Mini-Project

Component Based Development

Compte-rendu

“In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. For example, on the read side, the application may perform many different queries, returning data transfer objects (DTOs) with different shapes. Object mapping can become complicated. On the write side, the model may implement complex validation and business logic. As a result, you can end up with an overly complex model that does too much.”

Presented By:
Oussema Zouaghi
Cyrine Zaouali

1/ The problem:

- Create a solution that implements CQRS using the following:

    - A Read service that makes calls to ElasticSearch
    - A Write service that makes calls to MongoDB
    - Assert coherence between the contents of MongoDB & ElasticSearch

2/ Choice of Technologies:

A.ElasticSearch

- Supports JSON based APIs. Making it easily integrated in today's REST dominated architectures.
- has SDKs and Clients for almost every language: Python, JavaScript, .NET, JAVA...
- supports filters, sorts pagination and aggregation in the same query.
- handles unstructured data automatically, enabling its users to index JSON documents without pre-defining its schema. Which is a powerful advantage over typical SQL databases.
- Is highly scalable within high-on-demand networks. It natively supports clustering, replication of data...
- Provides a world-renown powerful indexing engine. Not only that, but also automatic text spelling correction, and type-ahead functionalities. Which is the bread and butter of every search engine nowadays. Hence the reason we went with this technology.

B.MongoDB

- High performance, since it stores most the data in RAM before persisting it on Disk. Allowing faster queries execution time.
- Simple query syntax
- Flexibility in data schemas. Businesses keep evolving and so do the data they maintain. It is important to have a flexible database model that could adapt to these changes.
- Native adoption of Sharding. Mongo divides large data sets and distribute them among multiple servers.
- Sharding allows MongoDB to use horizontal scalability efficiently

C.NestJS

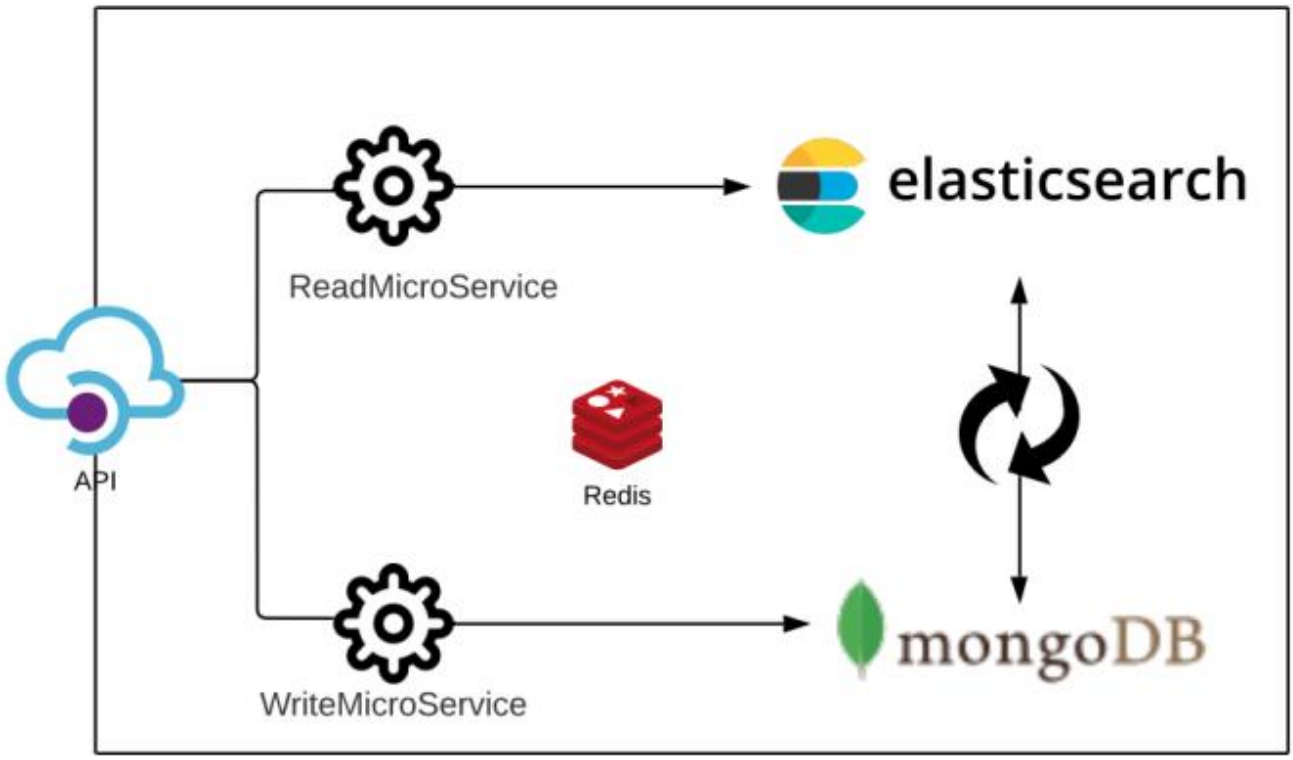
- Created for Monoliths and Micro-services (an entire section in the documentation regarding Microservice types of NestJS applications as well as techniques and recipes)
- Easy to use, learn and master
- Powerful Command Line Interface (CLI) to boost productivity and ease development
- Detailed and well-maintained documentation
- Support for dozens of nest-specific modules that help you easily integrate with common technologies and concepts like TypeORM, Mongoose, GraphQL, Logging, Validation.

N.B: Because the task required Spring Boot and we used NestJS. We'll try to dive in details about our solutions to further explain the process better!

D. Redis

- Very fast in-memory Database.
- Redis can be easily configured to act automatically as a cache and use one of the popular eviction algorithms such as the LRU algorithm which makes it act like memcached database.
- publish/subscribe messaging paradigm and it is suitable for implementing chatting related use cases.
- Redis provides a master-slave distributed system called Redis Sentinel to guarantee high availability.

3/ Solution's architecture



General points:

In real life scenarios, Micro services clusters get so massive that the traffic between the components becomes crippling. This resulted in the need of ditching HTTP communications in favor of Messages/events. We will explain in details down below how we disabled our microservices listeners to HTTP requests, and made them consumes messages from our REDIS Queues.

A. API Gateway

This is a simple web application that the end-user will be interacting with to invoke calls.

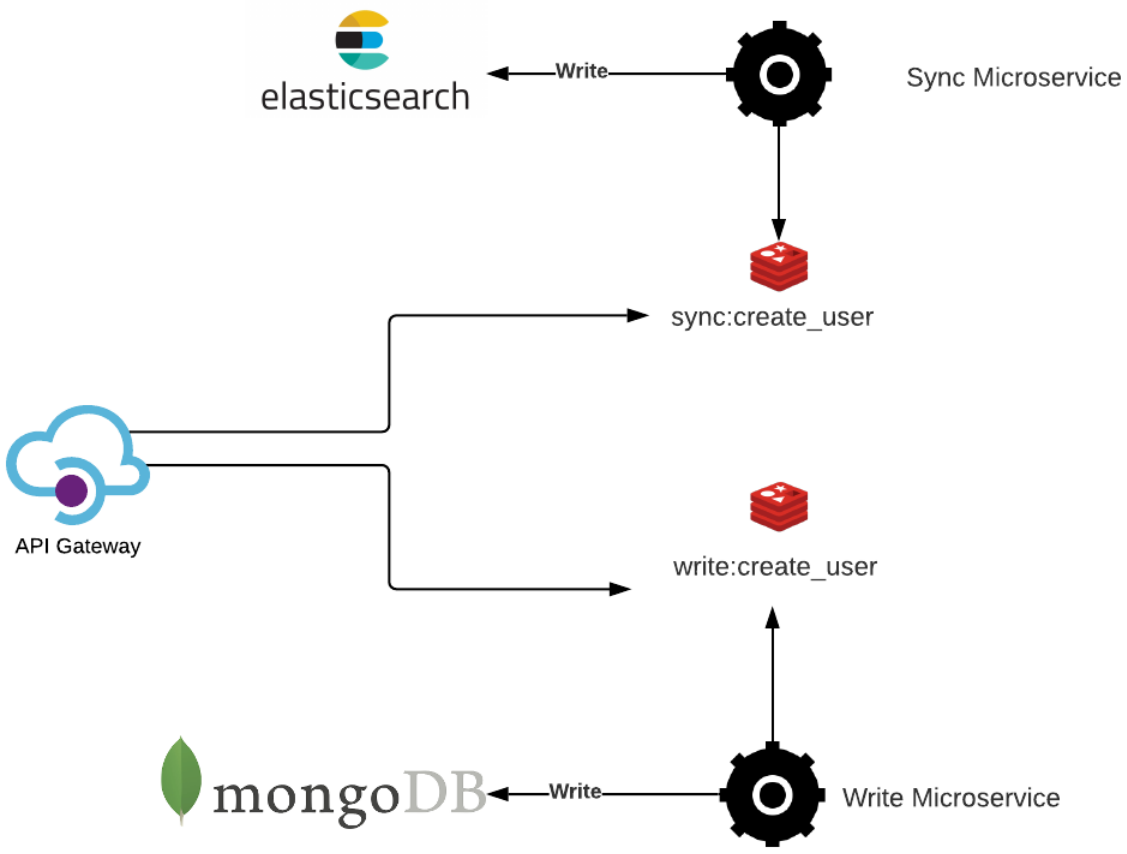
However, this app is making use of Client Proxies.
A ClientProxy is an abstraction of the remote Microservice, that provides the API Gateway with a set of method to populate the Queue with messages/events.
This is how it's defined:

```
provide: 'WRITE_USER_SERVICE',
useFactory: () =>
  ClientProxyFactory.create({
    transport: Transport.REDIS,
    options: {
      url: 'redis://redis:6379',
    },
  }),
},
```

This is done for the Read Microservice, Write Microservice and the Sync Microservice. And the gateway now can assign jobs based on the incoming HTTP Request.
This results in Isolating our inner Network from the outside malicious user, and having finer control on the behavior of requests, as we can filter, cache and rate limit them.

How it works?

Let's assume we are looking at the create_user request now.
When a user invokes the POST request to the Gateway. The gateway does the following:



This way, we achieve **asynchronous** coherence between the two databases.

Messaging vs Events?

We need to understand the difference between them to choose what to use correctly.

When a user invokes a Get request, he's expecting a response. So the TCP connection has to be kept open until the cluster is done processing and returns the result to the API gateway so it returns it to the user. This is what we call a response overhead. When such case is expected, This is the correct place to use Messaging.

However when we are not expecting a response overhead from within the cluster. It's advised to use Events.

So, this raises the question, How did we use them in our code?

This is where the power of ClientProxy from NestJS really manifests. As it allows you to use Both Messaging based communication and Event based communication with the same underlying architecture.
Here is how:

Message Based communication:

```
@Get('userByEmail')
async getUserByEmail(@Body() payload: { email: string }, @Res() response) {
  this.readUserService
    .send('read_user_by_email', payload.email)
    .pipe(
      take(1),
      map((e) => {
        Logger.log(
          `Successfully sent a get_user_by_email message`,
          'Get User By Email',
        );
        return e;
      }),
    )
    .subscribe((e) => response.status(HttpStatus.OK).send(e));
}
```

Event Based Communication:

```
@Post('create')
async createUser(@Body() createUserPayload: CreateUserDto, @Res() response) {
  let err: Error;
  this.syncService.emit('create_user', createUserPayload).subscribe({
    complete: () => {
      Logger.log(
        `Successfully pushed a sync event for user ${createUserPayload.email}`,
        'User Sync',
      );
    },
  });
  this.writeUserService.emit('create_user', createUserPayload).subscribe({
    error: (e) => {
      err = e;
    },
    complete: () => {
      Logger.log(
        `Successfully pushed a create_user event for user: ${createUserPayload.
        username} `,
        'User Creation',
      );
    },
  });
```

Did you spot the difference?

If not, the difference lies in the method we call from the ClientProxy. When using CP.emit(), you're using event-based communication. Whereas CP.send() is message based, so the connection is maintained open until there is a response!

B. Read Microservice

This microservice responds to **messages** sent by the API gateway. It picks it up from the Message Broker with its appropriate payload, performs the query to ElasticSearch and returns the result when its done.

C. Write Microservice

This microservice responds to **events** sent by the API gateway. It picks it up from the Message Broker with the appropriate payload, performs the command to MongoDB.

D. Synchronization Microservice

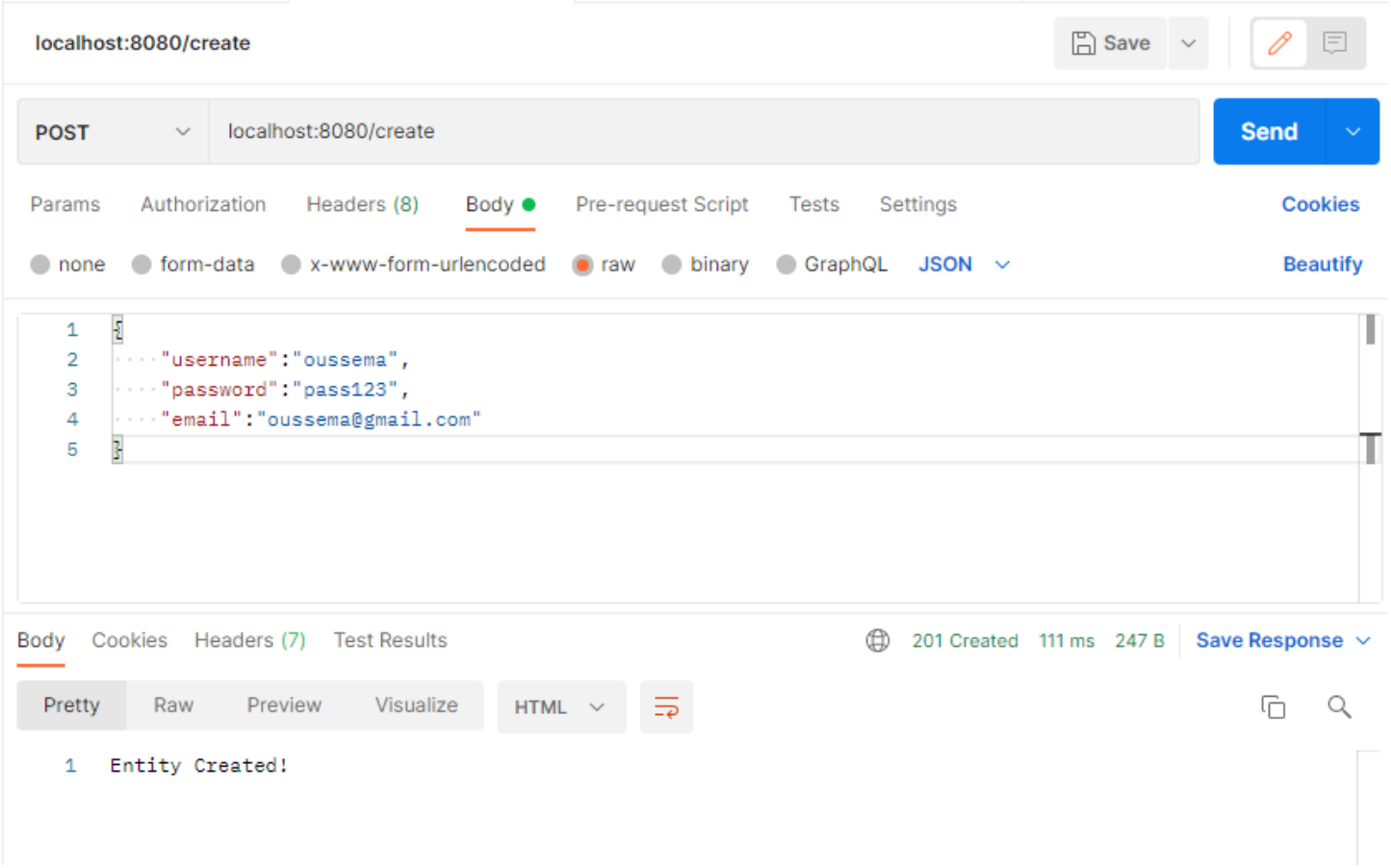
This microservice responds to events sent by the API gateway. It picks it up from the Message broker with the appropriate payload, then performs the command to ElasticSearch.

4/ Solution’s deployment

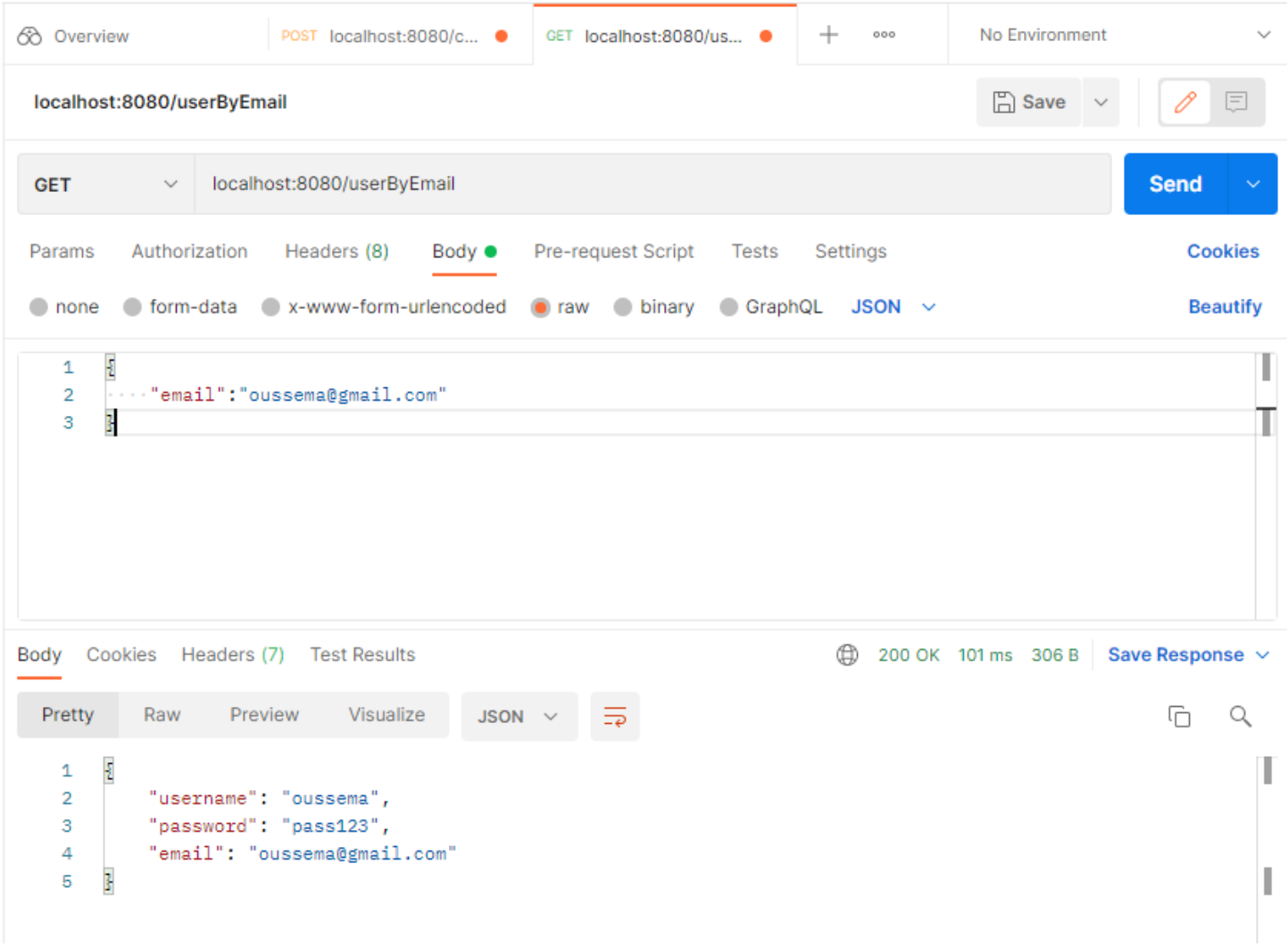
This solution is deployed on a docker-compose stack.
You can view the file on this Github link.

5/ Solution’s tests:

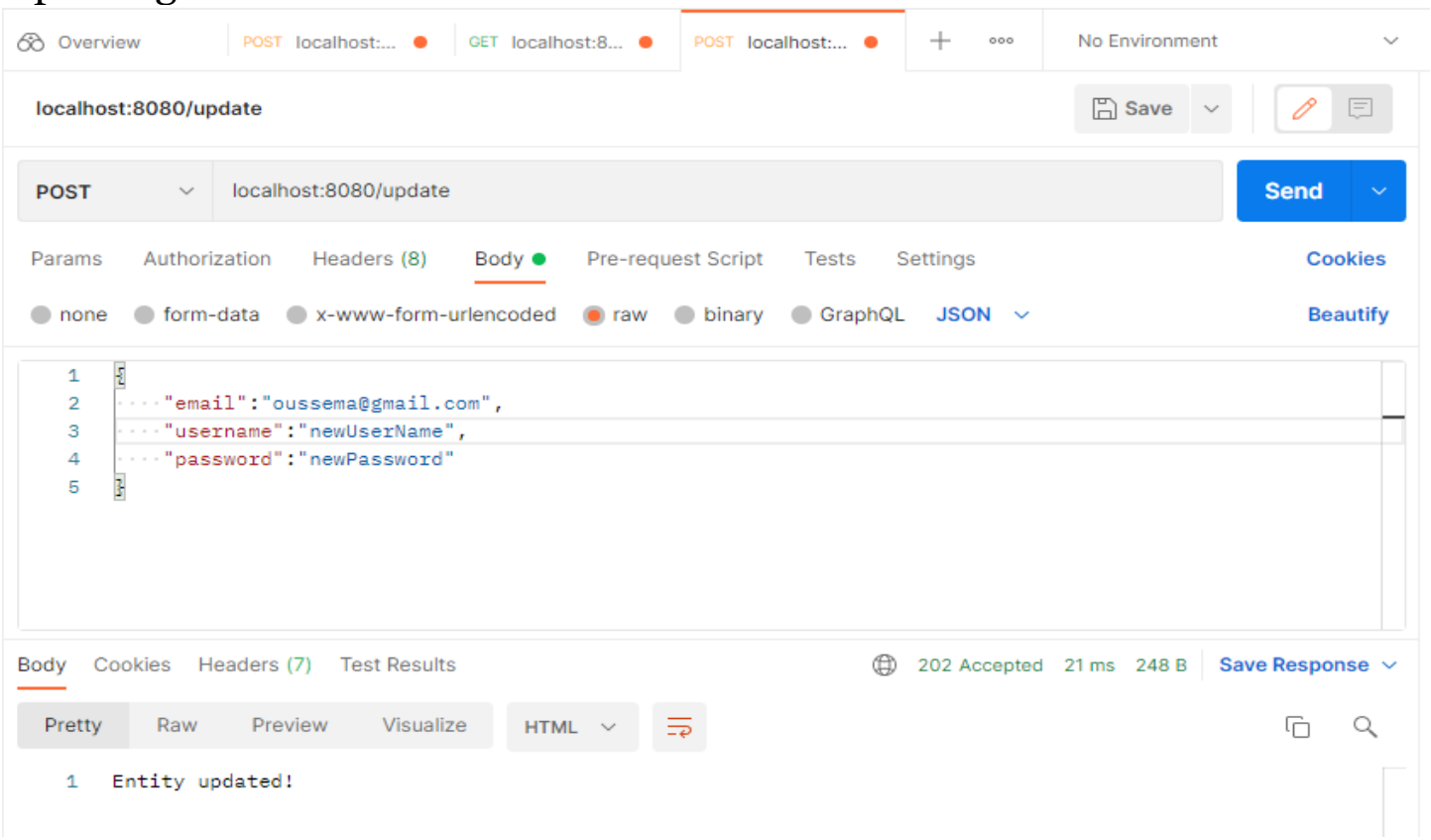
Creating a User:



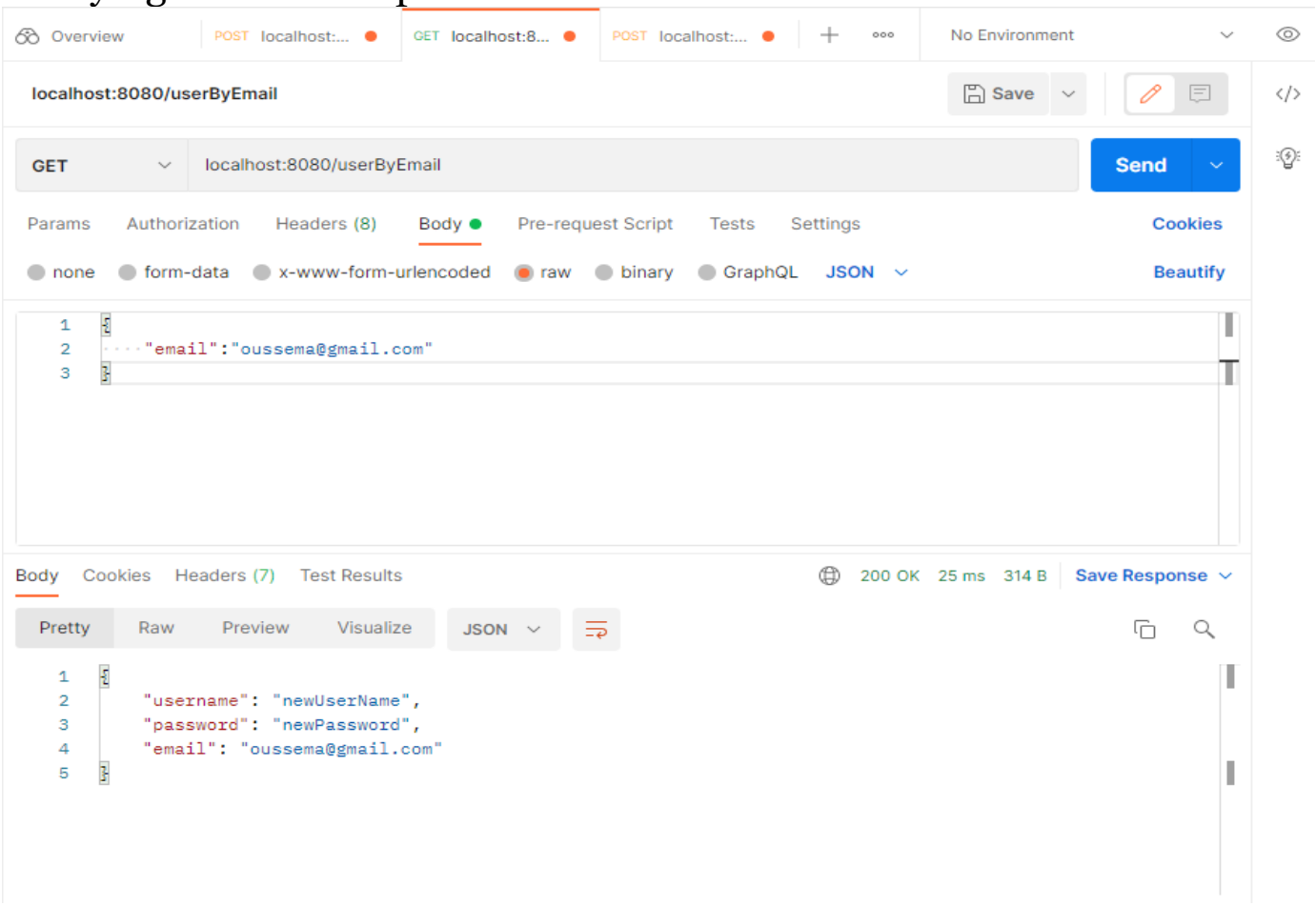
Fetching the user we created:



Updating the user we created:



Verifying that it was updated in ElasticSearch



The project's source code:

[Github link]