# Requirements:

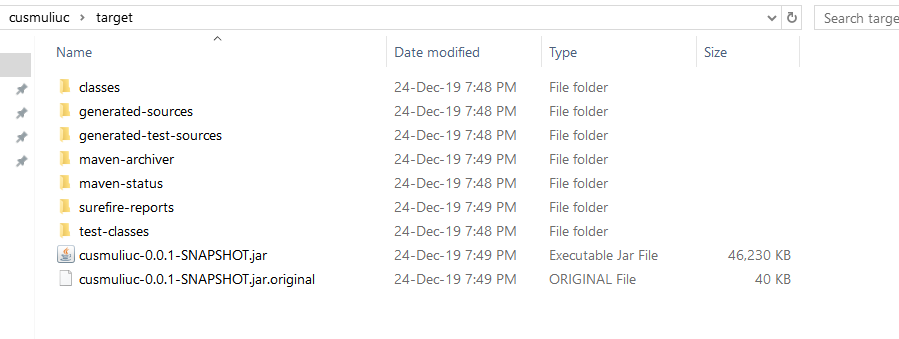
* The endpoints should be written in REST.
* Users can query the **type of metrics** that are stored in the system, also add, modify and delete them; a type of metric can be the following: CPU, GPU, NETWORKING etc.
  + Each metric should have the type of information stored, because each metric has a different method of measurement.
  + The information type can be Percentage (in %, e.g.: 100% utilization of CPU) or in Mb (e.g.: 1024 MB used of NETWORK).
* Users can create, edit and delete a **metric** that is associated with a metric type.
* Users can query the **metrics** available by the type of metric and they will receive a response.
  + Then can also specify a timeline (start date and end date) that the metrics should have but this is not mandatory.
    - If start date is not filled it will be taken with UNIX time 1.
    - If end date is not filled it will be put with current system date.
  + Then can also specify a time difference between the metrics (e.g.: is the time difference is 60s then the difference in time between each metric position should not be lower than that).
    - If the time difference is not put we can return data as we take it from the database.
* Users can also visualize multiple metrics at once, for all the metric types in the system.
  + The same functionality as the above bullet should be present.

# Tools used:

* Spring boot with Java 8
* Maven
* Google Guava
* Spring Test
* Spring Data and JPA
* H2 database
* Lombok
* Spring Web
* Swaggers
* Intellij

# How to start app:

The app runs an embedded tomcat server. Simply go to project root, shift right click to open terminal and write “**mvnw clean install package”**, there is no need to install anything. This command will run the build, tests and create the jar inside the **target** folder



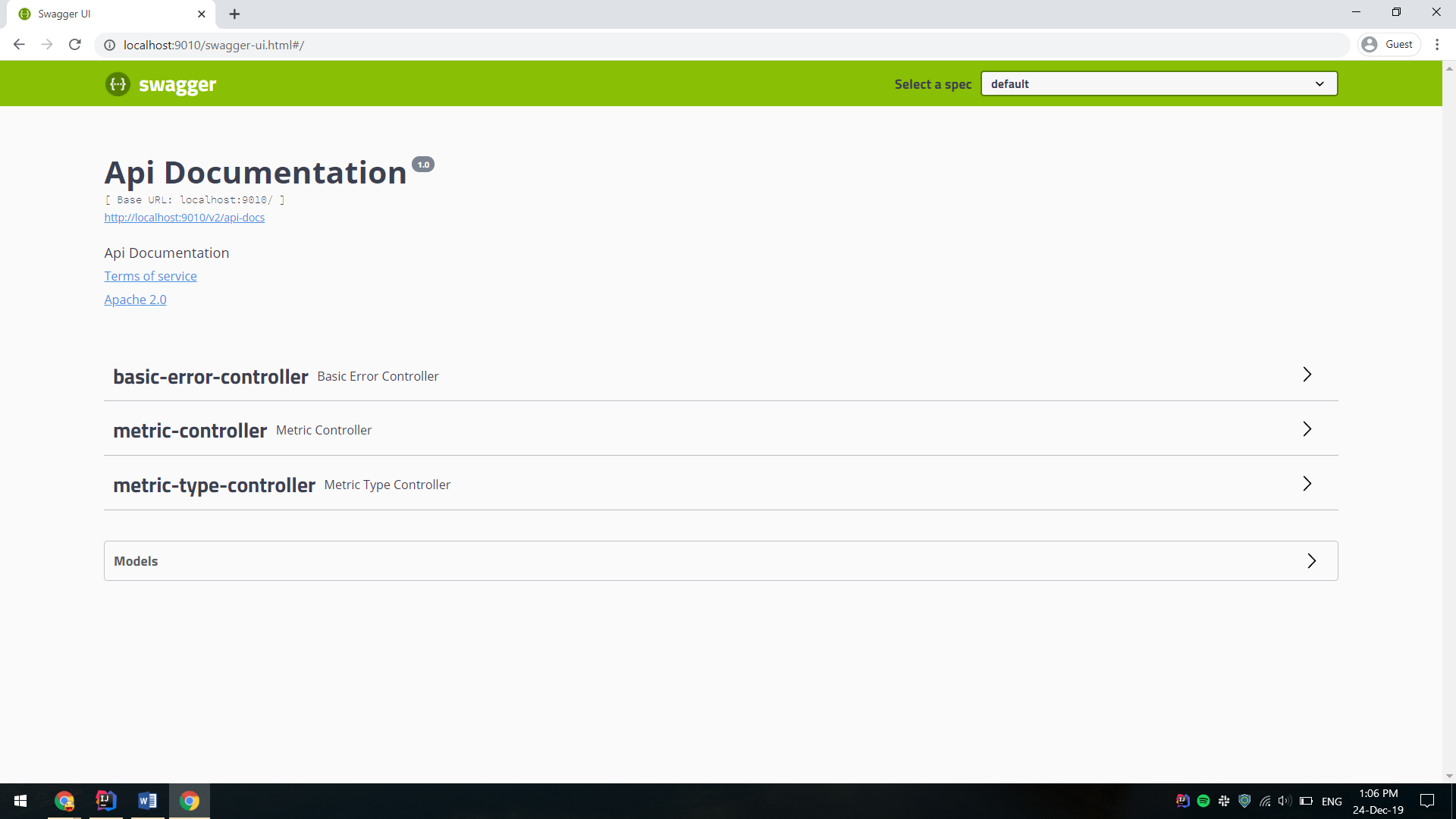
Then open a terminal and write **java -jar cusmuliuc-0.0.1-SNAPSHOT.jar** and the server should start.

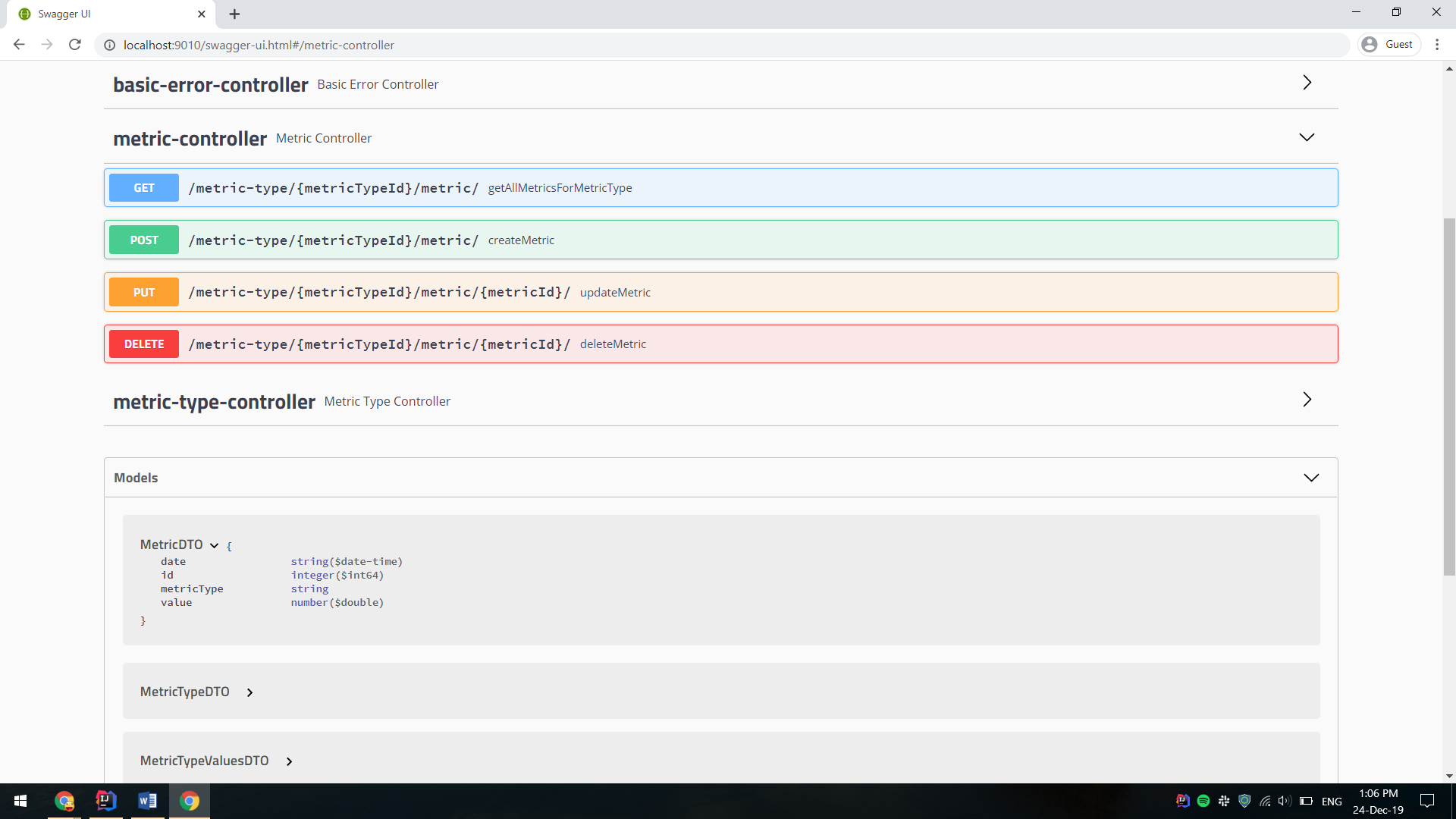
# REST API interface:

Rest API interfaces can be seen at this link with the app running: <http://localhost:9010/swagger-ui.html#/>

Swagger UI creates interface descriptions and also helps with calls to the server.

Inside the project there is a file called swagger.json that contains all the interfaces calls.



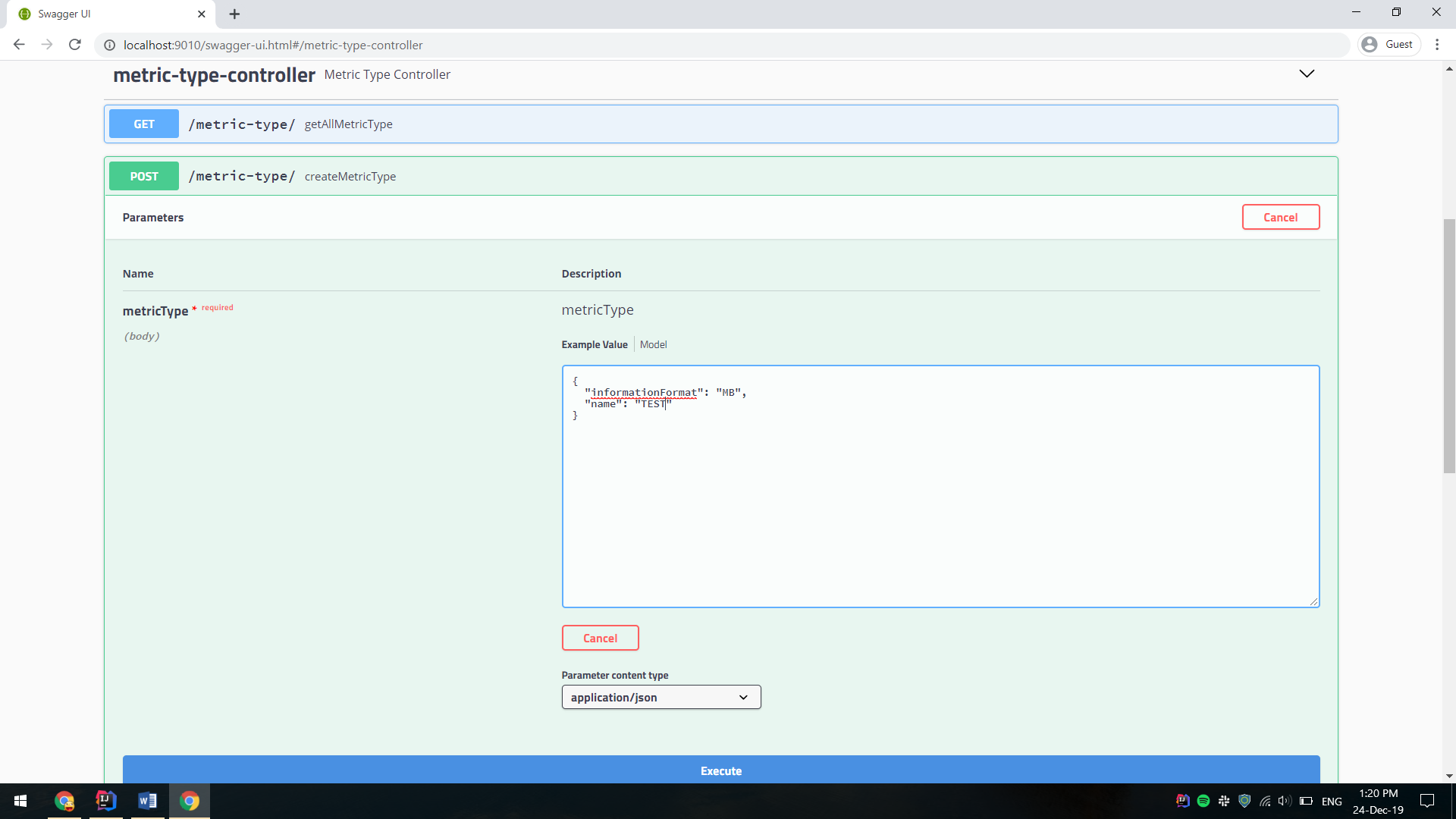


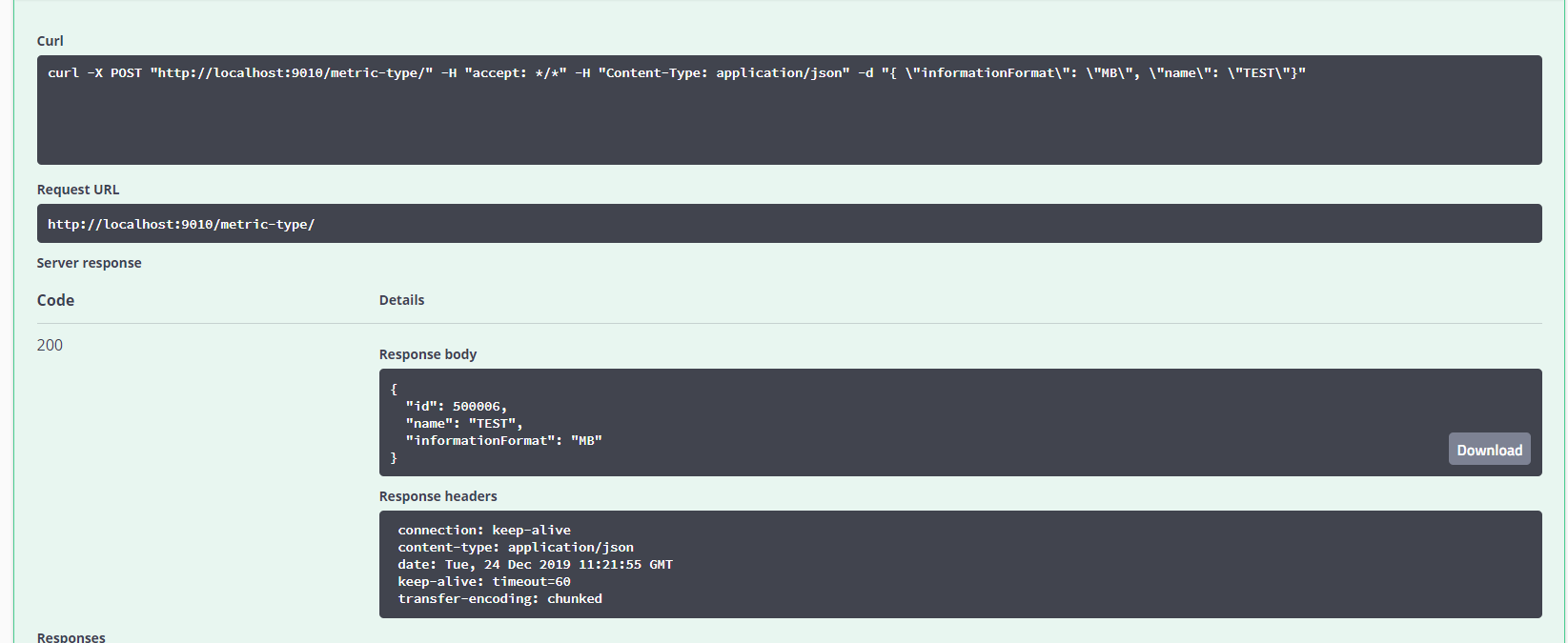
# Testing:

* I created unit tests for the application, they are in the test folder.
* I tried to have as best coverage as possible.
* The application is almost all covered with unit tests, except some classes that I considered it was not needed.

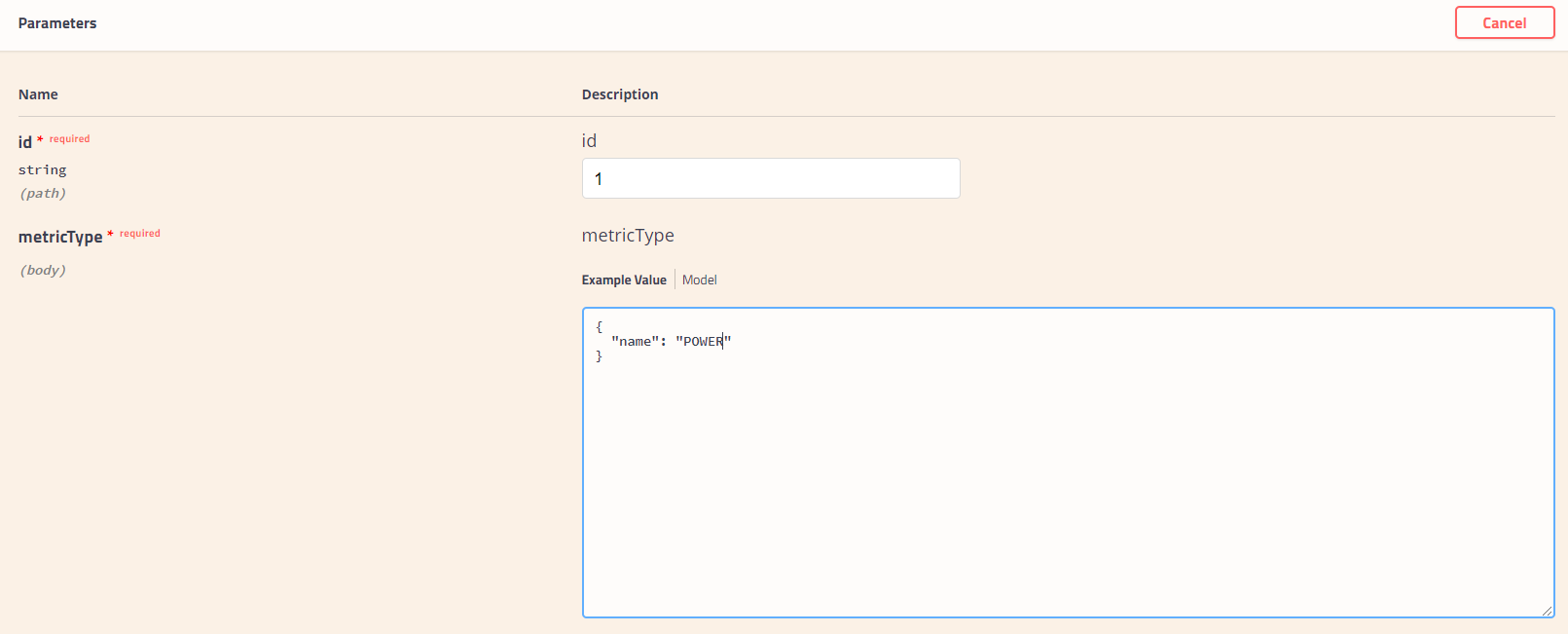
# Calls:

* MetricType – object that stores information about the type of metric, e.g.: CPU, GPU etc.
  + GET **[/metric-type/](http://localhost:9010/swagger-ui.html" \l "/operations/metric-type-controller/getAllMetricTypeUsingGET)** returns a list of all metric types
  + DELETE [**/metric-type/{id}**](http://localhost:9010/swagger-ui.html#/operations/metric-type-controller/deleteMetricTypeUsingDELETE) it will delete a metric type from system
  + GET [**/metric-type/{id}/**](http://localhost:9010/swagger-ui.html#/operations/metric-type-controller/getMetricTypeByIdUsingGET) returns a specific metric type, e.g.: [**/metric-type/1/**](http://localhost:9010/swagger-ui.html#/operations/metric-type-controller/getMetricTypeByIdUsingGET) should return CPU with id 1 and Percentage as information type.
  + POST **[/metric-type/](http://localhost:9010/swagger-ui.html" \l "/operations/metric-type-controller/createMetricTypeUsingPOST)**



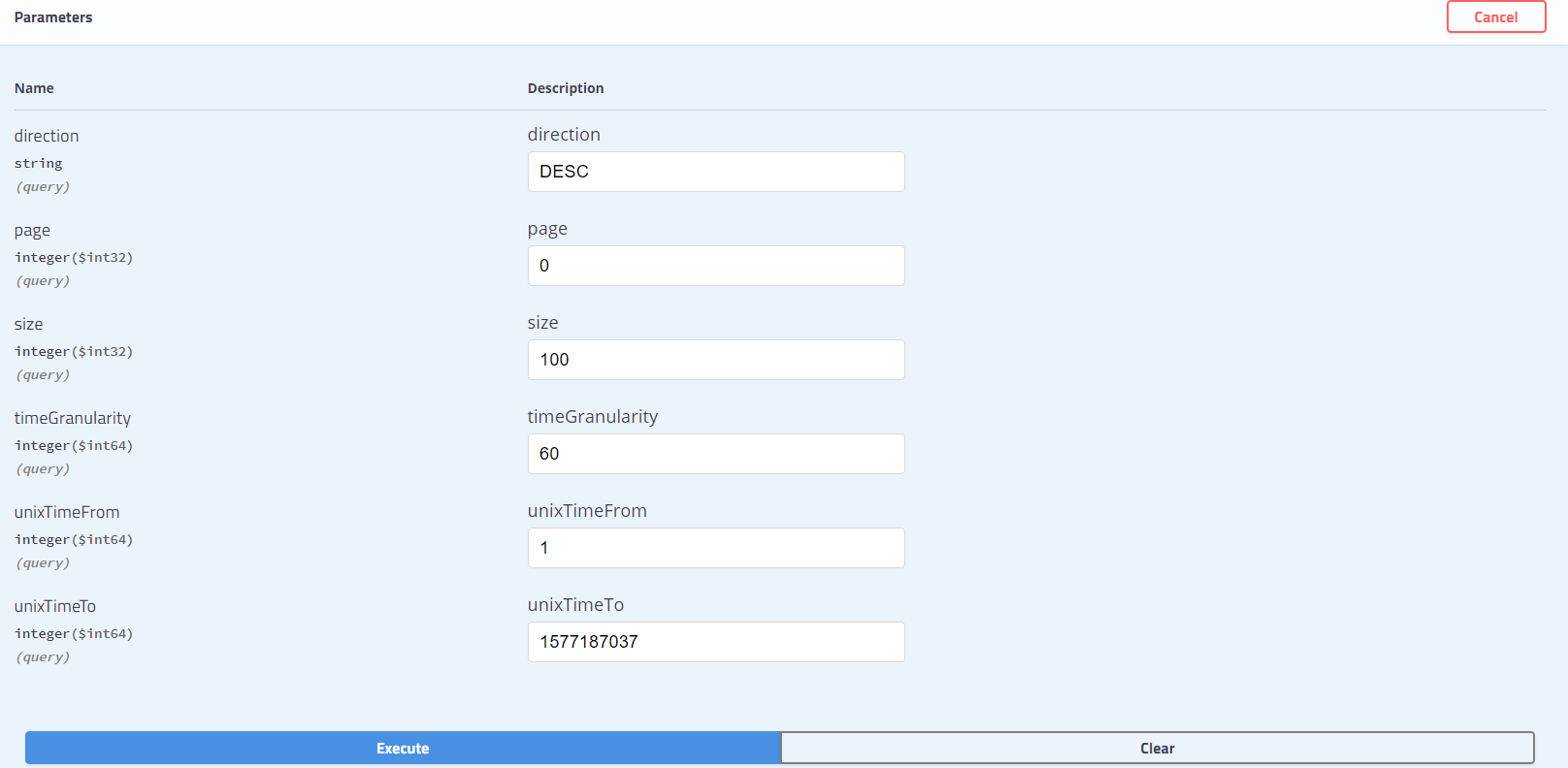


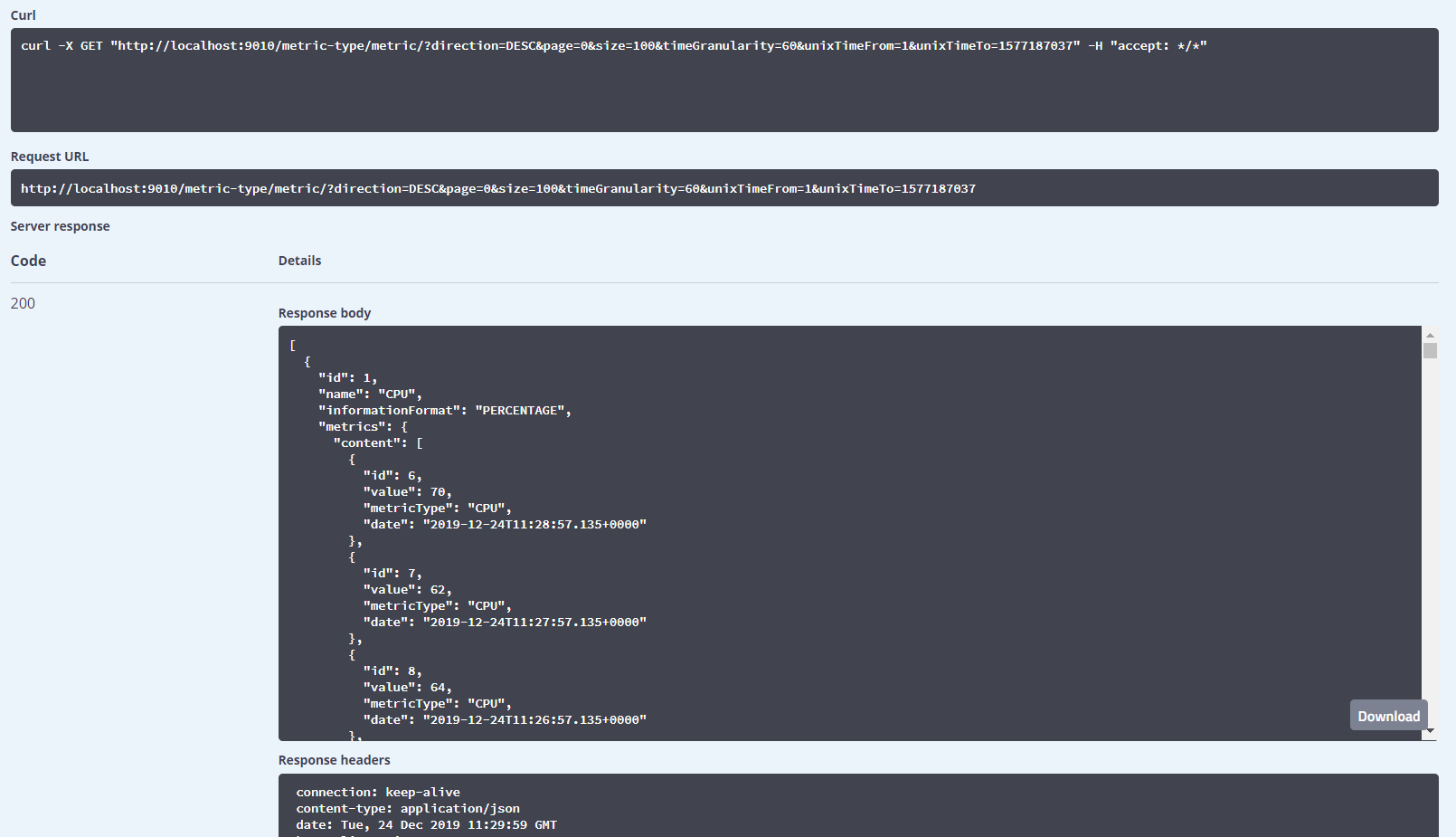
* + PUT [**/metric-type/{id}**](http://localhost:9010/swagger-ui.html#/operations/metric-type-controller/updateMetricTypeUsingPUT) update a metric



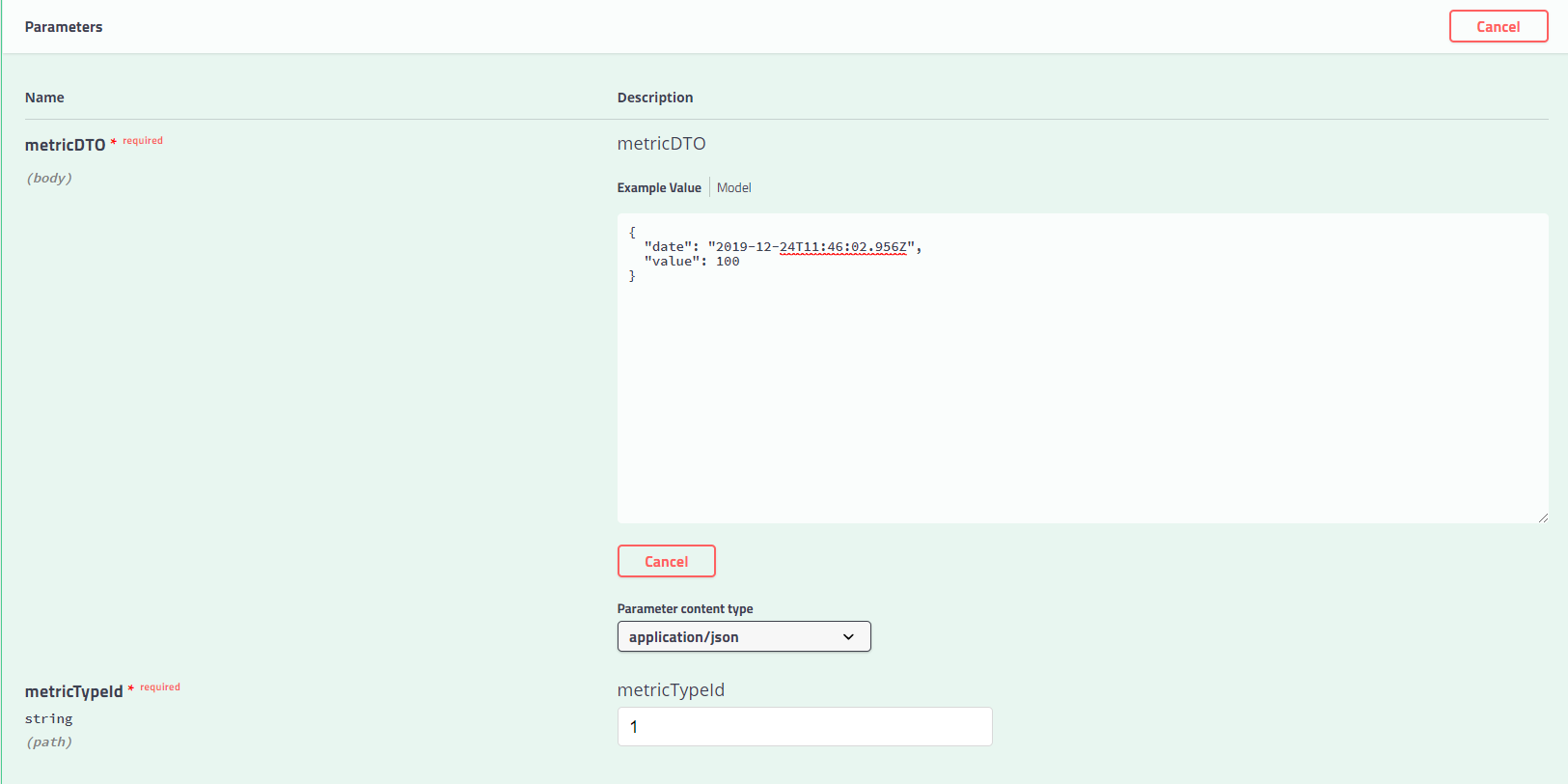


* + GET [**/metric-type/metric/**](http://localhost:9010/swagger-ui.html#/operations/metric-type-controller/getAllMetricTypeMetricsUsingGET) this will return a list of all the metric types but this endpoint also gives the metrics that a type has, the metrics are paged. The possibility of configuring the params will be described in the GET **[/metric-type/{metricTypeId}/metric/](http://localhost:9010/swagger-ui.html" \l "/operations/metric-controller/getAllMetricsForMetricTypeUsingGET)** that is in the Metric controller as they are very similar.



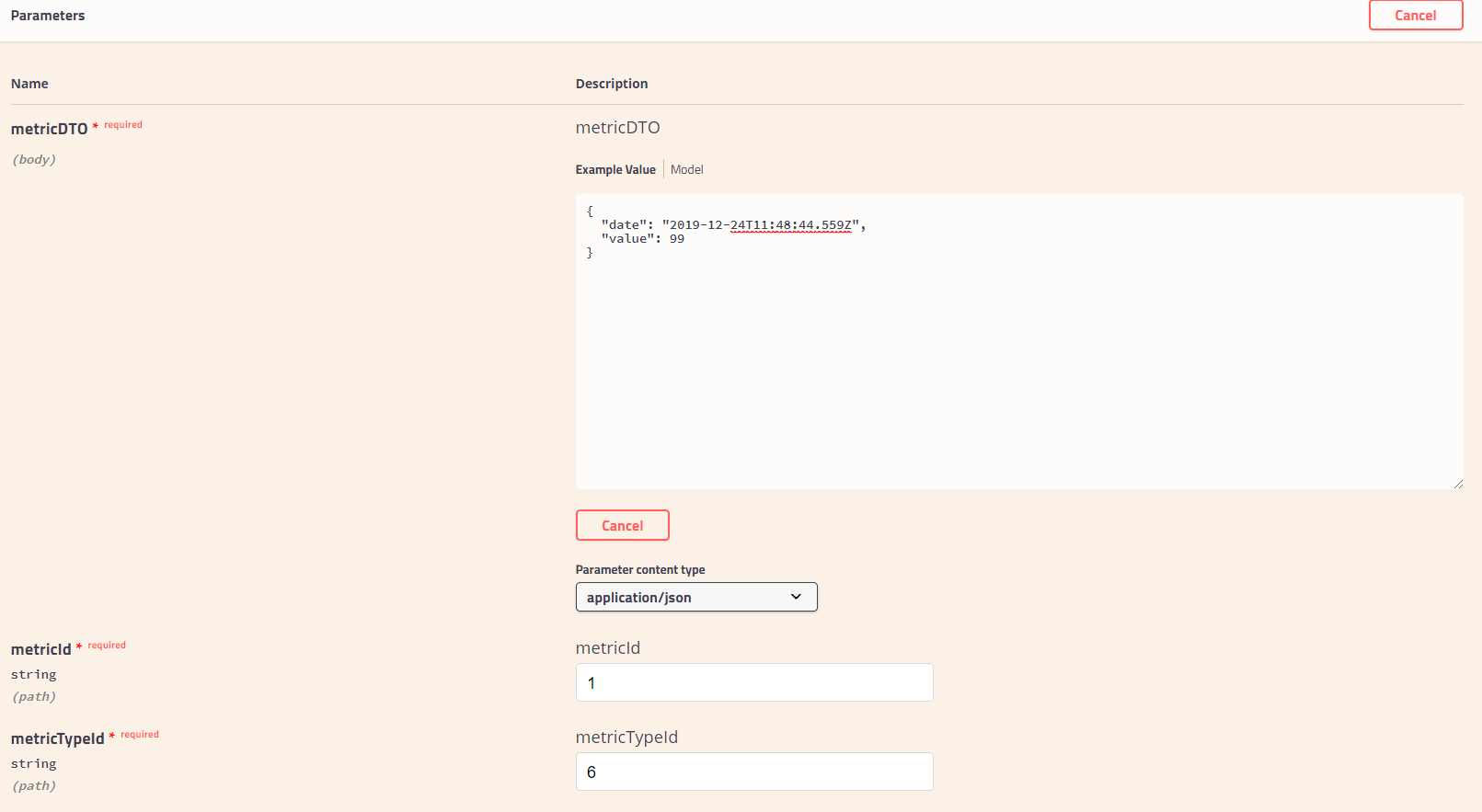


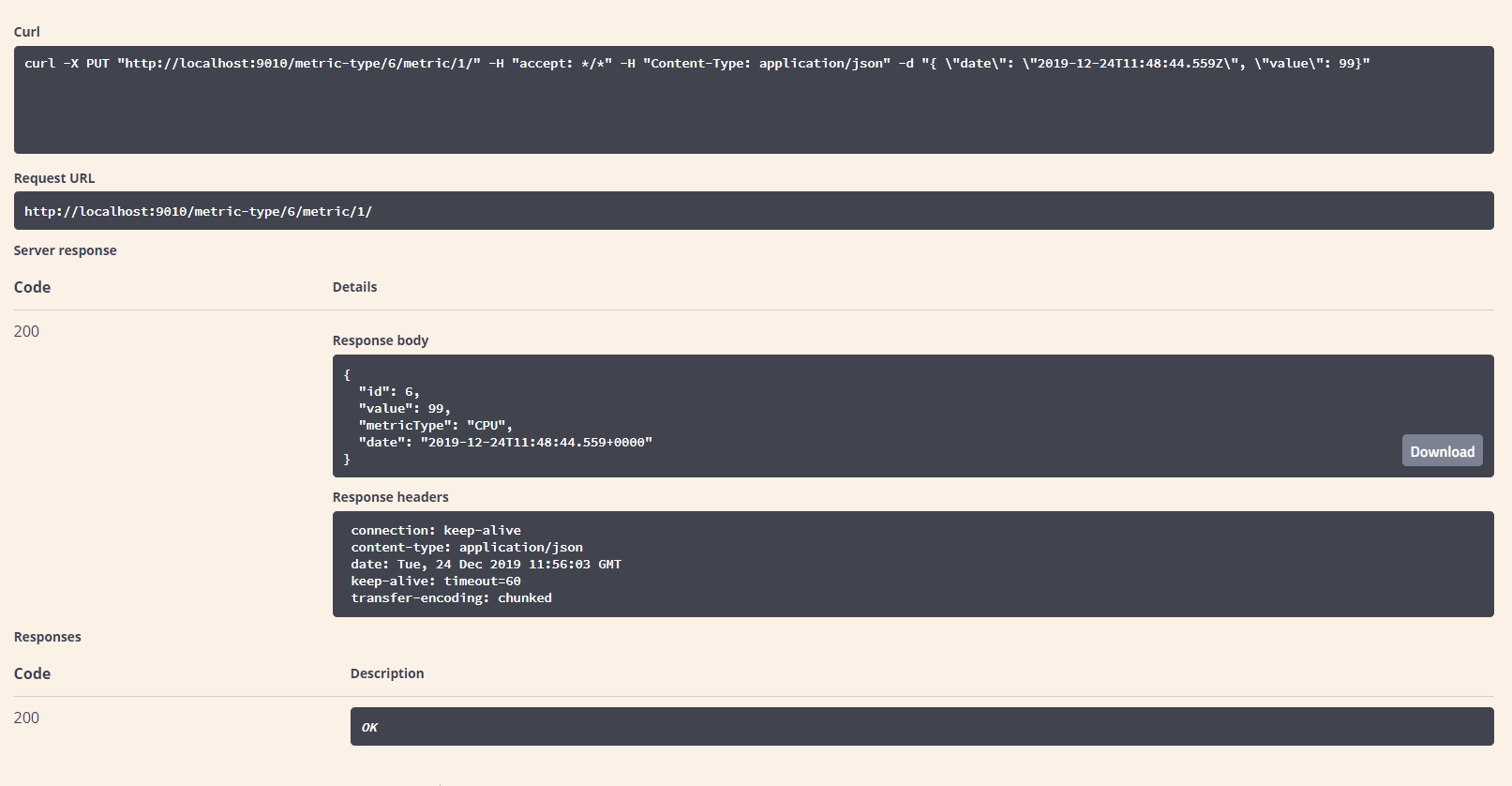
* Metric – object that stores a value at a point in time and it is associated with a MetricType
  + DELETE [**/metric-type/{metricTypeId}/metric/{metricId}/**](http://localhost:9010/swagger-ui.html#/operations/metric-controller/deleteMetricUsingDELETE)it will delete a specific metric for an id that is associated to a metric type that is identified by id.
  + POST [**/metric-type/{metricTypeId}/metric/**](http://localhost:9010/swagger-ui.html#/operations/metric-controller/deleteMetricUsingDELETE)it will delete a specific metric for an id that is associated to a metric type that is identified by id.



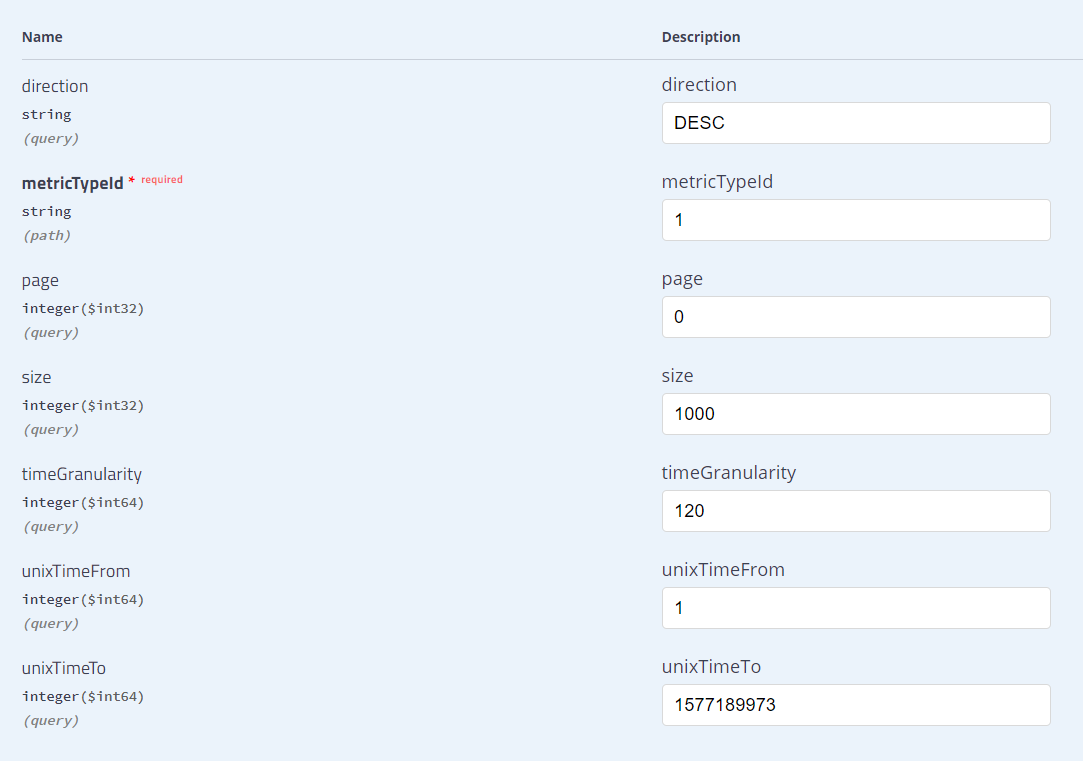


* + PUT [**/metric-type/{metricTypeId}/metric/**](http://localhost:9010/swagger-ui.html#/operations/metric-controller/deleteMetricUsingDELETE)[**{metricId}/**](http://localhost:9010/swagger-ui.html#/operations/metric-controller/updateMetricUsingPUT)it will delete a specific metric for an id that is associated to a metric type that is identified by id.

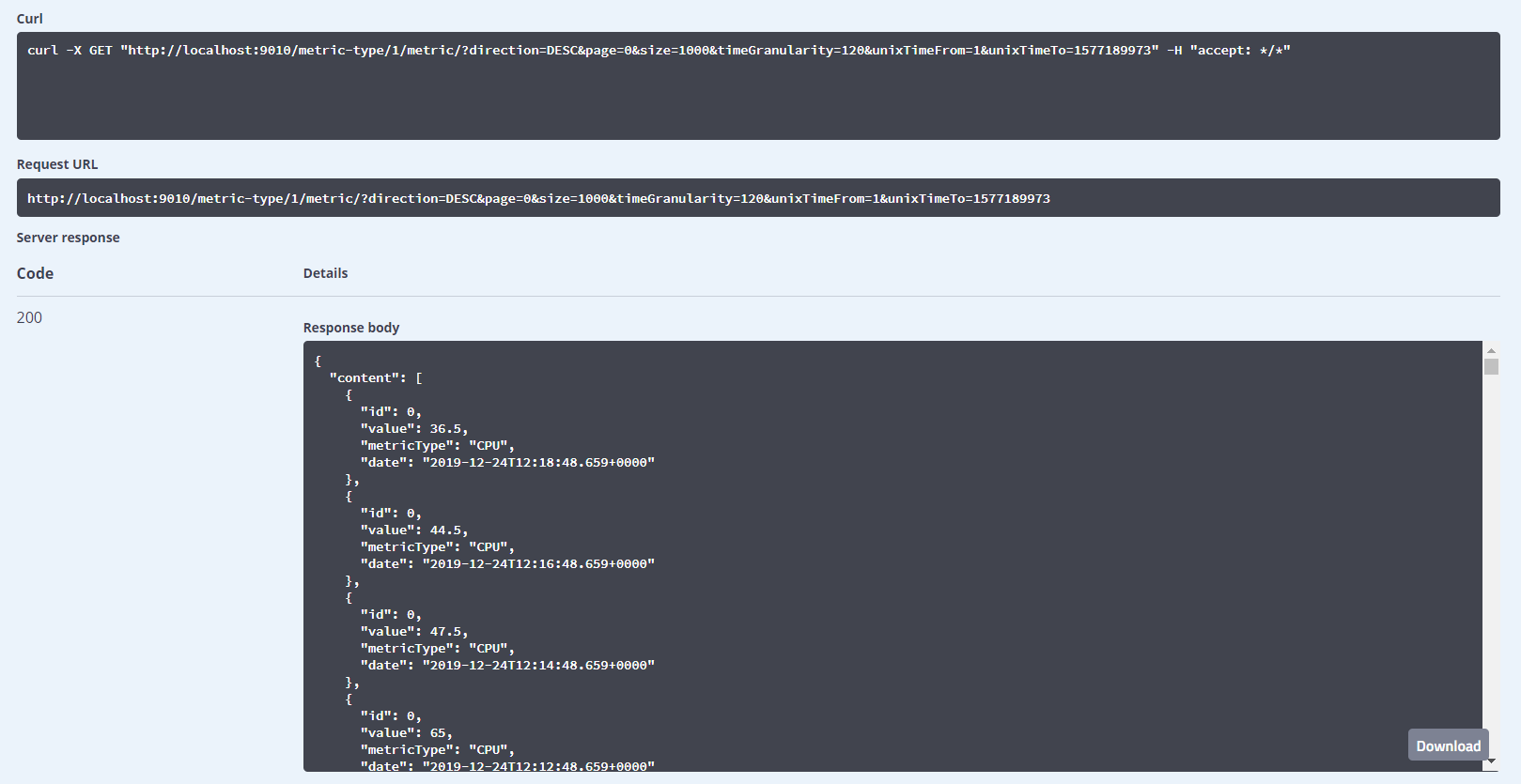


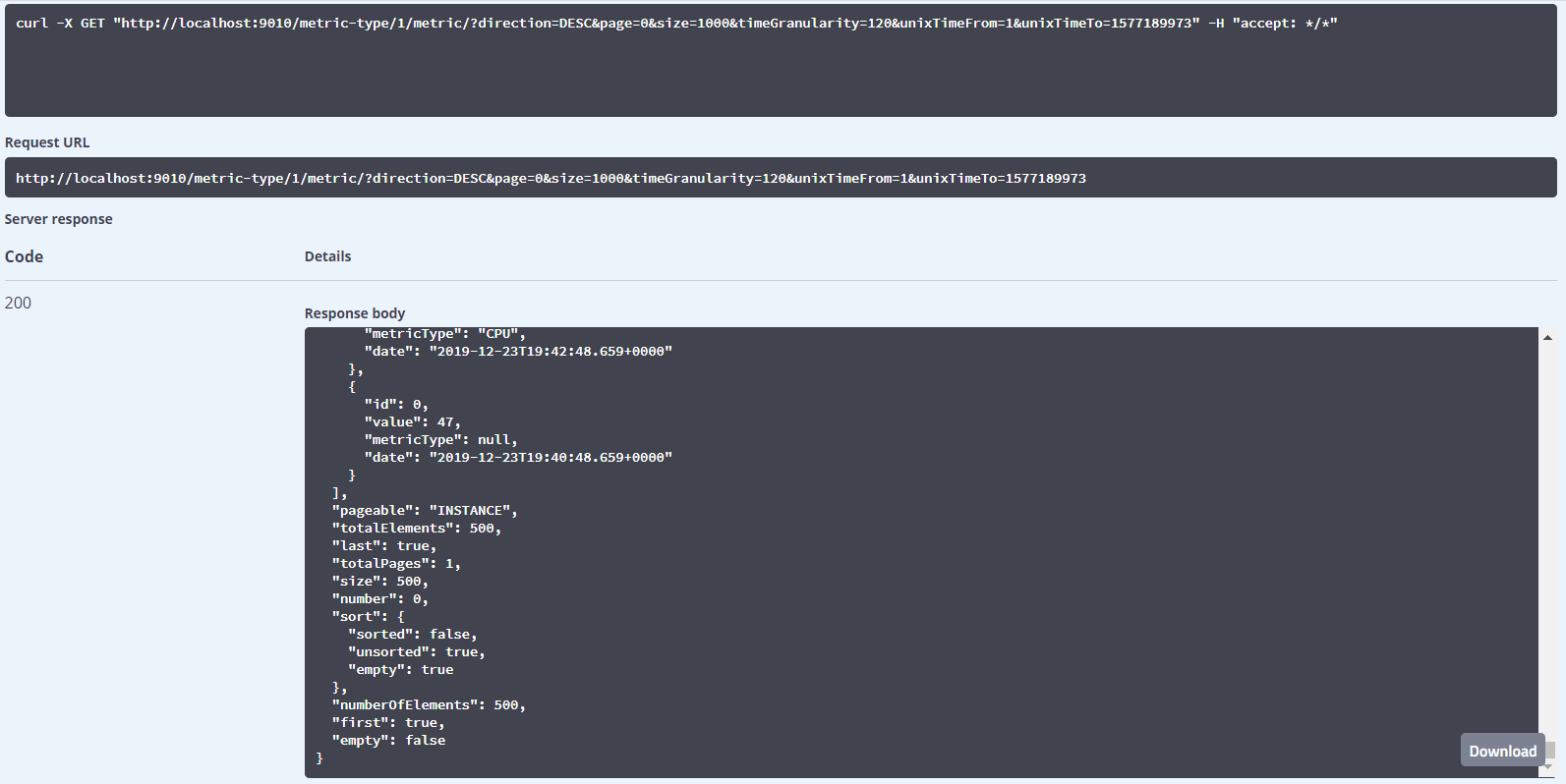


* + GET [**/metric-type/{metricTypeId}/metric/**](http://localhost:9010/swagger-ui.html#/operations/metric-controller/getAllMetricsForMetricTypeUsingGET) get metric paged values for a specific metric type. Parameters:
    - direction: can be „ASC” or „DESC” and it will sort by the filed date in the object. Not mandatory, defaulty is „DESC”.
    - metricTypeId: the metric type id that the metrics must have, e.g.: 1. MANDATORY
    - page: the page number, start from 0. Not mandatory, default is 0.
    - size: number of elements in a page, e.g.: 10. Not mandatory, default is 10.
    - unixTimeFrom this is the unix time that all the metrics should happen after it. This is not mandatory, default is 1.
    - unixTimeTo this is the unix time that all the metrics should happen before it. This is not mandatory, default is current date time, e.g.: 1577189673.
    - timeGranularity: this is also an unix time, it can be any number, for example: 60,120,360 and so on. This get translated in the following, for example 60, meaning that for every metric there should be a minimum distance of 60s between them; if there are values that are at a lower distance the system will create a time and value average.
      * E.g: we have 100 matrics at 1 min apart from each other, if the timeGranularity is 60 then the system will not do anything but if the distance is 2 minutes it will result in returning 50 metrics average two by two.



This example show 1000 elements at 120 second from each other, we should get around 500 of them.



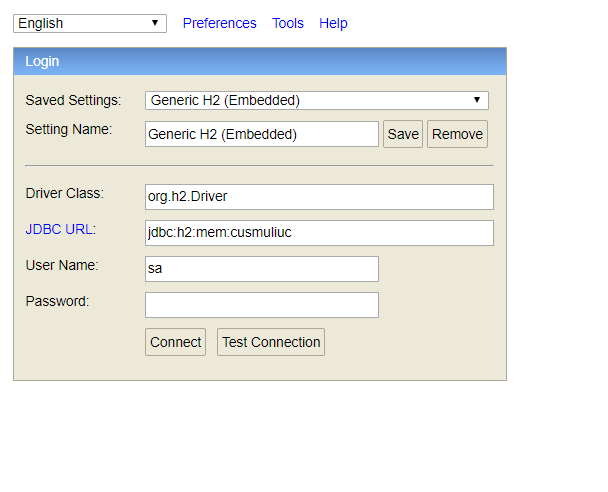


As it can be seen it returned 500 elements as predicted.

# Database:

For development reasons I used an embedded, in memory database called H2.

The dashboard can be accessed at <http://localhost:9010/h2/>



There is no password, simply press connect.

I created an initialization script that will create 100.000 metrics for 6 metric types, CPU(which values are only in %), RAM(in MB), GPU (in %), DISK(in MB) and NETWORK(in MB).

Values are generated randomly and the date is set at every minute, from the current start of the program counting downwards 100.000s. I did this so I can have test values.

Here comes the question if this embedded database is suited for our application which should run in a highly scalable and fault resistent environment...and the answer is, of course, no. I chose this database because it’s simple and easy to develop, as a POC. Since we have JPA as a layer and we do not have native queries the database layer can be changed quite easily with very minimal to none effort.

# Front-end:

The tools:

* I would use either Angular or React, depending on the complexity, if it were a big application I would use Angular, otherwise I would go with react.
* I’m not sure what plotting libraries are available for both of them but considering they are very popular front-end frameworks I would say it won’t be a problem.

Looks:

* On the first page I would create a graph for each MetricType that would get updated in real time.
  + The users would be able to select a certain period of time on the graph via mouse and this would make the period bigger.
* Things like CPU utilization can be represented with a circle and an arrow that show the utilization percentage.
* Users would see the MetricType names as tags and when they press on them all the charts would change to reflect the specific tag clicked and the data associated with it so the user would get more info.