

Objetivos:

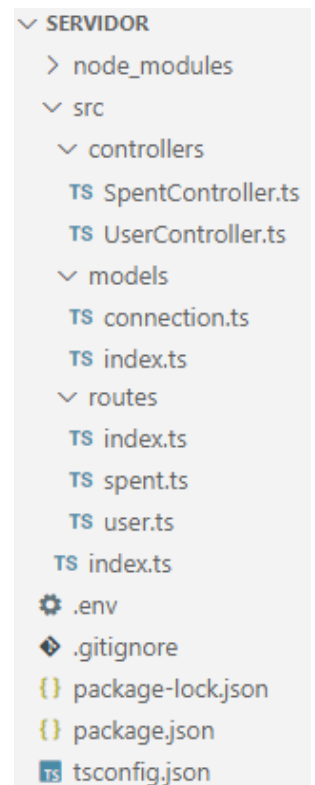
- I. Introdução ao MongoDB;
- II. Esquema, Modelo e Documento no Mongoose;
- III. Relacionamento entre documentos;
- IV. Validações no Mongoose;
- V. Aplicação usando MongoDB.

Siga as instruções para criar o projeto para reproduzir os exemplos:

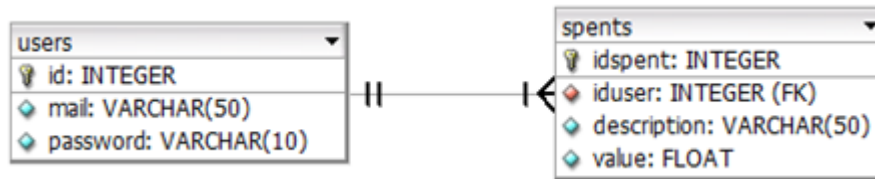
- a) Crie uma pasta de nome `servidor` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
- b) No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node, arquivo `package.json`;
- c) No terminal, execute o comando `npm i express dotenv mongoose` para instalar os pacotes. O mongoose é uma biblioteca usada para fazer o acesso ao MongoDB (<https://www.npmjs.com/package/mongoose>);
- d) No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém as definições de tipos do pacote express. Quando usamos um pacote é preciso ter acesso às declarações de tipo do pacote para que o TS saiba quais tipos de dados esperar do framework;
- e) No terminal, execute o comando `npm i -D ts-node ts-node-dev typescript` para instalar os pacotes `ts-node`, `ts-node-dev` e `typescript` como dependências de desenvolvimento;
- f) No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo `tsconfig.json`);
- g) Crie o arquivo `.gitignore` na raiz do projeto e coloque a linha para ignorar a pasta `node_modules`;
- h) Crie o arquivo `.env` na raiz do projeto e coloque a seguinte variável de ambiente:
`PORT = 3001`
- i) Coloque as seguintes propriedades no arquivo `package.json`. Elas serão utilizadas para criar as tabelas no SGBD e rodar a aplicação:

```
"scripts": {  
  "start": "ts-node ./src",  
  "dev": "ts-node-dev ./src"  
},
```
- j) Coloque no arquivo `src/index.ts` o código para subir o servidor express.

Estrutura de pastas e arquivos do projeto:



Nos exemplos considere as tabelas `users` e `spents` (gastos) representadas no modelo:



i. Introdução ao MongoDB

O MongoDB é um BD, mas não é um Sistema de Gerenciamento de Banco de Dados Relacional (SGBD-R) tradicional. Ele pertence à categoria de BD NoSQL (Not Only SQL). A principal diferença entre BD NoSQL, como o MongoDB, e BD relacionais é a forma como eles armazenam e organizam os dados:

- Modelo de dados NoSQL: o MongoDB utiliza um modelo de dados NoSQL baseado em **documentos**, onde os dados são armazenados em documentos BSON (Binary JSON -formato binário JSON-like);
- Esquema dinâmico: ao contrário dos BD relacionais, o MongoDB permite esquemas dinâmicos, o que significa que os documentos em uma coleção podem ter campos diferentes sem a necessidade de um esquema fixo, como nas tabelas de um SGBD-R;
- Consultas baseadas em documentos: as consultas no MongoDB são feitas utilizando a sintaxe de consulta de documentos BSON, o que facilita a interação com os dados. No SGBD-R são feitas usando a linguagem SQL.

Para instalar o MongoDB sugere-se fazer o download da versão Community (gratuita) <https://www.mongodb.com/try/download/community>.

O vídeo <https://www.youtube.com/watch?v=l4HeaNRi8f8> pode ajudar na instalação.

O MongoDB Compass é uma interface gráfica de usuário (GUI), instalada juntamente com o MongoDB, que facilita as tarefas administrativas, tais como, visualizar dados, criar bancos e collections (coleções) e gerenciar permissões de usuários.

No MongoDB, os dados são organizados hierarquicamente da seguinte forma:

- BD:
 - É a unidade mais alta de armazenamento de dados, análogo a um BD no SGBD-R;
 - No MongoDB podem existir vários bancos e eles são independentes entre si;
 - Um BD pode conter várias coleções.
- Coleção (collection):
 - Uma coleção é análoga a uma tabela no SGBD-R;
 - Uma coleção é um grupo de documentos;
 - As tabelas de um SGBD-R definem um esquema fixo para todos os registros, mas as coleções não impõem um esquema fixo aos documentos.
- Documento (document):
 - Um documento é uma unidade básica de dados no MongoDB e é representado em formato BSON (Binary JSON -formato binário JSON-like);
 - Um documento é análogo a uma linha (registro) de uma tabela do SGBD-R;

- Enquanto todos os registros de uma tabela no SGBD-R compartilham o mesmo esquema (estrutura de colunas), os documentos de uma coleção não compartilham o mesmo esquema (propriedades do JSON). Desta forma, não se tem um esquema fixo.

ii. Esquema, Modelo e Documento no Mongoose

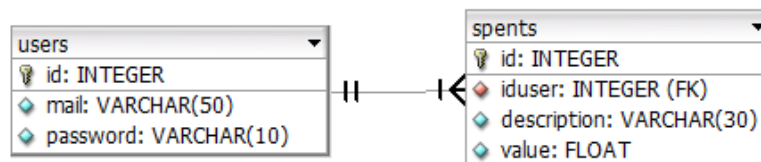
No MongoDB, um **esquema** refere-se à estrutura ou à definição de como os documentos em uma coleção específica devem ser organizados. Embora o MongoDB seja conhecido por ser um BD NoSQL orientado a documentos, que não impõe um esquema fixo, a utilização de esquemas pode ser aplicada em níveis mais alto, como na camada de aplicação, ou seja, nos programas que fazem a conexão como BD do MongoDB.

O Mongoose é uma biblioteca de modelagem de objetos MongoDB para Node.js (<https://mongoosejs.com>). Ele fornece uma camada de abstração sobre o MongoDB, simplificando a interação com o BD e adicionando funcionalidades extras.

O Mongoose permite que os desenvolvedores definam **esquemas** de dados no Node.js, que são modelos para os documentos que serão armazenados no MongoDB. Ele também fornece recursos adicionais, como validação de esquema, middlewares e métodos de consulta.

Enquanto o MongoDB é o BD NoSQL que armazena e gerencia dados, o Mongoose é uma biblioteca que facilita o desenvolvimento de aplicativos Node.js que interagem com o MongoDB, adicionando uma camada de abstração e funcionalidades extras para trabalhar com dados de maneira mais estruturada.

Como exemplo, considere o modelo de dados relacional a seguir. No modelo tem-se que 1 usuário pode ter N gastos.



Cada esquema no Mongoose mapeia para uma coleção o MongoDB e define a estrutura dos documentos da coleção. A seguir tem-se a definição dos esquemas que representam as estruturas para os documentos das coleções users e spends.

```

import mongoose from "mongoose";
const { Schema } = mongoose;

const UserSchema = new Schema({
  mail: { type: String, maxLength: 50, required: true },
  password: { type: String, minlength: 6, maxlength: 10, select: false, required: true }
});

const SpentSchema = new Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  description: { type: String, maxlength: 30, required: true },
  value: { type: Number, required: true }
});
  
```

Para usar um esquema é necessário converter em um modelo. A função `mongoose.model` compila o modelo:

- O 1º argumento precisa ser um nome singular da coleção que representa o modelo. O `mongoose` irá gerar a coleção com o nome plural, ou seja, o nome de modelo `User` é para a coleção `users` no BD;
- O 2º argumento precisa ser o modelo usado para gerar o modelo da coleção. A função `model` faz um cópia do schema.

```
const User = mongoose.model("User", UserSchema);  
const Spent = mongoose.model("Spent", SpentSchema);
```

O modelo é uma classe que usamos para construir os documentos. Neste caso, cada documento será um objeto com as propriedades e comportamentos declarados no esquema. A variável `doc` possui um documento criado usando o modelo `User`:

```
const doc = new User({ mail: 'a@teste.com', password: 'abcdef' });
```

Como exemplo, a variável `doc` terá o seguinte conteúdo. O MongoDB adiciona o campo `_id` para ser o identificador do documento, algo tal como uma chave primária. Esse valor é do tipo `Schema.Types.ObjectId` formado por 12 bytes, geralmente, representado como uma string hexadecimal de 24 caracteres:

```
{  
  mail: 'a@teste.com',  
  password: 'abcdef',  
  _id: new ObjectId('659acc6bd202c436bb835d2d')  
}
```

Ao chamar o método `save`, o documento será adicionado na coleção `users` do MongoDB:

```
const resp = await doc.save();
```

A seguir tem-se o documento na coleção `users` do MongoDB. O campo `__v` é usado para controlar a versão dos documentos e facilitar a resolução de conflitos em ambientes de atualização concorrente. Ele é incrementado automaticamente sempre que um documento é atualizado no BD:

```
_id: ObjectId('659acc6bd202c436bb835d2d')  
mail: "a@teste.com"  
password: "abcdef"  
__v: 0
```

Diferenças entre esquema, modelo e documento:

- Esquema:
 - Definição: é uma estrutura que define os campos, tipos de dados e opções de validação para documentos em uma coleção MongoDB;
 - Finalidade: definir a estrutura dos documentos em termos de campos, seus tipos e quaisquer validações específicas que devem ser aplicadas.
- Modelo:
 - Definição: é uma representação compilada de um esquema. Ele é usado para interagir com uma coleção específica no MongoDB;

- Finalidade: responsável por realizar operações CRUD (Create, Read, Update, Delete) na coleção associada ao esquema.
- Documento:
 - Definição: é uma instância específica de um modelo, representando um registro na coleção;
 - Finalidade: são os objetos reais armazenados no MongoDB. Eles seguem a estrutura definida pelo esquema associado ao modelo.

iii. Relacionamento entre documentos

No modelo relacional o relacionamento de chave estrangeira ocorre entre entidades, onde um campo de uma tabela faz referência a chave primária de outra tabela. No MongoDB implementamos o mesmo conceito de chave estrangeira usando relacionamento entre os documentos. O campo `user`, definido no esquema `SpentSchema`, recebe como conteúdo o identificador de um documento da coleção `users`. A referência é definida através da propriedade `ref`:

```
const SpentSchema = new Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  description: { type: String, maxlength: 30, required: true },
  value: { type: Number, required: true }
});
```

Para construir um documento precisamos fornecer no campo `user` o id de um documento disponível na coleção `users`:

```
const gasto = new Spent({
  user: '659acc6bd202c436bb835d2d',
  description: 'Combustível',
  value: 10.25
});
```

Como exemplo, a variável `gasto` terá o seguinte conteúdo:

```
{
  user: new ObjectId('659acc6bd202c436bb835d2d'),
  description: 'Combustível',
  value: 10.25,
  _id: new ObjectId('659adecb3ee01c5a994d8077')
}
```

Também é possível representar a coleção `spents` como subdocumento do esquema `users` no Mongoose. Isso significa que os documentos da coleção `spents` estarão aninhados dentro dos documentos da coleção `users`. No exemplo a seguir a propriedade `spents` recebe um array de documentos do tipo `Spent`:

- `SpentSchema` é definido independentemente e representa a estrutura para os documentos na coleção `spents`;
- No `UserSchema`, o campo `spents` é definido como um array de subdocumentos usando o esquema `SpentSchema`. Isso permite que vários documentos da coleção `spents` sejam aninhados dentro de cada documento da coleção `users`.

```
import mongoose from "mongoose";
const { Schema } = mongoose;
```

```
const SpentSchema = new Schema({
  description: { type: String, maxLength: 30, required: true },
  value: { type: Number, required: true }
});

// define o schema
const UserSchema = new Schema({
  mail: { type: String, maxLength: 50, required: true },
  password: { type: String, minLength: 6, maxLength: 10, select: false, required: true },
  spends: [SpentSchema]
});
```

A seguir tem-se um documento do esquema UserSchema, os subdocumentos do esquema SpentSchema são embutidos no array spends:

```
const object = new User({
  mail: "b@teste.com",
  password: "abcdef",
  spends: [
    { description: "Oficina", value: 191.75 },
    { description: "Mercado", value: 28.42 }
  ]
});
const resp = await object.save();
```

Na MongoDB será criada somente a coleção users e o um documento terá a seguinte estrutura. Observe que cada subdocumento possui o seu próprio identificador:

```
{
  mail: 'b@teste.com',
  password: 'abcdef',
  spends: [
    {
      description: 'Oficina',
      value: 191.75,
      _id: new ObjectId('659b3ef2d5f3a37ea511b9df')
    },
    {
      description: 'Mercado',
      value: 28.42,
      _id: new ObjectId('659b3ef2d5f3a37ea511b9e0')
    }
  ],
  _id: new ObjectId('659b3ef2d5f3a37ea511b9de')
}
```

A abordagem de subdocumento pode ser útil se os gastos estão fortemente relacionados aos usuários e não precisam ser acessados independentemente. No entanto, a escolha entre incorporar ou manter a relação como referência depende dos requisitos específicos da aplicação.

iv. Validações no Mongoose

No Mongoose, as validações são definidas no esquema e aplicadas quando tentamos criar ou atualizar um documento usando um modelo.

Na definição do esquema podemos adicionar as validações como parte da definição de cada campo, assim como fazemos em `required`: `[true, "O e-mail é obrigatório"]` ou podemos definir nossas próprias validações, assim como fizemos para validar se o e-mail possui o formato correto. No caso do `required` a validação já existe pronta (built-in) e no caso da formatação tivemos de usar a propriedade `validate` para construir a nossa validação.

```
import mongoose from "mongoose";
const { Schema } = mongoose;

const UserSchema = new Schema({
  mail: {
    type: String,
    maxlength: [50, "O e-mail pode ter no máximo 30 caracteres"],
    unique: true,
    required: [true, "O e-mail é obrigatório"],
    validate: {
      validator: function (value: string) {
        // expressão regular para validar o formato do e-mail
        const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return regex.test(value);
      },
      message: (props:any) => `${props.value} não é um formato de e-mail válido`,
    },
  },
  password: {
    type: String,
    trim: true,
    minlength: [6, "A senha precisa ter no mínimo 6 caracteres"],
    maxlength: [10, "A senha precisa ter no máximo 10 caracteres"],
    select: false,
    required: [true, "A senha é obrigatória"],
  }
});
```

Considere como exemplo o seguinte código de teste. A variável `document` recebe o documento criado usando o modelo `User`. O método `validateSync` é utilizado para verificar se os campos estão em conformidade com as regras definidas no esquema. A validação é realizada antes de salvar no BD usando o método `save`.

Os `ifs` no código são usados para tratar os erros de validação nos campos `mail` e `password`.

Suponha que o usuário esteja tentando criar um usuário com um e-mail que não atenda à expressão regular definida no esquema. O método `validateSync` identificará esse problema e retornará um objeto de erro contendo informações específicas sobre a falha de validação no e-mail.

```
public async create(req: Request, res: Response): Promise<Response> {
  const { mail, password } = req.body;
  try {
    const document = new User({ mail, password });
    let error = document.validateSync();
    if (error && error.errors["mail"]) {
      return res.json({ message: error.errors["mail"].message });
    } else if (error && error.errors["password"]) {
      return res.json({ message: error.errors["password"].message });
    } else {
      const resp = await document.save();
      return res.json(resp);
    }
  } catch (error: any) {
    if (error.code === 11000 || error.code === 11001) {
      // código 11000 e 11001 indica violação de restrição única (índice duplicado)
      return res.json({ message: "Este e-mail já está em uso" });
    }
    return res.json({ message: error.message });
  }
}
```

Exemplos de teste:

- O caso de teste

```
const document = new User({ mail:"", password:"abcdef" });
```

retornará a mensagem "O e-mail é obrigatório" por não satisfazer a regra
required: [true, "O e-mail é obrigatório"];

- O caso de teste

```
const document = new User({ mail:"@teste", password:"abcdef" });
```

retornará a mensagem "@teste não é um formato de e-mail válido" por não satisfazer a regra de validação de formato de e-mail:

```
validate: {
  validator: function (value: string) {
    // expressão regular para validar o formato do e-mail
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(value);
  },
}
```



```
message: (props:any) => `${props.value} não é um formato de e-mail válido`,
}
```

As validações podem ser built-in (prontas), assim como required, ou podemos definir nossas próprias validações assim como fizemos aqui para validar o formato de e-mail.

- Se repetirmos o caso de teste

```
const document = new User({ mail:"a@teste.com", password:"abcdef" });
```

ele apresentará o erro "Este e-mail já está em uso" a partir da 2ª vez, por não satisfazer a restrição de valor único de e-mail definido por unique: true. Porém, a restrição de índice único não consegue ser validada antes de salvar no BD, por este motivo, será lançada uma exceção e precisaremos tratar essa exceção no bloco catch.

v. Aplicação usando MongoDB

No arquivo connection.ts definiremos a conexão com BD do MongoDB.

Arquivo: src/models/connection.ts

```
import mongoose from "mongoose";

// A URI indica o IP, a porta e BD a ser conectado
const uri = "mongodb://127.0.0.1:27017/bdaula";

export default function connect() {
  // Configura manipuladores de eventos para diferentes estados de conexão
  // cada mensagem de log indica um estado específico da conexão.
  // É opcional configurar os manipuladores de estado,
  // mas é interessante para sabermos sobre a conexão
  mongoose.connection.on("connected", () => console.log("connected"));
  mongoose.connection.on("open", () => console.log("open"));
  mongoose.connection.on("disconnected", () => console.log("disconnected"));
  mongoose.connection.on("reconnected", () => console.log("reconnected"));
  mongoose.connection.on("disconnecting", () => console.log("disconnecting"));
  mongoose.connection.on("close", () => console.log("close"));
  // Utiliza o método connect do Mongoose para estabelecer a conexão com o MongoDB, usando a URI
  mongoose
    .connect(uri, {
      serverSelectionTimeoutMS: 5000,
      maxPoolSize: 10,
    })
    .then(() => console.log("Conectado ao MongoDB"))
    .catch((e) => {
      console.error("Erro ao conectar ao MongoDB:", e.message);
    });

  // o sinal SIGINT é disparado ao encerrar a aplicação, geralmente, usando Ctrl+C
  process.on("SIGINT", async () => {
    try {
      console.log("Conexão com o MongoDB fechada");
      await mongoose.connection.close();
      process.exit(0);
    }
  });
}
```

```
    } catch (error) {  
      console.error("Erro ao fechar a conexão com o MongoDB:", error);  
      process.exit(1);  
    }  
  });  
}
```

No arquivo `src/index.ts` subiremos o servidor express e abriremos uma conexão com uma instância do MongoDB chamando a função `connect`.

A conexão com o BD deve ser estabelecida antes que a aplicação comece a lidar com solicitações HTTP. Isso garante que a conexão esteja pronta para ser usada quando necessário.

Não é recomendado abrir e fechar a conexão com o BD em cada requisição HTTP, pois isso pode ser ineficiente e impactar no desempenho. Em vez disso, é mais comum usar um pool de conexões ou uma abordagem de conexão única durante a vida útil da aplicação. O Mongoose gerencia o pool de conexões, desta forma, a cada operação do Mongoose no BD será utilizada a conexão aberta.

Arquivo: `src/index.ts`

```
import express from "express";  
import routes from './routes';  
import dotenv from "dotenv";  
import connect from './models/connection';  
dotenv.config();  
  
// será usado 3000 se a variável de ambiente não tiver sido definida  
const PORT = process.env.PORT || 3000;  
const app = express(); // cria o servidor e coloca na variável app  
// suportar parâmetros JSON no body da requisição  
app.use(express.json());  
  
// conecta ao MongoDB no início da aplicação  
connect();  
  
// inicializa o servidor na porta especificada  
app.listen(PORT, () => {  
  console.log(`Rodando na porta ${PORT}`);  
});  
  
// define a rota para o pacote /routes  
app.use(routes);
```

No arquivo `models/index.ts` definimos os esquemas e modelos. O relacionamento 1:N entre o usuário e gastos foi implementado criando o campo `user` no esquema `SpentSchema`. Infelizmente, ao fazer a inserção/atualização de um documento na coleção `spents` não será feita a verificação se o `ObjectId` existe na coleção `users`, gerando assim uma

inconformidade no relacionamento entre as coleções. Para evitar esse problema foi adicionada a validação no campo `user`, com o objetivo de fazer uma consulta `User.findById(id)` na coleção `users` para verificar se o `ObjectId` existe.

Arquivo: `src/models/index.ts`

```
import mongoose from "mongoose";
const { Schema } = mongoose;

// define os schemas
const UserSchema = new Schema({
  mail: {
    type: String,
    maxlength: [50, "O e-mail pode ter no máximo 30 caracteres"],
    unique: true,
    required: [true, "O e-mail é obrigatório"],
    validate: {
      validator: function (value: string) {
        // expressão regular para validar o formato do e-mail
        const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return regex.test(value);
      },
      message: (props: any) =>
        `${props.value} não é um formato de e-mail válido`,
    },
  },
  password: {
    type: String,
    trim: true,
    minlength: [6, "A senha precisa ter no mínimo 6 caracteres"],
    maxlength: [10, "A senha precisa ter no máximo 10 caracteres"],
    select: false,
    required: [true, "A senha é obrigatória"],
  },
});

const SpentSchema = new Schema({
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: true,
    validate: {
      validator: async function (id:string) {
        const user = await User.findById(id); // verifica se id existe na coleção users
        return !!user; // true se o usuário existir
      },
      message: 'O usuário fornecido não existe',
    },
  },
  description: {
```

```

    type: String,
    maxlength: 30,
    required: [true, "A descrição é obrigatória"],
  },
  value: {
    type: Number,
    required: [true, "O valor é obrigatório"],
  },
});

// mongoose.model compila o modelo
const User = mongoose.model("User", UserSchema);
const Spent = mongoose.model("Spent", SpentSchema);

export { User, Spent };

```

No arquivo controllers/UserController.ts faremos o CRUD (Create, Read, Update e Delete) na coleção users. Serão aplicadas as validações do esquema UserSchema ao chamar o método **save** do documento.

O resultado da validação será tratado no bloco **catch**, pelo fato da validação de registro único ocorrer apenas ao inserir/atualizar na coleção do MongoDB, as demais validações podem ser feitas usando o método validateSync do documento, porém causaria redundância, pois aplicaríamos a validação ao chamar validateSync e, posteriormente, ao chamar o **save**.

Arquivo: src/controllers/UserController.ts

```

import { Request, Response } from "express";
import { User } from "../models";

class UserController {
  public async create(req: Request, res: Response): Promise<Response> {
    const { mail, password } = req.body;
    try {
      //a instância de um modelo é chamada de documento
      const document = new User({ mail, password });
      // ao salvar serão aplicadas as validações do esquema
      const resp = await document.save();
      return res.json(resp);
    } catch (error: any) {
      if (error.code === 11000 || error.code === 11001) {
        // código 11000 e 11001 indica violação de restrição única (índice duplicado)
        return res.json({ message: "Este e-mail já está em uso" });
      } else if (error && error.errors["mail"]) {
        return res.json({ message: error.errors["mail"].message });
      } else if (error && error.errors["password"]) {
        return res.json({ message: error.errors["password"].message });
      }
      return res.json({ message: error.message });
    }
  }
}

```

```
}

public async list(_: Request, res: Response): Promise<Response> {
  try {
    const objects = await User.find().sort({ mail: "asc" });
    return res.json(objects);
  } catch (error: any) {
    return res.json({ message: error.message });
  }
}

public async delete(req: Request, res: Response): Promise<Response> {
  const { id: _id } = req.body; // _id do registro a ser excluído
  try {
    const object = await User.findByIdAndDelete(_id);
    if (object) {
      return res.json({ message: "Registro excluído com sucesso" });
    } else {
      return res.json({ message: "Registro inexistente" });
    }
  } catch (error: any) {
    return res.json({ message: error.message });
  }
}

public async update(req: Request, res: Response): Promise<Response> {
  const { id, mail, password } = req.body;
  try {
    // busca o usuário existente na coleção antes de fazer o update
    const document = await User.findById(id);
    if (!document) {
      return res.json({ message: "Usuário inexistente" });
    }
    // atualiza os campos
    document.mail = mail;
    document.password = password;
    // ao salvar serão aplicadas as validações do esquema
    const resp = await document.save();
    return res.json(resp);
  } catch (error: any) {
    if (error.code === 11000 || error.code === 11001) {
      // código 11000 e 11001 indica violação de restrição única (índice duplicado)
      return res.json({ message: "Este e-mail já está em uso" });
    } else if (error && error.errors["mail"]) {
      return res.json({ message: error.errors["mail"].message });
    } else if (error && error.errors["password"]) {
      return res.json({ message: error.errors["password"].message });
    }
    return res.json({ message: error.message });
  }
}
```

```
    }  
  }  
}  
  
export default new UserController();
```

No arquivo controllers/SpentController.ts faremos o CRUD (Create, Read, Update e Delete) na coleção spents. Serão aplicadas as validações do esquema SpentSchema ao chamar o método `save` do documento.

Ao inserir/atualizar um gasto o método `save` do documento chamará o validador do SpentSchema (arquivo models/index.ts) para verificar se o usuário existe na coleção users.

Arquivo: src/controllers/SpentController.ts

```
import { Request, Response } from "express";  
import { Spent } from "../models";  
  
class SpentController {  
  public async create(req: Request, res: Response): Promise<Response> {  
    const { user, description, value } = req.body;  
    try {  
      const document = new Spent({ user, description, value });  
      // ao salvar serão aplicadas as validações do esquema  
      const response = await document.save();  
      return res.json(response);  
    } catch (error: any) {  
      if (error && error.errors["description"]) {  
        return res.json({ message: error.errors["description"].message });  
      } else if (error && error.errors["value"]) {  
        return res.json({ message: error.errors["value"].message });  
      } else if (error && error.errors["user"]) {  
        return res.json({ message: error.errors["user"].message });  
      }  
      return res.json({ message: error });  
    }  
  }  
  
  public async list(req: Request, res: Response): Promise<Response> {  
    const { user } = req.body; // _id do usuário da chave estrangeira  
    try {  
      // o método select recebe os campos incluídos no resultado  
      const objects = await Spent.find({ user })  
        .select("description value")  
        .sort({ description: "asc" });  
      return res.json(objects);  
    } catch (error: any) {  
      return res.json({ message: error.message });  
    }  
  }  
}
```

```
public async delete(req: Request, res: Response): Promise<Response> {
  const { id: _id } = req.body; // _id do registro a ser excluído
  try {
    const object = await Spent.findByIdAndDelete(_id);
    if (object) {
      return res.json({ message: "Registro excluído com sucesso" });
    } else {
      return res.json({ message: "Registro inexistente" });
    }
  } catch (error: any) {
    return res.json({ message: error.message });
  }
}

public async update(req: Request, res: Response): Promise<Response> {
  const { id, user, description, value } = req.body;
  try {
    // busca o gasto existente na coleção antes de fazer o update
    const document = await Spent.findById(id);
    if (!document) {
      return res.json({ message: "Gasto inexistente" });
    }
    // atualiza os campos
    document.user = user;
    document.description = description;
    document.value = value;
    // ao salvar serão aplicadas as validações do esquema
    const response = await document.save();
    return res.json(response);
  } catch (error: any) {
    if (error && error.errors["description"]) {
      return res.json({ message: error.errors["description"].message });
    } else if (error && error.errors["value"]) {
      return res.json({ message: error.errors["value"].message });
    } else if (error && error.errors["user"]) {
      return res.json({ message: error.errors["user"].message });
    }
    return res.json({ message: error });
  }
}

export default new SpentController();
```

Utilize os códigos a seguir para definir as rotas definidas na pasta routes.

Arquivo: src/routes/user.ts

```
import { Router } from "express";
import controller from "../controllers/UserController";
```

```
const routes = Router();  
-  
routes.post('/', controller.create);  
routes.get('/', controller.list);  
routes.delete('/', controller.delete);  
routes.put('/', controller.update);  
  
export default routes;
```

Arquivo: src/routes/spent.ts

```
import { Router } from "express";  
import controller from "../controllers/SpentController";  
  
const routes = Router();  
  
routes.post('/', controller.create);  
routes.get('/', controller.list);  
routes.delete('/', controller.delete);  
routes.put('/', controller.update);  
  
export default routes;
```

A seguir tem-se o código para as rotas.

Arquivo: src/routes/index.ts

```
import { Router, Request, Response } from "express";  
import user from './user';  
import spent from './spent';  
  
const routes = Router();  
  
routes.use("/usuario", user);  
routes.use("/gasto", spent);  
  
//aceita qualquer método HTTP ou URL  
routes.use( (_,res:Response) => res.json({error:"Requisição desconhecida"}) );  
  
export default routes;
```

Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

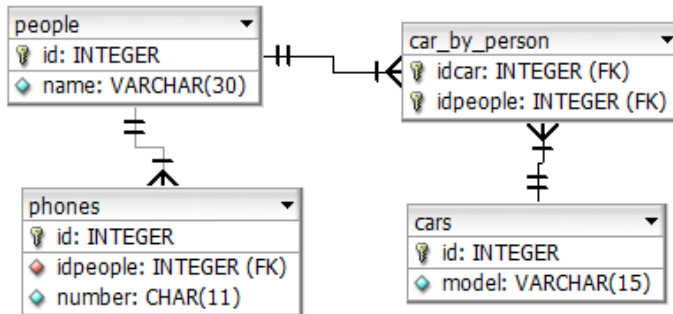
Exercício 1 - <https://youtu.be/TfBq40DWiKU>

Exercício 2 - <https://youtu.be/pwvwuyyB9Dk>

Exercício 1 – Criar uma aplicação servidora para persistir dados no MongoDB.

Considere os seguintes requisitos:

- O projeto deverá ter a estrutura mostrada ao lado;
- A aplicação deverá persistir os dados no BD bdexer01. Os dados de conexão com o MongoDB deverão estar no arquivo models/connection.ts;
- Os esquemas e modelos devem seguir o modelo de dados representado no diagrama a seguir. Os esquemas e modelos deverão estar no arquivo models/index.ts;



- Os seguintes campos não podem receber valores repetidos: model da coleção cars e name da coleção people;
- O campo number deverá ter exatamente 11 dígitos numéricos. Dica: use a propriedade match no esquema com a expressão regular `/^[0-9]{11}$/` .

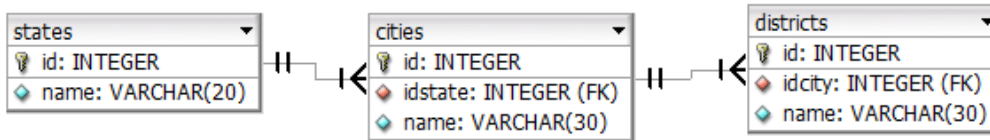
EXER01

```

> node_modules
└─ src
   └─ controllers
      TS CarByPersonController.ts
      TS CarController.ts
      TS PersonController.ts
      TS PhoneController.ts
   └─ models
      TS connection.ts
      TS index.ts
   └─ routes
      TS car.ts
      TS carbyperson.ts
      TS index.ts
      TS person.ts
      TS phone.ts
      TS index.ts
  .env
  .gitignore
  package-lock.json
  package.json
  tsconfig.json
    
```

Exercício 2 – Criar uma aplicação servidora para persistir dados no MongoDB. Considere os seguintes requisitos:

- O projeto deverá ter a estrutura mostrada ao lado;
- A aplicação deverá persistir os dados no BD bdexer02. Os dados de conexão com o MongoDB deverão estar no arquivo models/connection.ts;
- Os esquemas e modelos devem seguir o modelo de dados representado no diagrama a seguir. Os esquemas e modelos deverão estar no arquivo models/index.ts;



- Os relacionamentos de chave estrangeira deverão ser implementados usando subdocumentos, ou seja, city será subdocumento de state e district será subdocumento de city;
- Somente o campo name de State não aceita valores repetidos.

```

EXER02
├── node_modules
├── src
│   ├── controllers
│   │   ├── CityController.ts
│   │   ├── DistrictController.ts
│   │   └── StateController.ts
│   ├── models
│   │   ├── connection.ts
│   │   ├── index.ts
│   │   └── routes
│   │       ├── city.ts
│   │       ├── district.ts
│   │       ├── index.ts
│   │       ├── state.ts
│   │       └── index.ts
│   ├── .env
│   ├── .gitignore
│   ├── package-lock.json
│   ├── package.json
│   └── tsconfig.json
  
```