



Plataformas de aplica  es Web

Aula 11 - PetTopStore - Admin - Parte 3 (API)



Material Did tico do Instituto Metr pole Digital - IMD

Termo de uso

Os materiais did ticos aqui disponibilizados est o licenciados atrav s de Creative Commons **Atribui  o-SemDeriva  es-SemDerivados CC BY-NC-ND**. Voc  possui a permiss o para realizar o download e compartilhar, desde que atribua os cr ditos do autor. N o poder  alter -los e nem utiliza-los para fins comerciais.

Atribui  o-SemDeriva  es-SemDerivados

CC BY-NC-ND



<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Apresentação

Nessa aula iremos fazer uma pequena mudança no projeto do Admin para criar o recurso de categorias nos produtos e também criar uma API JSON dentro do mesmo projeto Admin para que seja possível sua integração com o sistema do PDV (Ponto de Venda) e com a Loja on-line, que ainda serão desenvolvidos.

Ajustando o projeto Admin para adicionar categorias no cadastro dos produtos

Criando a seed de categorias

Precisamos inicialmente criar uma seed para adicionar algumas categorias na nossa loja:

```
npx knex seed:make create_initial_categories
```

O arquivo da seed deve ser com o abaixo:

seeds/create_initial_categories.js

```
exports.seed = function(knex) {  
  // Remove todos as categories  
  return knex('categories').del()  
    .then(function () {  
    // Depois insere as categories:  
    return knex('categories').insert([  
      {  
        name: 'Alimentos'  
      },  
      {  
        name: 'Brinquedos'  
      },  
      {  
        name: 'Animais'  
      }  
    ]);  
  });  
};
```

Repare que o comando acima inicialmente remove as categorias já adicionadas

antes de criar as novas. Se você deseja criar uma seed que não remove registros anteriores, basta omitir o ".del()".

Executando a seed específica criada (com a opção --specific):

```
npx knex seed:run --specific=create_initial_categories.js
```

Usando as categorias cadastradas na seed nos formulários de cadastro de produto

Em routes/products.js, adicione na rota '/new' uma variável, disponível na renderização da view, com a lista de categorias (que será mostrada em um select) e também na rota '/create' adicione o atributo category_id ao salvar o produto. Na rota '/' (listagem de produtos) também é feita a busca da categoria do produto e para cada um dos listados. O arquivo routes/products.js ficará assim ao final:

routes/products.js

```
const express = require('express');
const router = express.Router();

const knexConfig = require('../knexfile');
const knex = require('knex')(knexConfig);

const multer = require('multer');

const upload = multer({ dest: './public/images' });

router.get('/', async (req, res) => {
  const products = await knex.table('products').select();
  for (const product of products) {
    product.category = await knex.table('categories').where('id',
    '=', product.category_id).first()
    console.log(product.category);
  }
  res.render('products/list', { products });
});

router.get('/new', async (req, res) => {
  const categories = await knex.table('categories').select();
  res.render('products/new', { categories });
});
```

```
router.post('/create', upload.single('photo'), async (req, res) => {
  await knex.table('products').insert({
    name: req.body.name,
    description: req.body.description,
    price: parseFloat(req.body.price),
    photo: req.file.filename,
    category_id: req.body.category_id
  });
  res.redirect('/products');
});

module.exports = router;
```

Altere o arquivo `views/products/new.ejs` para adicionar a categoria no cadastro do produto:

```
<h1>Criar produto</h1>
<form method="POST" action="/products/create"
  enctype="multipart/form-data">
  Foto: <input type="file" name="photo" /><br/>
  Nome: <input type="text" name="name" /><br/>
  Preço: <input type="text" name="price" /><br/>
  Descrição: <input type="text" name="description" /><br/>
  Categoria:
  <select name="category_id">
    <% for (const category of categories) { %>
      <option value="<%= category.id %>"><%= category.name
%></option>
    <% } %>
  </select><br/>
  <input type="submit" value="Salvar produto" />
</form>
```

Altere o arquivo `views/products/list.ejs` para adicionar a categoria na listagem:

```
<h1>Lista de produtos</h1>

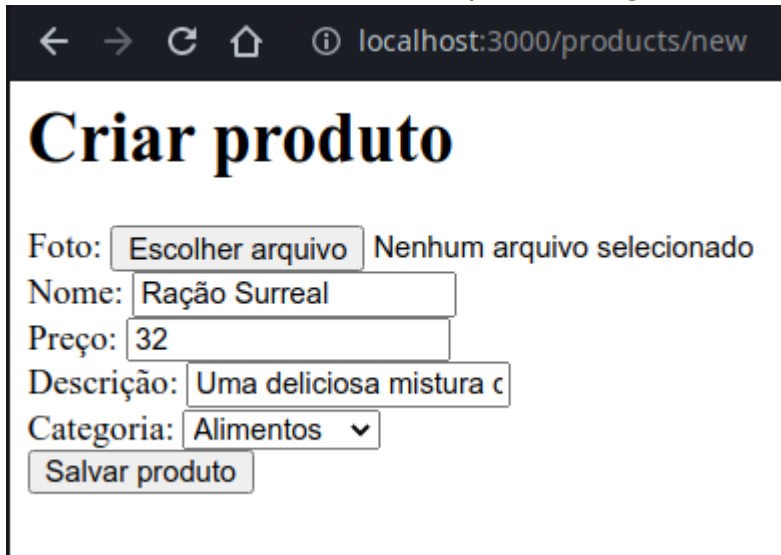
<a href="/">Voltar</a>
<a href="/products/new">Adicionar produto</a>

<table>
  <% for (const product of products) { %>
    <tr>
```

```
<td></td>
<td><%= product.name %></td>
<td><%= product.price %></td>
<td><%= product.category?.name %></td>
</tr>
<% } %>
</table>
```

Repare que `product.category?.name` tem uma "?" indicando que só será mostrado o nome se a categoria existir. Esse é um recurso interessante do Javascript e nos ajuda pois alguns produtos podem ter sido cadastrados sem categoria antes desse recurso ter sido criado.

O formulário de cadastro de produto agora tem a Categoria para ser escolhida:



← → ↻ 🏠 ⓘ localhost:3000/products/new

Criar produto

Foto: Nenhum arquivo selecionado

Nome:

Preço:

Descrição:

Categoria: ▼

Claro que para se ter uma gestão completa de produtos e categorias, vários outros recursos poderiam ser adicionados como "editar produto", "criar/editar categorias", poderíamos também adicionar uma biblioteca CSS para melhorar o visual e usabilidade, enfim... muito pode ainda ser feito nessa aplicação Admin. Vamos, entretanto, nos manter com o básico pois o nosso objetivo é ter uma visão geral de como integrar sistemas com diferentes plataformas e você pode depois incrementar o sistema com os conhecimentos que adquiriu e tecnologias que desejar.

API

A API da PetTopStore será criada como um conjunto de rotas extras dentro do projeto Admin, porém com uma estratégia de autenticação separada e ela permitirá a integração com os sistemas de PDV e Loja online.

Estratégia de autenticação

No projeto admin, atualmente temos uma autenticação para os administradores (employees com admin=true) realizada com a estratégia cookie-session. Para a API iremos utilizar uma outra estratégia: JSON Web Tokens (JWT).

É possível criar APIs com outras estratégias de autenticação, inclusive com cookies, porém o uso do JWT tem algumas vantagens e uma grande popularidade hoje em dia, por isso iremos utilizar somente para a parte da API.

Um JWT nada mais é que um token com dados assinado pelo serviço que o criou. Esses dados não podem ser alterados sem que a assinatura se invalide. Para a assinatura ser realizada com sucesso, é necessário que quem o gerou utilize um segredo (como uma senha) que também será utilizado para o decodificar e o verificar no futuro. Esse segredo não pode ser revelado para o mundo externo, devendo somente ficar em posse da aplicação que gera e verifica os tokens JWT (No nosso caso a API no projeto Admin).

A nossa estratégia será simples: No processo de sign_in, a API irá verificar se o e-mail e senha do usuário estão válidos e gerar um JWT assinado com a informação se esse usuário é um cliente ou funcionário, juntos com os dados dele. Sim, iremos autenticar com JWT tanto funcionários (usuários do PDV) como clientes (usuários da loja virtual). Esse JWT será retornado para o sistema correspondente e ele deverá se encarregar de guardar esse token JWT para realizar as operações que precisam que o usuário esteja autenticado, como por exemplo: criar vendas pelo PDV ou pela loja virtual.

A API deverá então, nessas ações que requerem autenticação, verificar se o token JWT enviado, junto com os restantes dos dados, é válido e de um cliente ou funcionário cadastrado no sistema, antes de realizar a operação.

Para mais informações sobre o JWT visite: <https://jwt.io/>

Para mais informações sobre JSON visite:
https://www.w3schools.com/js/js_json_intro.asp

Como estamos usando Node.js, vamos usar uma biblioteca chamada jsonwebtoken(<https://github.com/auth0/node-jwebtoken>) para criar JWTs assinados na autenticação e verificar JWTs enviados para a API nas ações que requerem usuário logado.

Já instale no sistema admin essa biblioteca:

```
npm install jsonwebtoken
```

API - Middleware requireJWT

O sistema Admin já tem um middleware chamado "requireAuth" criado anteriormente e responsável por restringir acesso às partes do sistema onde o administrador precisa estar logado. Vamos criar um arquivo em "middlewares/requireJWT.js" com uma proposta similar, porém para ser utilizada nas rotas da API.

Os tokens JWT devem ser enviados em um atributo do header da requisições chamado "Authorization" e com o valor no formato: "Bearer TOKEN_JWT_AQUI". Repare que existe a palavra "Bearer" antes, seguida de um espaço e depois o token JWT. É assim que o valor do cabeçalho Authorization deve ser enviado nas requisições à API pelas aplicações.

O conteúdo do "middlewares/requireJWT.js" é:

```
const jwt = require('jsonwebtoken');
const segredoJWT = 'frase segredo para criptografia do jwt';

module.exports = async (req, res, next) => {
  // verifica se a sessão está vazia (deslogado)
  if (!req.headers.authorization) {
    return res.status(401).json({
      message: 'Não autorizado'
    });
  }

  try {
    const token = req.headers.authorization.split(' ')[1];
    const decodedJWT = jwt.verify(token, segredoJWT);
    res.locals.jwt = decodedJWT;
    next();
  } catch (error) {
    return res.status(401).json({
      message: 'Token inválido'
    });
  }
}
```

O middleware requireJWT.js, como visto no código acima, realiza algumas verificações. A primeira é se o header authorization foi enviado, caso contrário ele retorna um json com status 401 e mensagem "Não autorizado".

Em seguida ele tenta remover o token do header authorization (que tem o "Bearer "

antes) com o comando:

```
const token = req.headers.authorization.split(' ')[1];
```

Esse comando pega somente a segunda parte do authorization header, que é justamente o token JWT (ainda codificado).

Em seguida ele tenta decodificar o token com o segredo armazenado em uma variável. Esse token JWT decodificado na variável "decodedJWT" é então colocado em res.locals.jwt para que as requisições que dependem desse middleware tenha acesso aos dados presentes dentro do token de forma fácil e já decodificada. Os comandos para isso foram:

```
const decodedJWT = jwt.verify(token, segredoJWT);  
res.locals.jwt = decodedJWT;
```

Repare que a biblioteca jsonwebtoken é usada aqui com o método jwt.verify que recebe o token codificado, o segredo e retorna o token decodificado (se o segredo estiver correto). Caso o segredo esteja incorreto, uma exceção de erro será disparada e capturada pelo catch.

API - rotas

Vamos criar as rotas para a API. Para isso crie uma pasta chamada "api" dentro de "routes" e lá crie 5 arquivos:

- index.js
- auth.js
- clients.js
- products.js
- sales.js

Cada um desses arquivos irá conter rotas com as ações correspondentes ao seu nome e o arquivo index.js irá basicamente agregar todos os demais para manter o projeto organizado.

Vamos ver o conteúdo de cada um desses arquivos:

API - routes/api/index.js

Conteúdo:

```
const express = require('express');
```



```
const authRouter = require('./auth');
const clientsRouter = require('./clients');
const productsRouter = require('./products');
const salesRouter = require('./sales');

const router = express.Router();

router.use('/auth', authRouter);
router.use('/clients', clientsRouter);
router.use('/products', productsRouter);
router.use('/sales', salesRouter);

module.exports = router;
```

Esse arquivo simplesmente inclui os demais e cria as rotas correspondentes para cada um deles. O index.js é o arquivo padrão que deve ser incluído no app.js para a rota raiz da api: '/api'. Iremos fazer esse procedimento ao final da apresentação dos arquivos restantes da API.

API - routes/api/auth.js

Conteúdo:

```
var express = require('express');
var router = express.Router();

const knexConfig = require('../../knexfile');
const knex = require('knex')(knexConfig);

const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const segredoJWT = 'frase segredo para criptografia do jwt';

// employee sign_in
router.post('/employee/sign_in', async(req, res) => {
  // busca employee no banco com esse email
  const employee = await knex.table('employees').where({ email:
req.body.email }).first();

  // Caso o employee não exista ou sua senha criptografada seja
diferente da armazenada no banco, retorna um erro:
  if (!employee || !bcrypt.compareSync(req.body.password,
employee.password)) {
```

```
    return res.status(401).json({
      message: 'Funcionário não existe'
    });
  }

  // criação do conteúdo (payload) do token JWT com o employee
  encontrado e com senha correta
  const conteudoJWT = {
    employee: employee
  };

  // gera um token com o conteúdo (payload) e assinado com o
  segredoJWT. É atribuído uma expiração de 2 dias para o token.
  const token = jwt.sign(conteudoJWT, segredoJWT, { expiresIn: '2
  days' });

  // o token é adicionado ao employee, que é retornado no JSON
  employee.token = token;
  return res.json({ employee });
});

// client sign_in
router.post('/client/sign_in', async(req, res) => {
  // busca client no banco com esse email
  const client = await knex.table('clients').where({ email:
  req.body.email }).first();

  // Caso o client não exista ou sua senha criptografada seja
  diferente da armazenada no banco, retorna um erro:
  if (!client || !bcrypt.compareSync(req.body.password,
  client.password)) {
    return res.status(401).json({
      message: 'Cliente não existe'
    });
  }

  // criação do conteúdo (payload) do token JWT com o employee
  encontrado e com senha correta
  const conteudoJWT = {
    client: client
  };

  // gera um token com o conteúdo (payload) e assinado com o
  segredoJWT. É atribuído uma expiração de 2 dias para o token.
```

```
const token = jwt.sign(conteudoJWT, segredoJWT, { expiresIn: '2
days' });

// o token é adicionado ao client, que é retornado no JSON
client.token = token;
return res.json({ client });
});

// client registration (from website)
router.post('/client/registration', async(req ,res) => {
  // busca client no banco com esse email
  const existingClient = await knex.table('clients').where({ email:
req.body.email }).first();

  // Se o client já existe, retorna um erro pois não pode cadastrar
dois com o mesmo e-mail
  if (existingClient) {
    return res.status(401).json({
      message: 'Cliente já existe com esse e-mail'
    });
  }

  // constroi o objeto com os dados do novo client(incluindo a senha
critografada com o bcrypt)
  let newClientData = {
    name: req.body.name,
    email: req.body.email,
    password: bcrypt.hashSync(req.body.password, 10)
  };

  // insere o client no banco e recebe um array que deve conter só
um elemento (o id do cliente inserido)
  let clientIDs = await knex.table('clients').insert(newClientData);

  // retorna uma mensagem de sucesso e o ID do client inserido no
banco (primeiro elemento do array retornado pelo comando insert)
  return res.json({
    message: 'Cliente registrado com sucesso',
    clienteID: clientIDs[0]
  });
});

module.exports = router;
```

O arquivo auth.js é responsável pela autenticação da API, utilizando a estratégia de tokens JWT.

Existem 3 rotas nesse arquivo:

- POST /employee/sign_in
- POST /client/sign_in
- POST /client/registration

Rotas /employee/sign_in e /client/sign_in e

Rota responsável por autenticar um employee (funcionário) ou um client (cliente) por email/senha e gerar um token. Os comentários no código da rota explicam o seu passo a passo.

- Inicialmente é buscado o employee no banco pelo email (Caso o employee/client não exista ou sua senha criptografada seja diferente da armazenada no banco, retorna um erro
- É criado um token JWT com e dentro do payload os dados do employee
- O token é inserido no employee e retornado como JSON

Node que a assinatura do token é setada para expirar em 2 dias com o comando:

```
const token = jwt.sign(conteudoJWT, segredoJWT, { expiresIn: '2 days' });
```

Isso cria um campo especial no token chamado "exp" com um valor que representa seu momento de expiração.

As rotas /employee/sign_in e /client/sign_in são muito parecidas, e única mudança é que uma autentica um funcionário e a outra um cliente da loja e o token JWT gerado muda um pouco pois retornam campos diferentes (employee para uma rota e client para a outra rota).

Essa diferença do JWT (client ou employee) permite que nos sistemas possa ser verificado se o JWT pertence a um cliente ou funcionário.

Rota /client/registration

Essa rota registra um novo cliente no banco de dados. Ela tem os detalhes o que acontece a cada comando comentado no seu código.

É uma rota simples que verifica se um cliente a ser registrado já existe no banco, caso contrário o registra com a senha informada criptografada com o bcrypt.

Tanto os funcionários logados no PDV poderão usar essa rota para cadastrar

clientes na loja física, quanto os próprios clientes poderão usar a loja virtual para se cadastrar através dessa rota na API.

API - routes/api/clients.js

Conteúdo

```
var express = require('express');
var router = express.Router();

const knexConfig = require('../../knexfile');
const knex = require('knex')(knexConfig);

const requireJWT = require('../../middlewares/requireJWT');

// Retorna uma lista de cliente. Requer que o usuário esteja logado
// pelo uso do middleware requireJWT
router.get('/', [requireJWT], async (req, res) => {

  // obtem o JWT decodificado pelo middleware requireJWT e salvo em
  // res.locals.jwt
  const jwt = res.locals.jwt;

  // verifica se não é um JWT de um employee e retorna um erro
  if (!jwt.employee) {
    return res.status(401).json({
      message: 'Não é um funcionário'
    });
  }

  // Obtém a lista de clientes da base de dados e retorna um JSON
  // com ela.
  const clients = await knex.table('clients').select(['id', 'name',
    'email']);
  res.json({
    clients
  });
});

module.exports = router;
```

Como visto nos comentários do código, esse arquivo só tem uma rota que lista todos os clientes da base caso o usuário que a chamou seja tenha passado um

token JWT e se esse token é de um funcionário (employee), ou seja, essa rota é restrita para funcionários. Será utilizada exclusivamente pelo PDV.

API - routes/api/products.js

Conteúdo:

```
var express = require('express');
var router = express.Router();

const knexConfig = require('../../knexfile');
const knex = require('knex')(knexConfig);

// Busca de produtos.
router.get('/search', async(req, res) => {

  // Obtem um query builder do knex da tabela products
  let productsQuery = knex.table('products');

  // se o "term" de busca foi passado na query string da URL
  if (req.query.term) {

    // adiciona no query builder as restrições da busca por nome e
    // descrição
    productsQuery = productsQuery
      .where('name', 'LIKE', `%${req.query.term}%`)
      .orWhere('description', 'LIKE', `%${req.query.term}%`);
  }

  // se for passado um category_id na query string da URL
  if (req.query.category_id) {

    // adiciona a restrição de buscar produtos somente por uma
    // determinada categoria (category_id)
    // Exemplo;
    http://localhost:3000/api/products/search?category_id=7
    productsQuery = productsQuery
      .where('category_id', '=', req.query.category_id);
  }

  // Caso tenha sido passado o campo "order", muda a ordenação da
  // busca (name ou price)
  // Exemplo: http://localhost:3000/api/products/search?order=price
```

```
if (req.query.order) {
  if (req.query.order == 'name') {
    productsQuery = productsQuery.orderBy('name');
  } else if (req.query.order == 'price') {
    productsQuery = productsQuery.orderBy('price');
  }
}

// faz um left join de products com categories para poder retornar
o nome da categoria do produto em uma consulta só.
productsQuery = productsQuery.leftJoin('categories',
'products.category_id', 'categories.id');
// informa que além dos campos de produtos, retornar também o nome
da categoria dele como categoryName
productsQuery = productsQuery.select('products.*',
'categories.name as categoryName')
const products = await productsQuery;

res.json({ products })
});

// Obtém um produto específico por ID
router.get('/:id', async (req, res) => {

  // busca um único produto, restrito por ID e com o nome da
  categoria no campo categoryName
  const product = await knex.table('products').where('products.id',
  '=', req.params.id)
  .leftJoin('categories', 'products.category_id', 'categories.id')
  .select('products.*', 'categories.name as categoryName')
  .first();
  return res.json({ product });
})

module.exports = router;
```

Esse arquivo tem uma rota de busca de produtos, com todos os detalhes da implementação com comentários no código. Além disso existe uma rota que busca um produto específico por ID.

Em ambas as rotas é retornado, além dos dados do produto, o nome da sua categoria, com o uso do leftJoin.

Conteúdo:

```
var express = require('express');
var router = express.Router();

const requireJWT = require('../middlewares/requireJWT');

const knexConfig = require('../knexfile');
const knex = require('knex')(knexConfig);

// Obtém uma venda (sale) através de seu id.
router.get('/:sale_id', [requireJWT], async (req, res) => {

  // obtem o token JWT decodificado pelo middleware requireJWT
  const jwt = res.locals.jwt;

  // Só permite que funcionários(employee) usem essa rota
  if (!jwt.employee) {
    return res.status(401).json({
      message: 'Não é funcionário'
    });
  }

  // Busca uma venda (sale) através do ID passado.
  const sale = await knex.table('sales').where('sales.id', '=',
req.params.sale_id).first();
  // busca todos os itens da venda encontrada
  sale.items = await knex.table('items').where('sale_id', '=',
sale.id);

  // para cada item da venda, buscar o produto desse item
  for (const item of sale.items) {
    item.product = await knex.table('products').where('id', '=',
item.product_id).first();
  }

  //retorna os dados da venda, já com os itens e produtos
  res.json({ sale });
});

// cria uma nova venda.
// pode ser chamado por um funcionário no PDV ou por um client ena
loja online
router.post('/', [requireJWT], async (req, res) => {

  // obtem o token JWT decodificado pelo middleware requireJWT
```



```
const jwt = res.locals.jwt;
let sale = {};

// Uma venda pode ser criada por um funcionário ou um cliente. O
token JWT tem a informação se ele é de um client ou employee
// caso seja por um funcionário:
if (jwt.employee) {
  // então a venda a ser criada recebe o id do cliente escolhido
no PDV, além do id do funcionário (employee) que está criando a
venda
  sale.client_id = req.body.client_id;
  sale.employee_id = jwt.employee.id;
} else if (jwt.client) {
  // caso seja um cliente, veio da loja on-line, então não existe
funcionário envolvido.
  // nesse caso o id do cliente é o id que está no token JWT dele,
já que ele mesmo criou a venda para ele mesmo pela loja (fez uma
compra).
  sale.client_id = jwt.client.id;

  // e o ID do employee deve ser nulo, pois não existe funcionário
envolvido no processo.
  sale.employee_id = null;
} else {
  return res.status(401).json({
    message: 'Não é cliente nem funcionário'
  })
}

// insere a venda no banco
const result = await knex.table('sales').insert(sale);

// o id da venda inserida é o primeiro item do result (array), de
acordo com o padrão do knex
sale.id = result[0];

// Com a venda já inserido podemos agora inserir no banco os itens
na venda criada
const items = [];
// é passado um array de produtos que são os itens da venda no PDV
ou compra no site on-line
// para cada um desses productIDs, criar noco elemento no array
items com o ID da venda e o ID do produto.
for (const productID of req.body.productIDs) {
```

```
    items.push({
      sale_id: sale.id,
      product_id: productID
    });
  }

  // Inserir todos os itens do array no banco de dados
  await knex.table('items').insert(items);

  // retornar a venda criada como JSON
  res.json({ sale });
});

module.exports = router;
```

São apenas duas rotas aqui, uma para se buscar uma venda(sale) do banco e a outra para se criar uma nova venda.

Todos os detalhes de implementação estão comentados no código e servem como explicação de cada comando.

Repare que a rota de criação de uma venda pode ser executada por um usuário do tipo employee ou client. Isso é determinado e verificado pelo seu token JWT.

No caso de um employee criando a venda, então é uma venda de balcão pelo PDV e deve ser utilizado o clientID passado para a API para se criar a venda. Caso seja uma venda criada por um cliente, então se trata de uma compra pela loja virtual e então o clientID utilizado nessa venda é o do usuário (client) logado, ou seja, o clientID é obtido a partir do token JWT.

Incluir o routes/api/index no app.js em "/api"

Agora que temos nossas rotas da API prontas em routes/api, vamos adicionar ela na rota raiz da aplicação no arquivo app.js.

Para isso adicione no app.js o seguintes comando na sessão de "requires":

```
var apiRouter = require('./routes/api');
```

Depois adicione (antes do "/auth") o seguinte comando:

```
app.use('/api', apiRouter);
```

O arquivo app.js deve ficar assim no final:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var cors = require('cors');

// importando o cookie-session
var cookieSession = require('cookie-session')

var indexRouter = require('./routes/index');

// importando rotas de autenticação
var authRouter = require('./routes/auth');

// importando rotas da api
var apiRouter = require('./routes/api');

// importando o requireAuth middleware
const requireAuth = require('./middlewares/requireAuth');

var app = express();

app.use(cors());

// usando o cookie-session
app.use(cookieSession({
  name: 'pettopstore_session', // nome do cookie no navegador
  keys: ['chave_secreta_para_criptografia'], // chave necessária
  para criptografia
  maxAge: 24 * 60 * 60 * 1000, // 24 horas de duração da sessão
  (usuário logado)
})));

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
```

```
app.use(express.static(path.join(__dirname, 'public')));

// usando rotas da API
app.use('/api', apiRouter);

// usando rotas de autenticação
app.use('/auth', authRouter);
// aplica o requireAuth middleware na raiz do site
app.use('/', [requireAuth], indexRouter);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err :
  {});

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

Testando a API

Utilize qualquer software para testar APIs, como o Insomnia, por exemplo. Segue alguns resultados:

POST http://localhost:3000/api/auth/client/registration Send

JSON Auth Query Header 1 Docs

```
1 {
2   "name": "Isaac Franco",
3   "email": "isaacfranco@imd.ufrn.br",
4   "password": "123456"
5 }
```

Beautify JSON

200 OK 117 ms 58 B 7 Days Ago

Preview Header 7 Cookie Timeline

```
1 {
2   "message": "Cliente registrado com sucesso",
3   "clienteID": 5
4 }
```

Registro de cliente

POST http://localhost:3000/api/auth/client/sign_in Send

JSON Auth Query Header 1 Docs

```
1 {
2   "email": "isaacfranco@imd.ufrn.br",
3   "password": "123456"
4 }
```

Beautify JSON

200 OK 124 ms 485 B Just Now

Preview Header 8 Cookie Timeline

```
1 {
2   "client": {
3     "id": 5,
4     "email": "isaacfranco@imd.ufrn.br",
5     "password":
6       "$2b$10$Aet18i/3g0jCfTWnH8rXmekhXzWk8Jfm0ni1eJTnRZcFsFqudu4py",
7     "name": "Isaac Franco",
8     "token":
9       "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjbG1bnQiOi0nsiaWQiOi0jUsImVtYWlsIjoiaXNhYWNmcmFuY29AaW1kLnVmcm4uYnIiLCJwYXNzd29yZCI6IiQyYiQxMCRBZXRsOGkvM2dPakNmVFduSDhyWG1la2hYe1drOEpmBTBuaTF1S1RuUlpjRnNGcXVkdTRweSIIm5hbWUiOiJJc2FhYyBGcmFuY28ifSwiaWF0IjoxNjMwMjYxNDQwLCJleHAiOi0jE2Mza0NjQyNDB9.S5t389o7jJmE6G4BxCx9I1_G5QwaPMwp1Q5w18pyAW0"
10   }
11 }
```

Login de cliente (repare o

token JWT retornado junto dos dados do cliente)

The screenshot shows a REST client interface with the following details:

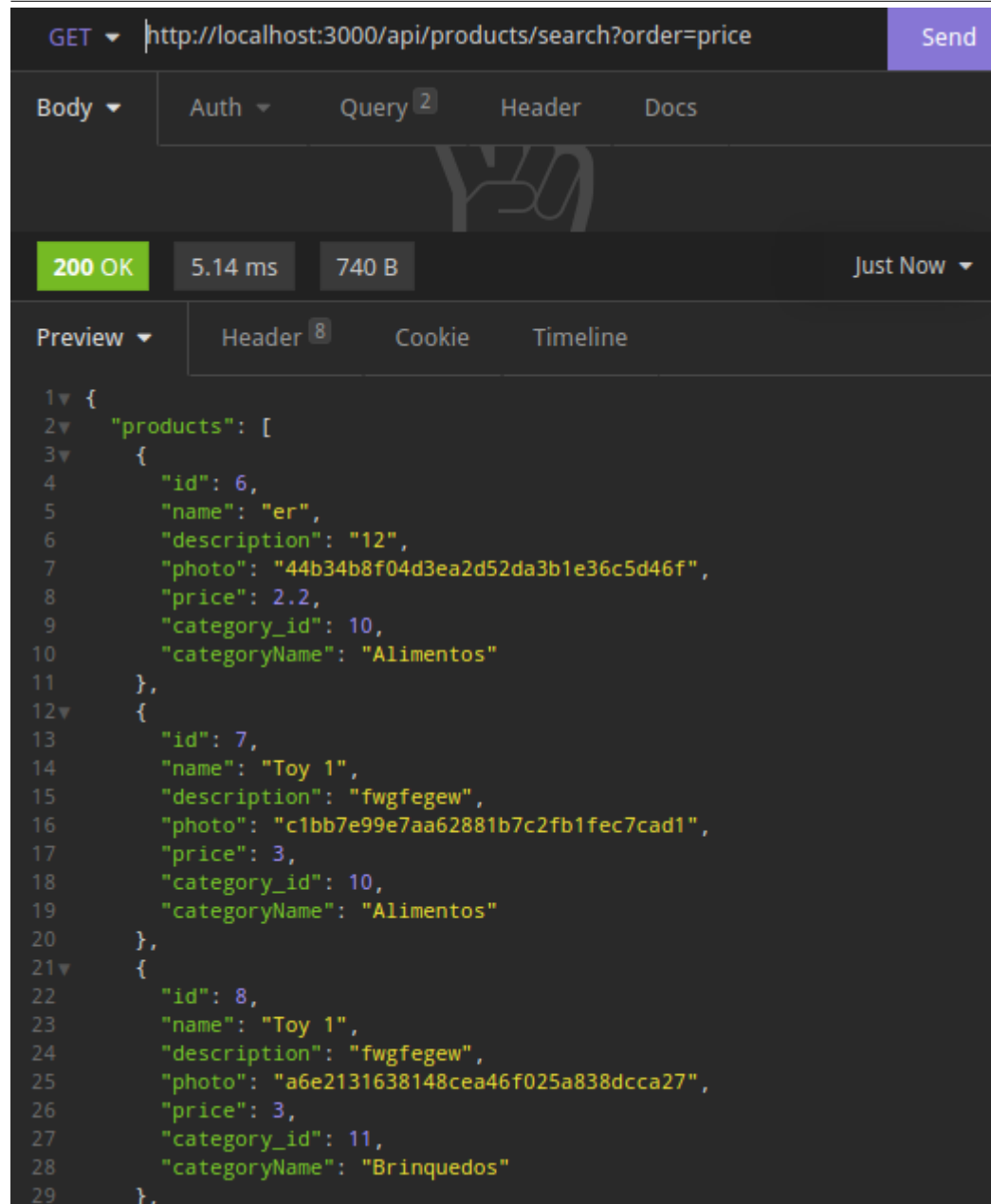
- Method:** POST
- URL:** http://localhost:3000/api/auth/employee/sign_in
- Request Body (JSON):**

```
{
  "email": "maria@pettopstore.com",
  "password": "123456"
}
```
- Status:** 200 OK
- Time:** 110 ms
- Size:** 513 B
- Response Body (JSON):**

```
{
  "employee": {
    "id": 1,
    "email": "maria@pettopstore.com",
    "password": "$2b$10$vaJcMgt2Dg7BC1J4fWcP0uV3/fo7tk0Q5Xsoo9K/y7vxnDqf9FcYS",
    "name": "Maria Admin",
    "is_admin": 1,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bXBsb311ZSI6eyJpZCI6MSwiZW1haWwiOiJtYXJpYUBwZXROb3BzdG9yZS5jb20iLCJwYXNzd29yZCI6IiQyYiQxMCR2YUpDbUdOMkRnNOJDMUo0ZldjUDB1VjMvZm83dGswUTVYc29vOUsvTd2eG5EcWY5RmNZUyIsIm5hbWUiOiJNYXJpYSBBZG1pb20iLCJpdiI6Im1zX2FkbWluIjoxfSwiaWF0IjoxNjM3MDM3NzkzLCJleHAiOiJlMzAyMTA1OTN9.hdJ7q5KeTk307hBSM7HASwgt1rfZCCom-dbZL45SstY"
  }
}
```

Login de funcionário

(repare o token JET retornado junto aos dados do funcionário)



GET Send

Body Auth Query 2 Header Docs

200 OK 5.14 ms 740 B Just Now

Preview Header 8 Cookie Timeline

```
1 {
2   "products": [
3     {
4       "id": 6,
5       "name": "er",
6       "description": "12",
7       "photo": "44b34b8f04d3ea2d52da3b1e36c5d46f",
8       "price": 2.2,
9       "category_id": 10,
10      "categoryName": "Alimentos"
11    },
12    {
13      "id": 7,
14      "name": "Toy 1",
15      "description": "fwgfegew",
16      "photo": "c1bb7e99e7aa62881b7c2fb1fec7cad1",
17      "price": 3,
18      "category_id": 10,
19      "categoryName": "Alimentos"
20    },
21    {
22      "id": 8,
23      "name": "Toy 1",
24      "description": "fwgfegew",
25      "photo": "a6e2131638148cea46f025a838dcca27",
26      "price": 3,
27      "category_id": 11,
28      "categoryName": "Brinquedos"
29    },
30  ]
31 }
```

Busca de produtos
ordenados por preço

GET ▼ http://localhost:3000/api/clients Send

Body ▼ Bearer ▼ Query Header Docs

TOKEN eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbXBsb3llZSI6eyJpZCI6MSwiZV

200 OK 4.78 ms 297 B Just Now ▼

Preview ▼ Header 8 Cookie Timeline

```
1 {
2   "clients": [
3     {
4       "id": 5,
5       "name": "Marcio Ferreira",
6       "email": "marcio@marcionet.com"
7     },
8     {
9       "id": 6,
10      "name": "Joaquim Pedrosa",
11      "email": "joaquim@gmail.com"
12     },
13    {
14      "id": 7,
15      "name": "Mario",
16      "email": "mario@gmail.com"
17    },
18    {
19      "id": 8,
20      "name": "Amanda",
21      "email": "amanda@gmail.com"
22    },
23    {
24      "id": 9,
25      "name": "Joaquim",
26      "email": "joaquim2@gmail.com"
27    }
28  ]
29 }
```

Lista de clientes.

Requer que seja passado o Token JWT retornado em um sign_in de um funcionário para funcionar.

GET ▼ http://localhost:3000/api/sales/12 Send

JSON ▼ Bearer ▼ Query Header 1 Docs

TOKEN eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbXBsb3llZSI6eyJpZCI6MSwiZV

PREFIX ⓘ

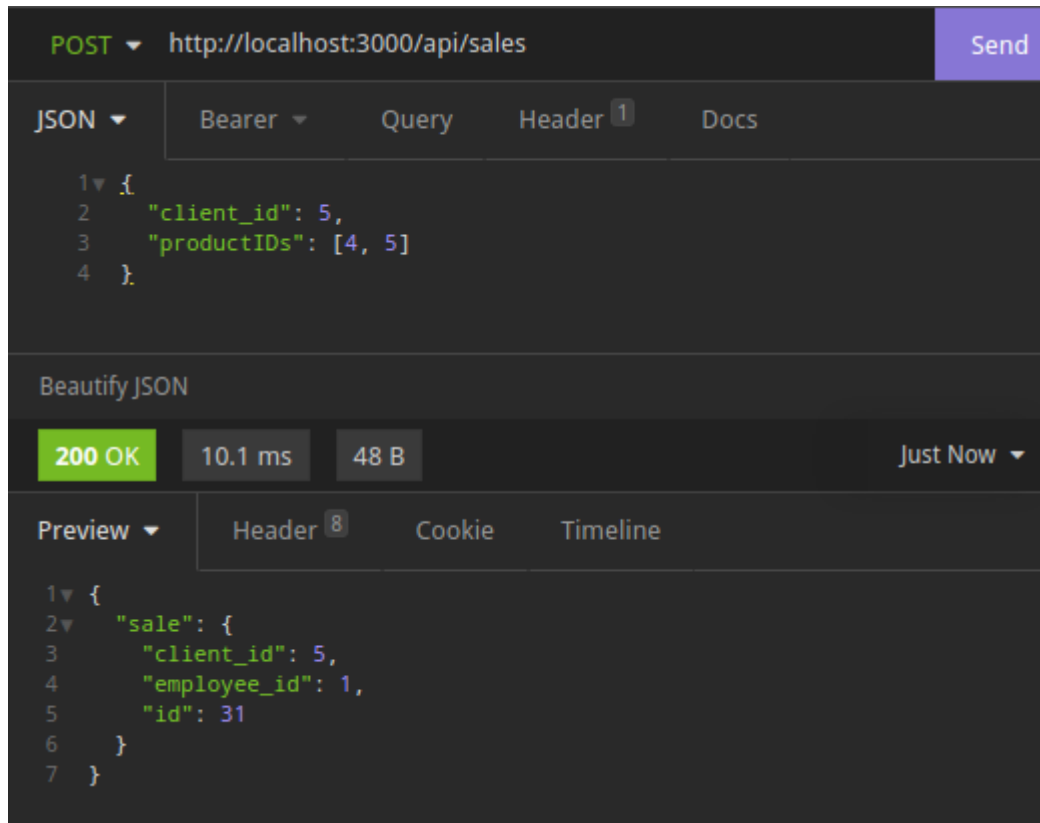
ENABLED ☒

200 OK 12.5 ms 392 B Just Now ▼

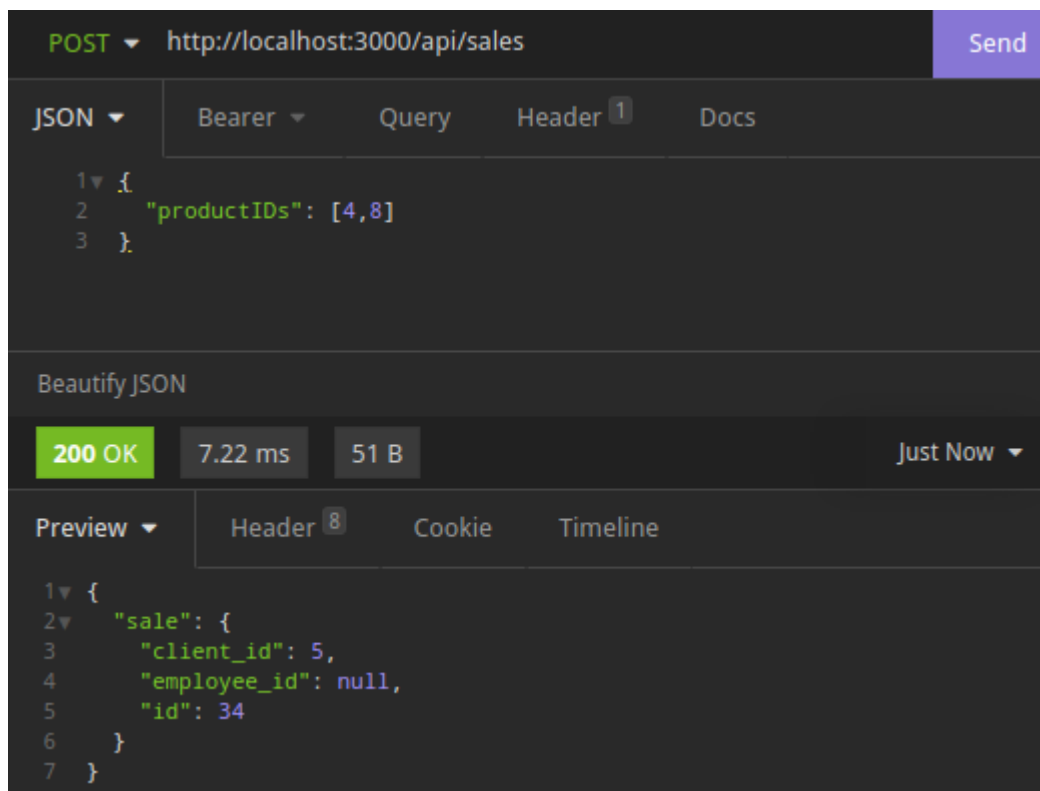
Preview ▼ Header 8 Cookie Timeline

```
1 {
2   "sale": {
3     "id": 12,
4     "client_id": 5,
5     "employee_id": 1,
6     "items": [
7       {
8         "id": 3,
9         "sale_id": 12,
10        "product_id": 4,
11        "product": {
12          "id": 4,
13          "name": "Balança de ração",
14          "description": "",
15          "photo": "7c18019abf3666b04351f6ef2852d339",
16          "price": 44,
17          "category_id": 10
18        }
19      },
20      {
21        "id": 4,
22        "sale_id": 12,
23        "product_id": 5,
24        "product": {
25          "id": 5,
26          "name": "Coleira",
27          "description": "",
28          "photo": "2589017ccadb36139db8335c6ee6f540",
29          "price": 30,
30          "category_id": 10
31        }
32      }
33    ]
34  }
35 }
```

Detalhes de uma venda (requer JWT de um funcionário)



Criação de uma venda (sale) por um funcionário, passando o id do cliente e a lista de ids de produtos.



Criação de uma venda através de uma compra de um cliente (passando o JWT do cliente) e somente a lista de IDs de produtos.

Repare que nessa última imagem a venda fica sem employee_id, já que a compra está sendo feita diretamente pelo cliente (validado pelo seu token JWT).

Adicionar lista de vendas no Admin

Já que nossa API está pronta no projeto do admin, vamos criar como bônus uma listagem de vendas no admin.

Para isso você precisa inicialmente criar o arquivo `routes/sales.js` com o conteúdo:

```
const express = require('express');
const router = express.Router();
const knexConfig = require('../knexfile');
const knex = require('knex')(knexConfig);

router.get('/report', async(req, res) => {
  // obtém todas as vendas do banco de dados
  const sales = await knex.table('sales');

  // para cada venda obtida
  for (const sale of sales) {
    // adiciona os itens da venda no objeto
    sale.items = await knex.table('items').where('sale_id', '=',
sale.id);
    if (sale.employee_id !== null) {
      // caso existe um funcionário, buscar ele do banco e adiciona
no objeto
      sale.employee = await knex.table('employees').where('id', '=',
sale.employee_id).first();
    }
    if (sale.client_id !== null) {
      // caso existe um cliente, buscar ele do banco e adiciona no
objeto
      sale.client = await knex.table('clients').where('id', '=',
sale.client_id).first();
    }

    for (const item of sale.items) {
      // para cada item da venda adicionar no item o produto que ele
se referencia
      item.product = await knex.table('products').where('id', '=',
item.product_id).first();
    }
  }

  // renderizar o relatório de vendas passando a lista de vendas
(sales)
```

```
res.render('sales/report', { sales });
});

module.exports = router;
```

Existe somente uma rota nesse arquivo `"/report"` que vai buscar todas as vendas(sales) e adicionar nelas as suas relações com clients, employee, items e products através de outras consultas. Repare que os comentários no código retalham o processo.

Depois adicione em `routes/index.js` essa nova rota de sales, deixando o arquivo assim:

```
var express = require('express');
var router = express.Router();
var productsRouter = require('./products');
var salesRouter = require('./sales');

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', {
    title: 'Pet Top Store',
    employee: res.locals.employee
  });
});

router.use('/products', productsRouter);
router.use('/sales', salesRouter);

module.exports = router;
```

Crie a pasta `views/sales` e nela adicione o arquivo `report.ejs` com o seguinte conteúdo:

```
<h1>Relatório de vendas</h1>
<a href="/">Voltar</a>
<% for (const sale of sales) { %>
<div style="border: 1px solid gray; margin-bottom: 10px">
  <% if (sale.employee_id) { %>
    Funcionário: <%= sale.employee.name %><br/>
  <% } %>
  <% if (sale.client_id) { %>
    Cliente: <%= sale.client.name %><br/>
  <% } %>
}
```

```
<% } %>
Items:
<ul>
  <% for (const item of sale.items) { %>
    <li><%= item.product.name %></li>
  <% } %>
</ul>
</div>
<% } %>
```

Crie um link para a listagem de vendas que criamos em views/index.ejs, deixando arquivo assim:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
    <p>Logado como <%= employee.name %></p>
    <p>
      <a href="/auth/sign_out">Sair do sistema</a>
    </p>
    <a>
      <a href="/products">Gerenciar produtos</a>
    </a>
    |
    <a>
      <a href="/sales/report">Listagem de vendas</a>
    </a>
  </body>
</html>
```

Pronto. Agora você, logado como um administrador no admin pode ir para "<http://localhost:3000/sales/report>" (ou clicar em "Listagem de vendas") para ver uma simples listagem de todas as vendas criadas pela API, com o funcionário que a realizou, o cliente e os produtos da venda. A listagem deve ficar similar a essa:

Relatório de vendas

[Voltar](#)

Funcionário: Maria Admin

Cliente: Marcio Ferreira

Items:

- Balança de ração
- Coleira

Funcionário: Maria Admin

Cliente: Marcio Ferreira

Items:

- Balança de ração
- Coleira

Funcionário: João Vendedor

Cliente: Marcio Ferreira

Items:

- Coleira
- Toy 1
- Toy 1
- Coleira
- Balança de ração

Funcionário: João Vendedor

Cliente: Marcio Ferreira

Items:

- Coleira
- Toy 1
- Toy 1
- Coleira
- Balança de ração

Conclusão

Finalizamos o sistema do administrador (admin) com a adição dos seguintes recursos:

- Adicionado seeds de categorias
- Adicionado categoria no form de adicionar produtos
- Criação da API com:
 - json web tokens
 - index
 - auth

- products
- clients
- sales
- Listagem de vendas no Admin

Sim, é verdade que o sistema do admin pode melhorar bastante, tanto visualmente como com mais funcionalidades.

Como sugestão você pode adicionar uma biblioteca de componentes UI como o Bootstrap ou o de repente usar o TailwindCSS para estilizar seu sistema de administração, melhorando sua usabilidade. Outras sugestões é adicionar mais validações de campos nas ações e mensagens de erros personalizadas para o usuário.

O que pretendemos por enquanto é ter uma base simples para se criar os outros sistemas (PDV e Loja on-line) e deixar o código puro e para que seja fácil de alterar de acordo com o seu gosto e usando as tecnologias que desejar.

Vamos nas próximas aulas criar o sistema de PDV (Ponto de venda), para permitir que funcionários entrem nesse sistema, criem clientes e vendas como se fosse em uma loja física. Vamos também mais a frente criar uma loja virtual simples para o cliente poder realizar compras de forma autônoma. Esses dois sistemas que criaremos irão utilizar tecnologias diferentes e usarão exclusivamente a nossa API que criamos como fonte de dados e não farão conexão direta com o banco de dados.

Baixe o projeto admin+API da PetTopStore

Projeto final do PetTopStore(Admin + API): [Clique aqui para download](#)