FlexMeasures Documentation

Release 0.11.0

Seita B.V.

Aug 28, 2022

CONTENTS

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 CO_2 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*.

ONE

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 & a user, a battery (Id
↪2) and a market (Id 3)
$ flexmeasures add beliefs --sensor-id 3 --source toy-user prices-tomorrow.csv  # load
↪prices, also possible per API
$ flexmeasures add schedule --sensor-id 2 --consumption-price-sensor 3 \
    --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 2 --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, CO_2, 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:

THREE
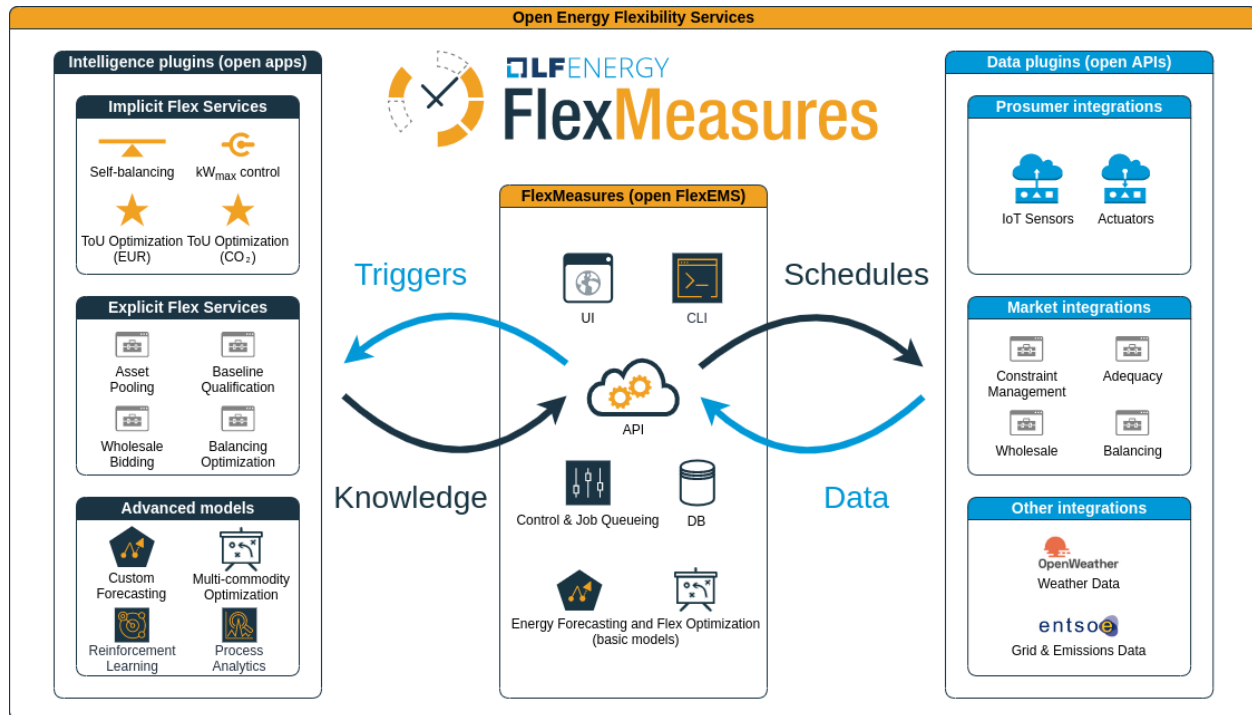
A POSSIBLE ROAD TO START USING FLEXMEASURES IN YOUR OPERATION

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!

FOUR

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!

FIVE

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

v0.11.0 | 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 anymore; 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 08, 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 08, 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 *fm1* 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.

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

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` in the API [see PR #25]
- Replace `solarpy` with `pvlib` (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 CO_2 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 CO_2.

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 CO_2 emissions. Then, the realised value is avoided CO_2, 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	Building	Charge Points	Solar	Wind (offshore)	Day-ahead market
Average power per asset	204 W	75 W	140 W	518 W	
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 CO_2 emissions are sometimes required. There are multiple ways to measure this, e.g. against the CO_2 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 *admin*s who can perform all actions, and *admin-reader*s 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, a battery (Id 2) and a market (Id 3)
$ flexmeasures add toy-account --kind battery
# load prices to optimise the schedule against
$ flexmeasures add beliefs --sensor-id 3 --source toy-user prices-tomorrow.csv
# make the schedule
$ flexmeasures add schedule --sensor-id 2 --consumption-price-sensor 3 \
    --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

On your PC

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"
```

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-C or type "exit".

To stop the containers, you can type

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

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.

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

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
$ 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.

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`
```

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

```
The sensor for battery charging is <Sensor 2: charging, unit: MW res.: 0:15:00>.
The sensor for Day ahead prices is <Sensor 3: Day ahead prices, unit: EUR/MWh res.:␣
→1:00:00>.
```

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
----  -----------  --------  ----------------
   3  toy-battery  battery   (52.374, 4.88969)
   2  toy-building  building  (52.374, 4.88969)
   1  toy-solar    solar     (52.374, 4.88969)

$ flexmeasures show asset --id 3

==========================
Asset toy-battery (ID:3):
==========================

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

All sensors in asset:

  Id  Name      Unit    Resolution    Timezone          Attributes
----  --------  ------  ------------  ----------------  ------------
   2  charging  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/assets (username is "toy-user@flexmeasures.io", password is "toy-password") and select "toy-battery":

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 3 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
```

(continues on next page)

```
$ ${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 3 --source toy-user prices-tomorrow.csv
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 will then interpret them as UTC time. So if you are in Amsterdam time, the start time for the first price, when expressed in your time zone, is actually *2022-03-03 01:00:00+01:00*.
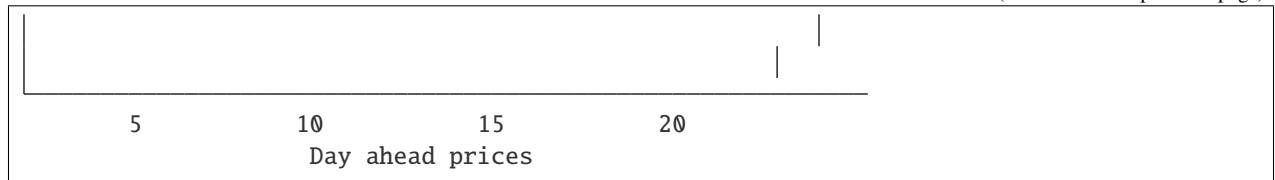

Let's look at the price data we just loaded:

```
$ flexmeasures show beliefs --sensor-id 3 --start ${TOMORROW}T01:00:00+01:00 --duration␣
↪PT24H
Beliefs for Sensor 'Day ahead prices' (Id 3).
Data spans a day and starts at 2022-03-03 01:00:00+01:00.
The time resolution (x-axis) is an hour.
```
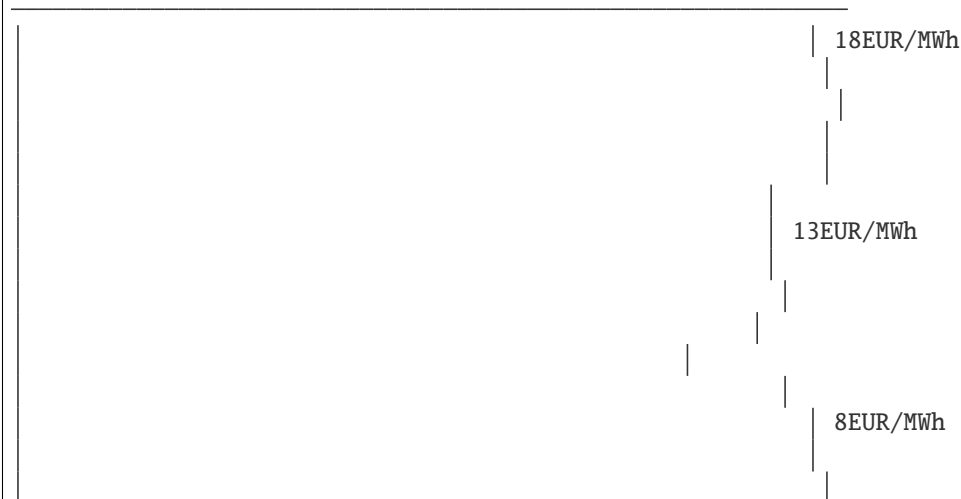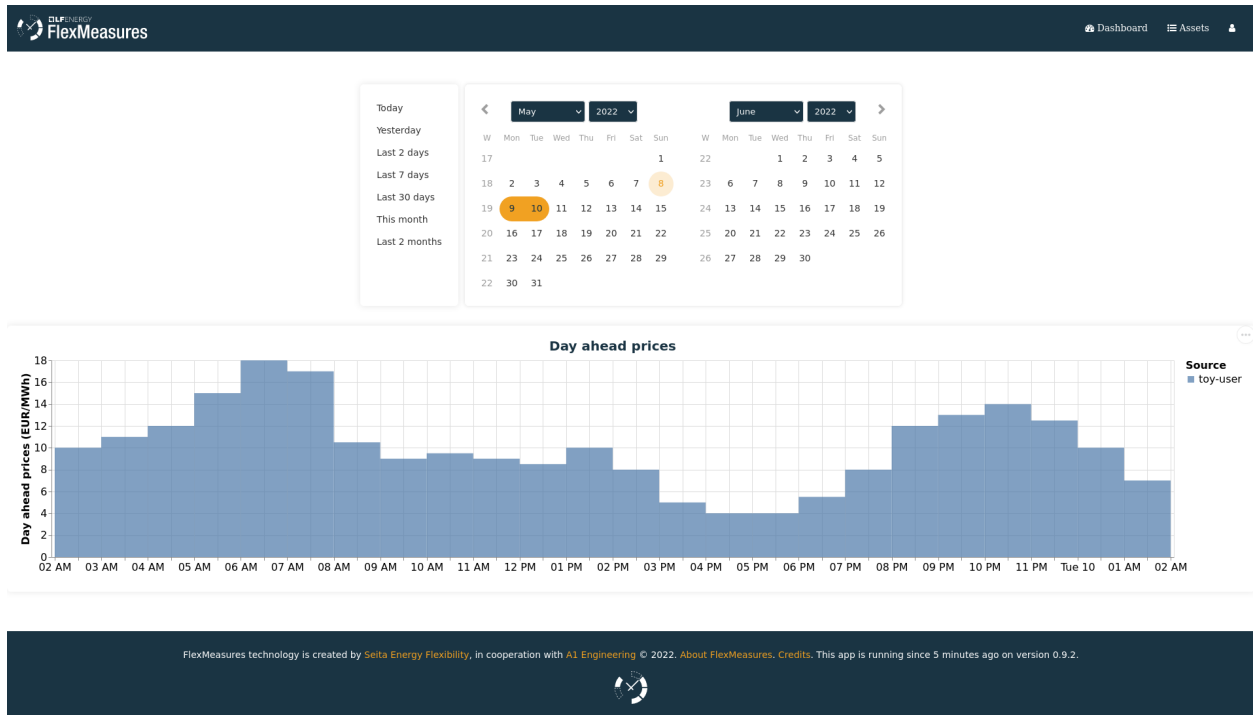

Again, we can also view these prices in the FlexMeasures UI:

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 charging sensor (Id 2). 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 --sensor-id 2 --consumption-price-sensor 3 \
    --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 2 --start ${TOMORROW}T07:00:00+01:00 --duration␣
↪PT12H
Beliefs for Sensor 'charging' (Id 2).
Data spans 12 hours and starts at 2022-03-04 07:00:00+01:00.
The time resolution (x-axis) is 15 minutes.
```
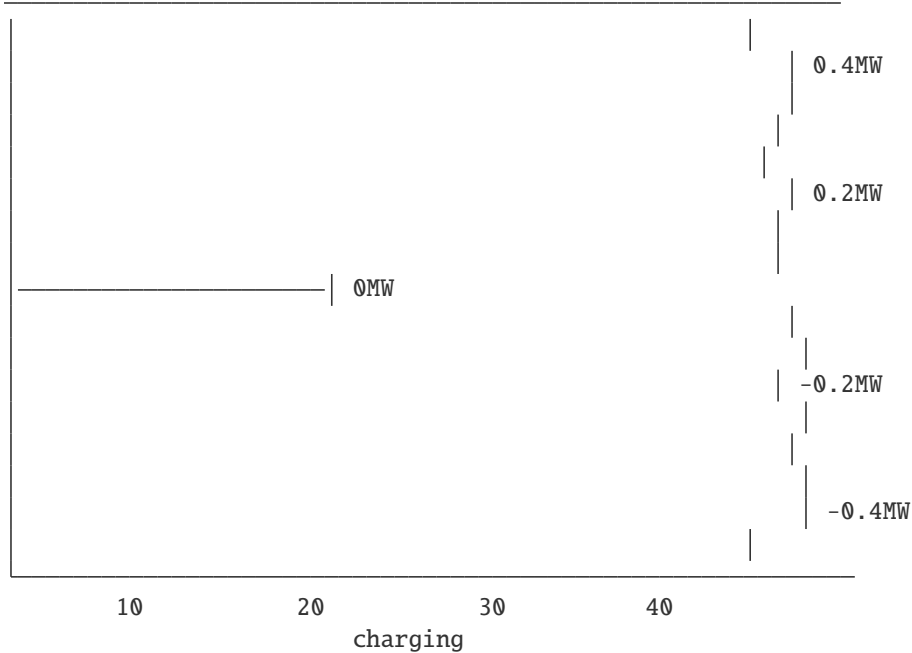

Here, negative values denote output from the grid, so that's when the battery gets charged.

We can also look at the charging schedule in the FlexMeasures UI (reachable via the asset page for the battery):
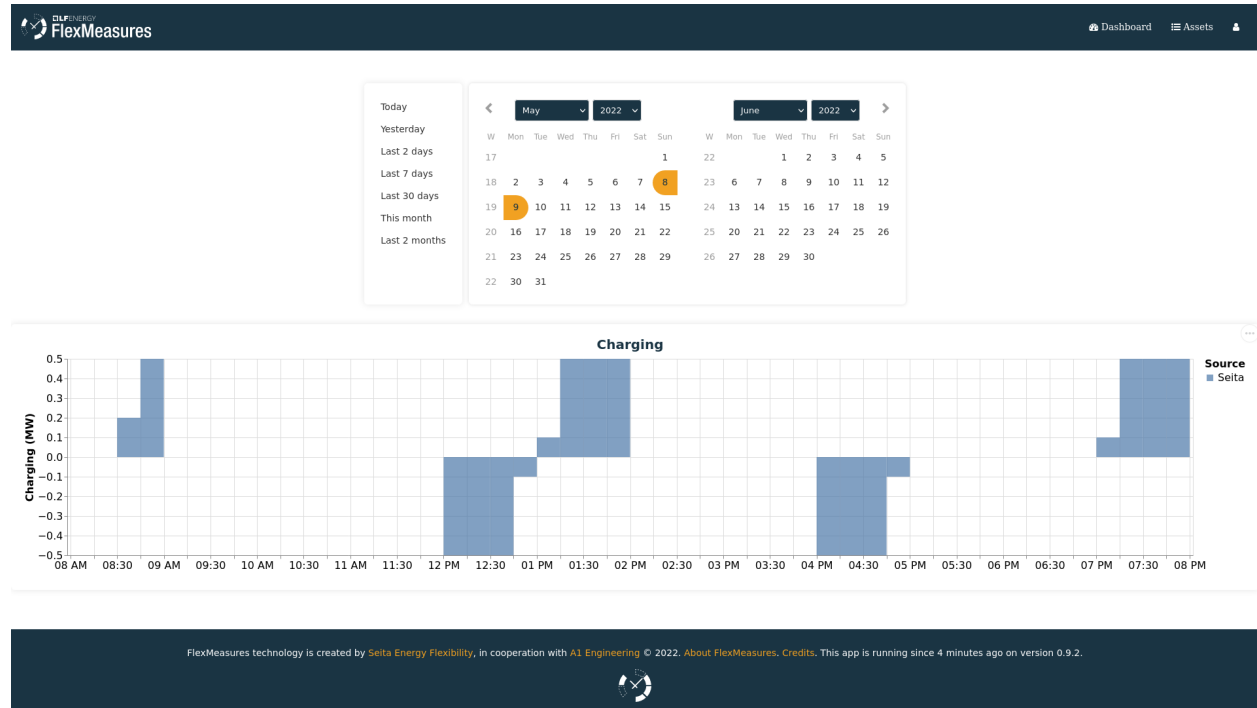
Recall that we only asked for a 12 hour schedule here. We started our schedule *after* the high price peak (at 5am) and it also had to end *before* the second price peak fully realised (at 9pm). Our scheduler didn't have many opportunities to optimize, but it found some. For instance, it does buy at the lowest price (around 3pm) and sells it off when prices start rising again (around 6pm).

Note: The `flexmeasures add schedule` 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 --help`.

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 _tut_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

- 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*, *Resolutions*, *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).

```json
{
    "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*):

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

```json
{
    "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.

```json
{
    "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.

```json
{
    "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.

Owners of such devices can post these states 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
```

This 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 allows to limit the flexibility range and also to set target values.

```
{
    "value": 12.1,
    "datetime": "2015-06-02T10:00:00+00:00",
    "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 not persisted. To record a 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 FlexMeasurers 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*.

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 scheduling scripts (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 example with an additional target value, representing a desired future state of charge.

```json
{
    "value": 12.1,
    "datetime": "2015-06-02T10:00:00+00:00",
    "unit": "kWh",
    "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 --sensor-id 2 --optimization-context-id 3 \
    --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 --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.

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

(continues on next page)

```
    "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**
>

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:

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

It only expects you to set `email` and `password` somewhere (you could also pass them in). 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?

Here is some code to find out and display that information in a simple HTML table:

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

```javascript
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);
        })
    }
}
```

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 user ID and we want to show which assets FlexMeasures administrates for that user.

```html
<table id="assetTable">
    <thead>
      <tr>
        <th>Asset name</th>
        <th>Type</th>
        <th>Capacity</th>
      </tr>
    </thead>
    <tbody></tbody>
</table>
```

```javascript
function loadAssets(userId, authToken) {
    var params = new URLSearchParams();
    params.append("owner_id", userId);
    fetch(flexmeasures_domain + '/api/v2_0/assets?' + params.toString(),
        {
            method: "GET",
            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>${row.display_name}</td><td>${row.asset_type_name}</td><td>$
↪{row.capacity_in_mw} MW</td>`;
        tbody.appendChild(tr);
```

```
        });
    })
}

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

The result looks like this in your browser:

Asset name	Type	Capacity
Gujwa-eup - PV	solar	0.66 MW
Millennium Seoul Hilton - charger 2	two-way_evse	0.05 MW
Millennium Seoul Hilton - charger 1	two-way_evse	0.05 MW
Jeju Haenyeo Museum - charger 2	one-way_evse	0.05 MW
Jeju Haenyeo Museum - charger 1	one-way_evse	0.05 MW
Tamra-owp - WT	wind	200 MW

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 here (`owner_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.

Embedding plots

Creating plots from data can consume lots of development time. FlexMeasures can help here by delivering ready-made plots.

In this tutorial, let's display two plots: one with power measurements and forecasts (a solar panel installation) and one with schedules of several EV chargers on the same location, next to each other for easy comparison.

First, we define two div tags for the two plots and a basic layout for them. We also load the Bokeh library, more about that below.

```
<style>
    #flexbox {
        display: flex;
    }
```

```
    #plot-div1, #plot-div2 {
        height: 450px;
        width: 450px;
        border: 1px solid grey;
    }
    /* a fix we have to do if we position absolutely-positioned Bokeh plots in a flexbox␣
→design */
    .bk-plot-layout, .bk-plot-wrapper {
        position: relative !important;
    }
</style>
```

```
<script src="https://cdn.pydata.org/bokeh/release/bokeh-1.0.4.min.js"></script>
<div id="flexbox">
    <div id="plot-div1"></div>
    <div id="plot-div2"></div>
</div>
```

Now we define a JavaScript function to ask the FlexMeasures API for a plot:

```
function renderPlot(params, authToken, divId){
    fetch(flexmeasures_domain + '/api/v2_0/charts/power?' + params.toString(),
        {
            method: "GET",
            mode: "cors",
            headers:
                {
                "Content-Type": "application/json",
                "Authorization": authToken
                },
        }
    )
    .then(function(response) { return response.json(); })
    .then(function(item) { Bokeh.embed.embed_item(item, divId); })
    .then(console.log("Got plot specifications from server and rendered it ..."))
}
```

This function allows us to request a plot (actually, HTML and JavaScript code to render a plot), and then render the plot within a `div` tag of our choice.

As FlexMeasures uses the Bokeh Visualization Library internally, we also need to import the Bokeh client library to render the plots (see the `script` tag above). It's crucial to note that FlexMeasures is not transferring images across HTTP here, just information needed to render them.

Note: The Bokeh library version you use in your frontend needs to match the version which FlexMeasures uses internally, check `requirements/app.txt` when in doubt.

Now let's call this function when the HTML page is opened, to load our two plots:

```
document.onreadystatechange = () => {
    if (document.readyState === 'complete') {
        getAuthToken()
```

```javascript
        .then(function(response) {
            var authToken = response.auth_token;

            var urlData1 = new URLSearchParams();
            urlData1.append("resource", "ss_pv");
            urlData1.append("start_time", "2015-06-01T10:00:00");
            urlData1.append("end_time", "2015-06-03T10:00:00");
            urlData1.append("resolution", "PT15M");
            urlData1.append("forecast_horizon", "PT6H");
            urlData1.append("show_individual_traces_for", "none");
            renderPlot(urlData1, authToken, "plot-div1");

            var urlData2 = new URLSearchParams();
            urlData2.append("resource", "Test station (Charge Point)");
            urlData2.append("start_time", "2015-01-01T00:00:00");
            urlData2.append("end_time", "2015-01-01T03:00:00");
            urlData2.append("resolution", "PT15M");
            urlData2.append("show_individual_traces_for", "schedules");
            renderPlot(urlData2, authToken, "plot-div2");
        })
    }
}
```

For each of the two plots we request, we pass in several query parameters to describe what we want to see. We define which asset and what time range, which resolution and forecasting horizon. Note the `show_individual_traces_for` setting - it allows us to split data from individual assets (usually measurements, forecasts and schedules are visually aggregated in FlexMeasure's power plots, see analytics for example).

The result looks like this in your browser:
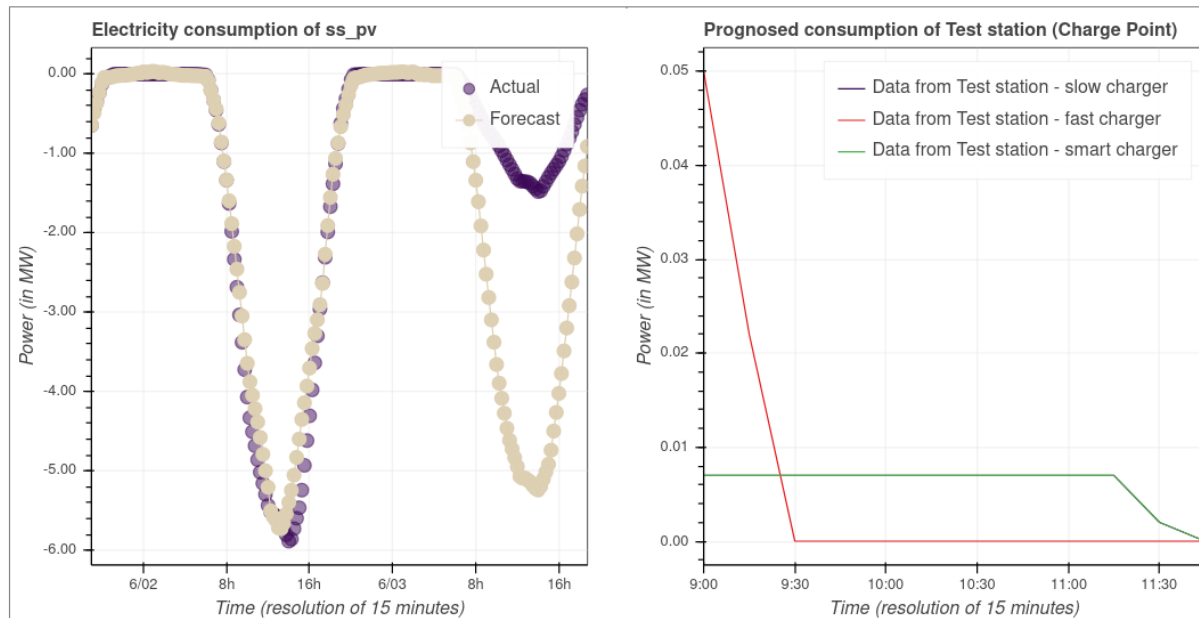
From FlexMeasures, we are using the [GET] /charts/power endpoint, which loads HTML and JavaScript. Browse the endpoint documentation to learn more about it.

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*
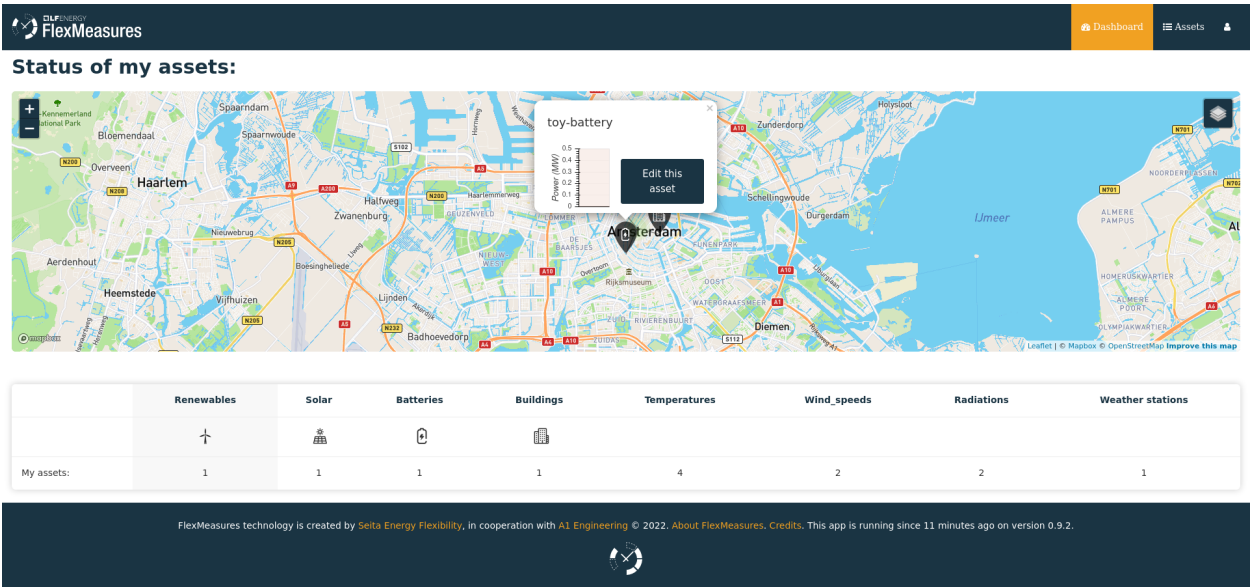
Interactive map of assets

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.

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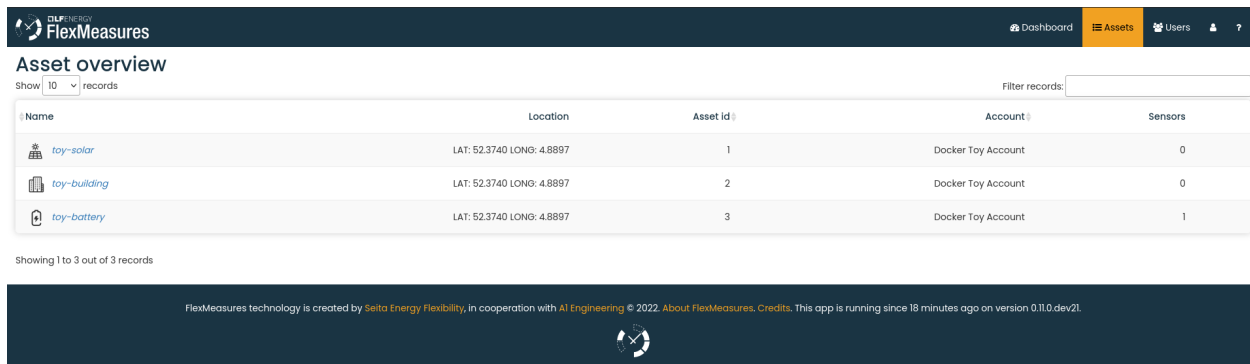
Note: Asset attributes can be edited through the CLI as well, with the CLI command `flexmeasures edit attribute`.

5.3.16 Administration

The administrator can see assets and users here.

Assets

Listing all assets:

Users

Listing all users:

Viewing one user:

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")
>>> 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': ['v1', 'v1_1', 'v1_2', 'v1_3', 'v2_0', '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.

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 auth.

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>
```

Todo: UDI events are not yet modelled in the fm1 scheme

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 for the most part and is no longer supported officially. Only UDI events still need to be sent using the fm0 scheme.

```
ea1.2021-01.io.flexmeasures:fm0.40:30:302:soc
ea1.2021-01.io.flexmeasures:fm0.<owner_id>:<sensor_id>:<event_id>:<event_type>
```

Timeseries

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

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:

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

Technically, this is equal to:

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

(continues on next page)

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

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

Notation for v1, v2 and v3

For version 1, 2 and 3 of the 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.

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": [
        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.

Resolutions

Specifying a resolution is redundant for POST requests that contain both "values" and a "duration" — FlexMeasures computes the resolution by dividing the duration by the number of values.

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

GET requests (such as *getMeterData*) return data in the resolution which the sensor is configured for. A "resolution" may be specified explicitly to obtain the data in downsampled form, which can be very beneficial for download speed. 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.

5.3.19 Sources

Requests for data may limit the data selection by specifying a source, for example, a specific user. Account roles are also valid source selectors. For example, to obtain data originating from either a meter data company or user 42, include the following:

```
{
    "sources": ["MDC", "42"],
}
```

Here, "MDC" is the name of the account role for meter data companies.

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.

5.3.20 Version 3.0

Summary

Resource	Operation	Description
Asset	GET /api/v3_0/assets	Download asset list
	POST /api/v3_0/assets	Create a new asset
	DELETE /api/v3_0/assets/(id)	Delete an asset
	GET /api/v3_0/assets/(id)	Get an asset
	PATCH /api/v3_0/assets/(id)	Update an asset
	GET /api/v3_0/assets/public	Return all public assets.
Chart	GET /api/v3_0/assets/(id)/chart/	Download a chart with time series
	GET /api/v3_0/assets/(id)/chart_data/	Download time series for use in charts
Data	GET /api/v3_0/sensors/data	Download sensor data
	POST /api/v3_0/sensors/data	Upload sensor data
Schedule	GET /api/v3_0/sensors/(id)/schedules/(uuid)	Download schedule from the platform
	POST /api/v3_0/sensors/(id)/schedules/trigger	Trigger scheduling job
Sensor	GET /api/v3_0/sensors	Download sensor list
User	GET /api/v3_0/users	Download user list
	GET /api/v3_0/users/(id)	Get a user
	PATCH /api/v3_0/users/(id)	Patch data for an existing user
	PATCH /api/v3_0/users/(id)/password-reset	Password reset

API Details

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, UNEX-
PECTED_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, UNEX-
PECTED_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

---

5.3. I want to build new features quickly, not spend days solving basic problems 73

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, UNEX-PECTED_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/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",
        "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, UN-KNOWN_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.

The message should contain a flexibility model.

Example request A

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

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

Example request B

This message triggers a schedule 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 minimum and maximum soc are set to 10 and 25 kWh, respectively. Roundtrip efficiency for use in scheduling is set to 98%. 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",
    "soc-at-start": 12.1,
    "soc-unit": "kWh",
    "soc-targets": [
        {
            "value": 25,
            "datetime": "2015-06-02T16:00:00+00:00"
        }
    ],
    "soc-min": 10,
    "soc-max": 25,
    "roundtrip-efficiency": 0.98,
    "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",
    "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_TIMEZONE, INVALID_DATETIME, INVALID_DOMAIN, INVALID_UNIT, PTUS_INCOMPLETE
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

GET /api/v3_0/sensors/data
Get sensor data from FlexMeasures.

Example request

5.3. I want to build new features quickly, not spend days solving basic problems 77

```
{
    "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.

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 values 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.

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 user themselves 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, UNEX-
> > PECTED_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. Any subset of user fields can be sent. Only the user themselves or admins are allowed to update its data, while a non-admin can only edit a few of their own fields.
>
> **The following fields are not allowed to be updated:**
>
> > • 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],
```

(continues on next page)

```
    '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, UNEX-PECTED_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, UNEX-PECTED_PARAMS

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

5.3.21 Version 2.0

Summary

Resource	Operation	Description
Asset	*DELETE /api/v2_0/asset/(id)*	Delete an asset, together with its existing data.
	GET /api/v2_0/asset/(id)	Get an asset
	PATCH /api/v2_0/asset/(id)	Patch data for an existing asset
	GET /api/v2_0/assets	Download asset list
	POST /api/v2_0/assets	Post a new asset
	GET /api/v2_0/getConnection	Retrieve entity addresses of connections
Chart	*GET /api/v2_0/charts/power*	Get a power chart
	GET /api/v2_0/charts/power	
Control	*GET /api/v2_0/getDeviceMessage*	Download control signal from the platform
	POST /api/v2_0/postUdiEvent	Upload flexibility constraints to the platform
Data	*GET /api/v2_0/getMeterData*	Download meter data from the platform
	GET /api/v2_0/getPrognosis	Download prognosis from the platform
	POST /api/v2_0/postMeterData	Upload meter data to the platform
	POST /api/v2_0/postPriceData	Upload price data to the platform
	POST /api/v2_0/postPrognosis	Upload prognosis to the platform
	POST /api/v2_0/postWeatherData	Upload weather data to the platform
Public	*GET /api/*	List available API versions
	POST /api/requestAuthToken	Obtain an authentication token
	GET /api/v2_0/getService	Obtain a service listing for this version
User	*GET /api/v2_0/user/(id)*	Get a user
	PATCH /api/v2_0/user/(id)	Patch data for an existing user
	PATCH /api/v2_0/user/(id)/password-reset	Password reset
	GET /api/v2_0/users	Download user list

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)

DELETE /api/v2_0/asset/(*id***)**
> API endpoint to delete an asset, and its sensed data.

> This endpoint deletes an existing asset, as well as all measurements recorded for it. Only users who own the asset are allowed to delete the asset.

> **Warning:** This API will be replaced by the new-style asset and sensor APIs. The former is already working at at `/api/dev/generic_assets`. See also note_on_datamodel_transition.

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, UNEX-PECTED_PARAMS
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER

GET `/api/v2_0/asset/`(*id*)

API endpoint to get an asset.

This endpoint gets an asset. Only users who own the asset can use this endpoint.

> **Warning:** This API will be replaced by the new-style asset and sensor APIs. The former is already working at at `/api/dev/generic_assets`. See also note_on_datamodel_transition.

Example response

```json
{
    "asset_type": "battery",
    "capacity_in_mw": 2.0,
    "display_name": "Test battery",
    "event_resolution": 5,
    "id": 1,
    "latitude": 10,
    "longitude": 100,
    "market_id": 1,
    "max_soc_in_mwh": 5,
    "min_soc_in_mwh": 0,
    "name": "Test battery",
    "owner_id": 2,
    "soc_datetime": "2015-01-01T00:00:00+00:00",
    "soc_in_mwh": 2.5,
    "soc_udi_event_id": 203,
    "unit": "kW"
}
```

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, UNEX-PECTED_PARAMS

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

PATCH `/api/v2_0/asset/`(*id*)

API endpoint to patch asset data.

This endpoint sets data for an existing asset. Any subset of asset fields can be sent. Only users who own the asset are allowed to update its data.

> **Warning:** This API will be replaced by the new-style asset and sensor APIs. The former is already working at at `/api/dev/generic_assets`. See also note_on_datamodel_transition.

Several fields are not allowed to be updated, e.g. id. They are ignored.

Example request

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

Note that event_resolution is expected as the number of minutes and soc_datetime is expected as ISO8601 date-time string.

Example response

The whole asset is returned in the response:

```
{
    "asset_type": "battery",
    "capacity_in_mw": 2.0,
    "display_name": "Test battery",
    "event_resolution": 5,
    "id": 1,
    "latitude": 11.1,
    "longitude": 99.9,
    "market_id": 1,
    "max_soc_in_mwh": 5,
    "min_soc_in_mwh": 0,
    "name": "Test battery",
    "owner_id": 2,
    "soc_datetime": "2015-01-01T00:00:00+00:00",
    "soc_in_mwh": 2.5,
    "soc_udi_event_id": 203,
```

(continues on next page)

```
      "unit": "kW"
}
```

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

GET /api/v2_0/assets

API endpoint to get assets.

This endpoint returns all accessible assets for a given owner. The *owner_id* query parameter can be used to set an owner. If no owner is set, all accessible assets are returned. A non-admin user can only access their own assets.

> **Warning:** This API will be replaced by the new-style asset and sensor APIs. The former is already working at at `/api/dev/generic_assets`. See also note_on_datamodel_transition.

Example response

An example of one asset being returned:

```
[
    {
        "asset_type": "battery",
        "capacity_in_mw": 2.0,
        "display_name": "Test battery",
        "event_resolution": 10,
        "id": 1,
        "latitude": 10,
        "longitude": 100,
        "market_id": 1,
        "max_soc_in_mwh": 5,
        "min_soc_in_mwh": 0,
        "name": "Test battery",
        "owner_id": 2,
        "soc_datetime": "2015-01-01T00:00:00+00:00",
        "soc_in_mwh": 2.5,
```

```
            "soc_udi_event_id": 203,
            "unit": "MW"
    }
]
```

Note that event_resolution is returned as the number of minutes and soc_datetime is returned as ISO8601 datetime string.

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

POST `/api/v2_0/assets`

API endpoint to post a new asset.

This endpoint creates a new asset. Only users with the admin role are allowed to create assets.

> **Warning:** This API will be replaced by the new-style asset and sensor APIs. The former is already working at at `/api/dev/generic_assets`. See also note_on_datamodel_transition.

Example request

The following example contains the required fields only, plus the two state of charge (soc) fields which a battery asset needs to specify:

```
{
    "name": "Test battery",
    "asset_type": "battery",
    "unit": "kW",
    "owner_id": 2,
    "market_id": 1,
    "event_resolution": 5,
    "capacity_in_mw": 4.2,
    "latitude": 40,
    "longitude": 170.3,
    "max_soc_in_mwh": 5,
    "min_soc_in_mwh": 0
}
```

Note that event_resolution is expected as the number of minutes and soc_datetime is expected as ISO8601 datetime string.

Example response

The newly posted asset, including all fields, is returned in the response:

```
{
    "id": 1,
    "asset_type": "battery",
    "unit": "kW"
    "capacity_in_mw": 4.2,
    "display_name": "Test battery",
    "event_resolution": 5,
    "latitude": 40,
    "longitude": 170.3,
    "max_soc_in_mwh": 5,
    "min_soc_in_mwh": 0,
    "name": "Test battery",
    "owner_id": 2,
    "market_id": 1,
    "soc_datetime": null,
    "soc_in_mwh": null,
    "soc_udi_event_id": null
}
```

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

GET /api/v2_0/charts/power

GET /api/v2_0/charts/power

API endpoint to get a chart for power data which can be embedded in web pages.

This endpoint returns a Bokeh chart with power data which can be embedded in a website. It includes forecasts and even schedules, if available.

Example request

An example of a chart request:

```
{
    "resource": ""my-battery,
    "start_time": "2020-02-20:10:00:00UTC",
    "end_time": "2020-02-20:11:00:00UTC",
    "resolution": "PT15M",
    "consumption_as_positive": true
```

```
    "resolution": "PT6H",
    "show_individual_traces_for": "none"  // can be power or schedules
}
```

On your webpage, you need to include the Bokeh libraries, e.g.:

> <script src="https://cdn.pydata.org/bokeh/release/bokeh-1.0.4.min.js"></script>

(The version needs to match the version used by the FlexMeasures server, see requirements/app.txt)

Then you can call this endpoint and include the result like this:

```
<script>
    fetch('http://localhost:5000/api/v2_0/charts/power?' + urlData.toString(),
    {
        method: "GET",
        mode: "cors",
        headers:
            {
                "Content-Type": "application/json",
                "Authorization": "<users auth token>"
            },
    })
    .then(function(response) { return response.json(); })
    .then(function(item) { Bokeh.embed.embed_item(item, "<ID of the div >"); });
</script>
```

where *urlData* is a *URLSearchData* object and contains the chart request parameters (see above).

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/v2_0/getConnection
    API endpoint to get the user's connections as entity addresses ordered from newest to oldest.

Example request

```
{
    "type": "GetConnectionRequest",
}
```

Example response

This "GetConnectionResponse" message indicates that the user had access rights to retrieve four entity addresses owned by three different users.

```
{
    "type": "GetConnectionResponse",
    "connections": [
        "ea1.2021-01.io.flexmeasures.company:3:4",
        "ea1.2021-01.io.flexmeasures.company:8:3",
        "ea1.2021-01.io.flexmeasures.company:9:2",
        "ea1.2021-01.io.flexmeasures.company:3:1"
    ],
    "names": [
        "CS 4",
        "CS 3",
        "CS 2",
        "CS 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_MESSAGE_TYPE

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v2_0/getDeviceMessage
API endpoint to get device message.

Optional fields

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

Example request

This "GetDeviceMessageRequest" message requests targeted consumption for UDI event 203 of device 10 of owner 7.

```
{
    "type": "GetDeviceMessageRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc"
}
```

Example response

This "GetDeviceMessageResponse" message indicates that the target for UDI event 203 is to consume at various power rates from 10am UTC onwards for a duration of 45 minutes.

```json
{
    "type": "GetDeviceMessageResponse",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_DOMAIN, INVALID_UNIT, UNKNOWN_SCHEDULE, UNRECOG-NIZED_CONNECTION_GROUP, or UNRECOGNIZED_UDI_EVENT
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v2_0/getMeterData

API endpoint to get meter data.

Optional fields

- "resolution" (see *Resolutions*)
- "horizon" (see *Tracking the recording time of beliefs*)
- "prior" (see *Tracking the recording time of beliefs*)
- "source" (see *Sources*)

Example request

This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

GET /api/v2_0/getPrognosis
API endpoint to get prognosis.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetPrognosisRequest" message requests prognosed consumption between 0.00am and 1.30am for charging station 1, with a rolling horizon of 6 hours before the end of each 15 minute time interval.

```
{
    "type": "GetPrognosisRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT6H",
    "resolution": "PT15M",
    "unit": "MW"
}
```

Example response

This "GetPrognosisResponse" message indicates that a prognosis of consumption for charging station 1 was available 6 hours before the start of each 15 minute time interval.

```
{
    "type": "GetPrognosisResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_MESSAGE_TYPE, INVALID_SOURCE, IN-
  VALID_TIMEZONE, INVALID_UNIT, UNRECOGNIZED_ASSET, or UNRECOG-
  NIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v2_0/getService

API endpoint to get a service listing for this version.

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – PROCESSED

POST /api/v2_0/postMeterData

API endpoint to post meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

- "prior" (see *Setting the recording time*)

Example request

This "PostMeterDataRequest" message posts measured consumption for 15-minute intervals between 0.00am and 1.30am for connections 3, 4 and 5 (negative values denote production).

```json
{
    "type": "PostMeterDataRequest",
    "groups": [
        {
            "connections": [
                "ea1.2021-01.io.flexmeasures.company:fm1.3",
                "ea1.2021-01.io.flexmeasures.company:fm1.4"
            ],
            "values": [
                306.66,
                306.66,
                0,
                0,
                306.66,
                306.66
            ]
        },
        {
            "connections": [
                "ea1.2021-01.io.flexmeasures.company:fm1.5"
            ],
            "values": [
                306.66,
                0,
                0,
                0,
                306.66,
                306.66
```

```
            ]
        }
    ],
    "start": "2021-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostMeterDataResponse" message indicates that the measurement has been processed without any error.

```
{
    "type": "PostMeterDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UNRECOGNIZED_CONNECTION_GROUP
> >
> > • 401 Unauthorized – UNAUTHORIZED
> >
> > • 403 Forbidden – INVALID_SENDER
> >
> > • 405 Method Not Allowed – INVALID_METHOD

POST /api/v2_0/postPriceData

> API endpoint to post price data.
>
> **Optional fields**
>
> > • "horizon" (see *Setting the recording time*)
> >
> > • "prior" (see *Setting the recording time*)
>
> **Example request**
>
> This "PostPriceDataRequest" message posts prices for hourly intervals between midnight and midnight the next day for the EPEX SPOT day-ahead auction, registered as sensor 12. The prior indicates that the prices were published at 1pm on December 31st 2020.

```
{
    "type": "PostPriceDataRequest",
    "market": "ea1.2021-01.io.flexmeasures.company:fm1.12",
    "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": "2021-01-01T00:00:00+01:00",
    "duration": "PT24H",
    "prior": "2020-12-31T13:00:00+01:00",
    "unit": "EUR/MWh"
}
```

Example response

This "PostPriceDataResponse" message indicates that the prices have been processed without any error.

```
{
    "type": "PostPriceDataResponse",
    "status": "PROCESSED",
    "message": "Request has been processed."
}
```

> **Request Headers**
>
> > • Authorization – The authentication token
> >
> > • Content-Type – application/json
>
> **Response Headers**
>
> > • Content-Type – application/json
>
> **Status Codes**

5.3. I want to build new features quickly, not spend days solving basic problems 95

- 200 OK – PROCESSED

- 400 Bad Request – INVALID_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_MARKET

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

POST /api/v2_0/postPrognosis

API endpoint to post prognoses about meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

- "prior" (see *Setting the recording time*)

Example request

This "PostPrognosisRequest" message posts prognosed consumption for 15-minute intervals between 0.00am
and 1.30am for connections 3, 4 and 5 (negative values denote production), prognosed at 6pm the previous day.

```json
{
    "type": "PostPrognosisRequest",
    "groups": [
        {
            "connections": [
                "ea1.2021-01.io.flexmeasures.company:fm1.3",
                "ea1.2021-01.io.flexmeasures.company:fm1.4"
            ],
            "values": [
                300,
                300,
                300,
                0,
                0,
                300
            ]
        },
        {
            "connections": [
                "ea1.2021-01.io.flexmeasures.company:fm1.5"
            ],
            "values": [
                300,
                0,
                0,
                0,
                300,
                300
            ]
        }
    ],
    "start": "2021-01-01T00:00:00Z",
```

```
    "duration": "PT1H30M",
    "prior": "2020-12-31T18:00:00Z",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostPrognosisResponse" message indicates that the prognosis has been processed without any error.

```
{
    "type": "PostPrognosisResponse",
    "status": "PROCESSED",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UN-RECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v2_0/postUdiEvent

API endpoint to post UDI event. (UDI is the Universal Device Interface proposed by USEF for flexible device states)

Example request A

This "PostUdiEventRequest" message posts a state of charge (soc) of 12.1 kWh at 10.00am as UDI event 203 of device 10 of owner 7.

```
{
    "type": "PostUdiEventRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:7:10:203:soc",
    "value": 12.1,
    "unit": "kWh",
    "datetime": "2015-06-02T10:00:00+00:00"
}
```

Example request B

This "PostUdiEventRequest" message posts a state of charge (soc) of 12.1 kWh at 10.00am, and a target state of
charge of 25 kWh at 4.00pm, as UDI event 204 of device 10 of owner 7. The minimum and maximum soc are
set to 10 and 25 kWh, respectively. Roundtrip efficiency for use in scheduling is set to 98%.

```json
{
    "type": "PostUdiEventRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:204:soc-with-targets",
    "value": 12.1,
    "unit": "kWh",
    "datetime": "2015-06-02T10:00:00+00:00",
    "targets": [
        {
            "value": 25,
            "datetime": "2015-06-02T16:00:00+00:00"
        }
    ],
    "soc_min": 10,
    "soc_max": 25,
    "roundtrip_efficiency": 0.98
}
```

Example response

This "PostUdiEventResponse" message indicates that the UDI event has been processed without any error.

```json
{
    "type": "PostUdiEventResponse",
    "status": "PROCESSED",
    "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 – INCOMPLETE_UDI_EVENT, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_DATETIME, INVALID_DOMAIN, INVALID_UNIT,
  OUTDATED_UDI_EVENT, PTUS_INCOMPLETE, OUTDATED_UDI_EVENT or UN-
  RECOGNIZED_UDI_EVENT
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v2_0/postWeatherData
API endpoint to post weather data, such as:

- "irradiance" (with kW/m^2 as unit)

- "temperature" (with °C as unit)

- "wind speed" (with m/s as unit)

Optional fields

- "horizon" (see *Setting the recording time*)

- "prior" (see *Setting the recording time*)

Example request

This "PostWeatherDataRequest" message posts temperature forecasts for 15-minute intervals between 3.00pm and 4.30pm for a weather sensor with id 602. The forecasts were made at noon.

```
{
    "type": "PostWeatherDataRequest",
    "groups": [
        {
            "sensor": "ea1.2021-01.io.flexmeasures.company:fm1.602",
            "values": [
                20.04,
                20.23,
                20.41,
                20.51,
                20.55,
                20.57
            ]
        }
    ],
    "start": "2021-01-01T15:00:00+09:00",
    "duration": "PT1H30M",
    "prior": "2021-01-01T12:00:00+09:00",
    "unit": "°C"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostWeatherDataResponse" message indicates that the forecast has been processed without any error.

```
{
    "type": "PostWeatherDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_SENSOR

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v2_0/user/(*id*)

API endpoint to get a user.

This endpoint gets a user. Only admins or the user themselves 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, UNEX-
  PECTED_PARAMS

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

PATCH /api/v2_0/user/(*id*)

API endpoint to patch user data.

This endpoint sets data for an existing user. Any subset of user fields can be sent. Only the user themselves or
admins are allowed to update its data, while a non-admin can only edit a few of their own fields.

Several fields are not allowed to be updated, e.g. id and account_id. They are ignored.

Example request

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

Example response

The whole user is returned in the 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 – UPDATED
> >
> > • 400 Bad Request – INVALID_REQUEST, REQUIRED_INFO_MISSING, UNEX-PECTED_PARAMS
> >
> > • 401 Unauthorized – UNAUTHORIZED
> >
> > • 403 Forbidden – INVALID_SENDER
> >
> > • 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

PATCH `/api/v2_0/user/`(*id*)`/password-reset`
> API endpoint to reset the user password. They'll get an email to choose a new password.
>
> 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, UNEX-PECTED_PARAMS

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

GET /api/v2_0/users

API endpoint to get users.

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

5.3.22 Version 1.3

Summary

Resource	Operation	Description
Asset	*GET /api/v1_3/getConnection*	Retrieve entity addresses of connections
Control	*GET /api/v1_3/getDeviceMessage*	Download control signal from the platform
	POST /api/v1_3/postUdiEvent	Upload flexibility constraints to the platform
Data	*GET /api/v1_3/getMeterData*	Download meter data from the platform
	GET /api/v1_3/getPrognosis	Download prognosis from the platform
	POST /api/v1_3/postMeterData	Upload meter data to the platform
	POST /api/v1_3/postPriceData	Upload price data to the platform
	POST /api/v1_3/postPrognosis	Upload prognosis to the platform
	POST /api/v1_3/postWeatherData	Upload weather data to the platform
Public	*GET /api/*	List available API versions
	POST /api/requestAuthToken	Obtain an authentication token
	GET /api/v1_3/getService	Obtain a service listing for this version

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/v1_3/getConnection
> API endpoint to get the user's connections as entity addresses ordered from newest to oldest.

> **Example request**

```
{
    "type": "GetConnectionRequest",
}
```

> **Example response**

> This "GetConnectionResponse" message indicates that the user had access rights to retrieve four entity addresses owned by three different users.

```
{
    "type": "GetConnectionResponse",
    "connections": [
        "ea1.2021-01.io.flexmeasures.company:3:4",
        "ea1.2021-01.io.flexmeasures.company:8:3",
```

```
        "ea1.2021-01.io.flexmeasures.company:9:2",
        "ea1.2021-01.io.flexmeasures.company:3:1"
    ],
    "names": [
        "CS 4",
        "CS 3",
        "CS 2",
        "CS 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_MESSAGE_TYPE

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_3/getDeviceMessage

API endpoint to get device message.

Optional fields

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

Example request

This "GetDeviceMessageRequest" message requests targeted consumption for UDI event 203 of device 10 of owner 7.

```
{
    "type": "GetDeviceMessageRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc"
}
```

Example response

This "GetDeviceMessageResponse" message indicates that the target for UDI event 203 is to consume at various power rates from 10am UTC onwards for a duration of 45 minutes.

```
{
    "type": "GetDeviceMessageResponse",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_DOMAIN, INVALID_UNIT, UNKNOWN_SCHEDULE, UNRECOG-NIZED_CONNECTION_GROUP, or UNRECOGNIZED_UDI_EVENT

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v1_3/getMeterData

API endpoint to get meter data.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```json
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_3/getPrognosis

API endpoint to get prognosis.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetPrognosisRequest" message requests prognosed consumption between 0.00am and 1.30am for charging station 1, with a rolling horizon of 6 hours before the end of each 15 minute time interval.

```
{
    "type": "GetPrognosisRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT6H",
    "resolution": "PT15M",
    "unit": "MW"
}
```

Example response

This "GetPrognosisResponse" message indicates that a prognosis of consumption for charging station 1 was available 6 hours before the start of each 15 minute time interval.

```
{
    "type": "GetPrognosisResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_MESSAGE_TYPE, INVALID_SOURCE, IN-VALID_TIMEZONE, INVALID_UNIT, UNRECOGNIZED_ASSET, or UNRECOG-NIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_3/getService

API endpoint to get a service listing for this version.

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – PROCESSED

POST /api/v1_3/postMeterData

API endpoint to post meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostMeterDataRequest" message posts measured consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production).

```json
{
    "type": "PostMeterDataRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                306.66,
                306.66,
                0,
                0,
                306.66,
                306.66
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                306.66,
                0,
                0,
                0,
                306.66,
                306.66
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostMeterDataResponse" message indicates that the measurement has been processed without any error.

```
{
    "type": "PostMeterDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UNRECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_3/postPriceData

API endpoint to post price data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPriceDataRequest" message posts prices for hourly intervals between midnight and midnight the next day for the EPEX SPOT day-ahead auction. The horizon indicates that the prices were published at 1pm on December 31st 2014 (i.e. 35 hours ahead of midnight the next day).

```
{
    "type": "PostPriceDataRequest",
    "market": "ea1.2021-01.io.flexmeasures.company:epex_da",
    "values": [
        52.37,
        51.14,
        49.09,
        48.35,
        48.47,
        49.98,
        58.7,
        67.76,
        69.21,
        70.26,
        70.46,
        70,
```

(continues on next page)

```
            70.7,
            70.41,
            70,
            64.53,
            65.92,
            69.72,
            70.51,
            75.49,
            70.35,
            70.01,
            66.98,
            58.61
        ],
        "start": "2015-01-01T15:00:00+09:00",
        "duration": "PT24H",
        "horizon": "PT35H",
        "unit": "EUR/MWh"
}
```

Example response

This "PostPriceDataResponse" message indicates that the prices have been processed without any error.

```
{
        "type": "PostPriceDataResponse",
        "status": "PROCESSED",
        "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-NIZED_ASSET or UNRECOGNIZED_MARKET
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_3/postPrognosis
API endpoint to post prognoses about meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPrognosisRequest" message posts prognosed consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production), prognosed at 6pm the previous day.

```
{
    "type": "PostPrognosisRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                300,
                300,
                300,
                0,
                0,
                300
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                300,
                0,
                0,
                0,
                300,
                300
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT7H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostPrognosisResponse" message indicates that the prognosis has been processed without any error.

```
{
    "type": "PostPrognosisResponse",
    "status": "PROCESSED",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UN-RECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_3/postUdiEvent

API endpoint to post UDI event. (UDI is the Universal Device Interface proposed by USEF for flexible device states)

Example request A

This "PostUdiEventRequest" message posts a state of charge (soc) of 12.1 kWh at 10.00am as UDI event 203 of device 10 of owner 7.

```
{
    "type": "PostUdiEventRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:7:10:203:soc",
    "value": 12.1,
    "unit": "kWh",
    "datetime": "2015-06-02T10:00:00+00:00"
}
```

Example request B

This "PostUdiEventRequest" message posts a state of charge (soc) of 12.1 kWh at 10.00am, and a target state of charge of 25 kWh at 4.00pm, as UDI event 204 of device 10 of owner 7. The minimum and maximum soc are set to 10 and 25 kWh, respectively. Roundtrip efficiency for use in scheduling is set to 98%.

```
{
    "type": "PostUdiEventRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:204:soc-with-targets",
    "value": 12.1,
    "unit": "kWh",
    "datetime": "2015-06-02T10:00:00+00:00",
    "targets": [
        {
            "value": 25,
            "datetime": "2015-06-02T16:00:00+00:00"
        }
    ],
    "soc_min": 10,
```

(continues on next page)

```
    "soc_max": 25,
    "roundtrip_efficiency": 0.98
}
```

Example response

This "PostUdiEventResponse" message indicates that the UDI event has been processed without any error.

```
{
    "type": "PostUdiEventResponse",
    "status": "PROCESSED",
    "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 – INCOMPLETE_UDI_EVENT, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_DATETIME, INVALID_DOMAIN, INVALID_UNIT, OUTDATED_UDI_EVENT, PTUS_INCOMPLETE, OUTDATED_UDI_EVENT or UN-RECOGNIZED_UDI_EVENT
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_3/postWeatherData

API endpoint to post weather data, such as:

- "irradiance" (with kW/m^2 as unit)
- "temperature" (with °C as unit)
- "wind speed" (with m/s as unit)

The sensor type is part of the unique entity address for each sensor, together with the sensor's latitude and longitude.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostWeatherDataRequest" message posts temperature forecasts for 15-minute intervals between 3.00pm and 4.30pm for a weather sensor located at latitude 33.4843866 and longitude 126.477859. The forecasts were made at noon.

```
{
    "type": "PostWeatherDataRequest",
    "groups": [
        {
            "sensor": "ea1.2021-01.io.flexmeasures.company:temperature:33.
↪4843866:126.477859",
            "values": [
                20.04,
                20.23,
                20.41,
                20.51,
                20.55,
                20.57
            ]
        }
    ],
    "start": "2015-01-01T15:00:00+09:00",
    "duration": "PT1H30M",
    "horizon": "PT3H",
    "unit": "°C"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostWeatherDataResponse" message indicates that the forecast has been processed without any error.

```
{
    "type": "PostWeatherDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_SENSOR

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

5.3.23 Version 1.2

Summary

Resource	Operation	Description
Asset	*GET /api/v1_2/getConnection*	Retrieve entity addresses of connections
Control	*GET /api/v1_2/getDeviceMessage*	Download control signal from the platform
	POST /api/v1_2/postUdiEvent	Upload flexibility constraints to the platform
Data	*GET /api/v1_2/getMeterData*	Download meter data from the platform
	GET /api/v1_2/getPrognosis	Download prognosis from the platform
	POST /api/v1_2/postMeterData	Upload meter data to the platform
	POST /api/v1_2/postPriceData	Upload price data to the platform
	POST /api/v1_2/postPrognosis	Upload prognosis to the platform
	POST /api/v1_2/postWeatherData	Upload weather data to the platform
Public	*GET /api/*	List available API versions
	POST /api/requestAuthToken	Obtain an authentication token
	GET /api/v1_2/getService	Obtain a service listing for this version

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/v1_2/getConnection
    API endpoint to get the user's connections as entity addresses ordered from newest to oldest.

    Example request

```
{
    "type": "GetConnectionRequest",
}
```

    Example response

    This "GetConnectionResponse" message indicates that the user had access rights to retrieve four entity addresses owned by three different users.

```
{
    "type": "GetConnectionResponse",
    "connections": [
        "ea1.2021-01.io.flexmeasures.company:3:4",
        "ea1.2021-01.io.flexmeasures.company:8:3",
```

```
        "ea1.2021-01.io.flexmeasures.company:9:2",
        "ea1.2021-01.io.flexmeasures.company:3:1"
    ],
    "names": [
        "CS 4",
        "CS 3",
        "CS 2",
        "CS 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_MESSAGE_TYPE
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_2/getDeviceMessage

API endpoint to get device message.

Optional fields

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

Example request

This "GetDeviceMessageRequest" message requests targeted consumption for UDI event 203 of device 10 of owner 7.

```
{
    "type": "GetDeviceMessageRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc"
}
```

Example response

This "GetDeviceMessageResponse" message indicates that the target for UDI event 203 is to consume at various power rates from 10am UTC onwards for a duration of 45 minutes.

```
{
    "type": "GetDeviceMessageResponse",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, UNKNOWN_PRICES, UNRECOG-NIZED_CONNECTION_GROUP, or UNRECOGNIZED_UDI_EVENT
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD
- 422 Unprocessable Entity – UNPROCESSABLE_ENTITY

GET /api/v1_2/getMeterData

API endpoint to get meter data.

Optional fields

- "resolution" (see *Resolutions*)
- "horizon" (see *Tracking the recording time of beliefs*)
- "prior" (see *Tracking the recording time of beliefs*)
- "source" (see *Sources*)

Example request

This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```json
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_2/getPrognosis
API endpoint to get prognosis.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetPrognosisRequest" message requests prognosed consumption between 0.00am and 1.30am for charging station 1, with a rolling horizon of 6 hours before the end of each 15 minute time interval.

```json
{
    "type": "GetPrognosisRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT6H",
    "resolution": "PT15M",
    "unit": "MW"
}
```

Example response

This "GetPrognosisResponse" message indicates that a prognosis of consumption for charging station 1 was available 6 hours before the start of each 15 minute time interval.

```json
{
    "type": "GetPrognosisResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_MESSAGE_TYPE, INVALID_SOURCE, IN-VALID_TIMEZONE, INVALID_UNIT, UNRECOGNIZED_ASSET, or UNRECOG-NIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_2/getService

API endpoint to get a service listing for this version.

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – PROCESSED

POST /api/v1_2/postMeterData

API endpoint to post meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostMeterDataRequest" message posts measured consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production).

```json
{
    "type": "PostMeterDataRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                306.66,
                306.66,
                0,
                0,
                306.66,
                306.66
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                306.66,
                0,
                0,
                0,
                306.66,
                306.66
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostMeterDataResponse" message indicates that the measurement has been processed without any error.

```json
{
    "type": "PostMeterDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_2/postPriceData

API endpoint to post price data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPriceDataRequest" message posts prices for hourly intervals between midnight and midnight the next
day for the EPEX SPOT day-ahead auction. The horizon indicates that the prices were published at 1pm on
December 31st 2014 (i.e. 35 hours ahead of midnight the next day).

```json
{
    "type": "PostPriceDataRequest",
    "market": "ea1.2021-01.io.flexmeasures.company:epex_da",
    "values": [
        52.37,
        51.14,
        49.09,
        48.35,
        48.47,
        49.98,
        58.7,
        67.76,
        69.21,
        70.26,
        70.46,
        70,
```

(continues on next page)

5.3. I want to build new features quickly, not spend days solving basic problems   **121**

```
            70.7,
            70.41,
            70,
            64.53,
            65.92,
            69.72,
            70.51,
            75.49,
            70.35,
            70.01,
            66.98,
            58.61
        ],
        "start": "2015-01-01T15:00:00+09:00",
        "duration": "PT24H",
        "horizon": "PT35H",
        "unit": "EUR/MWh"
}
```

Example response

This "PostPriceDataResponse" message indicates that the prices have been processed without any error.

```
{
    "type": "PostPriceDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_MARKET

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_2/postPrognosis
    API endpoint to post prognoses about meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPrognosisRequest" message posts prognosed consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production), prognosed at 6pm the previous day.

```
{
    "type": "PostPrognosisRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                300,
                300,
                300,
                0,
                0,
                300
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                300,
                0,
                0,
                0,
                300,
                300
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT7H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostPrognosisResponse" message indicates that the prognosis has been processed without any error.

```
{
    "type": "PostPrognosisResponse",
    "status": "PROCESSED",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UN-RECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_2/postUdiEvent
    API endpoint to post UDI event.

    Example request

    This "PostUdiEventRequest" message posts a state of charge (soc) of 12.1 kWh at 10.00am as UDI event 203 of device 10 of owner 7.

```
{
    "type": "PostUdiEventRequest",
    "event": "ea1.2021-01.io.flexmeasures.company:fm0.7:10:203:soc",
    "value": 12.1,
    "unit": "kWh",
    "datetime": "2015-06-02T10:00:00+00:00",
}
```

    Example response

    This "PostUdiEventResponse" message indicates that the UDI event has been processed without any error.

```
{
    "type": "PostUdiEventResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_DATETIME, INVALID_UNIT, PTUS_INCOMPLETE, OUTDATED_UDI_EVENT or UNRECOGNIZED_UDI_EVENT

- **401 Unauthorized** – UNAUTHORIZED

- **403 Forbidden** – INVALID_SENDER

- **405 Method Not Allowed** – INVALID_METHOD

POST /api/v1_2/postWeatherData

API endpoint to post weather data, such as:

- "irradiance" (with kW/m^2 as unit)

- "temperature" (with °C as unit)

- "wind speed" (with m/s as unit)

The sensor type is part of the unique entity address for each sensor, together with the sensor's latitude and longitude.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostWeatherDataRequest" message posts temperature forecasts for 15-minute intervals between 3.00pm and 4.30pm for a weather sensor located at latitude 33.4843866 and longitude 126.477859. The forecasts were made at noon.

```
{
    "type": "PostWeatherDataRequest",
    "groups": [
        {
            "sensor": "ea1.2021-01.io.flexmeasures.company:temperature:33.
→4843866:126.477859",
            "values": [
                20.04,
                20.23,
                20.41,
                20.51,
                20.55,
                20.57
            ]
        }
    ],
    "start": "2015-01-01T15:00:00+09:00",
    "duration": "PT1H30M",
    "horizon": "PT3H",
    "unit": "°C"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostWeatherDataResponse" message indicates that the forecast has been processed without any error.

```
{
    "type": "PostWeatherDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-NIZED_ASSET or UNRECOGNIZED_SENSOR

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

5.3.24 Version 1.1

Summary

Resource	Operation	Description
Asset	*GET /api/v1_1/getConnection*	Retrieve entity addresses of connections
Data	*GET /api/v1_1/getMeterData*	Download meter data from the platform
	GET /api/v1_1/getPrognosis	Download prognosis from the platform
	POST /api/v1_1/postMeterData	Upload meter data to the platform
	POST /api/v1_1/postPriceData	Upload price data to the platform
	POST /api/v1_1/postPrognosis	Upload prognosis to the platform
	POST /api/v1_1/postWeatherData	Upload weather data to the platform
Public	*GET /api/*	List available API versions
	POST /api/requestAuthToken	Obtain an authentication token
	GET /api/v1_1/getService	Obtain a service listing for this version

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/v1_1/getConnection
>   API endpoint to get the user's connections as entity addresses ordered from newest to oldest.
>
>   **Example request**

```
{
    "type": "GetConnectionRequest",
}
```

>   **Example response**
>
>   This "GetConnectionResponse" message indicates that the user had access rights to retrieve four entity addresses
>   owned by three different users.

```
{
    "type": "GetConnectionResponse",
    "connections": [
        "ea1.2021-01.io.flexmeasures.company:3:4",
        "ea1.2021-01.io.flexmeasures.company:8:3",
        "ea1.2021-01.io.flexmeasures.company:9:2",
        "ea1.2021-01.io.flexmeasures.company:3:1"
    ],
    "names": [
        "CS 4",
        "CS 3",
        "CS 2",
        "CS 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_MESSAGE_TYPE
>   - 401 Unauthorized – UNAUTHORIZED
>   - 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_1/getMeterData
API endpoint to get meter data.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-
  NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_1/getPrognosis

API endpoint to get prognosis.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetPrognosisRequest" message requests prognosed consumption between 0.00am and 1.30am for charging station 1, with a rolling horizon of 6 hours before the end of each 15 minute time interval.

```
{
    "type": "GetPrognosisRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT6H",
    "resolution": "PT15M",
    "unit": "MW"
}
```

Example response

This "GetPrognosisResponse" message indicates that a prognosis of consumption for charging station 1 was available 6 hours before the start of each 15 minute time interval.

```
{
    "type": "GetPrognosisResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_MESSAGE_TYPE, INVALID_SOURCE, IN-VALID_TIMEZONE, INVALID_UNIT, UNRECOGNIZED_ASSET, or UNRECOG-NIZED_CONNECTION_GROUP
> >
> > - 401 Unauthorized – UNAUTHORIZED
> >
> > - 403 Forbidden – INVALID_SENDER
> >
> > - 405 Method Not Allowed – INVALID_METHOD

GET /api/v1_1/getService

> API endpoint to get a service listing for this version.
>
> > **Response Headers**
> >
> > > - Content-Type – application/json
> >
> > **Status Codes**
> >
> > > - 200 OK – PROCESSED

POST /api/v1_1/postMeterData

> API endpoint to post meter data.
>
> **Optional fields**
>
> - "horizon" (see *Setting the recording time*)
>
> **Example request**
>
> This "PostMeterDataRequest" message posts measured consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production).

```json
{
    "type": "PostMeterDataRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                306.66,
                306.66,
                0,
                0,
                306.66,
                306.66
            ]
        }
```

<div align="right">(continues on next page)</div>

```
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                306.66,
                0,
                0,
                0,
                306.66,
                306.66
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostMeterDataResponse" message indicates that the measurement has been processed without any error.

```
{
    "type": "PostMeterDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-NIZED_ASSET or UNRECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_1/postPriceData
    API endpoint to post price data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPriceDataRequest" message posts prices for hourly intervals between midnight and midnight the next day for the EPEX SPOT day-ahead auction. The horizon indicates that the prices were published at 1pm on December 31st 2014 (i.e. 35 hours ahead of midnight the next day).

```json
{
    "type": "PostPriceDataRequest",
    "market": "ea1.2021-01.io.flexmeasures.company:epex_da",
    "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": "2015-01-01T15:00:00+09:00",
    "duration": "PT24H",
    "horizon": "PT35H",
    "unit": "EUR/MWh"
}
```

Example response

This "PostPriceDataResponse" message indicates that the prices have been processed without any error.

```json
{
    "type": "PostPriceDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_MARKET

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_1/postPrognosis
API endpoint to post prognoses about meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostPrognosisRequest" message posts prognosed consumption for 15-minute intervals between 0.00am
and 1.30am for charging stations 1, 2 and 3 (negative values denote production), prognosed at 6pm the previous
day.

```
{
    "type": "PostPrognosisRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                300,
                300,
                300,
                0,
                0,
                300
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                300,
                0,
                0,
```

```
                0,
                300,
                300
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "horizon": "PT7H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostPrognosisResponse" message indicates that the prognosis has been processed without any error.

```
{
    "type": "PostPrognosisResponse",
    "status": "PROCESSED",
    "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_MESSAGE_TYPE, INVALID_TIMEZONE, IN-VALID_UNIT, REQUIRED_INFO_MISSING, UNRECOGNIZED_ASSET or UN-RECOGNIZED_CONNECTION_GROUP
- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

POST /api/v1_1/postWeatherData

API endpoint to post weather data, such as:

- "irradiance" (with kW/m^2 as unit)
- "temperature" (with °C as unit)
- "wind speed" (with m/s as unit)

The sensor type is part of the unique entity address for each sensor, together with the sensor's latitude and longitude.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostWeatherDataRequest" message posts temperature forecasts for 15-minute intervals between 3.00pm and 4.30pm for a weather sensor located at latitude 33.4843866 and longitude 126.477859. The forecasts were made at noon.

```
{
    "type": "PostWeatherDataRequest",
    "groups": [
        {
            "sensor": "ea1.2021-01.io.flexmeasures.company:temperature:33.
→4843866:126.477859",
            "values": [
                20.04,
                20.23,
                20.41,
                20.51,
                20.55,
                20.57
            ]
        }
    ],
    "start": "2015-01-01T15:00:00+09:00",
    "duration": "PT1H30M",
    "horizon": "PT3H",
    "unit": "°C"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostWeatherDataResponse" message indicates that the forecast has been processed without any error.

```
{
    "type": "PostWeatherDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-NIZED_ASSET or UNRECOGNIZED_SENSOR

- 401 Unauthorized – UNAUTHORIZED
- 403 Forbidden – INVALID_SENDER
- 405 Method Not Allowed – INVALID_METHOD

5.3.25 Version 1.0

Summary

Resource	Operation	Description
Data	*POST /api/v1/getMeterData*	Download meter data from the plat-form
	GET /api/v1/getMeterData	
	POST /api/v1/postMeterData	Upload meter data to the platform
Public	*GET /api/*	List available API versions
	POST /api/requestAuthToken	Obtain an authentication token
	GET /api/v1/getService	Obtain a service listing for this version

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)

POST /api/v1/getMeterData
    API endpoint to get meter data.

    Optional fields

    - "resolution" (see *Resolutions*)
    - "horizon" (see *Tracking the recording time of beliefs*)
    - "prior" (see *Tracking the recording time of beliefs*)
    - "source" (see *Sources*)

    Example request

    This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```json
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

GET /api/v1/getMeterData
API endpoint to get meter data.

Optional fields

- "resolution" (see *Resolutions*)

- "horizon" (see *Tracking the recording time of beliefs*)

- "prior" (see *Tracking the recording time of beliefs*)

- "source" (see *Sources*)

Example request

This "GetMeterDataRequest" message requests measured consumption between 0.00am and 1.30am for charging station 1.

```json
{
    "type": "GetMeterDataRequest",
    "connection": "CS 1",
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

Example response

This "GetMeterDataResponse" message indicates that consumption for charging station 1 was measured in 15-minute intervals.

```json
{
    "type": "GetMeterDataResponse",
    "connection": "CS 1",
    "values": [
        306.66,
        306.66,
        0,
        0,
        306.66,
        306.66
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-VALID_SOURCE, INVALID_TIMEZONE, INVALID_UNIT, UNRECOG-NIZED_ASSET, or UNRECOGNIZED_CONNECTION_GROUP
> >
> > • 401 Unauthorized – UNAUTHORIZED
> >
> > • 403 Forbidden – INVALID_SENDER
> >
> > • 405 Method Not Allowed – INVALID_METHOD

GET /api/v1/getService

> API endpoint to get a service listing for this version.
>
> > **Response Headers**
> >
> > > • Content-Type – application/json

Status Codes

- 200 OK – PROCESSED

POST /api/v1/postMeterData
    API endpoint to post meter data.

Optional fields

- "horizon" (see *Setting the recording time*)

Example request

This "PostMeterDataRequest" message posts measured consumption for 15-minute intervals between 0.00am and 1.30am for charging stations 1, 2 and 3 (negative values denote production).

```json
{
    "type": "PostMeterDataRequest",
    "groups": [
        {
            "connections": [
                "CS 1",
                "CS 3"
            ],
            "values": [
                306.66,
                306.66,
                0,
                0,
                306.66,
                306.66
            ]
        },
        {
            "connections": [
                "CS 2"
            ],
            "values": [
                306.66,
                0,
                0,
                0,
                306.66,
                306.66
            ]
        }
    ],
    "start": "2015-01-01T00:00:00Z",
    "duration": "PT1H30M",
    "unit": "MW"
}
```

It is allowed to send higher resolutions (in this example for instance, 30 minutes) which will be upsampled.

Example response

This "PostMeterDataResponse" message indicates that the measurement has been processed without any error.

```
{
    "type": "PostMeterDataResponse",
    "status": "PROCESSED",
    "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_DOMAIN, INVALID_MESSAGE_TYPE, IN-
  VALID_TIMEZONE, INVALID_UNIT, REQUIRED_INFO_MISSING, UNRECOG-
  NIZED_ASSET or UNRECOGNIZED_CONNECTION_GROUP

- 401 Unauthorized – UNAUTHORIZED

- 403 Forbidden – INVALID_SENDER

- 405 Method Not Allowed – INVALID_METHOD

5.3.26 Developer API

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

Summary

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

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).

5.3.27 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.

---

v3.0-3 | 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* -> 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.

v2.0-4 | 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).

v1.3-11 | 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 (*FLEXMEA-SURES_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 (*FLEXMEA-SURES_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

v1.2-3 | 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

v1.1-5 | 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

v1.0-1 | 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)

- 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

5.3.28 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

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

show - Show data

`flexmeasures show accounts`	List accounts.
`flexmeasures show account`	Show an account, its users and assets.
`flexmeasures show asset-types`	List available asset types.
`flexmeasures show asset`	Show an asset and its sensors.
`flexmeasures show roles`	List available account- and user roles.
`flexmeasures show data-sources`	List available data sources.
`flexmeasures show beliefs`	Plot time series data.

edit - Edit data

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

delete - Delete data

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

`jobs` - Job queueing

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

`db-ops` - Operations on the whole database

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

5.3.29 FlexMeasures CLI Changelog

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`

5.3.30 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.

5.3.31 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.

Getting ready to use

Notes:

- We use postgres 12 at the moment, but any version starting with 9 probably works.

- 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

On Unix:

```
sudo apt-get install postgresql-12
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.

```
service postgresql restart
```

Setup the "flexmeasures" Unix user

This may in fact not be needed:

```
sudo /usr/sbin/adduser flexmeasures
```

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
```

(continues on next page)

```
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 SECU-RITY_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 `new-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 it's `--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.

5.3.32 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:

```python
# 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'
# 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.

5.3.33 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:

```python
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)

5.3.34 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 task to import prices from a day-ahead market, which you depend on later for scheduling. We added a new CLI task called `flexmeasures monitor tasks`, so you can be alerted when tasks have not successfully run at least so-and-so many minutes ago. The alerts will also come in via Sentry, but you can also send them to email addresses with the config setting *FLEXMEASURES_MONITORING_MAIL_RECIPIENTS*.

For illustration of the latter monitoring, here is one example of how we monitor 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 tasks --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 tasks` 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():
    ...
```

5.3.35 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).

5.3.36 Writing Plugins

You can extend FlexMeasures with functionality like UI pages, API endpoints, or CLI functions. This is eventually how energy flexibility services are built on top of FlexMeasures!

In an nutshell, a FlexMeasures plugin adds functionality via one or more Flask Blueprints.

Todo: We'll use this to allow for custom forecasting and scheduling algorithms, as well.

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*.

5.3.37 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.

```python
__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
```

<div align="right">(continues on next page)</div>

```python
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


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 *FLEXMEA-SURES_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
<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`:

```python
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
→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.

5.3.38 Plugin Customizations

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 *FLEXMEA-SURES_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",
}
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
from typing import Optional

import click
from flexmeasures.data.schemas.times 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.
```

```python
    """

@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: Optional[datetime] = 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"
```

5.3.39 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*
- *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: `python3.8 -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=3.8`.
- Activate it, e.g.: `source flexmeasures-venv/bin/activate`

Dependencies

Install all dependencies including the ones needed for development:

```
make install-for-dev
```

Configuration

Follow the configuration Quickstart advice in *Getting started* and *Configuration*.

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}
```

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

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.

A coverage report can be created like this:

```
pytest --cov=flexmeasures --cov-config .coveragerc
```

You can add –cov-report=html after which a htmlcov/index.html is generated.

It's also possible to use:

```
python setup.py test
```

Versioning

We use setuptool_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 :)

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_bitbucket
}
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.

5.3.40 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, these required settings can be set by environment variables. 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`

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).

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: `[]`

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"]}`

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 UDI event ids of successful scheduling jobs. Set a negative timedelta to persist forever.

Default: `timedelta(days=7)`

FLEXMEASURES_PLANNING_HORIZON

The horizon to use when making schedules.

Default: `timedelta(hours=2 * 24)`

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`

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`

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`

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.

5.3. I want to build new features quickly, not spend days solving basic problems　　　　**179**

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.

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.

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.

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`

5.3.41 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.

Todo: A guide for endpoint design, e.g. using Marshmallow schemas and common validators.

Table of contents

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
```

5.3.42 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 setuptool_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.

```bash
#!/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`.

5.3.43 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")},
```

---

[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).

```
    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`.

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.

5.3.44 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 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 38↵
→seconds (health: starting)   0.0.0.0:5000->5000/tcp   flexmeasures-server-1
e36cd54a7fd5    flexmeasures_worker   "flexmeasures jobs r..."   44 seconds ago   Up 5↵
→seconds                      5000/tcp              flexmeasures-worker-1
c9985de27f68    postgres              "docker-entrypoint.s..."   45 seconds ago   Up 40↵
→seconds                      5432/tcp              flexmeasures-test-db-1
```

```
03582d37230e    postgres               "docker-entrypoint.s..."   45 seconds ago   Up 40␣
↪seconds                        5432/tcp                  flexmeasures-dev-db-1
792ec3d86e71    redis                  "docker-entrypoint.s..."   45 seconds ago   Up 40␣
↪seconds                        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).

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 --sensor-id 2 --optimization-context-id 3 \
    --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 2 --
→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 DOCUMENTATION

Go To source.