MAC0475 - Laboratório de Sistemas Computacionais Complexos

Exercício-Programa 1

Contexto

Nas últimas aulas, tivemos o primeiro bloco de tecnologia do curso, focado em ferramentas para desenvolvimento back-end: Docker, Docker-Compose, Node.js, Express.js, MongoDB e NATS Streaming. Agora é hora de aplicá-las em conjunto!

Neste exercício-programa, você deve utilizar as tecnologias acima para implementar um único serviço. A especificação, apresentada a seguir, deverá ser seguida à risca. **Sua implementação será avaliada com o uso de testes automatizados**.

Especificação

Seu serviço deverá implementar duas ações do CRUD de uma entidade Usuário: a **Criação** e a **Deleção**.

Para este exercício, define-se a entidade `Usuário` composto pelos seguintes atributos e seus respectivos domínios:

- **`UUID`**: identificador universal gerado automaticamente pelo serviço.
- **`Name`**: texto obrigatório e não-vazio.
- **`Email`**: texto obrigatório, aderente ao formato de um email convencional e único dentre todos os registros conhecidos.
- **`Password`**: texto alfanumérico, de comprimento entre 8 e 32 (inclusive), permitindo letras maiúsculas e minúsculas, sem qualquer obrigatoriedade entre as escolhas.

Ambas ações deverão ser implementadas como rotas na API Web do serviço, seguindo o padrão REST. Para tanto, você deve utilizar o *framework* Express.js, conforme visto em aula.

Criação

A ação de Criação (C de CRUD) deverá ser implementada para atender requisições:

`POST /users`

Essa ação deve esperar o seguinte JSON no corpo da requisição:

```
{
      "name": Name¹,
      "email": Email¹,
      "password": Password¹,
      "passwordConfirmation": Password¹,
}
                                  ¹ Domínio especificado anteriormente
```

Primeiramente, você deve verificar se todos os campos estão presentes. Caso não estejam, este é um **erro de contrato**, logo você deverá devolver uma resposta com código 400 (BAD REQUEST) e a seguinte mensagem de erro no formato JSON como corpo:

```
{
      "error": "Request body had missing field {field_name}",
}
```

Em seguida, você deve verificar se os campos apresentam os tipos conforme o domínio especificado. Caso não possuam, este é um **erro de formatação**, logo você deverá devolver uma resposta com código 400 (BAD REQUEST) e a seguinte mensagem de erro no formato JSON como corpo:

```
{
      "error": "Request body had malformed field {field_name}",
}
```

Finalmente, você deverá verificar se os campos **`password`** e **`passwordConfirmation`** possuem o mesmo conteúdo. Caso não sejam, este é um **erro semântico**, logo você deverá devolver uma resposta com código 422 (UNPROCESSABLE ENTITY) e a seguinte mensagem de erro no formato JSON como corpo:

```
{
      "error": "Password confirmation did not match",
}
```

Se todas as validações forem bem sucedidas, você deverá devolver uma resposta com código 201 (CREATED) e o seguinte conteúdo no formato JSON como corpo:

```
{
      "id": UUID[1],
      "name": Name[1],
      "email": Email[1]
}
```

[1] Domínio especificado anteriormente

Deleção

A ação de Deleção (D de CRUD) deverá ser implementada para atender requisições:

```
DELETE /users/:uuid
```

Essa ação não deve esperar nenhum conteúdo no corpo da requisição.

Em contrapartida, essa ação deve devolver uma resposta com código 200 (OK) e o seguinte conteúdo no formato JSON como corpo:

```
{
    "id": UUID[1],
}
```

[1] Domínio especificado anteriormente

Ao contrário da ação de Criação, a ação de Deleção não deve ter livre acesso, pois ela deve ser permitida apenas ao usuário identificado pelo UUID recebido como parâmetro. Para impor essa regra de negócio, o serviço deve implementar uma estratégia de controle de acesso, que inclui tarefas como **autenticação** e **autorização**, também abreviadas como **auth**.

Neste EP, implementaremos controle de acesso utilizando um token de acesso no formato JSON Web Token (JWT). Você pode ler mais sobre JWT neste link aqui.

Para limitarmos o escopo da tarefa, você não precisará se preocupar com a funcionalidade de **autenticação**, que demandaria a implementação do *login*, responsável pela geração do token de acesso. No entanto, você deverá implementar a funcionalidade de **autorização**, que fará parte do processamento de toda solicitação de deleção.

A requisição deverá conter um **cabeçalho de autenticação** com o seguinte formato:

```
Authentication: Bearer <token>
```

Primeiramente, você deverá verificar a presença do cabeçalho *Authentication* na requisição. Caso ele não exista, você deverá devolver uma resposta com código 401 (UNAUTHORIZED) e a seguinte mensagem de erro no formato JSON como corpo:

```
{
    "error": "Access Token not found",
}
```

Em seguida, você deverá verificar se UUID presente dentro do token é igual ao UUID informado como parâmetro na rota. Caso não seja, você deverá devolver o código 403 (FORBIDDEN) e a seguinte mensagem de erro no formato JSON como corpo:

```
{
      "error": "Access Token did not match User ID",
}
```

Para lidar com tokens JWT, você deverá utilizar a biblioteca jsonwebtoken, pré-instalada no *template* fornecido para este EP. Com ela, você poderá extrair o conteúdo (*payload*) criptografado dentro do token de acesso para, finalmente, acessar o UUID do usuário identificado pelo token.

Eventos

Em sistemas complexos, serviços fazem parte de um ecossistema maior. É importante, nesse cenário, manter a consistência de dados com outros serviços. Além de implementar a API REST especificada acima, você deve publicar Eventos de Domínio que vão notificar outros serviços interessados de mudanças de estado ocorridas no seu serviço, habilitando assim padrões como Event Sourcing, CQRS e Saga.

Para toda operação de criação de usuário, você deve emitir um evento ***UserCreated*** no **tópico *users*** com o seguinte conteúdo com formato JSON:

```
{
      "eventType": "UserCreated",
      "entityId": UUID¹,
      "entityAggregate": {
            "name": Name¹,
            "email": Email¹,
            "password": Password¹,
      }
}
                                  ¹ Domínio especificado anteriormente
```

Para toda operação de deleção de usuário, você deve emitir um evento ***UserDeleted*** no **tópico *users*** com o seguinte conteúdo com formato JSON:

```
{
      "eventType": "UserDeleted",
      "entityId": UUID¹,
      "entityAggregate": {}
}
                                  ¹ Domínio especificado anteriormente
```

Para lidar com a publicação e leitura de eventos, você deverá utilizar a biblioteca node-nats-streaming, pré-instalada no *template* fornecido para este EP. Com ela, você poderá publicar eventos e se inscrever em tópicos (ou canais) de mensagens do NATS Streaming.

Template

Como mencionado anteriormente, forneceremos um *template* para ser usado como base para o EP, presente neste repositório. Nele, você encontrará um serviço pré-configurado com a seguinte estrutura de diretórios: (obs.: alguns arquivos foram omitidos para fins de clareza)

- `docker-compose.yml`
- `Dockerfile`
- `package.json`
- `package-lock.json`
- `src/`
  - `api.js`
  - `mongo.js`
  - `nats.js`
  - `start.js`
- `tests/`
  - `api.test.js`

Você deve focar seu desenvolvimento nos arquivos `src/api.js` e `src/nats.js`. No entanto, é importante consultar os outros arquivos também, pois eles auxiliarão na compreensão do EP.

Como o serviço está configurado com Docker e Docker -Compose, eles serão a sua interface de comando. Para rodar, primeiro construa uma imagem:

```
[sudo] docker-compose build
```

Tendo a imagem, é possível criar e executar os *containers*:

```
[sudo] docker-compose up [-d]
```

Com isso, seu serviço estará acessível no endereço http://localhost:3000

A avaliação deste EP será feita com base nos **testes automatizados** existentes, que podem ser encontrados em `tests/api.test.js`. **Não altere os testes existentes**. Para verificar se seu código está condizente com os testes, execute:

```
[sudo] docker-compose run --rm api npm run test:dev
```

O Jest imprimirá uma mensagem avisando que ele está observando seu repositório. Ao salvar um arquivo observado, a bateria de testes será reexecutada automaticamente. **Para executar os testes, não é necessário (nem recomendável) que o serviço esteja rodando.**

Entrega

A entrega deverá ser feita via GitLab. Você deve criar um *fork* do repositório para sua conta pessoal, sendo que sua entrega consistirá em enviar a URL do seu repositório. **A avaliação será baseada nos testes automatizados**, conforme explicado na seção anterior. A versão considerada será o último *commit* feito antes do horário limite da entrega.

A data limite para entrega é 17/05 (segunda-feira), até às 23h55, com submissão via e-disciplinas do nosso curso.

Avaliação

O repositório conta com 6 testes escritos, sendo **3 deles desenvolvidos como exemplo**, que já estão passando, e **3 deles descrevendo o que você deve implementar**. Cada um desses testes que não estão passando vale ⅓ (um terço) da nota, ou seja, ~3.33. A avaliação dos mesmos será binária, isto é, para cada teste que passar, o aluno terá 3.33 a mais em sua nota; caso um teste não passe, não haverá notas intermediárias.

EXTRA -- Não existe um teste automatizado que cubra os casos de falha na criação de usuário. Fica a critério do aluno buscar implementá-lo, de modo complementar, valendo até 1 ponto extra na nota deste EP. Neste caso, a avaliação não será feita de modo binário, podendo ser avaliado com pontuações intermediárias.