API + CRUD

PROGRAMAÇÃO WEB 1

Objetivos de Aprendizagem

Agenda

- CRUD
- CRUD + REST
- Exemplos

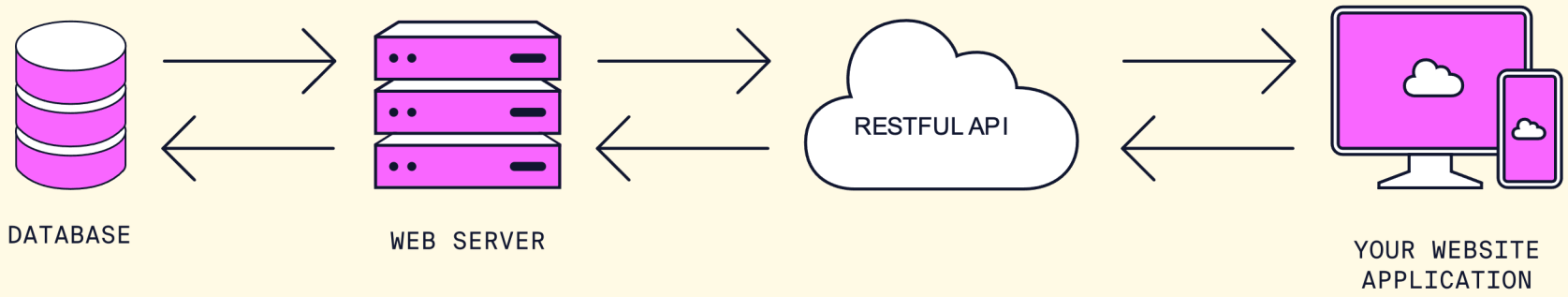
“Create, Read, Update and Delete (CRUD) são as quatro funções básicas que modelos ou bancos de dados devem realizar.”

https://www.codecademy.com/article/what-is-crud

CRUD

- Create, Retriev, Update e Delete
- Paradigma comumente utilizado no desenvolvimento de *software*, especialmente aplicações web

What is Rest API?



Exemplo

- Considerando como exemplo uma aplicação que gerencia bibliotecas
- Espera-se que exista um recurso associado a livros, como mostrado
- A aplicação deve ser capaz de completar operações CRUD

```
“book”: {
  "id": <Integer>,
  “title”: <String>,
  “author”: <String>,
  “isbn”: <Integer>
}
```

CREATE

- Função que deve ser utilizada para **adicionar** um novo livro ao catálogo
- Deve passar valores: *title*, *author* e ISBN

```
“book”: {
  "id": <Integer>,
  “title”: <String>,
  “author”: <String>,
  “isbn”: <Integer>
}
```

READ

- Função que deve ser utilizada para **ler**, acessar ou buscar os livros no catálogo
- Variações que busquem um livro ou grupos devem informar id ou algum dos outros parâmetros
- Apenas leitura, sempre

```
“book”: {
  "id": <Integer>,
  “title”: <String>,
  “author”: <String>,
  “isbn”: <Integer>
}
```

UPDATE

- Função que deve ser utilizada para **atualizar**, alterar informações sobre um livro ou um grupo no catálogo
- Um critério de seleção de grupo deve ser fornecido
  - De um mesmo autor
  - Título que contém uma palavra chave
  - ISBN maior que um determinado valor
- Novos valores de *title* e/ou *author* e/ou *isbn* devem ser fornecidos para realizar a operação

```
“book”: {
  "id": <Integer>,
  “title”: <String>,
  “author”: <String>,
  “isbn”: <Integer>
}
```

DELETE

- Função que deve ser utilizada para **remover** um livro
- Variações dessa função podem apagar múltiplos livros selecionados por algum critério similar ao UPDATE
- Deve ser usado com cuidado

```
“book”: {
  "id": <Integer>,
  “title”: <String>,
  “author”: <String>,
  “isbn”: <Integer>
}
```

Métodos

- Boas práticas de implementação REST recomendam que cada chamada CRUD deva ser realizada com um método HTTP distinto
  - CREATE -> POST
  - RETRIEVE -> GET
  - UPDATE -> PUT
  - DELETE -> DELETE
- Trata-se apenas de uma recomendação

Métodos

- Cada método retorna códigos de *status* distintos, após as solicitações
  - POST, Status Code 201 (CREATED)
  - GET, Status Code 200 (OK)
  - PUT, Status Code 200 (OK)
  - DELETE, Status Code 204 (NO CONTENT)

Status Codes

- Outros códigos de *status* disponíveis no protocolo HTTP

Status Code	Significado
200 (OK)	Resposta padrão para respostas bem sucedidas do HTTP
201 (CREATED)	Item criado com sucesso
204 (NO CONTENT)	Resposta HTTP bem sucedida com corpo vazio
400 (BAD REQUEST)	Requisição HTTP possui erro de sintaxe, tamanho além do permitido ou erros similares
404 (NOT FOUND)	O recurso solicitado não foi encontrado
500 (INTERNAL SERVER ERROR)	Resposta padrão para respostas com falhas inesperadas. Um erro de programação de um *endpoint* pode causar essa resposta

Exemplo

USERS API

Users API

- API para gerenciamento de usuários em um banco de dados
- *Endpoints* que serão utilizados:
  - Para **buscar** todos os usuários, GET /users
  - Para **cadastrar** um novo usuário, POST /users
  - Para **atualizar** um novo usuário, PUT /users/:id
  - Para **apagar** um usuário, DELETE /users/:id
- *Endpoint* extra para buscar um usuário específico
  - Para buscar todos os usuários, GET /users/:id

/users

- *Endpoint* para buscar os usuários cadastrados
- Operações RETRIEV são as mais comuns
- Saída em formato JSON
- Considere a lista **db** a representação do Banco de Dados

```
const db = [
  {
    id: 1,
    firstName: 'John',
    lastName: 'Doe',
    email: 'jd@example.com'
  },
  {
    id: 2,
    fistName: 'Jane',
    lastName: 'Warwick',
    email: 'jane@example.com'
  },
  {
    id: 3,
    firstName: 'Jim',
    lastName: 'Smith',
    email: 'jim@example.com'
  }
];
```

/users

```
const express = require("express");
const app = express();
const port = 3000;

app.listen(port, () => {
  console.log(`Users API listening at ${port}`);
});

const db = [ 18 hidden lines ];

app.get("/users", (req, res) => {
  res.json(users);
});
```

/users/:id

- *Endpoint* para buscar informação sobre um usuário específico

```
const express = require("express");
const app = express();
const port = 3000;

app.listen(port, () => {
  console.log(`Users API listening at ${port}`);
});

const db = [ 18 hidden lines ];

app.get("/users", (req, res) => { 1 hidden line });
```

```
app.get("/users/:id", (req, res) => {
  const user = db.find(u => u.id ===
parseInt(req.params.id));
  res.json(user);
});
```

POST `/users`

- Para adicionar novos usuários é necessário observar se o `id` é único dentre os já existentes
- Em aplicações reais com BD essa tarefa é feita pelo próprio BD

POST /users

```
app.use(express.json())

app.listen(port, () => { 1 hidden line });

const db = [ 18 hidden lines ];

app.get("/users", (req, res) => { 1 hidden line });

app.get("/users/:id", (req, res) => { 2 hidden lines });
```

```
app.post("/users", (req, res) => {
  let lastId = Math.max(...db.map(u => u.id));
  const user = {
    id: ++lastId,
    firstName: req.body.fName,
    lastName: req.body.lName,
    email: req.body.e,
  };
  db.push(user);
  res.json(db);
});
```

DELETE /users/:id

```
app.get("/users", (req, res) => { 1 hidden line });

app.get("/users/:id", (req, res) => { 2 hidden lines });

app.post("/users", (req, res) => { 9 hidden lines });

app.delete("/users/:id", (req, res) => {
  db = db.filter(u => u.id !== req.params.id)
  res.json(db);
});
```

PUT /users/:id

```
app.get("/users/:id", (req, res) => { 2 hidden lines });

app.post("/users", (req, res) => { 9 hidden lines });

app.delete("/users/:id", (req, res) => { 2 hidden lines });
```

```
app.put("/users/:id", (req, res) => {
  const index = db.findIndex(u => u.id === parseInt(req.params.id));
  db[index] =
    {
      id: req.params.id,
      firstName: req.body.fName,
      lastName: req.body.lName,
      email: req.body.e
    }
  res.json(db);
});
```

Exercício

API PARA LIVRARIA

Exercício

- Criar uma API que realize funções CRUD em um BD de uma livraria com os seguintes requisitos:
  - Cada objeto livro deve ter, pelo menos, as propriedades id, titulo, autor, editora, valor, ano, quant, preço
  - Implementar as operações CRUD básicas
  - Implementar uma operação para buscar os livros de uma determinada editora
  - Implementar operação para buscar o livro que o título possui uma palavra chave específica
  - Implementar operação para buscar os livros acima de um determinado valor
  - Implementar operação para buscar os livros abaixo de um determinado valor
  - Implementar operação para buscar os livros mais recentes
  - Implementar operação para buscar os livros mais antigos
  - Implementar operação para buscar os livros sem estoque
  - Implementar operação para buscar os livros acima de um determinado valor
  - Implementar operação para buscar os livros abaixo de um determinado valor
  - Em caso de acesso a um *endpoint* inexistente deve ser exibido o erro 404

Referências

- https://www.freecodecamp.org/news/create-crud-api-project/#crud-api-example
- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/find
- https://www.codecademy.com/article/what-is-rest