

dasa

Dasa Coding Challenge

Documentação

Histórico de Revisões

Versão	Data	Responsável	Comentário
1.0	25/03/2020	Danilo Paggi	Criação do documento.
2.0	23/04/2020	Danilo Paggi	Reestruturação para Docker.

Sumário

1. Tecnologias utilizadas.....	5
2. Pré-requisitos.....	5
2.1. Docker	5
2.1.1. Node	5
2.1.2. Mongo	5
2.2. API Client.....	5
3. Configurações	5
3.1. Configurando a aplicação	5
3.2. Configurando o banco de dados	6
3.3. Configurando os tipos de exame disponíveis na aplicação.....	6
3.4. Configurando o docker	6
3.4.1. Caso mudança na porta da aplicação	7
3.4.2. Caso mudança na porta do banco de dados.....	7
3.5. Configurando o Insomnia (API Client).....	7
4. Rodando a aplicação.....	8
4.1. Montar o container da aplicação	9
4.2. Subir o container da aplicação	9
4.3. Parar o container da aplicação.....	9
5. Organização das pastas do projeto.....	10
5.1. <Pasta raiz>	10
5.2. docs	10
5.3. node_modules	10
5.4. src	10
5.5. app	10
5.6. controllers	10
5.7. enum	10
5.8. exceptions.....	10
5.9. functions	10
5.10. middlewares	11
5.11. models.....	11
5.12. config.....	11
6. Banco de dados da aplicação	11
6.1. Laboratory.....	11

6.2.	Exam.....	12
6.3.	ExamType.....	12
7.	Endpoints	12
7.1.	Listar laboratórios.....	12
7.2.	Retornar todos os dados de um laboratório específico	12
7.3.	Criar um ou mais novos laboratórios.....	13
7.4.	Atualizar um ou mais laboratório existente	13
7.5.	Apagar um ou mais laboratórios existente	13
7.6.	Listar exames	13
7.7.	Retornar todos os dados de um exame específico	14
7.8.	Listar todos os laboratórios vinculados a um exame	14
7.9.	Criando um ou mais novos exames	14
7.10.	Vincular um ou mais laboratórios a um exame.....	15
7.11.	Atualizar um ou mais exames existente	15
7.12.	Apagar um ou mais exames existente	15
7.13.	Desvincular um ou mais laboratórios de um exame.....	16
7.14.	Listar tipos de exame.....	16
7.15.	Retornar todos os dados de um tipo de exame específico	16
8.	Melhorias	16

1. Tecnologias utilizadas

A stack utilizada no projeto seguiu conforme abaixo:

- NodeJs (v.10) nos módulos referentes a API.
- MongoDB (v. 4) no banco de dados.
- Docker (v. 19.03.8)
- Docker compose (v. 3)

2. Pré-requisitos

É necessário que o ambiente aonde será rodado o projeto tenha instalado o serviço do Docker com as imagens Node e Mongo. Abaixo seguem os sites oficiais e tutorial de instalação de cada componente.

2.1. Docker

A instalação deste componente já instala o *Docker compose*.

Site oficial: <https://www.docker.com/>

Instalação: <https://docs.docker.com/get-docker/>

2.1.1. Node

Site da imagem oficial: https://hub.docker.com/_/node?tab=description

2.1.2. Mongo

Site da imagem oficial: https://hub.docker.com/_/mongo

2.2. API Client

O projeto consta com interface gráfica web criada pelo swagger, mas se pode optar por utilizar um programa do tipo API Client para acessar suas funcionalidades. Neste documento detalharemos a configuração para o programa *Insomnia*, porém é possível utilizar qualquer outro programa de sua preferência:

Site oficial: <https://insomnia.rest/>

Instalação: <https://support.insomnia.rest/article/23-installation>

3. Configurações

Os arquivos responsáveis pelas configurações do projeto, citados neste capítulo, ficam situados no seguinte endereço <Pasta raiz do projeto>/src/config e já vem previamente configurações com valores padrões.

3.1. Configurando a aplicação

O responsável pelas configurações da aplicação é o arquivo **app.js**, nele temos as seguintes opções:

- **Environment;** contêm o ambiente em que o projeto irá executar, aceitando os valor conforme abaixo:

- development
- test
- production

É **importante** se atentar com o valor da variável, pois trata-se de um valor *case sensitive*.

- **AppPort**; define a porta em que a API será disponibilizada, no caso de não preenchimento será usado o padrão 3000.

É **importante** se atentar que no caso de alteração para uma porta diferente de 3000, deve-se seguir os procedimentos do [capítulo 3.4](#) para reconfigurar as portas dos containers do docker.

3.2. Configurando o banco de dados

O responsável pelas configurações da comunicação da API com o banco de dados mongoDB está no arquivo **database.js**, nele teremos as seguintes opções editáveis:

- **host**; o endereço da base de dados
- **user**; usuário utilizado no login para o banco de dados.
- **psw**; senha utilizada no login para o banco de dados.
- **host**; o endereço da base de dados.
- **port**; a porta de comunicação a ser utilizada.

É **importante** se atentar que no caso de alteração para uma porta diferente de 27017, deve-se seguir os procedimentos do [capítulo 3.4](#) para reconfigurar as portas dos containers do docker.

- **db**; base de dados no banco que será utilizada para armazenar as collections do projeto.

3.3. Configurando os tipos de exame disponíveis na aplicação

Na linha 2 do arquivo *ExamType.js* localizado na pasta <Pasta raiz do projeto>/src/app/models teremos uma array que armazena quais serão os tipos de exame disponíveis no sistema conforme a Figura 1:

```
2  const defaultExamTypes = [ "Análise clínica", "Imagem" ];
```

Figura 1 -Tipos de exame

É **importante** se atentar que no caso de alteração do padrão, estas devem ser realizadas previamente a geração do container da aplicação, pois os mesmos são adicionados automaticamente no banco.

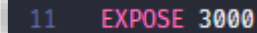
3.4. Configurando o docker

Os arquivos de configuração do docker já estão pré-configurados e só há necessidade de alteração no caso de mudança das portas padrões apresentadas no [capítulo 3.1](#) e [capítulo 3.2](#).

3.4.1. Caso mudança na porta da aplicação

Caso tenha ocorrido alteração no valor padrão (3000) da chave **AppPort** pertencente ao arquivo de configuração **app.js** descrito no capítulo 3.1 se faz necessário as seguintes alterações:

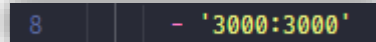
- Alterar linha 11 do arquivo *Dockerfile* localizado na raiz da pasta do projeto (Figura 2) , substituindo o valor “3000” para o novo valor a ser utilizado como porta:



```
11 EXPOSE 3000
```

Figura 2- Porta no Dockerfile a ser alterada

- Alterar linha 8 do arquivo *docker-compose.yml* localizado na raiz da pasta do projeto (Figura 3) , substituindo ambos os valores “3000” para o novo valor a ser utilizado como porta:



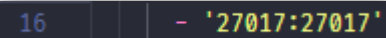
```
8 - '3000:3000'
```

Figura 3 - Portas do docker-compose.yml a serem alteradas para a api

3.4.2. Caso mudança na porta do banco de dados

Caso tenha ocorrido alteração no valor padrão (27017) da chave **port** pertencente ao arquivo de configuração **database.js** descrito no capítulo 3.2 se faz necessário a seguinte alteração:

- Alterar linha 16 do arquivo *docker-compose.yml* localizado na raiz da pasta do projeto (Figura 4) , substituindo ambos os valores “27017” para o novo valor a ser utilizado como porta:



```
16 - '27017:27017'
```

Figura 4 - Portas do docker-compose.yml a serem alteradas para o banco de dados

3.5. Configurando o Insomnia (API Client)

Após seguir os procedimentos de instalação do Insomnia, abra o aplicativo e siga o procedimento abaixo:

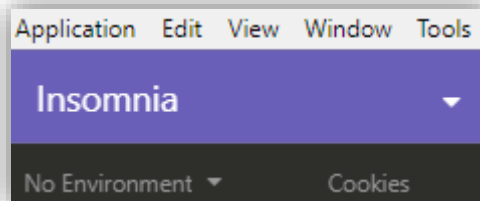


Figura 5- Clicar no **botão triângulo** para abertura do menu

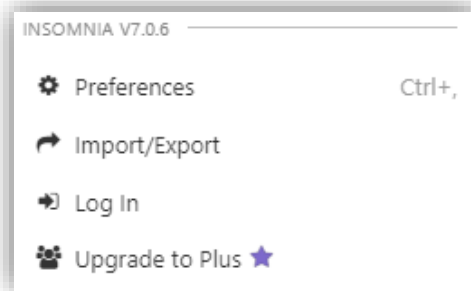


Figura 6- Clicar em **Import/Export**

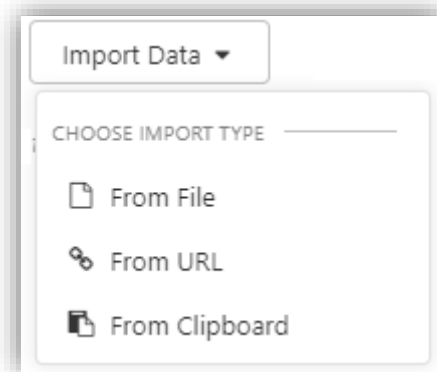


Figura 7- Acionar o botão **Import Data** e na sequência clicar em **FromFile**

Haverá a abertura de uma tela de seleção de arquivo com nome Import Insomnia Data, através dela, vá até <Pasta raiz do projeto>/docs, selecione o arquivo **Insomnia.json** e clique no botão **Import**. Será apresentado conforme Figura 8:

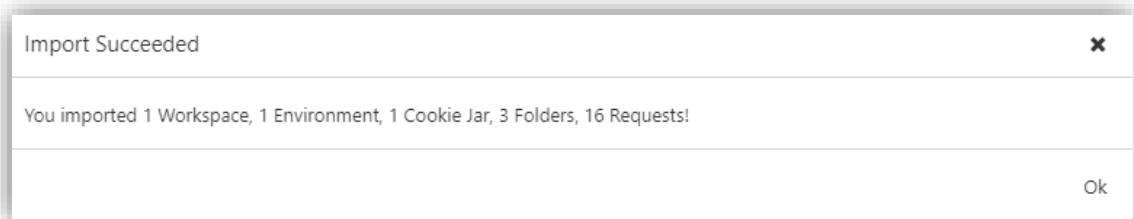


Figura 8- Após clicar em OK, a configuração já estará finalizada.

4. Rodando a aplicação

Para subirmos os containers e disponibilizar a API para que comece a responder as chamadas é necessário efetuarmos os passos descritos neste capítulo.

Importante: Os comandos descritos neste capítulo, devem ser executados através do console de seu sistema operacional, sempre com o caminho do diretório apontando para a pasta raiz de seu projeto.

4.1. Montar o container da aplicação

Inicialmente devemos gerar o container do projeto executando o seguinte comando:

```
docker-compose build
```

Este processo gerará a imagem que será utilizada pelo Docker para usarmos a aplicação e, quando finalizado, apresentará uma resposta no console semelhante a Figura 9:

```
Successfully built 71edcc0ddebb  
Successfully tagged back_app:latest
```

Figura 9 - Mensagem de montagem de container com sucesso

4.2. Subir o container da aplicação

Para que a API comece a responder as chamadas é necessário utilizar o comando:

```
docker-compose up
```

Este processo disponibilizará pelo Docker a imagem gerada no [capítulo 4.1](#) permitindo que utilizemos a API. Quando finalizado, entre diversas linhas de processamento do Docker, será apresentado uma resposta no console conforme a Figura 10:

```
prj-back-dasa | API documentation: http://localhost:3000/doc
```

Figura 10 - Link para a documentação web da API

Neste momento a imagem da API já está disponível para o acesso aos endpoint, permitindo a interação pelo próprio Swagger disponibilizado no link da Figura 10 ou através de um software de API Client conforme descrito no [capítulo 2.2](#).

4.3. Parar o container da aplicação

Para encerrar a execução da API você pode seguir por dois fluxos:

- 1) Com foco na mesma janela do console aonde foi executado o comando do [capítulo 4.2](#), clicar nas teclas Ctrl+C
- 2) Em uma nova janela do console, mas ainda apontando pra raiz do projeto, executar o comando:

```
docker-compose down
```

5. Organização das pastas do projeto

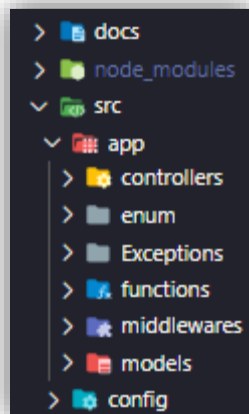


Figura 11 - Estrutura de pastas do projeto

5.1. <Pasta raiz>

Pasta raiz do módulo, contendo todos os arquivos de configuração e pacotes.

5.2. docs

Contém os documentos técnicos do projeto, além do arquivo para configuração do software de API-client.

5.3. node_modules

Pasta com todas as bibliotecas do projeto.

5.4. src

Contendo todos os arquivos da API e, em sua raiz, os arquivos responsáveis pelo carregamento da API.

5.5. app

Contém os arquivos de lógica da aplicação.

5.6. controllers

Guarda os controladores da aplicação, responsáveis por orquestrar o fluxo entre as bibliotecas, funções e base de dados.

5.7. enum

Guarda os enums da API, que seriam arquivos para tipos complexos de variável.

5.8. exceptions

Contém as bibliotecas de erros utilizados pela aplicação.

5.9. functions

Pasta com todos os arquivos de regras, validações e funcionalidades orquestradas pelos controladores.

5.10. middlewares

Contém os middlewares responsáveis por orquestrar a validação dos dados que serão repassados para seus respectivos controllers.

5.11. models

Contém as definições e relacionamentos das entidades representadas por cada collection do banco de dados.

5.12. config

Contém as configurações internas da API.

6. Banco de dados da aplicação

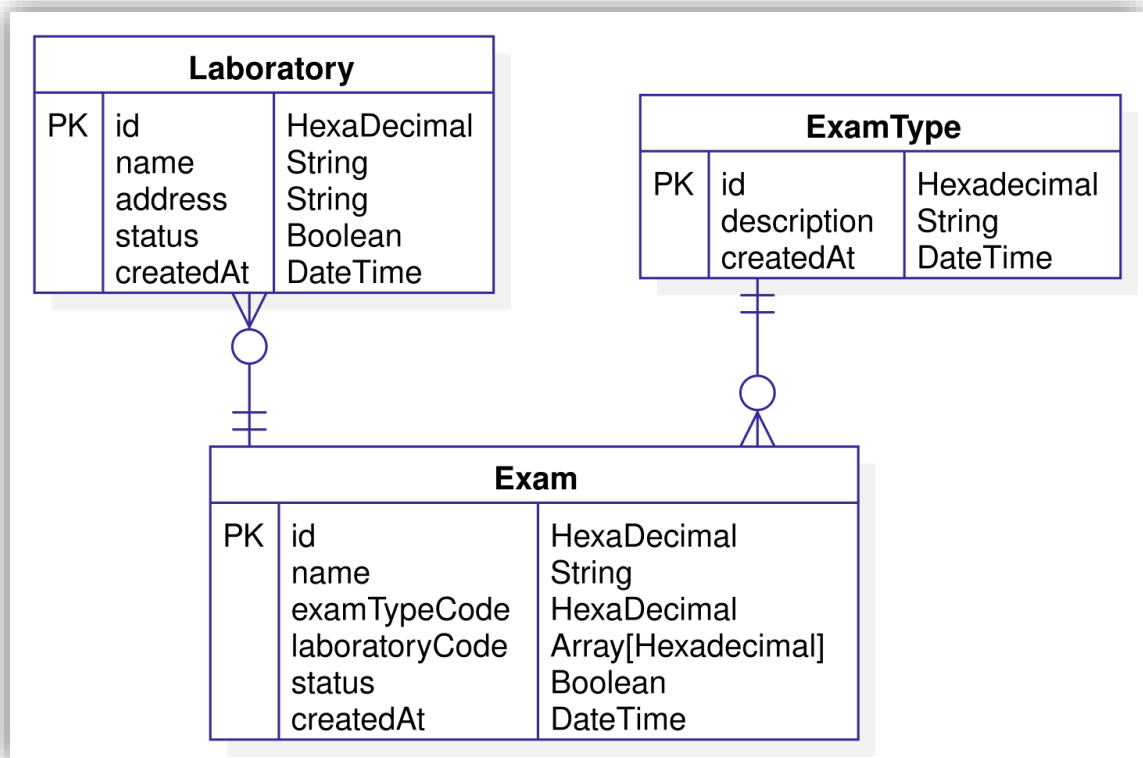


Figura 12 - Modelo de entidade relacional

6.1. Laboratory

Tabela responsável pelo armazenamento dos dados dos laboratórios da aplicação:

id (Chave primária, hexadecimal) Código de identificação do laboratório.

name (String) Nome do laboratório.

address (String) Local onde fica situado o laboratório.

status (Boolean) Flag que define se o laboratório está ativo ou inativo.

createdAt (DateTime) Data e hora da criação do laboratório no banco.

6.2. Exam

Tabela responsável pelo armazenamento dos exames da aplicação:

id (Chave primária, hexadecimal) Código de identificação do exame.

name (String) Nome do exame.

examTypeCode (hexadecimal) Código referente a tabela *ExamType*, para definir o tipo de exame correspondente ao registro.

laboratoryCode (Array[Hexadecimal]) Código referente a tabela *Laboratory*, para definir a quais laboratórios o exame está vinculado.

status (Boolean) Flag que define se o exame está ativo ou inativo.

createdAt (DateTime) Data e hora da criação do exame no banco.

6.3. ExamType

Tabela responsável pelo armazenamento dos possíveis tipos de um exame:

id (Chave primária, hexadecimal) Código de identificação do tipo de exame.

description (Boolean) Descrição do tipo.

createdAt (DateTime) Data e hora da criação do tipo de exame no banco.

7. Endpoints

A aplicação conta com um endpoint denominado **/doc**, que traz um relatório web estruturado em Swagger (<https://swagger.io/>), com a definição técnica da comunicação com a API por meio de cada rota, enquanto neste documento será realizada uma abordagem mais funcional de cada funcionalidade.

A apresentação será realizada na seguinte estrutura. **<Verbo HTTP utilizado> / <Rota>**

7.1. Listar laboratórios

GET /laboratory/index/:status

Parâmetros de url:

- **:status (obrigatório);** descrição do status do laboratório que queremos buscar, podendo variar em:
 - **“active”;** Busca apenas pelos laboratórios ativos.
 - **“inactive”;** Busca apenas pelos laboratório inativos.
 - **“all”;** Busca todos os laboratórios.

Retorno:

Uma lista com todos os laboratórios do sistema que respeitem o filtro **:status**

7.2. Retornar todos os dados de um laboratório específico

GET /laboratory/show/:labId

Parâmetros de url:

- **: labId (obrigatório);** valor de identificação (Campo id da collection *Laboratory*) do laboratório a ser procurado

Retorno:

Um objeto json contendo todos os dados do laboratório

7.3. Criar um ou mais novos laboratórios

POST /laboratory/store

Parâmetros de body (array de objetos):

- **name (obrigatório);** Nome do novo laboratório.
- **address (obrigatório);** Local onde fica situado o novo laboratório.

Retorno:

Uma lista com os mesmos objetos enviados acrescidos dos dados de cadastro.

Comportamento:

O laboratório recém criado sempre terá o campo status como ativo (true).

7.4. Atualizar um ou mais laboratório existente

PUT /laboratory/update

Parâmetros de body (array de objetos):

- **Id (obrigatório);** valor de identificação (Campo id da collection *Laboratory*) do laboratório a ser alterado
- **name;** Novo nome do laboratório.
- **address;** Local onde fica situado o novo laboratório.
- **status;** Flag que define se o laboratório está ativo (true) ou inativo (false).

Retorno:

Uma lista com os mesmos objetos enviados com seus respectivos dados atualizados

Comportamento:

Esse endpoint possibilita, além da alteração propriamente dita, também a remoção lógica do laboratório, alterando-se o campo *status* para false.

7.5. Apagar um ou mais laboratórios existente

DELETE /laboratory/destroy/:labIds

Parâmetros de url:

- **: labIds;** valor de identificação (Campo id da collection *Laboratory*) dos laboratório a serem apagados, separados entre si pelo caractere pipeline “|”

7.6. Listar exames

GET / exam/index/:status

Parâmetros de url:

- **:status (obrigatório)**; descrição do status do exame que queremos buscar, podendo variar em:
 - **"active"**; Busca apenas pelos exames ativos.
 - **"inactive"**; Busca apenas pelos exames inativos.
 - **"all"**; Busca todos os laboratórios.

Retorno:

Uma lista com todos os exames do sistema que respeitem o filtro *:status*

7.7. Retornar todos os dados de um exame específico .

GET / exam/show/: examId

Parâmetros de url:

- **: examId (obrigatório)**; valor de identificação (Campo id da collection *Exam*) do exame a ser procurado

Retorno:

Um objeto json contendo todos os dados do exame

7.8. Listar todos os laboratórios vinculados a um exame

GET /exam/:examName/laboratories

Parâmetros de url:

- **: examName (obrigatório)**; nome do exame (Campo name da collection *Exam*) a ser listado os laboratórios vinculados.

Retorno:

Uma lista com todos os dados dos laboratórios vinculados com o exame passado como referência.

7.9. Criando um ou mais novos exames

POST /exam/store

Parâmetros de body (array de objetos):

- **name (obrigatório)**; Nome do novo exame.
- **examTypeCode (obrigatório)**; valor de identificação (Campo id da collection *ExamType*) que definira o tipo do exame a ser criado.
- **laboratoryCode**; Uma array com todos os valores de identificação (Campo id da collection *Laboratory*) que serão vinculados ao novo exame.
-

Retorno:

Uma lista com os mesmos objetos enviados acrescidos dos dados de cadastro.

Comportamento:

Esse endpoint aceita no parâmetro *laboratoryCode* apenas ids de laboratórios que estejam ativos (status com valor true). O exame recém criado sempre terá o campo status como ativo (true).

7.10. Vincular um ou mais laboratórios a um exame

POST /exam/:examId/linkLaboratory

Parâmetros de url:

- **: examId (obrigatório)**; valor de identificação (Campo id da collection *Exam*) do exame que terá o vínculo dos laboratórios.

Parâmetros de body (array de objetos):

- **<array de strings com ids de laboratórios>**; Uma array com todos os valores de identificação (Campo id da collection *Laboratory*) que serão vinculados ao exame.

Comportamento:

Esse endpoint permite fazer vínculos se o exame e os laboratórios estiverem ativos (status com valor true). Ele incrementa a lista dos laboratórios anteriormente vinculados ao exame.

7.11. Atualizar um ou mais exames existente

PUT /exam/update

Parâmetros de body (array de objetos):

- **Id (obrigatório)**; valor de identificação (Campo id da collection *Exam*) do laboratório a ser alterado
- **name**; Novo nome do exame.
- **examTypeCode (obrigatório)**; valor de identificação (Campo id da collection *ExamType*) que definirá o novo tipo do exame.
- **laboratoryCode**; Uma array com todos os valores de identificação (Campo id da collection *Laboratory*) que substituirão os atualmente vinculados ao exame.
- **status**; Flag que define se o exame está ativo (true) ou inativo (false).

Retorno:

Uma lista com os mesmos objetos enviados com seus respectivos dados atualizados

Comportamento:

Esse endpoint possibilita, além da alteração propriamente dita, também a remoção lógica do laboratório, alterando-se o campo *status* para false.

Permite fazer vínculos de laboratórios apenas se o exame e os laboratórios estiverem ativos (status com valor true) ou forem ser atualizados para ativos.

7.12. Apagar um ou mais exames existente

DELETE /exam/destroy/:examIds

Parâmetros de url:

- **:examsIds**; valor de identificação (Campo id da collection *Exam*) dos exames a serem apagados, separados entre si pelo caractere pipeline “|”

7.13. Desvincular um ou mais laboratórios de um exame

DELETE /exam/:examId/ unlinkLaboratory

Parâmetros de url:

- **: examId (obrigatório)**; valor de identificação (Campo id da collection *Exam*) do exame que perderá o vínculo com os laboratórios.

Parâmetros de body (array de objetos):

- **<array de strings com ids de laboratórios>**; Uma array com todos os valores de identificação (Campo id da collection *Laboratory*) que serão desvinculados do exame.

Comportamento:

Esse endpoint permite desvincular apenas se o exame e os laboratórios estiverem ativos (status com valor true). Ele decrementa a lista dos laboratórios anteriormente vinculados ao exame.

7.14. Listar tipos de exame

GET /exam/type/index

Retorno:

Uma lista com todos os tipos de exames disponíveis na aplicação.

7.15. Retornar todos os dados de um tipo de exame específico

GET / exam/type/show/: examTypeId

Parâmetros de url:

- **: examTypeId (obrigatório)**; valor de identificação (Campo id da collection *ExamType*) do exame a ser procurado

Retorno:

Um objeto json contendo todos os dados do tipo de exame.

8. Melhorias

Infelizmente, não consegui concluir tudo que esperava implantar neste projeto até a data deste documento. Listo abaixo as features e melhorias que ficaram pendentes:

- Refatorar o tratamento de erros para que as mensagens do tipo Invalid input (405) fiquem mais assertivas na razão do erro.
- Adicionar métricas com o Prometheus (<https://prometheus.io/>) ou um sistema de log para acompanhamento das exceções da aplicação.
- Implantar testes automaticados com Jest (<https://jestjs.io/>),
- Publicar a API em um serviço cloud.