Preparação

Diretórios e pastas

Antes de iniciar, crie uma pasta para este projeto dentro de alguma unidade de disco de sua máquina. Nossa sugestão é que, dentro dessa pasta, você crie uma pasta **back-end** e outra **front-end** dentro dessa.

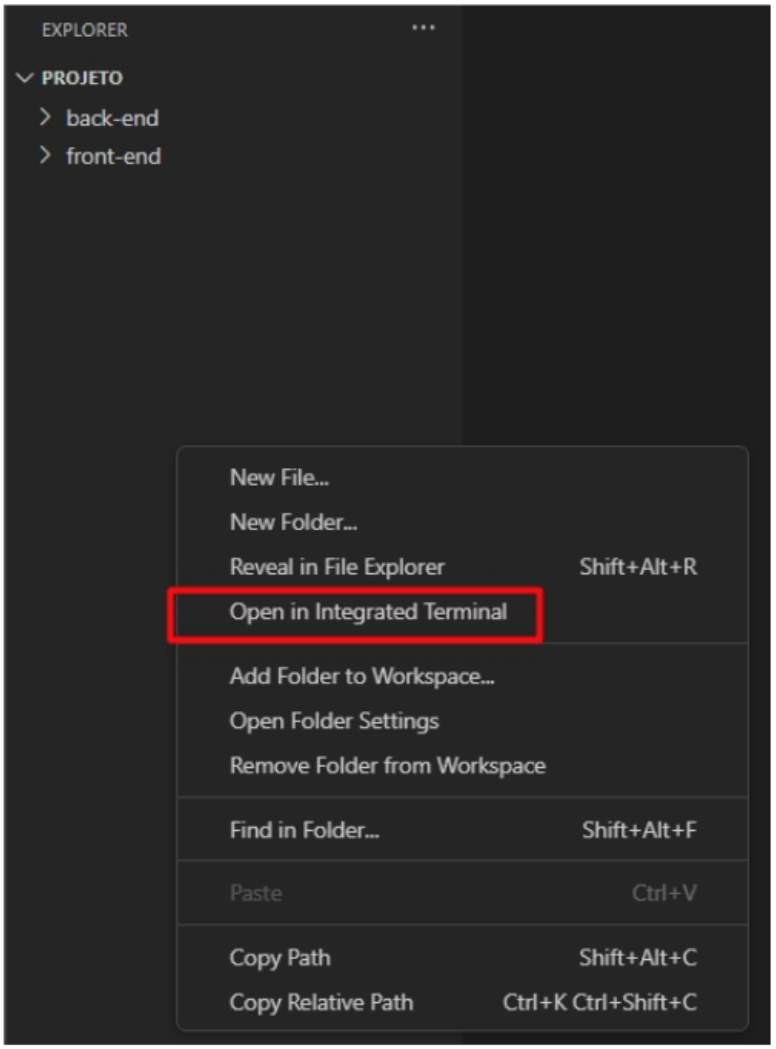
Instalação do Visual Studio Code

No Visual Studio Code é que vamos fazer todo nosso desenvolvimento. Desde a configuração e instalação de pacotes, como os desenvolvimentos back e front-end.

Para instalar o Visual Studio Code, você pode seguir os seguintes passos:

1. Acesse o site oficial do VS Code (https://code.visualstudio.com/)
   Clique no botão "Download" para baixar a versão mais recente para o seu sistema operacional
2. Execute o arquivo baixado e siga as instruções para instalar o aplicativo
3. Certifique-se de deixar marcada a opção "**abrir com code**" que será extremamente útil.
4. Quando a instalação for concluída, abra o VS Code e comece a usar! Ou, se preferir, vá até a pasta criada para o projeto, clique com o botão direito, e selecione "abrir com code".
5. Para abrir o terminal, você pode pressionar CTRL + ‘ (aspas-simples) ou ir até o navegador de pastas à esquerda, clicar com o botão direito, e selecionar a opção "Open in integrated terminal" ou "Abrir no terminal integrado" se estiver em português.

5. Para abrir o terminal, você pode pressionar CTRL + ‘ (aspas-simples) ou ir até o navegador de pastas à esquerda, clicar com o botão direito, e selecionar a opção “Open in integrated terminal” ou “Abrir no terminal integrado” se estiver em português.



Instalação do Node JS

Antes de prosseguir, precisamos instalar o Node JS em nossa máquina.

Nessa primeira parte, vamos preparar todo nosso ambiente em Node e React para executarmos nossas tarefas.

Para isso, acesse o site oficial do Node.js em https://nodejs.org/en/
- Clique no botão "Download" na página inicial
- Escolha a versão LTS (Long Term Support) do Node.js para a sua plataforma (Windows, MacOS, Linux)
- Execute o arquivo de instalação baixado e siga as instruções fornecidas pelo instalador
- Verifique se a instalação foi bem-sucedida abrindo o terminal e digitando "node -v" (sem as aspas) e pressionando enter. Isso deve exibir a versão do Node.js instalada.

Instalação do MySQL

Para nosso projeto, vamos utilizar o banco de dados MySQL.

- Faça o download do pacote de instalação do MySQL para o seu sistema operacional a partir do site oficial do MySQL (https://dev.mysql.com/downloads/mysql/).

- Execute o arquivo de instalação baixado. Siga as instruções do assistente de instalação, escolhendo as opções de acordo com suas necessidades.

- Durante a instalação, você será solicitado a definir a **senha do usuário root** do MySQL. Certifique-se de anotá-la, pois ela será necessária para acessar o banco de dados.

- Depois que a instalação for concluída, verifique se o MySQL está funcionando corretamente executando o comando "mysql -u root -p" no terminal, e digite a senha do root.

- Se a conexão for bem-sucedida, você será redirecionado para o prompt do MySQL, indicando que o MySQL está instalado e funcionando corretamente.

Em seguida, baixe um gerenciador para trabalhar com o seu MySQL. Nós sugerimos o HeidiSQL. Mas, caso você já tenha familiaridade com algum outro (como o DBeaver ou Workbench), fique à vontade.

Para instalar o **HeidiSQL**, você pode seguir os seguintes passos:

- Acesse o site oficial do HeidiSQL em https://www.heidisql.com/download.php
- Selecione o sistema operacional que você está usando (Windows, Linux ou macOS)
- Faça o download do arquivo de instalação correspondente
- Abra o arquivo baixado e siga as instruções para instalar o software
- Uma vez que a instalação esteja completa, abra o HeidiSQL e conecte-se ao seu banco de dados MySQL local, usando os dados criados na etapa anterior.

Criando a base de Dados

Crie um novo banco de dados para o seu projeto. Vamos usar esse banco para criar a tabela utilizada em nosso projeto. Existem duas formas de criarmos a estrutura do banco de dados: code-first ou database-first.

Na abordagem code-first, nós implementamos a model via código, e o Express irá gerar a estrutura na sua base de dados.

Na segunda abordagem, nós fazemos a estrutura da base de dados manualmente, e só utilizamos o express para consultar e acessar os dados.

Para fins deste exemplo, vamos trabalhar com **code-first**.

Para uma abordagem completa sobre o banco de dados MySQL, fique à vontade para acessar nosso livro específico sobre esse SGDB: https://mazon.com.br/mysql-essencial

Configuração do projeto em Node

No back-end de nosso projeto, primeiro precisamos inicializar um novo projeto. Navegue até a pasta *back-end*, abra o terminal no Visual Studio Code e digite o comando:

npm init

Preencha as informações solicitadas. Dê um nome para o projeto e confirme os próximos passos. Após esse procedimento, será criado automaticamente um arquivo chamado package.json, que contém as configurações do seu projeto.

Crie um arquivo "app.js" ou "index.js" na raiz do projeto.

Agora, o passo a passo abaixo mostra a sequência exata da instalação dos pacotes necessários para execução do projeto.

1. *Instalação do Dotenv* - Para gerenciar variáveis de ambiente
   npm install dotenv

2. *Instalação do Sequelize* - Um ORM para manipular a base de dados (essencial para
   implementarmos a estrutura da base através da abordagem code-first)
   npm install sequelize

3. *Instalação do Mysql2* - Para fazer a conexão com a base de dados do MySQL
   npm install mysql2

4. *Instalação do Express* - Framework para controle das rotas
   npm install express

5. *Instalação do Cors* - Pacote para permitir o acesso de qualquer url
   npm install cors

Instalação e configuração do React

Para instalar e criar um projeto React, siga os seguintes passos:

1. Instale o Create React App (CRA) globalmente na sua máquina usando o seguinte comando:
   npm install -g create-react-app

2. Crie um novo projeto React com o seguinte comando:
   npx create-react-app task-list

3. Entre na pasta front-end dentro da pasta do projeto criado:
   cd task-list/front-end

4. Inicie o projeto:
   npm start

5. Acesse o projeto no seu navegador, geralmente em *http://localhost:3000/*

Projeto

Tudo instalado e devidamente configurado, vamos ao projeto. Para fins didáticos, vamos dividir os desenvolvimentos em back-end e front-end.

O projeto a ser desenvolvido é um gerenciador de tarefas simples, com uma única tabela. Será possível exibir dados, consultar e excluir.

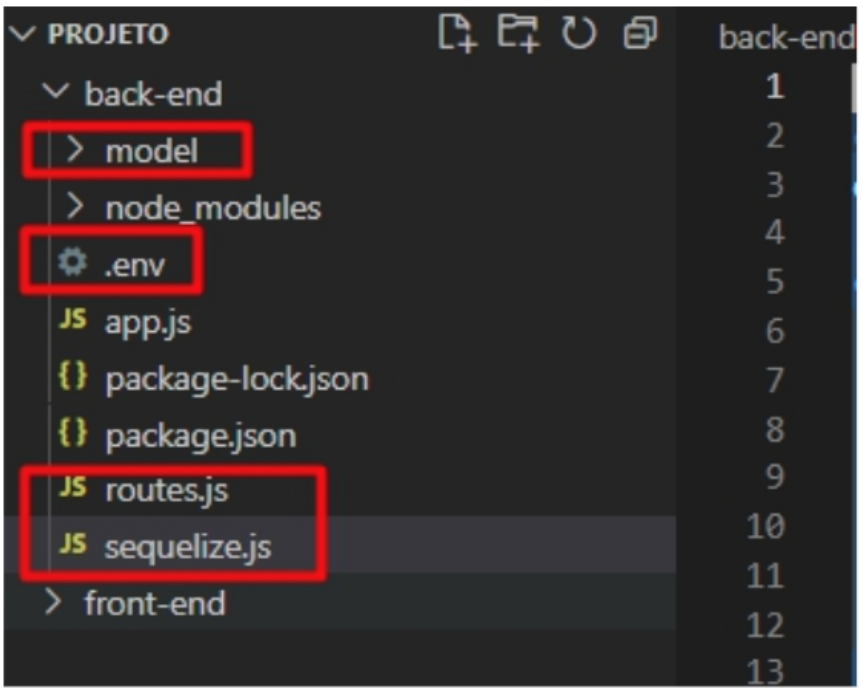
Por ser um exemplo de projeto real, feito de ponta a ponta, você vai ser capaz de entender todas as etapas do desenvolvimento, além de trabalhar com um número muito grande de ferramentas, pacotes, estratégias, padrões, tecnologias.

Back-end

Estrutura

Na imagem abaixo, você vê a estrutura sugerida para você criar as pastas dentro de back-end.

1. Crie uma pasta model. Essa pasta irá conter os arquivos que montam a estrutura da base de dados. Neste exemplo, terá um único arquivo (Task)
2. .env - Arquivo de variáveis de ambiente. Crie um arquivo com esse nome (apenas um ponto seguido das letras *env*). O pacote *dotenv* que instalamos vai se encarregar de ler e converter os dados.
3. routes.js - O arquivo de rotas de nosso projeto.
4. sequelize.js - O arquivo de conexão com banco de dados.



sequelize.js

Esse é nosso arquivo de conexão com o banco de dados. Aqui, utilizamos a ORM.

```
const Sequelize = require("sequelize");
const dotenv = require("dotenv");
dotenv.config();

const db = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASSWORD,
```

```
  {
    host: process.env.DB_HOST,
    dialect: "mysql",
  }
);

module.exports = db
```

Task.js

Esse é o nosso arquivo de modelo. Ele "espelha" os dados do banco de dados. Nesse caso, como estamos trabalhando com *code-first,* esse mesmo modelo é utilizado para construir a estrutura do banco de dados.

Em um modelo de dados, definimos as características do campo. Por exemplo: nome, tipo, se é chave, se é auto-incremento, se aceita null, e assim por diante, conforme as necessidades.

Em nosso caso, só temos 2 campos simples: Uma chave primária (id) e um campo texto de descrição (description).

```
const db = require("../sequelize");
const Sequelize = require("sequelize");

const Task = db.define("task", {
  id: {
    type: Sequelize.INTEGER,
```

```
    primaryKey: true,
    autoIncrement: true,
  },
  description: {
    type: Sequelize.STRING,
    allowNull: false,
  },
});

Task.sync();

module.exports = Task;
```

App.js

O app.js é nosso arquivo principal. É aqui o ponto de partida do servidor em execução, quando a URL é solicitada.

Repare que nós importamos o *express* para usarmos as rotas. Também importamos nosso arquivo de rotas.

Aqui também importamos o *cors* para utilizar em nosso projeto e permitir o acesso de qualquer URL. Também definimos a porta. Nesse caso, colocamos 8081, para não conflitar com a porta 3000 que será usada em localhost pela aplicação em React.

Caso você encontre algum erro, é possível que essa porta já esteja sendo utilizada. Nesse caso, tente trocar (para 8080, 8181, 8282, 2222, etc).

Observação: sempre que importamos um arquivo (módulo) que está na mesma pasta do arquivo que está chamando, nós usamos o formato './nome-do-modulo'. Para navegar um nível acima, utilize '../'.

```
const express = require('express');
const cors = require('cors');
const app = express();
const bodyParser = require('body-parser');
const routes = require('./routes');
const PORT = 8081;

app.use(cors());
app.use(bodyParser.json());
app.use(routes);

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
});
```

Pronto. A partir deste momento, você já pode executar o servidor com o comando:

```
node app.js
```

Você verá algo parecido com isso:

Rotas

Nessa seção, vamos criar todos os endpoints necessários para nossa aplicação:

- *GET /tasks* - Retorna todas as tarefas com paginação
- *GET /tasks/:id - Retorna uma tarefa por ID*
- *PUT /tasks/:id - Atualiza uma tarefa*
- *POST /tasks - Cria uma nova tarefa*
- *DELETE /tasks/:id - Exclui uma tarefa*

Na introdução do nosso arquivo de rotas, vamos inserir o código:

```
const express = require("express");
const router = express.Router();
const Task = require("./model/Task");
```

Em seguida, vamos escrever todas as nossas rotas e no final, vamos exportar:

```
module.exports = router;
```

Consultando tarefas (com paginação)

```
// Retorna tarefas (com paginação e ordenação)
router.get("/tasks", (req, res) => {
  const { page = 1, limit = 10 } = req.query;
  Task.findAll({
    offset: (page - 1) * limit,
    limit: +limit,
    order: [
      ['updatedAt', 'desc']
    ]
  }).then((tasks) => {
    res.json(tasks);
  });
});
```

Consultando tarefa por id

```
router.get("/tasks/:id", async (req, res) => {
  try {
    const task = await Task.findByPk(req.params.id);
    if (!task) {
      res.status(404).json({
```

```
        success: false,
        message: "Tarefa não encontrada",
      });
    } else {
      res.json({
        success: true,
        task: task,
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message,
    });
  }
});
```

Inserindo uma nova tarefa

```
router.post("/tasks", async (req, res) => {
  try {
    const task = new Task({
      description: req.body.description,
    });
```

```
    await task.save();
    res.status(201).json({
      success: true,
      message: "Tarefa criada com sucesso!",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message,
    });
  }
});
```

Atualizando uma tarefa

```
router.put("/tasks/:id", async (req, res) => {
  try {
    const task = await Task.findByPk(req.params.id);
    if (!task) {
      res.status(404).json({
        success: false,
        message: "Tarefa não encontrada",
      });
    } else {
```

```
      await task.update({
        description: req.body.description,
      });
      res.json({
        success: true,
        message: "Tarefa atualizada com sucesso!",
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message,
    });
  }
});
```

Excluindo uma tarefa

```
router.delete("/tasks/:id", async (req, res) => {
  try {
    const task = await Task.findByPk(req.params.id);
    if (!task) {
      res.status(404).json({
        success: false,
```

```
        message: "Tarefa não encontrada",
      });
    } else {
      await task.destroy();
      res.json({
        success: true,
        message: "Tarefa excluída com sucesso!",
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message,
    });
  }
});
```

Finalização do back-end

Pronto, seu back-end em ambiente de desenvolvimento está pronto para ser utilizado, acessar banco de dados para listar, inserir, editar e excluir informações (o famoso CRUD).

Nas próximas linhas você vai entender como fazer o front-end (em React) se conectar com nossa plataforma de back-end para exibir uma interface amigável.

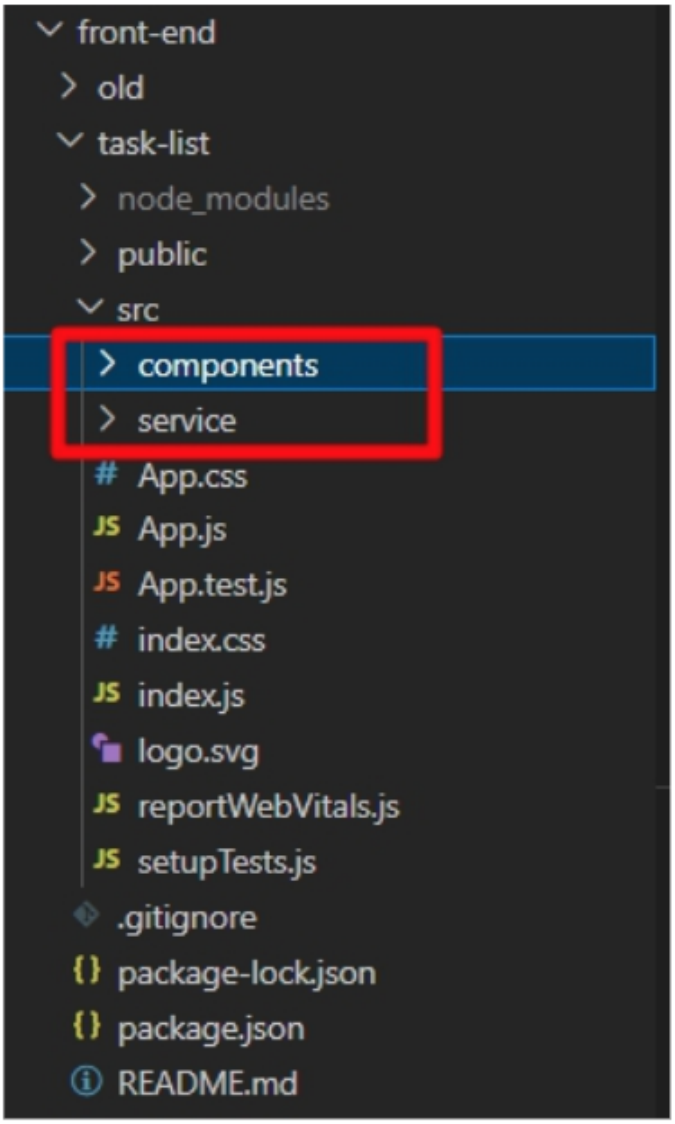
Front-end

Nesta seção do livro, vamos abordar a aplicação front-end.
Não é a intenção neste momento ensinar sintaxe ou detalhes do javascript. Mas, vamos abordar a estrutura de como construir uma aplicação front-end.

Caso você queira se aprofundar no desenvolvimento em React, nosso livro Desenvolvimento de aplicativos web com React está disponível na Amazon: https://amazon.com.br/react

Estrutura

A estrutura de nosso front-end terá uma pasta *service*, que conterá o serviço que acessa nosso back-end. Também terá uma pasta *components*, onde ficarão nossos componentes da aplicação.



Para nosso projeto, vamos utilizar dois pacotes:

- *Axios* - para fazer consultas à API.
  npm install axios

- *React Router Dom* - para controlar as rotas.

npm install react-router-dom

Service

Nós teremos um único serviço que se chamará *task-service.js.* Esse arquivo terá toda as chamadas à API, utilizando axios

Em seguida, vamos implementar todas as chamadas ao nosso back-end:

- *Listar tarefas*
- *Listar uma tarefa passando o id*
- *Atualizar as informações de uma tarefa*
- *Inserir uma nova tarefa*
- *Excluir uma tarefa.*

```
import axios from 'axios';

  // Define a URL da API
  apiUrl = 'https://localhost:8081/tasks';

const axiosInstance = axios.create({
  baseURL: apiUrl
});

class _TaskService {

  // Método para buscar tarefas paginadas
  async getTasks(page = 1, limit = 10) {
    try {
```

```
    // Faz uma chamada GET à API passando os parâmetros de página e limite
    const response = await axiosInstance.get(`?page=${page}&limit=${limit}`);

    // Retorna os dados da resposta
    return response.data;
  } catch (error) {
    // Trata o erro
    throw error;
  }
};

async getTask(id) {
  try {
    const response = await axiosInstance.get(`/${id}`);
    return response.data.task;
  } catch (error) {
    // Trata o erro
    throw error;
  }
}

async createTask(task) {
  try {
    const response = await axiosInstance .post('/ task);
    return response.data;
  } catch (error) {
```

```
      throw error;
    }
  };

  async updateTask(id, task) {
    try {
      const response = await axiosInstance .patch(`/${id}`, task);
      return response.data;
    } catch (error) {
      throw error;
    }
  };

  async deleteTask(id) {
    try {
      const response = await axiosInstance .delete(`/${id}`);
      return response.data;
    } catch (error) {
      throw error;
    }
  };

const TaskService = new _TaskService();
export default TaskService;
```

Componentes

Vamos criar 2 componentes:

- *TaskList* - Que exibe todas as tarefas, com paginação.
- *TaskForm* - Formulário utilizado para editar e/ou inserir uma nova tarefa.

App.js

Este é o componente principal de uma aplicação em React. Aqui, vamos configurar nossas rotas e as chamadas de cada componente.

Em nosso exemplo, temos somente um componente. Porém, caso existissem outros, você poderia criar novas chamadas de Route, passando o *path* e o componente desejado em *element*.

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import TaskList from "./components/TaskList";

const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<TaskList />} />
      </Routes>
    </BrowserRouter>
  );
};
```

export default App;

Listar tarefas

O fluxo da listagem das tarefas, funciona da seguinte forma: Visualização de todas as tarefas cadastradas, cada uma com as ações de editar e excluir e funcionalidades adicionais.

Quando o usuário clica no botão "Criar nova tarefa", é aberto um modal com um formulário para inserir uma nova tarefa. O usuário preenche os dados da tarefa e clica em "Salvar" para enviar os dados para o banco de dados.

Após a criação da nova tarefa, a listagem de tarefas é atualizada automaticamente, para mostrar a nova tarefa criada.

Quando o usuário clica em "Excluir" em uma tarefa existente, é apresentada uma notificação via Toastr para confirmar se deseja realmente excluir. Se o usuário confirma a exclusão, a tarefa é removida do banco de dados e a listagem de tarefas é atualizada novamente.

Além disso, é possível editar uma tarefa existente clicando no botão "Editar" e o fluxo é semelhante ao da criação, com o modal aberto, formulário preenchido e lista atualizada automaticamente após a edição.

Não se preocupe com a formatação e CSS agora. Já estamos implementando os componentes com so nomes das classes para facilitar, mas vamos abordar os estilos no decorrer do conteúdo.

Instalação de pacotes

Vamos instalar alguns pacotes auxiliares para nosso projeto:

- *React Toastify* - um pacote de notificações para exibir notificações amigáveis no front-end. Utilize o seguinte comando:

npm install react-toastify

- *React Modal* - Um modal para exibir uma tela rápida de edição, sem precisar sair da tela principal. Utilize o comando:

npm install react-modal

Configuração inicial

```
import React, { useState, useEffect } from "react";
import { ToastContainer, toast } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";
import Modal from "react-modal";

import TaskForm from "./TaskForm";
import TaskService from "./../service/task-service";
```

```
import "./../style/style.css";
```

Base do componente

```
const TaskList = () => {
  const [modalIsOpen, setModalIsOpen] = useState(false);
  const [tasks, setTasks] = useState([]);
  const [page, setPage] = useState(1);
  const [currentTask, setCurrentTask] = useState(null);
  const [loading, setLoading] = useState(false);
}
```

Atualização da lista

Vamos configurar um simples e importante **hook** do React chamado useEffect. Usamos ele para controlar "efeitos colaterais" dos comportamentos do componente..

Repare que ele é uma função, que recebe dois parâmetros:

1. Uma função de callback, que é executada
2. Array de dependências.

O array de dependências é utilizado quando queremos controlar em que momento a atualização vai acontecer. Nesse caso, passamos no array apenas um elemento, que é o page. Isso significa que toda vez que o número da página alterar, fará o load de novas tasks.

```
useEffect(() => {
    fetchTasks();
 }, [page]);
```

Em seguida, vamos adicionar todos os outros métodos necessários na tela de listagem:

- *fetchTasks* - para consultar as tarefas
- *loadMore* - para atualizar a paginação e carregar novamente os dados
- *handleCloseModal* - executado quando o modal for fechado. Importante para atualizarmos nossa lista após uma nova tarefa ser inserida ou editada.
- *handleNew* - abre o modal para inserir um novo item.
- *handleEdit* - abre o modal para editar um novo item.
- *handleDelete* - para excluir uma tarefa.

Repare que dentro do *fetchTasks*, verificamos se é a primeira página e zeramos a contagem. Caso contrário, é feito um *concat*, um *merge* de arrays, usando o comando do ecmascript chamado *spread*.

```
const fetchTasks = async () => {
    const data = await TaskService.getTasks(page, 4);

    if (page === 1) {
      setTasks(data);
      return;
    }
    setTasks([...tasks, ...data]);
```

```
};

const loadMore = () => {
  setPage(page + 1);
};

const handleCloseModal = () => {
  setModalIsOpen(false);
};

const handleNew = () => {
  setModalIsOpen(true);
  setCurrentTaskId(0);
};

const handleEdit = (id) => {
  setCurrentTaskId(id);
  setModalIsOpen(true);
};

const handleDelete = (id) => {
  TaskService.deleteTask(id)
    .then(() => {
      fetchTasks();
      toast.success("Tarefa excluída com sucesso!");
    })
    .catch((err) => {
```

```
      console.log(err);
      toast.error("Erro ao excluir tarefa");
    });
};

const handleSave = async () => {
  handleCloseModal();
  toast.success("Dados atualizados com sucesso!");
  setPage(1);
  await fetchTasks();
};
```

Para fins de estudo, o método *fetchTasks* reinicia a paginação sempre que um elemento for salvo. Porém, as regras de negócio do projeto poderiam ser diferentes, como manter a mesma lista exibida e apenas incluir e/ou alterar o elemento em questão.

E agora vamos inserir o conteúdo em JSX, que é uma linguagem de marcação que utiliza o HTML para ser renderizado dentro do Javascript. O “return” de um componente React deve sempre retornar um único elemento. Ou seja: Uma section, uma div ou um elemento vazio, assim:

```
return (<>
  ...
</>);
```

Ações da lista (editar / excluir)

Em nosso caso, temos um botão para inserir nova tarefa, uma tabela com as tarefas e sua descrição, e 2 botões de ação (editar e excluir).

Em seguida, teremos um botão de paginação, que carrega mais itens, infinitamente, até que não existam mais itens para carregar.

Observe que neste momento, chamamos um novo componente, TaskForm, que é o nosso componente de inclusão / edição de tarefa.

Caso queira ver funcionando seu projeto antes de implementar essa parte, remova o trecho do código que contém o **modal:** *<Modal>...</Modal>*.

```
 return (
<div className="main">
      <h1>Lista de Tarefas</h1>
      <div className="button-new-task-container">
        <button className="success" onClick={() => handleNew()}>Nova tarefa</button>
      </div>

      <table>
        <thead>
          <tr>
              <th>Id</th>
              <th>Descrição</th>
              <th>Ações</th>
          </tr>
        </thead>
```

```
  <tbody>
    {tasks.map((task) => (
        <tr key={task.id}>
          <td>{task.id}</td>
          <td>{task.description}</td>
          <td>
            <button onClick={() => handleEdit(task.id)}>Editar</button>
            <button onClick={() => handleDelete(task.id)}>Excluir</button>
          </td>
        </tr>
    ))}
  </tbody>
</table>
<div className="load-button-container">
  {loading ? (
    <p>Carregando...</p>
  ) : (
    <button onClick={loadMore}>Carregar Mais</button>
  )}
</div>

<Modal
  className="modal"
  ariaHideApp={false}
  isOpen={modalIsOpen}
```

```
        onRequestClose={handleCloseModal}
      >
        <h2>{currentTaskId ? "Editar tarefa" : "Nova Tarefa"}</h2>
        <TaskForm id={currentTaskId} onSave={handleSave} />
      </Modal>
      <ToastContainer />
    </div>
  );
```

Código completo

```
import React, { useState, useEffect } from "react";
import { ToastContainer, toast } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";
import Modal from "react-modal";

import TaskForm from "./TaskForm";
import TaskService from "./../service/task-service";

import "./../style/style.css";

const TaskList = () => {
  const [modalIsOpen, setModalIsOpen] = useState(false);
  const [tasks, setTasks] = useState([]);
  const [page, setPage] = useState(1);
  const [currentTaskId, setCurrentTaskId] = useState(null);
```

```
const [loading, setLoading] = useState(false);

useEffect(() => {
  fetchTasks();
}, [page]);

const fetchTasks = async () => {
  const data = await TaskService.getTasks(page, 4);

  if (page === 1) {
    setTasks(data);
    return;
  }

  setTasks([...tasks, ...data]);
};

const loadMore = () => {
  setPage(page + 1);
};

const handleCloseModal = () => {
  setModalIsOpen(false);
};

const handleNew = () => {
  setModalIsOpen(true);
  setCurrentTaskId(0);
```

```
};

const handleEdit = (id) => {
  setCurrentTaskId(id);
  setModalIsOpen(true);
};

const handleDelete = (id) => {
  TaskService.deleteTask(id)
    .then(() => {
      setTasks(tasks.filter((task) => task._id !== id));
      fetchTasks();
      toast.success("Tarefa excluída com sucesso!");
    })
    .catch((err) => {
      console.log(err);
      toast.error("Erro ao excluir tarefa");
    });
};

const handleSave = async () => {
  handleCloseModal();
  toast.success("Dados atualizados com sucesso!");
  setPage(1);
  await fetchTasks();
};
```

```
return (
  <div className="main">
    <h1>Lista de Tarefas</h1>
    <div className="button-new-task-container">
      <button onClick={() => handleNew()}>Nova tarefa</button>
    </div>

    <table>
      <thead>
        <tr>
            <th>Id</th>
            <th>Descrição</th>
            <th>Ações</th>
        </tr>
      </thead>
      <tbody>
        {tasks.map((task) => (
            <tr key={task.id}>
              <td>{task.id}</td>
              <td>{task.description}</td>
              <td>
                <button onClick={() => handleEdit(task.id)}>Editar</button>
                <button onClick={() => handleDelete(task.id)}>Excluir</button>
              </td>
            </tr>
```

```
          ))}
        </tbody>
      </table>
      <div className="load-button-container">
        {loading ? (
          <p>Carregando...</p>
        ) : (
          <button onClick={loadMore}>Carregar Mais</button>
        )}
      </div>

      <Modal
        className="modal"
        ariaHideApp={false}
        isOpen={modalIsOpen}
        onRequestClose={handleCloseModal}
      >
        <h2>{currentTaskId ? "Editar tarefa" : "Nova Tarefa"}</h2>
        <TaskForm id={currentTaskId} onSave={handleSave} />
      </Modal>
      <ToastContainer />
    </div>
  );
};
```

```
export default TaskList;
```

Inserir e Editar Tarefa

Vamos criar o componente TaskForm para nos permitir inserir um novo item ou editar um item já existente.
Observe que ele faz o controle baseado na propriedade id.

Ou seja: Se um id for passado, ele irá consultar uma tarefa pelo id. E então, fará a atualização.

Caso não tenha um id, ele fará a inclusão de um novo item.

```
import React, { useState, useEffect } from "react";
import TaskService from "../service/task-service";

const TaskForm = (props) => {
  const [task, setTask] = useState({});
  const { id, onSave } = props;

  useEffect(() => {
    if (!id) return;
    const load = async () => {
      const task = await TaskService.getTask(id);
      setTask(task);
    }
    load();
```

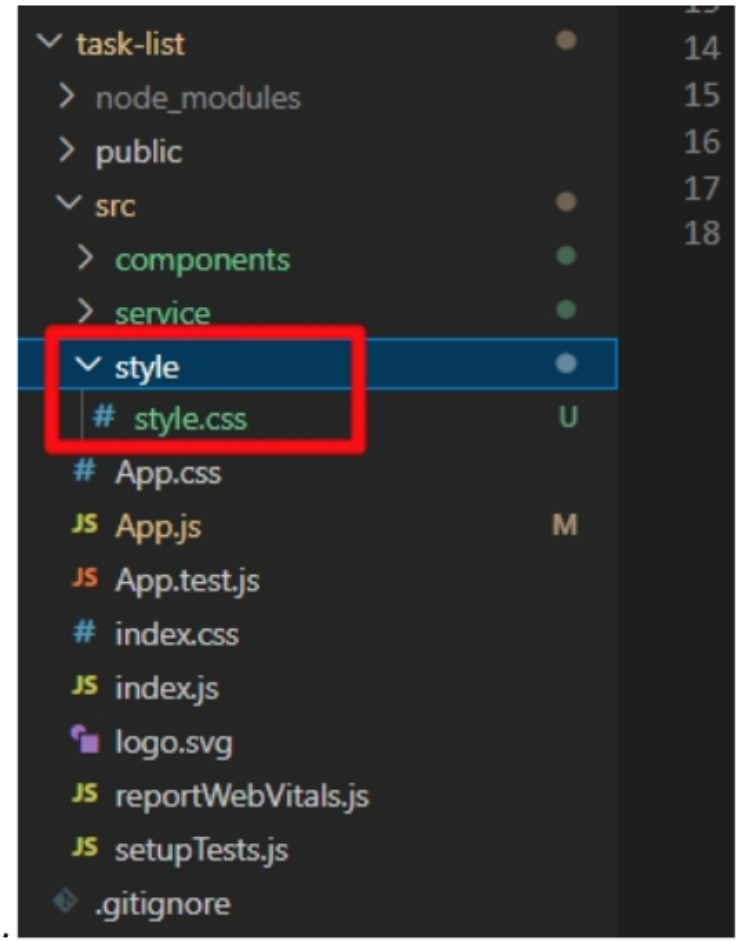
```
}, [id]);

const handleChange = (event) => {
  setTask({
    ...task,
    description: event.target.value,
  });
};

const handleSubmit = (event) => {
  event.preventDefault();
  if (props.id) {
    TaskService.updateTask(props.id, task)
      .then(() => {
        props.onSave();
      })
      .catch((error) => {
        console.log(error);
      });
  } else {
    TaskService.createTask(task)
      .then(() => {
        props.onSave();
      })
      .catch((error) => {
```

```
          console.log(error);
        });
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        name="description"
        placeholder="Task description"
        onChange={handleChange}
        value={task.description ? task.description : ""}
      />
      <button type="submit">Save</button>
    </form>
  );
};

export default TaskForm;
```

Estilos

Agora, vamos estilizar nosso projeto. Para deixá-lo mais bonito e agradável aos olhos do usuário. Vamos criar uma nova pasta dentro de src. E dentro dessa pasta nosso

arquivo css, como na imagem:




```
.modal {
  width: 50%;
  background: #ccc;
  margin: 0 auto;
  border-radius: 10px;
  padding: 20px;
```

```
  transform: translate(-50%, -50%);
  position: absolute;
  left: 50%;
  top: 50%;
}

input,
button {
  padding: 8px;
  border-radius: 8px;
  outline: none;
  border: none;

}

button {
  transition: all 300ms;
}

button:hover {
    background: #aaa;
}

button.success {
  background: #0a0;
  color: #fff;
}
```

```
button.danger {
  background: #f00;
}

input {
  border: 1px solid #ccc;
}

button {
  cursor: pointer;
}

table {
  width: 100%;
  border: 1px solid #ccc;
  border-radius: 8px;
}

table tr td {
  text-align: center;
}

.main {
  padding: 20px;
}
.load-button-container {
  margin-top: 20px;
```

```
  text-align: center;
}

.button-new-task-container {
  margin-bottom: 20px;
  text-align: right;
}
```

Nosso estilo tem abordagens simples: Cria um padrão para todos os componentes de forma geral (button, input, table). Definimos também algumas classes de container e alguns estilos de botão (success e danger, ainda que no projeto, só utilizamos o modelo *success.*

Esse tipo de abordagem é uma boa prática comum de CSS. Caso você queira saber mais sobre CSS, aprender design responsivo, flexbox, grid layout, manipular as variáveis e muito mais, fique à vontade para conhecer nossos livros sobre o assunto:

CSS: Design responsivo
CSS: Flexbox
CSS: Grid Layout
CSS: Cores, gradiente e geometria

Se tudo funcionou corretamente, você deve ver seu projeto finalizado, mais ou menos como seguem as telas:

Lista de Tarefas

Nova tarefa

Id	Descrição	Ações

Carregar Mais

Lista de Tarefas

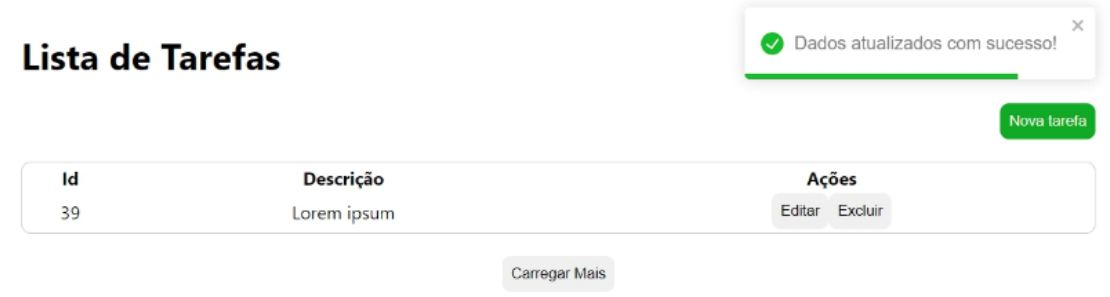
Nova tarefa

Id Descrição Ações

Carregar Mais

Nova Tarefa

Task description Save



Publicação

Back-end

Existem várias maneiras de publicar um projeto em Node Express, dependendo do seu objetivo e do ambiente em que você está trabalhando. Algumas das opções mais comuns incluem:

Hospedagem em um provedor de nuvem: Existem muitos provedores de nuvem, como o Heroku, o AWS ou o Azure, que oferecem ferramentas fáceis para publicar e escalar aplicativos Node.js.

Hospedagem em um servidor dedicado: Se você possui um servidor dedicado, pode instalar o Node.js e o Express nele e publicar seu aplicativo.

Hospedagem em um serviço de hospedagem compartilhada: Alguns provedores de hospedagem compartilhada, como o Bluehost, oferecem suporte para aplicativos Node.js.

Antes de publicar, certifique-se de que seu aplicativo está configurado corretamente e de que todas as dependências estão instaladas. Além disso, é recomendável testar seu aplicativo em um ambiente de desenvolvimento local antes de publicá-lo.

Para criar a build de um projeto Node.js, siga os seguintes passos:

- Instale todas as dependências do seu projeto com o comando npm install.

- Adicione uma tarefa "build" ao seu arquivo package.json, por exemplo:
  ```
  "scripts": {
      "build": "babel src -d dist"
    }
  ```

- *Execute o comando **npm run build** no terminal para criar o build.*

- *Verifique se a pasta "dist" foi criada, ela contém o seu código transpilado e pronto para ser executado.*

- *Quando for fazer o deploy, basta copiar os arquivos da pasta dist para o seu servidor.*

Front-end

A publicação do projeto front-end é mais simples por ser apenas conteúdo estático. Pode ser armazenado em qualquer lugar que permita arquivos estáticos. Ou seja, qualquer hospedagem compartilhada, ou o ECS da Amazon Web Services.

Para criar um build de um projeto React, siga os seguintes passos:

- Execute o comando **npm run build** no terminal para criar o build.

  Verifique se a pasta "build" foi criada, ela contém os arquivos transpilados e minificados do seu projeto.
  Basta copiar os arquivos da pasta build para o seu servidor.

Se você estiver usando uma ferramenta de automação de build, como o Jenkins ou o Travis CI, pode configurá-las para criar o build automaticamente toda vez que o código é enviado para o repositório.

Glossário

Este glossário é um pequeno dicionário para te ajudar a entender melhor alguns conceitos que discutimos e utilizamos ao longo do livro.

Banco de dados

Um banco de dados é um sistema de armazenamento de dados que permite o armazenamento, recuperação e gerenciamento de informações. Ele é usado para armazenar e gerenciar grandes quantidades de dados de forma organizada e estruturada.

CORS (Cross-Origin Resource Sharing)

Este é um mecanismo de segurança que permite que um recurso de um domínio seja acessado por outro domínio. Ele é usado para restringir o acesso a recursos que não estão no mesmo domínio da aplicação que os requisita.

CSS

CSS é a sigla para Cascading Style Sheets, é uma linguagem de folhas de estilo utilizadas

para separar a formatação e a estrutura dos documentos HTML. Ele permite que você aplique estilos consistentes e reutilizáveis a vários elementos HTML em uma página web. Isso inclui coisas como fontes, cores, layouts e espaçamento. Utilizando CSS, é possível separar a formatação do conteúdo e melhorar a manutenção e escalabilidade do código.

JSX

JSX é uma extensão da sintaxe JavaScript que permite a incorporação de elementos de HTML em código JavaScript. Ele é usado principalmente em conjunto com o ReactJS, uma biblioteca JavaScript para criar interfaces de usuário reativas e escaláveis. JSX permite a criação de componentes de interface de usuário que podem ser facilmente compartilhados e reutilizados em diferentes partes da aplicação.

MySQL

MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto. Ele é amplamente utilizado para aplicações web e é conhecido por sua escalabilidade, desempenho e estabilidade. Ele suporta uma variedade de linguagens de programação, incluindo PHP, Java, C# e Python.

Node JS

Node JS é uma plataforma de desenvolvimento JavaScript que permite a criação de aplicativos de servidor usando JavaScript. Ele fornece uma API para acessar recursos do sistema, como arquivos e redes, e é amplamente utilizado para criar aplicações web escaláveis e de alta performance.

ORM (Object-Relational Mapping)

É uma técnica que permite que o código de aplicação acesse banco de dados de forma orientada a objetos, em vez de utilizar linguagem SQL. Isso simplifica a escrita de código e ajuda a manter a consistência e integridade dos dados.

React JS

React JS é uma biblioteca JavaScript desenvolvida pelo Facebook para criar interfaces de usuário reativas e escaláveis. Ele permite a criação de componentes de interface de usuário independentes que podem ser facilmente compartilhados e reutilizados em diferentes partes da aplicação.

SGDB

SGDB é a sigla para Sistema Gerenciador de Banco de Dados, é um software que permite a criação, manipulação, gerenciamento e acesso aos dados armazenados em um banco de dados. Ele é responsável por garantir a consistência, integridade e segurança dos dados, bem como fornecer mecanismos para recuperação de desastres. Um SGDB também fornece uma interface para o usuário ou aplicação para acessar e manipular os dados, geralmente através de linguagens de consulta específicas.

Sequelize

Sequelize é um ORM para Node.js que suporta vários bancos de dados, como MySQL, PostgreSQL e SQLite. Ele permite que você defina modelos de tabela no seu código e trabalhe com esses modelos usando métodos de instância, sem precisar escrever consultas SQL.

Tratamento de erros

Try / Catch é uma estrutura de controle de exceções utilizada em muitas linguagens de programação para lidar com erros ou exceções que podem ocorrer em um código. O bloco try é usado para executar o código que pode gerar uma exceção, e o bloco catch é usado para lidar com a exceção quando ela é lançada.

Verbos HTTP

Os verbos HTTP (ou métodos) são usados para especificar a ação que deve ser realizada em um recurso. Os verbos mais comuns são GET (obter), POST (criar), PUT (atualizar) e DELETE (excluir).