Tutoorial API RESTful com Fastify e Mongoose

MES, Sistemas de Informação para a Internet - 2022

Grupo

Nome	**Número de Estudante**
Edgar Santos	201600264

Professor: José Pereira

Setúbal, 5 de junho de 2022

Conteúdo

1 | Introdução

Neste tutorial vamos construir uma aplicação simples para gerir os produtos da ementa de um restaurante. O foco estará no desenvolvimento de uma solução backend em Node.js [10] utilizando a *framework* **fastify** [5] para a criação das rotas responsáveis por gerir todas as operações CRUD [11] dos nossos produtos. A nível de persistência será utilizada a biblioteca **mongoose** [9] para simplificar a modelação de objetos de uma base de dados MongoDB [7].

1.1 | Pré-requisitos

Para tirar o máximo partido deste tutorial, deverá compreender o fundamental dos princípios REST [12], bem como conhecer o básico de MongoDB e de uma *framework* web de Node.js, como o Express.js [4].

Será utilizada a sintaxe do ES6 [3], pelo qual se espera o conhecimento de conceitos como desestruturação, funções *arrow* e outros. De qualquer modo, todo o código o apresentado será explicado em detalhe.

De modo geral, os princípios demonstrados neste projeto não são particularmente complexos, pelo que este tutorial pode ser considerado como uma introdução às tecnologias listadas acima.

2 | Tutorial

2.1 | Preparando o Projeto

Assegure-se de que tem o Node.js instalado globalmente na sua máquina antes de começarmos; isto permitir-nos-á importar todas as nossas dependências e inicializar o projeto. Além disso, para a base de dados será utilizada uma *cluster* gratuito hospedado no MongoDB Atlas [8], mas poderá, em alternativa, utilizar uma instância local do MongoDB.

Vamos começar por criar uma nova pasta para o projeto, instalar as nossas dependências, e montar a estrutura. No terminal do seu computador execute o seguinte:

```
mkdir api-ementa
cd api-ementa
npm init -y
npm install fastify@3.29.0 mongoose@6.3.5 --save
mkdir src
touch src/index.js
```

Código 1: Comandos para a inicialização do projeto

Então, linha por linha: (1) criámos a pasta `api-ementa`, (2) entramos na pasta criada, (3) inicializámos um projeto Node, (4) instalámos o Fastify e o Mongoose como dependências, (5)criámos uma nova pasta `src` com (6) um ficheiro `index.js` que será a raiz da nossa aplicação.

2.1.1 | Inicializando o Fastify e Mongoose

Em seguida, vamos fazer a nossa aplicação Fastify funcionar com a ligação ao MongoDB.

Portanto, no ficheiro `index.js` que acabamos de criar vamos colocar o seguinte:

```
// Importar o fastify e o mongoose
const fastify = require("fastify");
const mongoose = require("mongoose");

// Inicializar o fastify
const app = fastify({
  logger: true,
});

// Conectar à base de dados
try {
  mongoose.connect(
    "mongodb+srv://tutorial:C86nzqqez2qeIFr9@cluster0.494hi.mongodb.net/api-ementa?retryWrites=true&w=
     majority"
  );
} catch (e) {
  console.error(e);
}

// Rota principal
app.get("/", (request, reply) => {
  try {
    reply.send("Server alive!");
  } catch (e) {
    console.error(e);
  }
});

// Colocar à aplicação à escuta na porta 5000 em localhost
app.listen(8080, (err, address) => {
  if (err) {
    app.log.error(err);
    process.exit(1);
  }
  console.log(`Server running on ${address}`);
});
```

Código 2: Ficheiro index.js inicial

Começamos por importar o `fastify` e `mongoose` e utilizámos a função `fastify()` para iniciar a aplicação. Como a função `mongoose.connect()` nos permite especificar um URI para ligar a nossa base de dados à aplicação, utilizou-se o URI de conexão fornecido pelo MongoDB Atlas na secção de ligações, como dito anteriormente, poderá ser utilizada qualquer instância local ou remota de MongoDB. Deverá ainda de atribuir um nome à base de dados do projeto no final do URI, no caso deste exemplo foi atribuído o nome `api-ementa`. Poderá atribuir o nome que quiser, se a instância do MongoDB não tiver uma base de dados com esse nome, a mesma será automaticamente criada.

A aplicação trata então da rota principal com a função `app.get()`. Isto trata de um pedido GET para a raiz da nossa aplicação (`"/"`). Ao utilizar uma aplicação Fastify para tratar uma rota, devemos fornecer uma função que contenha o `request` e `reply` como parâmetros. Como seria de esperar, a primeira contém todos os detalhes do pedido, enquanto a segunda é utilizada para enviar uma resposta formatada para o destino. Por enquanto, simplesmente devolvemos uma string com `reply.send()`.

Por fim, utilizamos a função `app.listen` para fazer a nossa aplicação ficar à escuta na porta 8080. Agora é altura de testarmos. Para o fazer, vá até ao ficheiro `package.json`, na raiz da nossa aplicação, e adicione o seguinte script `"start"` ao objeto `"scripts"`:

```
{
  "name": "api-ementa",
  "version": "1.0.0",
  "description": "Example of a RESTful API using fastify and mongoose",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js"
  },
  "author": "Edgar Santos <edgarbs1998@gmail.com>",
  "license": "MIT",
  "dependencies": {
    "fastify": "^3.29.0",
    "mongoose": "^6.3.5"
  }
}
```

Código 3: Ficheiro package.json

Agora basta executar `npm run start` a partir do seu terminal e deverá ver uma mensagem a dizer ”Server running on http://127.0.0.1:8080”. Abra o seu browser em http://localhost:8080 e deverá ver a nossa string ”Server alive!”, o que significa que a nossa aplicação está a funcionar!

2.2 | Definindo o Modelo de Dados

Precisamos de criar uma coleção para os produtos da nossa ementa na nossa base de dados, bem como descrever as características que um documento de produto deve conter. Podemos faze-lo estabelecendo um Modelo de Produto baseado recorrendo a um `Model Schema`, que é um objeto que contém todas as especificações de um documento normal de um produto. Vamos fazer uma nova pasta e ficheiro dentro do nosso projeto; num terminal dentro da sua pasta de projeto, escreva:

```
mkdir src/models
touch src/models/product.model.js
```

Código 4: Comandos para criar os modelos

Vamos adicionar ao novo ficheiro o seguinte código:

```
const mongoose = require("mongoose");
const { Schema } = mongoose;

const productSchema = new Schema({
  name: { type: String, required: true },
  description: { type: String },
  price: { type: Number, required: true },
```

```
});

const Product = mongoose.model("product", productSchema);

module.exports = Product;
```

Código 5: Ficheiro models/product.model.js

Essencialmente, estamos a utilizar o objeto `Schema` do Mongoose para dizer a MongoDB que os nossos produtos devem conter uma propriedade obrigatória chamada `name` do tipo string para o nome do produto, uma propriedade opcional chamada `description` também do tipo string para a descrição do produto e uma propriedade obrigatória chamada `price` do tipo número para o preço do produto.

A função `mongoose.model` é então utilizada para transformar este esquema num modelo. A primeira opção é uma string que MongoDB utilizará para definir o nome da coleção: se tivermos um modelo `"product"`, veremos uma coleção `"products"` na nossa base de dados após a nossa primeira ação com um `Product`. Finalmente, o modelo de produto recentemente construído é exportado e disponibilizado em toda a aplicação.

2.3 | Estruturando a API REST

Devemos agora criar as nossas rotas de api para as operações de CRUD que iremos realizar nos nossos produtos. Utilizaremos as bem conhecidas e amplamente aceites convenções REST para abordar as nossas APIs. Para os não familiarizados com REST, é um estilo arquitetónico que estabelece um conjunto de requisitos para ajudar os programadores na construção de APIs de uma forma clara e consistente.

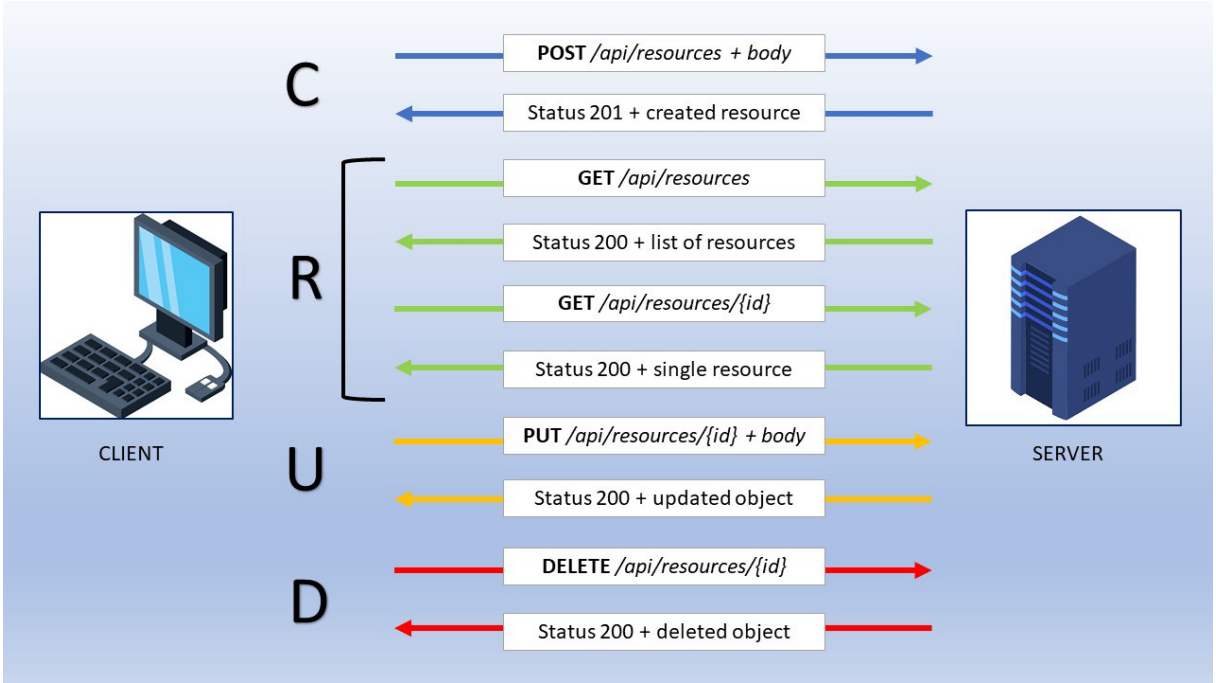

Figura 2.1: Princípios CRUD

No diagrama acima é possível visualizar como as várias rotas devem ser estruturados para todas as operações (Criar, Ler, Atualizar, e Apagar) e o que cada um deles deverá retornar. De acordo com estes princípios, vamos separar as nossas rotas em dois ficheiros, um para especificar as rotas e o outro para definir o seu comportamento, isto é, as funções que irão fazer as operações reais e devolver a resposta certa a partir das rotas.

2.3.1 | Definindo as Rotas (Endpoints)

Dentro da nossa pasta `src`, vamos agora criar uma nova pasta `routes`, com um ficheiro `products.routes.js` dentro. No terminal, vamos escrever o seguinte:

```
mkdir src/routes
touch src/routes/products.routes.js
```

Código 6: Comandos para criar as rotas

Neste ficheiro iremos exportar uma função que leva três parâmetro: a aplicação Fastify, e devolve uma lista de rotas a serem adicionadas à aplicação. Especificamos o verbo HTTP a ser tratado, um URL, e, finalmente, uma função *handler*, que iremos construir numa fase posterior, para cada rota.

```
module.exports = (api) => {
  // Rota para criar um produto
  api.post("/", (request, reply) => {});

  // Rota para obter uma lista de produtos
  api.get("/", (request, reply) => {});

  // Rota para obter um único produto
  api.get("/:id", (request, reply) => {});

  // Rota para atualizar um produto
  api.patch("/:id", (request, reply) => {});

  // Rota para apagar um produto
  api.delete("/:id", (request, reply) => {});
};
```

Código 7: Ficheiro models/products.routes.js

Com estas rotas, somos agora capazes de realizar uma chamada HTTP do tipo:

- **POST** à rota `api/products` para Criar um produto
- **GET** à rota `api/products` para obter uma Lista de todos os produtos
- **GET** à rota `api/products/:id` para Obter os detalhes de um dado produto
- **PATCH** à rota `api/products/:id` para Atualizar um dado produto
- **DELETE** à rota `api/products/:id` para Apagar um dado produto

O `:id` é um *placeholder* para o identificador do nosso documento, permitindo-nos assim aceder ao mesmo através dos parâmetros de entrada do nosso pedido HTTP, na prática utilizando `request.params.id`.

Vale notar que por pré-definição o Fastify aceita apenas JSON (`application/json`) como tipo de entrada para o `body`. É possível adicionar suporte ao tipo de dados de um formulário HTML comum (`application/x-www-form-urlencoded`) bastando para isso adicionar o *plugin* **@fastify/formbody** [1], demonstrado na secção ???. à semelhança do **Express.js** também é possível suportar o carregamento de ficheiros através do `multipart/form-data` recorrendo ao *plugin* **@fastify/multipart** [2], não abordado neste tutorial.

Como visto acima, precisamos de acesso ao objeto da aplicação fastify para executar os manipuladores das rotas sobre ele, por isso devemos importar este ficheiro para o nosso `src/index.js` e chamá-lo com o app fastify como seu parâmetro. O Fastify permite-nos registar rotas através da função `app.register()` e definir prefixos para as mesmas, o que poderemos utilizar para simplificar a nomeação das nossas rotas e evitar erros. Deste modo, começamos por definir um roteador geral para a aplicação com o prefixo `"/api"` e dentro dele registamos um novo roteador para os nossos produtos que fará uso do nosso ficheiro de rotas e terá o prefixo `"/products"`. Resultado das alterações ao ficheiro `src/index.js` em seguida:

```
// ...

// Importar o ficheiro de rotas de produtos
const productRoutes = require("./routes/product.routes");
```

```
5
6  // Inicializar o fastify
7  const app = fastify({
8    logger: true,
9  });
10
11 // ...
12
13 // Regista um roteador para as rotas da api
14 app.register(
15   async (api) => {
16     // Regista o roteador dos produtos
17     api.register(productsRoutes, {
18       prefix: "/products",
19     });
20   },
21   {
22     prefix: "/api",
23   }
24 );
25
26 // Colocar à aplicação à escuta na porta 5000 em localhost
27 app.listen(8080, (err, address) => {
28   if (err) {
29     app.log.error(err);
30     process.exit(1);
31   }
32   console.log(`Server running on ${address}`);
33 });
```

Código 8: Ficheiro index.js atualizado com as rotas

2.3.2 | Definindo os Controladoras (Handlers)

Depois de termos definido as nossas rotas, teremos de definir *handlers* para cada uma delas. Por uma questão de simplicidade deslocaremos as nossas funções de manipulação para um ficheiro externo, onde criaremos a lógica necessária e a exportaremos e as designaremos para as suas respetivas rotas. Estes ficheiros são referidos como controladores. Como primeiro passo, estabelecer uma pasta de controladores no nosso diretório `src` e adicionar um novo ficheiro intitulado `products.controller.js`. Vamos abrir o nosso terminal dentro da pasta raiz do projeto e escrever:

```
1 mkdir src/controllers
2 touch src/controllers/products.controller.js
```

Código 9: Comandos para criar os controladores

Vamos agora trabalhar no novo controlador:

```
1  const Product = require("../models/product.model");
2
3  module.exports = {
4    // Criar um produto
5    create: async (request, reply) => {},
6
7    // Obter a lista de produtos
8    fetch: async (request, reply) => {},
9
10   // Obter um produto
11   get: async (request, reply) => {},
12
13   // Atualizar um produto
14   update: async (request, reply) => {},
15
16   // Apagar um produto
17   delete: async (request, reply) => {},
18 };
```

Código 10: Ficheiro controllers/products.controller.js

Antes de mais, temos de importar o nosso modelo de Produto, que será utilizado para realizar as transações na base de dados utilizando os métodos do Mongoose. Construímos então uma função para cada operação e marcámos-los como assíncronos, porque a nossa base de dados exigirá algum tempo para completar o seu processamento e não poderemos continuaras nossas operações enquanto tais pedidos não forem concluídos.

Vamos colocar a implementação dos *handlers* em espera por agora e apenas conectar o nosso ficheiro de rotas a estas funções recentemente declaradas. Importamos o controlador e atribuímos as funções apropriadas dentro do ficheiro `products.routes.js`:

```
const productsController = require("../controllers/products.controller");

module.exports = async (api) => {
  // Rota para criar um produto
  api.post("/", productsController.create);

  // Rota para obter uma lista de produtos
  api.get("/", productsController.fetch);

  // Rota para obter um único produto
  api.get("/:id", productsController.get);

  // Rota para atualizar um produto
  api.patch("/:id", productsController.update);

  // Rota para apagar um produto
  api.delete("/:id", productsController.delete);
};
```

Código 11: Ficheiro routes/products.routes.js atualizado com o controlador

2.4 | Implementar os Controladores

Vamos agora concentrar-nos em cada uma das nossas funções *handler* dentro do `products.controller.js`.

2.4.1 | Criar um Produto

Para criar um produto, devemos primeiro extrair informações do corpo (`request`) do pedido sobre o novo produto que vamos criar. Em seguida, utilizaremos o método `create` do Mongoose para criar um novo documento e devolvê-lo. Podemos agora definir o estado HTTP apropriado (201 Created) e devolver o nosso novo produto, de acordo com as convenções REST:

```
// Criar um produto
create: async (request, reply) => {
  try {
    const product = request.body;
    const newProduct = await Product.create(product);
    reply.code(201).send(newProduct);
  } catch (e) {
    reply.code(500).send(e);
  }
},
```

Código 12: Função para criar um produto

2.4.2 | Obter a Lista de Proputos

Esta é provavelmente a função mais simples. Receberemos o nosso pedido GET e executaremos a função de pesquisa no nosso modelo Product, passando um objeto vazio como parâmetro. O primeiro parâmetro é um objeto de consulta do MongoDB que especifica quais os produtos a obter com base em determinados critérios. Como resultado, o objeto vazio devolverá toda a coleção de produtos sem quaisquer restrições. Podemos então retornar os mesmos com o seguinte:

```
// Obter a lista de produtos
fetch: async (request, reply) => {
```

```
  try {
    const products = await Product.find({});
    reply.code(200).send(products);
  } catch (e) {
    reply.code(500).send(e);
  }
},
```

Código 13: Função para listar os produtos

2.4.3 | Obter um Dado Produto

O primeiro passo desta vez será obter o identificador único do produto que queremos do URL de pedido. Quando incluí-mos `placeholders` no URL da rota (como fizemos com o `:id` nas nossas rotas), eles aparecerão no objeto `request.params`. Após a obtenção do `id`, podemos utilizar o método `findById()` no modelo Product para recuperar o produto desejado, fornecendo o `id` como única parâmetro.

```
// Obter um produto
get: async (request, reply) => {
  try {
    const productId = request.params.id;
    const product = await Product.findById(productId);
    reply.code(200).send(product);
  } catch (e) {
    reply.code(500).send(e);
  }
},
```

Código 14: Função para obter um dado produto

2.4.4 | Atualizar um Produto

A atualização implica que extraímos tanto o corpo do pedido como o identificador do produto do URL. As características do produto que necessitam de ser alteradas estarão no `body`, enquanto o `id` será utilizado para determinar qual o produto que precisa de ser atualizada. Podemos utilizar o método `findByIdAndUpdate()` com estas duas informações, fornecendo o `id` da nota como primeiro argumento e os dados do `body` como segundo argumento.

Ao contrário dos métodos `create`, `find` e `findById`, `findByIdAndUpdate` não devolve o objeto alterado por pré-definição, para alterar este comportamento é necessário fornecer um terceiro argumento à mesma, argumento este que deverá ser um objeto que defina a propriedade `"new"` como verdadeiro, de modo a que possamos responder com a nota recentemente alterada.

```
// Atualizar um produto
update: async (request, reply) => {
  try {
    const productId = request.params.id;
    const updates = request.body;
    const updatedProduct = await Product.findByIdAndUpdate(
      productId,
      updates,
      {
        new: true,
      }
    );
    reply.code(200).send(updatedProduct);
  } catch (e) {
    reply.code(500).send(e);
  }
},
```

Código 15: Função para atualizar um dado produto

2.4.5 | Apagar um Produto

A eliminação de um produto será o último **handler** a ser feito. Neste caso não precisamos do **body**, tudo o que precisamos de fazer é obter o identificador do URL e fornecê-la ao método **findByIdAndDelete**. Esta função ao ser executada retorna o documento atual antes de eliminar o mesmo, pelo qual basta apenas devolve-lo na nossa resposta:

```
// Apagar um produto
delete: async (request, reply) => {
  try {
    const productId = request.params.id;
    const deletedProduct = await Product.findByIdAndDelete(productId);
    reply.code(200).send(deletedProduct);
  } catch (e) {
    reply.code(500).send(e);
  }
},
```

Código 16: Função para apagar um dado produto

2.4.6 | Controlador Final

Pronto! Finaliza-mos o nosso ficheiro do controlador de produtos:

```
const Product = require("../models/product.model");

module.exports = {
  // Criar um produto
  create: async (request, reply) => {
    try {
      const product = request.body;
      const newProduct = await Product.create(product);
      reply.code(201).send(newProduct);
    } catch (e) {
      reply.code(500).send(e);
    }
  },

  // Obter a lista de produtos
  fetch: async (request, reply) => {
    try {
      const products = await Product.find({});
      reply.code(200).send(products);
    } catch (e) {
      reply.code(500).send(e);
    }
  },

  // Obter um produto
  get: async (request, reply) => {
    try {
      const productId = request.params.id;
      const product = await Product.findById(productId);
      reply.code(200).send(product);
    } catch (e) {
      reply.code(500).send(e);
    }
  },

  // Atualizar um produto
  update: async (request, reply) => {
    try {
      const productId = request.params.id;
      const updates = request.body;
      const updatedProduct = await Product.findByIdAndUpdate(
        productId,
        updates,
        {
          new: true,
        }
```

```
      );
      reply.code(200).send(updatedProduct);
    } catch (e) {
      reply.code(500).send(e);
    }
  },

  // Apagar um produto
  delete: async (request, reply) => {
    try {
      const productId = request.params.id;
      const deletedProduct = await Product.findByIdAndDelete(productId);
      reply.code(200).send(deletedProduct);
    } catch (e) {
      reply.code(500).send(e);
    }
  },
};
```

Código 17: Ficheiro controller/products.controller.js atualizado com os handlers

2.5 | Plugin @fastify/formbody

O Fastify por pré-definição suporta apenas JSON no `body`. Este *plugin* simples adiciona um *parser* para o tipo de conteúdo `application/x-www-form-urlencoded`.

Primeiramente precisamos de instalar este plugin como uma dependência do nosso projeto:

```
npm install @fastify/formbody@6.0.1 --save
```

Código 18: Comandos para a instalar o plugin @fastify/formbody

Em Fastify podemos considerar que um *Middleware* é um *Plugin*, na realidade, à semelhança do JavaScript em que tudo é um objeto, em Fastify tudo é um *plugin*. Para utilizar este *plugin* na aplicação basta registar o mesmo na aplicação fastify:

```
// Importar o fastify e o mongoose
const fastify = require("fastify");
const mongoose = require("mongoose");
const formbody = require("@fastify/formbody");

// ...

// Inicializar o fastify
const app = fastify({
  logger: true,
});

// Registar plugins do Fastify
app.register(formbody);

// ...
```

Código 19: Ficheiro index.js com @fastify/formbody

3 | Conclusão

Com este projeto foi possível explorar como utilizar a *framework* fastify para o desenvolvimento de operações CRUD básicas do lado do servidor. Podemos ainda explorar como utilizar plugins/middlewares no fastify, mais concretamente como suportar diferentes *parsers* para o *body*.

Esta aplicação de exemplo, apesar que simples, demonstra o potencial das *frameworks* web e o quanto estas podem ajudar tanto no tempo de desenvolvimento como na qualidade do produto final.

Poderá encontrar todo o código do projeto desenvolvido em https://github.com/IPS-MES-Projects/mes-sii-trabalho-tp/Exemplo/api-ementa.

Referências

[1] *@fastify/formbody*. npm. URL: https://www.npmjs.com/package/@fastify/formbody.

[2] *@fastify/multipart*. npm. URL: https://www.npmjs.com/package/@fastify/multipart.

[3] *ECMAScript 6: New Features: Overview and Comparison*. URL: http://es6-features.org/#Constants.

[4] *Express - Node.js web application framework*. URL: https://expressjs.com/.

[5] *Fastify, Fast and low overhead web framework, for Node.js*. URL: https://www.fastify.io/.

[6] Marco Lancellotti. *Fullstack CRUD Application With Fastify, Mongoose and React Admin (Part 1 — Backend)*. The Startup. URL: https://medium.com/swlh/fullstack-crud-application-with-fastify-mongoose-and-react-admin-86d3e743dcdf.

[7] *MongoDB — Build Faster. Build Smarter*. MongoDB. URL: https://www.mongodb.com.

[8] *MongoDB Atlas Database — Multi-Cloud Database Service*. MongoDB. URL: https://www.mongodb.com/atlas/database.

[9] *Mongoose ODM v6.3.5*. URL: https://mongoosejs.com/.

[10] Node.js. *Node.js*. Node.js. URL: https://nodejs.org/en/.

[11] *What is CRUD?* Codecademy. URL: https://www.codecademy.com/article/what-is-crud.

[12] *What is REST*. REST API Tutorial. URL: https://restfulapi.net/.