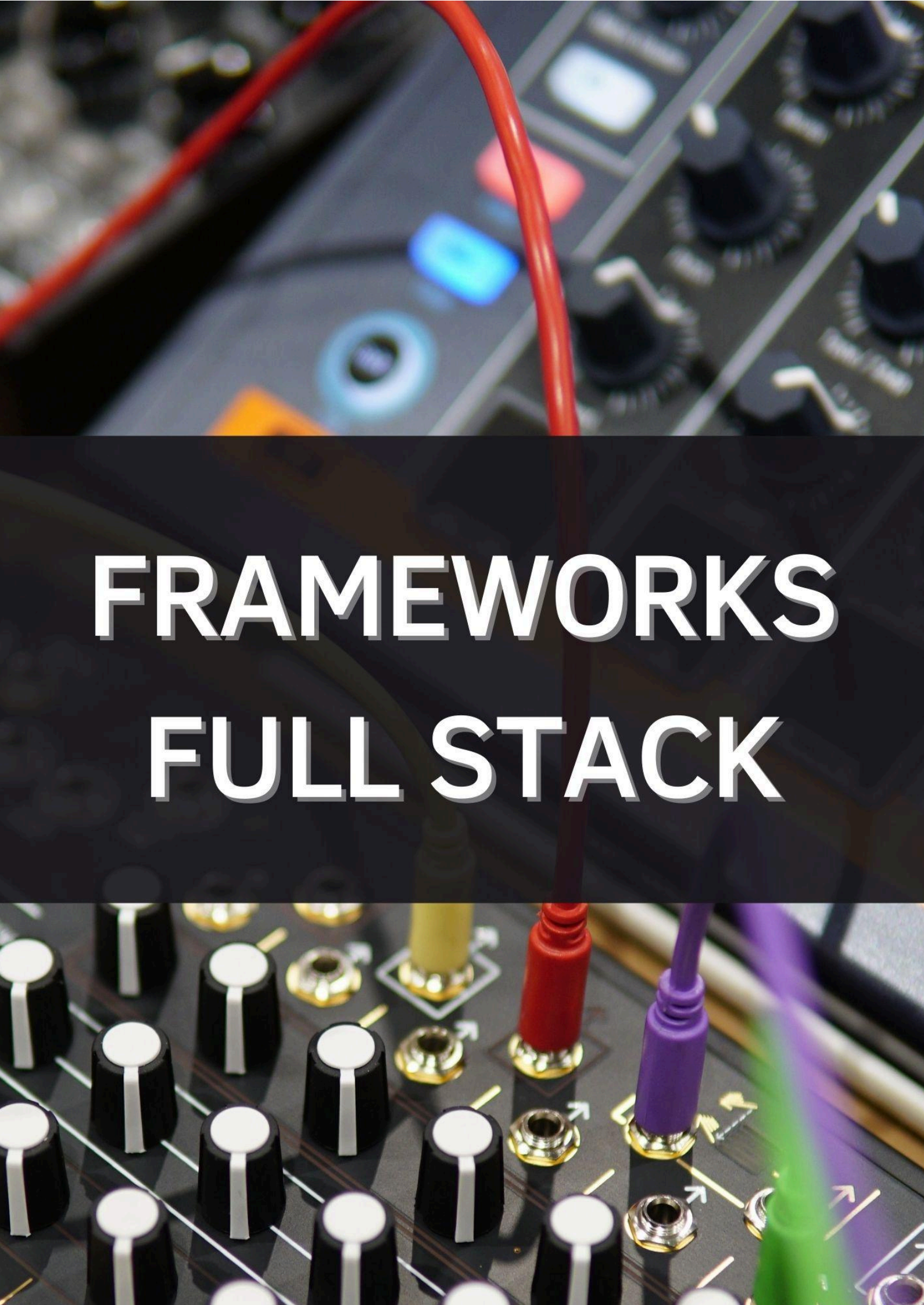

FRAMEWORKS
FULL STACK

Texto base

7

Revisão de Python + Flask e implementação da busca de produtos

Prof. André Nascimento Maia
Prof. Caio Nascimento Maia

Resumo

Grande parte das aplicações de front-end utilizando Next.js precisam, frequentemente, fazer consultas em API REST para compor informações de tela para visualização e manipulação de dados pelos usuários. Neste texto, construiremos uma busca de produtos disponíveis para venda em um e-commerce utilizando o framework Next.js. Adicionalmente, também será construída uma API REST utilizando a linguagem de programação Python e o framework Flask e a integração entre o front-end e back-end.

7.1. Introdução

Frequentemente aplicações web de *front-end* precisam consultar dados de uma API REST para exibi-los aos usuários. Utilizaremos o contexto de um e-commerce simples e fictício, que exibe uma lista de produtos disponíveis para compra.

A lista de produtos contém informações básicas como foto, título, valor e valor parcelado, com juros ou sem juros. Esta lista, inicialmente é implementada em memória, com valores fixos e em seguida, o código é ajustado para obter os dados de uma API REST local.

Para a criação de uma API REST, é utilizada a linguagem de programação Python e o *framework* Flask. Essa aplicação de *back-end* disponibiliza uma única API REST para consultar a lista de produtos disponíveis.

7.2. Revisão de Python e Flask

Flask é um dos mais famosos *frameworks* de desenvolvimento de serviços web utilizando a linguagem Python.

Para a criação de uma aplicação mínima com Flask, é necessário importar a classe Flask, criar uma instância dessa classe, criar uma função que retorne um texto HTML que será renderizado no navegador e decorar essa função com @app.route(). O Código 7.1 mostra o código mínimo para uma aplicação que retorna o HTML com a frase Olá, Mundo! dentro de um parágrafo.

Código 7.1: Aplicação mínima utilizando Flask.

```
01. from flask import Flask
02.
03. app = Flask(__name__)
04.
05. @app.route("/")
06. def ola_mundo():
07.     return "<p>Olá, Mundo!</p>"
```

Fonte: autores, 2022.

Para executar essa aplicação é necessário salvar o texto do Código 7.1 em um arquivo chamado hello.py, por exemplo, exportar a variável de ambiente com chamada FLASK_APP com o conteúdo igual ao nome dos arquivos e executar o comando flask run. O Código 7.2 mostra os comandos para executar a aplicação hello e a mensagem que deve aparecer no terminal após os comandos serem executados com sucesso.

Código 7.2: Comandos para inicializar uma aplicação Flask chamada hello.

```
$ export FLASK_APP=hello
$ flask run
 * Running on http://127.0.0.1:5000/
```

Fonte: autores, 2022.

Para evoluir essa aplicação para uma web api completa, é necessário criar todos os recursos conforme as funcionalidades que são necessárias serem expostas via APIs seguindo os padrões REST. O Código 7.3 mostra uma API que retorna a data e hora do servidor no formato UTC, cujo retorno é uma objeto JSON com um único atributo chamado *now* que contém a data e hora. Utilizando *jsonify* do Flask, garantimos que o resultado é um JSON válido, além disso, automaticamente, o *framework* já adiciona o *Content-Type HTTP Header* ao *response* da chamada com o valor de *application/json*, para garantir que o cliente saiba interpretar o *payload* que chega até ele no formato correto.

Código 7.3: API para obter data e hora do servidor em UTC em um objeto JSON.

```
01. from flask import Flask
00. from datetime import datetime
00. from flask import jsonify
02.
03. app = Flask(__name__)
04.
00. @app.route("/timestamps", methods=['GET'])
00. def get():
00.     return jsonify({
00.         'now': datetime.utcnow().isoformat(),
00.     })
```

Fonte: autores, 2022.

Grande parte das aplicações utilizam banco de dados relacionais para armazenar e consultar dados de usuário. Uma escolha muito comum para acesso a dados que tem uma ótima integração com Flask é o SQLAlchemy.

7.2.1. Acesso a dados com SQLAlchemy

SQLAlchemy é considerado um conjunto de ferramentas e também um sistema de mapeamento objeto-relacional, do inglês object-relational mapping (ORM) (Brown & Wilson, 2012, 291). Através de diversos dialetos, o SQLAlchemy consegue se conectar a diversos bancos de dados diferentes sempre utilizando os mesmos objetos e APIs.

O Código 7.4, 7.5 e 7.6 demonstram alguns exemplos para configuração da conexão com o banco de dados, criação de uma entidade chamada Product, criação de alguns objetos e a escrita desses objetos no banco de dados utilizando o banco de dados SQLite (SQLite Consortium, 2022) em memória para manipulação e armazenamento de dados.

Código 7.4: Inicializando o banco de dados SQLite.

```
01. # Inicializa a aplicação Flask com configurações padrão
02. app = Flask(__name__)
03.
04. # Inicializa a conexão com um banco de dados SQLite em memória.
05. app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
06. db = SQLAlchemy(app)
```

Fonte: autores, 2022.

Código 7.5: Criação da entidade Product utilizando SQLAlchemy

```
# Cria a entidade de produto
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(255), unique=False, nullable=False)
    amount = db.Column(db.Numeric, unique=False, nullable=False)
    installments = db.Column(db.Integer, unique=False, nullable=False)
    installments_fee = db.Column(db.Boolean, unique=False, nullable=False)

    def __repr__(self):
        return '<Product %r>' % self.title

    @property
    def serialized(self):
        return {
            'id': self.id,
            'title': self.title,
            'amount': round(self.amount, 2),
            'installments': {
                'number': self.installments,
                'total': round(self.amount / self.installments, 2)
            }
        }
```

Fonte: autores, 2022.

Código 7.6: manipulação de dados com SQLAlchemy.

```
# Cria o banco de dados em memória
db.create_all()

# Cria dois produtos
product1 = Product(
    title='Caneca Personalizada de Porcelana', amount=123.45,
    installments=3, installments_fee=False)

product2 = Product(
    title='Caneca de Tulipa', amount=123.45,
    installments=3, installments_fee=False)

# Insere os dois produtos no banco de dados em memória
db.session.add(product1)
db.session.add(product2)

```

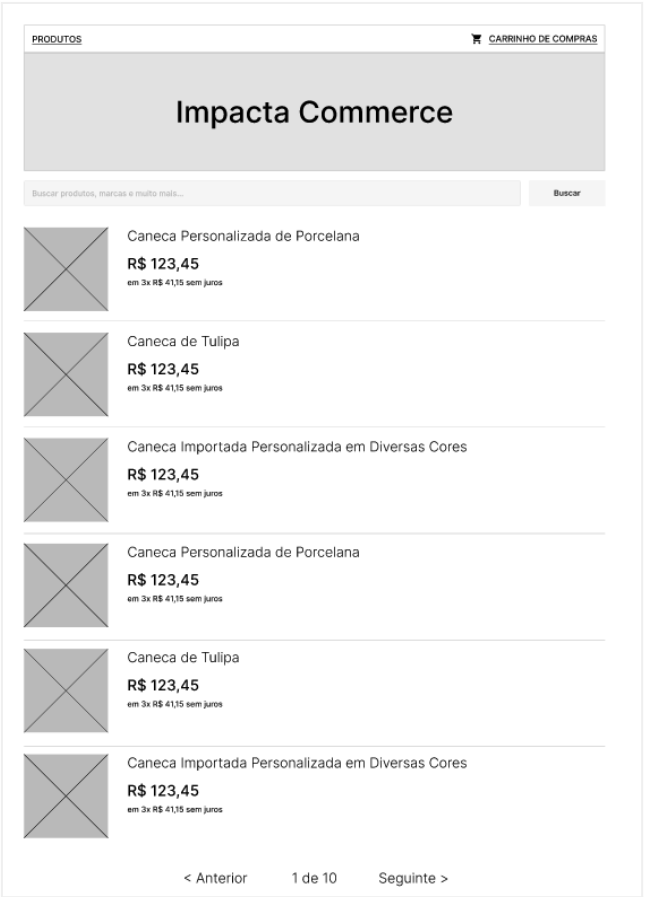
```
# Commit da transaction do banco de dados
db.session.commit()

# Consulta de Product utilizando LIKE
all_products = Product.query.filter(
    Product.title.like(f'%{query}%')
).all()
```

Fonte: autores, 2022.

7.2. Busca de produtos em um e-commerce

Como exemplos de busca de dados em uma API REST, utilizaremos um e-commerce fictício. Considere o protótipo de tela de busca de produtos disponíveis para venda a Figura 7.1.

Figura 7.1: Protótipo de tela de busca e listagem de produtos.



Fonte: autores, 2022.

No protótipo da Figura 7.1, temos uma campo livre de busca e um botão buscar. Logo abaixo, temos o resultado da busca que é uma lista de produtos onde cada linha é um produto com foto, título, valor e valor parcelado.

Decompondo o protótipo da Figura 7.1 em componentes React, teremos a hierarquia a seguir:

- FilterableProductsForSaleList;
  - SearchBar;
  - ProductsForSaleList;
    - ProductListItem;
      - Installment;

Conforme as recomendações para a criação de componentes React, iniciaremos com esse protótipo e todas as informações que serão exibidas estáticas na *web application*. O Código 7.7 exibe o modelo de dados utilizado para a exibição de lista de produtos.

Código 7.7: Modelo de dados para o preenchimento da lista de produtos.

```
01. {
02.   "query": "Caneca",
03.   "size": 10,
04.   "start": "MA=="
05.   "results": [
06.     {
07.       "code": "8C9F552D-D22C-4B76-AAF6-233F806F6484",
08.       "title": "Caneca Personalizada de Porcelana",
09.       "amount": 123.45,
10.       "installments": { "number": 3, "total": 41.15, "hasFee": true }
11.     },
12.     {
13.       "code": "ED1F26D1-81B2-424E-A79F-DA74E9DDBAB7",
14.       "title": "Caneca de Tulipa",
15.       "amount": 123.45,
16.       "installments": { "number": 3, "total": 41.15 }
17.     }
18.   ]
19. }
```

Fonte: autores, 2022

O modelo de dados proposto no Código 7.7 é o resultado paginado de uma API REST que contém uma lista de produtos (linha 5), e algumas informações sobre o

resultado da busca, como: o parâmetro de busca informado para o filtro (linha 02); o tamanho máximo de produtos retornados na busca (linha 03); um parâmetro que serve de ponteiro para a posição da página de busca, codificado em *base64* (linha 04).

Este modelo é mais que suficiente para exibirmos as informações necessárias no protótipo proposto na Figura 7.1. O código 7.8 mostra a implementação do componente SearchBar utilizado para manipular os dados informados pelo usuário na busca e o Código 7.9 mostra a utilização da SearchBar dentro de um componente chamado FilterableProductsForSaleList.

Código 7.8: Implementação do componente SearchBar.

```
01. function SearchBar(props) {
02.   const [query, setQuery] = useState("");
03.
04.   function searchProducts() { props.onSearch(query); }
05.   function handleChange(e) { setQuery(e.target.value); }
06.
07.   function handleKeyDown(e) {
08.     if (e.key === 'Enter') {
09.       searchProducts()
10.     }
11.   }
12.
13.   return (
14.     <div className="row g-3">
15.       <div className="col">
16.         <input
17.           type="text"
18.           className="form-control"
19.           placeholder="Buscar produtos, marcas e muito mais..."
20.           onChange={handleChange}
21.           onKeyDown={handleKeyDown}
22.         />
23.       </div>
24.       <div className="col-auto">
25.         <button className="btn btn-primary" onClick={searchProducts}>
26.           Buscar
27.         </button>
28.       </div>
29.     </div>
30.   );
31. }
```

Fonte: autores, 2022.

O SearchBar implementa duas formas de interatividade do usuário: através do clique no botão Buscar (linha 25) ou quando o usuário pressiona o botão Enter do teclado (linha 21). Em ambos os casos, um evento onSearch deste componente elevará o seu estado para o componente pai.

Código 7.8: Implementação do componente FilterableProductsForSaleList.

```
01.import { queryProducts } from "../api/api";
02.
03. function FilterableProductsForSaleList() {
04.   const [productsSearchResult, setProductsSearchResult] = useState({
05.     results: [],
06.   });
07.
08.   function clientSideQueryProduct(query) {
09.     queryProducts(query).then(
10.       (json) => {
11.         setIsLoaded(true);
12.         setProductsSearchResult(json);
13.       },
14.       (error) => {
15.         setIsLoaded(true);
16.         setError(error);
17.       }
18.     );
19.   }
20.   useEffect(() => { clientSideQueryProduct(""); }, []);
21.
22.  function handleOnSearch(query) { clientSideQueryProduct(query); }
23.  const result = productsSearchResult.results.map((x, index) => (
24.    <ProductListItem product={x} key={index} />
25.  ));
26.
27.  return (
28.    <>
29.      <SearchBar onSearch={handleOnSearch} />
30.      <div>{result}</div>
31.    </>
32.  );
33. }
```

Fonte: autores, 2022.

O Código 7.8 do componente FilterableProductsForSaleList importa uma função chamada queryProducts(). Essa função foi criada para executar uma chamada para a API de busca de produtos disponíveis para venda criada em Python, utilizando o *framework* Flask e manipulando os dados de produtos em um banco de dados relacional em memória. O Código 7.9, mostra a implementação da chamada para essa API.

Código 7.9: Biblioteca para chamada para as APIs.

```
01. export async function queryProducts(query) {
02.   return fetch(`http://127.0.0.1:5000/products?query=${query}`)
03.     .then((response) => response.json())
04. }
```

Fonte: autores, 2022.

No Código 7.9, a chamada para a API utiliza um endereço fixo local na porta 5000. O recurso consultado é *products*, que retorna uma lista de produtos disponíveis para venda. Também é passado um parâmetro adicional chamado *query* com o texto informado pelo usuário para filtro dos produtos cujo o título combina com o texto informado.

7.4. API REST de busca de produtos

A API REST utiliza Flask e SQLAlchemy para acessar um banco de dados SQLite em memória.

Após configurar o SQLAlchemy junto do Flask, basta apenas que seja criado uma nova função que mapeie a rota */products* e aceite o parâmetro *query* que forçará o filtro dos produtos, quando informado. O Código 7.10 mostra a implementação da função que mapeia essa nova rota.

Conforme esperado pela *web application* o resultado é sempre uma página de produtos disponíveis baseado no parâmetro de filtro informado pelo usuário. Esse resultado em JSON contém mais informações que o necessário para mostrar os produtos, e já está preparado para uma futura paginação dos dados retornados pela API, contendo o tamanho da página (a quantidade de elementos retornados na consulta) e também um cursor chamado *start* (linha 17) codificado em *base64,* que aponta em qual elemento da página a consulta deve começar.

Código 7.10: Função que mapeia rota /products

```
01. @app.route("/products", methods=['GET'])
02. def get_products():
03.     """Cria uma rota para consulta de produtos baseada no parâmetro
query."""
04.     args = request.args.to_dict()
05.     query = args.get("query")
06.
```

```
    if query is None:
        all_products = Product.query.all()
    else:
        all_products = Product.query.filter(
            Product.title.like(f'%{query}%')
        ).all()

    return jsonify({
        'query': query,
        'size': 10,
        'start': 'MA==',
        'results': [p.serialized for p in all_products]
    })
```

Fonte: autores, 2022.

Referências

BROWN, A; WILSON, G. (Eds.). (2012). **The architecture of open source applications**, Volume II. CreativeCommons.

SQLite Consortium. (2022). **SQLite.** SQLite Home Page. Disponível em: <https://www.sqlite.org/index.html>. Acesso em: 6 fev. 2022.