1. Background

The idea was to create some sort of IoT Dashboard for the data provided by Tinkerforge. We have client and server components as one application that can run without the need of the other. The client gathers environmental data with Tinkerforge, keeps them local, and sends them to the server, but also offers if wanted a local dashboard. The Server collects all the data provided by the clients and offers a unified interface to see the data.

The server checks every 5 seconds whether all peers are still connected with a ping - pong message. The client sends every second all collected environmental data to the server.

2. Application Component

For the communication, we used ZeroMQ to make use of their already defined asynchronous patterns.

WebSockets might have been a smarter choice as they are quite similar but don't handle everything in bytes and the construct around it is rather raw and not as defined in ZeroMQ already.

As a communication pattern, we decided for Dealer to Router, as it provides us with a bidirectional channel and the option to talk to specific peers with as many Dealers as we want.

Other options we looked at:
Dealer to Dealer - quite interesting at first glance, but as soon as you add another peer your application won't work because of Round-robin and the missing concept of identities. If you only have two peers then this is a valid option, but as we wanted to be able to add lots of clients, therefore, this was not an option.

Subscriber / Publisher - it's unreasonable to open two TCP sockets to just communicate in a bidirectional way and on top of that there is no security that other peers might not be listening to the communication, meaning everyone will receive all messages and there is no direct bidirectional communication between two specific peers.

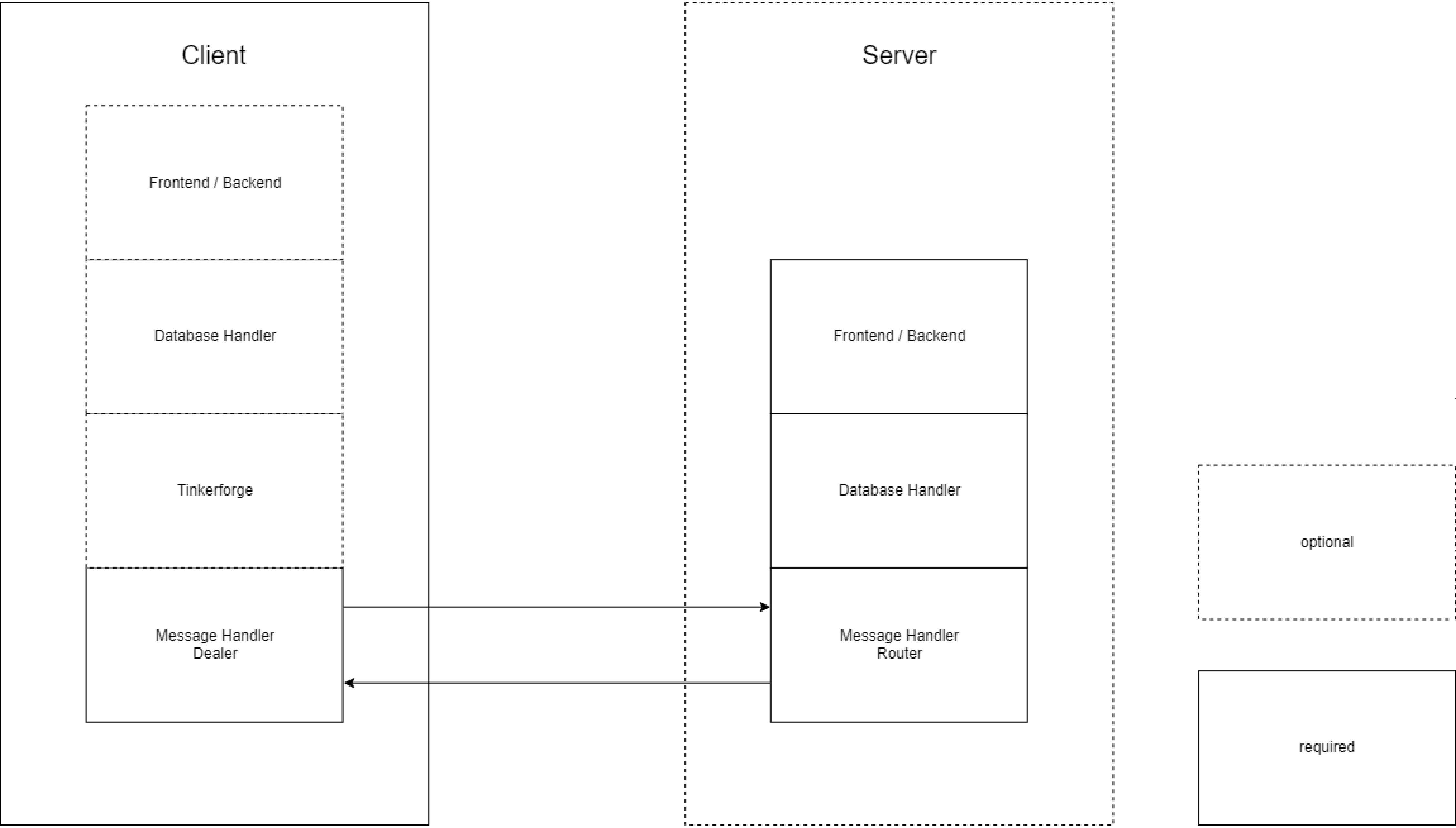
One thing that stuck from the lecture the most was that applications have to be created in a layered fashion. In our case, this means that specific parts of the application can be turned off in case the fog node does not have the required computational power.

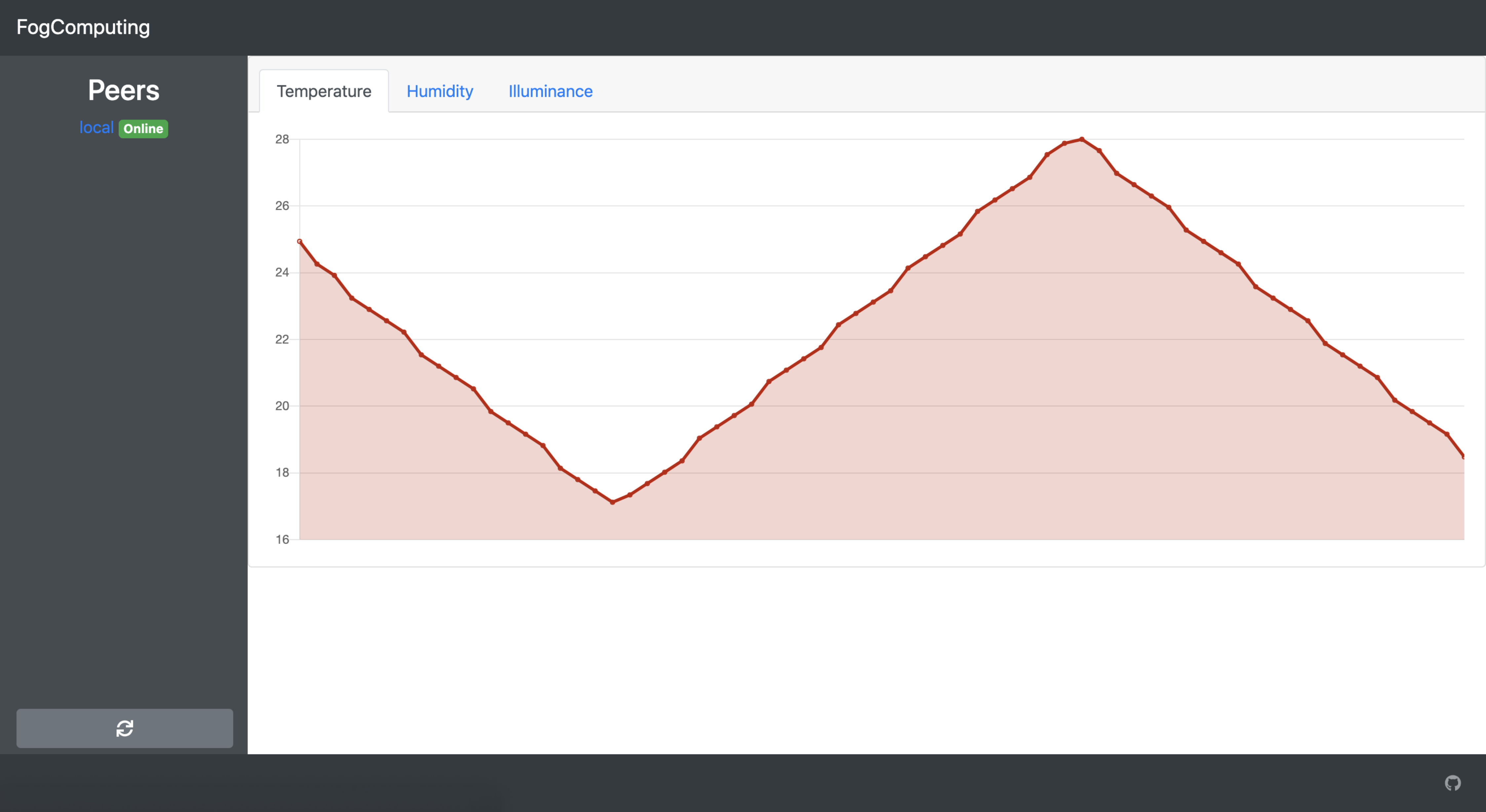
First, we have the message layer, in which we have two defined options for server/client depending on where the application runs. The server part consists of a frontend and a backend combined as one and could be split up into two parts in the future if wanted.
The layered architecture allows us to run a client with frontend as a standalone application without the need of a server.

ZeroMQ will take care of buffering messages from Dealer to Router, this is currently set to unlimited. The Router won't buffer messages for the simple reason that we will never know whether a Dealer will connect again to the Router and is implemented like that in ZeroMQ and cannot be changed. It does make sense as the Router will have an unspecified amount of Dealers and buffering all messages for all Dealers could potentially kill the Router rather quickly. Transmitted data is obviously saved in a database.

As running an extra database, like MongoDB, can be quite expensive on small devices and for prototyping purposes we decided to use an in-memory database called Loki.js Without a lot of effort Loki.js can be changed to MongoDB as their defined functions are rather similar. On top of that we created a class called DatabaseHandler which can easily be switched for another implementation without needing to change any code in the rest of the application.

3. Architecture View - Logical



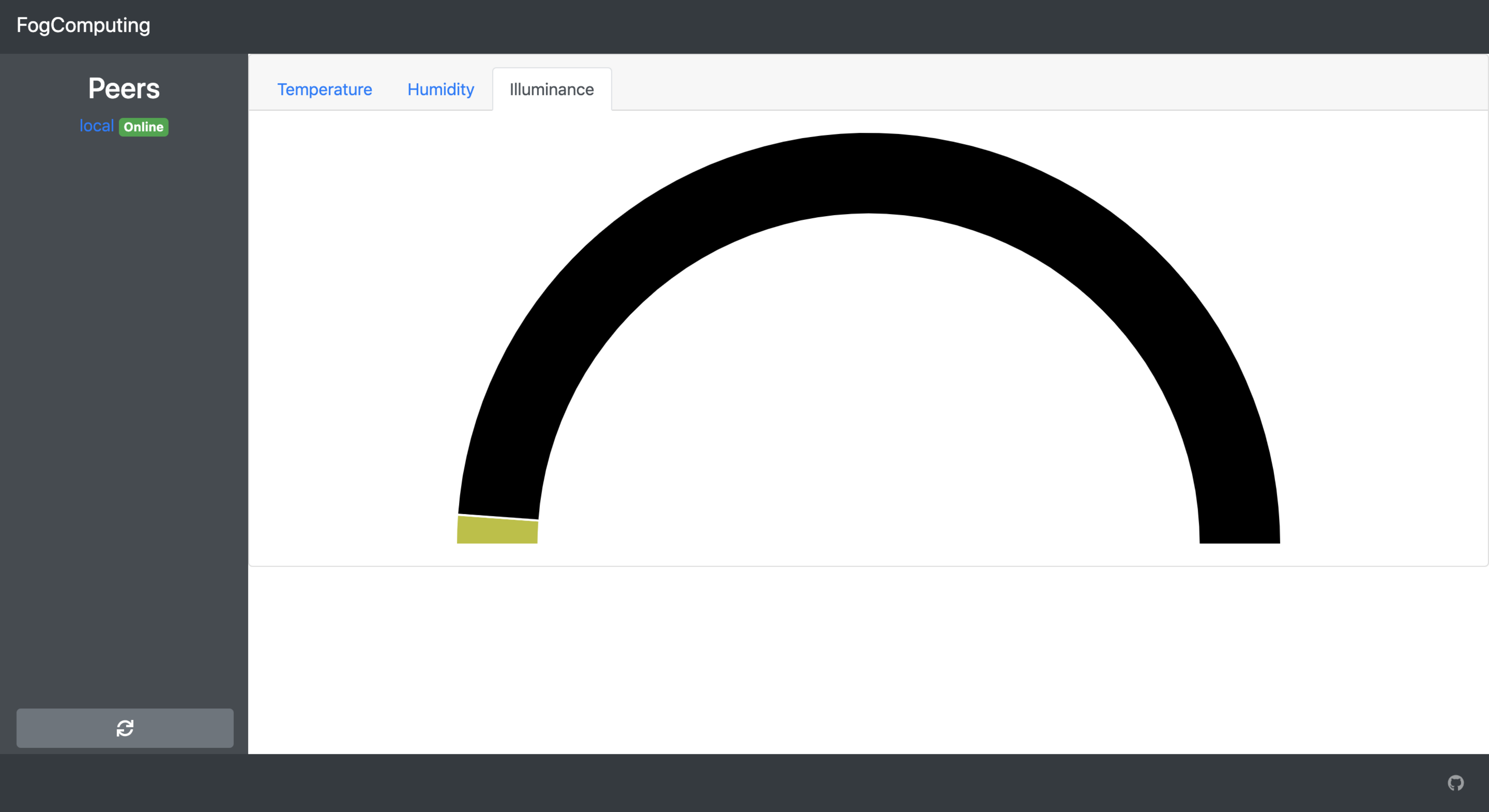


FogComputing

Peers

local Online

Temperature Humidity Illuminance

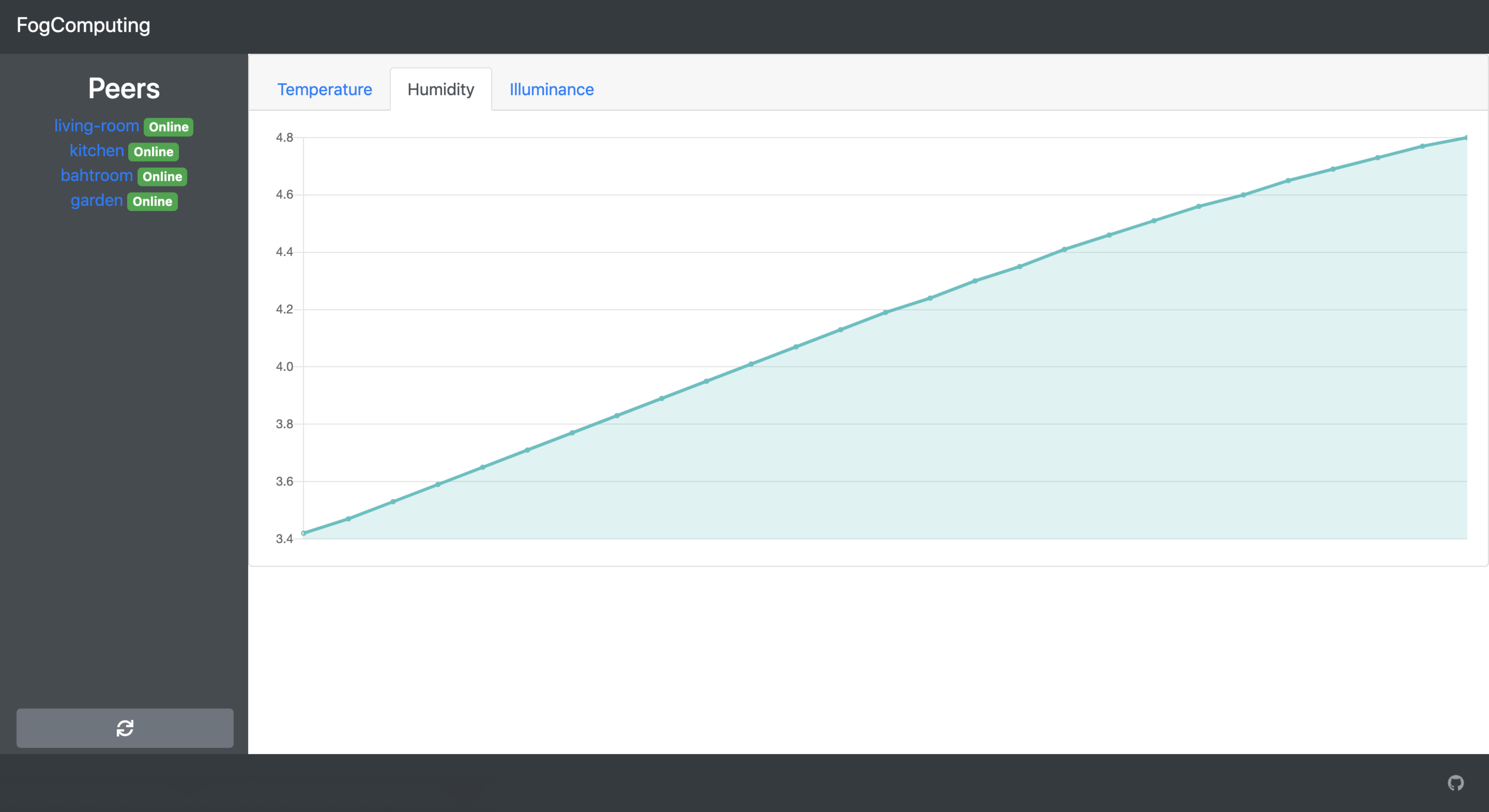


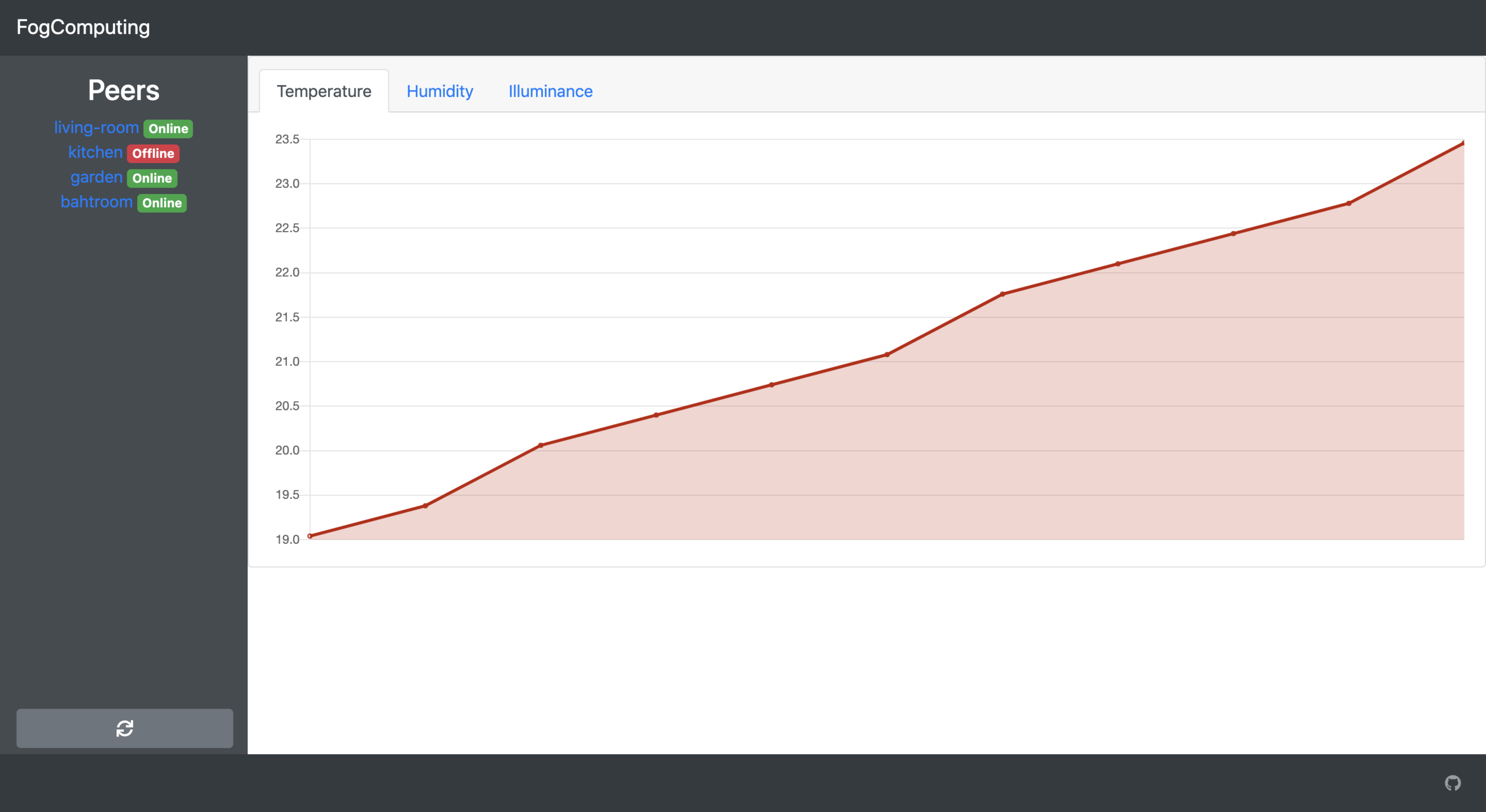
FogComputing

Peers

local Online

Temperature Humidity Illuminance


FogComputing
Peers
living-room Online
kitchen Online
bahtroom Online
garden Online
Temperature
Humidity
Illuminance
4.8
4.6
4.4
4.2
4.0
3.8
3.6
3.4


FogComputing
Peers
living-room Online
kitchen Offline
garden Online
bahtroom Online
Temperature
Humidity
Illuminance
23.5
23.0
22.5
22.0
21.5
21.0
20.5
20.0
19.5
19.0