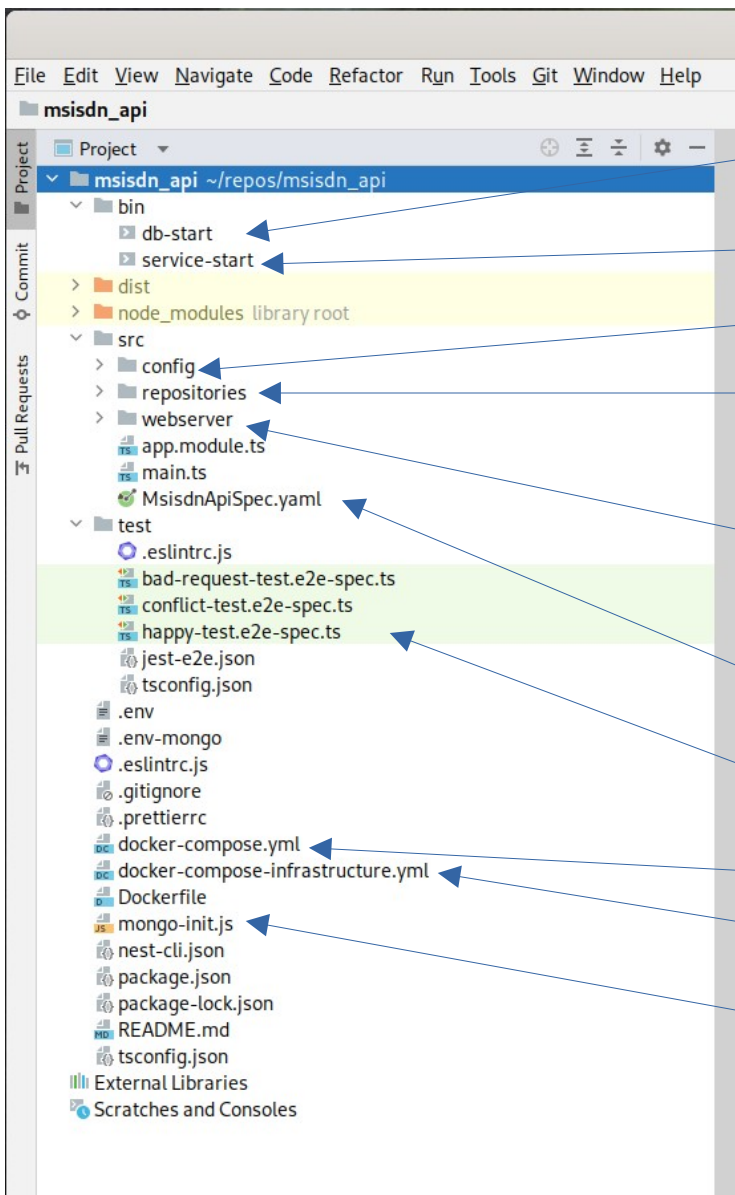


# Msisdn api README

In this document I will briefly describe howto:

1. navigate thru the project
2. run the service
3. run the tests
4. design decisions
5. things that can be improved or are missing

## Navigating thru the project folder structure



The screenshot shows the file explorer of a code editor for the `msisdn_api` project. The project structure is as follows:

- `bin`
  - `db-start`
  - `service-start`
- `dist`
- `node_modules` (library root)
- `src`
  - `config`
  - `repositories`
  - `webserver`
  - `app.module.ts`
  - `main.ts`
  - `MsisdnApiSpec.yaml`
- `test`
  - `.eslintrc.js`
  - `bad-request-test.e2e-spec.ts`
  - `conflict-test.e2e-spec.ts`
  - `happy-test.e2e-spec.ts`
  - `jest-e2e.json`
  - `tsconfig.json`
- `.env`
- `.env-mongo`
- `.eslintrc.js`
- `.gitignore`
- `.prettierrc`
- `docker-compose.yml`
- `docker-compose-infrastructure.yml`
- `Dockerfile`
- `mongo-init.js`
- `nest-cli.json`
- `package.json`
- `package-lock.json`
- `README.md`
- `tsconfig.json`

Annotations on the right side of the image explain the purpose of various files and folders:

- `db-start` starts the mongodb container only
- `service-start` starts the msisdn api container AND mongodb container
- `config` contains yaml config files per environment (user, passwd, etc...)
- `repositories` contains the module for the only repository needed for this project, which is *mongodb*.
- `webserver` contains the webservice and controller logic. *Express* with some nice decorators provided by the *nest* framework
- Api documentation based on OpenApi version 3.0
- All tests inside the test folder are **e2e** tests
- `compose` for this msisdn api service
- `compose` for mongodb (infrastructure)
- When mongodb is started it is **pre-populated** with a list of msisdns for testing. In production mongodb would not run inside a container.

# Running the Service

There are two ways to run the web server. Containerized or not-containerized. In both cases we have to have a mongodb running and in this project, at least for development purposes we run a mongodb container.

## Running the web server not-containerized

build the project *(output will go in the dist folder)*:

**npm run build**

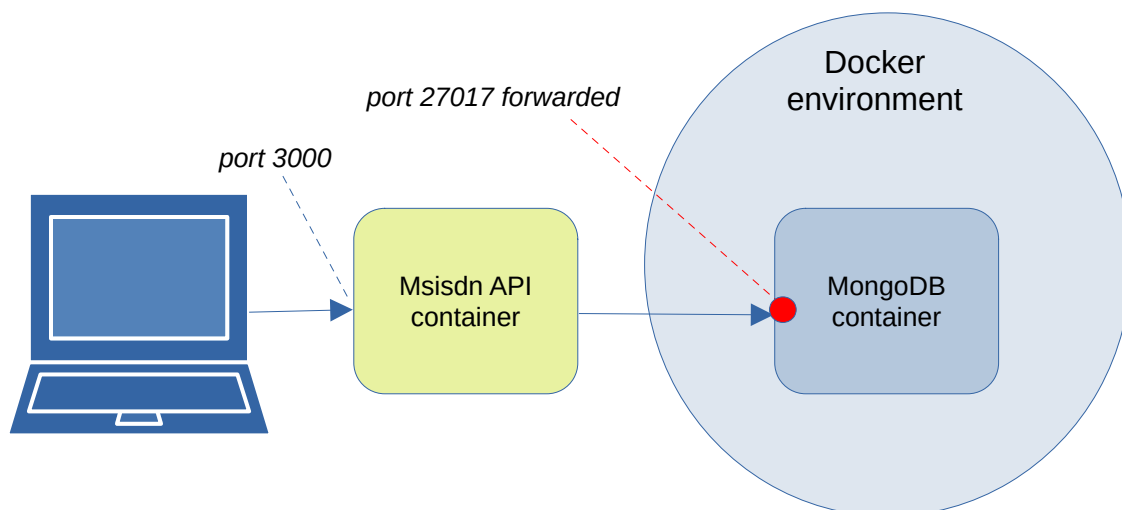
start the mongodb container:

**./bin/db-start**

start the msisdn api web server

**npm run start**

Connect a browser or a tool like Postman to port 3000



## Running the web server containerized

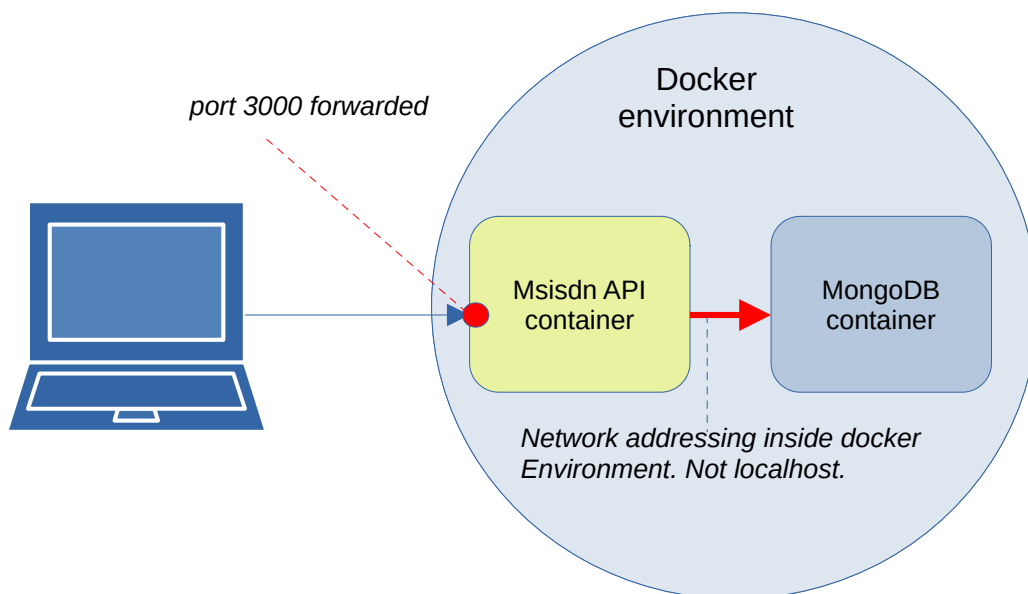
build the project (*output will go in the dist folder*):

**npm run build**

start the web server and mongodb container:

**./bin/service-start**

Connect a browser or a tool like Postman to port 3000



## Running the tests

start the web server and mongodb container:

**./bin/service-start**

There are only end to end tests, no unit tests

**npm run test:e2e**

*Expect the result to look like that below:*

```
rogier@fedora:~/repos/msisdn_api$ npm run test:e2e

> msisdn_api@0.0.1 test:e2e
> NODE_ENV=test jest --config ./test/jest-e2e.json

PASS test/bad-request-test.e2e-spec.ts
  WebserverController (e2e) conflict test
    ✓ POST /msisdn returns 400 when schema does not match (73 ms)

PASS test/happy-test.e2e-spec.ts
  WebserverController (e2e) happy test
    ✓ Unallocated msisdns can be fetched (23 ms)
    ✓ Users can be created (51 ms)
    ✓ Users can be fetched by Organisation (41 ms)
    ✓ Users can be deleted (46 ms)

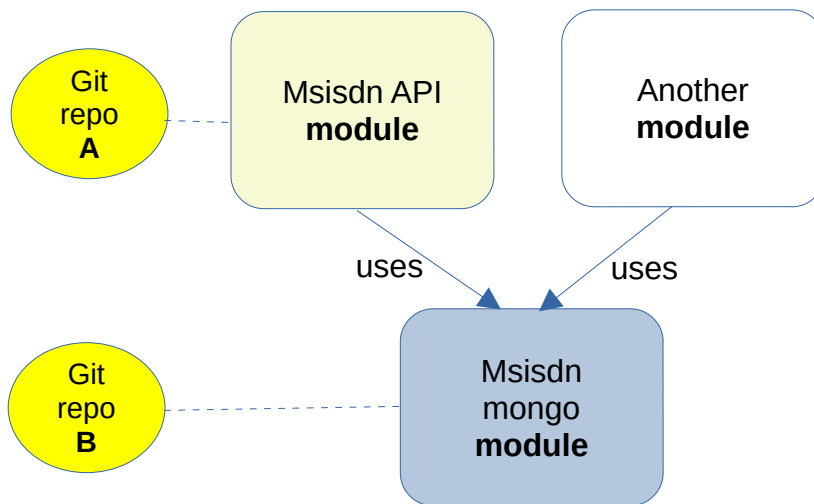
PASS test/conflict-test.e2e-spec.ts
  WebserverController (e2e) conflict test
    ✓ Allocate different msisdn to existing user results in error (36 ms)
    ✓ Allocate same msisdn to other user results in error (24 ms)
    ✓ Allocate non existing msisdn results in error (9 ms)

Test Suites: 3 passed, 3 total
Tests:      8 passed, 8 total
Snapshots:  0 total
Time:       5.14 s, estimated 6 s
Ran all test suites.
```

# Design decisions

## Modularity

Web server and the MongoDB repository are two different modules. Ideally we separate them even more and they both go in their own git repository so that we allow for better re-use and we allow different developers to work these projects simultaneously in a safer manner.



## (REST) API paths

Lets take for example the query to get users belonging to an organisation

This can be done by specifying the department name in the query

GET /msisdn/organisation/**retrotech**

The advantage is that the above is very readable. There is no need to have any information inside the body.

The disadvantage of supplying the name inside the query is that the name may contain special characters and spaces.

GET /msisdn/organisation/**Federación Andaluza de Ciclismo**

has to be encoded and becomes:

GET /msisdn/organisation/**Federaci%C3%B3n%20Andaluza%20de%20Ciclismo**

My experience with this has not been good and that's why I chose to place the organisation name inside the body:

```
GET /msisdn/organisation/  
{ "organisation" : "Federación Andaluza de Ciclismo" }
```

## TESTS unit tests, e2e tests, etc...

This project only contains e2e tests. Ideally there would be unit tests as well. However given that this project has very little to no logic to test with unit testing I chose to only include end to end tests.

## Usage of NestJS

I have good experience with NEST. It provides good wrappers and decorators for frameworks such as Express. It makes it easy to write **modular** code. **Dependency injection** is very nicely implemented.

## Improvements or missing parts

- **Transaction handling** is needed when writing to MongoDB. This is extremely important and in a real life project needs to be implemented.
- More **logging** could be added.
- More **comments** inside the code could be added.
- **Event** writing to for example Elastic Search. It's important to be able to generate statistics over time. To be able to see which person owned which msisdn at what time, etc...

Rogier Taal  
Álora, June 25 2024