.asset_grouping.**AssetGroup**(name: *str*, asset_query: *Query* | *None* = *None*)

This class represents a group of assets of the same type, offering some convenience functions for displaying their properties.

When initialised with an asset type name, the group will contain all assets of the given type that are accessible to the current user's account.

When initialised with a query for GenericAssets, as well, the group will list the assets returned by that query. This can be useful in combination with get_asset_group_queries, see above.

TODO: On a conceptual level, we can model two functionally useful ways of grouping assets: - AggregatedAsset if it groups assets of only 1 type, - GeneralizedAsset if it groups assets of multiple types There might be specialised subclasses, as well, for certain groups, like a market and consumers.

__init__(name: *str*, asset_query: *Query* | *None* = *None*)

The asset group name is either the name of an asset group or an individual asset.

property display_name: *str*

Attempt to get a beautiful name to show if possible.

property hover_label: *str* | *None*

Attempt to get a hover label to show if possible.

is_eligible_for_comparing_individual_traces(max_traces: *int* = 7) → *bool*

Decide whether comparing individual traces for assets in this asset group is a useful feature. The number of assets that can be compared is parametrizable with max_traces. Plot colors are reused if max_traces > 7, and run out if max_traces > 105.

property is_unique_asset: *bool*

Determines whether the resource represents a unique asset.

property parameterized_name: *str*

Get a parametrized name for use in javascript.

flexmeasures.data.services.data_sources

Functions

flexmeasures.data.services.data_sources.**get_or_create_source**(source: *User* | *str*, source_type: *str* | *None* = *None*, model: *str* | *None* = *None*, version: *str* | *None* = *None*, flush: *bool* = *True*) → *DataSource*

flexmeasures.data.services.data_sources.**get_source_or_none**(source: *int* | *str*, source_type: *str* | *None* = *None*) → *DataSource* | *None*

Parameters

- **source** – source id
- **source_type** – optionally, filter by source type

flexmeasures.data.services.forecasting

Logic around scheduling (jobs)

Functions

```
flexmeasures.data.services.forecasting.create_forecasting_jobs(sensor_id: int, start_of_roll:
    datetime, end_of_roll: datetime,
    resolution: timedelta | None =
    None, horizons:
    list[datetime.timedelta] | None =
    None,
    model_search_term='linear-OLS',
    custom_model_params: dict |
    None = None, enqueue: bool =
    True) → list[rq.job.Job]
```

Create forecasting jobs by rolling through a time window, for a number of given forecast horizons. Start and end of the forecasting jobs are equal to the time window (start_of_roll, end_of_roll) plus the horizon.

For example (with shorthand notation):

```
start_of_roll = 3pm end_of_roll = 5pm resolution = 15min horizons = [1h, 6h, 1d]
```

This creates the following 3 jobs:

- 1) forecast each quarter-hour from 4pm to 6pm, i.e. the 1h forecast
- 2) forecast each quarter-hour from 9pm to 11pm, i.e. the 6h forecast
- 3) forecast each quarter-hour from 3pm to 5pm the next day, i.e. the 1d forecast

If not given, relevant horizons are derived from the resolution of the posted data.

The job needs a model configurator, for which you can supply a model search term. If omitted, the current default model configuration will be used.

It's possible to customize model parameters, but this feature is (currently) meant to only be used by tests, so that model behaviour can be adapted to test conditions. If used outside of testing, an exception is raised.

if enqueue is True (default), the jobs are put on the redis queue.

Returns the redis-queue forecasting jobs which were created.

```
flexmeasures.data.services.forecasting.handle_forecasting_exception(job, exc_type, exc_value,
    traceback)
```

Decide if we can do something about this failure: * Try a different model * Re-queue at a later time (using rq_scheduler)

```
flexmeasures.data.services.forecasting.make_fixed_viewpoint_forecasts(sensor_id: int, horizon:
    timedelta, start:
    datetime, end: datetime,
    custom_model_params:
    dict | None = None) →
    int
```

Build forecasting model specs, make fixed-viewpoint forecasts, and save the forecasts made.

Each individual forecast is a belief about a time interval. Fixed-viewpoint forecasts share the same belief time. See the timely-beliefs lib for relevant terminology.

```
flexmeasures.data.services.forecasting.make_rolling_viewpoint_forecasts(sensor_id: int,  
                                                                           horizon: timedelta,  
                                                                           start: datetime, end:  
                                                                           datetime, cus-  
                                                                           tom_model_params:  
                                                                           dict | None = None)  
    → int
```

Build forecasting model specs, make rolling-viewpoint forecasts, and save the forecasts made.

Each individual forecast is a belief about a time interval. Rolling-viewpoint forecasts share the same belief horizon (the duration between belief time and knowledge time). Model specs are also retrained in a rolling fashion, but with its own frequency set in `custom_model_params`. See the `timely-beliefs` lib for relevant terminology.

Parameters

param sensor_id

int To identify which sensor to forecast

param horizon

timedelta duration between the end of each interval and the time at which the belief about that interval is formed

param start

datetime start of forecast period, i.e. start time of the first interval to be forecast

param end

datetime end of forecast period, i.e end time of the last interval to be forecast

param custom_model_params

dict pass in params which will be passed to the model specs configurator, e.g. `outcome_var_transformation`, only advisable to be used for testing.

returns

int the number of forecasts made

```
flexmeasures.data.services.forecasting.num_forecasts(start: datetime, end: datetime, resolution:  
                                                       timedelta) → int
```

Compute how many forecasts a job needs to make, given a resolution

Exceptions

exception `flexmeasures.data.services.forecasting.MisconfiguredForecastingJobException`

`flexmeasures.data.services.resources`

Generic services for accessing asset data.

TODO: This works with the legacy data model (esp. Assets), so it is marked for deprecation.

We are building `data.services.asset_grouping`, porting much of the code here. The data access logic here might also be useful for sensor data access logic we'll build elsewhere, but that's not quite certain at this point in time.

Functions

`flexmeasures.data.services.resources.can_access_asset(asset_or_sensor: Asset | Sensor) → bool`

Return True if: - the current user is an admin, or - the current user is the owner of the asset, or - the current user's organisation account owns the corresponding generic asset, or - the corresponding generic asset is public

todo: refactor to `def can_access_sensor(sensor: Sensor) -> bool` once `ui.views.state.state_view` stops calling it with an Asset
 todo: let this function use our new auth model (row-level authorization)
 todo: deprecate this function in favor of an authz decorator on the API route

`flexmeasures.data.services.resources.check_cache(attribute)`

Decorator for Resource class attributes to check if the resource has cached the attribute.

Example usage: `@check_cache("cached_data") def some_property(self):`

`return self.cached_data`

`flexmeasures.data.services.resources.get_asset_group_queries(custom_additional_groups: List[str] | None = None, all_users: bool = False) → Dict[str, Query]`

An asset group is defined by Asset queries. Each query has a name, and we prefer pluralised display names. They still need an executive call, like `all()`, `count()` or `first()`.

Parameters

- **custom_additional_groups** – list of additional groups next to groups that represent unique asset types. Valid names are: - “renewables”, to query all solar and wind assets - “EVSE”, to query all Electric Vehicle Supply Equipment - “location”, to query each individual location with assets
 (i.e. all EVSE at 1 location or each household)
- **all_users** – if True, do not filter out assets that do not belong to the user (use with care)

`flexmeasures.data.services.resources.get_assets(owner_id: int | None = None, order_by_asset_attribute: str = 'id', order_direction: str = 'desc') → List[Asset]`

Return a list of all Asset objects owned by `current_user` (or all users or a specific user - for this, admins can set an `owner_id`).

`flexmeasures.data.services.resources.get_center_location(user: User | None) → Tuple[float, float]`

Find the center position between all assets. If user is passed and not admin then we only consider assets owned by the user. TODO: if we introduce accounts, this logic should look for these assets.

`flexmeasures.data.services.resources.get_demand_from_bdf(bdf: DataFrame | BeliefsDataFrame) → DataFrame | BeliefsDataFrame`

Positive values become 0 and negative values become positive values.

`flexmeasures.data.services.resources.get_location_queries() → Dict[str, Query]`

We group EVSE assets by location (if they share a location, they belong to the same Charge Point) Like `get_asset_group_queries`, the values in the returned dict still need an executive call, like `all()`, `count()` or `first()`.

The Charge Points are named on the basis of the first EVSE in their list, using either the whole EVSE display name or that part that comes before a “-” delimiter. For example: If:

`evse_display_name = “Seoul Hilton - charger 1”`

Then:

`charge_point_display_name = “Seoul Hilton (Charge Point)”`

A Charge Point is a special case. If all assets on a location are of type EVSE, we can call the location a “Charge Point”.

`flexmeasures.data.services.resources.get_markets()` → `List[Market]`

Return a list of all Market objects.

`flexmeasures.data.services.resources.get_sensor_types(resource: Resource)` →
`List[WeatherSensorType]`

Return a list of WeatherSensorType objects applicable to the given resource.

`flexmeasures.data.services.resources.get_sensors(owner_id: int | None = None,`
`order_by_asset_attribute: str = 'id', order_direction:`
`str = 'desc')` → `List[Sensor]`

Return a list of all Sensor objects owned by current_user’s organisation account (or all users or a specific user - for this, admins can set an owner_id).

`flexmeasures.data.services.resources.get_supply_from_bdf(bdf: DataFrame | BeliefsDataFrame)` →
`DataFrame | BeliefsDataFrame`

Negative values become 0.

`flexmeasures.data.services.resources.group_assets_by_location(asset_list: List[Asset])` →
`List[List[Asset]]`

`flexmeasures.data.services.resources.has_assets(owner_id: int | None = None)` → `bool`

Return True if the current user owns any assets. (or all users or a specific user - for this, admins can set an owner_id).

`flexmeasures.data.services.resources.mask_inaccessible_assets(asset_queries: Query | Dict[str,`
`Query])` → `Query | Dict[str,`
`Query]`

Filter out any assets that the user should not be able to access.

We do not explicitly check user authentication here, because non-authenticated users are not admins and have no asset ownership, so applying this filter for non-admins masks all assets.

Classes

class `flexmeasures.data.services.resources.Resource(name: str)`

This class represents a group of assets of the same type, and provides helpful functions to retrieve their time series data and derived statistics.

Resolving asset type names

When initialised with a plural asset type name, the resource will contain all assets of the given type that are accessible to the user. When initialised with just one asset name, the resource will list only that asset.

Loading structure

Initialization only loads structural information from the database (which assets the resource groups).

Loading and caching time series

To load time series data for a certain time window, use the `load_sensor_data()` method. This loads beliefs data from the database and caches the results (as a named attribute). Caches are cleared when new time series data is loaded (or when the Resource instance ceases to exist).

Loading and caching derived statistics

Cached time series data is used to compute derived statistics, such as aggregates and scores. More specifically:

- demand and supply - aggregated values (summed over assets) - total values (summed over time) - mean values (averaged over time) (todo: add this property) - revenue and cost - profit/loss

When a derived statistic is called for, the results are also cached (using `@functools.cached_property`).

- `Resource(session["resource"]).assets`
- `Resource(session["resource"]).display_name`
- `Resource(session["resource"]).get_data()`

Usage

```
>>> from flask import session
>>> resource = Resource(session["resource"])
>>> resource.assets
>>> resource.display_name
>>> resource.load_sensor_data(Power)
>>> resource.cached_power_data
>>> resource.load_sensor_data(Price, sensor_key_attribute="market.name")
>>> resource.cached_price_data
```

`__init__(name: str)`

The resource name is either the name of an asset group or an individual asset.

property `aggregate_cost`: float

Returns total aggregate cost from demand.

property `aggregate_demand`: BeliefsDataFrame

Returns aggregate demand as positive values.

property `aggregate_profit_or_loss`: float

Returns total aggregate profit (loss is negative).

property `aggregate_revenue`: float

Returns total aggregate revenue from supply.

property `aggregate_supply`: BeliefsDataFrame

Returns aggregate supply (as positive values).

property cost: `Dict[str, float]`

Returns each asset's total cost from demand.

property demand: `Dict[str, BeliefsDataFrame]`

Returns each asset's demand as positive values.

property display_name: `str`

Attempt to get a beautiful name to show if possible.

property hover_label: `str | None`

Attempt to get a hover label to show if possible.

is_eligible_for_comparing_individual_traces(*max_traces: int = 7*) → `bool`

Decide whether comparing individual traces for assets in this resource is a useful feature. The number of assets that can be compared is parametrizable with `max_traces`. Plot colors are reused if `max_traces > 7`, and run out if `max_traces > 105`.

property is_unique_asset: `bool`

Determines whether the resource represents a unique asset.

load_sensor_data(*sensor_types: List[SensorType] | None = None, start: datetime | None = None, end: datetime | None = None, resolution: str | None = None, belief_horizon_window=(None, None), belief_time_window=(None, None), source_types: List[str] | None = None, exclude_source_types: List[str] | None = None*) → `Resource`

Load data for one or more assets and cache the results. If the time range parameters are `None`, they will be gotten from the session. The horizon window will default to the latest measurement (anything more in the future than the end of the time interval. To load data for a specific source, pass a source id.

Returns

self (to allow piping)

Usage

```
>>> resource = Resource()
>>> resource.load_sensor_data([Power], start=datetime(2014, 3, 1),
└end=datetime(2014, 3, 1))
>>> resource.cached_power_data
>>> resource.load_sensor_data([Power, Price], start=datetime(2014, 3, 1),
└end=datetime(2014, 3, 1)).cached_price_data
```

property parameterized_name: `str`

Get a parametrized name for use in javascript.

property revenue: `Dict[str, float]`

Returns each asset's total revenue from supply.

property supply: `Dict[str, BeliefsDataFrame]`

Returns each asset's supply as positive values.

property total_aggregate_demand: `float`

Returns total aggregate demand as a positive value.

property total_aggregate_supply: `float`

Returns total aggregate supply as a positive value.

property total_demand: `Dict[str, float]`

Returns each asset's total demand as a positive value.

property total_supply: `Dict[str, float]`

Returns each asset's total supply as a positive value.

flexmeasures.data.services.scheduling

Logic around scheduling (jobs)

Functions

`flexmeasures.data.services.scheduling.create_scheduling_job(sensor: Sensor, job_id: str | None = None, enqueue: bool = True, requeue: bool = False, force_new_job_creation: bool = False, **scheduler_kwargs) → Job`

Create a new Job, which is queued for later execution.

To support quick retrieval of the scheduling job, the job id is the unique entity address of the UDI event. That means one event leads to one job (i.e. actions are event driven).

As a rule of thumb, keep arguments to the job simple, and deserializable.

The life cycle of a scheduling job: 1. A scheduling job is born here (in `create_scheduling_job`). 2. It is run in `make_schedule` which writes results to the db. 3. If an error occurs (and the worker is configured accordingly), `handle_scheduling_exception` comes in.

Arguments: :param sensor: sensor for which the schedule is computed :param job_id: optionally, set a job id explicitly :param enqueue: if True, enqueues the job in case it is new :param requeue: if True, requeues the job in case it is not new and had previously failed

(this argument is used by the `@job_cache` decorator)

Parameters

force_new_job_creation – if True, this attribute forces a new job to be created (skipping cache) (this argument is used by the `@job_cache` decorator)

Returns

the job

`flexmeasures.data.services.scheduling.find_scheduler_class(sensor: Sensor) → type`

Find out which scheduler to use, given a sensor. This will morph into a logic store utility, and schedulers should be registered for asset types there, instead of this fixed lookup logic.

`flexmeasures.data.services.scheduling.get_data_source_for_job(job: Job | None) → DataSource | None`

Try to find the data source linked by this scheduling job.

We expect that enough info on the source was placed in the meta dict. This only happened with v0.12. For a transition period, we might have to support older jobs who haven't got that info. TODO: We should expect a job, once API v1.3 is deprecated.

`flexmeasures.data.services.scheduling.handle_scheduling_exception(job, exc_type, exc_value, traceback)`

Store exception as job meta data.

`flexmeasures.data.services.scheduling.load_custom_scheduler(scheduler_specs: dict) → type`

Read in custom scheduling spec. Attempt to load the Scheduler class to use.

The scheduler class should be derived from `flexmeasures.data.models.planning.Scheduler`. The Callable is assumed to be named “schedule”.

Example specs:

```
{
    "module": "/path/to/module.py", # or sthg importable, e.g. "package.module"
    "class": "NameOfSchedulerClass",
}
```

`flexmeasures.data.services.scheduling.make_schedule(sensor_id: int, start: datetime, end: datetime, resolution: timedelta, belief_time: datetime | None = None, flex_model: dict | None = None, flex_context: dict | None = None, flex_config_has_been_deserialized: bool = False) → bool`

This function computes a schedule. It returns True if it ran successfully.

It can be queued as a job (see `create_scheduling_job`). In that case, it will probably run on a different FlexMeasures node than where the job is created. In any case, this function expects `flex_model` and `flex_context` to not have been deserialized yet.

This is what this function does: - Find out which scheduler should be used & compute the schedule - Turn scheduled values into beliefs and save them to db

flexmeasures.data.services.sensors

Functions

`flexmeasures.data.services.sensors.get_sensors(account: Account | list[Account] | None, include_public_assets: bool = False, sensor_id_allowlist: list[int] | None = None, sensor_name_allowlist: list[str] | None = None) → list[Sensor]`

Return a list of Sensor objects that belong to the given account, and/or public sensors.

Parameters

- **account** – select only sensors from this account (or list of accounts)
- **include_public_assets** – if True, include sensors that belong to a public asset
- **sensor_id_allowlist** – optionally, allow only sensors whose id is in this list
- **sensor_name_allowlist** – optionally, allow only sensors whose name is in this list

flexmeasures.data.services.time_series

Functions

`flexmeasures.data.services.time_series.aggregate_values`(*bdf_dict*: *dict*[*Any*, *timely_beliefs.beliefs.classes.BeliefsDataFrame*])
→ *BeliefsDataFrame*

`flexmeasures.data.services.time_series.collect_time_series_data`(*old_sensor_names*: *str* | *list*[*str*],
make_query: *QueryCallType*,
query_window: *tuple*[*datetime* | *None*, *datetime* | *None*] = (*None*, *None*), *belief_horizon_window*:
tuple[*timedelta* | *None*, *timedelta* | *None*] = (*None*, *None*),
belief_time_window:
tuple[*datetime* | *None*, *datetime* | *None*] = (*None*, *None*),
belief_time: *datetime* | *None* = *None*, *user_source_ids*: *int* | *list*[*int*] = *None*, *source_types*:
list[*str*] | *None* = *None*,
exclude_source_types: *list*[*str*] | *None* = *None*, *resolution*: *str* | *timedelta* | *None* = *None*,
sum_multiple: *bool* = *True*) →
tb.BeliefsDataFrame | *dict*[*str*, *tb.BeliefsDataFrame*]

Get time series data from one or more old sensor models and rescale and re-package it to order.

We can (lazily) look up by pickle, or load from the database. In the latter case, we are relying on time series data (power measurements and prices at this point) to have the same relevant column names (*datetime*, *value*). We require an old sensor model name or list thereof. If the time range parameters are *None*, they will be gotten from the session. Response is a 2D *BeliefsDataFrame* with the column *event_value*. If data from multiple assets is retrieved, the results are being summed. Or, if *sum_multiple* is *False*, the response will be a dictionary with asset names as keys, each holding a *BeliefsDataFrame* as its value. The response might be an empty data frame if no data exists for these assets in this time range.

`flexmeasures.data.services.time_series.convert_query_window_for_demo`(*query_window*:
tuple[*datetime.datetime*, *datetime.datetime*]) →
tuple[*datetime.datetime*, *datetime.datetime*]

`flexmeasures.data.services.time_series.drop_non_unique_ids`(*a*: *int* | *list*[*int*], *b*: *int* | *list*[*int*]) →
list[*int*]

Removes all elements from B that are already in A.

`flexmeasures.data.services.time_series.drop_unchanged_beliefs`(*bdf*: *BeliefsDataFrame*) →
BeliefsDataFrame

Drop beliefs that are already stored in the database with an earlier belief time.

Also drop beliefs that are already in the data with an earlier belief time.

Quite useful function to prevent cluttering up your database with beliefs that remain unchanged over time.

`flexmeasures.data.services.time_series.find_sensor_by_name(name: str)`

Helper function: Find a sensor by name. TODO: make obsolete when we switched to collecting sensor data by sensor id rather than name

`flexmeasures.data.services.time_series.query_time_series_data(old_sensor_names: tuple[str],
make_query: QueryCallType,
query_window: tuple[datetime | None, datetime | None] = (None, None), belief_horizon_window:
tuple[timedelta | None, timedelta | None] = (None, None),
belief_time_window:
tuple[datetime | None, datetime | None] = (None, None), belief_time:
datetime | None = None,
user_source_ids: int | list[int] | None = None, source_types:
list[str] | None = None,
exclude_source_types: list[str] | None = None, resolution: str |
timedelta | None = None) →
dict[str, tb.BeliefsDataFrame]`

Run a query for time series data on the database for a tuple of assets. Here, we need to know that postgres only stores naive datetimes and we keep them as UTC. Therefore, we localize the result. Then, we resample the result, to fit the given resolution. * Returns a dictionary of asset names (as keys) and BeliefsDataFrames (as values), with each BeliefsDataFrame having an “event_value” column.

- Note that we convert string resolutions to datetime.timedelta objects.

`flexmeasures.data.services.time_series.set_bdf_source(bdf: BeliefsDataFrame, source_name: str) →
BeliefsDataFrame`

Set the source of the BeliefsDataFrame. We do this by re-setting the index (as source probably is part of the BeliefsDataFrame multi index), setting the source, then restoring the (multi) index.

flexmeasures.data.services.timerange

Functions

`flexmeasures.data.services.timerange.get_timerange(sensor_ids: list[int]) → tuple[datetime.datetime,
datetime.datetime]`

Get the start and end of the least recent and most recent event, respectively.

In case of no data, defaults to (now, now).

flexmeasures.data.services.users

Functions

`flexmeasures.data.services.users.create_user`(password: *str* = None, user_roles: *dict*[*str*, *str*] | *list*[*dict*[*str*, *str*]] | *str* | *list*[*str*] | None = None, check_email_deliverability: *bool* = True, account_name: *str* | None = None, **kwargs) → *User*

Convenience wrapper to create a new User object.

It hashes the password.

In addition to the user, this function can create - new Role objects (if user roles do not already exist) - an Account object (if it does not exist yet) - a new DataSource object that corresponds to the user

Remember to commit the session after calling this function!

`flexmeasures.data.services.users.delete_user`(user: *User*)

Delete the user (and also his assets and power measurements!).

Deleting oneself is not allowed.

Remember to commit the session after calling this function!

`flexmeasures.data.services.users.find_user_by_email`(user_email: *str*, keep_in_session: *bool* = True) → *User*

`flexmeasures.data.services.users.get_user`(id: *str*) → *User*

Get a user, raise if not found.

`flexmeasures.data.services.users.get_users`(account_name: *str* | None = None, role_name: *str* | None = None, account_role_name: *str* | None = None, only_active: *bool* = True) → *list*[*User*]

Return a list of User objects. The role_name parameter allows to filter by role. Set only_active to False if you also want non-active users.

`flexmeasures.data.services.users.remove_cookie_and_token_access`(user: *User*)

Remove access of current cookies and auth tokens for a user. This might be useful if you feel their password, cookie or tokens are compromised. in the former case, you can also call `set_random_password`.

Remember to commit the session after calling this function!

`flexmeasures.data.services.users.set_random_password`(user: *User*)

Randomise a user's password.

Remember to commit the session after calling this function!

Exceptions

exception `flexmeasures.data.services.users.InvalidFlexMeasuresUser`

flexmeasures.data.services.utils

Functions

`flexmeasures.data.services.utils.get_or_create_model(model_class: Type[GenericAsset | GenericAssetType | Sensor], **kwargs) → GenericAsset | GenericAssetType | Sensor`

Get a model from the database or add it if it's missing.

For example: `>>> weather_station_type = get_or_create_model(>>> GenericAssetType, >>> name="weather station", >>> description="A weather station with various sensors.", >>>)`

`flexmeasures.data.services.utils.hash_function_arguments(args, kwargs)`

Combines the hashes of the args and kwargs

The way to go to do `h(x,y) = hash(hash(x) || hash(y))` because it avoid the following:

- 1) `h(x,y) = hash(x || y)`, might create a collision if we delete the last n characters of x and we append them in front of y. e.g `h("abc", "d") = h("ab", "cd")`
- 2) we don't want to sort x and y, because we need the function `h(x,y) != h(y,x)`
- 3) extra hashing just avoid that we can't decompose the input arguments and track if the same args or kwarg are called several times. More of a security measure I think.

source: <https://crypto.stackexchange.com/questions/55162/best-way-to-hash-two-values-into-one>

`flexmeasures.data.services.utils.job_cache(queue: str)`

To avoid recomputing the same task multiple times, this decorator checks if the function has already been called with the same arguments. Input arguments are hashed and stored as Redis keys with the values being the job IDs `input_arguments_hash:job_id`.

The benefits of using redis to store the input arguments over a local cache, such as LRU Cache, are: 1) It will work in distributed environments (in computing clusters), where multiple workers will avoid repeating

work as the cache will be shared across them.

- 2) Cached calls are logged, which means that we can easily debug.
- 3) Cache will still be there on restarts.

Arguments :param queue: name of the queue

`flexmeasures.data.services.utils.make_hash_sha256(o)`

SHA256 instead of Python's hash function because apparently, python native hashing function yields different results on restarts. Source: <https://stackoverflow.com/a/42151923>

`flexmeasures.data.services.utils.make_hashable(o)`

Function to create hashes for dictionaries with nested objects Source: <https://stackoverflow.com/a/42151923>

Business logic

flexmeasures.data.transactional

These, and only these, functions should help you with treating your own code in the context of one database transaction. Which makes our lives easier.

Functions

`flexmeasures.data.transactional.after_request_exception_rollback_session(exception)`

Central place to handle transactions finally. So - usually your view code should not have to deal with rolling back. Our policy *is* that we don't auto-commit (we used to do that here). Some more reading is e.g. here <https://github.com/pallets/flask-sqlalchemy/issues/216>

Register this on your app via the `teardown_request` setup method. We roll back the session if there was any error (which only has an effect if the session has not yet been committed).

Flask-SQLAlchemy is closing the scoped sessions automatically.

`flexmeasures.data.transactional.as_transaction(db_function)`

Decorator for handling any function which contains SQLAlchemy commands as one database transaction (ACID). Calls db operation function and when it is done, commits the db session. Rolls back the session if anything goes wrong. If useful, the first argument can be the db (SQLAlchemy) object and the rest of the args are sent through to the function. If this happened, the session is closed at the end.

Exceptions

exception `flexmeasures.data.transactional.PartialTaskCompletionException`

By raising this Exception in a task, no rollback will happen even if not everything was successful and the data which was generated will get committed. The task status will still be False, so the non-successful parts can be inspected.

flexmeasures.data.utils

Utils around the data models and db sessions

Functions

`flexmeasures.data.utils.get_data_source(data_source_name: str, data_source_model: str | None = None, data_source_version: str | None = None, data_source_type: str = 'script') → DataSource`

Make sure we have a data source. Create one if it doesn't exist, and add to session. Meant for scripts that may run for the first time.

`flexmeasures.data.utils.save_to_db(data: BeliefsDataFrame | BeliefsSeries | list[BeliefsDataFrame | BeliefsSeries], bulk_save_objects: bool = False, save_changed_beliefs_only: bool = True) → str`

Save the timed beliefs to the database.

Note: This function does not commit. It does, however, flush the session. Best to keep transactions short.

We make the distinction between updating beliefs and replacing beliefs.

Updating beliefs

An updated belief is a belief from the same source as some already saved belief, and about the same event, but with a later belief time. If it has a different event value, then it represents a changed belief. Note that it is possible to explicitly record unchanged beliefs (i.e. updated beliefs with a later belief time, but with the same event value), by setting `save_changed_beliefs_only` to `False`.

Replacing beliefs

A replaced belief is a belief from the same source as some already saved belief, and about the same event and with the same belief time, but with a different event value. Replacing beliefs is not allowed, because messing with the history corrupts data lineage. Corrections should instead be recorded as updated beliefs. Servers in ‘play’ mode are exempt from this rule, to facilitate replaying simulations.

Parameters

- **data** – BeliefsDataFrame (or a list thereof) to be saved
- **bulk_save_objects** – if `True`, objects are bulk saved with `session.bulk_save_objects()`, which is quite fast but has several caveats, see: https://docs.sqlalchemy.org/orm/persistence_techniques.html#bulk-operations-caveats
- **save_changed_beliefs_only** – if `True`, unchanged beliefs are skipped (updated beliefs are only stored if they represent changed beliefs) if `False`, all updated beliefs are stored

Returns

status string, one of the following: - ‘success’: all beliefs were saved - ‘success_with_unchanged_beliefs_skipped’: not all beliefs represented a state change - ‘success_but_nothing_new’: no beliefs represented a state change

`flexmeasures.data.utils.save_to_session(objects: list[sqlalchemy.orm.decl_api.Model], overwrite: bool = False)`

Utility function to save to database, either efficiently with a bulk save, or inefficiently with a merge save.

Models & schemata, as well as business logic (queries & services).

Functions

`flexmeasures.data.register_at(app: Flask)`

5.3.44 flexmeasures.ui

Modules

<code>flexmeasures.ui.crud</code>	Backoffice UI for CRUD functionality
<code>flexmeasures.ui.error_handlers</code>	Error views for UI purposes.
<code>flexmeasures.ui.utils</code>	Utility functions for UI logic
<code>flexmeasures.ui.views</code>	This module hosts the views.

flexmeasures.ui.crud

Modules

<code>flexmeasures.ui.crud.accounts</code>
<code>flexmeasures.ui.crud.api_wrapper</code>
<code>flexmeasures.ui.crud.assets</code>
<code>flexmeasures.ui.crud.users</code>

flexmeasures.ui.crud.accounts

Functions

`flexmeasures.ui.crud.accounts.get_account(account_id: str) → dict`

`flexmeasures.ui.crud.accounts.get_accounts() → list[dict]`
`/accounts`

Classes

class `flexmeasures.ui.crud.accounts.AccountCrudUI`

```

    get(account_id: str)
        /accounts/<account_id>

    index()
        /accounts

```

flexmeasures.ui.crud.api_wrapper

Classes

class `flexmeasures.ui.crud.api_wrapper.InternalApi`

Simple wrapper around the requests lib, which we use to talk to our actual internal JSON Api via requests. It can only be used to perform requests on the same URL root as the current request. - We use this because it is cleaner than calling the API code directly.

That would re-use the same request we are working on here, which works differently in some ways like content-type and authentication. The Flask/Werkzeug request is also immutable, so we could not adapt the request anyways.

- Also, we implement auth token handling
- Finally we have some logic to control which error codes we want to raise.

`_maybe_raise`(*response: requests.Response, do_not_raise_for: list | None = None*)

Raise an error in the API (4xx, 5xx) if the error code is not in the list of codes we want to ignore / handle explicitly.

flexmeasures.ui.crud.assets

Functions

`flexmeasures.ui.crud.assets.get_assets_by_account`(*account_id: int | str | None*) → list[*GenericAsset*]

`flexmeasures.ui.crud.assets.process_internal_api_response`(*asset_data: dict, asset_id: int | None = None, make_obj=False*) → *GenericAsset* | dict

Turn data from the internal API into something we can use to further populate the UI. Either as an asset object or a dict for form filling.

If we add other data by querying the database, we make sure the asset is not in the session afterwards.

`flexmeasures.ui.crud.assets.user_can_create_assets`() → bool

`flexmeasures.ui.crud.assets.user_can_delete`(*asset*) → bool

`flexmeasures.ui.crud.assets.with_options`(*form: AssetForm | NewAssetForm*) → *AssetForm* | *NewAssetForm*

Classes

class `flexmeasures.ui.crud.assets.AssetCrudUI`

These views help us offer a Jinja2-based UI. The main focus on logic is the API, so these views simply call the API functions, and deal with the response. Some new functionality, like fetching accounts and asset types, is added here.

`delete_with_data`(*id: str*)

Delete via /assets/delete_with_data/<id>

`get`(*id: str*)

GET from /assets/<id> where id can be ‘new’ (and thus the form for asset creation is shown)

`index`(*msg=""*)

GET from /assets

List the user’s assets. For admins, list across all accounts.

`owned_by`(*account_id: str*)

/assets/owned_by/<account_id>

`post`(*id: str*)

POST to /assets/<id>, where id can be ‘create’ (and thus a new asset is made from POST data) Most of the code deals with creating a user for the asset if no existing is chosen.

class `flexmeasures.ui.crud.assets.AssetForm`(*args, **kwargs)

The default asset form only allows to edit the name and location.

`process_api_validation_errors`(*api_response: dict*)

Process form errors from the API for the WTForm

to_json() → dict

turn form data into a JSON we can POST to our internal API

validate_on_submit()

Call `validate()` only if the form is submitted. This is a shortcut for `form.is_submitted()` and `form.validate()`.

class flexmeasures.ui.crud.assets.**NewAssetForm**(*args, **kwargs)

Here, in addition, we allow to set asset type and account.

flexmeasures.ui.crud.users

Functions

flexmeasures.ui.crud.users.**get_users_by_account**(account_id: int | str, include_inactive: bool = False) → list[User]

flexmeasures.ui.crud.users.**process_internal_api_response**(user_data: dict, user_id: int | None = None, make_obj=False) → User | dict

Turn data from the internal API into something we can use to further populate the UI. Either as a user object or a dict for form filling.

flexmeasures.ui.crud.users.**render_user**(user: User | None, asset_count: int = 0, msg: str | None = None)

Classes

class flexmeasures.ui.crud.users.**UserCrudUI**

get(id: str)

GET from /users/<id>

index()

/users

reset_password_for(id: str)

/users/reset_password_for/<id> Set the password to something random (in case of worries the password might be compromised) and send instructions on how to reset.

toggle_active(id: str)

Toggle activation status via /users/toggle_active/<id>

class flexmeasures.ui.crud.users.**UserForm**(*args, **kwargs)

Backoffice UI for CRUD functionality

flexmeasures.ui.error_handlers

Error views for UI purposes.

Functions

`flexmeasures.ui.error_handlers.add_html_error_views(app: Flask)`

`flexmeasures.ui.error_handlers.handle_500_error(e: InternalServerError)`

`flexmeasures.ui.error_handlers.handle_bad_request(e: BadRequest)`

`flexmeasures.ui.error_handlers.handle_generic_http_exception(e: HTTPException)`

This handles all known exception as fall-back

`flexmeasures.ui.error_handlers.handle_not_found(e)`

`flexmeasures.ui.error_handlers.unauthenticated_handler()`

An unauthenticated handler which renders an HTML error page

`flexmeasures.ui.error_handlers.unauthorized_handler()`

An unauthorized handler which renders an HTML error page

flexmeasures.ui.utils

Modules

<i>flexmeasures.ui.utils.chart_defaults</i>	
<i>flexmeasures.ui.utils.view_utils</i>	Utilities for views

flexmeasures.ui.utils.chart_defaults

flexmeasures.ui.utils.view_utils

Utilities for views

Functions

`flexmeasures.ui.utils.view_utils.accountname(account_id) → str`

`flexmeasures.ui.utils.view_utils.asset_icon_name(asset_type_name: str) → str`

Icon name for this asset type.

This can be used for UI html templates made with Jinja. `ui.__init__` makes this function available as the filter “asset_icon”.

For example:

```
<i class={{ asset_type.name | asset_icon }}></i>
```

becomes (for a battery):

```
<i class="icon-battery"></i>
```

`flexmeasures.ui.utils.view_utils.clear_session()`

`flexmeasures.ui.utils.view_utils.ensure_timing_vars_are_set(time_window: tuple[datetime | None, datetime | None], resolution: str | None) → tuple[tuple[datetime, datetime], str]`

Ensure that time window and resolution variables are set, even if we don't have them available — in that case, get them from the session.

`flexmeasures.ui.utils.view_utils.get_git_description() → tuple[str, int, str]`

Get information about the SCM (git) state if possible (if a .git directory exists).

Returns the latest git version (tag) as a string, the number of commits since then as an int and the current commit hash as string.

`flexmeasures.ui.utils.view_utils.render_flexmeasures_template(html_filename: str, **variables)`

Render template and add all expected template variables, plus the ones given as ***variables*.

`flexmeasures.ui.utils.view_utils.set_individual_traces_for_session()`

Set session["showing_individual_traces_for"] to a value ("none", "power", "schedules").

`flexmeasures.ui.utils.view_utils.set_session_market(resource: Resource) → Market`

Set session["market"] to something, based on the available markets or the request. Returns the selected market, or None.

`flexmeasures.ui.utils.view_utils.set_session_resource(assets: list[Asset], groups_with_assets: list[str]) → Resource | None`

Set session["resource"] to something, based on the available asset groups or the request.

Returns the selected resource instance, or None.

`flexmeasures.ui.utils.view_utils.set_session_sensor_type(accepted_sensor_types: list[flexmeasures.data.models.weather.WeatherSensorType]) → WeatherSensorType`

Set session["sensor_type"] to something, based on the available sensor types or the request. Returns the selected sensor type, or None.

`flexmeasures.ui.utils.view_utils.set_time_range_for_session()`

Set period on session if they are not yet set. The daterangepicker sends times as tz-aware UTC strings. We re-interpret them as being in the server's timezone. Also set the forecast horizon, if given.

TODO: event_[stars|ends]_before are used on the new asset and sensor pages.

We simply store the UTC strings. It might be that the other settings & logic can be deprecated when we clean house. Tip: grep for timerangeEnd, where end_time is used in our base template,

and then used in the daterangepicker. We seem to use litepicker now.

`flexmeasures.ui.utils.view_utils.username(user_id) → str`

Utility functions for UI logic

flexmeasures.ui.views

Modules

<code>flexmeasures.ui.views.control</code>
<code>flexmeasures.ui.views.logged_in_user</code>
<code>flexmeasures.ui.views.new_dashboard</code>
<code>flexmeasures.ui.views.sensors</code>

flexmeasures.ui.views.control

Functions

`flexmeasures.ui.views.control.control_view()`

Control view. This page lists balancing opportunities for a selected time window. The user can place manual orders or choose to automate the ordering process.

flexmeasures.ui.views.logged_in_user

Functions

`flexmeasures.ui.views.logged_in_user.logged_in_user_view()`

Basic information about the currently logged-in user. Plus basic actions (logout, reset pwd)

flexmeasures.ui.views.new_dashboard

Functions

`flexmeasures.ui.views.new_dashboard.dashboard_view()`

Dashboard view. This is the default landing page. It shows a map with the location of all of the assets in the user's account, or all assets if the user is an admin. Assets are grouped by asset type, which leads to map layers and a table with asset counts by type. Admins get to see all assets.

TODO: Assets for which the platform has identified upcoming balancing opportunities are highlighted.

flexmeasures.ui.views.sensors

Classes

class flexmeasures.ui.views.sensors.SensorUI

This view creates several new UI endpoints for viewing sensors.

todo: consider extending this view for crud purposes

get(*id: int*)

GET from /sensors/<id>

get_chart(*id, **kwargs*)

GET from /sensors/<id>/chart

This module hosts the views. This file registers blueprints and hosts some helpful functions

Functions

flexmeasures.ui.views.docs_view()

Render the Sphinx documentation

Backoffice user interface & charting support.

Functions

flexmeasures.ui.add_jinja_filters(*app*)

flexmeasures.ui.add_jinja_variables(*app*)

flexmeasures.ui.register_at(*app: Flask*)

This can be used to register this blueprint together with other ui-related things

flexmeasures.ui.register_rq_dashboard(*app*)

5.3.45 flexmeasures.utils

Modules

<code>flexmeasures.utils.app_utils</code>	Utils for serving the FlexMeasures app
<code>flexmeasures.utils.calculations</code>	Various calculations
<code>flexmeasures.utils.coding_utils</code>	Various coding utils (e.g.
<code>flexmeasures.utils.config_defaults</code>	Our configuration requirements and defaults
<code>flexmeasures.utils.config_utils</code>	Reading in configuration
<code>flexmeasures.utils.entity_address_utils</code>	
<code>flexmeasures.utils.error_utils</code>	Utils for handling of errors
<code>flexmeasures.utils.flexmeasures_inflection</code>	FlexMeasures way of handling inflection
<code>flexmeasures.utils.geo_utils</code>	
<code>flexmeasures.utils.grid_cells</code>	
<code>flexmeasures.utils.plugin_utils</code>	Utils for registering FlexMeasures plugins
<code>flexmeasures.utils.time_utils</code>	Utils for dealing with time
<code>flexmeasures.utils.unit_utils</code>	Utility module for unit conversion

flexmeasures.utils.app_utils

Utils for serving the FlexMeasures app

Functions

`flexmeasures.utils.app_utils.find_first_applicable_config_entry`(*configs*: *list*, *setting_name*: *str*, *app*: *Flask* | *None* = *None*) → *str* | *None*

`flexmeasures.utils.app_utils.init_sentry`(*app*: *Flask*)

Configure Sentry. We need the app to read the Sentry DSN from configuration, and also to send some additional meta information.

`flexmeasures.utils.app_utils.parse_config_entry_by_account_roles`(*config*: *str* | *tuple*[*str*, *list*[*str*]], *setting_name*: *str*, *app*: *Flask* | *None* = *None*) → *str* | *None*

Parse a config entry (which can be a string, e.g. “dashboard” or a tuple, e.g. (“dashboard”, [“MDC”])). In the latter case, return the first item (a string) only if the current user’s account roles match with the list of roles in the second item. Otherwise, return None.

`flexmeasures.utils.app_utils.root_dispatcher`()

Re-routes to root views fitting for the current user, depending on the FLEXMEASURES_ROOT_VIEW setting.

`flexmeasures.utils.app_utils.set_secret_key`(*app*, *filename*=‘secret_key’)

Set the SECRET_KEY or exit.

We first check if it is already in the config.

Then we look for it in environment var SECRET_KEY.

Finally, we look for *filename* in the app’s instance directory.

If nothing is found, we print instructions to create the secret and then exit.

flexmeasures.utils.calculations

Various calculations

Functions

`flexmeasures.utils.calculations.apply_stock_changes_and_losses`(*initial*: *float*, *changes*: *list[float]*, *storage_efficiency*: *float* | *list[float]*, *how*: *str* = 'linear', *decimal_precision*: *int* | *None* = *None*) → *list[float]*

Assign stock changes and determine losses from storage efficiency.

The initial stock is exponentially decayed, as with each consecutive (constant-resolution) time step, some constant percentage of the previous stock remains. For example:

$$100 \rightarrow 90 \rightarrow 81 \rightarrow 72.9 \rightarrow \dots$$

For computing the decay of the changes, we make an assumption on how a delta d is distributed within a given time step. In case it happens at a constant rate, this leads to a linear stock change from one time step to the next.

An e is introduced when we apply exponential decay to that. To see that, imagine we cut one time step in n pieces (each with a stock change $\frac{d}{n}$), apply the efficiency to each piece k (for the corresponding fraction of the time step k/n), and then take the limit $n \rightarrow \infty$:

$$\lim_{n \rightarrow \infty} \sum_{k=0}^n \frac{d}{n} \eta^{k/n}$$

which is:

$$d \cdot \frac{\eta - 1}{e^\eta}$$

Parameters

- **initial** – initial stock
- **changes** – stock change for each step
- **storage_efficiency** – ratio of stock left after a step (constant ratio or one per step)
- **how** – left, right or linear; how stock changes should be applied, which affects how losses are applied
- **decimal_precision** – Optional decimal precision to round off results (useful for tests failing over machine precision)

`flexmeasures.utils.calculations.drop_nan_rows`(*a*, *b*)

`flexmeasures.utils.calculations.integrate_time_series`(*series*: *pd.Series*, *initial_stock*: *float*, *up_efficiency*: *float* | *pd.Series* = 1, *down_efficiency*: *float* | *pd.Series* = 1, *storage_efficiency*: *float* | *pd.Series* = 1, *decimal_precision*: *int* | *None* = *None*) → *pd.Series*

Integrate time series of length n and `inclusive="left"` (representing a flow) to a time series of length $n+1$ and `inclusive="both"` (representing a stock), given an initial stock (i.e. the constant of integration). The unit of time is hours: i.e. the stock unit is flow unit times hours (e.g. a flow in kW becomes a stock in kWh). Optionally, set a decimal precision to round off the results (useful for tests failing over machine precision).

```
>>> s = pd.Series([1, 2, 3, 4], index=pd.date_range(datetime(2001, 1, 1, 5),  
↳datetime(2001, 1, 1, 6), freq=timedelta(minutes=15), inclusive="left"))  
>>> integrate_time_series(s, 10)  
2001-01-01 05:00:00    10.00  
2001-01-01 05:15:00    10.25  
2001-01-01 05:30:00    10.75  
2001-01-01 05:45:00    11.50  
2001-01-01 06:00:00    12.50  
Freq: D, dtype: float64
```

```
>>> s = pd.Series([1, 2, 3, 4], index=pd.date_range(datetime(2001, 1, 1, 5),  
↳datetime(2001, 1, 1, 7), freq=timedelta(minutes=30), inclusive="left"))  
>>> integrate_time_series(s, 10)  
2001-01-01 05:00:00    10.0  
2001-01-01 05:30:00    10.5  
2001-01-01 06:00:00    11.5  
2001-01-01 06:30:00    13.0  
2001-01-01 07:00:00    15.0  
dtype: float64
```

`flexmeasures.utils.calculations.mean_absolute_error(y_true: ndarray, y_forecast: ndarray)`

`flexmeasures.utils.calculations.mean_absolute_percentage_error(y_true: ndarray, y_forecast: ndarray)`

`flexmeasures.utils.calculations.weighted_absolute_percentage_error(y_true: ndarray, y_forecast: ndarray)`

flexmeasures.utils.coding_utils

Various coding utils (e.g. around function decoration)

Functions

`flexmeasures.utils.coding_utils.deprecated(alternative, version: str | None = None)`

Decorator for printing a warning error. `alternative`: importable object to use as an alternative to the function/method decorated version: `version` in which the function will be sunset

`flexmeasures.utils.coding_utils.find_classes_module(module, superclass)`

`flexmeasures.utils.coding_utils.find_classes_modules(module, superclass, skiptest=True)`

`flexmeasures.utils.coding_utils.flatten_unique(nested_list_of_objects: list) → list`

Returns unique objects in a possibly nested (one level) list of objects.

For example: `>>> flatten_unique([1, [2, 3, 4], 3, 5]) <<< [1, 2, 3, 4, 5]`

`flexmeasures.utils.coding_utils.get_classes_module(module, superclass, skiptest=True) → dict`

`flexmeasures.utils.coding_utils.make_registering_decorator(foreign_decorator)`

Returns a copy of `foreign_decorator`, which is identical in every way(*), except also appends a `.decorator` property to the callable it spits out.

(*) We can be somewhat “hygienic”, but `new_decorator` still isn’t signature-preserving, i.e. you will not be able to get a runtime list of parameters. For that, you need hackish libraries... but in this case, the only argument is `func`, so it’s not a big issue

Works on outermost decorators, based on Method 3 of <https://stackoverflow.com/a/5910893/13775459>

`flexmeasures.utils.coding_utils.methods_with_decorator(cls, decorator)`

Returns all methods in `CLS` with `DECORATOR` as the outermost decorator.

`DECORATOR` must be a “registering decorator”; one can make any decorator “registering” via the `make_registering_decorator` function.

Doesn’t work for the `@property` decorator, but does work for the `@functools.cached_property` decorator.

Works on outermost decorators, based on Method 3 of <https://stackoverflow.com/a/5910893/13775459>

`flexmeasures.utils.coding_utils.optional_arg_decorator(fn)`

A decorator which `_optionally_` accepts arguments.

So a decorator like this:

```
@optional_arg_decorator def register_something(fn, optional_arg = 'Default Value'):
    ... return fn
```

will work in both of these usage scenarios:

```
@register_something('Custom Name') def custom_name():
    pass

@register_something def default_name():
    pass
```

Thanks to https://stackoverflow.com/questions/3888158/making-decorators-with-optional-arguments#comment65959042_24617244

`flexmeasures.utils.coding_utils.rgetattr(obj, attr, *args)`

Get chained properties.

Usage

```
>>> class Pet:
    def __init__(self):
        self.favorite_color = "orange"
>>> class Person:
    def __init__(self):
        self.pet = Pet()
>>> p = Person()
>>> rgetattr(p, 'pet.favorite_color') # "orange"
```

From <https://stackoverflow.com/a/31174427/13775459>

`flexmeasures.utils.coding_utils.sort_dict(unsorted_dict: dict) → dict`

`flexmeasures.utils.coding_utils.timeit(func)`

Decorator for printing the time it took to execute the decorated function.

flexmeasures.utils.config_defaults

Our configuration requirements and defaults

This can be adjusted per environment here. Anything confidential should be handled outside of source control (e.g. a SECRET KEY file is generated on first install, and confidential settings can be set via the <app-env>-conf.py file.

Classes

class flexmeasures.utils.config_defaults.**Config**

If there is a useful default value, set it here. Otherwise, set to None, so that it can be set either by subclasses or the env-specific config script.

class flexmeasures.utils.config_defaults.**DevelopmentConfig**

class flexmeasures.utils.config_defaults.**DocumentationConfig**

class flexmeasures.utils.config_defaults.**ProductionConfig**

class flexmeasures.utils.config_defaults.**StagingConfig**

class flexmeasures.utils.config_defaults.**TestingConfig**

flexmeasures.utils.config_utils

Reading in configuration

Functions

flexmeasures.utils.config_utils.**are_required_settings_complete**(app) → bool

Check if all settings we expect are not None. Return False if they are not. Printout helpful advice.

flexmeasures.utils.config_utils.**check_app_env**(env: str | None)

flexmeasures.utils.config_utils.**configure_logging**()

Configure and register logging

flexmeasures.utils.config_utils.**get_config_warnings**(app) → tuple[list[str], list[str]]

return missing settings and the warnings for them.

flexmeasures.utils.config_utils.**get_configuration_keys**(app) → list[str]

Collect all members of DefaultConfig who are not in-built fields or callables.

flexmeasures.utils.config_utils.**read_config**(app: Flask, custom_path_to_config: str | None)

Read configuration from various expected sources, complain if not setup correctly.

flexmeasures.utils.config_utils.**read_custom_config**(app: Flask, suggested_path_to_config,
path_to_config_home, path_to_config_instance)
→ str

Read in a custom config file and env vars. For the config, there are two fallback options, tried in a specific order: If no custom path is suggested, we'll try the path in the home dir first, then in the instance dir.

Return the path to the config file.

`flexmeasures.utils.config_utils.read_env_vars(app: Flask)`

Read in what we support as environment settings. At the moment, these are: - All required and warnable variables
- Logging settings - access tokens - plugins (handled in plugin utils)

flexmeasures.utils.entity_address_utils

Functions

`flexmeasures.utils.entity_address_utils.build_ea_scheme_and_naming_authority(host: str, host_auth_start_month: str | None = None) → str`

This function creates the host identification part of USEF's EA1 addressing scheme, so everything but the locally unique string.

If not given nor configured, `host_auth_start_month` is the start of the next month for localhost.

`flexmeasures.utils.entity_address_utils.build_entity_address(entity_info: dict, entity_type: str, host: str | None = None, fm_scheme: str = 'fm1') → str`

Build an entity address.

`fm1` type entity address should use `entity_info["sensor_id"]` todo: implement entity addresses for actuators with `entity_info["actuator_id"]` (first ensuring globally unique ids across sensors and actuators)

If the host is not given, it is attempted to be taken from the request. `entity_info` is expected to contain the required fields for the custom string.

Returns the address as string.

`flexmeasures.utils.entity_address_utils.get_domain_parts(domain: str) → ExtractResult`
wrapper for calling `tlextract` as it logs things about file locks we don't care about.

`flexmeasures.utils.entity_address_utils.get_host() → str`

Get host from the context of the request.

Strips off `www.` but keeps subdomains. Can be localhost, too.

`flexmeasures.utils.entity_address_utils.parse_entity_address(entity_address: str, entity_type: str, fm_scheme: str = 'fm1') → dict`

Parses an entity address into an info dict.

Returns a dictionary with `scheme`, `naming_authority` and various other fields, depending on the entity type and FlexMeasures scheme (see examples above). Returns `None` if entity type is unknown or `entity_address` is not parse-able. We recommend to *return invalid_domain()* in that case.

Examples for the `fm1` scheme:

```
sensor      =      ea1.2021-01.io.flexmeasures:fm1.42      sensor      =      ea1.2021-01.io.flexmeasures:fm1.<sensor_id>
connection =      ea1.2021-01.io.flexmeasures:fm1.<sensor_id>
market     =      ea1.2021-01.io.flexmeasures:fm1.<sensor_id> weather_station =      ea1.2021-01.io.flexmeasures:fm1.<sensor_id>
todo: UDI events are not yet modelled in the fm1 scheme, but will probably be ea1.2021-01.io.flexmeasures:fm1.<actuator_id>
```

Examples for the `fm0` scheme:

```
connection      =      ea1.2021-01.localhost:fm0.40:30      connection      =      ea1.2021-
01.io.flexmeasures:fm0.<owner_id>:<asset_id>      weather_sensor      =      ea1.2021-
01.io.flexmeasures:fm0.temperature:52:73.0      weather_sensor      =      ea1.2021-
01.io.flexmeasures:fm0.<sensor_type>:<latitude>:<longitude>      market      =      ea1.2021-
01.io.flexmeasures:fm0.epex_da      market      =      ea1.2021-01.io.flexmeasures:fm0.<market_name>
event      =      ea1.2021-01.io.flexmeasures:fm0.40:30:302:soc      event      =      ea1.2021-
01.io.flexmeasures:fm0.<owner_id>:<asset_id>:<event_id>:<event_type>
```

For the fm0 scheme, the ‘fm0.’ part is optional, for backwards compatibility.

`flexmeasures.utils.entity_address_utils.reverse_domain_name(domain: str | TldExtractResult) → str`

Returns the reverse notation of the domain. You can pass in a string domain or an extraction result from `tlldextract`

Exceptions

exception `flexmeasures.utils.entity_address_utils.EntityAddressException`

flexmeasures.utils.error_utils

Utils for handling of errors

Functions

`flexmeasures.utils.error_utils.add_basic_error_handlers(app: Flask)`

Register classes we care about with the generic handler. See also the `auth` package for auth-specific error handling (Unauthorized, Forbidden)

`flexmeasures.utils.error_utils.error_handling_router(error: HTTPException)`

Generic handler for errors. We respond in json if the request content-type is JSON. The `ui` package can also define how it wants to render HTML errors, by setting a function.

`flexmeasures.utils.error_utils.get_err_source_info(original_traceback=None) → dict`

Use this when an error is handled to get info on where it occurred.

`flexmeasures.utils.error_utils.log_error(exc: Exception, error_msg: str)`

Collect meta data about the exception and log it. `error_msg` comes in as an extra attribute because `Exception` implementations differ here.

`flexmeasures.utils.error_utils.print_query(query: Query) → str`

Print full SQLAlchemy query with compiled parameters.

Recommended use as developer tool only.

Adapted from <https://stackoverflow.com/a/63900851/13775459>

flexmeasures.utils.flexmeasures_inflection

FlexMeasures way of handling inflection

Functions

`flexmeasures.utils.flexmeasures_inflection.capitalize(x: str, lower_case_remainder: bool = False) → str`

Capitalize string with control over whether to lower case the remainder.

`flexmeasures.utils.flexmeasures_inflection.humanize(word)`

`flexmeasures.utils.flexmeasures_inflection.join_words_into_a_list(words: list[str]) → str`

`flexmeasures.utils.flexmeasures_inflection.parameterize(word)`

Parameterize the word, so it can be used as a python or javascript variable name. For example: `>>> word = "Acme® EV-Charger™"` `"acme_ev_chargertm"`

`flexmeasures.utils.flexmeasures_inflection.pluralize(word, count: str | int | None = None)`

`flexmeasures.utils.flexmeasures_inflection.titleize(word)`

Acronym exceptions are not yet supported by the inflection package, even though Ruby on Rails, of which the package is a port, does.

In most cases it's probably better to use our capitalize function instead of titleize, because it has less unintended side effects. For example:

```
>>> word = "two PV panels"
>>> titleize(word)
"Two Pv Panels"
>>> capitalize(word)
"Two PV panels"
```

flexmeasures.utils.geo_utils

Functions

`flexmeasures.utils.geo_utils.cos_rad_lat(latitude: float) → float`

`flexmeasures.utils.geo_utils.earth_distance(location: tuple[float, float], other_location: tuple[float, float]) → float`

Great circle distance in km between two locations on Earth.

`flexmeasures.utils.geo_utils.parse_lat_lng(kwargs) → tuple[float, float] | tuple[None, None]`

Parses latitude and longitude values stated in kwargs.

Can be called with an object that has latitude and longitude properties, for example:

```
lat, lng = parse_lat_lng(object=asset)
```

Can also be called with latitude and longitude parameters, for example:

```
lat, lng = parse_lat_lng(latitude=32, longitude=54) lat, lng = parse_lat_lng(lat=32, lng=54)
```

`flexmeasures.utils.geo_utils.rad_lng(longitude: float) → float`

`flexmeasures.utils.geo_utils.sin_rad_lat(latitude: float) → float`

flexmeasures.utils.grid_cells

Functions

`flexmeasures.utils.grid_cells.get_cell_nums`(*tl: tuple[float, float], br: tuple[float, float], num_cells: int = 9*) → `tuple[int, int]`

Compute the number of cells in both directions, latitude and longitude. By default, a square grid with N=9 cells is computed, so 3 by 3. For N with non-integer square root, the function will determine a nice cell pattern. :param tl: top-left (lat, lng) tuple of ROI :param br: bottom-right (lat, lng) tuple of ROI :param num_cells: number of cells (9 by default, leading to a 3x3 grid)

Classes

`class flexmeasures.utils.grid_cells.LatLngGrid`(*top_left: tuple[float, float], bottom_right: tuple[float, float], num_cells_lat: int, num_cells_lng: int*)

Represents a grid in latitude and longitude notation for some rectangular region of interest (ROI). The specs are a top-left and a bottom-right coordinate, as well as the number of cells in both directions. The class provides two ways of conceptualising cells which nicely cover the grid: square cells and hexagonal cells. For both, locations can be computed which represent the corners of said cells. Examples:

- 4 cells in square: 9 unique locations in a 2x2 grid (4*4 locations, of which 7 are covered by another cell)
- 4 cells in hex: 13 unique locations in a 2x2 grid (4*6 locations, of which 11 are already covered)
- 10 cells in square: 18 unique locations in a 5x2 grid (10*4 locations, of which 11 are already covered)
- 10 cells in hex: 34 unique locations in a 5x2 grid (10*6 locations, of which 26 are already covered)

The top-right and bottom-left locations are always at the center of a cell, unless the grid has 1 row or 1 column. In those case, these locations are closer to one side of the cell.

`__init__`(*top_left: tuple[float, float], bottom_right: tuple[float, float], num_cells_lat: int, num_cells_lng: int*)

`compute_cell_size_lat`() → `float`

Calculate the step size between latitudes

`compute_cell_size_lng`() → `float`

Calculate the step size between longitudes

`get_locations`(*method: str*) → `list[tuple[float, float]]`

Get locations by method (“square” or “hex”)

`locations_hex`() → `list[tuple[float, float]]`

The hexagonal pattern - actually leaves out one cell for every even row.

`locations_square`() → `list[tuple[float, float]]`

square pattern

flexmeasures.utils.plugin_utils

Utils for registering FlexMeasures plugins

Functions

`flexmeasures.utils.plugin_utils.check_config_settings(app, settings: dict[str, dict])`

Make sure expected config settings exist.

For example:

```
settings = {
    "MY_PLUGIN_URL": {
        "description": "URL used by my plugin for x.", "level": "error",
    }, "MY_PLUGIN_TOKEN": {
        "description": "Token used by my plugin for y.", "level": "warning", "message": "With-
        out this token, my plugin will not do y.", "parse_as": str,
    }, "MY_PLUGIN_COLOR": {
        "description": "Color used to override the default plugin color.", "level": "info",
    },
}
```

`flexmeasures.utils.plugin_utils.log_missing_config_setting(app, setting_name: str, setting_fields: dict)`

Log a message for this missing config setting.

The logging level is taken from the 'level' key. If missing, we default to error. If present, we also log the 'description' and the 'message_if_missing' keys.

`flexmeasures.utils.plugin_utils.log_wrong_type_for_config_setting(app, setting_name: str, setting_fields: dict, setting_type: type)`

Log a message for this config setting that has the wrong type.

`flexmeasures.utils.plugin_utils.register_plugins(app: Flask)`

Register FlexMeasures plugins as Blueprints. This is configured by the config setting FLEXMEASURES_PLUGINS.

Assumptions: - a setting EITHER points to a plugin folder containing an `__init__.py` file

OR it is the name of an installed module, which can be imported.

- each plugin defines at least one Blueprint object. These will be registered with the Flask app, so their functionality (e.g. routes) becomes available.

If you load a plugin via a file path, we'll refer to the plugin with the name of your plugin folder (last part of the path).

flexmeasures.utils.time_utils

Utils for dealing with time

Functions

`flexmeasures.utils.time_utils.apply_offset_chain(dt: pd.Timestamp | datetime, offset_chain: str) → pd.Timestamp | datetime`

Apply an offset chain to a date.

An offset chain consist of multiple (pandas) offset strings separated by commas. Moreover, this function implements the offset string “DB”, which stands for Day Begin, to get a date from a datetime, i.e. removing time details finer than a day.

Args:

`dt` (*pd.Timestamp* | *datetime*) `offset_chain` (*str*)

Returns:

pd.Timestamp | *datetime* (same type as given `dt`)

`flexmeasures.utils.time_utils.as_server_time(dt: datetime) → datetime`

The datetime represented in the timezone of the FlexMeasures platform. If `dt` is naive, we assume it is UTC time.

`flexmeasures.utils.time_utils.decide_resolution(start: datetime | None, end: datetime | None) → str`

Decide on a practical resolution given the length of the selected time period. Useful for querying or plotting.

`flexmeasures.utils.time_utils.determine_minimum_resampling_resolution(event_resolutions: list[datetime.timedelta]) → timedelta`

Return minimum non-zero event resolution, or zero resolution if none of the event resolutions is non-zero.

`flexmeasures.utils.time_utils.duration_isoformat(duration: timedelta)`

Adapted version of `isodate.duration_isoformat` for formatting a *datetime.timedelta*.

The difference is that absolute days are not formatted as nominal days. Workaround for <https://github.com/gweis/isodate/issues/74>.

`flexmeasures.utils.time_utils.ensure_local_timezone(dt: pd.Timestamp | datetime, tz_name: str = 'Europe/Amsterdam') → pd.Timestamp | datetime`

If no timezone is given, assume the datetime is in the given timezone and make it explicit. Otherwise, if a timezone is given, convert to that timezone.

`flexmeasures.utils.time_utils.forecast_horizons_for(resolution: str | timedelta) → list[str] | list[timedelta]`

Return a list of horizons that are supported per resolution. Return values of the same type as the input.

`flexmeasures.utils.time_utils.freq_label_to_human_readable_label(freq_label: str) → str`

Translate pandas frequency labels to human-readable labels.

`flexmeasures.utils.time_utils.get_default_end_time() → datetime`

`flexmeasures.utils.time_utils.get_default_start_time() → datetime`

`flexmeasures.utils.time_utils.get_first_day_of_next_month() → datetime`

`flexmeasures.utils.time_utils.get_max_planning_horizon(resolution: timedelta) → timedelta | None`

Determine the maximum planning horizon for the given sensor resolution.

`flexmeasures.utils.time_utils.get_most_recent_clocktime_window(window_size_in_minutes: int, now: datetime | None = None, grace_period_in_seconds: int | None = 0) → tuple[datetime, datetime]`

Calculate a recent time window, returning a start and end minute so that a full hour can be filled with such windows, e.g.:

Calling this function at 15:01:xx with window size 5 -> (14:55:00, 15:00:00) Calling this function at 03:36:xx with window size 15 -> (03:15:00, 03:30:00)

We can demand a grace period (of x seconds) to have passed before we are ready to accept that we're in a new window: Calling this function at 15:00:16 with window size 5 and grace period of 30 seconds -> (14:50:00, 14:55:00)

window_size_in_minutes is assumed to > 0 and <= 60, and a divisor of 60 (1, 2, ..., 30, 60).

If now is not given, the current server time is used. if now / the current time lies within a boundary minute (e.g. 15 when window_size_in_minutes=5), then the window is not deemed over and the previous one is returned (in this case, [5, 10])

Returns two datetime objects. They'll be in the timezone (if given) of the now parameter, or in the server timezone (see FLEXMEASURES_TIMEZONE setting).

`flexmeasures.utils.time_utils.get_most_recent_hour() → datetime`

`flexmeasures.utils.time_utils.get_most_recent_quarter() → datetime`

`flexmeasures.utils.time_utils.get_timezone(of_user=False) → BaseTzInfo`

Return the FlexMeasures timezone, or if desired try to return the timezone of the current user.

`flexmeasures.utils.time_utils.localized_datetime(dt: datetime) → datetime`

Localise a datetime to the timezone of the FlexMeasures platform. Note: this will change nothing but the tzinfo field.

`flexmeasures.utils.time_utils.localized_datetime_str(dt: datetime, dt_format: str = '%Y-%m-%d %I:%M %p') → str`

Localise a datetime to the timezone of the FlexMeasures platform. If no datetime is passed in, use server_now() as basis.

Hint: This can be set as a jinja filter, so we can display local time in the app, e.g.: `app.jinja_env.filters['localized_datetime'] = localized_datetime_str`

`flexmeasures.utils.time_utils.naive_utc_from(dt: datetime) → datetime`

Return a naive datetime, that is localised to UTC if it has a timezone. If dt is naive, we assume it is already in UTC time.

`flexmeasures.utils.time_utils.naturalized_datetime_str(dt: datetime | None, now: datetime | None = None) → str`

Naturalise a datetime object (into a human-friendly string). The dt parameter (as well as the now parameter if you use it) can be either naive or tz-aware. We assume UTC in the naive case.

We use the the humanize library to generate a human-friendly string. If dt is not longer ago than 24 hours, we use humanize.naturaltime (e.g. "3 hours ago"), otherwise humanize.naturaldate (e.g. "one week ago")

Hint: This can be set as a jinja filter, so we can display local time in the app, e.g.: `app.jinja_env.filters['naturalized_datetime'] = naturalized_datetime_str`

`flexmeasures.utils.time_utils.resolution_to_hour_factor(resolution: str | timedelta) → float`

Return the factor with which a value needs to be multiplied in order to get the value per hour, e.g. 10 MW at a resolution of 15min are 2.5 MWh per time step.

Parameters

resolution – *timedelta* or pandas offset such as “15T” or “1H”

`flexmeasures.utils.time_utils.round_to_closest_hour(dt: datetime) → datetime`

`flexmeasures.utils.time_utils.round_to_closest_quarter(dt: datetime) → datetime`

`flexmeasures.utils.time_utils.server_now() → datetime`

The current time (timezone aware), converted to the timezone of the FlexMeasures platform.

`flexmeasures.utils.time_utils.supported_horizons() → list[datetime.timedelta]`

`flexmeasures.utils.time_utils.timedelta_to_pandas_freq_str(resolution: timedelta) → str`

`flexmeasures.utils.time_utils.to_http_time(dt: pd.Timestamp | datetime) → str`

Formats *datetime* using the Internet Message Format fixdate.

```
>>> to_http_time(pd.Timestamp("2022-12-13 14:06:23Z"))
Tue, 13 Dec 2022 14:06:23 GMT
```

References

IMF-fixdate: <https://www.rfc-editor.org/rfc/rfc7231#section-7.1.1.1>

`flexmeasures.utils.time_utils.tz_index_naively(data: pd.DataFrame | pd.Series | pd.DatetimeIndex) → pd.DataFrame | pd.Series | pd.DatetimeIndex`

Turn any *DatetimeIndex* into a tz-naive one, then return. Useful for bokeh, for instance.

flexmeasures.utils.unit_utils

Utility module for unit conversion

FlexMeasures stores units as strings in short scientific notation (such as ‘kWh’ to denote kilowatt-hour). We use the pint library to convert data between compatible units (such as ‘m/s’ to ‘km/h’). Three-letter currency codes (such as ‘KRW’ to denote South Korean Won) are valid units. Note that converting between currencies requires setting up a sensor that registers conversion rates over time. The preferred compact form for combinations of units can be derived automatically (such as ‘kW*EUR/MWh’ to ‘EUR/h’). Time series with fixed resolution can be converted from units of flow to units of stock (such as ‘kW’ to ‘kWh’), and vice versa. Percentages can be converted to units of some physical capacity if a capacity is known (such as ‘%’ to ‘kWh’).

Functions

`flexmeasures.utils.unit_utils.convert_units(data: tb.BeliefsSeries | pd.Series | list[int | float] | int | float, from_unit: str, to_unit: str, event_resolution: timedelta | None = None, capacity: str | None = None) → pd.Series | list[int | float] | int | float`

Updates data values to reflect the given unit conversion.

```
flexmeasures.utils.unit_utils.determine_flow_unit(stock_unit: str, time_unit: str = 'h')
```

For example: >>> determine_flow_unit("m³") # m³/h >>> determine_flow_unit("kWh") # kW

```
flexmeasures.utils.unit_utils.determine_stock_unit(flow_unit: str, time_unit: str = 'h')
```

Determine the shortest unit of stock, given a unit of flow.

For example: >>> determine_stock_unit("m³/h") # m³ >>> determine_stock_unit("kW") # kWh

```
flexmeasures.utils.unit_utils.determine_unit_conversion_multiplier(from_unit: str, to_unit: str,
                                                                    duration: timedelta | None =
                                                                    None)
```

Determine the value multiplier for a given unit conversion. If needed, requires a duration to convert from units of stock change to units of flow, or vice versa.

```
flexmeasures.utils.unit_utils.is_energy_price_unit(unit: str) → bool
```

For example: >>> is_energy_price_unit("EUR/MWh") True >>> is_energy_price_unit("KRW/MWh") True
>>> is_energy_price_unit("KRW/MW") False >>> is_energy_price_unit("beans/MW") False

```
flexmeasures.utils.unit_utils.is_energy_unit(unit: str) → bool
```

For example: >>> is_energy_unit("kW") False >>> is_energy_unit("°C") False >>> is_energy_unit("kWh") True
>>> is_energy_unit("EUR/MWh") False

```
flexmeasures.utils.unit_utils.is_power_unit(unit: str) → bool
```

For example: >>> is_power_unit("kW") True >>> is_power_unit("°C") False >>> is_power_unit("kWh") False
>>> is_power_unit("EUR/MWh") False

```
flexmeasures.utils.unit_utils.is_valid_unit(unit: str) → bool
```

Return True if the pint library can work with this unit identifier.

```
flexmeasures.utils.unit_utils.to_preferred(x: Quantity) → Quantity
```

From <https://github.com/hgrecco/pint/issues/676#issuecomment-689157693>

```
flexmeasures.utils.unit_utils.units_are_convertible(from_unit: str, to_unit: str, duration_known:
                                                    bool = True) → bool
```

For example, a sensor with W units allows data to be posted with units: >>> units_are_convertible("kW", "W")
True (units just have different prefixes) >>> units_are_convertible("J/s", "W") # True (units can be converted
using some multiplier) >>> units_are_convertible("Wh", "W") # True (units that represent a stock delta can,
knowing the duration, be converted to a flow) >>> units_are_convertible("°C", "W") # False

Utilities for the FlexMeasures project.

PYTHON MODULE INDEX

f

- `flexmeasures.api`, 200
- `flexmeasures.api.common`, 181
- `flexmeasures.api.common.implementations`, 162
- `flexmeasures.api.common.responses`, 162
- `flexmeasures.api.common.routes`, 164
- `flexmeasures.api.common.schemas`, 170
- `flexmeasures.api.common.schemas.generic_assets`, 164
- `flexmeasures.api.common.schemas.sensor_data`, 165
- `flexmeasures.api.common.schemas.sensors`, 168
- `flexmeasures.api.common.schemas.users`, 169
- `flexmeasures.api.common.utils`, 181
- `flexmeasures.api.common.utils.api_utils`, 171
- `flexmeasures.api.common.utils.args_parsing`, 173
- `flexmeasures.api.common.utils.decorators`, 174
- `flexmeasures.api.common.utils.deprecation_utils`, 174
- `flexmeasures.api.common.utils.migration_utils`, 176
- `flexmeasures.api.common.utils.validators`, 177
- `flexmeasures.api.dev`, 182
- `flexmeasures.api.dev.sensors`, 181
- `flexmeasures.api.play`, 184
- `flexmeasures.api.play.implementations`, 183
- `flexmeasures.api.play.routes`, 183
- `flexmeasures.api.sunset`, 184
- `flexmeasures.api.sunset.routes`, 184
- `flexmeasures.api.v3_0`, 200
- `flexmeasures.api.v3_0.accounts`, 185
- `flexmeasures.api.v3_0.assets`, 187
- `flexmeasures.api.v3_0.health`, 191
- `flexmeasures.api.v3_0.public`, 191
- `flexmeasures.api.v3_0.sensors`, 191
- `flexmeasures.api.v3_0.users`, 197
- `flexmeasures.app`, 201
- `flexmeasures.auth`, 204
- `flexmeasures.auth.decorators`, 201
- `flexmeasures.auth.error_handling`, 203
- `flexmeasures.auth.policy`, 203
- `flexmeasures.cli`, 208
- `flexmeasures.cli.data_add`, 205
- `flexmeasures.cli.data_delete`, 205
- `flexmeasures.cli.data_edit`, 205
- `flexmeasures.cli.data_show`, 205
- `flexmeasures.cli.db_ops`, 206
- `flexmeasures.cli.jobs`, 206
- `flexmeasures.cli.monitor`, 206
- `flexmeasures.cli.utils`, 206
- `flexmeasures.data`, 286
- `flexmeasures.data.config`, 209
- `flexmeasures.data.models`, 249
- `flexmeasures.data.models.annotations`, 210
- `flexmeasures.data.models.assets`, 212
- `flexmeasures.data.models.charts`, 215
- `flexmeasures.data.models.charts.belief_charts`, 214
- `flexmeasures.data.models.charts.defaults`, 215
- `flexmeasures.data.models.data_sources`, 215
- `flexmeasures.data.models.forecasting`, 221
- `flexmeasures.data.models.forecasting.exceptions`, 216
- `flexmeasures.data.models.forecasting.model_spec_factory`, 216
- `flexmeasures.data.models.forecasting.model_specs`, 219
- `flexmeasures.data.models.forecasting.model_specs.linear_regression`, 219
- `flexmeasures.data.models.forecasting.model_specs.naive`, 219
- `flexmeasures.data.models.forecasting.utils`, 220
- `flexmeasures.data.models.generic_assets`, 221
- `flexmeasures.data.models.legacy_migration_utils`, 225
- `flexmeasures.data.models.markets`, 226
- `flexmeasures.data.models.parsing_utils`, 227
- `flexmeasures.data.models.planning`, 235
- `flexmeasures.data.models.planning.battery`, 228
- `flexmeasures.data.models.planning.charging_station`, 228

`flexmeasures.data.models.planning.exceptions`, 228
`flexmeasures.data.models.planning.linear_optimization`, 229
`flexmeasures.data.models.planning.storage`, 230
`flexmeasures.data.models.planning.utils`, 234
`flexmeasures.data.models.reporting`, 238
`flexmeasures.data.models.reporting.aggregator`, 236
`flexmeasures.data.models.reporting.pandas_reporter`, 237
`flexmeasures.data.models.task_runs`, 239
`flexmeasures.data.models.time_series`, 239
`flexmeasures.data.models.user`, 245
`flexmeasures.data.models.validation_utils`, 247
`flexmeasures.data.models.weather`, 248
`flexmeasures.data.queries`, 256
`flexmeasures.data.queries.analytics`, 250
`flexmeasures.data.queries.annotations`, 251
`flexmeasures.data.queries.data_sources`, 251
`flexmeasures.data.queries.generic_assets`, 251
`flexmeasures.data.queries.portfolio`, 252
`flexmeasures.data.queries.sensors`, 253
`flexmeasures.data.queries.utils`, 254
`flexmeasures.data.schemas`, 268
`flexmeasures.data.schemas.account`, 257
`flexmeasures.data.schemas.assets`, 257
`flexmeasures.data.schemas.generic_assets`, 258
`flexmeasures.data.schemas.reporting`, 263
`flexmeasures.data.schemas.reporting.aggregation`, 260
`flexmeasures.data.schemas.reporting.pandas_reporter`, 261
`flexmeasures.data.schemas.scheduling`, 264
`flexmeasures.data.schemas.scheduling.storage`, 264
`flexmeasures.data.schemas.sensors`, 265
`flexmeasures.data.schemas.sources`, 265
`flexmeasures.data.schemas.times`, 266
`flexmeasures.data.schemas.units`, 266
`flexmeasures.data.schemas.users`, 267
`flexmeasures.data.schemas.utils`, 267
`flexmeasures.data.scripts`, 269
`flexmeasures.data.scripts.data_gen`, 269
`flexmeasures.data.scripts.visualize_data_model`, 269
`flexmeasures.data.services`, 284
`flexmeasures.data.services.accounts`, 270
`flexmeasures.data.services.annotations`, 270
`flexmeasures.data.services.asset_grouping`, 271
`flexmeasures.data.services.data_sources`, 272
`flexmeasures.data.services.forecasting`, 273
`flexmeasures.data.services.resources`, 274
`flexmeasures.data.services.scheduling`, 279
`flexmeasures.data.services.sensors`, 280
`flexmeasures.data.services.time_series`, 281
`flexmeasures.data.services.timerange`, 282
`flexmeasures.data.services.users`, 283
`flexmeasures.data.services.utils`, 284
`flexmeasures.data.transactional`, 285
`flexmeasures.data.utils`, 285
`flexmeasures.ui`, 293
`flexmeasures.ui.crud`, 289
`flexmeasures.ui.crud.accounts`, 287
`flexmeasures.ui.crud.api_wrapper`, 287
`flexmeasures.ui.crud.assets`, 288
`flexmeasures.ui.crud.users`, 289
`flexmeasures.ui.error_handlers`, 290
`flexmeasures.ui.utils`, 291
`flexmeasures.ui.utils.chart_defaults`, 290
`flexmeasures.ui.utils.view_utils`, 290
`flexmeasures.ui.views`, 293
`flexmeasures.ui.views.control`, 292
`flexmeasures.ui.views.logged_in_user`, 292
`flexmeasures.ui.views.new_dashboard`, 292
`flexmeasures.ui.views.sensors`, 293
`flexmeasures.utils`, 307
`flexmeasures.utils.app_utils`, 294
`flexmeasures.utils.calculations`, 295
`flexmeasures.utils.coding_utils`, 296
`flexmeasures.utils.config_defaults`, 298
`flexmeasures.utils.config_utils`, 298
`flexmeasures.utils.entity_address_utils`, 299
`flexmeasures.utils.error_utils`, 300
`flexmeasures.utils.flexmeasures_inflection`, 301
`flexmeasures.utils.geo_utils`, 301
`flexmeasures.utils.grid_cells`, 302
`flexmeasures.utils.plugin_utils`, 303
`flexmeasures.utils.time_utils`, 304
`flexmeasures.utils.unit_utils`, 306

HTTP ROUTING TABLE

/api

GET /api/, 85
GET /api/dev/asset/(id)/, 98
GET /api/dev/sensor/(id)/, 98
GET /api/dev/sensor/(id)/chart/, 98
GET /api/dev/sensor/(id)/chart_annotations/,
98
GET /api/dev/sensor/(id)/chart_data/, 98
GET /api/v3_0, 85
GET /api/v3_0/assets, 85
GET /api/v3_0/assets/(id), 87
GET /api/v3_0/assets/(id)/chart/, 89
GET /api/v3_0/assets/(id)/chart_data/, 89
GET /api/v3_0/assets/public, 89
GET /api/v3_0/health/ready, 89
GET /api/v3_0/sensors, 89
GET /api/v3_0/sensors/(id)/schedules/(uuid),
90
GET /api/v3_0/sensors/data, 93
GET /api/v3_0/users, 94
GET /api/v3_0/users/(id), 95
POST /api/requestAuthToken, 85
POST /api/v3_0/assets, 86
POST /api/v3_0/sensors/(id)/schedules/trigger,
91
POST /api/v3_0/sensors/data, 94
DELETE /api/v3_0/assets/(id), 87
PATCH /api/v3_0/assets/(id), 88
PATCH /api/v3_0/users/(id), 96
PATCH /api/v3_0/users/(id)/password-reset, 97

INDEX

Symbols

`__init__()` (*flexmeasures.api.common.responses.BaseMessage* method), 164
`__init__()` (*flexmeasures.api.common.schemas.sensors.SensorField* method), 168
`__init__()` (*flexmeasures.api.common.schemas.users.UserField* method), 170
`__init__()` (*flexmeasures.cli.utils.DeprecatedDefaultGroup* method), 207
`__init__()` (*flexmeasures.data.models.annotations.AccountAnnotationRelationship* method), 211
`__init__()` (*flexmeasures.data.models.annotations.Annotation* method), 211
`__init__()` (*flexmeasures.data.models.annotations.GenericAssetAnnotationRelationship* method), 212
`__init__()` (*flexmeasures.data.models.annotations.SensorAnnotationRelationship* method), 212
`__init__()` (*flexmeasures.data.models.assets.Asset* method), 212
`__init__()` (*flexmeasures.data.models.assets.AssetType* method), 213
`__init__()` (*flexmeasures.data.models.assets.Power* method), 213
`__init__()` (*flexmeasures.data.models.data_sources.DataSource* method), 215
`__init__()` (*flexmeasures.data.models.forecasting.model_spec_factory.LBSeriesSpec* method), 218
`__init__()` (*flexmeasures.data.models.forecasting.model_spec_factory.Naive* method), 219
`__init__()` (*flexmeasures.data.models.generic_assets.GenericAsset* method), 222
`__init__()` (*flexmeasures.data.models.generic_assets.GenericAssetType* method), 225
`__init__()` (*flexmeasures.data.models.markets.Market* method), 226
`__init__()` (*flexmeasures.data.models.markets.MarketType* method), 227
`__init__()` (*flexmeasures.data.models.markets.Price* method), 227
`__init__()` (*flexmeasures.data.models.planning.Scheduler* method), 235
`__init__()` (*flexmeasures.data.models.reporting.Reporter* method), 238
`__init__()` (*flexmeasures.data.models.task_runs.LatestTaskRun* method), 239
`__init__()` (*flexmeasures.data.models.time_series.Sensor* method), 239
`__init__()` (*flexmeasures.data.models.time_series.TimedBelief* method), 242
`__init__()` (*flexmeasures.data.models.user.Account* method), 245
`__init__()` (*flexmeasures.data.models.user.AccountRole* method), 246
`__init__()` (*flexmeasures.data.models.user.Role* method), 246
`__init__()` (*flexmeasures.data.models.user.RolesAccounts* method), 246
`__init__()` (*flexmeasures.data.models.user.RolesUsers* method), 246
`__init__()` (*flexmeasures.data.models.user.User* method), 246
`__init__()` (*flexmeasures.data.models.weather.Weather* method), 248
`__init__()` (*flexmeasures.data.models.weather.WeatherSensor* method), 248
`__init__()` (*flexmeasures.data.models.weather.WeatherSensorType* method), 249
`__init__()` (*flexmeasures.data.schemas.assets.LatitudeField* method), 257
`__init__()` (*flexmeasures.data.schemas.assets.LatitudeLongitudeValidator* method), 257
`__init__()` (*flexmeasures.data.schemas.assets.LatitudeValidator* method), 258
`__init__()` (*flexmeasures.data.schemas.assets.LongitudeField* method), 258
`__init__()` (*flexmeasures.data.schemas.assets.LongitudeValidator* method), 258
`__init__()` (*flexmeasures.data.schemas.scheduling.storage.EfficiencyField* method), 264
`__init__()` (*flexmeasures.data.schemas.scheduling.storage.SOCValueSchedule* method), 264
`__init__()` (*flexmeasures.data.schemas.scheduling.storage.StorageFlexModel* method), 264
`__init__()` (*flexmeasures.data.schemas.units.QuantityField* method), 264

<code>method</code>), 267	<code>sures.data.schemas.times.AwareDateTimeField</code>
<code>__init__()</code> (<code>flexmeasures.data.schemas.units.QuantityValidator</code>	<code>method</code>), 266
<code>method</code>), 267	<code>_deserialize()</code> (<code>flexmea-</code>
<code>__init__()</code> (<code>flexmeasures.data.schemas.utils.MarshmallowClickMixin</code>	<code>sures.data.schemas.times.DurationField</code>
<code>method</code>), 268	<code>method</code>), 266
<code>__init__()</code> (<code>flexmeasures.data.services.asset_grouping.AssetGrouping</code>	<code>_deserialize()</code> (<code>flexmea-</code>
<code>method</code>), 272	<code>sures.data.schemas.units.QuantityField</code>
<code>__init__()</code> (<code>flexmeasures.data.services.resources.Resource</code>	<code>method</code>), 267
<code>method</code>), 277	<code>_load_series()</code> (<code>flexmea-</code>
<code>__init__()</code> (<code>flexmeasures.utils.grid_cells.LatLngGrid</code>	<code>sures.data.models.forecasting.model_spec_factory.TBSeriesSpecs</code>
<code>method</code>), 302	<code>method</code>), 219
<code>_apply_transformations()</code> (<code>flexmea-</code>	<code>_maybe_raise()</code> (<code>flexmea-</code>
<code>sures.data.models.reporting.pandas_reporter.PandasReporter</code>	<code>sures.ui.crud.api_wrapper.InternalApi</code>
<code>method</code>), 237	<code>method</code>), 287
<code>_compute()</code> (<code>flexmeasures.data.models.reporting.Reporter</code>	<code>_process_pandas_args()</code> (<code>flexmea-</code>
<code>method</code>), 238	<code>sures.data.models.reporting.pandas_reporter.PandasReporter</code>
<code>_compute()</code> (<code>flexmeasures.data.models.reporting.aggregator.Aggregator</code>	<code>method</code>), 237
<code>method</code>), 236	<code>_process_pandas_kwargs()</code> (<code>flexmea-</code>
<code>_compute()</code> (<code>flexmeasures.data.models.reporting.pandas_reporter.PandasReporter</code>	<code>sures.data.models.reporting.pandas_reporter.PandasReporter</code>
<code>method</code>), 237	<code>method</code>), 237
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.generic_assets.AssetIdField</code>	<code>sures.api.common.schemas.generic_assets.AssetIdField</code>
<code>method</code>), 164	<code>method</code>), 165
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.sensor_data.SingleValueField</code>	<code>sures.api.common.schemas.sensor_data.SingleValueField</code>
<code>method</code>), 167	<code>method</code>), 167
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.sensors.SensorField</code>	<code>sures.api.common.schemas.sensors.SensorField</code>
<code>method</code>), 168	<code>method</code>), 168
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.sensors.SensorIdField</code>	<code>sures.api.common.schemas.sensors.SensorIdField</code>
<code>method</code>), 168	<code>method</code>), 168
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.users.AccountIdField</code>	<code>sures.api.common.schemas.users.AccountIdField</code>
<code>method</code>), 169	<code>method</code>), 169
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.api.common.schemas.users.UserIdField</code>	<code>sures.api.common.schemas.users.UserIdField</code>
<code>method</code>), 170	<code>method</code>), 170
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.data.schemas.account.AccountIdField</code>	<code>sures.data.schemas.account.AccountIdField</code>
<code>method</code>), 257	<code>method</code>), 257
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.data.schemas.generic_assets.GenericAssetIdField</code>	<code>sures.data.schemas.generic_assets.GenericAssetIdField</code>
<code>method</code>), 258	<code>method</code>), 258
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.data.schemas.generic_assets.JSON</code>	<code>sures.data.schemas.generic_assets.JSON</code>
<code>method</code>), 259	<code>method</code>), 259
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.data.schemas.sensors.SensorIdField</code>	<code>sures.data.schemas.sensors.SensorIdField</code>
<code>method</code>), 265	<code>method</code>), 265
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>
<code>sures.data.schemas.sources.DataSourceIdField</code>	<code>sures.data.schemas.sources.DataSourceIdField</code>
<code>method</code>), 265	<code>method</code>), 265
<code>_deserialize()</code> (<code>flexmea-</code>	<code>_serialize()</code> (<code>flexmea-</code>

asures.data.schemas.times.DurationField
method), 266

_serialize() (flexmeasures.data.schemas.units.QuantityField
method), 267

A

Account (class in flexmeasures.data.models.user), 245

account_roles_accepted() (in module flexmeasures.auth.decorators), 202

account_roles_required() (in module flexmeasures.auth.decorators), 202

AccountAnnotationRelationship (class in flexmeasures.data.models.annotations), 211

AccountAPI (class in flexmeasures.api.v3_0.accounts), 185

AccountCrudUI (class in flexmeasures.ui.crud.accounts), 287

AccountIdField (class in flexmeasures.api.common.schemas.users), 169

AccountIdField (class in flexmeasures.data.schemas.account), 257

accountname() (in module flexmeasures.ui.utils.view_utils), 290

AccountRole (class in flexmeasures.data.models.user), 246

AccountRoleSchema (class in flexmeasures.data.schemas.account), 257

AccountRoleSchema.Meta (class in flexmeasures.data.schemas.account), 257

AccountSchema (class in flexmeasures.data.schemas.account), 257

AccountSchema.Meta (class in flexmeasures.data.schemas.account), 257

add() (flexmeasures.data.models.annotations.Annotation class method), 211

add() (flexmeasures.data.models.time_series.TimedBelief class method), 242

add_annotations() (flexmeasures.data.models.generic_assets.GenericAsset method), 222

add_basic_error_handlers() (in module flexmeasures.utils.error_utils), 300

add_default_account_roles() (in module flexmeasures.data.scripts.data_gen), 269

add_default_asset_types() (in module flexmeasures.data.scripts.data_gen), 269

add_default_data_sources() (in module flexmeasures.data.scripts.data_gen), 269

add_default_user_roles() (in module flexmeasures.data.scripts.data_gen), 269

add_html_error_views() (in module flexmeasures.ui.error_handlers), 290

add_jinja_filters() (in module flexmeasures.ui), 293

add_jinja_variables() (in module flexmeasures.ui), 293

add_storage_constraints() (in module flexmeasures.data.models.planning.storage), 230

add_tiny_price_slope() (in module flexmeasures.data.models.planning.utils), 234

add_transmission_zone_asset() (in module flexmeasures.data.scripts.data_gen), 269

after_request_exception_rollback_session() (in module flexmeasures.data.transactional), 285

aggregate_cost (flexmeasures.data.services.resources.Resource property), 277

aggregate_demand (flexmeasures.data.services.resources.Resource property), 277

aggregate_profit_or_loss (flexmeasures.data.services.resources.Resource property), 277

aggregate_revenue (flexmeasures.data.services.resources.Resource property), 277

aggregate_supply (flexmeasures.data.services.resources.Resource property), 277

aggregate_values() (in module flexmeasures.data.services.time_series), 281

AggregatorReporter (class in flexmeasures.data.models.reporting.aggregator), 236

AggregatorSchema (class in flexmeasures.data.schemas.reporting.aggregation), 260

already_received_and_successfully_processed() (in module flexmeasures.api.common.responses), 162

Annotation (class in flexmeasures.data.models.annotations), 211

append_doc_of() (in module flexmeasures.api.common.utils.api_utils), 171

apply_chart_defaults() (in module flexmeasures.data.models.charts.defaults), 215

apply_offset_chain() (in module flexmeasures.utils.time_utils), 304

apply_stock_changes_and_losses() (in module flexmeasures.utils.calculations), 295

are_required_settings_complete() (in module flexmeasures.utils.config_utils), 298

as_response_type() (in module flexmeasures.api.common.utils.decorators), 174

as_server_time() (in module flexmeasures.utils.time_utils), 304

as_transaction() (in module flexmeasures.data.models.transactional), 285

asures.data.transactional), 285

Asset (class in *flexmeasures.data.models.assets*), 212

asset_icon_name() (in module *flexmeasures.ui.utils.view_utils*), 290

asset_replace_name_with_id() (in module *flexmeasures.api.common.utils.api_utils*), 171

asset_type (*flexmeasures.data.models.generic_assets.GenericAsset* property), 222

AssetAPI (class in *flexmeasures.api.dev.sensors*), 182

AssetAPI (class in *flexmeasures.api.v3_0.assets*), 187

AssetCrudUI (class in *flexmeasures.ui.crud.assets*), 288

AssetForm (class in *flexmeasures.ui.crud.assets*), 288

AssetGroup (class in *flexmeasures.data.services.asset_grouping*), 272

AssetIdField (class in *flexmeasures.api.common.schemas.generic_assets*), 164

assets_required() (in module *flexmeasures.api.common.utils.validators*), 177

assets_share_location() (in module *flexmeasures.data.models.assets*), 212

assets_share_location() (in module *flexmeasures.data.models.generic_assets*), 221

AssetSchema (class in *flexmeasures.data.schemas.assets*), 257

AssetSchema.Meta (class in *flexmeasures.data.schemas.assets*), 257

AssetType (class in *flexmeasures.data.models.assets*), 213

AuthModelMixin (class in *flexmeasures.auth.policy*), 204

AwareDateTimeField (class in *flexmeasures.data.schemas.times*), 266

B

bar_chart() (in module *flexmeasures.data.models.charts.belief_charts*), 214

BaseMessage (class in *flexmeasures.api.common.responses*), 164

belief_horizon (*flexmeasures.data.models.time_series.TimedBelief* attribute), 243

BeliefsSearchConfigSchema (class in *flexmeasures.data.schemas.reporting*), 263

build_device_soc_targets() (in module *flexmeasures.data.models.planning.storage*), 230

build_device_soc_values() (in module *flexmeasures.data.models.planning.storage*), 230

build_ea_scheme_and_naming_authority() (in module *flexmeasures.utils.entity_address_utils*), 299

build_entity_address() (in module *flexmeasures.utils.entity_address_utils*), 299

C

can_access_asset() (in module *flexmeasures.data.services.resources*), 275

capitalize() (in module *flexmeasures.utils.flexmeasures_inflection*), 301

catch_timed_belief_replacements() (in module *flexmeasures.api.common.utils.api_utils*), 171

chart() (*flexmeasures.data.models.generic_assets.GenericAsset* method), 222

chart() (*flexmeasures.data.models.time_series.Sensor* method), 239

chart_for_multiple_sensors() (in module *flexmeasures.data.models.charts.belief_charts*), 214

chart_type_to_chart_specs() (in module *flexmeasures.data.models.charts*), 215

check_access() (in module *flexmeasures.auth.policy*), 204

check_account_membership() (in module *flexmeasures.auth.policy*), 204

check_account_role() (in module *flexmeasures.auth.policy*), 204

check_app_env() (in module *flexmeasures.utils.config_utils*), 298

check_cache() (in module *flexmeasures.data.services.resources*), 275

check_config_settings() (in module *flexmeasures.utils.plugin_utils*), 303

check_data() (*flexmeasures.data.models.forecasting.model_spec_factory.TBSeriesSpecs* method), 219

check_data_availability() (in module *flexmeasures.data.models.forecasting.utils*), 220

check_errors() (in module *flexmeasures.cli.data_add*), 205

check_required_attributes() (*flexmeasures.data.models.time_series.Sensor* method), 240

check_required_attributes() (in module *flexmeasures.data.models.validation_utils*), 247

check_resolution_compatibility_of_sensor_data() (*flexmeasures.api.common.schemas.sensor_data.PostSensorData* method), 166

check_schema_unit_against_sensor_unit() (*flexmeasures.api.common.schemas.sensor_data.SensorDataDesc* method), 167

check_sqlalchemy_schemadisplay_installation() (in module *flexmeasures.data.scripts.visualize_data_model*), 269

check_timezone() (in module *flexmeasures.cli.data_add*), 205

check_user_identity() (in module *flexmeasures.auth.policy*), 204

check_user_role() (in module *flexmeasures*

- `...sures.auth.policy)`, 204
- `clear_session()` (in module `flexmeasures.ui.utils.view_utils`), 290
- `collect_time_series_data()` (in module `flexmeasures.data.services.time_series`), 281
- `commit_and_start_new_session()` (in module `flexmeasures.data.config`), 209
- `compute()` (`flexmeasures.data.models.planning.Scheduler` method), 235
- `compute()` (`flexmeasures.data.models.planning.storage.StorageScheduler` method), 233
- `compute()` (`flexmeasures.data.models.reporting.Reporter` method), 238
- `compute_cell_size_lat()` (`flexmeasures.utils.grid_cells.LatLngGrid` method), 302
- `compute_cell_size_lng()` (`flexmeasures.utils.grid_cells.LatLngGrid` method), 302
- `compute_schedule()` (`flexmeasures.data.models.planning.Scheduler` method), 236
- `compute_schedule()` (`flexmeasures.data.models.planning.storage.StorageScheduler` method), 233
- `Config` (class in `flexmeasures.utils.config_defaults`), 298
- `configure_db_for()` (in module `flexmeasures.data.config`), 209
- `configure_logging()` (in module `flexmeasures.utils.config_utils`), 298
- `configure_regressors_for_nearest_weather_sensor()` (in module `flexmeasures.data.models.forecasting.model_spec_factory`), 216
- `conflicting_resolutions()` (in module `flexmeasures.api.common.responses`), 162
- `contains_empty_items()` (in module `flexmeasures.api.common.utils.api_utils`), 171
- `control_view()` (in module `flexmeasures.ui.views.control`), 292
- `convert()` (`flexmeasures.data.schemas.utils.MarshmallowClickMixin` method), 268
- `convert_query_window_for_demo()` (in module `flexmeasures.data.services.time_series`), 281
- `convert_units()` (in module `flexmeasures.utils.unit_utils`), 306
- `copy_old_sensor_attributes()` (in module `flexmeasures.data.models.legacy_migration_utils`), 225
- `cos_rad_lat()` (in module `flexmeasures.utils.geo_utils`), 301
- `cost` (`flexmeasures.data.services.resources.Resource` property), 277
- `count_annotations()` (`flexmeasures.data.models.generic_assets.GenericAsset` method), 222
- `create()` (in module `flexmeasures.app`), 201
- `create_beliefs_query()` (in module `flexmeasures.data.queries.utils`), 254
- `create_circle_layer()` (in module `flexmeasures.data.models.charts.belief_charts`), 214
- `create_constraint_violations_message()` (in module `flexmeasures.data.models.planning.storage.StorageScheduler` method), 233
- `create_forecasting_jobs()` (in module `flexmeasures.data.services.forecasting`), 273
- `create_generic_asset()` (in module `flexmeasures.data.models.generic_assets`), 221
- `create_initial_model_specs()` (in module `flexmeasures.data.models.forecasting.model_spec_factory`), 217
- `create_lags()` (in module `flexmeasures.data.models.forecasting.utils`), 220
- `create_line_layer()` (in module `flexmeasures.data.models.charts.belief_charts`), 214
- `create_rect_layer()` (in module `flexmeasures.data.models.charts.belief_charts`), 214
- `create_scheduling_job()` (in module `flexmeasures.data.services.scheduling`), 279
- `create_schema_pic()` (in module `flexmeasures.data.scripts.visualize_data_model`), 269
- `createuml_pic()` (in module `flexmeasures.data.scripts.visualize_data_model`), 269
- `create_user()` (in module `flexmeasures.data.services.users`), 283
- `cumulative_probability` (`flexmeasures.data.models.time_series.TimedBelief` attribute), 243
- ## D
- `dashboard_view()` (in module `flexmeasures.ui.views.new_dashboard`), 292
- `data_source_id` (`flexmeasures.data.models.assets.Power` attribute), 213
- `data_source_id` (`flexmeasures.data.models.markets.Price` attribute), 227
- `data_source_id` (`flexmeasures.data.models.weather.Weather` attribute), 248
- `DataGeneratorMixin` (class in `flexmeasures.data.models.data_sources`), 215
- `DataSource` (class in `flexmeasures.data.models.data_sources`), 215
- `DataSourceIdField` (class in `flexmeasures.data.schemas.sources`), 265

- `datetime` (*flexmeasures.data.models.assets.Power attribute*), 213
- `datetime` (*flexmeasures.data.models.markets.Price attribute*), 227
- `datetime` (*flexmeasures.data.models.weather.Weather attribute*), 248
- `decide_resolution()` (in module *flexmeasures.utils.time_utils*), 304
- `delete()` (*flexmeasures.api.v3_0.assets.AssetAPI method*), 187
- `delete_user()` (in module *flexmeasures.data.services.users*), 283
- `delete_with_data()` (*flexmeasures.ui.crud.assets.AssetCrudUI method*), 288
- `demand` (*flexmeasures.data.services.resources.Resource property*), 278
- `deprecate_blueprint()` (in module *flexmeasures.api.common.utils.deprecation_utils*), 174
- `deprecate_fields()` (in module *flexmeasures.api.common.utils.deprecation_utils*), 175
- `deprecated()` (in module *flexmeasures.utils.coding_utils*), 296
- `deprecated_api_version()` (in module *flexmeasures.api.common.responses*), 162
- `DeprecatedDefaultGroup` (class in *flexmeasures.cli.utils*), 207
- `description` (*flexmeasures.data.models.data_sources.DataSource property*), 215
- `deserialize_config()` (*flexmeasures.data.models.planning.Scheduler method*), 236
- `deserialize_config()` (*flexmeasures.data.models.reporting.aggregator.Aggregator method*), 237
- `deserialize_config()` (*flexmeasures.data.models.reporting.pandas_reporter.PandasReporter method*), 237
- `deserialize_config()` (*flexmeasures.data.models.reporting.Reporter method*), 238
- `deserialize_flex_config()` (*flexmeasures.data.models.planning.Scheduler method*), 236
- `deserialize_flex_config()` (*flexmeasures.data.models.planning.storage.StorageScheduler method*), 233
- `deserialize_timing_config()` (*flexmeasures.data.models.planning.Scheduler method*), 236
- `determine_belief_timing()` (in module *flexmeasures.api.common.utils.api_utils*), 171
- `determine_flow_unit()` (in module *flexmeasures.utils.unit_utils*), 306
- `determine_minimum_resampling_resolution()` (in module *flexmeasures.utils.time_utils*), 304
- `determine_shared_sensor_type()` (in module *flexmeasures.data.models.charts.belief_charts*), 214
- `determine_shared_unit()` (in module *flexmeasures.data.models.charts.belief_charts*), 214
- `determine_stock_unit()` (in module *flexmeasures.utils.unit_utils*), 307
- `determine_unit_conversion_multiplier()` (in module *flexmeasures.utils.unit_utils*), 307
- `DevelopmentConfig` (class in *flexmeasures.utils.config_defaults*), 298
- `device_scheduler()` (in module *flexmeasures.data.models.planning.linear_optimization*), 229
- `display_name` (*flexmeasures.data.services.asset_grouping.AssetGroup property*), 272
- `display_name` (*flexmeasures.data.services.resources.Resource property*), 278
- `docs_view()` (in module *flexmeasures.ui.views*), 293
- `DocumentationConfig` (class in *flexmeasures.utils.config_defaults*), 298
- `drop_nan_rows()` (in module *flexmeasures.utils.calculations*), 295
- `drop_non_unique_ids()` (in module *flexmeasures.data.services.time_series*), 281
- `drop_unchanged_beliefs()` (in module *flexmeasures.data.services.time_series*), 281
- `dump_bdf()` (*flexmeasures.api.common.schemas.sensor_data.GetSensorData method*), 166
- `duration_isoformat()` (in module *flexmeasures.utils.time_utils*), 304
- `DurationField` (class in *flexmeasures.data.schemas.times*), 266
- `DurationValidationError`, 266
- ## E
- `earth_distance()` (in module *flexmeasures.utils.geo_utils*), 301
- `EfficiencyField` (class in *flexmeasures.data.schemas.scheduling.storage*), 264
- `enqueue_forecasting_jobs()` (in module *flexmeasures.api.common.utils.api_utils*), 171
- `ensure_local_timezone()` (in module *flexmeasures.utils.time_utils*), 304
- `ensure_soc_min_max()` (*flexmeasures.data.models.planning.storage.StorageScheduler*

- method), 233
- ensure_timing_vars_are_set() (in module *flexmeasures.ui.utils.view_utils*), 291
- entity_address (flexmeasures.data.models.assets.Asset property), 212
- entity_address (flexmeasures.data.models.markets.Market property), 226
- entity_address (flexmeasures.data.models.weather.WeatherSensor property), 248
- entity_address_fm0 (flexmeasures.data.models.assets.Asset property), 212
- entity_address_fm0 (flexmeasures.data.models.markets.Market property), 226
- entity_address_fm0 (flexmeasures.data.models.weather.WeatherSensor property), 248
- EntityAddressException, 300
- EntityAddressValidationError, 169
- error_handling_router() (in module *flexmeasures.utils.error_utils*), 300
- event_resolution (flexmeasures.data.models.assets.Asset attribute), 212
- event_resolution (flexmeasures.data.models.markets.Market attribute), 226
- event_resolution (flexmeasures.data.models.time_series.Sensor attribute), 240
- event_resolution (flexmeasures.data.models.weather.WeatherSensor attribute), 248
- event_start (flexmeasures.data.models.time_series.TimedBelief attribute), 243
- event_value (flexmeasures.data.models.time_series.TimedBelief attribute), 243
- F**
- fallback_charging_policy() (in module *flexmeasures.data.models.planning.utils*), 234
- fetch_data() (flexmeasures.data.models.reporting.Reporter method), 238
- fetch_one() (flexmeasures.api.v3_0.assets.AssetAPI method), 187
- find_classes_module() (in module *flexmeasures.utils.coding_utils*), 296
- find_classes_modules() (in module *flexmeasures.utils.coding_utils*), 296
- find_closest() (flexmeasures.data.models.time_series.Sensor class method), 240
- find_first_applicable_config_entry() (in module *flexmeasures.utils.app_utils*), 294
- find_scheduler_class() (in module *flexmeasures.data.services.scheduling*), 279
- find_sensor_by_name() (in module *flexmeasures.data.services.time_series*), 281
- find_user_by_email() (in module *flexmeasures.data.services.users*), 283
- flatten_unique() (in module *flexmeasures.utils.coding_utils*), 296
- FlexContextSchema (class in *flexmeasures.data.schemas.scheduling*), 264
- flexmeasures.api module, 200
- flexmeasures.api.common module, 181
- flexmeasures.api.common.implementations module, 162
- flexmeasures.api.common.responses module, 162
- flexmeasures.api.common.routes module, 164
- flexmeasures.api.common.schemas module, 170
- flexmeasures.api.common.schemas.generic_assets module, 164
- flexmeasures.api.common.schemas.sensor_data module, 165
- flexmeasures.api.common.schemas.sensors module, 168
- flexmeasures.api.common.schemas.users module, 169
- flexmeasures.api.common.utils module, 181
- flexmeasures.api.common.utils.api_utils module, 171
- flexmeasures.api.common.utils.args_parsing module, 173
- flexmeasures.api.common.utils.decorators module, 174
- flexmeasures.api.common.utils.deprecation_utils module, 174
- flexmeasures.api.common.utils.migration_utils module, 176
- flexmeasures.api.common.utils.validators module, 177
- flexmeasures.api.dev module, 182
- flexmeasures.api.dev.sensors

- [module, 181](#)
- [flexmeasures.api.play](#)
 - [module, 184](#)
- [flexmeasures.api.play.implementations](#)
 - [module, 183](#)
- [flexmeasures.api.play.routes](#)
 - [module, 183](#)
- [flexmeasures.api.sunset](#)
 - [module, 184](#)
- [flexmeasures.api.sunset.routes](#)
 - [module, 184](#)
- [flexmeasures.api.v3_0](#)
 - [module, 200](#)
- [flexmeasures.api.v3_0.accounts](#)
 - [module, 185](#)
- [flexmeasures.api.v3_0.assets](#)
 - [module, 187](#)
- [flexmeasures.api.v3_0.health](#)
 - [module, 191](#)
- [flexmeasures.api.v3_0.public](#)
 - [module, 191](#)
- [flexmeasures.api.v3_0.sensors](#)
 - [module, 191](#)
- [flexmeasures.api.v3_0.users](#)
 - [module, 197](#)
- [flexmeasures.app](#)
 - [module, 201](#)
- [flexmeasures.auth](#)
 - [module, 204](#)
- [flexmeasures.auth.decorators](#)
 - [module, 201](#)
- [flexmeasures.auth.error_handling](#)
 - [module, 203](#)
- [flexmeasures.auth.policy](#)
 - [module, 203](#)
- [flexmeasures.cli](#)
 - [module, 208](#)
- [flexmeasures.cli.data_add](#)
 - [module, 205](#)
- [flexmeasures.cli.data_delete](#)
 - [module, 205](#)
- [flexmeasures.cli.data_edit](#)
 - [module, 205](#)
- [flexmeasures.cli.data_show](#)
 - [module, 205](#)
- [flexmeasures.cli.db_ops](#)
 - [module, 206](#)
- [flexmeasures.cli.jobs](#)
 - [module, 206](#)
- [flexmeasures.cli.monitor](#)
 - [module, 206](#)
- [flexmeasures.cli.utils](#)
 - [module, 206](#)
- [flexmeasures.data](#)

- [module, 286](#)
- [flexmeasures.data.config](#)
 - [module, 209](#)
- [flexmeasures.data.models](#)
 - [module, 249](#)
- [flexmeasures.data.models.annotations](#)
 - [module, 210](#)
- [flexmeasures.data.models.assets](#)
 - [module, 212](#)
- [flexmeasures.data.models.charts](#)
 - [module, 215](#)
- [flexmeasures.data.models.charts.belief_charts](#)
 - [module, 214](#)
- [flexmeasures.data.models.charts.defaults](#)
 - [module, 215](#)
- [flexmeasures.data.models.data_sources](#)
 - [module, 215](#)
- [flexmeasures.data.models.forecasting](#)
 - [module, 221](#)
- [flexmeasures.data.models.forecasting.exceptions](#)
 - [module, 216](#)
- [flexmeasures.data.models.forecasting.model_spec_factory](#)
 - [module, 216](#)
- [flexmeasures.data.models.forecasting.model_specs](#)
 - [module, 219](#)
- [flexmeasures.data.models.forecasting.model_specs.linear_regression](#)
 - [module, 219](#)
- [flexmeasures.data.models.forecasting.model_specs.naive](#)
 - [module, 219](#)
- [flexmeasures.data.models.forecasting.utils](#)
 - [module, 220](#)
- [flexmeasures.data.models.generic_assets](#)
 - [module, 221](#)
- [flexmeasures.data.models.legacy_migration_utils](#)
 - [module, 225](#)
- [flexmeasures.data.models.markets](#)
 - [module, 226](#)
- [flexmeasures.data.models.parsing_utils](#)
 - [module, 227](#)
- [flexmeasures.data.models.planning](#)
 - [module, 235](#)
- [flexmeasures.data.models.planning.battery](#)
 - [module, 228](#)
- [flexmeasures.data.models.planning.charging_station](#)
 - [module, 228](#)
- [flexmeasures.data.models.planning.exceptions](#)
 - [module, 228](#)
- [flexmeasures.data.models.planning.linear_optimization](#)
 - [module, 229](#)
- [flexmeasures.data.models.planning.storage](#)
 - [module, 230](#)
- [flexmeasures.data.models.planning.utils](#)
 - [module, 234](#)
- [flexmeasures.data.models.reporting](#)

module, 238	module, 266
flexmeasures.data.models.reporting.aggregator	flexmeasures.data.schemas.units
module, 236	module, 266
flexmeasures.data.models.reporting.pandas_reporter	flexmeasures.data.schemas.users
module, 237	module, 267
flexmeasures.data.models.task_runs	flexmeasures.data.schemas.utils
module, 239	module, 267
flexmeasures.data.models.time_series	flexmeasures.data.scripts
module, 239	module, 269
flexmeasures.data.models.user	flexmeasures.data.scripts.data_gen
module, 245	module, 269
flexmeasures.data.models.validation_utils	flexmeasures.data.scripts.visualize_data_model
module, 247	module, 269
flexmeasures.data.models.weather	flexmeasures.data.services
module, 248	module, 284
flexmeasures.data.queries	flexmeasures.data.services.accounts
module, 256	module, 270
flexmeasures.data.queries.analytics	flexmeasures.data.services.annotations
module, 250	module, 270
flexmeasures.data.queries.annotations	flexmeasures.data.services.asset_grouping
module, 251	module, 271
flexmeasures.data.queries.data_sources	flexmeasures.data.services.data_sources
module, 251	module, 272
flexmeasures.data.queries.generic_assets	flexmeasures.data.services.forecasting
module, 251	module, 273
flexmeasures.data.queries.portfolio	flexmeasures.data.services.resources
module, 252	module, 274
flexmeasures.data.queries.sensors	flexmeasures.data.services.scheduling
module, 253	module, 279
flexmeasures.data.queries.utils	flexmeasures.data.services.sensors
module, 254	module, 280
flexmeasures.data.schemas	flexmeasures.data.services.time_series
module, 268	module, 281
flexmeasures.data.schemas.account	flexmeasures.data.services.timerange
module, 257	module, 282
flexmeasures.data.schemas.assets	flexmeasures.data.services.users
module, 257	module, 283
flexmeasures.data.schemas.generic_assets	flexmeasures.data.services.utils
module, 258	module, 284
flexmeasures.data.schemas.reporting	flexmeasures.data.transactional
module, 263	module, 285
flexmeasures.data.schemas.reporting.aggregation	flexmeasures.data.utils
module, 260	module, 285
flexmeasures.data.schemas.reporting.pandas_reporter	flexmeasures.ui
module, 261	module, 293
flexmeasures.data.schemas.scheduling	flexmeasures.ui.crud
module, 264	module, 289
flexmeasures.data.schemas.scheduling.storage	flexmeasures.ui.crud.accounts
module, 264	module, 287
flexmeasures.data.schemas.sensors	flexmeasures.ui.crud.api_wrapper
module, 265	module, 287
flexmeasures.data.schemas.sources	flexmeasures.ui.crud.assets
module, 265	module, 288
flexmeasures.data.schemas.times	flexmeasures.ui.crud.users

- module, 289
- `flexmeasures.ui.error_handlers`
 - module, 290
- `flexmeasures.ui.utils`
 - module, 291
- `flexmeasures.ui.utils.chart_defaults`
 - module, 290
- `flexmeasures.ui.utils.view_utils`
 - module, 290
- `flexmeasures.ui.views`
 - module, 293
- `flexmeasures.ui.views.control`
 - module, 292
- `flexmeasures.ui.views.logged_in_user`
 - module, 292
- `flexmeasures.ui.views.new_dashboard`
 - module, 292
- `flexmeasures.ui.views.sensors`
 - module, 293
- `flexmeasures.utils`
 - module, 307
- `flexmeasures.utils.app_utils`
 - module, 294
- `flexmeasures.utils.calculations`
 - module, 295
- `flexmeasures.utils.coding_utils`
 - module, 296
- `flexmeasures.utils.config_defaults`
 - module, 298
- `flexmeasures.utils.config_utils`
 - module, 298
- `flexmeasures.utils.entity_address_utils`
 - module, 299
- `flexmeasures.utils.error_utils`
 - module, 300
- `flexmeasures.utils.flexmeasures_inflection`
 - module, 301
- `flexmeasures.utils.geo_utils`
 - module, 301
- `flexmeasures.utils.grid_cells`
 - module, 302
- `flexmeasures.utils.plugin_utils`
 - module, 303
- `flexmeasures.utils.time_utils`
 - module, 304
- `flexmeasures.utils.unit_utils`
 - module, 306
- `FMValidationError`, 268
- `forecast_horizons_for()` (in module `flexmeasures.utils.time_utils`), 304
- `freq_label_to_human_readable_label()` (in module `flexmeasures.utils.time_utils`), 304

G

- `GenericAsset` (class in `flexmeasures.data.models.generic_assets`), 222
- `GenericAssetAnnotationRelationship` (class in `flexmeasures.data.models.annotations`), 211
- `GenericAssetIdField` (class in `flexmeasures.data.schemas.generic_assets`), 258
- `GenericAssetSchema` (class in `flexmeasures.data.schemas.generic_assets`), 258
- `GenericAssetSchema.Meta` (class in `flexmeasures.data.schemas.generic_assets`), 258
- `GenericAssetType` (class in `flexmeasures.data.models.generic_assets`), 225
- `GenericAssetTypeSchema` (class in `flexmeasures.data.schemas.generic_assets`), 258
- `GenericAssetTypeSchema.Meta` (class in `flexmeasures.data.schemas.generic_assets`), 258
- `get()` (`flexmeasures.api.dev.sensors.AssetAPI` method), 182
- `get()` (`flexmeasures.api.dev.sensors.SensorAPI` method), 182
- `get()` (`flexmeasures.api.v3_0.accounts.AccountAPI` method), 185
- `get()` (`flexmeasures.api.v3_0.users.UserAPI` method), 197
- `get()` (`flexmeasures.ui.crud.accounts.AccountCrudUI` method), 287
- `get()` (`flexmeasures.ui.crud.assets.AssetCrudUI` method), 288
- `get()` (`flexmeasures.ui.crud.users.UserCrudUI` method), 289
- `get()` (`flexmeasures.ui.views.sensors.SensorUI` method), 293
- `get_account()` (in module `flexmeasures.ui.crud.accounts`), 287
- `get_account_roles()` (in module `flexmeasures.data.services.accounts`), 270
- `get_accounts()` (in module `flexmeasures.data.services.accounts`), 270
- `get_accounts()` (in module `flexmeasures.ui.crud.accounts`), 287
- `get_affected_classes()` (in module `flexmeasures.data.scripts.data_gen`), 269
- `get_asset_group_queries()` (in module `flexmeasures.data.queries.generic_assets`), 251
- `get_asset_group_queries()` (in module `flexmeasures.data.services.asset_grouping`), 271
- `get_asset_group_queries()` (in module `flexmeasures.data.services.resources`), 275
- `get_assets()` (in module `flexmeasures.data.services.resources`), 275
- `get_assets_by_account()` (in module `flexmeasures.ui.crud.assets`), 288
- `get_attribute()` (`flexmea-`

- `asures.data.models.assets.Asset` *method*), 212
- `get_attribute()` (*flexmeasures.data.models.markets.Market* *method*), 226
- `get_attribute()` (*flexmeasures.data.models.time_series.Sensor* *method*), 240
- `get_attribute()` (*flexmeasures.data.models.weather.WeatherSensor* *method*), 248
- `get_belief_timing_criteria()` (*in module flexmeasures.data.queries.utils*), 254
- `get_cell_nums()` (*in module flexmeasures.utils.grid_cells*), 302
- `get_center_location()` (*in module flexmeasures.data.services.resources*), 275
- `get_center_location_of_assets()` (*in module flexmeasures.data.models.generic_assets*), 221
- `get_chart()` (*flexmeasures.api.dev.sensors.SensorAPI* *method*), 182
- `get_chart()` (*flexmeasures.api.v3_0.assets.AssetAPI* *method*), 188
- `get_chart()` (*flexmeasures.ui.views.sensors.SensorUI* *method*), 293
- `get_chart_annotations()` (*flexmeasures.api.dev.sensors.SensorAPI* *method*), 182
- `get_chart_data()` (*flexmeasures.api.dev.sensors.SensorAPI* *method*), 182
- `get_chart_data()` (*flexmeasures.api.v3_0.assets.AssetAPI* *method*), 188
- `get_classes_module()` (*in module flexmeasures.utils.coding_utils*), 296
- `get_command()` (*flexmeasures.cli.utils.DeprecatedDefaultGroup* *method*), 208
- `get_config_warnings()` (*in module flexmeasures.utils.config_utils*), 298
- `get_configuration_keys()` (*in module flexmeasures.utils.config_utils*), 298
- `get_data()` (*flexmeasures.api.v3_0.sensors.SensorAPI* *method*), 191
- `get_data_downsampling_allowed()` (*in module flexmeasures.api.common.utils.validators*), 177
- `get_data_source()` (*in module flexmeasures.data.utils*), 285
- `get_data_source_for_job()` (*in module flexmeasures.data.services.scheduling*), 279
- `get_data_source_info()` (*flexmeasures.data.models.data_sources.DataGeneratorModel* *class method*), 215
- `get_data_source_info()` (*flexmeasures.data.models.planning.Scheduler* *class method*), 236
- `get_default_end_time()` (*in module flexmeasures.utils.time_utils*), 304
- `get_default_start_time()` (*in module flexmeasures.utils.time_utils*), 304
- `get_demand_from_bdf()` (*in module flexmeasures.data.services.resources*), 275
- `get_domain_parts()` (*in module flexmeasures.utils.entity_address_utils*), 299
- `get_err_source_info()` (*in module flexmeasures.utils.error_utils*), 300
- `get_first_day_of_next_month()` (*in module flexmeasures.utils.time_utils*), 304
- `get_form_from_request()` (*in module flexmeasures.api.common.utils.api_utils*), 171
- `get_git_description()` (*in module flexmeasures.ui.utils.view_utils*), 291
- `get_host()` (*in module flexmeasures.utils.entity_address_utils*), 299
- `get_location_queries()` (*in module flexmeasures.data.queries.generic_assets*), 252
- `get_location_queries()` (*in module flexmeasures.data.services.resources*), 275
- `get_locations()` (*flexmeasures.utils.grid_cells.LatLngGrid* *method*), 302
- `get_market()` (*in module flexmeasures.data.models.planning.utils*), 234
- `get_markets()` (*in module flexmeasures.data.services.resources*), 276
- `get_max_planning_horizon()` (*in module flexmeasures.utils.time_utils*), 304
- `get_metavar()` (*flexmeasures.data.schemas.utils.MarshmallowClickMixin* *method*), 268
- `get_most_recent_clocktime_window()` (*in module flexmeasures.utils.time_utils*), 305
- `get_most_recent_hour()` (*in module flexmeasures.utils.time_utils*), 305
- `get_most_recent_quarter()` (*in module flexmeasures.utils.time_utils*), 305
- `get_normalization_transformation_from_sensor_attributes()` (*in module flexmeasures.data.models.forecasting.model_spec_factory*), 217
- `get_number_of_assets_in_account()` (*in module flexmeasures.data.services.accounts*), 270
- `get_object_or_literal()` (*flexmeasures.data.models.reporting.pandas_reporter.PandasReporter* *method*), 238
- `get_old_model_type()` (*in module flexmeasures.data.models.legacy_migration_utils*),

- 225
- `get_or_create_annotation()` (in module *flexmeasures.data.models.annotations*), 210
- `get_or_create_model()` (in module *flexmeasures.data.services.utils*), 284
- `get_or_create_source()` (in module *flexmeasures.data.queries.data_sources*), 251
- `get_or_create_source()` (in module *flexmeasures.data.services.data_sources*), 272
- `get_pattern_match_word()` (in module *flexmeasures.data.models.planning.storage*), 231
- `get_ping()` (in module *flexmeasures.api.common.routes*), 164
- `get_power_data()` (in module *flexmeasures.data.queries.analytics*), 250
- `get_power_data()` (in module *flexmeasures.data.queries.portfolio*), 252
- `get_power_values()` (in module *flexmeasures.data.models.planning.utils*), 234
- `get_price_data()` (in module *flexmeasures.data.queries.portfolio*), 253
- `get_prices()` (in module *flexmeasures.data.models.planning.utils*), 234
- `get_prices_data()` (in module *flexmeasures.data.queries.analytics*), 250
- `get_query_window()` (in module *flexmeasures.data.models.forecasting.utils*), 220
- `get_revenues_costs_data()` (in module *flexmeasures.data.queries.analytics*), 250
- `get_schedule()` (*flexmeasures.api.v3_0.sensors.SensorAPI* method), 192
- `get_sensor_by_generic_asset_type_and_location@ground_from()` (in module *flexmeasures.api.common.utils.api_utils*), 171
- `get_sensor_by_unique_name()` (in module *flexmeasures.api.common.utils.migration_utils*), 176
- `get_sensor_or_abort()` (in module *flexmeasures.api.dev.sensors*), 181
- `get_sensor_types()` (in module *flexmeasures.data.services.resources*), 276
- `get_sensors()` (in module *flexmeasures.data.services.resources*), 276
- `get_sensors()` (in module *flexmeasures.data.services.sensors*), 280
- `get_source_criteria()` (in module *flexmeasures.data.queries.utils*), 254
- `get_source_or_none()` (in module *flexmeasures.data.queries.data_sources*), 251
- `get_source_or_none()` (in module *flexmeasures.data.services.data_sources*), 272
- `get_structure()` (in module *flexmeasures.data.queries.portfolio*), 253
- `get_supply_from_bdf()` (in module *flexmeasures.data.services.resources*), 276
- `get_task_run()` (in module *flexmeasures.api.common.implementations*), 162
- `get_task_run()` (in module *flexmeasures.api.common.routes*), 164
- `get_timerange()` (*flexmeasures.data.models.generic_assets.GenericAsset* class method), 223
- `get_timerange()` (in module *flexmeasures.data.services.timerange*), 282
- `get_timerange_from_flag()` (in module *flexmeasures.cli.utils*), 207
- `get_timezone()` (in module *flexmeasures.utils.time_utils*), 305
- `get_user()` (in module *flexmeasures.data.services.users*), 283
- `get_users()` (in module *flexmeasures.data.services.users*), 283
- `get_users_by_account()` (in module *flexmeasures.ui.crud.users*), 289
- `get_versions()` (in module *flexmeasures.api*), 201
- `get_weather_data()` (in module *flexmeasures.data.queries.analytics*), 250
- `GetSensorDataSchema` (class in *flexmeasures.api.common.schemas.sensor_data*), 165
- `great_circle_distance()` (*flexmeasures.data.models.generic_assets.GenericAsset* method), 223
- `great_circle_distance()` (*flexmeasures.data.models.weather.WeatherSensor* method), 248
- `great_circle_distance@ground_from()` (*flexmeasures.data.schemas.times.DurationField* static method), 266
- `group_assets_by_location()` (in module *flexmeasures.data.queries.generic_assets*), 252
- `group_assets_by_location()` (in module *flexmeasures.data.services.resources*), 276
- `groups_to_dict()` (in module *flexmeasures.api.common.utils.api_utils*), 172
- ## H
- `handle_500_error()` (in module *flexmeasures.ui.error_handlers*), 290
- `handle_bad_request()` (in module *flexmeasures.ui.error_handlers*), 290
- `handle_error()` (in module *flexmeasures.api.common.utils.args_parsing*), 173
- `handle_forecasting_exception()` (in module *flexmeasures.data.services.forecasting*), 273
- `handle_generic_http_exception()` (in module *flexmeasures.ui.error_handlers*), 290

- [handle_not_found\(\)](#) (in module `flexmeasures.ui.error_handlers`), 290
[handle_scheduling_exception\(\)](#) (in module `flexmeasures.data.services.scheduling`), 279
[handle_worker_exception\(\)](#) (in module `flexmeasures.cli.jobs`), 206
[has_assets\(\)](#) (in module `flexmeasures.data.services.resources`), 276
[has_energy_sensors](#) (`flexmeasures.data.models.generic_assets.GenericAsset` property), 223
[has_power_sensors](#) (`flexmeasures.data.models.generic_assets.GenericAsset` property), 223
[has_role\(\)](#) (`flexmeasures.data.models.user.Account` method), 245
[has_role\(\)](#) (`flexmeasures.data.models.user.User` method), 247
[hash_function_arguments\(\)](#) (in module `flexmeasures.data.services.utils`), 284
[HealthAPI](#) (class in `flexmeasures.api.v3_0.health`), 191
[horizon](#) (`flexmeasures.data.models.assets.Power` attribute), 213
[horizon](#) (`flexmeasures.data.models.markets.Price` attribute), 227
[horizon](#) (`flexmeasures.data.models.weather.Weather` attribute), 248
[hover_label](#) (`flexmeasures.data.services.asset_grouping.AssetGroup` property), 272
[hover_label](#) (`flexmeasures.data.services.resources.Resource` property), 278
[humanize\(\)](#) (in module `flexmeasures.utils.flexmeasures_inflection`), 301

I
[id](#) (`flexmeasures.data.models.assets.Asset` attribute), 212
[id](#) (`flexmeasures.data.models.data_sources.DataSource` attribute), 216
[id](#) (`flexmeasures.data.models.markets.Market` attribute), 226
[id](#) (`flexmeasures.data.models.time_series.Sensor` attribute), 240
[id](#) (`flexmeasures.data.models.weather.WeatherSensor` attribute), 248
[idle_after_reaching_target\(\)](#) (in module `flexmeasures.data.models.planning.utils`), 235
[implementation_gone\(\)](#) (in module `flexmeasures.api.sunset.routes`), 184
[include_current_user_source_id\(\)](#) (in module `flexmeasures.api.common.utils.validators`), 177
[incomplete_event\(\)](#) (in module `flexmeasures.api.common.responses`), 162

[index\(\)](#) (`flexmeasures.api.v3_0.accounts.AccountAPI` method), 186
[index\(\)](#) (`flexmeasures.api.v3_0.assets.AssetAPI` method), 188
[index\(\)](#) (`flexmeasures.api.v3_0.public.ServicesAPI` method), 191
[index\(\)](#) (`flexmeasures.api.v3_0.sensors.SensorAPI` method), 193
[index\(\)](#) (`flexmeasures.api.v3_0.users.UserAPI` method), 198
[index\(\)](#) (`flexmeasures.ui.crud.accounts.AccountCrudUI` method), 287
[index\(\)](#) (`flexmeasures.ui.crud.assets.AssetCrudUI` method), 288
[index\(\)](#) (`flexmeasures.ui.crud.users.UserCrudUI` method), 289
[init_db\(\)](#) (in module `flexmeasures.data.config`), 209
[init_sentry\(\)](#) (in module `flexmeasures.utils.app_utils`), 294
[initialize_df\(\)](#) (in module `flexmeasures.data.models.planning.utils`), 235
[initialize_index\(\)](#) (in module `flexmeasures.data.models.planning.utils`), 235
[initialize_series\(\)](#) (in module `flexmeasures.data.models.planning.utils`), 235
[integrate_time_series\(\)](#) (in module `flexmeasures.utils.calculations`), 295
[InternalApi](#) (class in `flexmeasures.ui.crud.api_wrapper`), 287
[invalid_datetime\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_domain\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_flex_config\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_horizon\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_market\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_message_type\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_method\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_period\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_ptu_duration\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_replacement\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_resolution_str\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_role\(\)](#) (in module `flexmeasures.api.common.responses`), 162
[invalid_sender\(\)](#) (in module `flexmeasures`), 162

`sures.api.common.responses`), 162
`invalid_source()` (in module `flexmeasures.api.common.responses`), 162
`invalid_timezone()` (in module `flexmeasures.api.common.responses`), 163
`invalid_unit()` (in module `flexmeasures.api.common.responses`), 163
`InvalidFlexMeasuresUser`, 283
`InvalidHorizonException`, 216
`is_authenticated` (flexmeasures.data.models.user.User property), 247
`is_eligible_for_comparing_individual_traces()` (flexmeasures.data.services.asset_grouping.AssetGroup method), 272
`is_eligible_for_comparing_individual_traces()` (flexmeasures.data.services.resources.Resource method), 278
`is_energy_price_unit()` (in module `flexmeasures.utils.unit_utils`), 307
`is_energy_unit()` (in module `flexmeasures.utils.unit_utils`), 307
`is_power_unit()` (in module `flexmeasures.utils.unit_utils`), 307
`is_pure_consumer` (flexmeasures.data.models.assets.Asset property), 213
`is_pure_producer` (flexmeasures.data.models.assets.Asset property), 213
`is_ready()` (flexmeasures.api.v3_0.health.HealthAPI method), 191
`is_response_tuple()` (in module `flexmeasures.api.common.responses`), 163
`is_running()` (in module `flexmeasures.cli`), 208
`is_strictly_non_negative` (flexmeasures.data.models.time_series.Sensor property), 240
`is_strictly_non_positive` (flexmeasures.data.models.time_series.Sensor property), 240
`is_unique_asset` (flexmeasures.data.services.asset_grouping.AssetGroup property), 272
`is_unique_asset` (flexmeasures.data.services.resources.Resource property), 278
`is_user()` (in module `flexmeasures.data.models.user`), 245
`is_valid_unit()` (in module `flexmeasures.utils.unit_utils`), 307

J

`job_cache()` (in module `flexmeasures.data.services.utils`), 284

`join_words_into_a_list()` (in module `flexmeasures.utils.flexmeasures_inflection`), 301
`JSON` (class in `flexmeasures.data.schemas.generic_assets`), 258

K

`knowledge_horizon_fnc` (flexmeasures.data.models.assets.Asset attribute), 213
`knowledge_horizon_fnc` (flexmeasures.data.models.markets.Market attribute), 227
`knowledge_horizon_fnc` (flexmeasures.data.models.time_series.Sensor attribute), 240
`knowledge_horizon_fnc` (flexmeasures.data.models.weather.WeatherSensor attribute), 249
`knowledge_horizon_par` (flexmeasures.data.models.assets.Asset attribute), 213
`knowledge_horizon_par` (flexmeasures.data.models.markets.Market attribute), 227
`knowledge_horizon_par` (flexmeasures.data.models.time_series.Sensor attribute), 241
`knowledge_horizon_par` (flexmeasures.data.models.weather.WeatherSensor attribute), 249

L

`label` (flexmeasures.data.models.data_sources.DataSource property), 216
`latest_state()` (flexmeasures.data.models.assets.Asset method), 213
`latest_state()` (flexmeasures.data.models.time_series.Sensor method), 241
`LatestTaskRun` (class in `flexmeasures.data.models.task_runs`), 239
`LatitudeField` (class in `flexmeasures.data.schemas.assets`), 257
`LatitudeLongitudeValidator` (class in `flexmeasures.data.schemas.assets`), 257
`LatitudeValidator` (class in `flexmeasures.data.schemas.assets`), 258
`LatLngGrid` (class in `flexmeasures.utils.grid_cells`), 302
`list_access()` (in module `flexmeasures.api.common.utils.api_utils`), 172
`list_items()` (in module `flexmeasures.cli.data_show`), 205

- `load_bdf()` (*flexmeasures.api.common.schemas.sensor_data.Resource* static method), 166
- `load_current()` (*flexmeasures.api.common.schemas.users.AccountIdField* class method), 169
- `load_custom_scheduler()` (in module *flexmeasures.data.services.scheduling*), 280
- `load_data()` (in module *flexmeasures.api.common.utils.args_parsing*), 173
- `load_default()` (*flexmeasures.data.schemas.times.PlanningDurationField* class method), 266
- `load_sensor_data()` (*flexmeasures.data.services.resources.Resource* method), 278
- `localized_datetime()` (in module *flexmeasures.utils.time_utils*), 305
- `localized_datetime_str()` (in module *flexmeasures.utils.time_utils*), 305
- `locations_hex()` (*flexmeasures.utils.grid_cells.LatLngGrid* method), 302
- `locations_square()` (*flexmeasures.utils.grid_cells.LatLngGrid* method), 302
- `log_error()` (in module *flexmeasures.utils.error_utils*), 300
- `log_missing_config_setting()` (in module *flexmeasures.utils.plugin_utils*), 303
- `log_wrong_type_for_config_setting()` (in module *flexmeasures.utils.plugin_utils*), 303
- `logged_in_user_view()` (in module *flexmeasures.ui.views.logged_in_user*), 292
- `LongitudeField` (class in *flexmeasures.data.schemas.assets*), 258
- `LongitudeValidator` (class in *flexmeasures.data.schemas.assets*), 258
- `lookup_model_specs_configurator()` (in module *flexmeasures.data.models.forecasting*), 221
- M**
- `make_fixed_viewpoint_forecasts()` (in module *flexmeasures.data.services.forecasting*), 273
- `make_hash_sha256()` (in module *flexmeasures.data.services.utils*), 284
- `make_hashable()` (*flexmeasures.data.models.time_series.Sensor* method), 241
- `make_hashable()` (in module *flexmeasures.data.services.utils*), 284
- `make_query()` (*flexmeasures.data.models.assets.Power* class method), 213
- `make_query()` (*flexmeasures.data.models.markets.Price* class method), 227
- `make_query()` (*DataSchema* (in module *flexmeasures.data.models.time_series.TimedValue* class method), 244
- `make_query()` (*flexmeasures.data.models.weather.Weather* class method), 248
- `make_registering_decorator()` (in module *flexmeasures.utils.coding_utils*), 296
- `make_rolling_viewpoint_forecasts()` (in module *flexmeasures.data.services.forecasting*), 273
- `make_schedule()` (in module *flexmeasures.data.services.scheduling*), 280
- `Market` (class in *flexmeasures.data.models.markets*), 226
- `MarketType` (class in *flexmeasures.data.models.markets*), 227
- `MarshmallowClickMixin` (class in *flexmeasures.data.schemas.utils*), 268
- `mask_inaccessible_assets()` (in module *flexmeasures.data.services.resources*), 276
- `mean_absolute_error()` (in module *flexmeasures.utils.calculations*), 296
- `mean_absolute_percentage_error()` (in module *flexmeasures.utils.calculations*), 296
- `measures_energy` (*flexmeasures.data.models.time_series.Sensor* property), 241
- `measures_power` (*flexmeasures.data.models.time_series.Sensor* property), 241
- `merge_vega_lite_specs()` (in module *flexmeasures.data.models.charts.defaults*), 215
- `message_replace_name_with_ea()` (in module *flexmeasures.api.common.utils.api_utils*), 172
- `methods_with_decorator()` (in module *flexmeasures.utils.coding_utils*), 297
- `MisconfiguredForecastingJobException`, 274
- `MissingAttributeException`, 228, 247
- `model` (*flexmeasures.data.schemas.account.AccountRoleSchema.Meta* attribute), 257
- `model` (*flexmeasures.data.schemas.account.AccountSchema.Meta* attribute), 257
- `model` (*flexmeasures.data.schemas.assets.AssetSchema.Meta* attribute), 257
- `model` (*flexmeasures.data.schemas.generic_assets.GenericAssetSchema.Meta* attribute), 258
- `model` (*flexmeasures.data.schemas.generic_assets.GenericAssetTypeSchema* attribute), 258
- `model` (*flexmeasures.data.schemas.sensors.SensorSchema.Meta* attribute), 265
- `model` (*flexmeasures.data.schemas.users.UserSchema.Meta* attribute), 267
- `ModelException`, 249
- module *flexmeasures.api*, 200

`flexmeasures.api.common`, 181
`flexmeasures.api.common.implementations`, 162
`flexmeasures.api.common.responses`, 162
`flexmeasures.api.common.routes`, 164
`flexmeasures.api.common.schemas`, 170
`flexmeasures.api.common.schemas.generic_assets`, 164
`flexmeasures.api.common.schemas.sensor_data`, 165
`flexmeasures.api.common.schemas.sensors`, 168
`flexmeasures.api.common.schemas.users`, 169
`flexmeasures.api.common.utils`, 181
`flexmeasures.api.common.utils.api_utils`, 171
`flexmeasures.api.common.utils.args_parsing`, 173
`flexmeasures.api.common.utils.decorators`, 174
`flexmeasures.api.common.utils.deprecation_utils`, 174
`flexmeasures.api.common.utils.migration_utils`, 176
`flexmeasures.api.common.utils.validators`, 177
`flexmeasures.api.dev`, 182
`flexmeasures.api.dev.sensors`, 181
`flexmeasures.api.play`, 184
`flexmeasures.api.play.implementations`, 183
`flexmeasures.api.play.routes`, 183
`flexmeasures.api.sunset`, 184
`flexmeasures.api.sunset.routes`, 184
`flexmeasures.api.v3_0`, 200
`flexmeasures.api.v3_0.accounts`, 185
`flexmeasures.api.v3_0.assets`, 187
`flexmeasures.api.v3_0.health`, 191
`flexmeasures.api.v3_0.public`, 191
`flexmeasures.api.v3_0.sensors`, 191
`flexmeasures.api.v3_0.users`, 197
`flexmeasures.app`, 201
`flexmeasures.auth`, 204
`flexmeasures.auth.decorators`, 201
`flexmeasures.auth.error_handling`, 203
`flexmeasures.auth.policy`, 203
`flexmeasures.cli`, 208
`flexmeasures.cli.data_add`, 205
`flexmeasures.cli.data_delete`, 205
`flexmeasures.cli.data_edit`, 205
`flexmeasures.cli.data_show`, 205
`flexmeasures.cli.db_ops`, 206
`flexmeasures.cli.jobs`, 206
`flexmeasures.cli.monitor`, 206
`flexmeasures.cli.utils`, 206
`flexmeasures.data`, 286
`flexmeasures.data.config`, 209
`flexmeasures.data.models`, 249
`flexmeasures.data.models.annotations`, 210
`flexmeasures.data.models.assets`, 212
`flexmeasures.data.models.charts`, 215
`flexmeasures.data.models.charts.belief_charts`, 214
`flexmeasures.data.models.charts.defaults`, 215
`flexmeasures.data.models.data_sources`, 215
`flexmeasures.data.models.forecasting`, 221
`flexmeasures.data.models.forecasting.exceptions`, 216
`flexmeasures.data.models.forecasting.model_spec_factor`, 216
`flexmeasures.data.models.forecasting.model_specs`, 219
`flexmeasures.data.models.forecasting.model_specs.linear`, 219
`flexmeasures.data.models.forecasting.model_specs.naive`, 219
`flexmeasures.data.models.forecasting.utils`, 220
`flexmeasures.data.models.generic_assets`, 221
`flexmeasures.data.models.legacy_migration_utils`, 225
`flexmeasures.data.models.markets`, 226
`flexmeasures.data.models.parsing_utils`, 227
`flexmeasures.data.models.planning`, 235
`flexmeasures.data.models.planning.battery`, 228
`flexmeasures.data.models.planning.charging_station`, 228
`flexmeasures.data.models.planning.exceptions`, 228
`flexmeasures.data.models.planning.linear_optimization`, 229
`flexmeasures.data.models.planning.storage`, 230
`flexmeasures.data.models.planning.utils`, 234
`flexmeasures.data.models.reporting`, 238
`flexmeasures.data.models.reporting.aggregator`, 236
`flexmeasures.data.models.reporting.pandas_reporter`, 237
`flexmeasures.data.models.task_runs`, 239
`flexmeasures.data.models.time_series`, 239

- `flexmeasures.data.models.user`, 245
 - `flexmeasures.data.models.validation_utils`, 247
 - `flexmeasures.data.models.weather`, 248
 - `flexmeasures.data.queries`, 256
 - `flexmeasures.data.queries.analytics`, 250
 - `flexmeasures.data.queries.annotations`, 251
 - `flexmeasures.data.queries.data_sources`, 251
 - `flexmeasures.data.queries.generic_assets`, 251
 - `flexmeasures.data.queries.portfolio`, 252
 - `flexmeasures.data.queries.sensors`, 253
 - `flexmeasures.data.queries.utils`, 254
 - `flexmeasures.data.schemas`, 268
 - `flexmeasures.data.schemas.account`, 257
 - `flexmeasures.data.schemas.assets`, 257
 - `flexmeasures.data.schemas.generic_assets`, 258
 - `flexmeasures.data.schemas.reporting`, 263
 - `flexmeasures.data.schemas.reporting.aggregation`, 260
 - `flexmeasures.data.schemas.reporting.pandas_reporting`, 261
 - `flexmeasures.data.schemas.scheduling`, 264
 - `flexmeasures.data.schemas.scheduling.storage`, 264
 - `flexmeasures.data.schemas.sensors`, 265
 - `flexmeasures.data.schemas.sources`, 265
 - `flexmeasures.data.schemas.times`, 266
 - `flexmeasures.data.schemas.units`, 266
 - `flexmeasures.data.schemas.users`, 267
 - `flexmeasures.data.schemas.utils`, 267
 - `flexmeasures.data.scripts`, 269
 - `flexmeasures.data.scripts.data_gen`, 269
 - `flexmeasures.data.scripts.visualize_data_model`, 269
 - `flexmeasures.data.services`, 284
 - `flexmeasures.data.services.accounts`, 270
 - `flexmeasures.data.services.annotations`, 270
 - `flexmeasures.data.services.asset_grouping`, 271
 - `flexmeasures.data.services.data_sources`, 272
 - `flexmeasures.data.services.forecasting`, 273
 - `flexmeasures.data.services.resources`, 274
 - `flexmeasures.data.services.scheduling`, 279
 - `flexmeasures.data.services.sensors`, 280
 - `flexmeasures.data.services.time_series`, 281
 - `flexmeasures.data.services.timerange`, 282
 - `flexmeasures.data.services.users`, 283
 - `flexmeasures.data.services.utils`, 284
 - `flexmeasures.data.transactional`, 285
 - `flexmeasures.data.utils`, 285
 - `flexmeasures.ui`, 293
 - `flexmeasures.ui.crud`, 289
 - `flexmeasures.ui.crud.accounts`, 287
 - `flexmeasures.ui.crud.api_wrapper`, 287
 - `flexmeasures.ui.crud.assets`, 288
 - `flexmeasures.ui.crud.users`, 289
 - `flexmeasures.ui.error_handlers`, 290
 - `flexmeasures.ui.utils`, 291
 - `flexmeasures.ui.utils.chart_defaults`, 290
 - `flexmeasures.ui.utils.view_utils`, 290
 - `flexmeasures.ui.views`, 293
 - `flexmeasures.ui.views.control`, 292
 - `flexmeasures.ui.views.logged_in_user`, 292
 - `flexmeasures.ui.views.new_dashboard`, 292
 - `flexmeasures.ui.views.sensors`, 293
 - `flexmeasures.utils`, 307
 - `flexmeasures.utils.app_utils`, 294
 - `flexmeasures.utils.calculations`, 295
 - `flexmeasures.utils.coding_utils`, 296
 - `flexmeasures.utils.config_defaults`, 298
 - `flexmeasures.utils.config_utils`, 298
 - `flexmeasures.utils.entity_address_utils`, 299
 - `flexmeasures.utils.error_utils`, 300
 - `flexmeasures.utils.flexmeasures_inflection`, 301
 - `flexmeasures.utils.geo_utils`, 301
 - `flexmeasures.utils.grid_cells`, 302
 - `flexmeasures.utils.plugin_utils`, 303
 - `flexmeasures.utils.time_utils`, 304
 - `flexmeasures.utils.unit_utils`, 306
 - `MsgStyle` (class in `flexmeasures.cli.utils`), 208
 - `multiply_dataframe_with_deterministic_beliefs()` (in module `flexmeasures.data.queries.utils`), 255
- ## N
- `Naive` (class in `flexmeasures.data.models.forecasting.model_specs.naive`), 219
 - `naive_specs_configurator()` (in module `flexmeasures.data.models.forecasting.model_specs.naive`), 219
 - `naive_utc_from()` (in module `flexmeasures.utils.time_utils`), 305
 - `name` (`flexmeasures.data.models.assets.Asset` attribute), 213
 - `name` (`flexmeasures.data.models.data_sources.DataSource` attribute), 216

name (*flexmeasures.data.models.markets.Market* attribute), 227

name (*flexmeasures.data.models.time_series.Sensor* attribute), 241

name (*flexmeasures.data.models.weather.WeatherSensor* attribute), 249

name (*flexmeasures.data.schemas.utils.MarshmallowClickMixin* attribute), 268

naturalized_datetime_str() (in module *flexmeasures.utils.time_utils*), 305

NewAssetForm (class in *flexmeasures.ui.crud.assets*), 289

no_backup() (in module *flexmeasures.api.common.responses*), 163

no_message_type() (in module *flexmeasures.api.common.responses*), 163

NotEnoughDataException, 216

num_forecasts() (in module *flexmeasures.data.services.forecasting*), 274

O

ols_specs_configurator() (in module *flexmeasures.data.models.forecasting.model_specs.linear_regression*), 219

optional_arg_decorator() (in module *flexmeasures.utils.coding_utils*), 297

optional_duration_accepted() (in module *flexmeasures.api.common.utils.validators*), 177

optional_horizon_accepted() (in module *flexmeasures.api.common.utils.validators*), 177

optional_prior_accepted() (in module *flexmeasures.api.common.utils.validators*), 178

optional_user_sources_accepted() (in module *flexmeasures.api.common.utils.validators*), 179

opts (*flexmeasures.data.schemas.account.AccountRoleSchema* attribute), 257

opts (*flexmeasures.data.schemas.account.AccountSchema* attribute), 257

opts (*flexmeasures.data.schemas.assets.AssetSchema* attribute), 257

opts (*flexmeasures.data.schemas.generic_assets.GenericAssetSchema* attribute), 258

opts (*flexmeasures.data.schemas.generic_assets.GenericAssetTypeSchema* attribute), 258

opts (*flexmeasures.data.schemas.sensors.SensorSchema* attribute), 265

opts (*flexmeasures.data.schemas.users.UserSchema* attribute), 267

outdated_event_id() (in module *flexmeasures.api.common.responses*), 163

override_from_config() (in module *flexmeasures.api.common.utils.deprecation_utils*), 176

owned_by() (*flexmeasures.ui.crud.assets.AssetCrudUI* method), 288

P

PandasMethodCall (class in *flexmeasures.data.schemas.reporting.pandas_reporter*), 261

PandasReporter (class in *flexmeasures.data.models.reporting.pandas_reporter*), 237

PandasReporterConfigSchema (class in *flexmeasures.data.schemas.reporting.pandas_reporter*), 261

parameterize() (in module *flexmeasures.utils.flexmeasures_inflection*), 301

parameterized_name (*flexmeasures.data.services.asset_grouping.AssetGroup* property), 272

parameterized_name (*flexmeasures.data.services.resources.Resource* property), 278

parse_as_list() (in module *flexmeasures.api.common.utils.api_utils*), 172

parse_attribute_value() (in module *flexmeasures.cli.data_edit*), 205

parse_config_entry_by_account_roles() (in module *flexmeasures.utils.app_utils*), 294

parse_duration() (in module *flexmeasures.api.common.utils.validators*), 179

parse_entity_address() (in module *flexmeasures.utils.entity_address_utils*), 299

parse_horizon() (in module *flexmeasures.api.common.utils.validators*), 179

parse_isodate_str() (in module *flexmeasures.api.common.utils.validators*), 179

parse_lat_lng() (in module *flexmeasures.utils.geo_utils*), 301

parse_queue_list() (in module *flexmeasures.cli.jobs*), 206

parse_source() (in module *flexmeasures.cli.data_add*), 205

parse_source_arg() (in module *flexmeasures.data.models.parsing_utils*), 227

PartialTaskCompletionException, 285

patch() (*flexmeasures.api.v3_0.assets.AssetAPI* method), 189

patch() (*flexmeasures.api.v3_0.users.UserAPI* method), 199

period_required() (in module *flexmeasures.api.common.utils.validators*), 179

permission_required_for_context() (in module *flexmeasures.auth.decorators*), 202

persist_flex_model() (*flexmeasures.data.models.planning.Scheduler* method),

- 236
- `persist_flex_model()` (flexmeasures.data.models.planning.storage.StorageSchedule method), 233
- `ping()` (in module flexmeasures.api.common.implementations), 162
- `PlanningDurationField` (class in flexmeasures.data.schemas.times), 266
- `pluralize()` (in module flexmeasures.api.common.responses), 163
- `pluralize()` (in module flexmeasures.utils.flexmeasures_inflection), 301
- `possibly_convert_units()` (flexmeasures.api.common.schemas.sensor_data.PostSensorDataSchema static method), 166
- `possibly_extend_end()` (flexmeasures.data.models.planning.storage.StorageSchedule method), 233
- `possibly_upsample_values()` (flexmeasures.api.common.schemas.sensor_data.PostSensorDataSchema static method), 166
- `post()` (flexmeasures.api.v3_0.assets.AssetAPI method), 189
- `post()` (flexmeasures.ui.crud.assets.AssetCrudUI method), 288
- `post_data()` (flexmeasures.api.v3_0.sensors.SensorAPI method), 194
- `post_data_checked_for_required_resolution()` (in module flexmeasures.api.common.utils.validators), 180
- `post_load_sequence()` (flexmeasures.api.common.schemas.sensor_data.PostSensorDataSchema method), 167
- `post_load_sequence()` (flexmeasures.data.schemas.scheduling.storage.StorageFlexModelSchema method), 264
- `post_task_run()` (in module flexmeasures.api.common.implementations), 162
- `post_task_run()` (in module flexmeasures.api.common.routes), 164
- `PostSensorDataSchema` (class in flexmeasures.api.common.schemas.sensor_data), 166
- `potentially_limit_assets_query_to_account()` (in module flexmeasures.data.queries.utils), 255
- `Power` (class in flexmeasures.data.models.assets), 213
- `power_unit` (flexmeasures.data.models.assets.Asset property), 213
- `power_value_too_big()` (in module flexmeasures.api.common.responses), 163
- `power_value_too_small()` (in module flexmeasures.api.common.responses), 163
- `preconditions` (flexmeasures.data.models.assets.AssetType property), 213
- `preconditions` (flexmeasures.data.models.markets.MarketType property), 227
- `prepare_annotations_for_chart()` (in module flexmeasures.data.services.annotations), 270
- `prepend_serie()` (in module flexmeasures.data.models.planning.storage), 231
- `Price` (class in flexmeasures.data.models.markets), 227
- `price_unit` (flexmeasures.data.models.markets.Market property), 227
- `print_query()` (in module flexmeasures.utils.error_utils), 300
- `process_api_validation_errors()` (flexmeasures.ui.crud.assets.AssetForm method), 288
- `process_internal_api_response()` (in module flexmeasures.ui.crud.assets), 288
- `process_internal_api_response()` (in module flexmeasures.ui.crud.users), 289
- `ProductionConfig` (class in flexmeasures.utils.config_defaults), 298
- `ptus_incomplete()` (in module flexmeasures.api.common.responses), 163
- `public()` (flexmeasures.api.v3_0.assets.AssetAPI method), 190
- ## Q
- `QuantityField` (class in flexmeasures.data.schemas.units), 266
- `QuantityValidator` (class in flexmeasures.data.schemas.units), 267
- `query_asset_annotations()` (in module flexmeasures.data.queries.annotations), 251
- `query_assets_by_type()` (in module flexmeasures.data.queries.generic_assets), 252
- `query_sensor_by_name_and_generic_asset_type_name()` (in module flexmeasures.data.queries.sensors), 253
- `query_sensors_by_proximity()` (in module flexmeasures.data.queries.sensors), 253
- `query_time_series_data()` (in module flexmeasures.data.services.time_series), 282
- `quickref_directive()` (in module flexmeasures.api.v3_0.public), 191
- ## R
- `rad_lng()` (in module flexmeasures.utils.geo_utils), 301
- `read_config()` (in module flexmeasures.utils.config_utils), 298
- `read_custom_config()` (in module flexmeasures.utils.config_utils), 298
- `read_env_vars()` (in module flexmeasures.utils.config_utils), 298

- `record_run()` (*flexmeasures.data.models.task_runs.LatestTaskRun static method*), 239
- `register_at()` (*in module flexmeasures.api*), 201
- `register_at()` (*in module flexmeasures.api.common*), 181
- `register_at()` (*in module flexmeasures.api.dev*), 183
- `register_at()` (*in module flexmeasures.api.play*), 184
- `register_at()` (*in module flexmeasures.api.sunset*), 184
- `register_at()` (*in module flexmeasures.api.v3_0*), 200
- `register_at()` (*in module flexmeasures.auth*), 204
- `register_at()` (*in module flexmeasures.cli*), 208
- `register_at()` (*in module flexmeasures.data*), 286
- `register_at()` (*in module flexmeasures.ui*), 293
- `register_plugins()` (*in module flexmeasures.utils.plugin_utils*), 303
- `register_rq_dashboard()` (*in module flexmeasures.ui*), 293
- `remember_last_seen()` (*in module flexmeasures.data.models.user*), 245
- `remember_login()` (*in module flexmeasures.data.models.user*), 245
- `remove_cookie_and_token_access()` (*in module flexmeasures.data.services.users*), 283
- `render_flexmeasures_template()` (*in module flexmeasures.ui.utils.view_utils*), 291
- `render_user()` (*in module flexmeasures.ui.crud.users*), 289
- `Reporter` (*class in flexmeasures.data.models.reporting*), 238
- `ReporterConfigSchema` (*class in flexmeasures.data.schemas.reporting*), 263
- `request_auth_token()` (*in module flexmeasures.api*), 201
- `request_processed()` (*in module flexmeasures.api.common.responses*), 163
- `required_info_missing()` (*in module flexmeasures.api.common.responses*), 163
- `reset_db()` (*in module flexmeasures.data.scripts.data_gen*), 269
- `reset_password_for()` (*flexmeasures.ui.crud.users.UserCrudUI method*), 289
- `reset_user_password()` (*flexmeasures.api.v3_0.users.UserAPI method*), 200
- `resolution_to_hour_factor()` (*in module flexmeasures.utils.time_utils*), 306
- `Resource` (*class in flexmeasures.data.services.resources*), 276
- `restore_data()` (*in module flexmeasures.api.play.routes*), 183
- `restore_data_response()` (*in module flexmeasures.api.play.implementations*), 183
- `revenue` (*flexmeasures.data.services.resources.Resource property*), 278
- `reverse_domain_name()` (*in module flexmeasures.utils.entity_address_utils*), 300
- `rgetattr()` (*in module flexmeasures.utils.coding_utils*), 297
- `Role` (*class in flexmeasures.data.models.user*), 246
- `roles` (*flexmeasures.data.models.user.User attribute*), 247
- `roles_accepted()` (*in module flexmeasures.auth.decorators*), 203
- `roles_required()` (*in module flexmeasures.auth.decorators*), 203
- `RolesAccounts` (*class in flexmeasures.data.models.user*), 246
- `RolesUsers` (*class in flexmeasures.data.models.user*), 246
- `root_dispatcher()` (*in module flexmeasures.utils.app_utils*), 294
- `round_to_closest_hour()` (*in module flexmeasures.utils.time_utils*), 306
- `round_to_closest_quarter()` (*in module flexmeasures.utils.time_utils*), 306
- ## S
- `sanitize_expression()` (*in module flexmeasures.data.models.planning.storage*), 231
- `save_and_enqueue()` (*in module flexmeasures.api.common.utils.api_utils*), 172
- `save_tables()` (*in module flexmeasures.data.scripts.data_gen*), 269
- `save_to_db()` (*in module flexmeasures.api.common.utils.api_utils*), 173
- `save_to_db()` (*in module flexmeasures.data.utils*), 285
- `save_to_session()` (*in module flexmeasures.data.utils*), 286
- `schedule_battery()` (*in module flexmeasures.data.models.planning.battery*), 228
- `schedule_charging_station()` (*in module flexmeasures.data.models.planning.charging_station*), 228
- `Scheduler` (*class in flexmeasures.data.models.planning*), 235
- `search()` (*flexmeasures.data.models.time_series.TimedBelief class method*), 243
- `search()` (*flexmeasures.data.models.time_series.TimedValue class method*), 245
- `search_annotations()` (*flexmeasures.data.models.generic_assets.GenericAsset method*), 223
- `search_annotations()` (*flexmeasures.data.models.time_series.Sensor method*), 241

- `search_annotations()` (*flexmeasures.data.models.user.Account* method), 246
- `search_beliefs()` (*flexmeasures.data.models.generic_assets.GenericAsset* method), 223
- `search_beliefs()` (*flexmeasures.data.models.time_series.Sensor* method), 241
- `select_schema_to_ensure_list_of_floats()` (in module *flexmeasures.api.common.schemas.sensor_data*), 165
- `send_lastseen_monitoring_alert()` (in module *flexmeasures.cli.monitor*), 206
- `send_task_monitoring_alert()` (in module *flexmeasures.cli.monitor*), 206
- `Sensor` (class in *flexmeasures.data.models.time_series*), 239
- `sensor_id` (*flexmeasures.data.models.time_series.TimedBelief* attribute), 244
- `SensorAnnotationRelationship` (class in *flexmeasures.data.models.annotations*), 212
- `SensorAPI` (class in *flexmeasures.api.dev.sensors*), 182
- `SensorAPI` (class in *flexmeasures.api.v3_0.sensors*), 191
- `SensorDataDescriptionSchema` (class in *flexmeasures.api.common.schemas.sensor_data*), 167
- `SensorField` (class in *flexmeasures.api.common.schemas.sensors*), 168
- `SensorIdField` (class in *flexmeasures.api.common.schemas.sensors*), 168
- `SensorIdField` (class in *flexmeasures.data.schemas.sensors*), 265
- `sensors_to_show` (*flexmeasures.data.models.generic_assets.GenericAsset* property), 224
- `SensorSchema` (class in *flexmeasures.data.schemas.sensors*), 265
- `SensorSchema.Meta` (class in *flexmeasures.data.schemas.sensors*), 265
- `SensorSchemaMixin` (class in *flexmeasures.data.schemas.sensors*), 265
- `SensorUI` (class in *flexmeasures.ui.views.sensors*), 293
- `server_now()` (in module *flexmeasures.utils.time_utils*), 306
- `ServicesAPI` (class in *flexmeasures.api.v3_0.public*), 191
- `set_bdf_source()` (in module *flexmeasures.data.services.time_series*), 282
- `set_individual_traces_for_session()` (in module *flexmeasures.ui.utils.view_utils*), 291
- `set_random_password()` (in module *flexmeasures.data.services.users*), 283
- `set_secret_key()` (in module *flexmeasures.utils.app_utils*), 294
- `set_session_market()` (in module *flexmeasures.ui.utils.view_utils*), 291
- `set_session_resource()` (in module *flexmeasures.ui.utils.view_utils*), 291
- `set_session_sensor_type()` (in module *flexmeasures.ui.utils.view_utils*), 291
- `set_time_range_for_session()` (in module *flexmeasures.ui.utils.view_utils*), 291
- `set_training_and_testing_dates()` (in module *flexmeasures.data.models.forecasting.utils*), 220
- `show_image()` (in module *flexmeasures.data.scripts.visualize_data_model*), 269
- `simplify_index()` (in module *flexmeasures.data.queries.utils*), 255
- `sin_rad_lat()` (in module *flexmeasures.utils.geo_utils*), 301
- `single_true()` (in module *flexmeasures.cli.data_edit*), 205
- `SingleValueField` (class in *flexmeasures.api.common.schemas.sensor_data*), 167
- `SOCValueSchema` (class in *flexmeasures.data.schemas.scheduling.storage*), 264
- `sort_dict()` (in module *flexmeasures.utils.coding_utils*), 297
- `source_id` (*flexmeasures.data.models.time_series.TimedBelief* attribute), 244
- `source_type_criterion()` (in module *flexmeasures.data.queries.utils*), 256
- `source_type_exclusion_criterion()` (in module *flexmeasures.data.queries.utils*), 256
- `split_response()` (in module *flexmeasures.api.common.utils.decorators*), 174
- `stack_annotations()` (in module *flexmeasures.data.services.annotations*), 271
- `StagingConfig` (class in *flexmeasures.utils.config_defaults*), 298
- `StorageFlexModelSchema` (class in *flexmeasures.data.schemas.scheduling.storage*), 264
- `StorageScheduler` (class in *flexmeasures.data.models.planning.storage*), 233
- `sunset_blueprint()` (in module *flexmeasures.api.common.utils.deprecation_utils*), 176
- `supply` (*flexmeasures.data.services.resources.Resource* property), 278
- `supported_horizons()` (in module *flexmeasures.utils.time_utils*), 306

T

- TBSeriesSpecs (class in flexmeasures.data.models.forecasting.model_spec_factory), 218
- TestingConfig (class in flexmeasures.utils.config_defaults), 298
- TimedBelief (class in flexmeasures.data.models.time_series), 242
- timedelta_to_pandas_freq_str() (in module flexmeasures.utils.time_utils), 306
- TimedValue (class in flexmeasures.data.models.time_series), 244
- timeit() (in module flexmeasures.utils.coding_utils), 297
- timerange (flexmeasures.data.models.generic_assets.GenericAsset property), 224
- timerange (flexmeasures.data.models.time_series.Sensor property), 242
- timerange_of_sensors_to_show (flexmeasures.data.models.generic_assets.GenericAsset property), 224
- timezone (flexmeasures.data.models.assets.Asset attribute), 213
- timezone (flexmeasures.data.models.generic_assets.GenericAsset property), 225
- timezone (flexmeasures.data.models.markets.Market attribute), 227
- timezone (flexmeasures.data.models.time_series.Sensor attribute), 242
- timezone (flexmeasures.data.models.weather.WeatherSensor attribute), 249
- titleize() (in module flexmeasures.utils.flexmeasures_inflection), 301
- to_annotation_frame() (in module flexmeasures.data.models.annotations), 210
- to_http_time() (in module flexmeasures.utils.time_utils), 306
- to_json() (flexmeasures.ui.crud.assets.AssetForm method), 288
- to_preferred() (in module flexmeasures.utils.unit_utils), 307
- toggle_active() (flexmeasures.ui.crud.users.UserCrudUI method), 289
- total_aggregate_demand (flexmeasures.data.services.resources.Resource property), 278
- total_aggregate_supply (flexmeasures.data.services.resources.Resource property), 278
- total_demand (flexmeasures.data.services.resources.Resource property), 278
- total_supply (flexmeasures.data.services.resources.Resource property), 279
- trigger_schedule() (flexmeasures.api.v3_0.sensors.SensorAPI method), 195
- type_accepted() (in module flexmeasures.api.common.utils.validators), 180
- tz_index_naively() (in module flexmeasures.utils.time_utils), 306

U

- unapplicable_resolution() (in module flexmeasures.api.common.responses), 163
- unauthenticated_handler() (in module flexmeasures.auth.error_handling), 203
- unauthenticated_handler() (in module flexmeasures.ui.error_handlers), 290
- unauthenticated_handler_e() (in module flexmeasures.auth.error_handling), 203
- unauthorized_handler() (in module flexmeasures.auth.error_handling), 203
- unauthorized_handler() (in module flexmeasures.ui.error_handlers), 290
- unauthorized_handler_e() (in module flexmeasures.auth.error_handling), 203
- unique_ever_seen() (in module flexmeasures.api.common.utils.api_utils), 173
- unit (flexmeasures.data.models.assets.Asset attribute), 213
- unit (flexmeasures.data.models.markets.Market attribute), 227
- unit (flexmeasures.data.models.time_series.Sensor attribute), 242
- unit (flexmeasures.data.models.weather.WeatherSensor attribute), 249
- unit_required() (in module flexmeasures.api.common.utils.validators), 180
- units_accepted() (in module flexmeasures.api.common.utils.validators), 180
- units_are_convertible() (in module flexmeasures.utils.unit_utils), 307
- unknown_prices() (in module flexmeasures.api.common.responses), 163
- unknown_schedule() (in module flexmeasures.api.common.responses), 163
- UnknownForecastException, 228
- UnknownMarketException, 228
- UnknownPricesException, 228
- unrecognized_asset() (in module flexmeasures.api.common.responses), 163
- unrecognized_backup() (in module flexmeasures.api.common.responses), 163
- unrecognized_connection_group() (in module flexmeasures.api.common.responses), 163

unrecognized_event() (in module *flexmeasures.api.common.responses*), 163
 unrecognized_event_type() (in module *flexmeasures.api.common.responses*), 163
 unrecognized_market() (in module *flexmeasures.api.common.responses*), 163
 unrecognized_sensor() (in module *flexmeasures.api.common.responses*), 163
 upsample_values() (in module *flexmeasures.api.common.utils.api_utils*), 173
 User (class in *flexmeasures.data.models.user*), 246
 user_can_create_assets() (in module *flexmeasures.ui.crud.assets*), 288
 user_can_delete() (in module *flexmeasures.ui.crud.assets*), 288
 user_has_admin_access() (in module *flexmeasures.auth.policy*), 204
 user_matches_principals() (in module *flexmeasures.auth.policy*), 204
 user_source_criterion() (in module *flexmeasures.data.queries.utils*), 256
 UserAPI (class in *flexmeasures.api.v3_0.users*), 197
 UserCrudUI (class in *flexmeasures.ui.crud.users*), 289
 UserForm (class in *flexmeasures.ui.crud.users*), 289
 UserIdField (class in *flexmeasures.api.common.schemas.users*), 169
 username() (in module *flexmeasures.ui.utils.view_utils*), 291
 UserSchema (class in *flexmeasures.data.schemas.users*), 267
 UserSchema.Meta (class in *flexmeasures.data.schemas.users*), 267
 uses_dot() (in module *flexmeasures.data.scripts.visualize_data_model*), 269

V

valid_sensor_units() (in module *flexmeasures.api.common.utils.validators*), 181
 validate_chaining() (flexmeasures.data.schemas.reporting.pandas_reporter.PandasReporterConfigSchema method), 263
 validate_constraint() (in module *flexmeasures.data.models.planning.storage*), 231
 validate_on_submit() (flexmeasures.ui.crud.assets.AssetForm method), 289
 validate_storage_constraints() (in module *flexmeasures.data.models.planning.storage*), 232
 validate_user_sources() (in module *flexmeasures.api.common.utils.validators*), 181
 validation_error_handler() (in module *flexmeasures.api.common.utils.args_parsing*), 173

W

Weather (class in *flexmeasures.data.models.weather*), 248
 weather_correlations (flexmeasures.data.models.assets.AssetType property), 213
 weather_unit (flexmeasures.data.models.weather.WeatherSensor property), 249
 WeatherSensor (class in *flexmeasures.data.models.weather*), 248
 WeatherSensorType (class in *flexmeasures.data.models.weather*), 249
 weighted_absolute_percentage_error() (in module *flexmeasures.utils.calculations*), 296
 with_appcontext_if_needed() (in module *flexmeasures.data.schemas.utils*), 267
 with_options() (in module *flexmeasures.ui.crud.assets*), 288
 WrongTypeAttributeException, 228, 247