

DESCUBRA O PODER DO NOVO VUE

VUE 3

NA PRÁTICA

BY DANIEL SCHMITZ
2024

Vue 3 na Prática

Descubra o poder do Vue 3

Daniel Schmitz

Esse livro está à venda em <http://leanpub.com/book-vue-br>

Essa versão foi publicada em 2024-03-12



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2024 Daniel Schmitz

Conteúdo

1. Vue3 na Prática	1
1.1 Código Fonte	1
1.2 Suporte	1
1.3 Preparação para o Vue	1
Instalação do Node	1
Instalação do Git	2
Instalação do Visual Studio Code	2
2. O Projeto Vue 3	4
2.1 Antes de Começar	4
2.2 Criando o Projeto	4
2.3 Limpando o Código	6
2.4 Git	7
2.5 Instalando o PicoCss	7
2.6 Servidor Backend	10
3. Categorias	14
3.1 Router	14
3.2 Exibindo uma Listagem de Categorias	17
3.3 Obtendo Dados	19
O Que Há de Novo Vue 3: Ref	20
Utilizando OnMounted()	20
3.4 Otimizando o Acesso Ao Servidor	22
Axios	23
Criando o Service de Categorias	23
3.5 Refatorando o ListView, Acessando o Service	24
3.6 Criando uma Nova Categoria	25
Criando o Componente e a Rota	27
Criando o Formulário	29

CONTEÚDO

	Persistindo Dados	33
	Exibindo um Feedback Ao Usuário	35
	Simulando uma Conexão Lenta	37
	Feedback Ao Salvar uma Categoria	38
	Validação de Dados	40
3.7	Editando uma Categoria	47
	Configurando o Service	47
	Configurando o Router	47
	Obtendo uma Categoria Pelo Id	49
	Formulário para Editar Categorias	51
3.8	Removendo uma Categoria	52
4.	Produtos	55
4.1	Service	55
	Estrutura de Arquivos	56
4.2	Router	58
4.3	Exibindo Dados em Forma Tabular	60
4.4	Corrigindo o CategoryId	62
4.5	Formulário para Criar um Novo Produto	65
	Adicionando o Campo Category	66
4.6	Editando um Produto	69
4.7	Formulário para Editar o Produto	71
4.8	Remover um Produto	75
5.	Estoque de Produtos	76
5.1	Service	76
5.2	Adicionando Produtos Ao Estoque	77
5.3	Router	78
5.4	Formulário para Adicionar um Novo Produto Ao Estoque	79
5.5	Exibindo a Lista de Produtos no Estoque	84
5.6	Ordenando a Lista de Stock	86
5.7	Exibindo Itens de Estoque Perto da Validade	87
5.8	Incluindo o Nome do Produto	89
5.9	Editando um Produto do Estoque	90
6.	Considerações Finais	94

1. Vue3 na Prática

A proposta principal deste livro é apresentar o Vue 3 e a nova forma como o Vue trabalha através do que chamamos de `Composition api`. Iniciaremos o aprendizado de Vue apresentando totalmente focados na prática, exibindo aos poucos os conceitos sobre Vue, de forma gradual. Nosso objetivo não é mostrar um grande capítulo sobre a parte teórica do Vue. Para isso temos uma excelente documentação localizada em <https://vuejs.org/guide/introduction.html>

1.1 Código Fonte

O código fonte deste livro encontra-se [nesta url](#)¹

1.2 Suporte

Você pode abrir uma issue [nesta url](#)²

Fique a vontade em comentar sobre possíveis erros encontrados neste livro, inclusive erros de tradução para o seu idioma.

1.3 Preparação para o Vue

Antes de iniciarmos o aprendizado sobre o Vue, é necessário configurar o ambiente de trabalho com alguns programas. O Vue utiliza a linguagem javascript e para que possamos executá-la no computador é necessário instalar o Node.

Instalação do Node

Para instalar o Node, acesse <https://nodejs.org/en> e clique na versão LTS atual, faça o download e instale. O Node será instalado juntamente com o NPM, que é um gerenciador de pacotes, e o NPX que é um executor de pacotes.

¹<https://github.com/danielschmitz/kitchen-app-vue3-book>

²<https://github.com/danielschmitz/kitchen-app-vue3-book/issues>

Instalação do Git

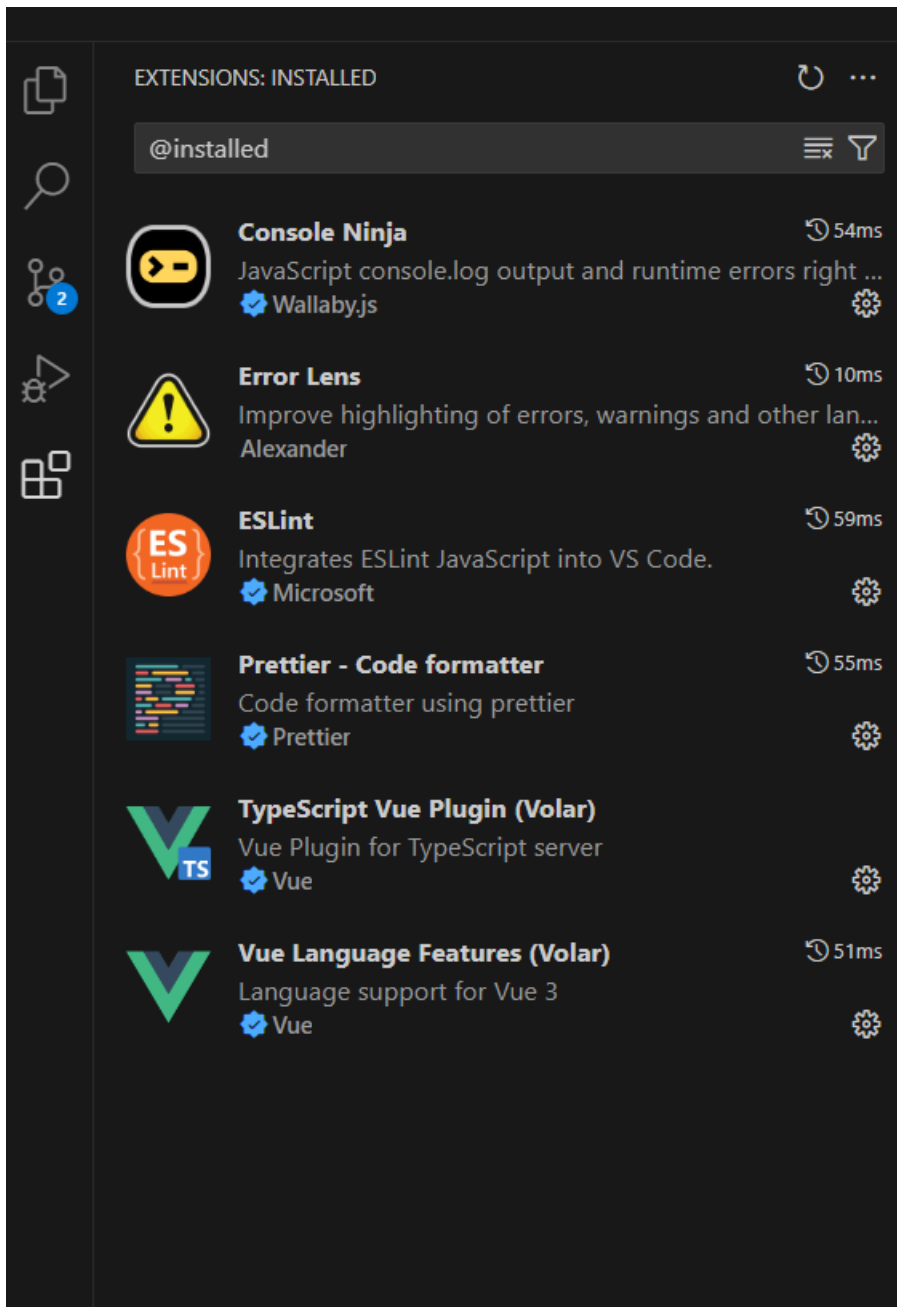
O Git é um sistema de controle de arquivos amplamente conhecido por ser utilizado em todos os códigos modernos. Neste livro iremos supor que você conhece o git, e possui uma conta no github para enviar seus códigos. O conhecimento sobre o git não é um requisito para o entendimento sobre Vue, mas é vital para qualquer desenvolvedor de software.

Para instalar o Git no Windows visite <https://git-scm.com/downloads>. Após a instalação, teremos acesso ao Git Bash que é um terminal que iremos utilizar neste livro. Iremos utilizar o Git Bash ao invés do Windows PowerShell. Se você está utilizando Linux ou MAC, fique a vontade em utilizar o terminal de sua escolha.

Instalação do Visual Studio Code

O nosso editor de códigos preferido é o Visual Studio Code, que pode ser instalado nesta url <https://code.visualstudio.com/>. O vscode possui suporte a centenas de linguagens e para o nosso livro, utilizaremos as seguintes extensões:

- ESLint: Esta extensão realiza algumas verificações no seu código fonte, buscando erros de formatação, estética etc. Por exemplo, se você configurar que o nome de um método do seu programa deve começar com letra minúscula, é o eslint que irá verificar isso, apresentando a mensagem de erro e sugerindo a correção.
- Error Lens: Displays the error, if any, at the end of the line.
- Console Ninja: O console ninja é um utilitário que exibe as mensagens do `console.log` diretamente no vscode, sem a necessidade que você verifique esta mensagem no navegador.
- Vue Language Features (Volar): A extensão principal do Vue, para que toda a integração do Vue com o vscode funcione.
- TypeScript Vue Plugin (Volar): Esta extensão adicione suporte do TypeScript ao Vue. Neste livro iremos utilizar o TypeScript em nossos códigos.
- Prettier: É um formatador de código, deixando todos os seus arquivos com uma formatação padronizada.



2. O Projeto Vue 3

Nosso projeto a ser criado em Vue é chamado de `KitchenStock`. Ele será utilizado para que possamos controlar a dispensa de uma cozinha, armazenando produtos e a validade de cada um deles. Desta forma, poderemos rastrear a validade de cada produto cadastrado.

2.1 Antes de Começar

Precisamos compreender que um projeto web possui diversas tecnologias envolvidas que vão além do Vue. Por exemplo, precisamos de um servidor para informações em um banco de dados, precisamos utilizar uma tecnologia para criar elementos visuais na tela etc. Para cada uma destas tecnologias complexas, iremos tentar utilizar neste livro algo simples.

Para simular um servidor web, iremos utilizar o `json-server`. Com ele poderemos ter acesso a uma simples API para criar e editar dados, com estes dados estando no formato json. Em um ambiente real, possivelmente este servidor seria em Java ou Node, e utilizando um banco de dados real como o PostgreSQL ou MySQL.

Para exibir elementos visuais na tela, podemos escolher um dos milhares de frameworks CSS existentes na web. O mais completo e atualmente mais utilizado deles seria o Tailwind. Para o nosso projeto, iremos utilizar o PicoCSS, um framework minimalista css muito versátil e que utilizo em todos os meus projetos pessoais.

2.2 Criando o Projeto

Para criar um projeto Vue, abra o terminal (Abra o GitBash, caso tenha instalado o Git, ou o Windows PowerShell) e digite o seguinte comando:

```
1 npm create vue@latest
```

Este comando irá iniciar um assistente onde você irá escolher algumas funcionalidades, tais como Typescript, JSX, etc. Escolha conforme a figura a seguir e siga as instruções fornecidas pelo assistente.


```
MINGW64/c/Users/dps
dps@daniel MINGW64 ~
$ npm create vue@latest
Need to install the following packages:
create-vue@3.9.2
Ok to proceed? (y) y

Vue.js - The Progressive JavaScript Framework

√ Project name: ... kitchen-app
√ Add TypeScript? ... No / Yes
√ Add JSX Support? ... No / Yes
√ Add Vue Router for Single Page Application development? ... No / Yes
√ Add Pinia for state management? ... No / Yes
√ Add Vitest for Unit Testing? ... No / Yes
√ Add an End-to-End Testing Solution? » No
√ Add ESLint for code quality? ... No / Yes
√ Add Prettier for code formatting? ... No / Yes

Scaffolding project in C:\Users\dps\kitchen-app...

Done. Now run:

  cd kitchen-app
  npm install
  npm run format
  npm run dev

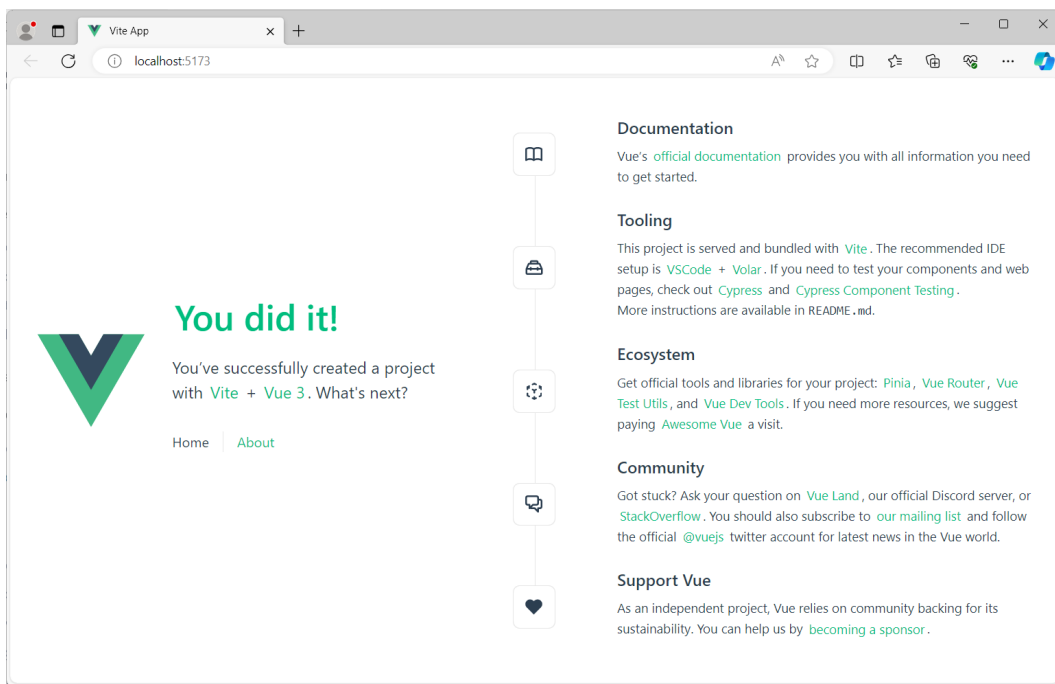
npm notice
npm notice New minor version of npm available! 10.2.4 -> 10.4.0
```

Após criar o projeto, acesse-o e execute o comando `npm install` para instalar os pacotes do projeto. Depois execute `npm run format` para formatar os arquivos de acordo com o prettier e `npm run dev` para iniciar o projeto. O projeto será executado e teremos a seguinte saída:

```
VITE v5.1.1 ready in 2595 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

Abra o navegador e acesse o endereço fornecido, teremos o seguinte resultado:



2.3 Limpando o Código

Vamos limpar todo o código que o assistente gerou para que possamos iniciar o desenvolvimento da aplicação. Abra o arquivo `src/App.vue` e remova todo o código que está na tag `template`, deixando apenas a tag `<RouterView />` e remova também a tag `<style>`. O resultado final do `App.vue` é:

```
1 <script setup lang="ts">
2 import { RouterView } from 'vue-router'
3 </script>
4
5 <template>
6   <RouterView />
7 </template>
```

Agora precisamos limpar o componente `src/view/HomeView.vue` e deixe somente uma mensagem de Hello World:

```
1 <script setup lang="ts">
2
3 </script>
4
5 <template>
6   Hello World
7 </template>
```

Se voltarmos ao navegador, veremos uma mensagem de Hello World no centro da tela. Precisamos também limpar as definições de estilo no diretório `src/assets/`. Exclua o arquivo `base.css` e no arquivo `main.css` remova todo o seu conteúdo.

Apos remover o conteúdo de `src/assets/main.css`, teremos no navegador a mensagem de Hello World completamente sem estilo.

2.4 Git

É recomendável iniciar o git e versionar os arquivos a partir desse momento. Vamos executar os seguintes comandos:

```
1 cd kitchen-app
2 git init
3 git branch -m "main"
4 git add .
5 git commit -m "Initial Files"
```

Iniciamos o git com o comando `git init` e então mudamos o nome do branch padrão para `main`. Após isso adicionamos todos os arquivos no stage, através do comando `git add .` e realizamos o commit através do comando `git commit`, fornecendo a mensagem através da opção `-m`.

2.5 Instalando o PicoCss

O PicoCss é uma mini biblioteca com alguns componentes Css básicos, mas totalmente funcionais. Para instalá-lo, utilize o seguinte comando:

```
1 npm install @picocss/pico
```

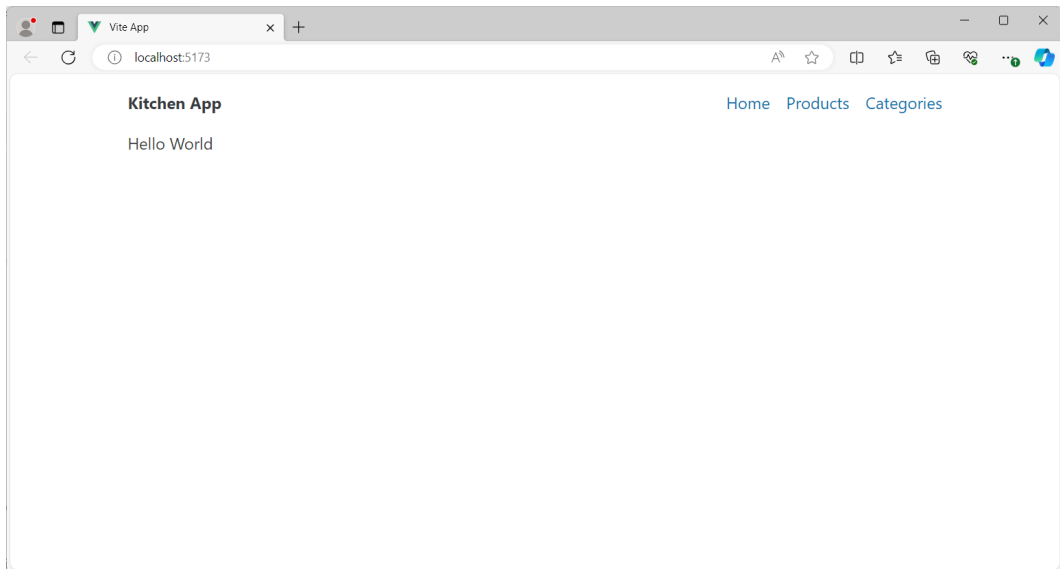
Após instalar via npm, adicione no arquivo `src/assets/main.css` o seguinte import:

```
1 @import url(../../node_modules/@picocss/pico/css/pico.min.css);
```

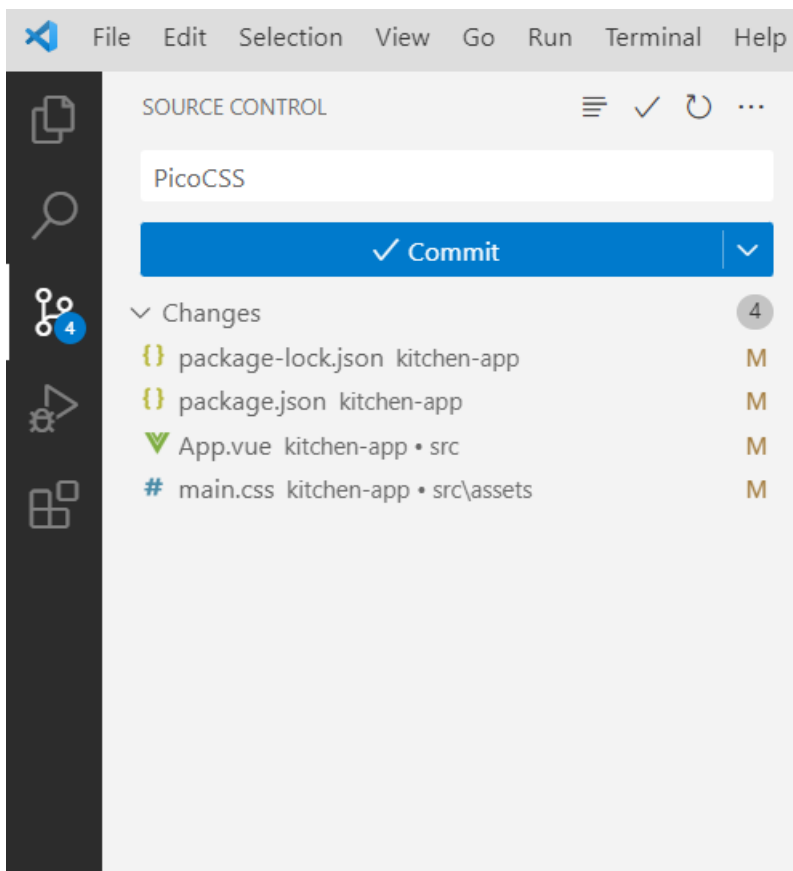
Agora no arquivo `src/App.vue` adicione o seguinte código:

```
1 <script setup lang="ts">
2 import { RouterView } from 'vue-router'
3 </script>
4
5 <template>
6   <div class="container">
7     <nav>
8       <ul>
9         <li><strong>Kitchen App</strong></li>
10      </ul>
11      <ul>
12        <li><a href="#">Home</a></li>
13        <li><a href="#">Products</a></li>
14        <li><a href="#">Categories</a></li>
15      </ul>
16    </nav>
17    <RouterView />
18  </div>
19 </template>
```

Após o `<template>` adicionamos uma `div` com o `class container`. Esta classe `css` já é do PicoCss, incluindo o `<nav>` logo após o `div`. Perceba que, utilizando apenas tags `html`, conseguimos criar uma aplicação estilizada, semelhante a figura a seguir:



Com o PicoCSS instalado, podemos realizar o commit as alterações pelo Git. Agora, pelo Visual Studio Code. Acesse a aba Git do pico, adicione um comentário pertinente a modificação que está sendo feita no projeto, e clique no botão `Commit`:



Após realizar o commit, você provavelmente verá o botão `Publish Branch`. Isso fará com que o seu código seja sincronizado com o GitHub. Se você tiver uma conta no GitHub e fizer o login pelo Visual Studio Code, poderá sincronizar seu projeto diretamente pelo `vscode`.

2.6 Servidor Backend

O Desenvolvimento Web moderno tem como principal filosofia a separação entre Backend e Frontend, onde cada um deles tem suas responsabilidades. Enquanto que o Frontend é responsável pela exibição de informações para o usuário, o Backend é responsável pela persistência de dados, autenticação, envio de emails etc.

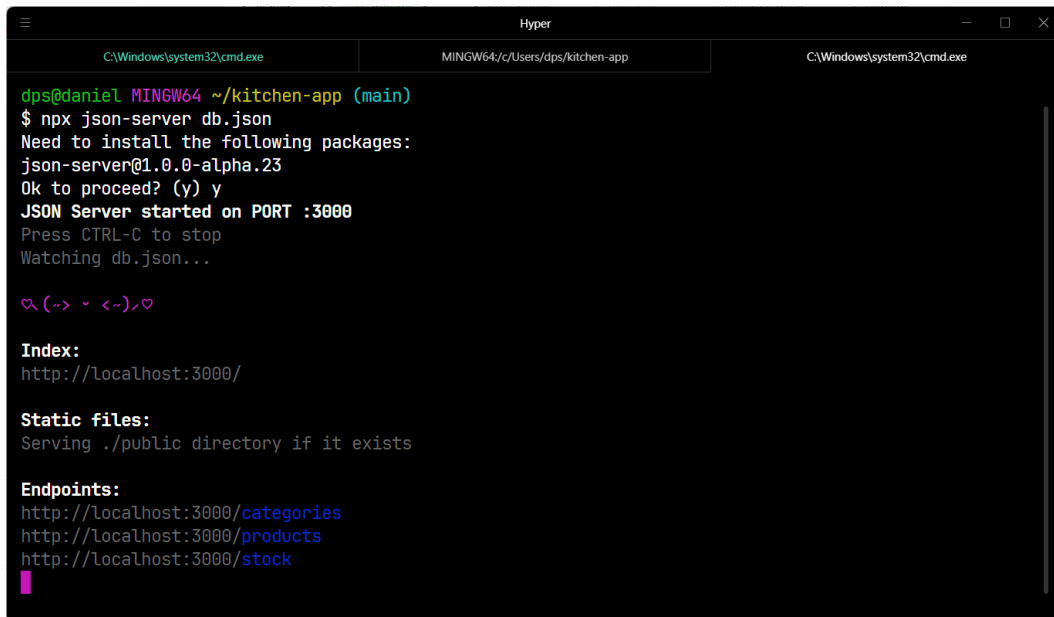
O Vue é um framework exclusivo para o Frontend. Ele não é responsável em persistir dados ou autenticar usuários. Para isso, podemos ter um servidor em Java, Node, Dotnet etc. Como

o foco deste livro é o Vue, vamos criar um servidor web fake que irá persistir os dados da aplicação em um arquivo json.

Baixe o arquivo `db.json` [desta url](#)¹ e salve na raiz do projeto vue. O próximo passo é executar o seguinte comando:

```
1 npx json-server db.json
```

A resposta deste comando é semelhante a figura a seguir:



```
Hyper
C:\Windows\system32\cmd.exe  MINGW64~/c/Users/dps/kitchen-app  C:\Windows\system32\cmd.exe

dps@daniel MINGW64 ~/kitchen-app (main)
$ npx json-server db.json
Need to install the following packages:
json-server@1.0.0-alpha.23
Ok to proceed? (y) y
JSON Server started on PORT :3000
Press CTRL-C to stop
Watching db.json...

💖 (~> ~ <~)💖

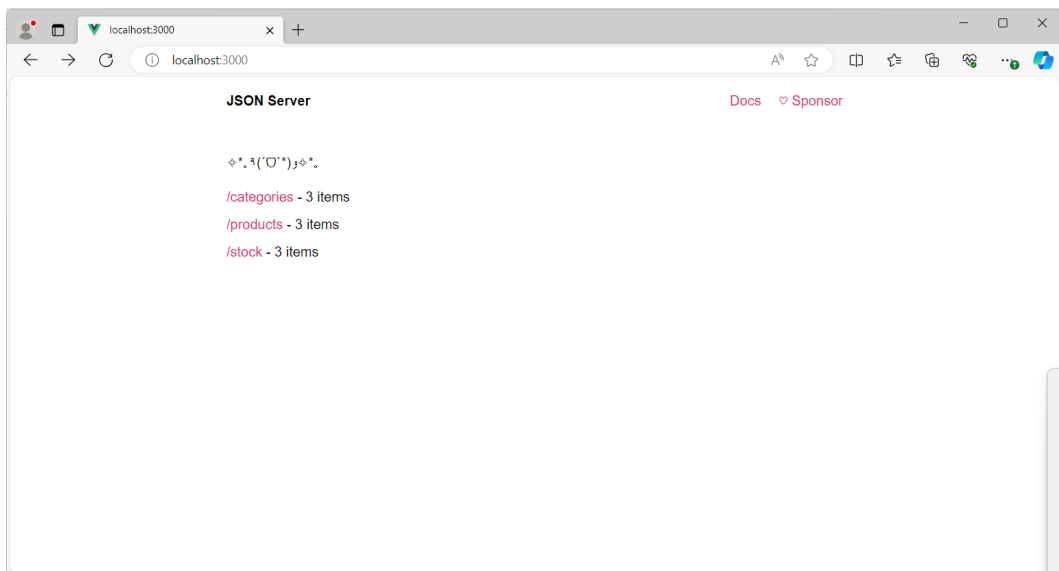
Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

Endpoints:
http://localhost:3000/categories
http://localhost:3000/products
http://localhost:3000/stock
```

O servidor Backend está sendo executado na porta 3000. Ao acessar `http://localhost:3000` temos a seguinte resposta:

¹<https://gist.github.com/danielschmitz/f48f70a00f0959506ec0c4ffd66d8440>



A partir deste momento, temos o servidor backend funcionando, e poderemos acessá-lo pelo Vue para criar a aplicação.

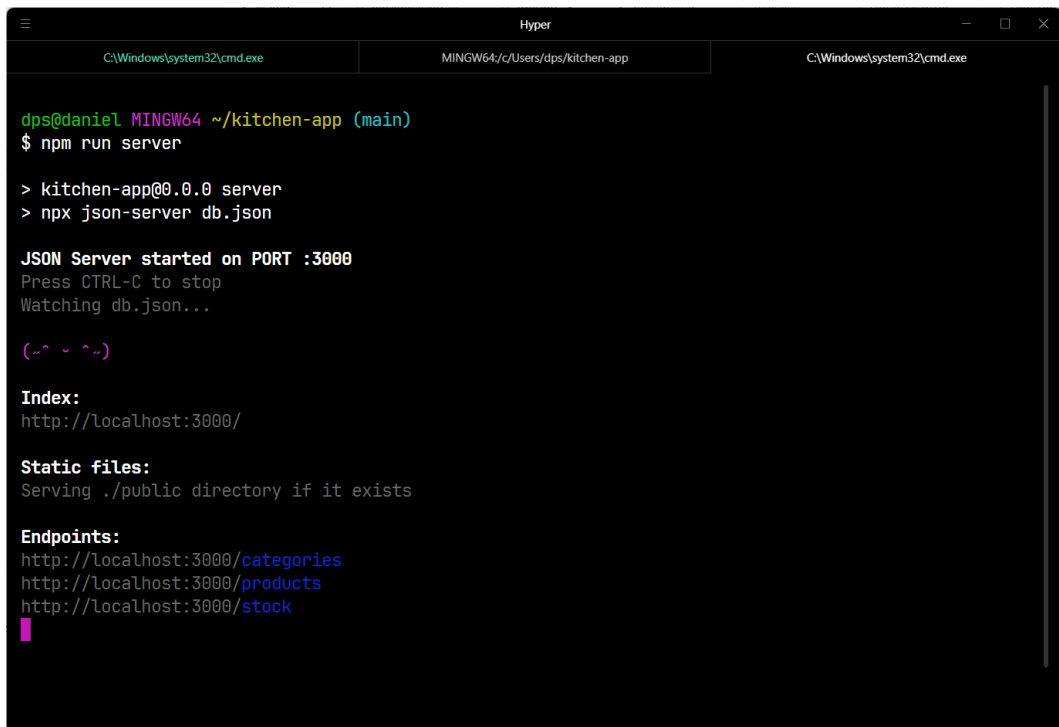
Se você quer aprender mais sobre como é a comunicação entre o Frontend e o Backend, veja este [artigo sobre REST](https://en.wikipedia.org/wiki/REST)^a

^a<https://en.wikipedia.org/wiki/REST>

Para finalizar, podemos criar um atalho para o comando `npx json-server db.json`, para que possa ser executado de uma forma mais fácil. Abra o arquivo `package.json` e adicione a seguinte linha:


```
1 {
2   "name": "kitchen-app",
3   "version": "0.0.0",
4   "private": true,
5   "type": "module",
6   "scripts": {
7     "dev": "vite",
8     "server": "npx json-server db.json", <<<<HERE
9     "build": "run-p type-check \"build-only {@}\" --",
10    "preview": "vite preview",
11    "build-only": "vite build",
12    ...
13    ...
14    ...
15    ...
```

Adicionamos um novo script chamado `server`. Com isso, podemos iniciar o servidor através do comando `npm run server`:



```
Hyper
C:\Windows\system32\cmd.exe  MINGW64/c/Users/dps/kitchen-app  C:\Windows\system32\cmd.exe

dps@daniel MINGW64 ~/kitchen-app (main)
$ npm run server

> kitchen-app@0.0.0 server
> npx json-server db.json

JSON Server started on PORT :3000
Press CTRL-C to stop
Watching db.json...

(^.^ ^.^)

Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

Endpoints:
http://localhost:3000/categories
http://localhost:3000/products
http://localhost:3000/stock
```

3. Categorias

Neste capítulo iremos criar a tela de categorias, onde será possível editar as categorias dos produtos. Apesar das categorias terem apenas dois campos, nome e descrição, iremos aprender diversos conceitos importantes que serão utilizados em todas as telas do sistema.

3.1 Router

O Router é utilizado para carregar as mais diferentes views da aplicação, de acordo com a url. Por exemplo, ao clicar no link Categories, queremos acessar a url `/categories` e carregar o a view `Categories`.

Primeiro, crie a pasta `views/categories` e o componente `view/categories/IndexView.vue`:

```
1 <script setup lang="ts">
2
3
4 </script>
5 <template>
6   <article>
7     <header>Categories</header>
8     Content...
9   </article>
10 </template>
```

Inicialmente este componente não tem muita informação. Apenas o título, na tag `header` do `article`. É necessário agora configurar o Router para que possamos acessá-lo. Edite o arquivo `src/router/index.ts` incluindo a entrada para `Categories`:

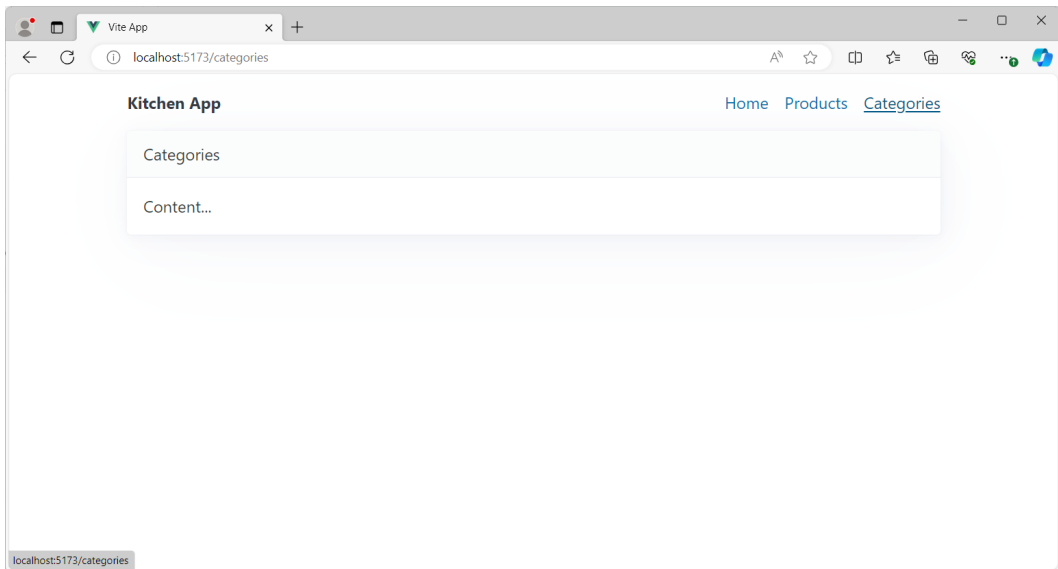
```
1 import { createRouter, createWebHistory } from 'vue-router'
2 import HomeView from '../views/HomeView.vue'
3 import CategoriesView from '../views/categories/IndexView.vue'
4
5 const router = createRouter({
6   history: createWebHistory(import.meta.env.BASE_URL),
7   routes: [
8     {
9       path: '/',
10      name: 'home',
11      component: HomeView
12    },
13    {
14      path: '/categories',
15      name: 'categories',
16      component: CategoriesView
17    },
18    {
19      path: '/about',
20      name: 'about',
21      // route level code-splitting
22      // this generates a separate chunk (About.[hash].js) for this route
23      // which is lazy-loaded when the route is visited.
24      component: () => import('../views/AboutView.vue')
25    }
26  ]
27 })
28
29 export default router
```

Agora que configuramos a url /categories, precisamos alterar o menu da aplicação que está no arquivo App.vue:

```
1 <script setup lang="ts">
2 import { RouterView, RouterLink } from 'vue-router'
3 </script>
4
5 <template>
6   <div class="container">
7     <nav>
8       <ul>
9         <li><strong>Kitchen App</strong></li>
10      </ul>
11      <ul>
12        <li><a href="#">Home</a></li>
13        <li><a href="#">Products</a></li>
14        <li><RouterLink to="/categories">Categories</RouterLink></li>
15      </ul>
16    </nav>
17    <RouterView />
18  </div>
19 </template>
```

Veja que a tag <a> foi substituída pela tag <RouterLink>, e o parâmetro to foi devidamente configurado para /categories, a mesma entrada da configuração do Router. Como usamos o componente <RouterLink>, é necessário importá-lo na tag de script, no início do arquivo.

Voltando ao navegador, clique no menu Categories e veja que o componente IndexView do Categories foi devidamente carregado:



Vamos também configurar o “Home”. Ao invés do `<a>` usaremos o `<RouterLink>`, veja:

```
1 <ul>
2   <li><RouterLink to="/">Home</RouterLink></li>
3   <li><a href="#">Products</a></li>
4   <li><RouterLink to="/categories">Categories</RouterLink></li>
5 </ul>
```

3.2 Exibindo uma Listagem de Categorias

Vamos utilizar o Router para configurar um componente que será responsável em exibir uma lista de categorias. Crie o arquivo `src/views/categories/ListView.vue` com o seguinte conteúdo:

```
1 <script setup lang="ts">
2
3 </script>
4 <template>
5   <div>Categories List</div>
6 </template>
```

Edite o arquivo `src/router/index.ts` adicionando a propriedade `children` na configuração de `categories`:

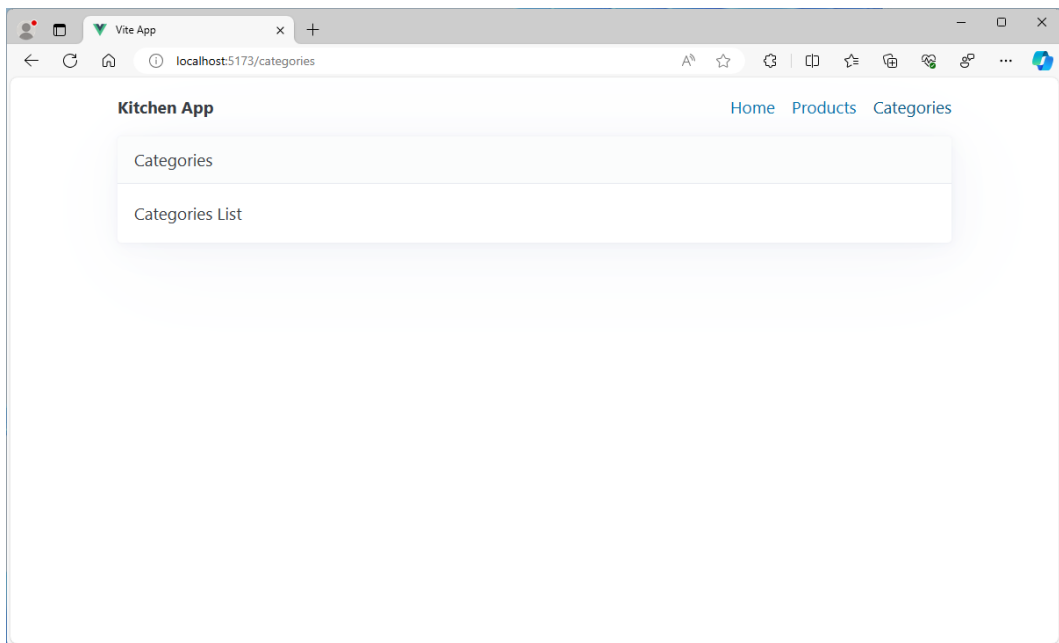
```
1 import { createRouter, createWebHistory } from 'vue-router'
2 import HomeView from '../views/HomeView.vue'
3 import CategoriesView from '../views/categories/IndexView.vue'
4
5 const router = createRouter({
6   history: createWebHistory(import.meta.env.BASE_URL),
7   routes: [
8     {
9       /// home
10    },
11    {
12      path: '/categories',
13      name: 'categories',
14      component: CategoriesView,
15      children: [
16        {
17          path: '',
18          name: 'ListCategories'
19          component: () => import('../views/categories/ListView.vue')
20        }
21      ]
22    },
23    {
24      /// about
25    }
26  ]
27 })
28
29 export default router
```

Desta vez, associamos o path /categories/ para que carregue dinamicamente o arquivo views/categories/ListView.vue. Neste arquivo, por enquanto, temos apenas uma mensagem Categories List.

Para que o roteamento funcione, é preciso adicionar o componente <RouterView /> no componente principal da tela de categorias (src/views/categories/IndexView.vue), da seguinte forma:

```
1 <script setup lang="ts">
2
3 </script>
4 <template>
5   <article>
6     <header>Categories</header>
7     <RouterView />
8   </article>
9 </template>
```

O resultado final é semelhante a figura a seguir:



3.3 Obtendo Dados

Para obter as categorias do servidor backend, podemos inicialmente usar a função `fetch` nativa do próprio javascript. Neste ponto, precisamos compreender duas funcionalidades do Vue.

O Que Há de Novo Vue 3: Ref

Uma das primeiras funcionalidades do Vue 3 que aprendemos é o `ref`, que irá marcar uma variável como um objeto de referência que será observado pelo Vue por ser mutável. Ou seja, uma variável `ref` ao mudar de valor irá ocasionar um redesenho dos componentes. No exemplo a seguir, criamos a variável `categories` que a princípio não possui nenhum valor:

```
1 <script setup lang="ts">
2 import { ref } from 'vue';
3
4 const categories = ref()
5
6 </script>
7 <template>
8   <div>Categories List</div>
9 </template>
```

Utilizando OnMounted()

Após criar esta constante, o valor de `categories` pode ser alterado através da propriedade `categories.value`. O método `onMounted()` é chamado quando todos os componentes estão devidamente incluídos na tela. Usamos o `onMounted` geralmente para carregar os dados iniciais que irão compor a tela, veja:

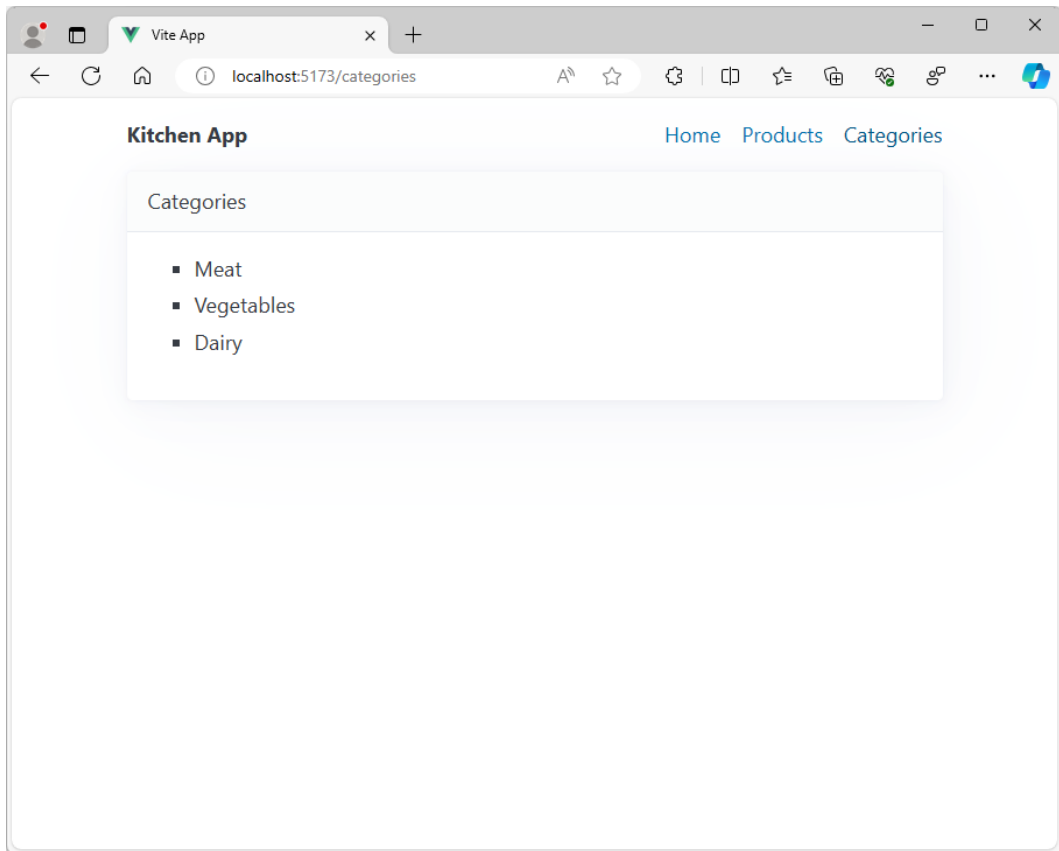
```
1 <script setup lang="ts">
2 import { onMounted, ref } from 'vue';
3
4 const categories = ref()
5
6 onMounted(async () => {
7   categories.value =
8     await fetch("http://localhost:3000/categories").then(
9       (r) => r.json()
10     )
11 })
12
13 </script>
14 <template>
15   <div>
16     <ul>
17       <li v-for="category in categories" :key="category.id">
```



```
18         {{ category.name }}
19     </li>
20 </ul>
21 </div>
22 </template>
```

Agora, quando o componente `categories/ListView` for devidamente carregado na tela, o método `onMounted` será chamado. Usamos o `fetch` para acessar diretamente os dados do servidor, e o resultado deste `fetch` é convertido para json e diretamente atribuído para `categories.value`. O `await` instrui ao javascript a aguardar que o `fetch` seja realizado antes de prosseguir com o fluxo de código.

No template, usamos o `v-for` para criar um loop pelos dados, exibindo o nome de cada categoria. O `v-for` necessita da definição de um atributo único do array, neste caso o `id` da categoria. O resultado deste código é semelhante a imagem a seguir:



3.4 Otimizando o Acesso Ao Servidor

Não é uma boa prática de programação ter que digitar a url completa do endereço do servidor em um componente Vue. Podemos realizar diversas otimizações neste código para melhorar a sua estrutura. Vamos comentar algumas melhorias:

- Como estamos utilizando TypeScript (`lang='ts'`), podemos criar um tipo que define a categoria
- Podemos criar uma classe chamada `CategoryService` que conterà todos os acessos ao servidor. Listar, criar, editar, remover etc.
- Podemos utilizar a biblioteca `axios` para facilitar o acesso ao servidor
- A url `localhost` precisa ser uma variável de ambiente para que possa ser substituída no servidor de produção.

Axios

Para instalar o axios, use o comando `npm install axios`. Após a instalação, vamos criar um arquivo chamado `HttpService.ts` no diretório `src/services`:

```
1  import axios from 'axios';
2
3  const HttpService = axios.create({
4    baseURL: "http://localhost:3000",
5    timeout: 10000,
6    headers: { 'Content-Type': 'application/json' }
7  })
8
9  export default HttpService;
```

Este arquivo contém a configuração inicial do Axios.

Criando o Service de Categorias

Agora, crie o arquivo `src/services/CategoryService.ts` com o seguinte código:

```
1  import HttpService from './HttpService'
2
3  export interface Category {
4    id?: number
5    name: string
6  }
7
8  const URI = '/categories'
9
10 export const CategoryService = {
11   getAll: async (): Category[] =>
12     (await HttpService.get(URI)).data,
13 }
```

Este arquivo possui uma interface chamada `Category`, que é a representação de uma categoria. Já o `CategoryService` possuirá métodos para o acesso ao backend, por enquanto temos apenas o método `getAll` que obtém todas as categorias.

O id? indica que esta propriedade pode ser opcional.

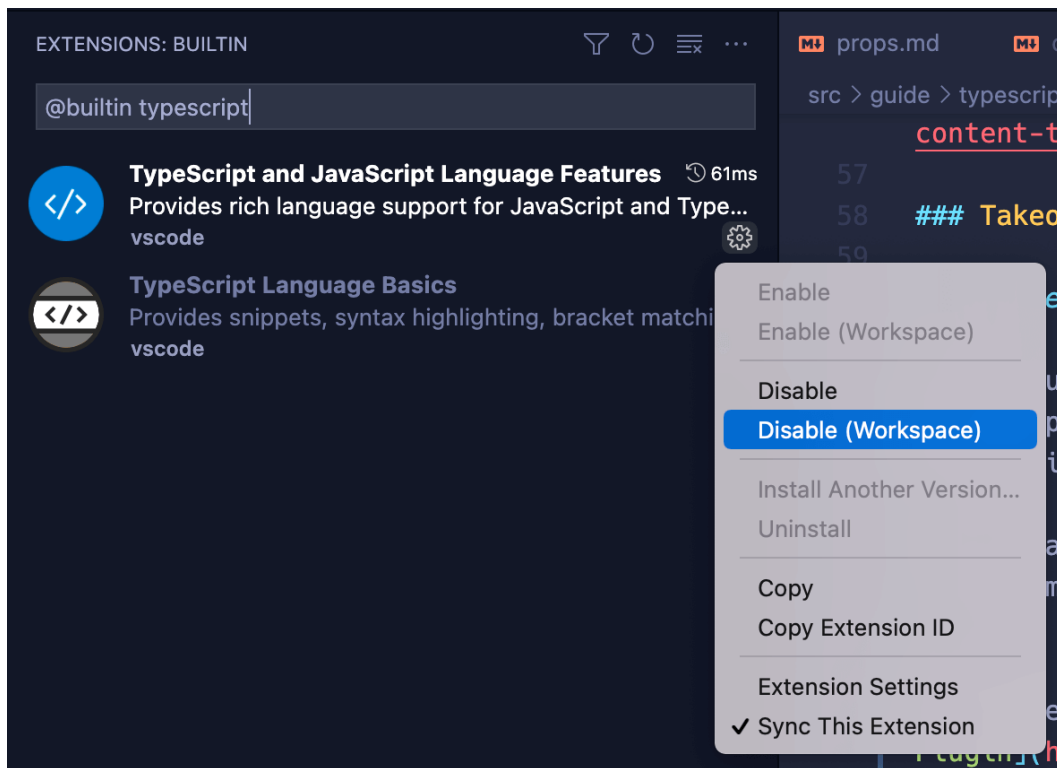
3.5 Refatorando o ListView, Acessando o Service

Com o service configurado, podemos voltar em views/categories/ListView e alterar o código para:

```
1  <script setup lang="ts">
2  import { onMounted, ref } from 'vue';
3  import { CategoryService, type Category } from '../services/CategoryService'
4
5  const categories = ref<Category[]>()
6
7  onMounted(async () => {
8    categories.value = await CategoryService.getAll()
9  })
10
11 </script>
12 <template>
13   <div>
14     <ul>
15       <li v-for="category in categories"
16           :key="category.id">
17         {{ category.name }}
18       </li>
19     </ul>
20   </div>
21 </template>
```

Nesta refatoração, criamos a constante `categories` com um tipo, definido como `ref<Category[]>()`. Ao invés de utilizar o `fetch` para obter os dados, estamos chamando o método `getAll` do `CategoryService`. A parte do `template` não muda.

Caso você tenha problemas de TypeScript nos códigos, experimente remover a extensão TypeScript and JavaScript Language Features:



3.6 Criando uma Nova Categoria

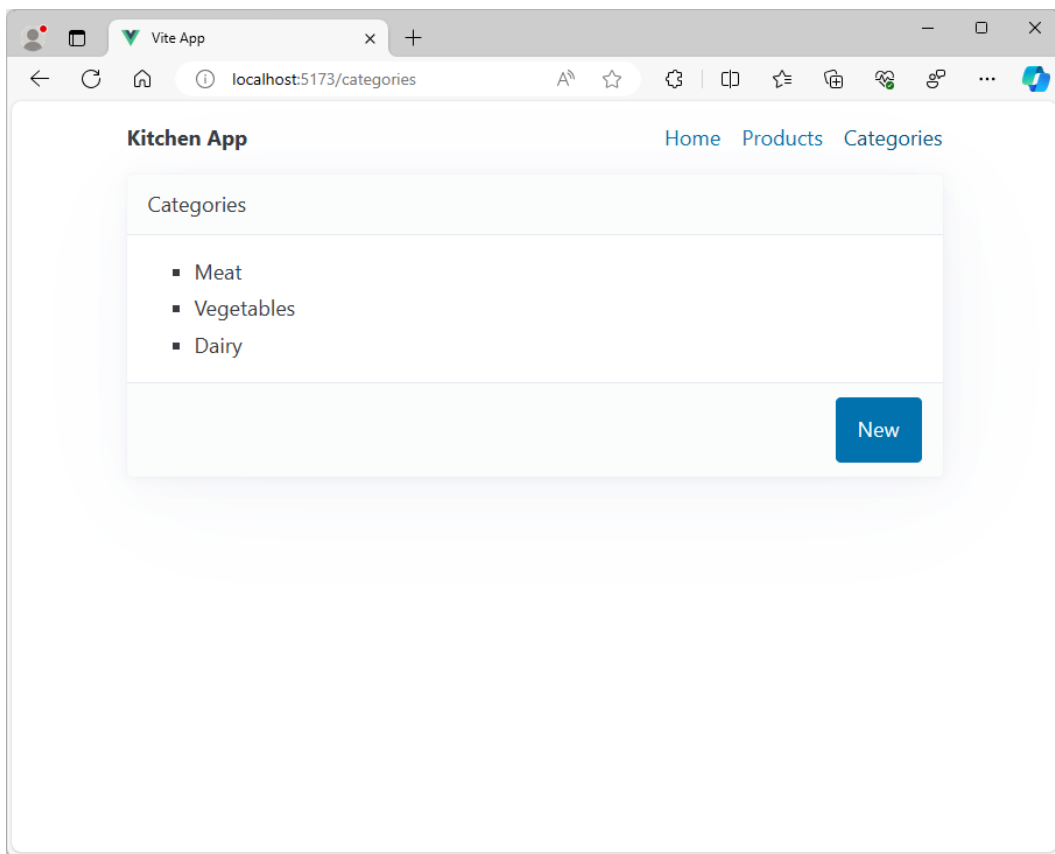
Para criarmos uma nova categoria, vamos adicionar o botão New no footer do card de categorias que está no componente `src/views/categories/ListView.vue`:

```
1 <script setup lang="ts">
2 // code
3 </script>
4 <template>
5 // code
6 <footer>
7   <router-link to="/categories/create">
8     <button>New</button>
9   </router-link>
10 </footer>
11 </template>
```

Adicionamos no <footer> um <router-link> em conjunto com um botão. Para deixar o botão no canto direito da tela, podemos usar CSS. Abra o arquivo `src/assets/main.css` e adicione o seguinte código:

```
1 @import url(../../node_modules/@picocss/pico/css/pico.min.css);
2
3 footer {
4   display: flex;
5   justify-content: end;
6   gap: 10px;
7 }
```

Utilizando CSS Flex Box, colocamos o botão New no lado direito:



Criando o Componente e a Rota

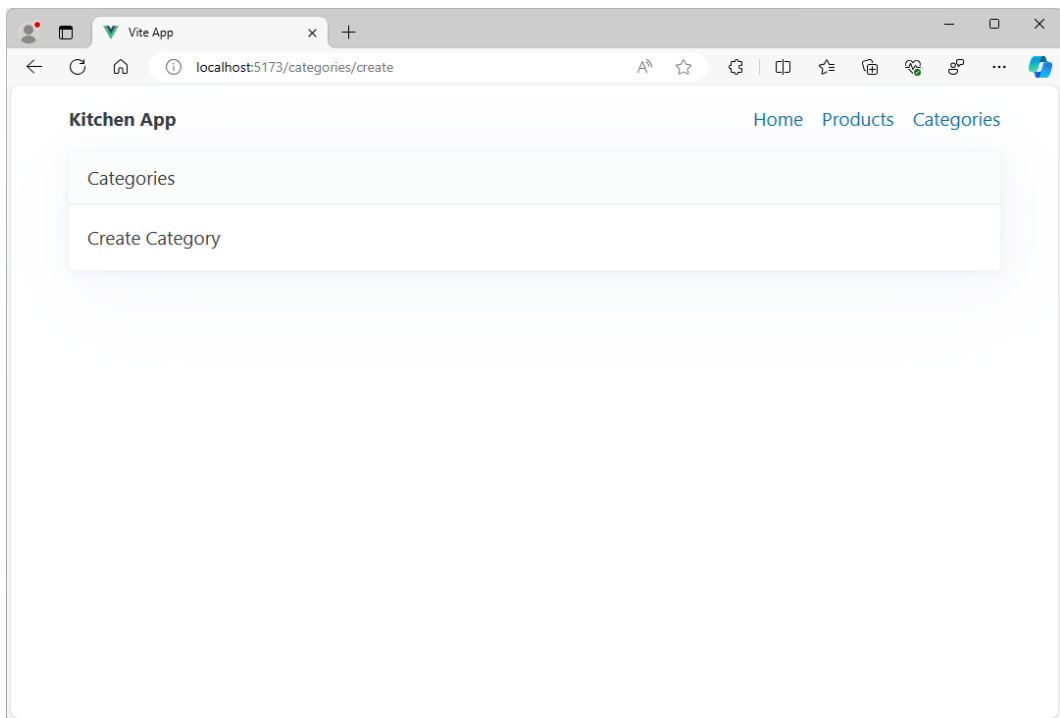
Como ainda não criamos o arquivo e a rota, ao clicar no botão, uma tela em branco irá surgir. Precisamos criar o componente e a rota. Crie o arquivo `src/views/categories/CreateView.vue` com inicialmente o seguinte código:

```
1 <script setup lang="ts">
2
3 </script>
4 <template>
5   Create Category
6 </template>
```

E no router:

```
1  import { createRouter, createWebHistory } from 'vue-router'
2  import HomeView from '../views/HomeView.vue'
3  import CategoriesView from '../views/categories/IndexView.vue'
4
5  const router = createRouter({
6    history: createWebHistory(import.meta.env.BASE_URL),
7    routes: [
8      {
9        /// router
10     },
11     {
12       path: '/categories',
13       name: 'categories',
14       component: CategoriesView,
15       children: [
16         {
17           path: '',
18           name: 'ListCategories',
19           component: () => import('../views/categories/ListView.vue')
20         },
21         {
22           path: 'create',
23           name: 'CreateCategory',
24           component: () => import('../views/categories/CreateView.vue')
25         }
26       ]
27     },
28     {
29       /// router
30     }
31   ]
32 })
33
34 export default router
```

O path: 'create' foi adicionado na lista de componentes do categories. Quando o botão New for clicado, você verá a princípio algo semelhante a figura a seguir:



Criando o Formulário

O formulário e o botão para salvar e voltar são criados a seguir:

```
1 <script>...</script>
2 <template>
3   <h5>Create Category</h5>
4   <form>
5     <label>
6       Name
7       <input name="name" placeholder="Category Name" />
8     </label>
9   </form>
10  <footer>
11    <router-link to="/categories">
12      <button class="outline">Back</button>
13    </router-link>
14    <button>Save</button>
```

```
15     </footer>  
16 </template>
```

O resultado deste formulário é exibido a seguir:

The screenshot shows a web browser window with the title 'Vite App' and the address bar displaying 'localhost:5173/categories/create'. The application is titled 'Kitchen App' and has a navigation menu with 'Home', 'Products', and 'Categories'. The 'Categories' page features a 'Create Category' form with a 'Name' label and a text input field containing the placeholder 'Category Name'. At the bottom right of the form are 'Back' and 'Save' buttons.

Com o formulário pronto, podemos adicionar as funcionalidades Vue para que os dados do formulário possam ser enviados ao servidor.

É preciso criar uma variável que na qual podemos chamar de `form`, que possui o tipo de dados do objeto que iremos enviar ao servidor. Veja:

```
1  <script setup lang="ts">
2  import type { Category } from '@/services/CategoryService'
3  import { ref } from 'vue'
4
5  const form = ref<Category>({
6    name: ''
7  })
8
9  </script>
10 <template>...</template>
```

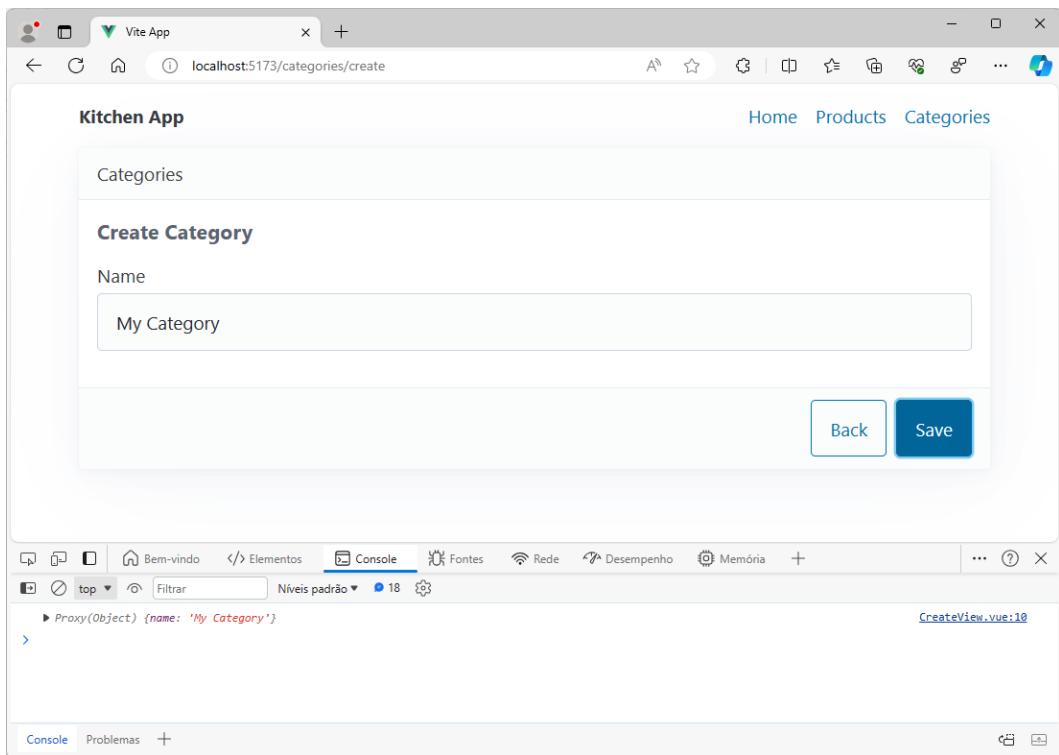
Como o formulário possui apenas o campo `name`, que é o nome da categoria, o objeto `form` possui apenas a propriedade `name`. Para ligar o `input` a variável `form.name`, usamos o `v-model`:

```
1  <script setup lang="ts">
2  ...
3  </script>
4  <template>
5    <h5>Create Category</h5>
6    <form>
7      <label>
8        Name
9        <input name="name" v-model="form.name"
10         placeholder="Category Name" />
11      </label>
12    </form>
13    <footer>
14      <router-link to="/categories">
15        <button class="outline">Back</button>
16      </router-link>
17      <button>Save</button>
18    </footer>
19  </template>
```

O `v-model` fará uma ligação entre a o objeto `form` e o `input`. Quando o usuário digitar algo na caixa de texto, o valor de `form.name` será preenchido. O botão `save` irá chamar uma função, que por enquanto apenas irá imprimir o valor do objeto `form`:

```
1  <script setup lang="ts">
2  import type { Category } from '@/services/CategoryService'
3  import { ref } from 'vue'
4
5  const form = ref<Category>({
6    name: ''
7  })
8
9  const save = () => {
10    console.log(form.value)
11  }
12
13 </script>
14 <template>
15   <h5>Create Category</h5>
16   <form>
17     <label>
18       Name
19       <input name="name" type="text" v-model="form.name"
20         placeholder="Category Name" />
21     </label>
22   </form>
23   <footer>
24     <router-link to="/categories">
25       <button class="outline">Back</button>
26     </router-link>
27     <button @click="save()">Save</button>
28   </footer>
29 </template>
```

Ao preencher uma Categoria e clicar no botão save, temos o seguinte resultado:



Persistindo Dados

Para realmente salvar os dados no servidor (no nosso exemplo, no arquivo Json), é necessário criar o método “create” no CategoryService que será responsável exclusivamente em criar uma nova categoria:

```
1 import HttpService from './HttpService'
2
3 export interface Category {
4   id?: number
5   name: string
6 }
7
8 const URI = '/categories'
9
10 export const CategoryService = {
11   getAll: async (): Category[] =>
```

```

12         (await HttpService.get(URI)).data,
13     create: async (category: Category): Category =>
14         (await HttpService.post(URI, category)).data,
15 }

```

O método `create` recebe como parâmetro um objeto do tipo `Category`, e usa o `HttpService` para realizar um POST para a URI, repassando este objeto. O resultado deste POST é retornado pelo servidor (possivelmente um objeto com o id preenchido).

Voltando ao formulário `src/views/categories/CreateView.vue`, temos:

```

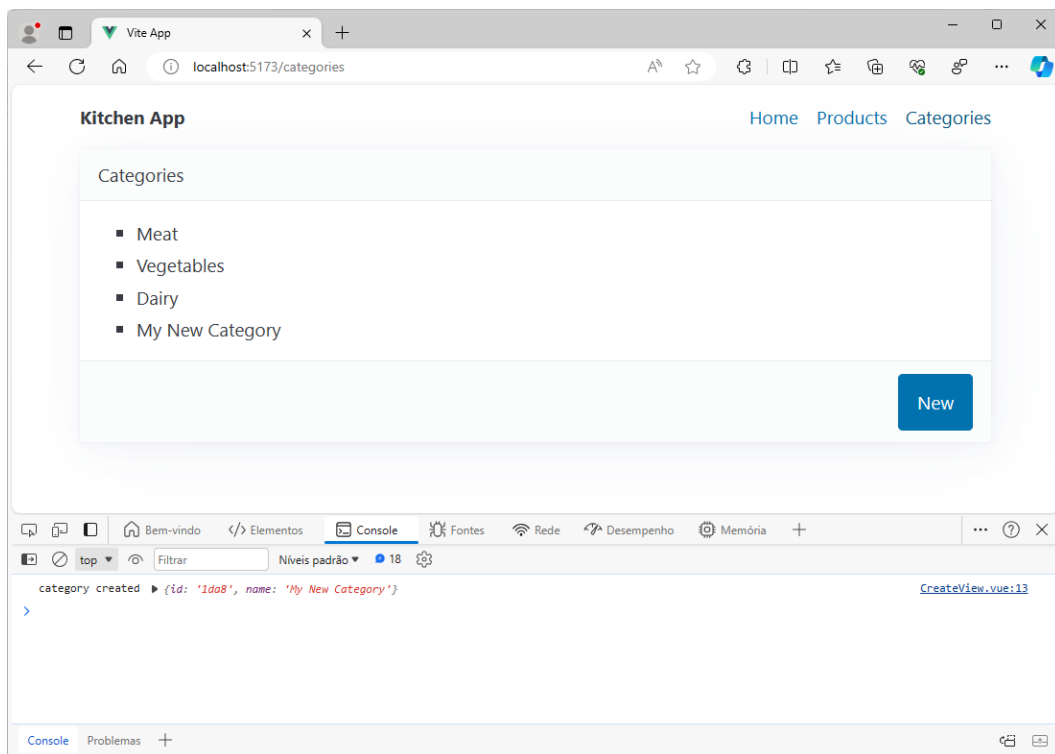
1  <script setup lang="ts">
2  import router from '@/router';
3  import { CategoryService, type Category } from '@/services/CategoryService'
4  import { ref } from 'vue'
5
6  const form = ref<Category>({
7    name: ''
8  })
9
10 const save = async () => {
11   //console.log(form.value)
12   const result = await CategoryService.create(form.value)
13   console.log("category created", result)
14   router.push("/categories")
15 }
16
17 </script>
18 <template>
19   <h5>Create Category</h5>
20   <form>
21     <label>
22       Name
23       <input name="name" type="text" v-model="form.name"
24         placeholder="Category Name" />
25     </label>
26   </form>
27   <footer>
28     <router-link to="/categories">
29       <button class="outline">Back</button>
30     </router-link>
31     <button @click="save()">Save</button>

```

```
32     </footer>
33 </template>
```

Usamos `await CategoryService.create(form.value)` para criar a categoria e logo após a criação, usamos `router.push("/categories")` para voltar a listagem de categorias. Esta é a forma mais básica para criar uma nova categoria.

Na imagem abaixo, temos a exibição do `console.log("category created", result)` e a listagem da nova categoria recém criada.



Exibindo um Feedback Ao Usuário

Até o momento não nos preocupamos muito com o usuário. Precisamos de certa forma exibir uma resposta ao usuário quando os dados estão sendo carregados. Mesmo com a velocidade da internet nos dias de hoje, é importante exibir um feedback ao usuário.

O primeiro feedback que iremos realizar é no `ListView`, quando a lista de categorias estiver sendo solicitada ao servidor. Para isso, vamos criar uma variável chamada `loading`:

```
1 <script setup lang="ts">
2 import { onMounted, ref } from 'vue'
3 import { CategoryService, type Category } from '../services/CategoryService'
4
5 const categories = ref<Category[]>()
6 const loading = ref<boolean>(false)
7
8 // code
9 </script>
10 <template>
11   ...
12 </template>
```

Veja que, sempre que criamos uma variável que irá interagir com o template, ela deve ser criada através do ref. A variável loading é do tipo boolean e inicialmente possui o valor false. No momento antes de buscar os dados no servidor, iremos setar esta variável para true, e quando o servidor retornar com os dados, voltaremos o valor dessa variável para false:

```
1 <script setup lang="ts">
2 import { onMounted, ref } from 'vue'
3 import { CategoryService, type Category } from '../services/CategoryService'
4
5 const categories = ref<Category[]>()
6 const loading = ref<boolean>(false)
7
8 onMounted(async () => {
9   loading.value = true;
10   categories.value = await CategoryService.getAll()
11   loading.value = false;
12 })
13 </script>
14 <template>
15   ...
16 </template>
```

Agora, iremos adicionar um novo componente no template, que será visível apenas quando loading for true:


```
1 <script setup lang="ts">
2 import { onMounted, ref } from 'vue'
3 import { CategoryService, type Category } from '../services/CategoryService'
4
5 const categories = ref<Category[]>()
6 const loading = ref<boolean>(false)
7
8 onMounted(async () => {
9   loading.value = true;
10   categories.value = await CategoryService.getAll()
11   loading.value = false;
12 })
13 </script>
14 <template>
15   <div>
16     <progress v-if="loading" />
17     <ul v-else>
18       <li v-for="category in categories"
19         :key="category.id">
20         {{ category.name }}
21       </li>
22     </ul>
23   </div>
24   <footer>
25     <router-link to="/categories/create">
26       <button>New</button>
27     </router-link>
28   </footer>
29 </template>
```

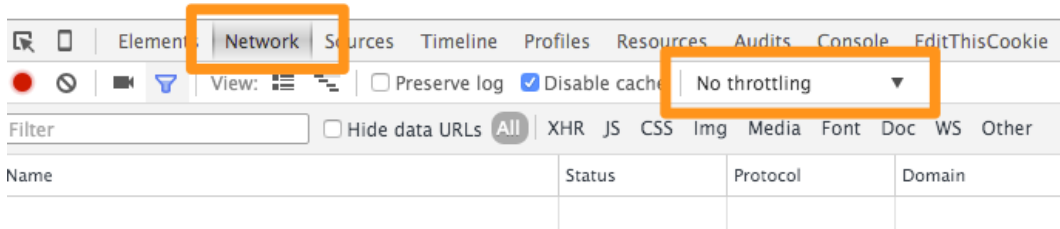
Antes da tag `` temos o componente `<progress>` do PicoCSS que exibe uma barra de loading indeterminada. Esta barra é visível apenas quando `loading` assume o valor `true`.

Em ambiente `localhost`, com o servidor `vue` e servidor `json-server` funcionando na própria máquina, quase não pode-se ver a barra de progresso em funcionamento. Mas com certeza todo este cuidado ao exibir dados para o usuário será útil em conexões mais lentas.

Simulando uma Conexão Lenta

O navegador possui recursos para que você possa testar a aplicação em conexões mais lentas. Para isso, aperte `F12` e vá até a aba `Networking` e a opção `Throttling`. Escolha por exemplo

Slow 3g e navegue entre Home e Categories. Você poderá perceber agora o componente <progress> em ação.



Ainda com a opção Slow 3g ativa, crie uma nova categoria e verifique que ao pressionar o botão Save, não há nenhuma resposta do servidor ao usuário sobre o processo de salvar uma categoria.

Feedback Ao Salvar uma Categoria

Quando o usuário clicar no botão Save, devemos adicionar um feedback para o usuário. O ideal seria que o botão Save possuisse uma indicação de loading e ficasse desabilitado enquanto o servidor estivesse sendo acessado para salvar a categoria. Iremos utilizar a mesma estratégia do ListView, criando uma variável chamada loading e manipulando ela no método save.

```
1 <script setup lang="ts">
2 import router from '@/router';
3 import { CategoryService, type Category } from '@/services/CategoryService'
4 import { ref } from 'vue'
5
6 const form = ref<Category>({
7   name: ''
8 })
9
10 const loading = ref<boolean>(false)
11
12 const save = async () => {
13   loading.value = true
14   const result = await CategoryService.create(form.value)
15   console.log("category created", result)
16   loading.value = false
17   router.push("/categories")
18 }
```

```
19
20 </script>
21 <template>
22 ...
23 </template>
```

Para o template, precisamos adicionar duas propriedades ao botão Save. A primeira é `aria-busy` no qual indica que uma região está com o status de ocupado. No caso, a região é o botão e o PicoCss tem essa capacidade de adicionar um ícone de loading no botão.

A segunda propriedade é o `disabled`, que indica que o botão estará com o status de desativo e não poderá ser clicado.

O código do template fica:

```
1 <script setup lang="ts">
2 ...
3 </script>
4 <template>
5   <h5>Create Category</h5>
6   ...form...
7   <footer>
8     <router-link to="/categories">
9       <button class="outline">Back</button>
10    </router-link>
11    <button
12      :aria-busy="loading"
13      :disabled="loading"
14      @click="save()">Save</button>
15  </footer>
16 </template>
```

Ao clicar no botão save, tanto `:aria-busy` quanto `:disabled` assumem o valor da variável `loading`, nesse caso `true`. O resultado é semelhante a figura a seguir:

Kitchen App[Home](#)[Products](#)[Categories](#)

Categories

Create Category

Name

BackSave

O uso de `:` antes do nome da propriedade define uma ligação entre a propriedade e uma variável do vue, e não o texto em si. Isso é conhecido como binding.

Validação de Dados

A validação de dados de um formulário é necessária para diminuir as chances de que dados indesejados sejam enviados ao servidor. Por exemplo, no caso do campo nome do formulário de categorias, o mínimo que se espera é que o campo não esteja vazio.

Existem algumas formas de aplicar validação de dados no formulário, inclusive com algumas bibliotecas tais como `VueIvalidate`, `VeeValidate`, `Zod` etc. Neste exemplo inicial, vamos aplicar a validação da forma mais simples possível, apenas ao clicar no botão `Save`.

É necessário criar um objeto que corresponde ao estado de validação do formulário e dos seus respectivos campos. Este objeto é criado da seguinte forma:

```
1 // fv = formValidation
2 const fv = ref<{
3   valid: boolean
4   name: {
5     valid: boolean
6     empty: boolean
7   }
8 }>({
9   valid: true,
10  name: {
11    valid: true,
12    empty: false
13  }
14 })
```

A variável `fv` que corresponde a `formValidation` é um objeto que possui a propriedade `valid` que irá indicar se o formulário é válido. A propriedade `name` possui duas propriedades, `invalid` e `empty`, para definirem se o campo `Name` é inválido, e se o campo `Name` está vazio.

Caso queria simplificar a criação desta variável, podemos alterar o código para:

```
1 interface FormValidation {
2   valid: boolean
3   name: {
4     valid: boolean
5     empty: boolean
6   }
7 }
8
9 // fv = formValidation
10 const fv = ref<FormValidation>({
11   valid: true,
12   name: {
13     valid: true,
14     empty: false
15   }
16 })
```

Quando o usuário clicar no botão `save`, devemos chamar uma função que irá verificar todos os campos e analisar se o formulário é válido:

```
1  const save = async () => {
2    if (validate()) {
3      loading.value = true
4      const result = await CategoryService.create(form.value)
5      console.log('category created', result)
6      loading.value = false
7      router.push('/categories')
8    }
9  }
```

A função `validate()` é exibida a seguir:

```
1  const validate = () => {
2    fv.value.valid = true
3    fv.value.name.valid = true
4    fv.value.name.empty = false
5    if (form.value.name === '') {
6      fv.value.valid = false
7      fv.value.name.valid = false
8      fv.value.name.empty = true
9    }
10   if (form.value.name.trim() === '') {
11     fv.value.valid = false
12     fv.value.name.valid = false
13     fv.value.name.empty = true
14   }
15   return fv.value.valid
16 }
```

Esta função analisa o campo `name` do formulário, e se ele for nulo ou vazio, o valor de `valid` será `false`, e `fv.value.name.empty` indicando que existe um erro do tipo `empty`.

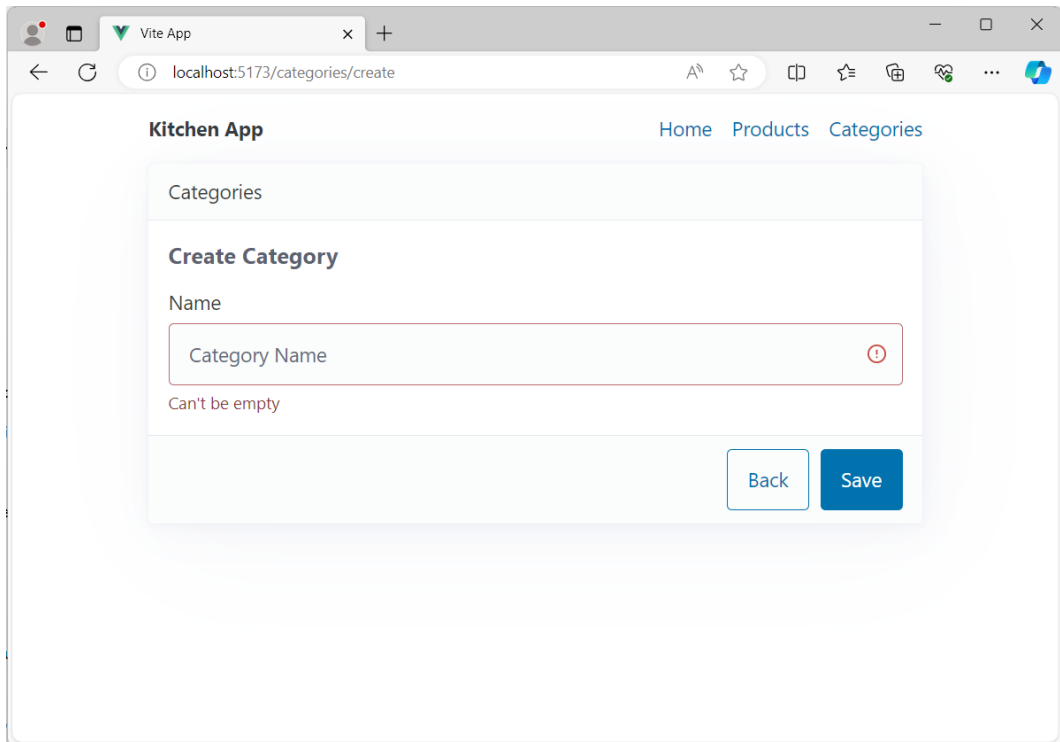
Como esta função retorna `return fv.value.valid`, quando retornar `false` devido a algum erro, a verificação `if (validate())` irá falhar e o método `save` não irá persistir os dados.

É necessário agora adicionar no formulário `html` as informações sobre um estado inválido do campo `name`. Para isso, podemos usar a classe `aria-invalid`:

```
1  <form>
2    <label>
3      Name
4      <input
5        name="name"
6        type="text"
7        v-model="form.name"
8        placeholder="Category Name"
9        required
10       :aria-invalid="!fv.name.valid||undefined"
11     />
12     <small v-if="fv.name.empty"> Can't be empty </small>
13   </label>
14 </form>
```

Como estamos utilizando o PicoCSS, a propriedade `aria-invalid` irá configurar o campo de texto para um estado de erro, caso existir algum erro no campo `name`. A tag `<small>` logo abaixo do `<input>` somente será exibida se `fv.name.empty` for `true`.

Após realizar estas alterações, experimente adicionar uma nova categoria com o campo `name` vazio. O resultado será semelhante a figura a seguir.



Adicionando Outra Validação

Agora que a estrutura básica de validação está pronta, é simples adicionar mais uma validação no campo `name`. Suponha que este campo necessita de pelo menos dois caracteres. Podemos então criar uma validação chamada de `minLength`:

```
1 interface FormValidation {
2   valid: boolean
3   name: {
4     valid: boolean
5     empty: boolean
6     minLength: boolean
7   }
8 }
9
10 // fv = formValidation
11 const fv = ref<FormValidation>({
12   valid: true,
```

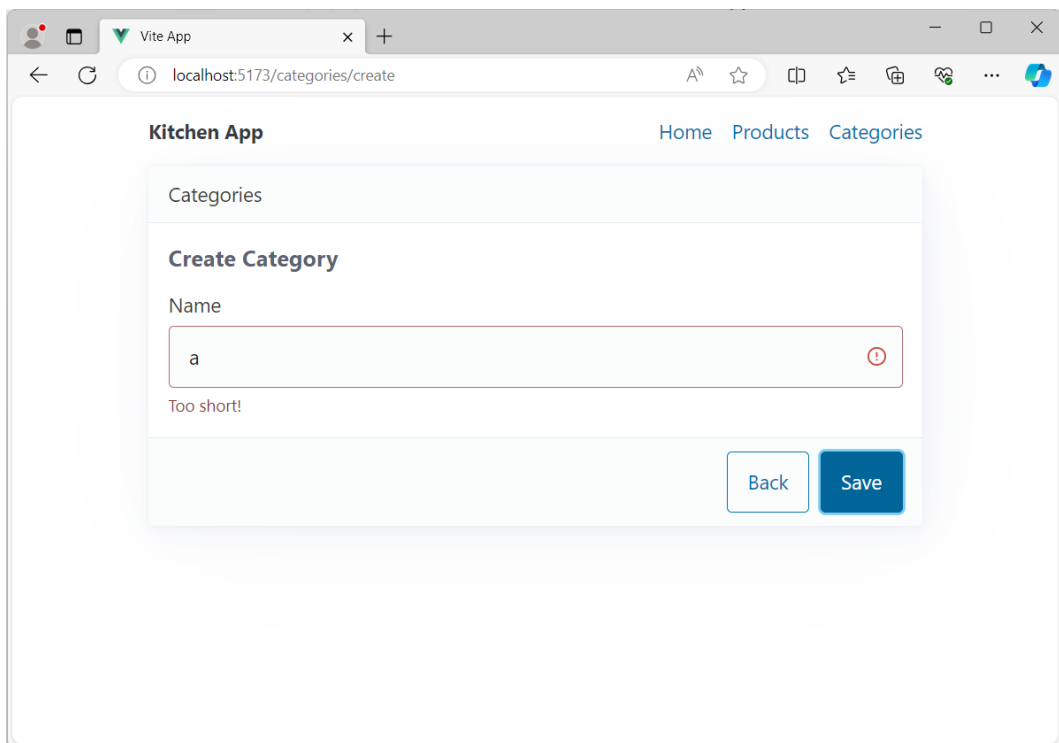


```
13     name: {
14         valid: true,
15         empty: false,
16         minLength: false
17     }
18 })
19
20 const loading = ref<boolean>(false)
21
22 const validate = () => {
23     fv.value.valid = true
24     fv.value.name.valid = true
25     fv.value.name.empty = false
26     fv.value.name.minLength = false
27     if (form.value.name === '') {
28         fv.value.valid = false
29         fv.value.name.valid = false
30         fv.value.name.empty = true
31     }
32     if (form.value.name.trim() === '') {
33         fv.value.valid = false
34         fv.value.name.valid = false
35         fv.value.name.empty = true
36     }
37     if (fv.value.valid && form.value.name.length<2){
38         fv.value.valid = false
39         fv.value.name.valid = false
40         fv.value.name.minLength = true
41     }
42     return fv.value.valid
43 }
```

Neste código adicionamos a propriedade `minLength`. No método `validate`, iremos avaliar o `minLength` no terceiro `if` do método `validate`. Este `if`, verifica primeiro se o formulário é válido, pois caso não seja, nem é necessário continuar, já que a validação do formulário já falhou em algum momento.

No formulário, devemos apenas adicionar a mensagem de erro relativo ao `minLength`, que será exibida somente se `fv.value.name.minLength` for `true`

```
1 <form>
2   <label>
3     Name
4     <input
5       name="name"
6       type="text"
7       v-model="form.name"
8       placeholder="Category Name"
9       required
10      :aria-invalid="!fv.name.valid||undefined"
11    />
12    <small v-if="fv.name.empty"> Can't be empty </small>
13    <small v-if="fv.name.minLength"> Too short! </small>
14  </label>
15 </form>
```



The screenshot shows a web browser window with the address bar displaying 'localhost:5173/categories/create'. The page title is 'Kitchen App' and there are navigation links for 'Home', 'Products', and 'Categories'. The main content area is titled 'Categories' and contains a 'Create Category' form. The form has a 'Name' label and a text input field containing the letter 'a'. Below the input field, the text 'Too short!' is displayed, indicating a validation error. At the bottom right of the form, there are two buttons: 'Back' and 'Save'.

Pereba que o código do método `validate` está extenso, isso apenas para um campo de formulário. Com mais campos esta prática de validação torna-se muito complexa, já que

teremos um método `validate` com dezenas de linhas. Por isso existem algumas bibliotecas dedicadas exclusivamente a validação, que são vistas em breve.

3.7 Editando uma Categoria

O processo para editar uma categoria envolve a criação de um link da lista de categorias para uma nova view chamada `EditView.vue`. Neste componente, obtemos o Id da categoria a ser editada e os realizamos uma consulta ao servidor para obter os dados da categoria. Os dados são exibidos no formulário, que neste caso possui apenas o campo `Name`. Quando o usuário clicar no botão salvar, reenviamos estes dados ao servidor para que possa ser realizada a alteração.

Configurando o Service

Adicione o método `get` no `CategorieService` para obter uma categoria dado o id:

```
1  import HttpService from './HttpService'
2
3  export interface Category {
4    id?: number
5    name: string
6  }
7
8  const URI = '/categories'
9
10 export const CategoryService = {
11   getAll: async (): Category[] =>
12     (await HttpService.get(URI)).data,
13   get: async (id): Category =>
14     (await HttpService.get(URI + "/" + id)).data,
15   create: async (category: Category): Category =>
16     (await HttpService.post(URI, category)).data,
17 }
```

O método `get` segue o padrão do servidor REST. Ao acessar `/categories/1` por exemplo, a categoria de id 1 será retornada.

Configurando o Router

O router é configurado para possui a variável `id` repassada pela url:

```
1 import { createRouter, createWebHistory } from 'vue-router'
2 import HomeView from '../views/HomeView.vue'
3 import CategoriesView from '../views/categories/IndexView.vue'
4
5 const router = createRouter({
6   history: createWebHistory(import.meta.env.BASE_URL),
7   routes: [
8     {
9       path: '/',
10      name: 'home',
11      component: HomeView
12    },
13    {
14      path: '/categories',
15      name: 'categories',
16      component: CategoriesView,
17      children: [
18        {
19          path: '',
20          name: 'ListCategories',
21          component: () => import('../views/categories/ListView.vue')
22        },
23        {
24          path: 'create',
25          name: 'CreateCategory',
26          component: () => import('../views/categories/CreateView.vue')
27        },
28        {
29          path: 'edit/:id',
30          name: 'EditCategory',
31          component: () => import('../views/categories/EditView.vue')
32        }
33      ]
34    },
35    {
36      path: '/about',
37      name: 'about',
38      // route level code-splitting
39      // this generates a separate chunk (About.[hash].js) for this route
40      // which is lazy-loaded when the route is visited.
41      component: () => import('../views/AboutView.vue')
42    }
43  ]
44 }
```

```
44  })  
45  
46  export default router
```

Quando o path possui uma variável, nesse caso `id` deve ser configurado com dois pontos, por exemplo path: `'edit/:id'`.

Obtendo uma Categoria Pelo Id

Com o router e o service configurados, podemos criar o arquivo `src/views/categories/EditView.vue`, inicialmente com o seguinte código:

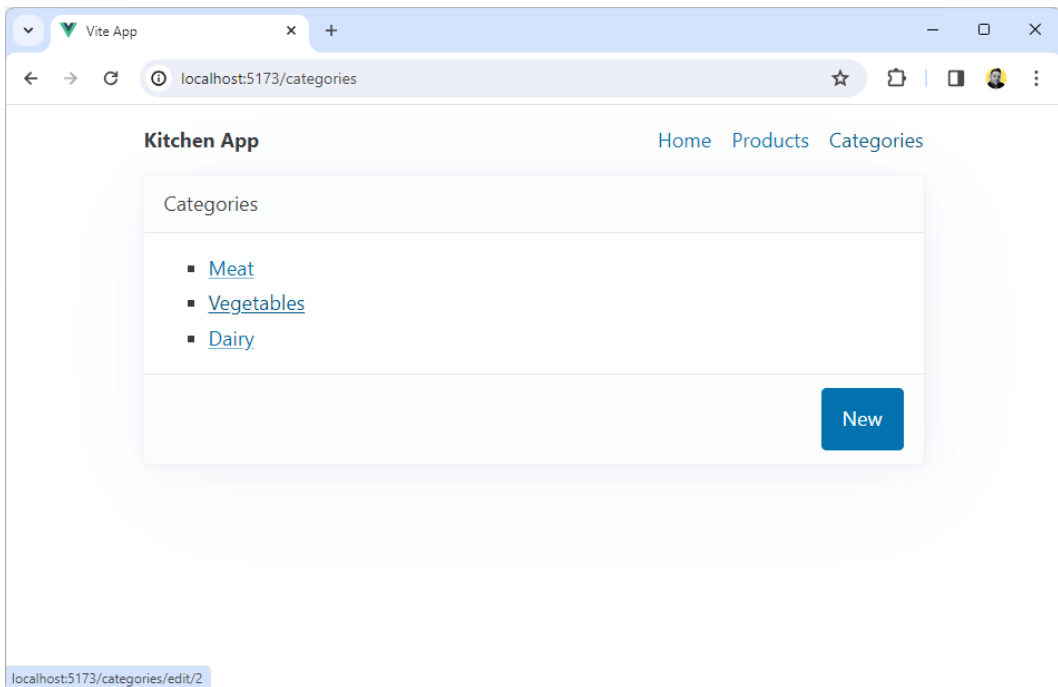
```
1  <script setup lang="ts">  
2  import { CategoryService, type Category } from '@/services/CategoryService'  
3  import { ref, onMounted } from 'vue'  
4  import { useRoute } from 'vue-router'  
5  
6  const form = ref<Category>({  
7    id: 0,  
8    name: ''  
9  })  
10  
11  const route = useRoute()  
12  
13  onMounted(async () => {  
14    const id = route.params.id  
15    const category = await CategoryService.get(id)  
16    form.value.id = category.id  
17    form.value.name = category.name  
18    console.log(form.value)  
19  })  
20  
21  </script>  
22  <template>  
23    Edit Category  
24  </template>
```

Neste código, usamos `route.params.id` para obter o `id` que foi repassado pela url do router. Com o `id`, utilizamos o `CategoryService` para obter os dados da categoria, neste caso `id` e `name`.

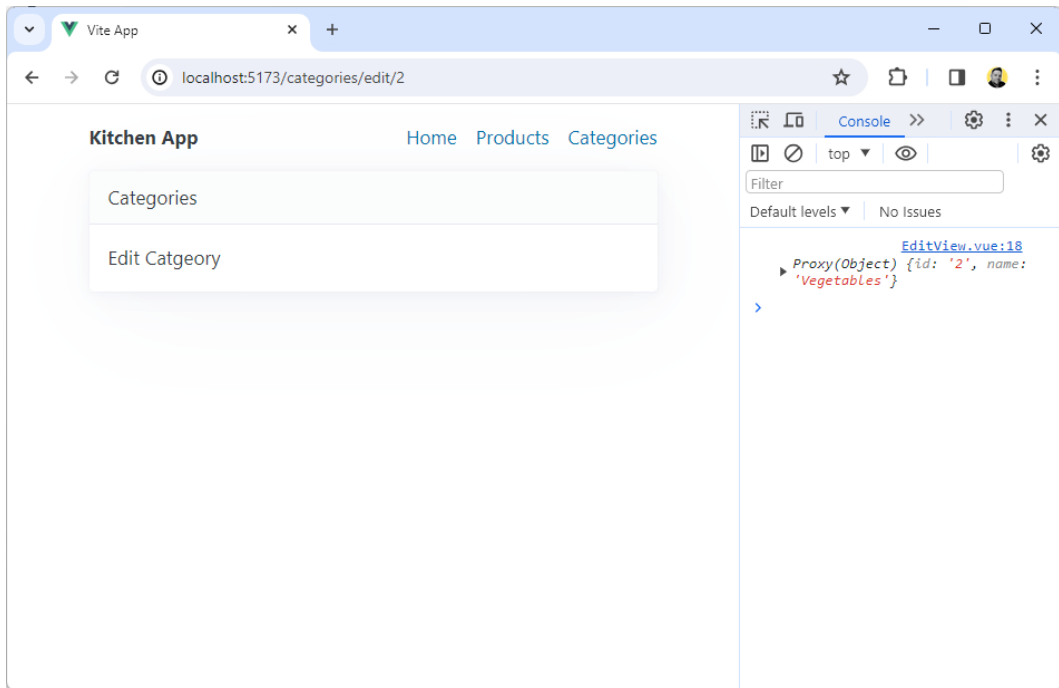
Para acessar o componente, precisamos no `categories/ListView` adicionar um link para editar a categoria:

```
1 <ul v-else>
2   <li v-for="category in categories" :key="category.id">
3     <router-link :to="/categories/edit/${category.id}">
4       {{ category.name }}
5     </router-link>
6   </li>
7 </ul>
```

Ao invés de exibir apenas o nome da categoria, adicionamos um link através do `router-link`, configurando o `to` de forma dinâmica para `/categories/edit/${category.id}`:



Ao clicar no link, e acessando o console do navegador, podemos ver o objeto `category` sendo carregado:



Formulário para Editar Categorias

O formulário para editar categorias é semelhante ao formulário para criar uma categoria:

```

1 <template>
2   <h5>Edit Category</h5>
3   <form>
4     <input type="hidden" name="id" v-model="form.id" />
5     <label>
6       Name
7       <input name="name" type="text" v-model="form.name"
8         placeholder="Category Name" required />
9     </label>
10  </form>
11  <footer>
12    <router-link to="/categories">
13      <button class="outline">Back</button>
14    </router-link>
15    <button :aria-busy="loading" :disabled="loading"

```

```
16         @click="save()">Save</button>
17     </footer>
18 </template>
```

O formulário possui dois campos: id e name. O campo id é do tipo hidden. O botão Save irá executar o método save:

```
1  const save = async () => {
2    try {
3      loading.value = true
4      const result = await CategoryService.update(form.value)
5      console.log('category created', result)
6      loading.value = false
7      router.push('/categories')
8    } catch (error) {
9      console.log(error)
10   } finally {
11     loading.value = false
12   }
13 }
```

O método update no service é exibido a seguir:

```
1  import HttpService from './HttpService'
2
3  // code
4
5  const URI = '/categories'
6
7  export const CategoryService = {
8    // code
9    update: async (category: Category): Category =>
10      (await HttpService.put(URI + "/" + category.id, category)).data
11  }
```

3.8 Removendo uma Categoria

Para remover uma categoria, podemos adicionar um botão ao editar a categoria, no arquivo `src\views\categories\EditView.vue`.


```
1 <template>
2   <h5>Edit Category</h5>
3   <form>
4     <input type="hidden" name="id" v-model="form.id" />
5     <label>
6       Name
7       <input name="name" type="text" v-model="form.name"
8         placeholder="Category Name" required />
9     </label>
10  </form>
11  <div style="display: flex; justify-content: center;">
12    <button :aria-busy="loading" :disabled="loading"
13      @click="remove()" class="outline secondary">Delete</button>
14  </div>
15  <footer>
16    <router-link to="/categories">
17      <button class="outline">Back</button>
18    </router-link>
19    <button :aria-busy="loading" :disabled="loading"
20      @click="save()">Save</button>
21  </footer>
22 </template>
```

O código do método `remove()` é exibido a seguir:

```
1 const remove = async () => {
2   if (confirm("Remove?")) {
3     console.log("remove", form.value.id)
4   }
5 }
```

Antes de remover a categoria, temos que adicionar o método `remove` no `CategoryService`:

```
1 export const CategoryService = {
2   // ...
3   delete: async (id: any) => {
4     (await HttpService.delete(`${URI}/${id}`)),
5     // ...
6   }
7 }
```

Agora podemos terminar o método `remove`:

```
1  const remove = async () => {
2    if (confirm("Remove?")) {
3      try {
4        loading.value = true
5        await CategoryService.delete(form.value.id)
6        router.push('/categories')
7      } catch (error) {
8        console.log(error)
9      } finally {
10       loading.value = false
11     }
12   }
13 }
```

Assim como fizemos no save, chamamos o `CategoryService.delete` utilizando a variável `loading` para informar ao usuário sobre o acesso ao servidor. Após remover a categoria, redirecionamos o fluxo da aplicação para `/categories`.

4. Produtos

Com a tela de categorias pronta, podemos criar a tela de produtos de forma mais rápida, bastando apenas copiar e colar o código e alterar onde é necessário.

Devemos configurar:

- Service
- Router
- Views
- Formulário

4.1 Service

Inicialmente podemos copiar o `CategoryService` e colar como `ProductService`, alterando onde está `categories` para `products` e `category` para `product`:

```
1  import HttpService from './HttpService'
2
3  export interface Product {
4    id?: number
5    name: string
6  }
7
8  const URI = '/products'
9
10 export const ProductService = {
11   getAll: async (): Promise<Product[]> =>
12     (await HttpService.get(URI)).data,
13   get: async (id:any): Promise<Product> =>
14     (await HttpService.get(URI + '/' + id)).data,
15   create: async (product: Product): Promise<Product> =>
16     (await HttpService.post(URI, product)).data,
17   update: async (product: Product): Promise<Product> =>
18     (await HttpService.put(URI + "/" + product.id, product)).data,
19   delete: async (id:any) => (await HttpService.delete(`${URI}/${id}`)),
20 }
```

Após criar o básico do ProductService precisamos definir melhor como será a interface Product, pois ela não possui apenas id e name.

Ao acessarmos `http://localhost:3000/products/` temos o seguinte resultado:

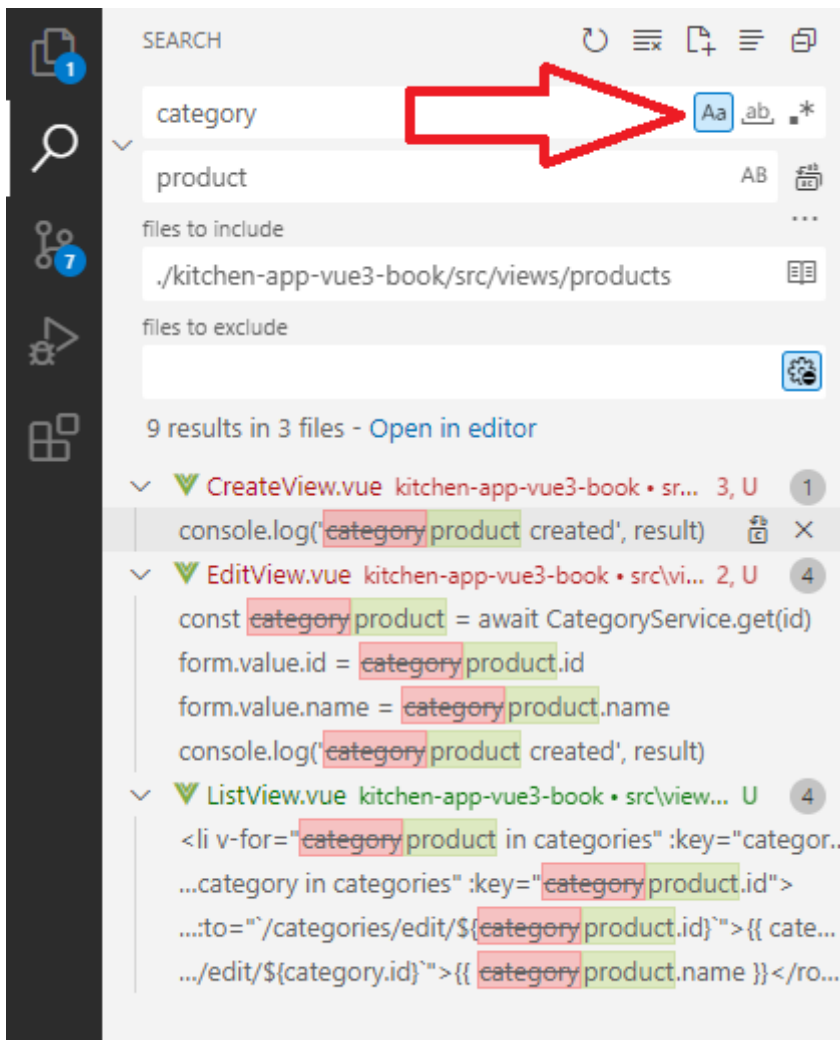
```
1  [  
2    {  
3      "id": "101",  
4      "name": "Beef",  
5      "supplier": "Maranata",  
6      "categoryId": 1  
7    },  
8    {  
9      "id": "102",  
10     "name": "Tomatoes",  
11     "supplier": "",  
12     "categoryId": 2  
13   },  
14   {  
15     "id": "103",  
16     "name": "Milk",  
17     "supplier": "Happy Coat",  
18     "categoryId": 3  
19   }  
20 ]
```

Além de id e name, temos o campo texto supplier e o campo categoryId, que podem ser adicionados a interface Product:

```
1  export interface Product {  
2    id?: number  
3    name: string,  
4    supplier?:string,  
5    categoryId:number  
6  }
```

Estrutura de Arquivos

Copie o diretório categories e cole como products. Use a ferramenta de search & replace do Vscod para alterar o nome das variáveis, conforme a figura a seguir:



Conforme o detalhe da figura, deixe a opção Match Case selecionada.

Altere:

- category para Product
- Category para Product

- categories para products
- Categories para products

4.2 Router

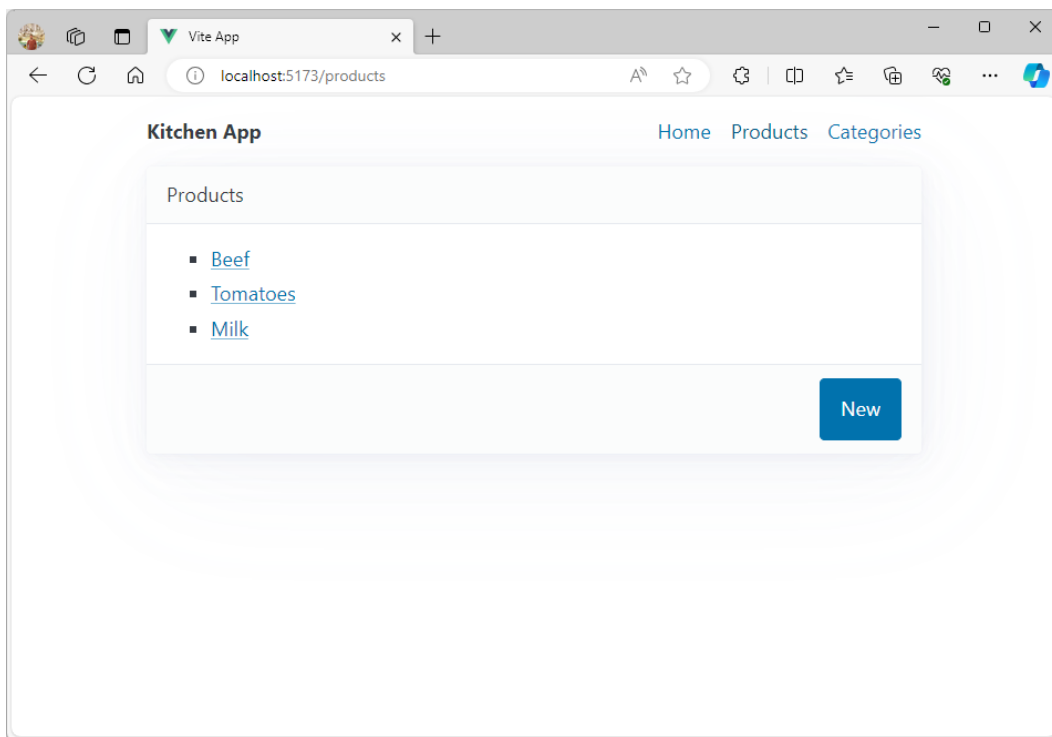
O router para o cadastro de produtos é exibido a seguir:

```
1 // src\router\index.ts
2 import { createRouter, createWebHistory } from 'vue-router'
3 import HomeView from '../views/HomeView.vue'
4 import CategoriesView from '../views/categories/IndexView.vue'
5
6 const router = createRouter({
7   history: createWebHistory(import.meta.env.BASE_URL),
8   routes: [
9     // ... another routes ...
10    {
11      path: '/products',
12      name: 'products',
13      component: () => import('../views/products/IndexView.vue'),
14      children: [
15        {
16          path: '',
17          name: 'ListProduct',
18          component: () => import('../views/products/ListView.vue')
19        },
20        {
21          path: 'create',
22          name: 'CreateProduct',
23          component: () => import('../views/products/CreateView.vue')
24        },
25        {
26          path: 'edit/:id',
27          name: 'EditProduct',
28          component: () => import('../views/products/EditView.vue')
29        }
30      ]
31    }
32  ]
33 })
34
35 export default router
```

Com as rotas criadas, podemos alterar o menu da aplicação e adicionar um link para /products em App.vue:

```
1  <script setup lang="ts">
2
3  </script>
4
5  <template>
6    <div class="container">
7      <nav>
8        <ul>
9          <li><strong>Kitchen App</strong></li>
10         </ul>
11         <ul>
12           <li><RouterLink to="/">Home</RouterLink></li>
13           <li><RouterLink to="/products">Products</RouterLink></li>
14           <li><RouterLink to="/categories">Categories</RouterLink></li>
15         </ul>
16       </nav>
17       <RouterView />
18     </div>
19 </template>
```

Ao acessarmos <http://localhost:5173/products> temos o seguinte resultado:



4.3 Exibindo Dados em Forma Tabular

Agora que criamos a base da tela de products, podemos realizar algumas modificações para deixá-la melhor.

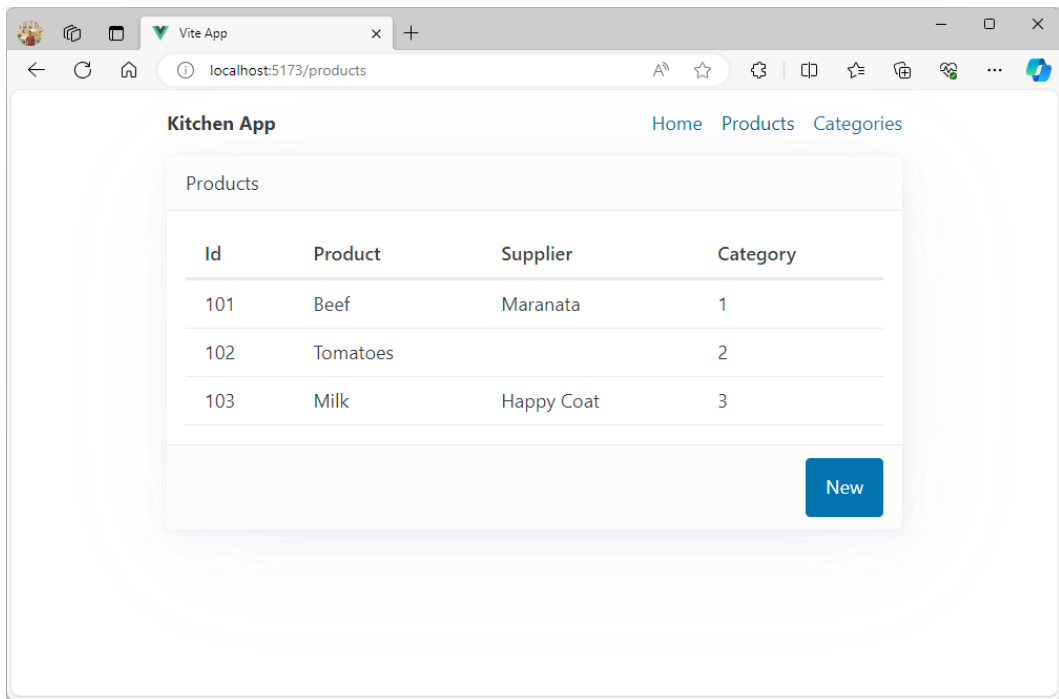
Ao invés de usar um `` para listar os dados, vamos usar `<table>`:

```
1 <script setup lang="ts">
2 // src\views\products\ListView.vue
3 import { onMounted, ref } from 'vue'
4 import { ProductService, type Product } from '../services/ProductService'
5
6 const products = ref<Product[]>()
7 const loading = ref<boolean>(false)
8
9 onMounted(async () => {
10   loading.value = true;
```



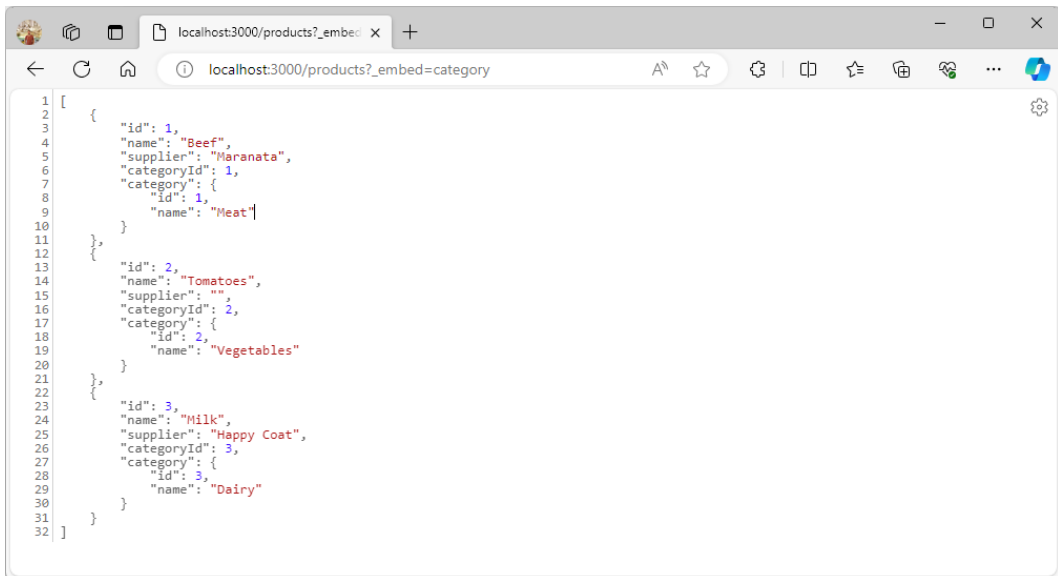
```
11     products.value = await ProductService.getAll()
12     loading.value = false;
13 })
14 </script>
15 <template>
16   <div>
17     <progress v-if="loading" />
18     <table v-else>
19       <thead>
20         <tr>
21           <th scope="col">Id</th>
22           <th scope="col">Product</th>
23           <th scope="col">Supplier</th>
24           <th scope="col">Category</th>
25         </tr>
26       </thead>
27       <tbody>
28         <tr v-for="product in products" :key="product.id">
29           <th>{{ product.id }}</th>
30           <td>{{ product.name }}</td>
31           <td>{{ product.supplier }}</td>
32           <td>{{ product.categoryId }}</td>
33         </tr>
34       </tbody>
35     </table>
36   </div>
37   <footer>
38     <router-link to="/products/create">
39       <button>New</button>
40     </router-link>
41   </footer>
42 </template>
```

Trocamos `` por `<table>` incluindo diversas informações. O resultado é exibido a seguir:



4.4 Corrigindo o CategoryId

Como pode ser visto na figura, a categoria do produto exibe um id ao invés do nome da categoria. Isso acontece porque o servidor backend está nos retornando apenas o id da categoria. Em um ambiente de produção real, iríamos pedir para que a equipe de backend adicionasse esta informação. No nosso exemplo de teste, com o servidor `json-server`, podemos adicionar uma configuração na url para trazer a categoria. Veja:



```

1  [
2    {
3      "id": 1,
4      "name": "Beef",
5      "supplier": "Maranata",
6      "categoryId": 1,
7      "category": {
8        "id": 1,
9        "name": "Meat"
10     }
11   },
12   {
13     "id": 2,
14     "name": "Tomatoes",
15     "supplier": " ",
16     "categoryId": 2,
17     "category": {
18       "id": 2,
19       "name": "Vegetables"
20     }
21   },
22   {
23     "id": 3,
24     "name": "Milk",
25     "supplier": "Happy Coat",
26     "categoryId": 3,
27     "category": {
28       "id": 3,
29       "name": "Dairy"
30     }
31   }
32 ]

```

Ao utilizarmos a url `http://localhost:3000/products?_embed=category` dizemos ao Json Server para trazer também os dados da categoria. Podemos alterar `ProductsService` para:

```

1  import HttpService from './HttpService'
2
3  export interface Product {
4    id?: number
5    name: string,
6    supplier?: string,
7    categoryId: number
8  }
9
10 const URI = '/products'
11
12 export const ProductService = {
13   getAll: async (): Promise<Product[]>
14     => (await HttpService.get(URI+"?_embed=category")).data,
15 }

```

Agora que configuramos o Service, podemos adicionar mais uma informação na interface `Product`. Vamos adicionar a propriedade `category`:

```

1  import type { Category } from './CategoryService'
2  import HttpService from './HttpService'
3
4  export interface Product {
5    id?: number
6    name: string,
7    supplier?:string,
8    categoryId:number,
9    category?:Category
10 }

```

A propriedade `category` pode ser nula já que, no formulário para criar um novo produto, não será necessário preencher `category`, apenas `categoryId`. Para exibir o nome da categoria no campo da tabela, ao invés de usar `product.categoryId` vamos utilizar:

```

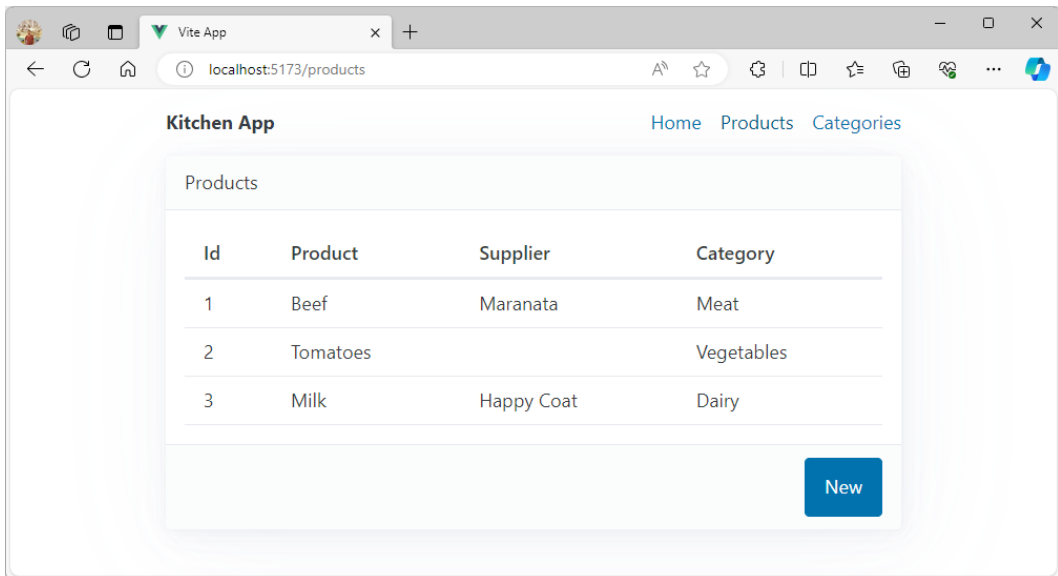
1  <td>{{ product.category?.name }}</td>

```

Por estarmos utilizando TypeScript e os plugins do Vue, podemos aproveitar a complementação de código para navegar pelos campos, conforma a figura a seguir:



O resultado após inserir o nome da categoria é exibido a seguir:



4.5 Formulário para Criar um Novo Produto

Após exibir os produtos, temos o botão New, que irá redirecionar para a tela Create Product. Esta tela possui apenas o campo Name. É necessário adicionar os outros campos, conforme o código a seguir:

```
1 <script setup lang="ts">
2 import router from '@/router'
3 import { ProductService, type Product } from '@/services/ProductService'
4 import { ref, reactive } from 'vue'
5
6 const form = ref<Product>({
7   name: '',
8   supplier: '',
9   categoryId: 0
10 })
11
12
13 const loading = ref<boolean>(false)
14
15 const save = async () => {
16   loading.value = true
```

```
17     const result = await ProductService.create(form.value)
18     console.log('product created', result)
19     loading.value = false
20     router.push('/products')
21 }
22 </script>
23 <template>
24   <h5>Create Product</h5>
25   <form>
26     <div class="grid">
27       <div>
28         <label>
29           Name
30           <input name="name" type="text" v-model="form.name"
31             placeholder="Product Name" required />
32         </label>
33       </div>
34       <div>
35         <label>
36           Supplier
37           <input name="name" type="text" v-model="form.supplier"
38             placeholder="Supplier Name" required />
39         </label>
40       </div>
41     </div>
42   </form>
43   <footer>
44     <router-link to="/products">
45       <button class="outline">Back</button>
46     </router-link>
47     <button :aria-busy="loading" :disabled="loading"
48       @click="save()">Save</button>
49   </footer>
50 </template>
```

Inicialmente, temos um formulário com dois campos: Name e Supplier. Para completar o formulário, é necessário adicionar o campo Category.

Adicionando o Campo Category

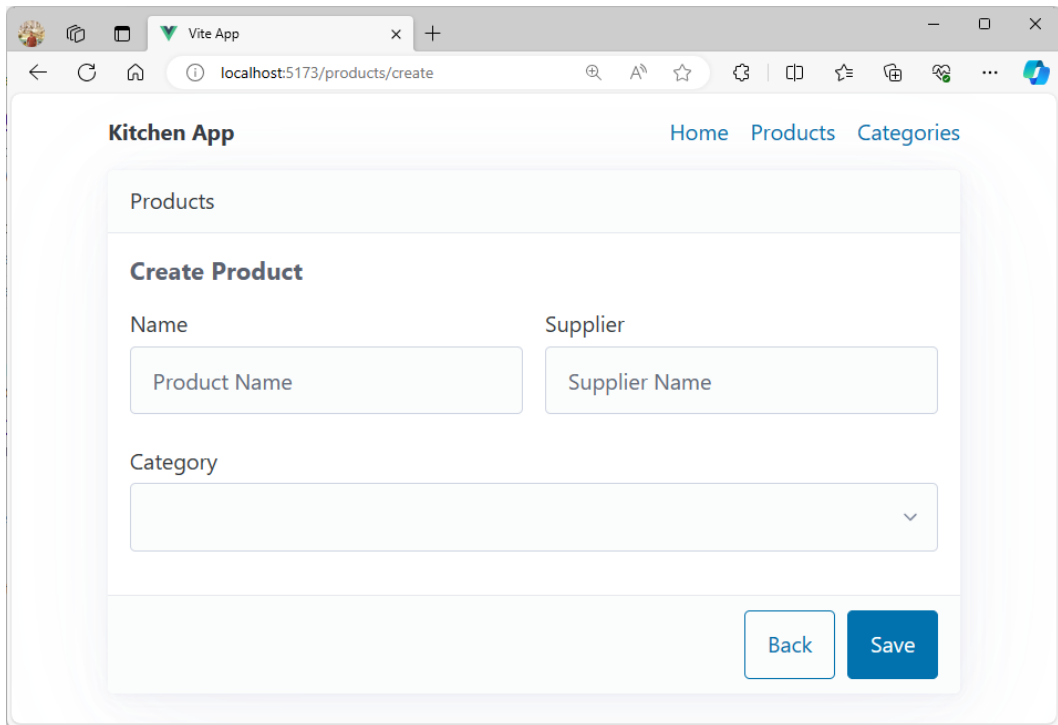
Para adicionar o campo category, precisamos inicialmente importar o CategoryService e criar uma variável categories para armazenar as categorias. Depois, criaremos uma caixa

de seleção e vincularemos o campo `categoryId` ao campo `categoryId` da interface `Product`.

```
1  <script setup lang="ts">
2    // imports
3
4    const form = ref<Product>({
5      name: '',
6      supplier: '',
7      categoryId: 0
8    })
9
10   const loading = ref<boolean>(false)
11   const categories = ref<Category[]>([])
12
13   onMounted(async () => {
14     categories.value = await CategoryService.getAll()
15   })
16
17   const save = async () => {
18     console.log(form.value)
19     // code
20   }
21 </script>
22 <template>
23   <h5>Create Product</h5>
24   <form>
25     <div class="grid">
26       <!-- FIELDS -->
27     </div>
28     <div>
29       <label>
30         Category
31         <select name="categories" aria-label="Select category..."
32           v-model="form.categoryId" required>
33           <option v-for="category in categories" :key="category.id"
34             :value="category.id">{{ category.name }}</option>
35         </select>
36       </label>
37     </div>
38   </form>
39   <footer>
40     <router-link to="/products">
41       <button class="outline">Back</button>
```

```
42     </router-link>
43     <button :aria-busy="loading" :disabled="loading"
44         @click="save()">Save</button>
45 </footer>
46 </template>
```

Neste código, criamos o campo Categories utilizando a tag select, nativa do html mas que o PicoCSS a estiliza e exibe uma caixa de seleção exibida a seguir:



The screenshot shows a web browser window with the address bar displaying 'localhost:5173/products/create'. The application is titled 'Kitchen App' and has navigation links for 'Home', 'Products', and 'Categories'. The main content area is titled 'Products' and contains a 'Create Product' form. The form has two input fields: 'Name' with the placeholder 'Product Name' and 'Supplier' with the placeholder 'Supplier Name'. Below these is a 'Category' dropdown menu. At the bottom right of the form are two buttons: 'Back' and 'Save'.

O select é ligado ao campo `categoryId` da variável `form` através da diretiva `v-model`. Os `option` são criados dinamicamente através de um `v-for` que percorre o array `categories` e exibe o nome de cada categoria.

Ao clicar no botão `save`, o método `save` é chamado. Este método, por enquanto, apenas exibe no console o valor do formulário. Em seguida, iremos implementar a chamada ao `ProductService` para criar um novo produto.


```
1  const save = async () => {
2    loading.value = true
3    const result = await ProductService.create(form.value)
4    console.log('product created', result)
5    loading.value = false
6    router.push('/products')
7  }
```

O save é semelhante ao save do Categories. A diferença é que, ao invés de chamar o CategoryService, chamamos o ProductService. O ProductService é exibido a seguir:

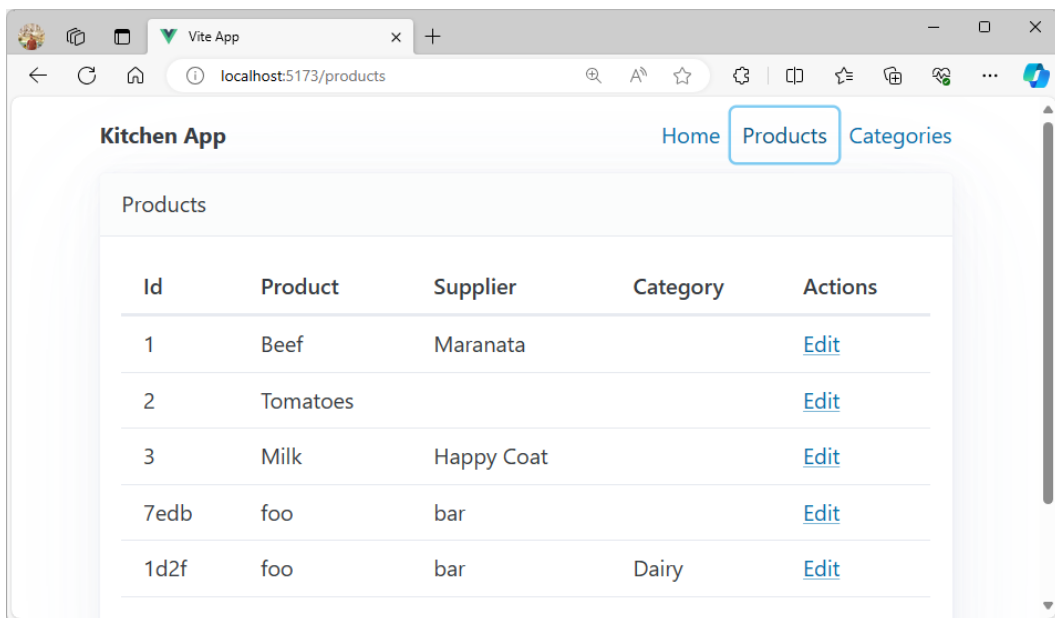
```
1  import type { Category } from './CategoryService'
2  import HttpService from './HttpService'
3
4  export interface Product {
5    id?: number
6    name: string,
7    supplier?:string,
8    categoryId:number,
9    category?:Category
10 }
11
12 const URI = '/products'
13
14 export const ProductService = {
15   // code...
16   create: async (product: Product): Promise<Product> =>
17     (await HttpService.post(URI, product)).data,
18   // code...
19 }
```

4.6 Editando um Produto

Para editar um produto, é necessário criar um link no componente products/ListView onde iremos repassar o id do produto. Com o id, podemos acessar o backend para obter os dados para o formulário, onde o usuário poderá editá-los.

```
1  <!-- src\views\products\ListView.vue -->
2  <!-- code -->
3
4  <table v-else>
5    <thead>
6      <tr>
7        <th scope="col">Id</th>
8        <th scope="col">Product</th>
9        <th scope="col">Supplier</th>
10       <th scope="col">Category</th>
11       <th scope="col">Actions</th>
12     </tr>
13   </thead>
14   <tbody>
15     <tr v-for="product in products" :key="product.id">
16       <th>{{ product.id }}</th>
17       <td>{{ product.name }}</td>
18       <td>{{ product.supplier }}</td>
19       <td>{{ product.category?.name }}</td>
20       <td>
21         <router-link :to="'`/products/edit/${product.id}`'">
22           Edit
23         </router-link>
24       </td>
25     </tr>
26   </tbody>
27 </table>
28
29 <!-- code -->
```

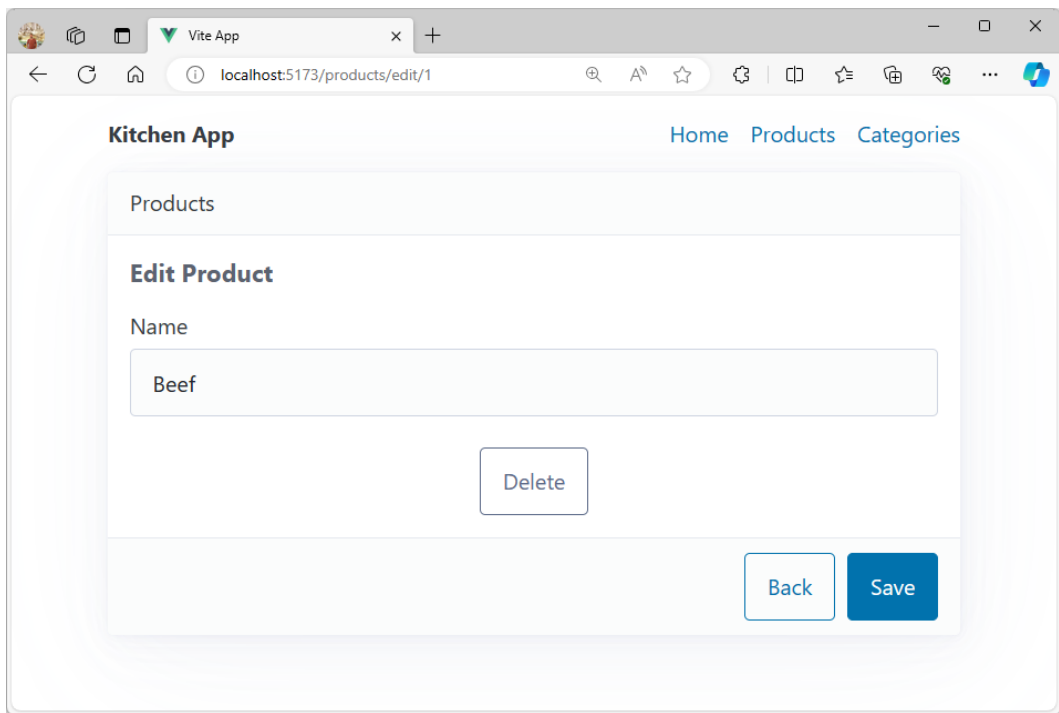
Adicionamos o `<th scope="col">Actions</th>` e a primeira ação é o link para editar, através do código `<router-link :to="'`/products/edit/'+product.id'">Edit</router-link>`:



Products				
Id	Product	Supplier	Category	Actions
1	Beef	Maranata		Edit
2	Tomatoes			Edit
3	Milk	Happy Coat		Edit
7edb	foo	bar		Edit
1d2f	foo	bar	Dairy	Edit

4.7 Formulário para Editar o Produto

Como nós copiamos e colamos a tela de categorias em produtos, ao editar um produto, temos o seguinte formulário:



Para editar o produto, é preciso adicionar os campos Supplier e Category, assim como foi feito no formulário para criar um produto:

```
1 <script setup lang="ts">
2 import router from '@/router'
3 import { ProductService, type Product } from '@/services/ProductService'
4 import { ref, onMounted } from 'vue'
5 import { useRoute } from 'vue-router'
6 import { Category, CategoryService } from '../../services/CategoryService';
7
8 const form = ref<Product>({
9   id: 0,
10  name: '',
11  supplier: '',
12  categoryId: 0
13 })
14
15 const route = useRoute()
16 const loading = ref<boolean>(false)
17 const categories = ref<Category[]>([]) // Add categories here
```

```
18
19
20 onMounted(async () => {
21   try {
22     loading.value = true
23     const id = route.params.id
24     const product = await ProductService.get(id)
25     form.value.id = product.id
26     form.value.name = product.name
27     form.value.supplier = product.supplier // supplier name
28     form.value.categoryId = product.categoryId // supplier category
29     categories.value = await CategoryService.getAll() // loading categories
30   } catch (error) {
31     console.log(error)
32   } finally {
33     loading.value = false
34   }
35 })
36
37 const save = async () => {
38   try {
39     loading.value = true
40     const result = await ProductService.update(form.value)
41     console.log('product updated', result)
42     loading.value = false
43     router.push('/products')
44   } catch (error) {
45     console.log(error)
46   } finally {
47     loading.value = false
48   }
49 }
50
51 const remove = async () => {
52   // ... code ...
53 }
54
55 </script>
56 <template>
57   <h5>Edit Product</h5>
58   <form>
59     <div class="grid">
60       <div>
```

```
61     <label>
62       Name
63       <input name="name" type="text"
64         v-model="form.name"
65         placeholder="Product Name" required />
66     </label>
67 </div>
68 <div>
69   <label>
70     Supplier
71     <input name="name" type="text"
72       v-model="form.supplier"
73       placeholder="Supplier Name" required />
74   </label>
75 </div>
76 </div>
77 <div>
78   <label>
79     Category
80     <select name="categories" aria-label="Select category..."
81       v-model="form.categoryId"
82       required>
83
84       <option v-for="category in categories"
85         :key="category.id"
86         :value="category.id">{{ category.name }}</option>
87
88     </select>
89   </label>
90 </div>
91 </form>
92 <div style="display: flex; justify-content: center;">
93   <button :aria-busy="loading" :disabled="loading"
94     @click="remove()" class="outline secondary">Delete</button>
95 </div>
96 <footer>
97   <router-link to="/products">
98     <button class="outline">Back</button>
99   </router-link>
100   <button :aria-busy="loading" :disabled="loading"
101     @click="save()">Save</button>
102 </footer>
103 </template>
```

Este é o código completo para editar um Produto. É necessário criar a variável `categories` para carregar as categorias, e adicionar o campo `Category` no formulário. O campo `Category` referencia a propriedade `categoryId` de `form.value` que é repassada no método `save`.

4.8 Remover um Produto

Como copiamos e colamos a tela de categorias em produtos, não é necessário alterar o código para remover um produto.

```
1  const remove = async () => {
2    if (confirm("Remove?")) {
3      try {
4        loading.value = true
5        await ProductService.delete(form.value.id)
6        router.push('/products')
7      } catch (error) {
8        console.log(error)
9      } finally {
10       loading.value = false
11     }
12   }
13 }
```

5. Estoque de Produtos

Com a tela de produtos pronta, podemos adicionar estes produtos ao estoque.

O produto adicionado possui os seguintes campos:

- Id
- Produto
- Preço
- Data de Validade
- Data de Entrada
- Quantidade

Estes campos correspondem ao seguinte json:

```
1 {  
2   "id": "1",  
3   "productId": 1,  
4   "quantity": 50,  
5   "expires": "2023-08-31",  
6   "price": 10.99,  
7   "added": "2023-08-01"  
8 }
```

A ideia desta tela é que possamos adicionar produtos e sua validade correspondente, e desta forma controlar quando os produtos estarão vencidos.

5.1 Service

O Service que conecta ao servidor backend é exibido a seguir:


```
1 import type { Product } from "../ProductService"
2 import HttpService from '../HttpService'
3
4 export interface Stock {
5   id?: number
6   productId?: number
7   quantity: number
8   expires: string
9   added: string
10  price: number
11  product?: Product
12 }
13
14 const URI = '/stock'
15
16 export const StockService = {
17   getAll: async (): Promise<Stock[]> =>
18     (await HttpService.get(URI)).data,
19   getById: async (id: string): Promise<Stock> =>
20     (await HttpService.get(`${URI}/${id}`)).data,
21   delete: async (id: number): Promise<Stock> =>
22     (await HttpService.delete(`${URI}/${id}`)).data,
23   create: async (Stock: Stock): Promise<Stock> =>
24     (await HttpService.post(URI, Stock)).data,
25   edit: async (Stock: Stock): Promise<Stock> =>
26     (await HttpService.put(`${URI}/${Stock.id}`, Stock)).data
27 }
```

Como podemos ver, o service é semelhante aos outros, contendo os mesmos métodos para incluir, editar, remover um Stock.

5.2 Adicionando Produtos Ao Estoque

No componente Home, vamos adicionar um botão para incluir um produto no estoque:

```
1  <!-- src\views\HomeView.vue -->
2  <script setup lang="ts">
3
4  </script>
5
6  <template>
7
8    <main class="container">
9      <RouterLink to="/add">Add Stock</RouterLink>
10    </main>
11
12
13  </template>
```

O componente para adicionar um novo produto ao estoque pode ser criado em `src\views\stock\AddView.vue`:

```
1  <script setup lang="ts">
2
3  </script>
4
5  <template>
6
7    <main class="container">
8      Add Products to Stock
9    </main>
10
11
12  </template>
```

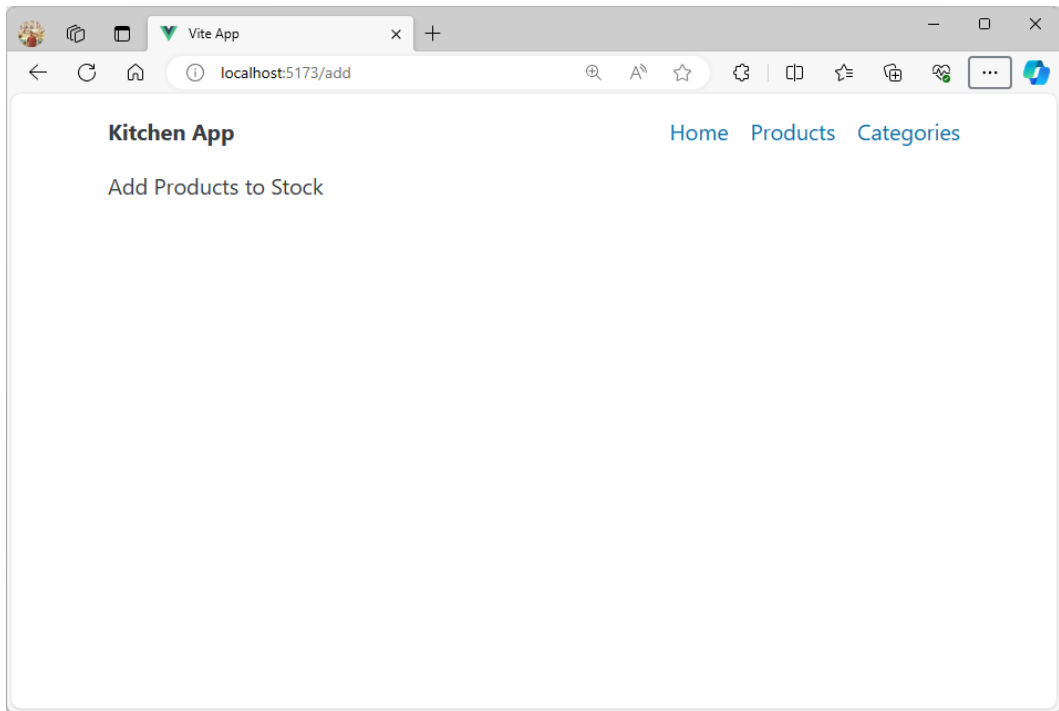
5.3 Router

Com o componente criado, podemos adicioná-lo ao router:

```
1 import { createRouter, createWebHistory } from 'vue-router'
2 import HomeView from '../views/HomeView.vue'
3 import CategoriesView from '../views/categories/IndexView.vue'
4 import AddView from '@views/stock/AddView.vue'
5
6 const router = createRouter({
7   history: createWebHistory(import.meta.env.BASE_URL),
8   routes: [
9     {
10      path: '/',
11      name: 'home',
12      component: HomeView
13    },
14    {
15      path: '/add',
16      name: 'addStock',
17      component: AddView
18    },
19    // routes ...
20  ]
21 })
22
23 export default router
```

5.4 Formulário para Adicionar um Novo Produto Ao Estoque

Ao clicar no link Add Stock, o componente AddView será exibido:



Este componente deverá adicionar um produto ao estoque, então precisamos criar um formulário com os seguintes campos:

- Produto (Campo select exibindo todos os produtos)
- Preço
- Quantidade
- Data de Validade
- Data de Entrada

O formulário para a criação destes campos é exibido a seguir:

```
1  <script setup lang="ts">
2  import { onMounted, ref } from 'vue';
3  import { Stock } from '../services/StockService';
4  import { Product, ProductService } from '../services/ProductService';
5
6  const products = ref<Product[]>([])
7  const loading = ref<boolean>(false)
8
9  const form = ref<Stock>({
10    productId: 0,
11    price: 0,
12    expires: '',
13    added: new Date().toISOString().slice(0,10),
14    quantity: 0
15  })
16
17  onMounted(async () => {
18    products.value = await ProductService.getAll()
19  })
20
21  const save = async ($event) => {
22    $event.preventDefault()
23    // TODO
24  }
25  </script>
26
27  <template>
28    <form @submit="save">
29      <article>
30        <header>Add Stock</header>
31        <div class="grid m-10">
32          <div>
33            <label>Product</label>
34            <select id="productId" name="productId"
35              v-model="form.productId">
36              <option v-for="product in products"
37                :key="product.id" :value="product.id">
38                {{ product.name }} ({{ product.supplier }})
39            </option>
40          </select>
41        </div>
42        <div>
43          <label>Quantity</label>
```

```
44     <input required class="input"
45           v-model="form.quantity" type="number"
46           min="0" id="quantity" name="quantity"
47           placeholder="Quantity" />
48   </div>
49 </div>
50 <div class="grid m-10">
51   <div>
52     <label>Expires</label>
53     <input required class="input"
54           v-model="form.expires" type="date"
55           id="expires" name="expires"
56           placeholder="Expires at" />
57   </div>
58   <div>
59     <label>Price</label>
60     <input required class="input"
61           v-model="form.price" type="float"
62           id="price" name="price" placeholder="Price" />
63   </div>
64 </div>
65 <footer>
66   <button :disabled="loading" type="submit">Add</button>
67 </footer>
68 </article>
69 <RouterLink to="/">
70   <a class="button is-light">Back</a>
71 </RouterLink>
72 </form>
73 </template>
74
75 <style scoped>
76 form>div {
77   margin: 20px;
78 }
79
80 .m-10 {
81   margin: 10px;
82 }
83
84 footer {
85   display: flex;
86   justify-content: center;
```

```
87 }  
88  
89 footer>button {  
90   max-width: 200px;  
91   margin: 0px;  
92 }  
93 </style>
```

Temos agora um formulário um pouco mais complexo que os anteriores. Inicialmente criamos a variável `products` que serão os produtos a serem selecionados de uma lista. Depois, criamos a variável `form` que será a estrutura que receberá os dados do formulário. No método `onMounted`, buscamos os produtos e atribuímos a variável `products`. Por fim, criamos o método `save` que será chamado quando o formulário for enviado. Este método é exibido a seguir:

```
1  const save = async ($event) => {  
2    $event.preventDefault()  
3    loading.value = true  
4    try {  
5      await StockService.create(form.value)  
6      router.push('/')  
7    } catch (error) {  
8      console.log(error)  
9    } finally {  
10     loading.value = false  
11   }  
12 }
```

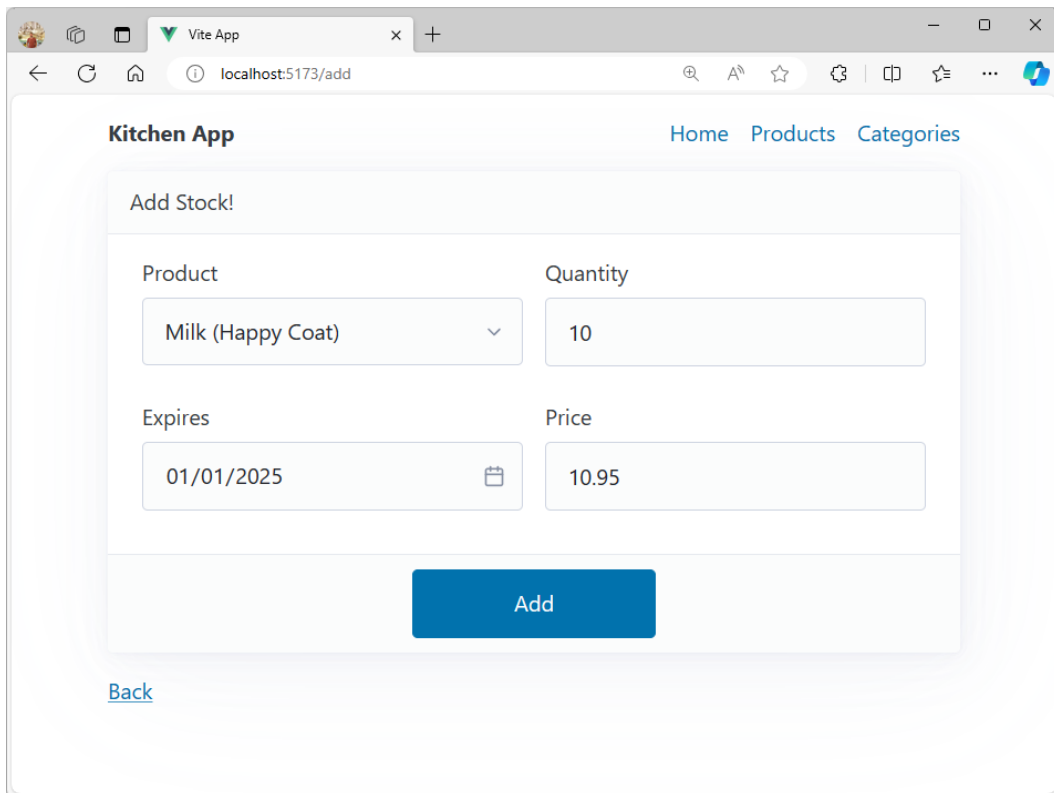
Ao contrário os outros formulários, que possuía no botão `Save`, neste criamos o método `save` na submissão do formulário, por isso temos `<form @submit="save">`. Esta é apenas uma outra forma de atribuir um botão para a submissão do formulário.

Também temos algumas novas propriedades dos campos que são nativas do `html`, e não necessariamente do `Vue`. Por exemplo, o campo `Quantity` aceita apenas valores positivos. Por fim, o campo `Price` aceita apenas valores positivos e decimais.

Na definição do `form`, onde temos `const form = ref<Stock>` criamos as variáveis que correspondem a cada campo do formulário. O campo `added` será preenchido automaticamente com a data de hoje.

Algumas definições desta `view` estão sendo configuradas pelo `css`, através da tag `style`, como por exemplo o `footer` que faz o botão de submissão ficar centralizado.

O resultado deste formulário é semelhante a figura a seguir:



The screenshot shows a web browser window with the address bar displaying 'localhost:5173/add'. The browser tab is labeled 'Vite App'. The page title is 'Kitchen App'. The navigation bar includes links for 'Home', 'Products', and 'Categories'. The main content area is titled 'Add Stock!' and contains a form with the following fields:

Product	Quantity
Milk (Happy Coat) ▼	10

Expires	Price
01/01/2025 📅	10.95

Below the form is a blue 'Add' button. At the bottom left, there is a 'Back' link.

Quando o usuário adiciona um produto ao estoque, o método `save` irá redirecionar para '/', que é o componente `Home` que será criado a seguir.

5.5 Exibindo a Lista de Produtos no Estoque

Agora que criamos a funcionalidade de adicionar produtos ao estoque, podemos exibir uma lista destes produtos.

No componente `HomeView`, vamos criar uma variável chamada `stockList`:


```

1  <script setup lang="ts">
2  import { onMounted, ref } from 'vue';
3  import { Stock, StockService } from '../services/StockService';
4
5
6  const stockList = ref<Stock[]>([])
7
8  onMounted(async () => {
9    stockList.value = await StockService.getAll()
10 })
11 </script>
12
13 <template>
14   <main class="container">
15     <RouterLink to="/add">Add Stock</RouterLink>
16   </main>
17 </template>

```

A constante `stockList` será a lista de todos os produtos do estoque. Por fim, criamos a lista de itens logo abaixo do link Add Stock:

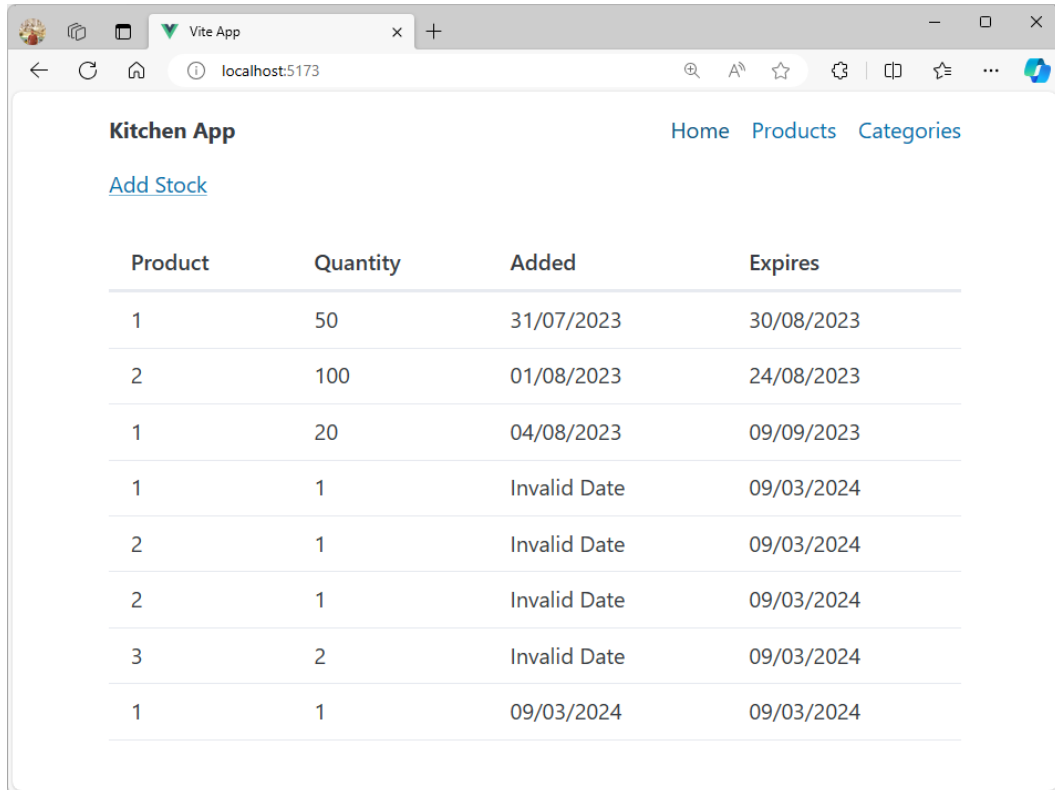
```

1  <template>
2    <main class="container">
3      <RouterLink to="/add">Add Stock</RouterLink>
4      <br/><br/>
5      <table>
6        <thead>
7          <tr>
8            <th>Product</th>
9            <th>Quantity</th>
10           <th>Added</th>
11           <th>Expires</th>
12         </tr>
13       </thead>
14       <tbody>
15         <tr v-for="stock in stockList" :key="stock.id">
16           <td>{{ stock.productId }}</td>
17           <td>{{ stock.quantity }}</td>
18           <td>{{ new Date(stock.added).toLocaleDateString() }}</td>
19           <td>{{ new Date(stock.expires).toLocaleDateString() }}</td>
20         </tr>
21       </tbody>

```

```
22     </table>
23   </main>
24 </template>
```

Ainda que de forma prematura, já temos uma ideia de como exibir a lista de estoque, algo semelhante a figura a seguir:



Product	Quantity	Added	Expires
1	50	31/07/2023	30/08/2023
2	100	01/08/2023	24/08/2023
1	20	04/08/2023	09/09/2023
1	1	Invalid Date	09/03/2024
2	1	Invalid Date	09/03/2024
2	1	Invalid Date	09/03/2024
3	2	Invalid Date	09/03/2024
1	1	09/03/2024	09/03/2024

5.6 Ordenando a Lista de Stock

Para melhorar a exibição desta lista, podemos adicionar as seguintes funcionalidades:

- Ordenar a lista de estoque pelo vencimento
- Exibir o nome do produto
- Não exibir produtos que tenham a quantidade igual a zero

- Diferenciar produtos que estão vencidos

Algumas destas funcionalidades podem ser realizadas no backend. Por exemplo, a ordenação pode ser realizada adicionado um parâmetro chamado “_sort” na url que busca a lista de estoque. Também podemos definir que uma quantidade seja maior ou menor que um valor, através da propriedade `gte` (greater than or equal to). Estas configurações são específicas do servidor, no nosso caso utilizamos o `json_server` e na sua [página¹](#) existem diversos tipos de parâmetros que podem ser configurados.

Para o exemplo, podemos adicionar um parâmetro na url que ordena a lista de estoque por vencimento. Por exemplo, podemos adicionar `?_sort=expires&_order=asc&_expand=product&quantity_gte=1` na url do método `StockService.getAll`, da seguinte forma:

```
1 // src\services\StockService.ts
2 // code
3 export const StockService = {
4   getAll: async (): Promise<Stock[]> =>
5     (await HttpService.get(
6       URI + '?_sort=-expires&_embed=product&quantity_gte=1')).data,
7   // code
8 }
```

Estes parâmetros indicam que a lista de estoque será ordenada por vencimento (de forma descendente) e que os produtos que possuírem uma quantidade menor que zero não serão exibidos.

Somente para testar, altere o arquivo `db.json` alterando a `quantity` para 0 de algum produto, e veja que ele não aparece mais na lista.

5.7 Exibindo Itens de Estoque Perto da Validade

Podemos usar uma função que irá avaliar se uma determinada data está perto da validade, como por exemplo uma data dentro de período de trinta dias.

Para isso, criamos a seguinte função no arquivo `HomeView`:

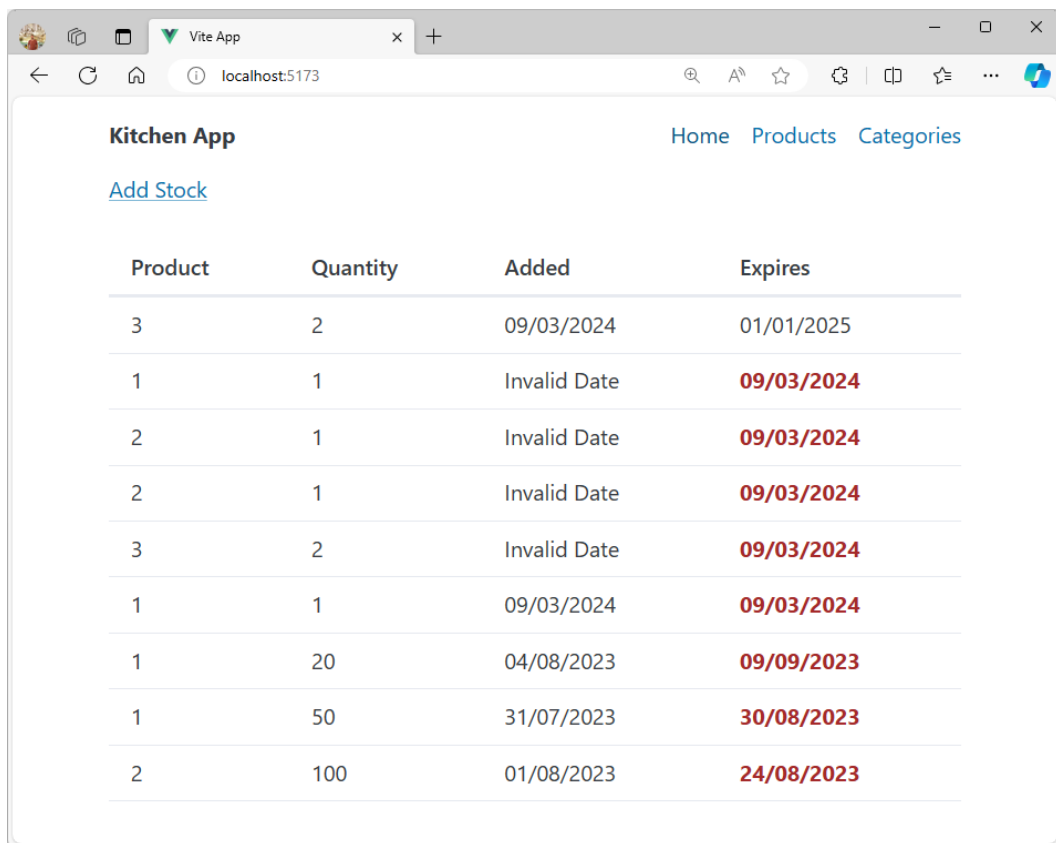
¹<https://github.com/typicode/json-server>

```
1  const isDateLessOneMonth =  
2    (date: string) =>  
3    (new Date().getTime() - new Date(date).getTime())  
4    / (1000 * 3600 * 24) > -30
```

Esta função retorna verdadeiro se uma data repassada está no período de 30 dias. No template, podemos renderizar um <td> diferenciado para itens que estão perto da validade:

```
1  <!-- code -->  
2  
3  <tr v-for="stock in stockList" :key="stock.id">  
4    <td>{{ stock.productId }}</td>  
5    <td>{{ stock.quantity }}</td>  
6    <td>{{ new Date(stock.added).toLocaleDateString() }}</td>  
7    <td v-if="isDateLessOneMonth(stock.expires)" class="warning">  
8      {{ new Date(stock.expires).toLocaleDateString() }}  
9    </td>  
10   <td v-else>  
11     {{ new Date(stock.expires).toLocaleDateString() }}  
12   </td>  
13 </tr>  
14  
15 <!-- code -->  
16  
17 <style scoped>  
18  
19 .warning {  
20   color: brown;  
21   font-weight: bold;  
22 }  
23  
24 </style>
```

Ao adicionarmos esta funcionalidade, temos na lista de estoque os itens que estão perto da validade sendo renderizados em vermelho:



Product	Quantity	Added	Expires
3	2	09/03/2024	01/01/2025
1	1	Invalid Date	09/03/2024
2	1	Invalid Date	09/03/2024
2	1	Invalid Date	09/03/2024
3	2	Invalid Date	09/03/2024
1	1	09/03/2024	09/03/2024
1	20	04/08/2023	09/09/2023
1	50	31/07/2023	30/08/2023
2	100	01/08/2023	24/08/2023

5.8 Incluindo o Nome do Produto

Ao adicionarmos `embed=products` na url de acesso ao servidor backend, poderemos ter os dados de produtos sendo retornados na lista de estoque, incluindo o nome do produto. Então podemos alterar o template para:

```
1 <!-- code -->
2
3 <tr v-for="stock in stockList" :key="stock.id">
4   <td>
5     <RouterLink :to="`~/edit/${stock.id}`">
6       {{ stock.product.name }}
7     </RouterLink>
8   </td>
9   <td>{{ stock.quantity }}</td>
10  <td>{{ new Date(stock.added).toLocaleDateString() }}</td>
11  <td v-if="isDateLessOneMonth(stock.expires)" class="warning">
12    {{ new Date(stock.expires).toLocaleDateString() }}
13  </td>
14  <td v-else>
15    {{ new Date(stock.expires).toLocaleDateString() }}
16  </td>
17 </tr>
```

Veja que a primeira coluna aparece com o nome do produto e também chama a url `/edit/:id` para editar o item de estoque, que será visto a seguir.

5.9 Editando um Produto do Estoque

A edição dos produtos do estoque segue o mesmo caminho da edição das categorias e dos produtos. Para isso, vamos criar um componente chamado `EditView.vue`:

```
1 <script setup lang="ts">
2 import { onMounted, ref } from 'vue';
3 import { type Stock, StockService } from '../services/StockService';
4 import { type Product, ProductService } from '../services/ProductService';
5 import router from '../router';
6 import { useRoute } from 'vue-router'
7
8 const form = ref<Stock>({
9   productId: 0,
10  price: 0,
11  expires: '',
12  quantity: 0,
13  added: new Date().toISOString().slice(0,10)
14 })
```

```
15 const products = ref<Product[]>([])
16 const loading = ref<boolean>(false)
17 const route = useRoute()
18
19 onMounted(async () => {
20   products.value = await ProductService.getAll()
21   const id = route.params.id
22   form.value = await StockService.getById(id);
23 })
24
25 const save = async ($event) => {
26   $event.preventDefault()
27   loading.value = true
28   try {
29     await StockService.edit(form.value)
30     router.push('/')
31   } catch (error) {
32     console.log(error)
33   } finally {
34     loading.value = false
35   }
36 }
37 </script>
38
39 <template>
40   <form @submit="save">
41     <article>
42       <header>Add Stock!</header>
43       <div class="grid m-10">
44         <div>
45           <label>Product</label>
46           <select id="productId" name="productId"
47             v-model="form.productId">
48             <option v-for="product in products"
49               :key="product.id" :value="product.id">
50               {{ product.name }} ({{ product.supplier }})
51             </option>
52           </select>
53         </div>
54         <div>
55           <label>Quantity</label>
56           <input required class="input" v-model="form.quantity"
57             type="number" min="0" id="quantity" name="quantity">
```

```
58         placeholder="Quantity" />
59     </div>
60 </div>
61 <div class="grid m-10">
62     <div>
63         <label>Expires</label>
64         <input required class="input" v-model="form.expires"
65             type="date" id="expires" name="expires"
66             placeholder="Expires at" />
67     </div>
68     <div>
69         <label>Price</label>
70         <input required class="input" v-model="form.price"
71             type="float" id="price" name="price" placeholder="Price" />
72     </div>
73 </div>
74 <footer>
75     <button :disabled="loading" type="submit">Save</button>
76 </footer>
77 </article>
78 <RouterLink to="/"><a class="button is-light">Back</a></RouterLink>
79 </form>
80 </template>
81
82 <style scoped>
83 form>div {
84     margin: 20px;
85 }
86
87 .m-10 {
88     margin: 10px;
89 }
90
91 footer {
92     display: flex;
93     justify-content: center;
94 }
95
96 footer>button {
97     max-width: 200px;
98     margin: 0px;
99 }
100 </style>
```


Apesar do conteúdo extenso, o funcionamento do componente `EditView.vue` é semelhante ao `AddView.vue` no formulário, e semelhante as outras telas de edição.

Um detalhe importante é que, se você colocar a quantidade de estoque em zero, o item do produto não aparecerá mais na lista de estoques. Alterando a data de expiração também temos a lista de estoques atualizada, sendo que o item pode ficar em vermelho ou não.

6. Considerações Finais

Neste livro aprendemos a criar um pequeno sistema para gerenciar o estoque de uma cozinha. Através do Vue 3 podemos ver como é fácil criar telas com formulários e tabelas, adicionando dados através de um servidor backend e exibir aos usuários estes dados.

Caso você tenha encontrado algum erro no código ou na tradução, crie uma Issue [neste link](#)¹.

Você também pode sugerir novos conteúdos para o livro [neste link](#)²

¹<https://github.com/danielschmitz/kitchen-app-vue3-book/issues>

²<https://github.com/danielschmitz/kitchen-app-vue3-book/discussions>