**Backend overview**

**DCS Project**

October 2020

*David Vicente Fuentes*

1. **Introduction**

The purpose of this document is to provide a detailed architecture overview of the DCS Backend.

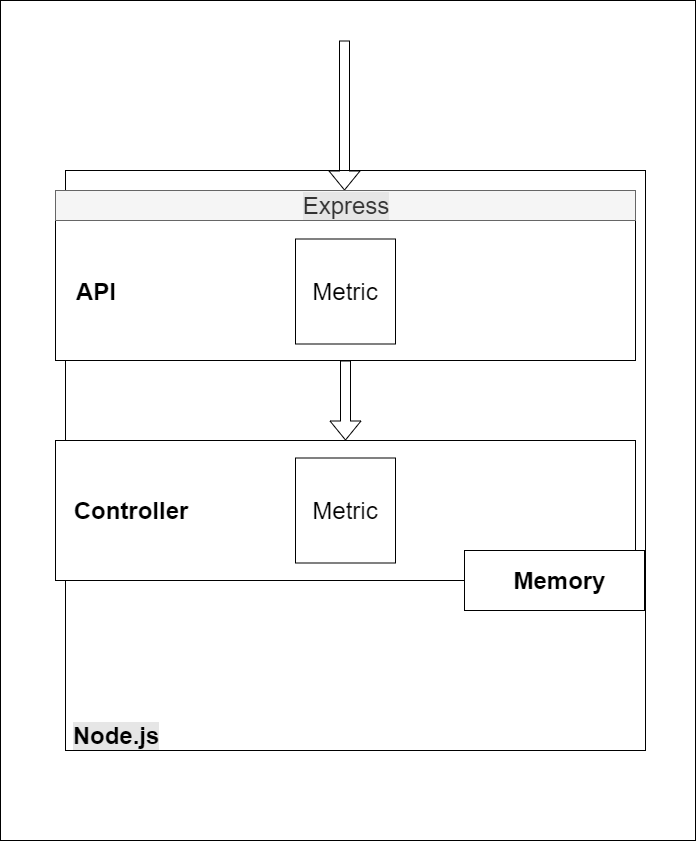
The intention is to help future developers to understand current architecture, so they can understand it quickly and add new functionalities using the same structure and taking advantage of the design.

This document has three main sections:

* Architecture overview that includes:
  + A high-level diagram that shows the design of the architecture that has been used in the implementation. It will help to guide during the reading of each part.
  + A brief explanation on the API layer that is held by NodeJS
  + A more detailed explanation on how has been done Controller layer since is the one that is managing business logic; therefore, the most important part here since will be the one that should meet the given requirements.
* Future extensions that includes a detailed description on which rules should be followed in case of extension in several scenarios:
  + New features related to Metrics will be added
  + New features of new entities added
  + New source of data needed: Database, file…
  + Splitting when logic gets very complex

This document should be maintained as well as the solution implemented in order to be used as an up to date and useful guide.

1. **Architecture overview**

****

The diagram above shows the architecture of the backend server. It is implemented over Node.js, which is a JavaScript runtime environment for the backend side.

* **API layer:** The express framework manages the HTTP requests that arrive to the server. It redirects them to the correspondent module in the *API layer*, based in the HTTP method + path.

In this case, we only have the ‘Metric’ module, but if the server grows, there should be one module for every logic entity identified in the analysis.

The API layer is in charge of the security of the API, validate parameter types, and route the request to the controller module that will implement the business logic.

To see the endpoints available, go to *API doc folder.*

* **Controller layer:** This layer is the one that manages the business logic.

It´s important to highlight that in order to do this solution more flexible, controller layer should be unaware of:

* Output interface (REST API, GraphQL, web sockets …)
* Information store system used (files, relational database, no-relational database, etc.).

As in the API layer, one module for each entity identified should exist.

All the business logic implemented should be covered by unit tests since this business logic is the one that implements all the client requirements (you can find unit tests coverage of the project in the *Unit test coverage* document). A high minimum coverage ensures that the code is reliable and easier to maintain.

In this case, Metric requests will arrive to Metric module in Controller layer, where logic regarding Metric will be done.

Information needed to be stored, will be saved in memory variables, as no databases are needed due to the simplicity of the project.

A cache variable would be used to keep updated the sum of every metric in order to improve the performance of the backend in case of metrics with many entries.

1. **Future extensions**

These rules should be followed in order to keep the current architecture when new endpoints are added:

* If the new feature is related to Metrics entity, if should be allocated in Metric component in the API layer
* Otherwise it should be added as part of a new one created for the correspondent entity

Same rules can be applied for the logic added for new features in controller layer:

* If it is related to Metrics, it will be added in Metrics module
* If it is related to a new different entity, it will be added in a new module that will be created for that purpose.

In case *Controller layer* gets very complex or several modules get dependencies among then, a good practice would be to split this layer in two sub-layers. In that case the rules for splitting would be:

* The most internal one would be in charge of the logic more related with the use and transformation of data.
* The most external one should allocate the high level logic (closer to user requirements).

At the moment that information has to be stored (in database, file, …) a new layer should be created for this purpose, being in charge of the connection and interface with the store system. Logic layers should be agnostic of the store system used, so it can be changed in the future minimizing impact in rest of layers as has been already explained in Controller section.