O que vamos aprender?

Hoje você continuará a aprender a melhorar a organização e divisão de responsabilidades nas suas aplicações Node.js e Express, utilizando um dos padrões arquiteturais mais famosos do mercado: o MSC !
Além disso, você verá uma aplicação em que o modelo acessa um banco MongoDB e entenderá em detalhes o que é a arquitetura de cliente-servidor.

Você será capaz de:

- Estruturar uma aplicação em camadas;
- Delegar responsabilidades específicas para cada parte do seu app;
- Melhorar manutenibilidade e reusabilidade do seu código.

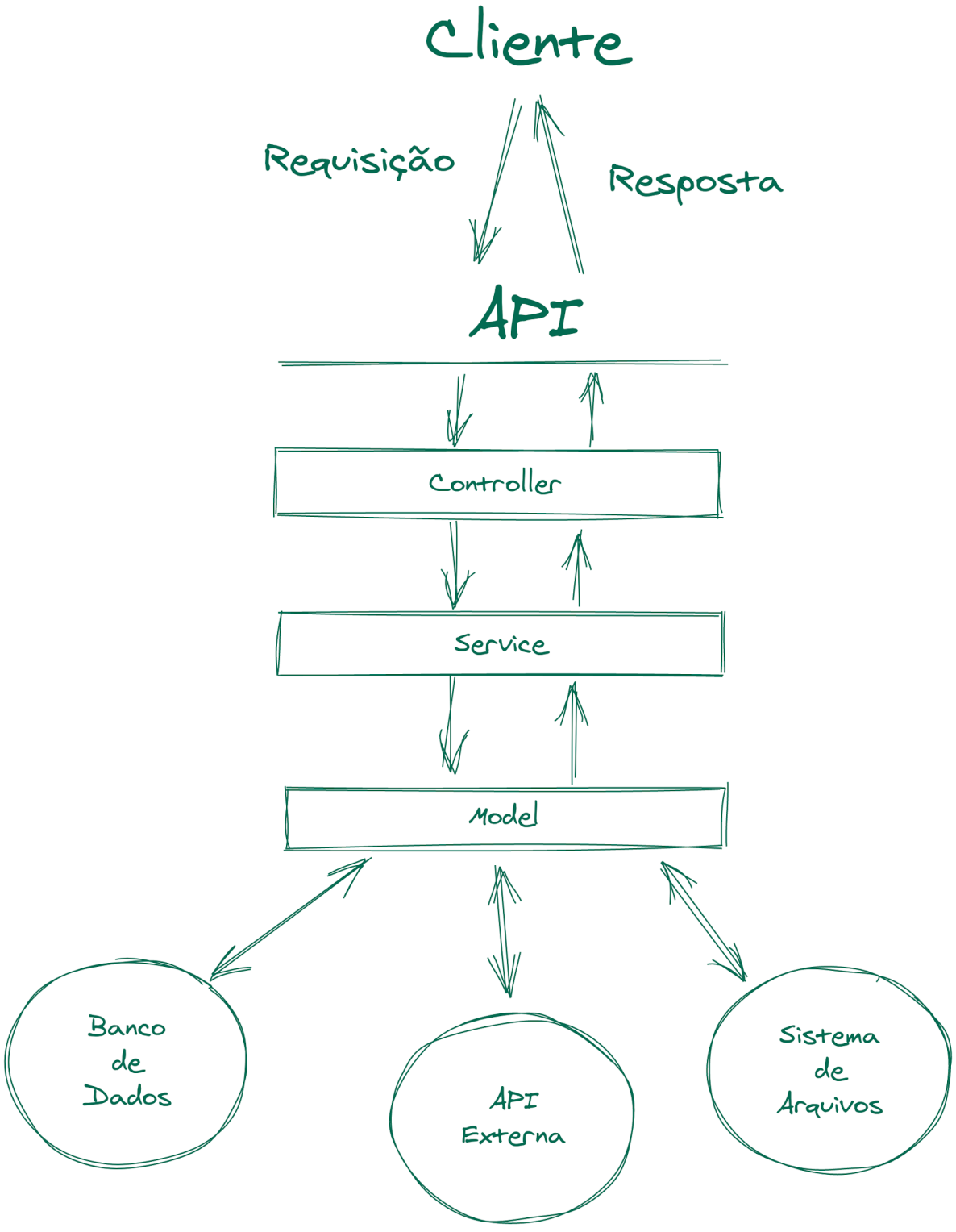
Por que isso é importante?

Ontem iniciamos o desenvolvimento de sua visão arquitetural. Para ampliar esse conhecimento é preciso que você conheça e entenda outras camadas de uma aplicação. Desse modo você terá mais insumos para decidir como organizar seu código, facilitando a manutenção e a adição de novas funcionalidades;

As camadas de Controller e Service

Hoje, você vai aprender duas camadas novas, que podem ser (e geralmente são) utilizadas de forma complementar à camada de Model. Vamos falar sobre as camadas de *Controllers* e de *Services* .
Essas duas camadas são, respectivamente, responsáveis por (1) receber e tratar os dados da requisição e (2) aplicar as regras de negócio da aplicação antes que qualquer comunicação com o banco seja realizada. Dessa forma, o Model precisa fazer menos coisas, o que quer dizer que temos uma arquitetura que delimita mais as responsabilidades de cada camada, de forma que, caso precisemos alterar uma parte do código, a quantidade de lugares em que precisaremos mexer é menor, visto que camada tem sua responsabilidade bem delimitada.

Para entender melhor como estão organizadas as camadas dessa arquitetura, observe o diagrama abaixo:



Organização das Camadas

Sempre que utilizarmos os termos "camadas abaixo" ou "camadas acima", lembre-se dessa ordem para se orientar.

Continue a leitura para uma explicação mais aprofundada sobre *controllers* e *services* , e bons estudos! 😄

A camada dos Controllers

Na verdade, desde o primeiro dia que estudou Express, você já vem usando o principal componente de sua camada de controllers: Os middlewares .
Isso porque a camada dos controllers é a primeira camada numa API. É nela onde os dados da requisição serão recebidos e tratados, pra depois serem passados para as próximas camadas.
O `controller` recebe as requisições e então consulta o `service` , enviando na resposta aquilo que o `service` retornar, que pode ser uma mensagem de erro, em caso de falha, ou as informações pedidas, em caso de sucesso.
Ao se comunicar com o `service` , o `controller` deve passar apenas as informações necessárias, sendo assim não é uma boa prática passar toda a `request` para o `service` , as informações devem ser extraídas e então apenas o que for necessário para determinada ação deve ser transferido.
Uma ótima analogia para o `controller` é que ele seria como um garçom em um restaurante. O garçom não sabe como preparar os pratos e nem como recepcionar as pessoas na porta. Ele apenas anota o pedido, sabe para que parte do restaurante levar o pedido e para qual mesa entregá-lo depois de pronto. Quando você monta seu software em uma camada só, é como se o garçom fizesse todas as funções dentro do seu restaurante (recepcionar, anotar os pedidos, preparar os pratos etc). É pedir pra dar confusão, não é?

A camada dos Services

Até agora, temos dito que regras de negócio ficam no modelo. E isso é verdade em outros padrões arquiteturais.
Mas é comum que, à medida que projetos vão crescendo, os modelos vão ficando cada vez maiores e mais complexos, pois vão acumulando cada vez mais regras de negócio.
Por isso, é comum vermos uma nova camada sendo adicionada em projetos que exigem uma lógica de negócio um pouco mais complexa e, principalmente, em APIs.

Essa camada é a Camada dos Services . Ela fica situada entre as camadas de controller e model e é responsável pela nossa lógica de negócio. O modelo, então, passa a ser responsável somente pelo acesso a dados.
Você pode ver isso de outra forma: para evitar que o modelo fique grande demais, ele é quebrado em duas outras camadas, cada uma com parte da responsabilidade.
Pense nessa camada como o chef da cozinha do nosso restaurante. Ele é quem sabe as receitas e delega as funções para os funcionários depois de receber o pedido do garçom.
Uma boa camada de serviço:

- Deve centralizar acesso a dados e funções externas. Exemplo: chamar um evento que dispara uma mensagem no Slack;
- Deve abstrair lógica de negócio complexa do seu modelo;
- Não deve ter nenhum tipo de informação sobre o acesso a camada de dados. Exemplo: não ter nenhuma query SQL;
- Não deve receber nada relacionado ao HTTP, seja o `request` ou o `response` . O controller deve mandar apenas o necessário para o `service` .

Praticando

Antes de começar a colocar a mão na massa, assista o seguinte vídeo, que vai te dar uma ideia do que vamos fazer:

VIDEO: [2] Refatorando com camada de service - 09:28

Para colocar em prática os conceitos de `controller` e `service` vamos adicionar essas camadas à aplicação de autores que você viu ontem no conteúdo.
Crie os arquivos abaixo numa pasta chamada `hello-msc` :

Copiar

```
// hello-msc/package.json

{
  "name": "hello-msc",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
```

```
  "keywords": [],
  "author": "Tryber",
  "license": "GPL-3.0",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "express-rescue": "^1.1.31",
    "joi": "^17.4.0",
    "mongodb": "^3.6.4",
    "nodemon": "^2.0.7"
  }
}
```

Copiar

```
// hello-msc/index.js

const express = require('express');
const bodyParser = require('body-parser');

const Author = require('./models/Author');

const app = express();

app.use(bodyParser.json());

app.get('/authors', async (_req, res) => {
  const authors = await Author.getAll();

  res.status(200).json(authors);
});

app.get('/authors/:id', async (req, res) => {
  const { id } = req.params;

  const author = await Author.findById(id);

  if (!author) return res.status(404).json({ message: 'Not found'
});

  res.status(200).json(author);
});

app.post('/authors', async (req, res) => {
```

```
  const { first_name, middle_name, last_name } = req.body;

  if (!Author.isValid(first_name, middle_name, last_name)) {
    return res.status(400).json({ message: 'Dados inválidos' });
  }

  await Author.create(first_name, middle_name, last_name);

  res.status(201).json({ message: 'Autor criado com sucesso! ' });
});

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Ouvindo a porta ${PORT}`);
});
```

Crie uma pasta `models` e dentro dela o arquivo os seguintes arquivos:

Copiar

```
// hello-msc/models/connection.js

const { MongoClient } = require('mongodb');

const OPTIONS = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}

const MONGO_DB_URL = 'mongodb://127.0.0.1:27017';

let db = null;

const connection = () => {
  return db
    ? Promise.resolve(db)
    : MongoClient.connect(MONGO_DB_URL, OPTIONS)
    .then((conn) => {
      db = conn.db('model_example');
      return db;
    })
};

module.exports = connection;
```

Copiar

```
// hello-msc/models/Author.js

const connection = require('./connection');
const { ObjectId } = require('mongodb');

// Cria uma string com o nome completo do autor

const getNewAuthor = (authorData) => {
const { id, firstName, middleName, lastName } = authorData;

const fullName = [firstName, middleName, lastName]
  .filter((name) => name)
  .join(' ');

return {
  id,
  firstName,
  middleName,
  lastName,
  name: fullName,
 };
};

// Busca todos os autores do banco.

const getAll = async () => {
  return connection()
    .then((db) => db.collection('authors').find().toArray())
    .then((authors) =>
      authors.map(({ _id, firstName, middleName, lastName }) =>
        getNewAuthor({
          id: _id,
          firstName,
          middleName,
          lastName,
        })
      )
  );
}

/*
Busca um autor específico, a partir do seu ID
```

```
@param {String} id ID do autor a ser recuperado
*/
const findById = async (id) => {
  if (!ObjectId.isValid(id)) {
    return null;
  }

  const authorData = await connection()
    .then((db) => db.collection('authors').findOne(new
ObjectId(id)));

  if (!authorData) return null;

  const { firstName, middleName, lastName } = authorData;

  return getNewAuthor({ id, firstName, middleName, lastName });
};

const isNonEmptyString = (value) => {
  if (!value) return false;

  return typeof value === 'string';
};

const isValid = (firstName, middleName, lastName) => {
  if (middleName && typeof middleName !== 'string') return false;

  return isNonEmptyString(firstName) && isNonEmptyString(lastName);
};

const create = async (firstName, middleName, lastName) =>
  connection()
    .then((db) => db.collection('authors').insertOne({ firstName,
middleName, lastName }))
    .then(result => getNewAuthor({ id: result.insertedId, firstName,
middleName, lastName }));

module.exports = {
  getAll,
  findById,
  isValid,
  create,
};
```

Por último, execute `npm install` dentro da pasta `hello-msc` para instalar as dependências.
Agora temos uma aplicação na qual as regras de negócio dizem respeito todas ao formato dos campos na entidade Author. Por exemplo: "Nome de ser uma string e não pode ser vazio". No entanto, para ilustrar melhor o tipo de regra de negócio que costuma ser tratada pelo service, vamos introduzir uma nova regra: "Um autor com mesmo nome completo não pode ser cadastrado duas vezes."
Essa é uma regra mais complexa, que exige mais lógica do que um simples `if` para ser validada. Sendo assim, é o tipo de regra que se encaixa perfeitamente no service que vamos criar agora!
Crie a pasta `services` e dentro dela o arquivo `Author.js` , adicione o código abaixo no arquivo:

Copiar

```
const Author = require('../models/Author');

const getAll = async () => Author.getAll();

const findById = async (id) => Author.findById(id);

const create = async (firstName, middleName, lastName) =>
  Author.create(firstName, middleName, lastName);

module.exports = {
  getAll,
  findById,
  create,
};
```

Até agora, nosso service realiza todas as três operações que o model também realiza, sem nenhuma lógica adicional. Repare como, em cada função, nós apenas retornamos uma chamada para aquela mesma função dentro do model. Chegou a hora de mudar isso!
Primeiro, vamos precisar de uma função no nosso model que nos permita buscar autores pelos três nomes. Isso vai permitir a implementação da regra "Um autor com mesmo nome completo não pode ser cadastrado duas vezes."
Altere o arquivo `hello-msc/models/Author.js` da seguinte maneira:

Copiar

```
// hello-msc/models/Author.js

/* ... */
```

```
// const create = async (firstName, middleName, lastName) =>
//   connection()
//     .then((db) => db.collection('authors').insertOne({ firstName, middleName, lastName }))
//     .then(result => getNewAuthor({ id: result.insertedId, firstName, middleName, lastName }));

const findByName = async (firstName, middleName, lastName) => {
  // Determinamos se devemos buscar com ou sem o nome do meio
  const query = middleName
    ? { firstName, middleName, lastName }
    : { firstName, lastName };

  // Executamos a consulta e retornamos o resultado
  const author = await connection()
    .then((db) => db.collection('authors').findOne(query));

  // Caso nenhum author seja encontrado, devolvemos null
  if (!author) return null;

  // Caso contrário, retornamos o author encontrado
  return getNewAuthor(author);
};

// module.exports = {
//   getAll,
//   findById,
//   isValid,
//   create,
    findByName
// };
```

Com essa função pronta, precisamos modificar o service para que ele a utilize e aplique nossa regra de negócio. Modifique o arquivo `services/Author.js` da seguinte forma:

Copiar

```
// const Author = require('../models/Author');

// const getAll = async () => Author.getAll();

// const findById = async (id) => Author.findById(id);
```

```
const create = async (firstName, middleName, lastName) => {
  // Buscamos um autor com o mesmo nome completo que desejamos criar
  const existingAuthor = await Author.findByName(firstName,
middleName, lastName);

  // Caso esse autor já exista, retornamos um objeto de erro
informando
  // que não é possível criar o autor pois ele já existe
  if (existingAuthor) {
    return {
      error: {
        code: 'alreadyExists',
        message: 'Um autor já existe com esse nome completo',
      },
    };
  }

  // Caso o autor não exista e, portanto, possa ser criado
  // chamamos o model e retornamos o resultado
  return Author.create(firstName, middleName, lastName);
};

// module.exports = {
//   getAll,
//   findById,
//   create,
// };
```

Agora, nosso service implementa a regra de negócio mais complexa que temos. Isso até poderia acontecer no model mas, com o tempo, o model começaria a acumular várias funções, indo desde validar dados e regras de negócio até montar queries complexas e comunicar com o banco. Deixando as duas coisas em camadas separadas é como se tanto model quanto service tivessem "espaço pra crescer" sem ficarem "apertados".
Existe ainda uma outra regra que é responsabilidade do service e que, até o momento, tem ficado no middleware: identificar e gerar erros.
Mas pera lá ... Gerar erros ? A ideia não é evitá-los?
Bom, de um certo ponto de vista, sim. 😅
Devemos codificar nossas aplicações de forma que erros não previstos sejam evitados ou contornados. No entanto, existem erros que derivam de regras de negócio que não foram atendidas. Vamos chamar esses erros de Erros de domínio . Numa aplicação em camadas, eles servem principalmente para que camadas inferiores possam informar camadas

superiores sobre erros ou falhas que, por sua vez, devem ser retornadas a quem fez a chamada.
No nosso caso, temos um exemplo de erro de domínio, com o código `alreadyExists` . O service retorna esse objeto de erro para que o controller saiba que ocorreu um erro e que o autor não foi criado com sucesso. e que permite que o controller saiba que o status da requisição não deve ser 200, por exemplo. Outro tipo de situação conhecida que deve ser notificada pelo service é quando um item buscado não é encontrado.
Note, na linha 23 do `index.js` , que quem faz esse tratamento até agora é o middleware. Vamos mudar isso!
Altere o arquivo `services/Author.js`

Copiar

```
// hello-msc/services/Author.js

// const Author = require('../models/Author');

// const getAll = async () => Author.getAll();

const findById = async (id) => {
  // Solicitamos que o model realize a busca no banco
  const author = await Author.findById(id);

  // Caso nenhum autor seja encontrado, retornamos um objeto de
erro.
  if (!author) {
    return {
      error: {
        code: 'notFound',
        message: `Não foi possível encontrar um autor com o id
${id}`,
      },
    };
  }

  // Caso haja um autor com o ID informado, retornamos esse autor
  return author;
};

// const create = async (firstName, middleName, lastName) => {
//   // Buscamos um autor com o mesmo nome completo que desejamos
criar
```

```
//   const existingAuthor = await Author.findByName(firstName,
middleName, lastName);

//   // Caso esse autor já exista, retornamos um objeto de erro
informando
//   // que não é possível criar o autor pois ele já existe
//   if (existingAuthor) {
//     return {
//       error: {
//         code: 'alreadyExists',
//         message: 'Um autor já existe com esse nome completo',
//       },
//     };
//   }

//   // Caso o autor não exista e, portanto, possa ser criado
//   // chamamos o model e retornamos o resultado
//   return Author.create(firstName, middleName, lastName);
// };

// module.exports = {
//   getAll,
//   findById,
//   create,
// };
```

Agora sim, nosso service está comunicando ao controller toda vez que algum erro de domínio acontece. A seguir, vamos ver como esse erro é recebido e tratado pelo controller.

Crie a pasta `controllers` e, dentro dela, o arquivo `Author.js` . Nesse arquivo, vamos implementar lógica para realizar todas as operações que nossa aplicação realiza até agora, começando por buscar todos os autores:

Copiar

```
// hello-msc/controllers/Author.js

const rescue = require('express-rescue');
const service = require('../services/Author');

const getAll = rescue(async (req, res) => {
  const authors = await service.getAll();

  res.status(200).json(authors);
```

```
});

module.exports = {
  getAll,
};
```

Repare que o código aqui é precisamente o mesmo que passamos ao registrar o endpoint `GET /authors` no `index.js` , e essa é a grande jogada! A camada de `controllers` é responsável por receber e tratar as requests, e, no express, é composta majoritariamente de middlewares. Sendo assim, para construir nosso controller, só precisamos trazer os middlewares do `index.js` para o controller, alterando-os para que utilizem o service ao invés do model. Parece bastante coisa? Não se preocupe, vamos fazer middleware a middleware.
Já trouxemos o endpoint `GET /authors` , então vamos para o próximo: `GET /authors/:id` :

Copiar

```
// hello-msc/controllers/Author.js

// const rescue = require('express-rescue');
// const service = require('../services/Author');

// const getAll = rescue(async (req, res) => {
//   const authors = await service.getAll();

//   res.status(200).json(authors);
// });

const findById = rescue(async (req, res, next) => {
  // Extraímos o id da request
  const { id } = req.params;

  // Pedimos para o service buscar o autor
  const author = await service.findById(id);

  // Caso o service retorne um erro, interrompemos o processamento
  // e inicializamos o fluxo de erro
  if (author.error) return next(author.error);

  // Caso não haja nenhum erro, retornamos o author encontrado
  res.status(200).json(author);
});
```

```
// module.exports = {
//   getAll,
    findById,
// };
```

Repare que o controller verifica se existe um erro e, se existir, chama `next(author.error)` . Isso faz com que esse objeto de erro vá parar no próximo middleware de erro registrado. Isso quer dizer que podemos utilizar um middleware de erro centralizado também para nossos erros de domínio. Vamos ver como fazer isso logo mais. Por hora, vamos trazer a terceira e última função: a criação de um novo autor. Aqui veremos mais uma funcionalidade do controller em ação: a validação dos dados da request.
Você pode estar se perguntando "Ué, mas por que não validar no model?". O fato é que a validação no model pode trazer algumas dificuldades à medida que nossa aplicação escala, por exemplo:

- Nem sempre queremos validar os mesmos campos (uma request de edição pode pedir dados diferentes de uma request de criação, por exemplo);
- Estamos delegando mais uma responsabilidade para o model: além de se comunicar com o banco, ele também faz validação de requests
- Ao validar no model, estamos validando os dados no final da request, ou seja, na saída . Ao validar no controller, estamos validando esses dados na entrada , garantindo que não vamos realizar nenhum processamento desnecessário utilizando dados que não são válidos, e que os dados vão trafegar *limpinhos* por todas as camadas da aplicação.

Dito isso, vamos usar uma biblioteca que vai nos ajudar muito: o Joi. Dá uma olhada:
Primeiro, vamos instalar o `joi` . Execute no terminal:

Copiar
```
npm i joi
```

Agora, vamos adicioná-lo ao controller:

Copiar
```
// hello-mvc/controllers/Author.js

const Joi = require('joi');

/* ... */
```

```
// const findById = rescue(async (req, res, next) => { /* ... */ }

const create = rescue(async (req, res, next) => {
  // Utilizamos o Joi para descrever o objeto que esperamos
  // receber na requisição. Para isso, chamamos Joi.object()
  // passando um objeto com os campos da requisição e suas
descrições
  const { error } = Joi.object({
    // Deve ser uma string (.string()) não vazia (.not().empty()) e
é obrigatório (.required())
    firstName: Joi.string().not().empty().required(),
    // Não é obrigatório mas, caso seja informado, deve ser uma
string não vazia
    middleName: Joi.string().not().empty(),
    // Deve ser uma string não vazia e é obrigatório
    lastName: Joi.string().not().empty().required(),
  })
    // Por fim, pedimos que o Joi verifique se o corpo da requisição
se adequa a essas regras
    .validate(req.body);

  // Caso exista algum problema com a validação, iniciamos o fluxo
de erro e interrompemos o middleware.
  if (error) {
    return next(error);
  }

  // Caso não haja erro de validação, prosseguimos com a criação do
usuário
  const { firstName, middleName, lastName } = req.body;

  const newAuthor = await service.create(firstName, middleName,
lastName);

  // Caso haja erro na criação do autor, iniciamos o fluxo de erro
  if (newAuthor.error) return next(newAuthor.error);

  // Caso esteja tudo certo, retornamos o status 201 Created, junto
com as informações
  // do novo autor
  return res.status(201).json(newAuthor);
});
```

```
// module.exports = {
//   getAll,
//   findById,
    create,
// };
```

E, agora sim, nosso controller está pronto! Para um resumo de como a camada de controller se comunica com as outras e de como realizamos essa migração, assista ao vídeo abaixo:

VIDEO: [3] Refatorando com controllers - 03:57

Agora que nosso controller está pronto, só falta "plugá-lo" no nosso `app` do express, no arquivo `index.js` . Bora lá?
Altere o arquivo `index.js`

Copiar

```
// hello-msc/index.js

// const express = require('express');
// const bodyParser = require('body-parser');

const Author = require('./controllers/Author');

// const app = express();

// app.use(bodyParser.json());

app.get('/authors', Author.getAll);
app.get('/authors/:id', Author.findById);
app.post('/authors', Author.create);

// const PORT = process.env.PORT || 3000;

// app.listen(PORT, () => {
//   console.log(`Ouvindo a porta ${PORT}`);
// });
```

A essa altura, você já pode executar a aplicação e ver que tudo funciona! *Uhuuu* 🥳 !
No entanto, ainda falta um detalhe importante: O tratamento de erros! 😬
Afinal, nem tudo são flores, certo? 🌹😖

No nosso controller, existem alguns momentos em que interrompemos o fluxo comum do middleware, e iniciamos o fluxo de erro. Esse fluxo de erro é também responsabilidade da camada de controller, que deve converter o erro em um formato padronizado e enviá-lo, junto com o status code adequado, para o *client* que realizou a requisição.
Para implementar esse comportamento, vamos criar um middleware de erro . Para esse exemplo, vamos criá-lo numa pasta `middlewares` , mas é comum que o middleware de erro seja criado como um `ErrorController` , dentro da pasta `controllers` . Não há nada de errado com essa abordagem, e as duas são formas válidas de implementar.
Crie a pasta `middlewares` e, dentro dela, o arquivo `error.js` :

Copiar

```
// hello-msc/middlewares/error.js
module.exports = (err, req, res, _next) => {
  // Qualquer erro será recebido sempre por esse middleware, então a primeira coisa que fazemos
  // é identificar qual o tipo do erro.

  // Se for um erro do Joi, sabemos que trata-se de um erro de validação
  if (err.isJoi) {
    // Logo, respondemos com o status 400 Bad Request
    return res.status(400)
      // E com a mensagem gerada pelo Joi
      .json({ error: { message: err.details[0].message } });
  }

  // Caso não seja um erro do Joi, pode ser um erro de domínio ou um erro inesperado.
  // Construímos, então, um mapa que conecta um erro de domínio a um status HTTP.
  const statusByErrorCode = {
    notFound: 404, // Erros do tipo `notFound` retornam status 404 Not Found
    alreadyExists: 409, // Erros do tipo `alreadyExists` retornam status 409 Conflict
    // Podemos adicionar quantos códigos novos desejarmos
  };

  // Buscamos o status adequado para o erro que estamos tratando.
  // Caso não haja um status para esse código, assumimos que é
```

```
  // um erro desconhecido e utilizamos o status 500 Internal Server
Error
  const status = statusByErrorCode[err.code] || 500;

  // Por último, retornamos o status e a mensagem de erro para o
client
  res.status(status).json({ error: { message: err.message } });
};
```

Agora, é só "plugar" nosso middleware de erro na aplicação do express e pronto!
Volte no `index.js` e faça as seguintes adições.

Copiar

```
// hello-msc/index.js

// const express = require('express');
// const bodyParser = require('body-parser');

// const Author = require('./controllers/Author');
const errorMiddleware = require('./middlewares/error');

// const app = express();

// app.use(bodyParser.json());

// app.get('/authors', Author.getAll);
// app.get('/authors/:id', Author.findById);
// app.post('/authors', Author.create);

app.use(errorMiddleware);

// const PORT = process.env.PORT || 3000;

// app.listen(PORT, () => {
//   console.log(`Ouvindo a porta ${PORT}`);
// });
```

E, agora sim, nossa aplicação está pronta! Utilizando as três camadas: Model, Service e Controllers.
Dessa forma, fica muito mais fácil realizar alterações nessa aplicação, principalmente se ela for crescer, como a maioria das aplicações acaba crescendo 😄.

Vamos praticar

Ontem, criamos um CRUD para a entidade `Books` . Vamos refatorar o código da aula de ontem aplicando a arquitetura MSC. Para isso:

1. Crie um arquivo `services/Book.js` e aplique as regras de negócio definidos no modelo `Book` para dentro do service. (lembre-se de remover de models/Book.js o que não vai ser mais utilizado na camada de modelo).
2. Crie um arquivo `controllers/BooksController.js` e transfira os middlewares relacionados ao nosso CRUD de livros para esse controller, aproveite também para criar o middleware de erro que foi ensinado no conteúdo de hoje.
3.

Validações

Assista os vídeos abaixo para entender como aplicar validações para a camada de serviço ou utilizando a camada de middlewares.

VIDEO: 27.2 Validações Parte 1 - 03:05
VIDEO: 27.2 Validações Parte 2 - 08:33
VIDEO: 27.2 Validações Parte 3 - 04:55
VIDEO: 27.2 Validações Parte 4 - 13:43
VIDEO: 27.2 Validações Parte 5 - 10:21

Boas Práticas em Arquitetura de Software

Indiferente de qual padrão arquitetural você vai usar, existem algumas boas práticas que você deve sempre manter em mente, independente do padrão a ser seguido.

Pense antes de escrever código!

A primeira coisa é você entender qual é o problema que será resolvido e, a partir daí, começar a pensar em uma solução em nível de arquitetura. Imagine o seguinte cenário:
"Quero criar uma aplicação que mostra todas as fotos que as pessoas tiraram com base na localização. As versões mobile native e web serão parecidas, mas apenas a mobile poderá tirar fotos." - Cliente, Seu Beleza! Pensando que vamos ter múltiplos clientes com funcionalidades semelhantes, faz sentido termos uma API, certo?

Pensando mais a fundo na arquitetura da API, é de se imaginar que vamos ter que subir as fotos em algum serviço de hospedagem (em vez de armazená-las nós mesmos), e vamos salvar no banco apenas a URL gerada após o *upload* . Nesse caso, faz bastante sentido termos uma camada de serviço que vai orquestrar essa parte de hospedagem.
Claro que, na prática, não é tão simples assim. 😬 Mas isso é só um exemplo de como você deve pensar em qual arquitetura faz mais sentido para o problema que está tentando resolver para, só depois, começarmos a codificar!

Pense em Componentes

Isso é bem parecido com o que nós fazemos com React! Você se lembra do princípio por trás dos componentes?
A intenção é que nossas aplicações sejam construídas com pequenos pedacinhos de código sem dependências entre si. A mesma coisa se aplica numa API também!
Dentro das suas camadas, mantenha cada controller, cada model e cada serviço pequeno e o mais desacoplado possível das outras partes. Faça com que eles se comuniquem somente através de interfaces muito bem definidas. Não deixe que um componente acesse diretamente o que está dentro de outro. Isso vai facilitar muito na hora de dar manutenção, reutilizar e testar seu código.

Mantenha suas pastas organizadas

Existem algumas maneiras de organizar as pastas em um projeto, mas vamos citar duas: por domínio/correlação e por papel técnico .

- Por domínio/correlação , nós mantemos todos os arquivos que têm relação com um `Author` , por exemplo, na mesma pasta, independente da responsabilidade de cada arquivo:

Copiar

```
└── author
│   ├── authorController.js
│   ├── authorService.js
│   └── authorModel.js
└── book
    └── bookController.js
    └── bookService.js
    └── bookModel.js
```

- Por papel técnico é como temos exemplificado até agora (não que seja necessariamente melhor). Todos os controllers em uma pasta, todos os services em outra e por aí vai:

Copiar

```
└── controllers
|   ├── authorController.js
|   └── bookController.js
└── services
|   ├── authorService.js
|   └── bookService.js
└── models
|   ├── authorModel.js
|   └── bookModel.js
```

Muitas vezes, você vai utilizar um framework em que essa decisão já foi tomada. Nesse caso, siga com o padrão.

Mantenha o Express o mais longe possível .

O mais longe possível quer dizer que devemos criar fronteiras bem definidas entre o Express e o "resto da sua aplicação".
Isso significa manter os objetos `req` e `res` dentro do escopo do controller e nunca passá-los inteiros para as partes do app que cuidam da lógica de negócio.
Tomando essa precaução simples, você vai evitar ficar criando mocks para esses objetos quando for escrever testes unitários, por exemplo.
Se o seu modelo precisa apenas dos campos `user` e `password` para fazer o login de alguém, para que passar para ele o objeto `req` e mandar todos os headers que vieram na requisição?
Observe este exemplo:

Copiar

```
const userController = async (req, res) => {
  try {
    // ruim 😟
    await UserService.create(req);

    // bom! 😊
    const { email, password } = req.body;
    await UserService.create(email, password);

    res.send({ message: 'Tudo certo!' });
  } catch (e) {
```

```
    res.status(500).send({ message: 'Algo deu errado' });
  }
};
```

Usando essas fronteiras como exemplo, nada além da camada de controle deveria saber que o Express existe .

Mantenha sua configuração separada (e segura)

Nos exemplos de aula, vimos que as informações sensíveis, como credenciais de acesso ao banco de dados, estavam todas expostas no nosso código. 😱
Só fizemos isso para fins didáticos. Uma ótima prática é usar variáveis de ambiente para controlar coisas relacionadas à configuração geral da sua aplicação (em qual banco se conectar, para qual URL apontar etc.).
Variáveis de ambiente são variáveis que podem ser definidas no sistema operacional e, portanto, podem ser diferentes para cada ambiente (computador). Por exemplo, no seu computador local, a *URL* do banco é uma, mas, no servidor da aplicação, a *URL* do banco é outra. Para fazer isso funcionar, você pode utilizar uma variável de ambiente chamada `DB_URL` e utilizar valores diferentes para ela no servidor e na sua máquina local.
OK, e como eu acesso essa variável no código?
O ambiente Node tem uma variável global que se chama `process` ; dentro dela temos um objeto `env` que armazena os valores de todas as variáveis de ambiente definidas no sistema operacional.
Podemos setar variáveis de ambiente pelo terminal:

Copiar
```
DB_URL="mongodb://localhost:27017" node index.js
```

Copiar
```
// index.js

console.log(process.env.DB_URL) // mongodb://localhost:27017
```

No entanto, uma forma melhor e mais fácil, quando temos muitas variáveis, é criar um arquivo `.env` na raiz do projeto e usar a biblioteca `dotenv` , que basicamente pega o conteúdo desse arquivo e o deixa acessível via `process.env` .

Copiar
```
npm install dotenv
```

Copiar
```
# .env
```

```
PORT=3000
DB_URL=mongodb://localhost:27017
DB_NAME=model_example
```

Copiar

```
// index.js

require('dotenv').config();
// ...

const PORT = process.env.PORT;
app.listen(PORT, () => console.log(`Server listening on port
${PORT}`));
// Server listening on port 3000
```

Copiar

```
// models/connection.js

const { MongoClient } = require('mongodb');

const OPTIONS = {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}

const MONGO_DB_URL = 'mongodb://127.0.0.1:27017';

let db = null;

const connection = () => {
    return db
    ? Promise.resolve(db)
    : MongoClient.connect(MONGO_DB_URL, OPTIONS)
    .then((conn) => {
    db = conn.db('model_example');
    return db;
    })
};

module.exports = connection;
```

Por último, não se esqueça de colocar o `.env` no `.gitignore` , pois não vamos querer versionar esse arquivo.

Dessa forma, as configurações da sua aplicação podem mudar de acordo com o ambiente, ou até mesmo com o tempo ficam separadas do código, que é o mesmo em qualquer ambiente. Além disso, você não estará mais adicionando dados sensíveis ao seu repositório, visto que o arquivo `.env` contém esses valores e não será versionado.

Exercícios

Hora de pôr a mão na massa!
back-end

Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma `branch`, com arquivos e `commits` específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

```
$ git checkout main
```

6. `$ git pull`
7. A partir da main, crie uma `branch` com o nome `exercicios/26.2` *(bloco 26, dia 2)*
8. Copiar
9. `$ git checkout -b exercicios/26.2`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

```
$ mkdir back-end
```

12. `$ cd back-end`
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

```
$ mkdir
bloco-26-nodejs-camada-de-servico-e-arquitetura-rest-e-restful
```

15. `$ cd bloco-26-nodejs-camada-de-servico-e-arquitetura-rest-e-restful`
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-2-arquitetura-de-software-camada-de-controller-e-service
```

18. `$ cd dia-2-arquitetura-de-software-camada-de-controller-e-service`
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório `~/trybe-exercicios/back-end/block-26-nodejs-camada-de-servico-e-arquitetura-rest-e-restful/dia-2-arquitetura-de-software-camada-de-controller-e-service`. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/26.2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
```

```
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:

24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`
26. Você pode visualizar o log de todos os commits já feitos naquela branch com `git log`.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.2)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e coloquei pra quando clicar aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300
```

28. `    Resolvendo o exercício 1 usando eventListener`
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch `exercicios/26.2` não vai existir no repositório remoto, então precisamos configurar o `remote` utilizando a opção `--set-upstream` (ou `-u`, que é a forma abreviada).

30. Copiar
31. `$ git push -u origin exercicios/26.2`
32. Após realizar o passo 9, podemos abrir a `Pull Request` a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com `-u` anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch `exercicios/26.2`, e disponíveis no repositório remoto do `GitHub`. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👉👉

Agora, a prática

Você vai desenvolver uma aplicação de busca de CEP, chamada `cep-lookup` . A aplicação fornecerá um serviço de busca e cadastro de CEPs em um banco de dados. Como bônus, ao invés de cadastrar novos CEPs manualmente, a aplicação consultará uma API externa para obter os dados do CEP desejado.
Utilize o banco de dados MySQL. Execute a seguinte query para criar o banco e a tabela:

Copiar

```
CREATE DATABASE IF NOT EXISTS cep_lookup;

USE cep_lookup;
```

```
CREATE TABLE IF NOT EXISTS ceps (
  cep VARCHAR(8) NOT NULL PRIMARY KEY,
  logradouro VARCHAR(50) NOT NULL,
  bairro VARCHAR(20) NOT NULL,
  localidade VARCHAR(20) NOT NULL,
  uf VARCHAR(2) NOT NULL
);
```

Bons estudos!

1. Crie uma nova API utilizando o express
2. A aplicação deve ser um pacote Node.js
3. Dê ao pacote o nome de `cep-lookup`
4. Utilize o express para gerenciar os endpoints da sua aplicação
5. A aplicação deve ter a rota `GET /ping` , que retorna o status `200 OK` e o seguinte JSON:

Copiar
```
{ "message": "pong!" }
```

5. A aplicação deve escutar na porta 3000
6. Deve ser possível iniciar a aplicação através do comando `npm start` .
7. Crie o endpoint `GET /cep/:cep`
8. O endpoint deve receber, no parâmetro `:cep` , um número de CEP válido.
9. O CEP precisa conter 8 dígitos numéricos e pode ou não possui traço.
    - Dica Utilize o regex `\d{5}-?\d{3}` para validar o CEP.
10. Caso o CEP seja inválido, retorne o status `400 Bad Request` e o seguinte JSON:

Copiar
```
 { "error": { "code": "invalidData", "message": "CEP inválido" } }
```

4. Caso o CEP seja válido, realize uma busca no banco de dados.
    1. Caso o CEP não exista no banco de dados, retorne o status `404 Not Found` e o seguinte JSON:

Copiar
```
{ "error": { "code": "notFound", "message": "CEP não encontrado" } }
```

Copiar

2. Caso o CEP exista, retorne o status `200 OK` e os dados do CEP no seguinte formato:

Copiar

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

3. Crie o endpoint `POST /cep`
4. O endpoint deve receber a seguinte estrutura no corpo da requisição:

Copiar

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

2. Todos os campos são obrigatórios
3. Utilize o Joi para realizar a validação. Em caso de erro, retorne o status `400 Bad Request` , com o seguinte JSON:

Copiar

```
{ "error": { "code": "invalidData", "message": "<mensagem do Joi>" }
}
```

4. O CEP deve ser composto por 9 dígitos com traço.
    - Dica : Utilize o seguinte regex para validar o CEP: `\d{5}-\d{3}`
5. Se o CEP já existir, retorne o status `409 Conflict` com o seguinte JSON:

Copiar

```
{
  "error": { "code": "alreadyExists", "message": "CEP já existente"
}
}
```

6. Se o CEP ainda não existir, armazene-o no banco de dados e retorne o status `201 Created` com os dados do novo CEP no seguinte formato:

Copiar
```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

Bônus

1. Utilize uma API externa para buscar CEPs que não existem no banco de dados
2. Quando um CEP não existir no banco de dados, utilize a API https://viacep.com.br/ws/[numero-do-cep]/json/ para obter suas informações.
3. Caso o CEP não exista na API externa, você receberá o JSON `{ "erro": true }` . Nesse caso, retorne status `404 Not Found` com o seguinte JSON:

Copiar
```
{ "error": { "code": "notFound", "message": "CEP não encontrado" } }
```

3. Caso o CEP exista na API externa, armazene-o no banco e devolva seus dados no seguinte formato:

Copiar
```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

Dica : Na arquitetura MSC, os models são responsáveis por toda a comunicação externa de uma aplicação, o que inclui APIs externas. Logo, você precisará de um model para acessar a API.

Conteúdos

Praticando

Vamos praticar

Ontem, criamos um CRUD para a entidade `Books` . Vamos refatorar o código da aula de ontem aplicando a arquitetura MSC. Para isso:

Exercício 1

Crie um arquivo `services/Book.js` e aplique as regras de negócio definidos no modelo `Book` para dentro do service. (lembre-se de remover de models/Book.js o que não vai ser mais utilizado na camada de modelo).

Solução

Copiar

```
// models/Book.js

const { ObjectId } = require('mongodb');
const connection = require('./connection');

const renameId = ({ _id, ...document }) => ({ id: _id, ...document
});

const getAll = () => connection()
    .then((db) => db.collection('books').find({}).toArray())
    .then((results) => results.map(renameId));

const getByAuthorId = (authorId) => connection()
    .then((db) => db.collection('books').find({ authorId
}).toArray())
    .then((result) => (result ? renameId(result) : result));

const findById = async (id) => {
  if (!ObjectId.isValid(id)) return null;
```

```
  const book = await connection()
    .then((db) => db.collection('books').findOne(new ObjectId(id)));

  if (!book) return null;

  return book;
};

const create = (title, authorId) => connection()
    .then((db) => db.collection('books').insertOne({ title, authorId
}))
    .then((result) => ({ id: result.insertedId, title, authorId }));

module.exports = {
  getAll,
  getByAuthorId,
  findById,
  create,
};
```

Copiar

```
// services/Book.js

const Author = require('../models/Author');
const Book = require('../models/Book');

const getAll = async () => Book.getAll();

const findById = async (id) => {
  const book = await Book.findById(id);

  if (!book) {
    return {
      error: {
        code: 'notFound',
        message: 'Livro não encontrado',
      },
    };
  }

  return book;
};
```

```
const create = async (title, authorId) => {
  const author = await Author.findById(authorId);

  if (!author) {
    return {
      error: {
        code: 'notFound',
        message: 'Autor não encontrado',
      },
    };
  }

  return Book.create(title, authorId);
};

module.exports = {
  getAll,
  findById,
  create,
};
```

Exercício 2

Crie um arquivo `controllers/BooksController.js` e transfira os middlewares relacionados ao nosso CRUD de livros para esse controller, aproveite também para criar o middleware de erro que foi ensinado no conteúdo de hoje.

Solução

Obs: Não esqueça de instalar o Joi e o Rescue com o comando `npm i joi express-rescue`

Copiar

```
// middlewares/error.js

module.exports = (err, req, res, _next) => {

  if (err.isJoi) {
    return res.status(400)
      .json({ error: { message: err.details[0].message } });
  }

  const statusByErrorCode = {
```

```
    notFound: 404,
    alreadyExists: 409,
  };

  const status = statusByErrorCode[err.code] || 500;

  res.status(status).json({ error: { message: err.message } });
};
```

Copiar

```
// controllers/Book.js

const Joi = require('joi');
const rescue = require('express-rescue');
const Book = require('../services/Book');

const getAll = rescue(async (req, res) => {
  const books = await Book.getAll();

  res.status(200).json(books);
});

const findById = rescue(async (req, res, next) => {
  const { id } = req.params;

  const book = await Book.findById(id);

  if (book.error) return next(book.error);

  res.status(200).json(book);
});

const create = rescue(async (req, res, next) => {
  const { error } = Joi.object({
    title: Joi.string().not().empty().required(),
    authorId: Joi.string().not().empty().required(),
  })
    .validate(req.body);

  if (error) return next(error);

  const { title, authorId } = req.body;
```

```
  const newBook = await Book.create(title, authorId);

  if (newBook.error) return next(newBook.error);

  res.status(201).json(newBook);
});

module.exports = {
  getAll,
  findById,
  create,
};
```

Copiar

```
// index.js

const express = require('express');
const bodyParser = require('body-parser');

const Author = require('./controllers/Author');
const Book = require('./controllers/Book');
const errorMiddleware = require('./middlewares/error');

const app = express();

app.use(bodyParser.json());

app.get('/authors', Author.getAll);
app.get('/authors/:id', Author.findById);
app.post('/authors', Author.create);

app.get('/books', Book.getAll);
app.get('/books/:id', Book.findById);
app.post('/books', Book.create);

app.use(errorMiddleware);

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Ouvindo a porta ${PORT}`);
});
```

Exercícios

Agora, a prática

Exercício 1

Crie uma nova API utilizando o express

1. A aplicação deve ser um pacote Node.js
2. Dê ao pacote o nome de `cep-lookup`
3. Utilize o express para gerenciar os endpoints da sua aplicação
4. A aplicação deve ter a rota `GET /ping` , que retorna o status `200 OK` e o seguinte JSON:

Copiar
```
{ "message": "pong!" }
```

5. A aplicação deve escutar na porta 3000
6. Deve ser possível iniciar a aplicação através do comando `npm start` .

Resolução

1. Crie uma nova pasta chamada `cep-lookup` e navegue até ela pelo terminal

Copiar
```
mkdir cep-lookup && cd cep-lookup
```

2. Dentro da pasta `cep-lookup` , inicie um novo pacote Node.js e instale as dependencies do projeto:

Copiar
```
npm init -y && npm i express express-rescue mysql2 joi dotenv
body-parser
```

3. Crie o arquivo `index.js`

Copiar
```
// index.js

// Carregamos as variáveis de ambiente
require('dotenv').config();
const express = require('express');
```

```
// Criamos a aplicação do express
const app = express();

// Registramos o endpoint `GET /ping`
app.get('/ping', (req, res) => {
  res.status(200).json({ message: 'pong!' });
});

// Lemos a porta da variável de ambiente, ou usamos 3000
const PORT = process.env.PORT || 3000;

// Iniciamos a aplicação ouvindo na porta informada na variável de ambiente.
app.listen(PORT, () => { console.log(`Listening on port ${PORT}`);
});
```

4. Adicione o script `start` ao `package.json` :

Copiar

```
/* ... */
// "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
// },
/* ... */
```

5. Crie o arquivo .env:

Copiar

```
# .env
PORT=3000
```

Exercício 2

2. Crie o endpoint `GET /cep/:cep`
3. O endpoint deve receber, no parâmetro `:cep` , um número de CEP válido.
4. O CEP precisa conter 8 dígitos númericos e pode ou não possui traço.
    - Dica Utilize o regex `\d{5}-?\d{3}` para validar o CEP.
5. Caso o CEP seja inválido, retorne o status `400 Bad Request` e o seguinte JSON:

Copiar
```
 { "error": { "code": "invalidData", "message": "CEP inválido" } }
```

4. Caso o CEP seja válido, realize uma busca no banco de dados.
    1. Caso o CEP não exista no banco de dados, retorne o status `404 Not Found` e o seguinte JSON:

Copiar
```
{ "error": { "code": "notFound", "message": "CEP não encontrado" } }
```

Copiar
```
 2. Caso o CEP exista, retorne o status `200 OK` e os dados do CEP no seguinte formato:
```

Copiar
```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

Resolução

1. Crie o arquivo `models/connection.js` :

OBS: Aproveite para colocar nas variáveis de ambiente as informações de conexão com o banco de dados.

Copiar
```
// models/connection.js

const mysql = require('mysql2/promise');

const connection = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_DATABASE});

module.exports = connection;
```

Copiar
```
# .env
```

```
PORT=3000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_DATABASE=cep_lookup
```

2. Crie o Model de CEP no arquivo `models/Cep.js`

Copiar

```
// models/Cep.js

// Importamos a conexão com o banco
const connection = require('./connection');

// Regex que identifica um CEP já formatado
const CEP_REGEX = /\d{5}-\d{3}/;

// Função que formata um CEP
const formatCep = (cep) => {
  // Caso o CEP já esteja formatado, retorna o próprio CEP
  if (CEP_REGEX.test(cep)) return cep;

  // Caso não esteja formatado, utiliza regex para adicionar a
formatação
  return cep.replace(/(\d{5})(\d{3})/, '$1-$2');
};

// Formata os campos para exibição
const getNewCep = ({ cep, logradouro, bairro, localidade, uf }) =>
({
  cep: formatCep(cep),
  logradouro,
  bairro,
  localidade,
  uf});

const findAddressByCep = async (cepToSearch) => {
  // Removemos todos os traços, pois armazenamos o CEP
  // puro no banco
  const treatedCep = cepToSearch.replace('-', '');

  const query = 'SELECT cep, logradouro, bairro, localidade, uf FROM
ceps WHERE cep = ?';
```

```
  // Executamos a query, selecionando o primeiro resultado, caso
exista
  // e assumindo `null`, caso contrário
  const result = await connection.execute(query, [treatedCep])
    .then(([results]) => (results.length ? results[0] : null));

  // Caso nenhum resultado seja encontrado, retornamos `null`
  if (!result) return null;

  // Retornamos os dados do CEP formatados pela função getNewCep
  return getNewCep(result);
};

module.exports = {
  findAddressByCep,
};
```

3. Crie o service de CEP em `services/Cep.js`

Copiar

```
// services/Cep.js

const Cep = require('../models/Cep');

const CEP_REGEX = /\d{5}-?\d{3}/;

const findAddressByCep = async (searchedCep) => {
  // Valida o CEP e em caso dele ser falso, retorna uma erro
  if (!CEP_REGEX.test(cep)) {
    return {
      error: {
        code: 'invalidData',
        message: 'CEP inválido',
      }
    }

  // Buscamos o CEP através do Model
  const cep = await Cep.findAddressByCep(searchedCep);

  // Caso não econtre nenhum CEP, o service retorna um objeto de
erro.
  if (!cep) {
```

```
    return {
      error: {
        code: 'notFound',
        message: 'CEP não encontrado'
      },
    };
  }

  // Por fim, retornamos o CEP correto
  return cep;
};

module.exports = {
  findAddressByCep,
};
```

4. Crie o controller de CEP em `controllers/Cep.js`

Copiar

```
// controllers/Cep.js

const rescue = require('express-rescue');
const service = require('../services/Cep');

const findAddressByCep = rescue(async (req, res, next) => {
  const { cep } = req.params;

  const address = await service.findAddressByCep(cep);

  if (address.error) {
    return next(address.error);
  }

  return res.status(200).json(address);
});

module.exports = {
  findAddressByCep,
};
```

5. Agora, vamos criar o middleware de erro. Crie o arquivo `middlewares/error.js`

Copiar

```
// middlewares/error.js

module.exports = (err, req, res, _next) => {
  if (err.isJoi) {
    return res.status(400)
      .json({ error: { message: err.details[0].message } });
  }

  // Verificamos se esse é um erro de domínio
  if (err.code) {
    const statusByErrorCode = {
      notFound: 404,
    };

    // Usamos o código do erro para determinar qual o status code
adequado
    const status = statusByErrorCode[err.code] || 500;

    // Enviamos o status code e o erro como resposta
    res.status(status).json(err);
  }

  // Caso não seja um erro de domínio, enviamos uma resposta de erro
desconhecido.
  console.error(err);
  res.status(500).json({ error: { code: 'internal', message:
'Internal server error' } });
};
```

6. Adicione o `body-parser` e o middleware de erro ao `index.js` :

Copiar

```
// index.js

// Carregamos as variáveis de ambiente
// require('dotenv').config();
// const express = require('express');
const bodyParser = require('body-parser');
const Cep = require('./controllers/Cep');
const errorMiddleware = require('./middlewares/error.js');

// // Criamos a aplicação do express
// const app = express();
```

```
app.use(bodyParser.json());

// // Registramos o endpoint `GET /ping`
// app.get('/ping', (req, res) => {
//   res.status(200).json({ message: 'pong!' });
// });

app.get('/cep/:cep', Cep.findAddressByCep);

app.use(errorMiddleware);

// // Lemos a porta da variável de ambiente, ou usamos 3000
// const PORT = process.env.PORT || 3000;

// // Iniciamos a aplicação ouvindo na porta informada na variável de ambiente.
// app.listen(PORT, () => { console.log(`Listening on port ${PORT}`); });
```

Exercício 3

Crie o endpoint `POST /cep`

1. O endpoint deve receber a seguinte estrutura no corpo da requisição:

Copiar

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

2. Todos os campos são obrigatórios
3. O CEP deve ser composto por 9 dígitos com traço.
    - Dica : Utilize o seguinte regex para validar o CEP: `\d{5}-\d{3}`
4. Se o CEP já existir, retorne o status `409 Conflict` com o segiunte JSON:

Copiar

```
{
```

```
  "error": { "code": "alreadyExists", "message": "CEP já existente"
}
}
```

5. Se o CEP ainda não existir, armazene-o no banco de dados e retorne o status `201 Created` com os dados do novo CEP no seguinte formato:

Copiar

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

Resolução

1. Crie a função `create` no model de cep:

Copiar

```
// models/Cep.js

// const connection = require('./connection');

// const CEP_REGEX = /\d{5}-\d{3}/;

// const formatCep = (cep) => {
//   if (CEP_REGEX.test(cep)) return cep;

//   return cep.replace(/(\d{5})(\d{3})/, '$1-$2');
// };

// const getNewCep = ({ cep, logradouro, bairro, localidade, uf }) =>
=> ({
//   cep: formatCep(cep),
//   logradouro,
//   bairro,
//   localidade,
//   uf,
// });
```

```
// const findAddressByCep = async (cepToSearch) => {
//   // Removemos todos os traços, pois armazenamos o CEP
//   // puro no banco
//   const treatedCep = cepToSearch.replace('-', '');

//   const query = 'SELECT cep, logradouro, bairro, localidade, uf
FROM cep WHERE cep = ?';

//   const result = await connection.execute(query, [treatedCep])
//     .then(([results]) => (results.length ? results[0] : null));

//   if (!result) return null;

//   return getNewCep(result);
// };

const create = async ({ cep: rawCep, logradouro, bairro, localidade,
uf }) => {
  // Removemos o traço do CEP para armazená-lo de forma limpa
  const cep = rawCep.replace(/-/ig, '');

  const query = 'INSERT INTO ceps (cep, logradouro, bairro,
localidade, uf) VALUES (?, ?, ?, ?, ?)';

  // Executamos a query
  await connection.execute(query, [cep, logradouro, bairro,
localidade, uf]);

  // Depois de inserir, retornamos os dados, como sinal de que foram
guardados no banco
  return { cep, logradouro, bairro, localidade, uf };
};

// module.exports = {
//   findAddressByCep,
    create,
// };
```

2. Agora, podemos utilizar essa função no Service:

Copiar

```
// services/Cep.js
```

```
// const Cep = require('../models/Cep');

// const CEP_REGEX = /\d{5}-?\d{3}/;

// const findAddressByCep = async (searchedCep) => {
//  if (!CEP_REGEX.test(cep)) {
//    return {
//      error: {
//        code: 'invalidData',
//        message: 'CEP inválido',
//      }
//    }

//   const cep = await Cep.findAddressByCep(searchedCep);

//   if (!cep) {
//     return {
//       error: {
//         code: 'notFound',
//         message: 'CEP não encontrado',
//       },
//     };
//   }

//   return cep;
// };

const create = async ({ cep, logradouro, bairro, localidade, uf })
=> {
  // Começamos buscando o cep que estamos tentando cadastrat
  const existingCep = await Cep.findAddressByCep(cep);

  // Caso o CEP já exista, retornamos um erro dizendo que ele já
existe
  if (existingCep) {
    return {
      error: {
        code: 'alreadyExists',
        message: 'CEP já existente',
      },
    };
  }
```

```
  // Caso o CEP ainda não exista, chamamos o Model para criá-lo no
banco, e devolvemos esse resultado
  return Cep.create({ cep, logradouro, bairro, localidade, uf });
};

// module.exports = {
//   findAddressByCep,
    create,
// };
```

3. E agora, o controller:

Copiar

```
// controllers/Cep.js

// const rescue = require('express-rescue');
// const service = require('../services/Cep');
const Joi = require('joi');

// const findAddressByCep = rescue(async (req, res, next) => {
//   const { cep } = req.params;

//   const address = await service.findAddressByCep(cep);

//   if (address.error) {
//     return next(address.error);
//   }

//   return res.status(200).json(address);
// });

const create = rescue(async (req, res, next) => {
  // Armazenamos essa parte do schema do Joi para reutilizá-la
  const requiredNonEmptyString =
Joi.string().not().empty().required();

  // Validamos o corpo da request
  const { error } = Joi.object({
    cep: Joi.string().regex(/\d{5}-\d{3}/).required(),
    logradouro: requiredNonEmptyString,
    bairro: requiredNonEmptyString,
    localidade: requiredNonEmptyString,
    uf: requiredNonEmptyString.length(2),
```

```
  }).validate(req.body);

  // Caso haja erro de validação, iniciamos o fluxo de erro
  if (error) return next(error);

  // Caso não haja erero de validação, pedimos para o service criar
o cep
  const newCep = await service.create(req.body);

  // Caso o service nos retorne um erro
  if (newCep.error) {
    // Iniciamos o fluxo de erro
    return next(newCep.error);
  }

  // Caso contrário, retornamos o status `201 Created`, e o novo
CEP, em formato JSON
  res.status(201).json(newCep);
});

// module.exports = {
//   findAddressByCep,
    create,
// };
```

4. Precisamos adicionar o código de erro `alreadyExists` , que estamos utilizando no service, ao middleware de erro:

Copiar

```
// middlewares/error.js

// module.exports = (err, req, res, _next) => {
//   if (err.isJoi) {
//     return res.status(400)
//       .json({ error: { message: err.details[0].message } });
//   }

//   // Verificamos se esse é um erro de domínio
//   if (err.code) {
//     const statusByErrorCode = {
//       notFound: 404,
        alreadyExists: 409,
//     };
```

```
//     // Usamos o código do erro para determinar qual o status code
adequado
//     const status = statusByErrorCode[err.code] || 500;

//     // Enviamos o status code e o erro como resposta
//     return res.status(status).json(err);
//   }

//   // Caso não seja um erro de domínio, enviamos uma resposta de
erro desconhecido.
//   console.error(err);
//   res.status(500).json({ error: { code: 'internal', message:
'Internal server error' } });
// };
```

5. E, agora, é só criar o endpoint no `index.js`

Copiar

```
// index.js

/* ... */

// app.get('/cep/:cep', Cep.findAddressByCep);
app.post('/cep', Cep.create);

// app.use(errorMiddleware);

/* ... */
```

Bônus

Exercício 1

Utilize uma API externa para buscar CEPs que não existem no banco de dados

1. Quando um CEP não existir no banco de dados, utilize a API https://viacep.com.br/ws/[numero-do-cep]/json/ para obter suas informações.

2. Caso o CEP não exista na API externa, você receberá o JSON `{ "erro": true }` . Nesse caso, retorne status `404 Not Found` com o seguinte JSON:

Copiar
```
{ "error": { "code": "notFound", "message": "CEP não encontrado" } }
```

3. Caso o CEP exista na API externa, armaezene-o no banco e devolva seus dados no seguinte formato:

Copiar
```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
}
```

Dica : Na arquitetura MSC, os models são responsáveis por toda a comunicação externa de uma aplicação, o que inclui APIs externas. Logo, você precisará de um model para acesar a API.

Resolução

1. Começamos instalando a biblioteca `node-fetch` , que vamos utilizar para fazer requisições. Execue o seguinte comando no terminal:

Copiar
```
npm i node-fetch
```

2. Agora, criamos o modelo que vai se comunicar com essa API. Crie o arquivo `models/ViaCep.js` :

Copiar
```
// models/ViaCep.js

const fetch = require('node-fetch');

const lookupCep = async (cepToLookup) => {
  const response = await
fetch(`https://viacep.com.br/ws/${cepToLookup}/json/`);

  if (!response.ok) return null;
```

```
  const address = await response.json();

  if (address.erro) return null;

  return address;
};

module.exports = {
  lookupCep,
};
```

3. Depois, podemos alterar nosso service de Cep para que utilize esse novo modelo. Altere a função `findAddressByCep` no arquivo `services/Cep.js` :

Copiar

```
// service/Cep.js

/* ... */

const findAddressByCep = async (searchedCep) => {
  // Começamos buscando o CEP no banco de dados
  const cep = await Cep.findAddressByCep(searchedCep);

  // Caso encontremos, retornamos sem consultar a API
  if (cep) {
    return cep;
  }

  // Caso o CEP não exista no banco de dados, buscamos na API
  const cepFromApi = await ViaCep.lookupCep(searchedCep);

  // Caso o CEP não exista na API,
  // retornamos um erro dizendo que
  // o CEP não foi encontrado
  if (!cepFromApi) {
    return {
      error: {
        code: 'notFound',
        message: 'CEP não encontrado',
      },
    };
  }
```

```
  // Caso o CEP exista na API, pedimos ao model
  // que armazene-o no banco e retornamos
  // o resultado
  return Cep.create(cepFromApi);
};

/* ... */
```

4. Não tem passo 4! Como nossa aplicação está bem separada em camadas, não precisamos mexer na camada de controller ou em qualquer outra camada, pois o que precisávamos era alterar uma regra de negócio, e conseguimos! 😄🥳