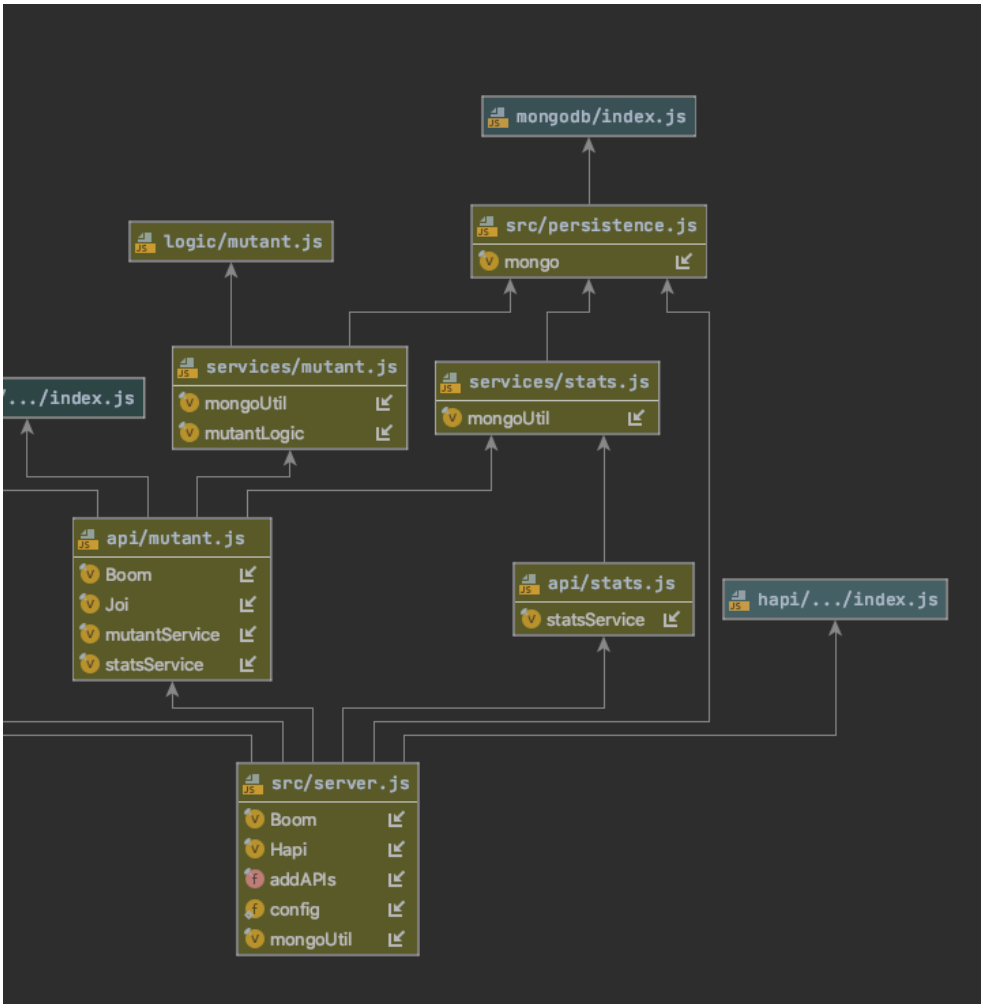
Mutant analyzer

The project was built using Node.JS. An overview of the modules is this:

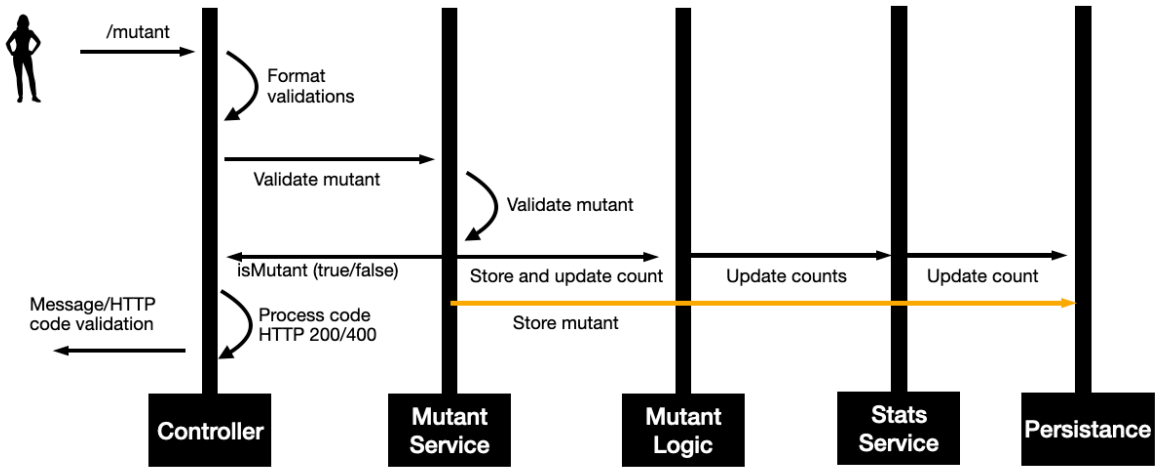


I used the Hapi framework to increase the speed of development and improve code quality.

The server consumes services from 2 controllers: The Stats controller and the Mutant controller, each of one defining its own routes and HTTP methods /mutant and /stats.

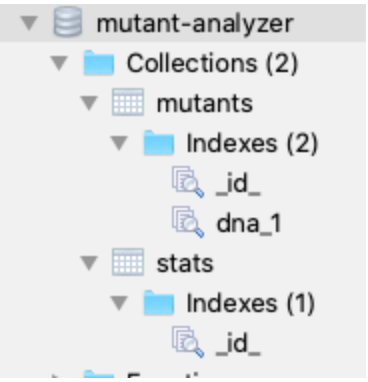
I created 2 services to handle the business logic (services/mutant.js and services/stats.js). They interact directly with the persistence layer. The mutant service consumes a Module called (logic/mutant.js). In this module, I store the matrix analyzing logic (this one is completely isolated and doesn't have access to the HTTP call nor the database logic)

Mutant validation



For a mutant validation, the user should call POST /mutant with the parameters defined in the requirements. The controller validate the format of the request (due to time restrictions, it only validates that the post contains an array of strings, however, additional validations can be added).

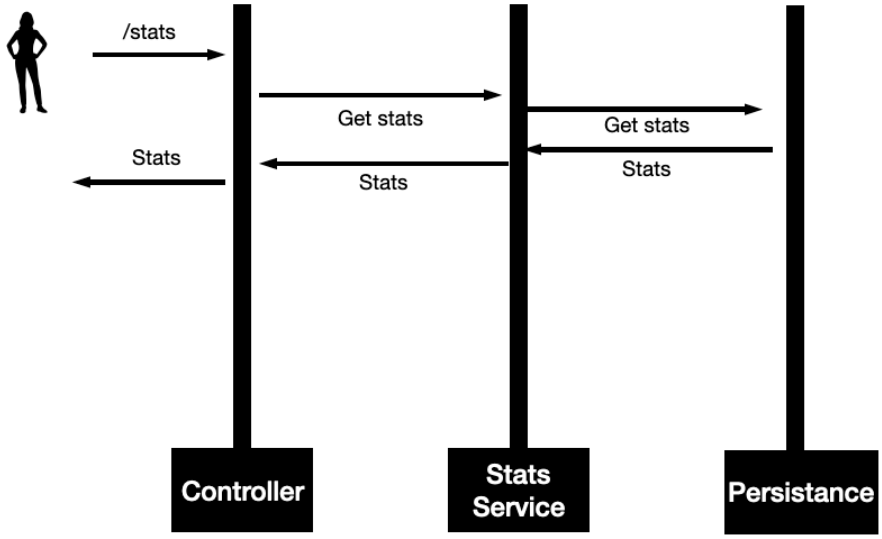
After the validations, the Mutant service is called. This service calls the logic module to validate if the input corresponds to a mutant. Asynchronously the count is updated. We are using a collection with a single object to store the stats. Each of the mutants is stored with his corresponding DNA and the validation result.



Finally, we store the mutant with his information. The Mutants collection has a unique index in the DNA field to guarantee that we don't store repeated documents for the same validation. The service responds as soon as the logic validation finishes: that way the user doesn't have to wait until the database finishes to store the data.

Stats

As I previously mentioned, the stats are pre-calculated each time a mutant is stored. That's the reason why for stats we have a Single document collection with the aggregated stats. The ratio is calculated after the stats service retrieve the human and mutant count from the database.



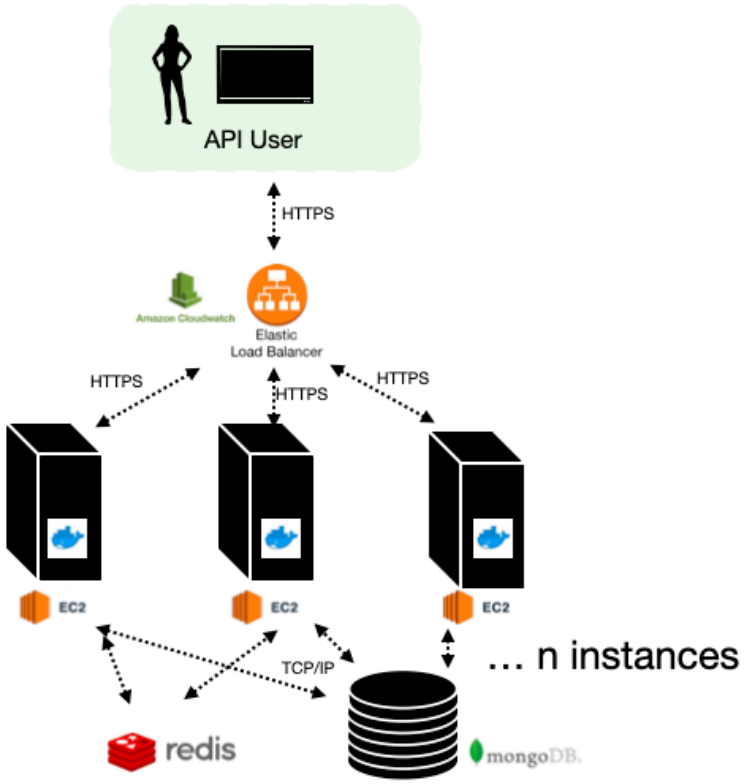
Ideal architecture

Due to time restrictions, and ideal architecture is depicted (it wasn't implemented). For the ideal architecture we would have a load balancer between several processing instances (as needed). As we have to process thousands of requests per second it should scale automatically based on the needs (eg, measuring the processor use with AWS CloudWatch)

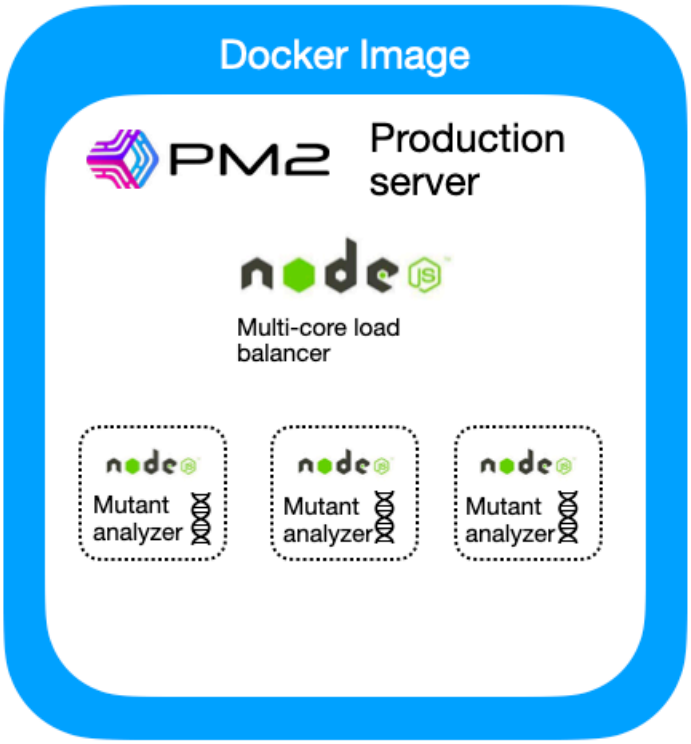
As we could have a horizontal growing pull of servers, we should have a fast database (sharing policies could be defined to make sure it is not a bottleneck).

A Redis cache could be used to store the most common requested DNAs and avoid to process long DNA sequences repeatedly.

Ideally, we could get the result of the processing from the database if the matrix is too big.

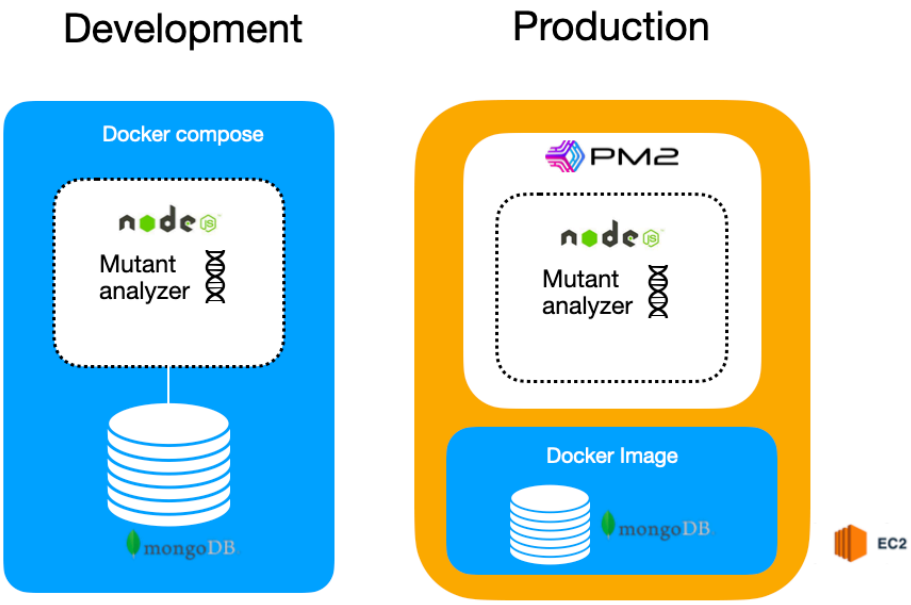


In the ideal architecture we could have a docker image in each AWS instance. A production server handling multiple node instances, taking advantage of the multi threading capabilities of the machine



Implemented architecture

For the development environment I set up docker-compose. Inside the docker-compose the image of MongoDB is set and the node project.



For the production environment I started a PM2 server with the node project running (in AWS EC2). An additional docker image is running with the database image.

Tests

Automated test were build (Unit and Integration). The report can be browsed in this URL: https://slinan2.github.io/mutants/

Algorithm

The algorithm has a complexity of O(N*N). It has to walk for every letter in the matrix. For every letter I check equality with the neighbors in the same column, row and diagonals. A performance optimization could be done splitting the matrix into smaller matrices and doing a multithreaded search, however in these scenarios it is easier to maintain the multithreading in the node server side and not in the algorithms logic.