PROJETO – SISTEMA WEB TODOLIST

Construindo um Sistema Todolist Full Stack

Javascript • React • Fastify • MySQL

Um guia passo a passo, do banco de dados à interface

Professor: Marcos de Melo

Nível: Iniciante / Intermediário

Pré-requisitos: lógica de programação básica e JavaScript

Sumário

Capítulo 1 — Visão Geral do Projeto

Bem-vindo(a)! Nesta apostila vamos construir, juntos e do zero, um sistema completo de lista de tarefas (Todolist). Não é um projeto de brinquedo: ele tem banco de dados de verdade, uma API HTTP e uma interface React reativa — exatamente a mesma estrutura que você encontraria em um sistema profissional em produção.

A ideia desta apostila não é apenas "colar código", mas entender o porquê de cada decisão: porque separamos backend e frontend, porque usamos um pool de conexões, porque o estado do React fica organizado do jeito que fica. Ao final, você deve ser capaz de recriar o projeto sozinho(a) e adaptá-lo para outras ideias.

1.1 O que vamos construir

Uma aplicação onde o usuário pode:

- Ver a lista de tarefas pendentes e concluídas;
- Criar uma nova tarefa, com nome e uma data limite opcional;
- Editar uma tarefa já existente (nome e data);
- Marcar uma tarefa como concluída ou reabri-la;
- Excluir tarefas;
- Buscar tarefas pelo nome, em tempo real;
- Ver, visualmente, quais tarefas estão "em atraso" (data limite vencida e ainda pendente).

1.2 Arquitetura da aplicação

O projeto é dividido em três camadas que conversam entre si por meio de requisições HTTP. Pense nisso como três funcionários numa empresa, cada um com uma responsabilidade clara:

```
[ Navegador ]              [ Servidor Node.js ]            [ Banco de Dados ]
React (Vite)   --HTTP-->   Fastify (API REST)   --SQL-->    MySQL
porta 5173                 porta 3333                       porta 3306

Interface visual           Regras de negócio               Armazenamento
(o que o usuário vê        (validações, cálculo            permanente dos
e clica)                   de atraso, acesso ao banco)     dados das tarefas
```

> **Por que separar em camadas?**
> Cada camada pode evoluir sozinha. Você poderia trocar o React por outro framework de tela sem tocar no backend, ou trocar o MySQL por PostgreSQL sem tocar no frontend — desde que o "contrato" entre elas (a API) continue o mesmo. Essa separação também facilita testar e reaproveitar cada parte.

1.3 Tecnologias utilizadas e por que escolhemos cada uma

React

Biblioteca para construir interfaces baseadas em componentes reutilizáveis. Cada pedaço da tela (o cabeçalho, o campo de busca, um item de tarefa) vira um componente independente, o que deixa o código organizado e fácil de manter.

Vite

Ferramenta que "empacota" o código do frontend para rodar no navegador. Escolhemos o Vite pela inicialização quase instantânea e pela configuração mínima comparado a alternativas mais antigas.

Tailwind CSS

Em vez de escrever arquivos CSS separados, aplicamos classes utilitárias diretamente no HTML/JSX (como bg-panel, rounded-xl, text-gray-400). Isso acelera muito a criação de interfaces consistentes.

Fastify

Framework para criar o servidor e a API em Node.js. É semelhante ao Express, mas com melhor desempenho e uma forma mais organizada de registrar plugins e rotas — ideal para quem está aprendendo boas práticas de backend.

MySQL

Sistema de banco de dados relacional, onde guardamos as tarefas em formato de tabela (linhas e colunas), com tipos de dados definidos e capacidade de fazer buscas complexas com SQL.

mysql2

Biblioteca Node.js que permite ao Fastify enviar comandos SQL para o MySQL e receber os resultados, com suporte a Promises (async/await) e a um "pool" de conexões reaproveitáveis.

Capítulo 2 — Preparando o Ambiente

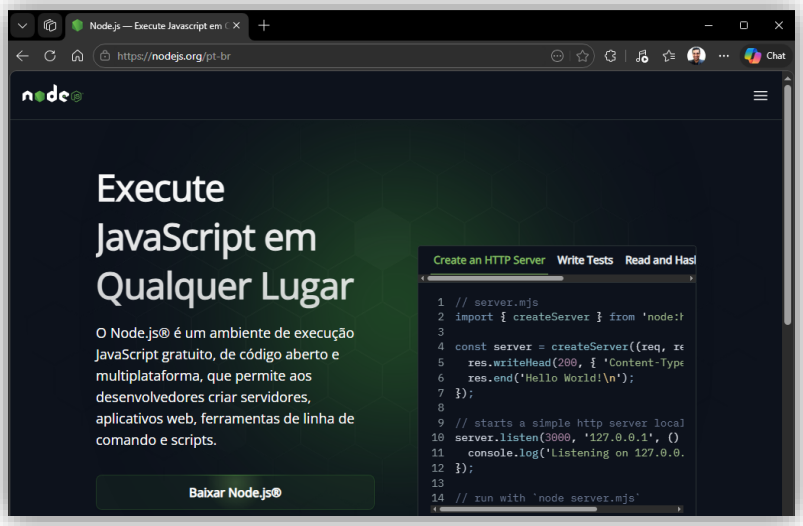
Antes de escrever qualquer linha de código do projeto, precisamos instalar as ferramentas que vamos usar. Siga estes passos com calma — um ambiente bem configurado evita 90% dos problemas de "na minha máquina funciona".

2.1 Instalando o Node.js

O Node.js é o que permite executar JavaScript fora do navegador — é ele quem roda tanto o servidor Fastify quanto as ferramentas do frontend (Vite, npm).

1. Baixe o instalador

Acesse nodejs.org e baixe a versão LTS (Long Term Support), recomendada para a maioria dos projetos.



2. Instale e verifique

Após instalar, abra um terminal e confirme a versão instalada:

```
node --version
npm --version
```

Se os dois comandos retornarem números de versão (ex.: v20.11.0), está tudo certo.

2.2 Instalando e configurando o MySQL

O MySQL é o banco de dados onde as tarefas serão armazenadas de forma permanente.

1. Instale o MySQL Server

Baixe em dev.mysql.com/downloads/mysql (Windows/Mac)

ou instale via gerenciador de pacotes no Linux, por exemplo:

```
sudo apt update
sudo apt install mysql-server
```

2. Defina uma senha para o usuário root

Durante a instalação (ou com mysql_secure_installation) você define a senha do usuário administrador. Guarde-a — vamos usá-la no arquivo .env do backend.

3. Teste o acesso

Confirme que consegue entrar no console do MySQL:

```
mysql -u root -p
```

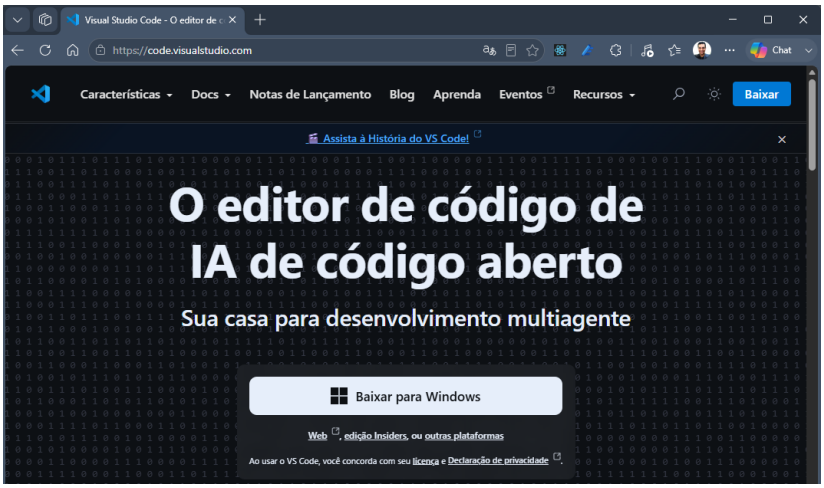
2.3 Estrutura de pastas do projeto

O projeto é organizado em duas pastas principais, cada uma com seu próprio package.json (ou seja, suas próprias dependências e comandos):

Escola um diretório no Sistema Operacional e crie a pasta ***todolist,*** essa será a pasta raiz de todo o projeto.

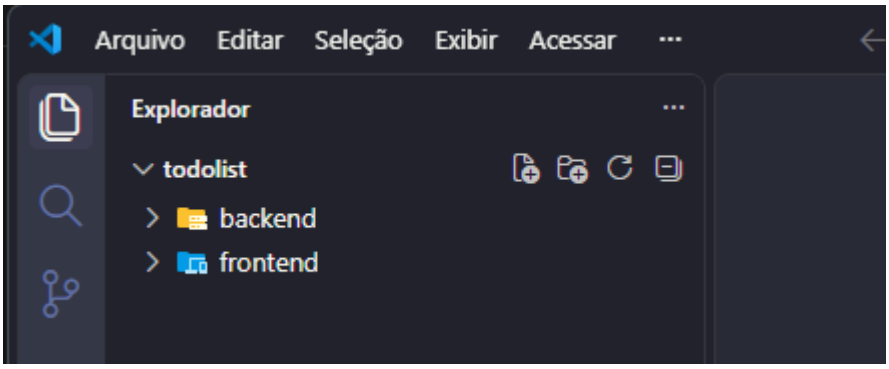
Vamos usar o programa **Visual Studio Code** ou simplesmente **VS Code.**

Nota: se você ainda não instalou o vscode instale-o baixando o site https://code.visualstudio.com/



Execute o programa VSCODE e abra a pasta ***todolist*** dentro do programa no ***Menu ➔ Arquivo ➔ Abrir pasta***.

Já dentro da pasta ***todolist*** vamos criar mais duas pastas, são elas; ***backend*** e ***frontend.***



Dentro delas a estrutura de pastas mostrada a seguir deverá ser criada. Mas, calma, não precisa criar a estrutura de pastas e arquivos ainda, isso será feito no decorrer do desenvolvimento do projeto.

```
todolist/
├── backend/              API em Fastify + MySQL
│   ├── src/
│   │   ├── db.js         Conexão com o banco
│   │   ├── server.js     Ponto de entrada do servidor
│   │   └── routes/
│   │       └── tasks.js  Rotas da API (CRUD de tarefas)
│   ├── schema.sql        Script de criação do banco
│   └── .env       Modelo de variáveis de ambiente
│
└── frontend/              Interface em React
```

```
├── src/
│   ├── main.jsx        Ponto de entrada do React
│   ├── App.jsx         Componente principal
│   ├── api.js          Comunicação com o backend
│   ├── index.css       Estilos globais + Tailwind
│   └── components/
│       ├── Header.jsx
│       ├── SearchBar.jsx
│       ├── Stats.jsx
│       ├── TaskList.jsx
│       ├── TaskItem.jsx
│       └── TaskModal.jsx
├── index.html
└── tailwind.config.js
```

Dica de organização

Sempre que um projeto tiver "duas pontas" (um servidor e uma interface), vale a pena mantê-las em pastas separadas, cada uma instalável e executável de forma independente. Isso facilita, inclusive, hospedá-las em servidores diferentes no futuro.

Capítulo 3 — Modelando o Banco de Dados

Todo sistema começa com uma pergunta simples: "quais informações eu preciso guardar?". Vamos responder isso desenhando a tabela tasks (tarefas).

3.1 Pensando nas colunas da tabela

Para cada tarefa, o sistema precisa saber:

- `id` — um identificador único, gerado automaticamente pelo banco;
- `title` — o nome/descrição da tarefa, obrigatório;
- `completed` — se a tarefa já foi concluída (verdadeiro ou falso);
- `due_date` — a data limite para conclusão, que é opcional;
- `created_at` e `updated_at` — quando a tarefa foi criada e alterada pela última vez, úteis para ordenar e auditar.

3.2 O script schema.sql explicado

Este é o script que cria o banco de dados e a tabela. Vamos linha por linha:

```sql
-- Execute este script no seu servidor MySQL antes de iniciar o backend.

CREATE DATABASE IF NOT EXISTS todolist
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

USE todolist;

CREATE TABLE IF NOT EXISTS tasks (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  completed BOOLEAN NOT NULL DEFAULT FALSE,
  due_date DATE NULL,
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

- `CREATE DATABASE IF NOT EXISTS` cria o banco chamado todolist apenas se ele ainda não existir — evita erro ao rodar o script mais de uma vez;
- `utf8mb4` é o conjunto de caracteres que suporta acentos, emojis e caracteres especiais corretamente;
- `AUTO_INCREMENT PRIMARY KEY` faz o MySQL gerar um novo número de id sempre que uma tarefa é inserida, sem precisarmos controlar isso manualmente;
- `NOT NULL` em title significa que o banco recusa gravar uma tarefa sem nome;
- `DEFAULT FALSE` faz com que toda tarefa nova comece como não concluída;
- `due_date DATE NULL` permite que a data fique em branco — nem toda tarefa precisa de prazo;
- os dois últimos campos são preenchidos automaticamente pelo próprio MySQL, sem que o backend precise informá-los.

Capítulo 4 — Construindo o Backend com Fastify

4.1 O que é uma API REST

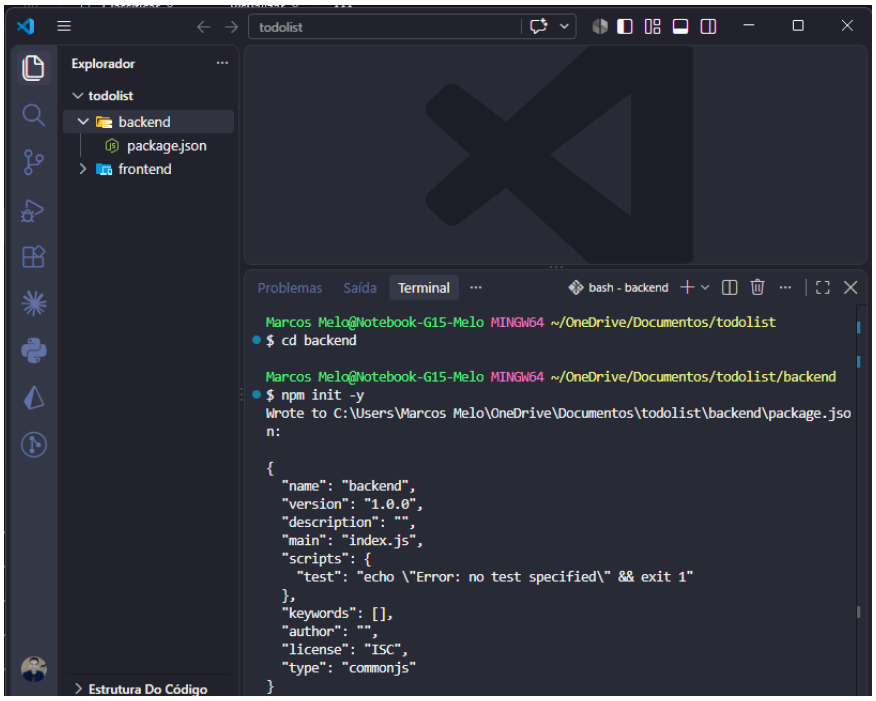
API significa "Interface de Programação de Aplicações": um conjunto de portas de entrada que um programa oferece para que outros programas conversem com ele. REST é um estilo de organizar essas portas usando o protocolo HTTP, onde cada "rota" representa um recurso (no nosso caso, tarefas) e o método HTTP indica a ação:

- `GET` — buscar/listar dados;
- `POST` — criar um novo registro;
- `PATCH` — atualizar parcialmente um registro existente;
- `DELETE` — remover um registro.

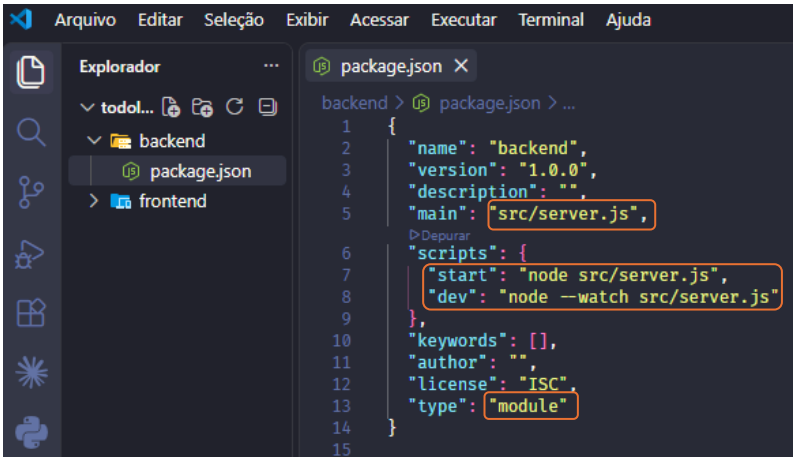
É exatamente esse padrão que vamos seguir nas rotas de tarefas.

4.2 Criando a estrutura de pastas e arquivos do Backend

Acesse o ***terminal de comandos*** no VSCODE digitando o atalho no teclado "**CTRL + J**" e em seguida digite no terminal "**cd backend**" para entrar pelo terminal na pasta backend. Dentro da pasta digite "***npm init -y*"** para criar o arquivo "***package.json*"** , este arquivo irá gerenciar os pacotes de bibliotecas JavaScript do projeto.



No painel lateral esquerdo, ***explorador*** clique no arquivo ***package.json*** para mostrar no lado direito o seu código. Faça as alterações conforme a imagem a seguir:



Agora sim, crie a estrutura de pastas e arquivos dentro da pasta backend conforme indicado a seguir;

```
todolist/
├── backend/              API em Fastify + MySQL
│   ├── src/
│   │   ├── db.js         Conexão com o banco
│   │   ├── server.js     Ponto de entrada do servidor
│   │   └── routes/
│   │       └── tasks.js  Rotas da API (CRUD de tarefas)
│   │
│   └── .env       Modelo de variáveis de ambiente
```

4.3 Módulos no Node.JS

No terminal devemos instalar alguns pacotes módulos de bibliotecas JavaScript que são necessários para o sistema.

- Podemos criar nossos próprios módulos (nossos arquivos) e importá-los
- O Node.js vem módulos pré-instalados (patch, fs, http etc.)

4.4 NPM – Node Package Manager

Podemos instalar módulos de terceiros utilizando o NPM
Esses módulos são armazenados em uma pasta chamada "**node_modules**"
Um arquivo chamado "**package.json**" lista todos os módulos que instalamos pelo NPM
Para instalar uma biblioteca digitamos no terminal o comando ***npm install nome_do_modulo***
Instale os módulos das bibliotecas listadas a seguir;

```
npm install fastify // instala o pacote do fastify
npm install mysql2 // instala o pacote do mysql server para fazer a comunicação com o banco de dados.
npm install @fastify/cors // instala o pacote do cors
npm install dotenv -D // instala o pacote do dotenv para acessar variáveis de ambiente.
```

4.5 Arquivo .env

O arquivo **.env** é usado para armazenar **variáveis de ambiente** de um projeto — ou seja, configurações e dados sensíveis que não devem ficar diretamente escritos no código-fonte.

Por que usar

- Segurança — chaves de API, senhas de banco de dados e tokens secretos ficam fora do código. Isso evita que sejam expostos caso o código seja publicado em um repositório público (GitHub, por exemplo).
- Flexibilidade entre ambientes — você pode ter configurações diferentes para desenvolvimento, teste e produção sem alterar o código, apenas trocando os valores no .env de cada ambiente.
- Organização — centraliza todas as configurações do projeto em um único lugar, facilitando manutenção.

Digite no arquivo .env o código a seguir;

```
PORT=3333

DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=senha_do_usuario_root
DB_NAME=todolist

# Origem permitida para o CORS (endereço do frontend)
CORS_ORIGIN=http://localhost:5173
```

4.6 Conectando ao banco (db.js) — pool de conexões

Agora vamos codificar o arquivo db.js, esse arquivo é responsável pela comunicação com o servidor de banco de dados e com a conexão ao nosso banco de dados chamado **todolist**.

Arquivo db.js

```
import mysql from 'mysql2/promise';
import 'dotenv/config';

const {DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME} = process.env;

export const pool = mysql.createPool({
  host: DB_HOST || 'localhost',
  port: Number(DB_PORT) || 3306,
  user: DB_USER || 'root',
  password: DB_PASSWORD || '',
  database: DB_NAME || 'todolist',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
  dateStrings: true,
});
```

> **Conceito importante: por que um "pool" e não uma conexão só?**
>
> Abrir e fechar uma conexão com o banco a cada requisição é lento. Um pool mantém várias conexões já abertas (até connectionLimit, aqui 10) e as empresta para quem precisar, devolvendo-as ao final. Isso deixa a API muito mais rápida sob uso simultâneo. Repare também em dateStrings: true — ele faz o mysql2 devolver datas como texto simples ("2026-09-20"), que é o formato mais fácil de manipular no JavaScript e enviar para o frontend.

Todas as configurações (host, senha, nome do banco) vêm de variáveis de ambiente, lidas pelo pacote dotenv a partir do arquivo .env. Isso evita deixar senhas escritas diretamente no código-fonte.

4.7 Criando as rotas (tasks.js)

Este arquivo concentra toda a "regra de negócio" das tarefas. O código está completo, porém mostrado em partes de cima para baixo, vamos compreender rota por rota.

Arquivo tasks.js

```
import { pool } from '../db.js';

const SELECT_FIELDS = 'id, title, completed, due_date, created_at';

export default async function tasksRoutes(app) {
```

As três linhas de código acima é o topo do arquivo

- A primeira linha é importado o pool de conexão com o banco de dados do arquivo db.js
- Segunda linha é definida uma constante com um trecho de uma consulta SQL que será usado mais adiante.
- Terceira linha define a função que tem que ser assíncrona e exportada chamada ***tasksRoutes(app)****, nesta linha no final é definida a chave de abertura da função que será fechada só lá no final.*

GET /api/tasks — listar e buscar

```
app.get('/api/tasks', async (request, reply) => {
    const { search = '', status = 'all' } = request.query;

    let sql = `SELECT ${SELECT_FIELDS} FROM tasks WHERE title LIKE ?`;
    const params = [`%${search}%`];

    if (status === 'pending') {
      sql += ' AND completed = 0';
    } else if (status === 'completed') {
      sql += ' AND completed = 1';
    }

    sql += ' ORDER BY (due_date IS NULL), due_date ASC, created_at DESC';

    const [rows] = await pool.query(sql, params);

    const [[totals]] = await pool.query(
      'SELECT COUNT(*) AS total, SUM(completed = 1) AS completedCount FROM
tasks'
    );

    return reply.send({
      tasks: rows,
      total: totals.total,
      completed: Number(totals.completedCount) || 0,
    });
  });
```

- `request.query` captura o que vem depois do "?" na URL, por exemplo /api/tasks?search=estudar;
- `LIKE ?` com `%termo%` faz uma busca "contém", encontrando a palavra em qualquer parte do título;
- usamos `?` como marcador de parâmetro em vez de concatenar a string diretamente — isso é chamado de "consulta parametrizada" e é a principal defesa contra ataques de SQL Injection;

- a cláusula `ORDER BY` coloca primeiro as tarefas que têm data (as sem data vão para o fim) e, entre as que têm data, a mais próxima do vencimento aparece primeiro;
- fazemos uma segunda consulta apenas para contar o total e quantas estão concluídas — esses números alimentam o painel "Total tasks / Completed" da interface.

POST /api/tasks — criar tarefa

```
// POST /api/tasks  { title, dueDate? }
  app.post('/api/tasks', async (request, reply) => {
    const { title, dueDate } = request.body ?? {};

    if (!title || !title.trim()) {
      return reply.status(400).send({ message: 'O nome da tarefa é
obrigatório.' });
    }

    const [result] = await pool.query(
      'INSERT INTO tasks (title, completed, due_date) VALUES (?, false, ?)',
      [title.trim(), dueDate || null]
    );

    const [[task]] = await pool.query(
      `SELECT ${SELECT_FIELDS} FROM tasks WHERE id = ?`,
      [result.insertId]
    );

    return reply.status(201).send(task);
  });
```

- a validação `if (!title || !title.trim())` recusa títulos vazios ou só com espaços, retornando o código HTTP 400 (Bad Request);
- `result.insertId` é o id gerado automaticamente pelo MySQL para a nova linha;
- depois de inserir, fazemos um novo SELECT para devolver a tarefa completa (com created_at já preenchido) ao frontend;
- o código de resposta `201 (Created)` é a convenção HTTP para "algo novo foi criado com sucesso".

PATCH /api/tasks/:id — atualizar (concluir ou editar)

```
// PATCH /api/tasks/:id  { completed?, title?, dueDate? }
  app.patch('/api/tasks/:id', async (request, reply) => {
    const { id } = request.params;
    const { completed, title, dueDate } = request.body ?? {};

    const [[existing]] = await pool.query('SELECT id FROM tasks WHERE id = ?',
[id]);
    if (!existing) {
      return reply.status(404).send({ message: 'Tarefa não encontrada.' });
    }

    const fields = [];
    const values = [];

    if (typeof completed === 'boolean') {
      fields.push('completed = ?');
      values.push(completed);
    }
```

```
    if (typeof title === 'string' && title.trim()) {
      fields.push('title = ?');
      values.push(title.trim());
    }
    // dueDate pode ser uma string 'YYYY-MM-DD', null (para remover) ou
undefined (não alterar)
    if (dueDate !== undefined) {
      fields.push('due_date = ?');
      values.push(dueDate || null);
    }

    if (fields.length === 0) {
      return reply.status(400).send({ message: 'Nada para atualizar.' });
    }

    values.push(id);
    await pool.query(`UPDATE tasks SET ${fields.join(', ')} WHERE id = ?`,
values);

    const [[task]] = await pool.query(
      `SELECT ${SELECT_FIELDS} FROM tasks WHERE id = ?`,
      [id]
    );

    return reply.send(task);
  });
```

Esta é a rota mais interessante porque ela é "genérica": serve tanto para marcar/desmarcar uma tarefa como concluída (quando o frontend manda só completed) quanto para editar o texto e a data (quando manda title e dueDate). O truque é montar o SQL de forma dinâmica:

- começamos com listas vazias fields e values;
- para cada campo que veio no corpo da requisição, adicionamos um pedaço "campo = ?" em fields e o valor correspondente em values;
- no final, juntamos tudo com `fields.join(', ')`, gerando algo como `completed = ?, title = ?`;
- isso evita ter uma rota para cada tipo de atualização — uma única rota PATCH atende os dois casos de uso da interface: o checkbox e o modal de edição.

> **Cuidado com dueDate ausente vs. nulo**
> Repare no if (dueDate !== undefined): se o campo nem foi enviado, não mexemos na data (undefined = "não me diga nada sobre isso"). Se foi enviado como null ou string vazia, apagamos a data. Essa distinção é o que permite ao checkbox de "concluir" atualizar só o campo completed, sem acidentalmente apagar a data limite da tarefa.

DELETE /api/tasks/:id — remover tarefa

```
// DELETE /api/tasks/:id
  app.delete('/api/tasks/:id', async (request, reply) => {
    const { id } = request.params;
    const [result] = await pool.query('DELETE FROM tasks WHERE id = ?', [id]);

    if (result.affectedRows === 0) {
      return reply.status(404).send({ message: 'Tarefa não encontrada.' });
```

```
    }

    return reply.status(204).send();
  });
}
```

O código 204 (No Content) informa que a operação deu certo, mas não há corpo de resposta para devolver — padrão para exclusões bem-sucedidas.

4.8 Arquivo server.js

Clique no arquivo ***server.js*** localizado dentro da pasta **src/** , digite o código a seguir;

```
import Fastify from 'fastify';
import cors from '@fastify/cors';
import 'dotenv/config';
import tasksRoutes from './routes/tasks.js';

const app = Fastify({ logger: true });

await app.register(cors, {
  origin: process.env.CORS_ORIGIN || 'http://localhost:5173',
});

app.get('/health', async () => ({ status: 'ok' }));

await app.register(tasksRoutes);

const port = Number(process.env.PORT) || 3333;

try {
  await app.listen({ port, host: '0.0.0.0' });
  console.log(`Servidor rodando em http://localhost:${port}`);
} catch (err) {
  app.log.error(err);
  process.exit(1);
}
```

Vamos entender cada trecho:

- `Fastify({ logger: true })` cria a aplicação e ativa logs automáticos de cada requisição — muito útil para depuração;
- `@fastify/cors` é um plugin que libera o navegador (rodando em outra porta, a 5173) a fazer requisições para esta API. Sem isso, o navegador bloquearia por segurança (política de mesma origem);
- a rota `/health` é uma convenção comum: um endpoint simples para verificar se o servidor está de pé, útil em monitoramento;
- `app.register(tasksRoutes)` "pluga" todas as rotas de tarefas (que veremos a seguir) na aplicação principal;
- antes de abrir a porta, chamamos `ensureSchema()` para garantir que a tabela do banco já existe;
- todo o bloco fica dentro de um `try/catch` para que, se algo falhar (por exemplo, banco fora do ar), o erro apareça no log em vez de travar silenciosamente.

4.9 Testando a API sozinha

Antes de conectar o React, é uma ótima prática confirmar que a API funciona sozinha, usando o terminal:

```
# Listar tarefas
curl http://localhost:3333/api/tasks

# Criar uma tarefa com data limite
curl -X POST http://localhost:3333/api/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Estudar Fastify", "dueDate": "2026-09-20"}'

# Marcar como concluída (troque 1 pelo id retornado)
curl -X PATCH http://localhost:3333/api/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"completed": true}'

# Excluir
curl -X DELETE http://localhost:3333/api/tasks/1
```

> **Por que testar assim?**
>
> Se algo der errado depois, esse hábito ajuda a descobrir rapidamente se o problema está no backend (a API responde errado) ou no frontend (a API está certa, mas a tela não está lendo a resposta corretamente).

Capítulo 5 — Construindo o Frontend com React

5.1 Vite + Tailwind: por que essa combinação

O Vite compila nosso código JSX (a mistura de JavaScript com HTML que o React usa) para algo que o navegador entende, e recarrega a tela automaticamente a cada alteração salva. O Tailwind, por sua vez, nos dá classes prontas (flex, rounded-xl, text-gray-400) para estilizar sem escrever CSS manualmente — o que é especialmente útil quando estamos reproduzindo um layout já definido, como o nosso mockup.

5.2 Estrutura de componentes

Pensamos a interface como uma árvore de componentes, cada um responsável por uma parte da tela:

```
App (estado geral: lista de tarefas, busca, modal)
├── Header            (logo "to do." e subtítulo)
├── SearchBar          (campo de busca)
├── Stats              (contadores total/concluídas)
├── TaskList           (decide entre lista ou estado vazio)
│     └── TaskItem     (cada linha de tarefa, repetida)
└── TaskModal          (formulário de criar/editar, aparece por cima)
```

> **Componentes "burros" vs. componente "inteligente"**
>
> Repare que Header, SearchBar, Stats, TaskItem e TaskList não sabem nada sobre como buscar dados do servidor — eles só recebem informações prontas (props) e avisam o componente pai quando algo acontece (por exemplo, "o usuário clicou em excluir"). Quem realmente conversa com a API e guarda o estado é o App. Essa separação é uma prática comum em React: mantém os componentes de tela simples e fáceis de reaproveitar.

Header.jsx

```
export default function Header() {
  return (
    <header className="flex flex-col items-center pt-14 pb-10 px-4 text-center">
      <div className="flex items-center gap-3 mb-4">
        <svg width="40" height="40" viewBox="0 0 64 64" fill="none">
          <defs>
            <linearGradient id="logoGradient" x1="0%" y1="0%" x2="100%" y2="100%">
              <stop offset="0%" stopColor="#22d3c8" />
              <stop offset="50%" stopColor="#6d7dee" />
              <stop offset="100%" stopColor="#8b5cf6" />
            </linearGradient>
          </defs>
          <circle cx="32" cy="32" r="24" stroke="url(#logoGradient)" strokeWidth="9" fill="none"
/>
          <circle cx="45" cy="19" r="6" fill="#0c0c10" />
        </svg>
        <h1 className="text-4xl font-bold tracking-tight">
          <span className="text-white">para</span>
          <span className="bg-gradient-to-r from-[#6d7dee] to-[#8b5cf6] bg-clip-text text-
transparent">
            fazer.
          </span>
        </h1>
      </div>
      <p className="text-gray-400 text-base">
```

```
        Organize e complete suas tarefas pendentes com facilidade.
      </p>
    </header>
  );
}
```

SearchBar.jsx

```
export default function SearchBar({ value, onChange }) {
  return (
    <div className="flex items-center gap-3">
      <div className="flex-1 flex items-center gap-3 bg-field border border-border rounded-xl
px-4 py-3">
        <svg width="18" height="18" viewBox="0 0 24 24" fill="none" className="text-gray-500
shrink-0">
          <circle cx="11" cy="11" r="7" stroke="currentColor" strokeWidth="2" />
          <line x1="16.5" y1="16.5" x2="21" y2="21" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
        </svg>
        <input
          value={value}
          onChange={(e) => onChange(e.target.value)}
          placeholder="Pesquisar tarefas"
          className="bg-transparent outline-none w-full text-sm text-gray-200 placeholder:text-
gray-500"
        />
      </div>
      <button
        type="button"
        aria-label="Search"
        className="w-11 h-11 shrink-0 rounded-xl bg-primary hover:bg-primaryHover transition-
colors flex items-center justify-center"
      >
        <svg width="18" height="18" viewBox="0 0 24 24" fill="none" className="text-white">
          <circle cx="11" cy="11" r="7" stroke="currentColor" strokeWidth="2" />
          <line x1="16.5" y1="16.5" x2="21" y2="21" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
        </svg>
      </button>
    </div>
  );
}
```

Stats.jsx

```
export default function Stats({ total, completed }) {
  return (
    <div className="flex items-center justify-between py-4 border-b border-border text-sm text-
gray-400">
      <div className="flex items-center gap-2">
        <span>Tarefas totais:</span>
        <span className="bg-field border border-border rounded-md px-2 py-0.5 text-gray-200
font-medium min-w-[1.75rem] text-center">
          {total}
        </span>
      </div>
      <div className="flex items-center gap-2">
        <span>Concluído</span>
        <span className="bg-field border border-border rounded-md px-2 py-0.5 text-gray-200
font-medium">
```

```
          {completed} de {total}
        </span>
      </div>
    </div>
  );
}
```

TaskItem.jsx

```
function formatDate(isoDate) {
  const [year, month, day] = isoDate.split('-');
  return `${day}/${month}/${year}`;
}

function getTodayISO() {
  return new Date().toISOString().slice(0, 10);
}

export default function TaskItem({ task, onToggle, onDelete, onEdit }) {
  const isOverdue = !task.completed && task.due_date && task.due_date < getTodayISO();

  return (
    <div
      className={
        'group flex items-center gap-3 bg-field border rounded-xl px-4 py-3.5 transition-colors
' +
        (isOverdue ? 'border-red-500/40 hover:border-red-500/60' : 'border-border hover:border-
[#3a3a44]')
      }
    >
      <input
        type="checkbox"
        className="task-checkbox"
        checked={task.completed}
        onChange={(e) => onToggle(task.id, e.target.checked)}
      />

      <div className="flex-1 min-w-0">
        <span
          className={
            'block text-sm truncate ' +
            (task.completed ? 'text-gray-500 line-through' : isOverdue ? 'text-red-400' : 'text-
gray-100')
          }
        >
          {task.title}
        </span>

        {task.due_date && (
          <div className="flex items-center gap-1.5 mt-1">
            <svg
              width="12"
              height="12"
              viewBox="0 0 24 24"
              fill="none"
              className={isOverdue ? 'text-red-400' : 'text-gray-500'}
            >
              <rect x="3" y="5" width="18" height="16" rx="2" stroke="currentColor"
strokeWidth="2" />
              <path d="M3 10h18M8 3v4M16 3v4" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
```

```
            </svg>
            <span className={'text-xs ' + (isOverdue ? 'text-red-400' : 'text-gray-500')}>
              {formatDate(task.due_date)}
            </span>
            {isOverdue && (
              <span className="ml-1 text-[10px] font-medium uppercase tracking-wide text-red-400
bg-red-500/10 border border-red-500/30 rounded-full px-2 py-0.5">
                Em atraso
              </span>
            )}
          </div>
        )}
      </div>

      <div className="flex items-center gap-2 opacity-0 group-hover:opacity-100 transition-
opacity shrink-0">
        <button
          type="button"
          onClick={() => onEdit(task)}
          aria-label="Edit task"
          className="text-gray-500 hover:text-primary"
        >
          <svg width="16" height="16" viewBox="0 0 24 24" fill="none">
            <path
              d="m16.5 3.5 4 4L7 21H3v-4L16.5 3.5Z"
              stroke="currentColor"
              strokeWidth="2"
              strokeLinecap="round"
              strokeLinejoin="round"
            />
          </svg>
        </button>
        <button
          type="button"
          onClick={() => onDelete(task.id)}
          aria-label="Delete task"
          className="text-gray-500 hover:text-red-400"
        >
          <svg width="16" height="16" viewBox="0 0 24 24" fill="none">
            <path
              d="M4 7h16M9 7V5a2 2 0 0 1 2-2h2a2 2 0 0 1 2 2v2m2 0-1 13a2 2 0 0 1-2 2H8a2 2 0 0
1-2-2L5 7h14Z"
              stroke="currentColor"
              strokeWidth="2"
              strokeLinecap="round"
              strokeLinejoin="round"
            />
          </svg>
        </button>
      </div>
    </div>
  );
}
```

TaskList.jsx

```
import TaskItem from './TaskItem.jsx';

export default function TaskList({ tasks, onToggle, onDelete, onEdit, onOpenCreate }) {
  if (tasks.length === 0) {
    return (
```

```
      <div className="flex flex-col items-center justify-center py-20 gap-4">
        <button
          type="button"
          onClick={onOpenCreate}
          aria-label="Create task"
          className="w-14 h-14 rounded-full border-2 border-[#3a3a44] flex items-center justify-
center text-gray-400 hover:border-primary hover:text-primary transition-colors"
        >
          <svg width="22" height="22" viewBox="0 0 24 24" fill="none">
            <path d="M12 5v14M5 12h14" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
          </svg>
        </button>
        <div className="text-center">
          <p className="text-gray-300">You have no pending tasks.</p>
          <p className="text-gray-500 text-sm mt-1">Create one to get started.</p>
        </div>
      </div>
    );
  }

  return (
    <div className="flex flex-col gap-3 py-6">
      {tasks.map((task) => (
        <TaskItem key={task.id} task={task} onToggle={onToggle} onDelete={onDelete}
onEdit={onEdit} />
      ))}
    </div>
  );
}
```

TaskModal.jsx

```
import { useEffect, useState } from 'react';

export default function TaskModal({ open, mode = 'create', initialTask, onClose, onSubmit }) {
  const [title, setTitle] = useState('');
  const [dueDate, setDueDate] = useState('');

  useEffect(() => {
    if (open) {
      setTitle(initialTask?.title || '');
      setDueDate(initialTask?.due_date || '');
    }
  }, [open, initialTask]);

  if (!open) return null;

  const isEdit = mode === 'edit';

  const handleAccept = () => {
    if (!title.trim()) return;
    onSubmit({ title: title.trim(), dueDate: dueDate || null });
  };

  const handleKeyDown = (e) => {
    if (e.key === 'Enter') handleAccept();
    if (e.key === 'Escape') onClose();
  };

  return (
```

```
    <div className="fixed inset-0 z-50 flex items-center justify-center bg-black/60 backdrop-
blur-sm px-4">
      <div className="w-full max-w-md bg-panel border border-border rounded-2xl p-6 shadow-2xl">
        <h2 className="text-center text-lg font-semibold text-white mb-5">
          {isEdit ? 'Edit Task' : 'Create Task'}
        </h2>

        <div className="flex items-center gap-3 bg-field border border-border rounded-xl px-4
py-3 mb-4">
          <svg width="16" height="16" viewBox="0 0 24 24" fill="none" className="text-gray-500
shrink-0">
            <path
              d="m16.5 3.5 4 4L7 21H3v-4L16.5 3.5Z"
              stroke="currentColor"
              strokeWidth="2"
              strokeLinecap="round"
              strokeLinejoin="round"
            />
          </svg>
          <input
            autoFocus
            value={title}
            onChange={(e) => setTitle(e.target.value)}
            onKeyDown={handleKeyDown}
            placeholder="Write the task name.."
            className="bg-transparent outline-none w-full text-sm text-gray-200
placeholder:text-gray-500"
          />
        </div>

        <label className="block text-xs text-gray-500 mb-2 px-1">Due date (optional)</label>
        <div className="flex items-center gap-3 bg-field border border-border rounded-xl px-4
py-3 mb-6">
          <svg width="16" height="16" viewBox="0 0 24 24" fill="none" className="text-gray-500
shrink-0">
            <rect x="3" y="5" width="18" height="16" rx="2" stroke="currentColor"
strokeWidth="2" />
            <path d="M3 10h18M8 3v4M16 3v4" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
          </svg>
          <input
            type="date"
            value={dueDate || ''}
            onChange={(e) => setDueDate(e.target.value)}
            onKeyDown={handleKeyDown}
            className="bg-transparent outline-none w-full text-sm text-gray-200 [color-
scheme:dark]"
          />
          {dueDate && (
            <button
              type="button"
              onClick={() => setDueDate('')}
              aria-label="Clear due date"
              className="text-gray-500 hover:text-gray-300 shrink-0"
            >
              <svg width="14" height="14" viewBox="0 0 24 24" fill="none">
                <path d="M6 6l12 12M18 6 6 18" stroke="currentColor" strokeWidth="2"
strokeLinecap="round" />
              </svg>
            </button>
          )}
        </div>
```

```
        <div className="flex items-center justify-center gap-3">
          <button
            type="button"
            onClick={onClose}
            className="px-6 py-2.5 rounded-full bg-gray-100 text-gray-900 text-sm font-medium
hover:bg-white transition-colors"
          >
            Cancel
          </button>
          <button
            type="button"
            onClick={handleAccept}
            className="px-6 py-2.5 rounded-full bg-primary text-white text-sm font-medium
hover:bg-primaryHover transition-colors"
          >
            Accept
          </button>
        </div>
      </div>
    </div>
  );
}
```

5.3 api.js — a ponte entre React e Fastify

```
const BASE_URL = import.meta.env.VITE_API_URL || 'http://localhost:3333';

async function request(path, options = {}) {
  const response = await fetch(`${BASE_URL}${path}`, {
    headers: { 'Content-Type': 'application/json' },
    ...options,
  });

  if (!response.ok) {
    const data = await response.json().catch(() => ({}));
    throw new Error(data.message || 'Erro ao comunicar com o servidor.');
  }

  if (response.status === 204) return null;
  return response.json();
}

export const api = {
  listTasks: (search = '') =>
    request(`/api/tasks?search=${encodeURIComponent(search)}`),

  createTask: (title, dueDate) =>
    request('/api/tasks', {
      method: 'POST',
      body: JSON.stringify({ title, dueDate: dueDate || null }),
    }),

  updateTask: (id, { title, dueDate }) =>
    request(`/api/tasks/${id}`, {
      method: 'PATCH',
      body: JSON.stringify({ title, dueDate: dueDate ?? null }),
    }),
```

```
  toggleTask: (id, completed) =>
    request(`/api/tasks/${id}`, {
      method: 'PATCH',
      body: JSON.stringify({ completed }),
    }),

  deleteTask: (id) =>
    request(`/api/tasks/${id}`, { method: 'DELETE' }),
};
```

A função request centraliza tudo o que se repetiria em cada chamada: montar a URL completa, definir o cabeçalho JSON, tratar erros e converter a resposta. Assim, o resto do app nunca chama fetch diretamente — apenas usa api.listTasks(), api.createTask() etc., o que deixa o código muito mais legível.

5.4 Componentes de apresentação

Header.jsx

Componente estático (sem estado próprio) que apenas desenha o logo em SVG com gradiente e o subtítulo. Um bom exemplo de componente puramente visual.

SearchBar.jsx

```
export default function SearchBar({ value, onChange }) {
  return (
    <input
      value={value}
      onChange={(e) => onChange(e.target.value)}
      placeholder="Search tasks"
    />
  );
}
```

Note o padrão: o componente não guarda o texto digitado dentro de si mesmo. Ele recebe value (o texto atual) e onChange (uma função) como props, e apenas avisa o componente pai a cada tecla digitada. Isso é chamado de "componente controlado" — quem manda no valor do campo é o React, não o HTML puro.

Stats.jsx

Recebe total e completed já calculados (vindos da resposta da API) e apenas os exibe dentro de duas etiquetas, reproduzindo o "Total tasks: 0" e "Completed 0 de 0" do mockup.

TaskItem.jsx — a lógica de atraso

Este componente ganhou uma responsabilidade extra: decidir se uma tarefa está "em atraso". Vamos ver a função central:

```
function getTodayISO() {
  return new Date().toISOString().slice(0, 10);
}

const isOverdue = !task.completed && task.due_date && task.due_date < getTodayISO();
```

- `new Date().toISOString()` gera a data e hora atuais em formato `AAAA-MM-DDTHH:mm:ss`; com `.slice(0, 10)` ficamos só com a parte da data, no formato `AAAA-MM-DD`;
- como o banco também guarda `due_date` nesse mesmo formato (ano-mês-dia), podemos comparar as duas strings diretamente com `<` — nesse formato específico, comparação de texto e comparação de data cronológica dão o mesmo resultado;
- a tarefa só é considerada em atraso se, ao mesmo tempo: (1) ainda não foi concluída, (2) tem uma data definida e (3) essa data já passou.

O restante do componente usa esse booleano para trocar as classes do Tailwind (texto e selo vermelhos) e exibir o texto "Em atraso":

```
<span className={task.completed ? "text-gray-500 line-through"
  : isOverdue ? "text-red-400" : "text-gray-100"}>
  {task.title}
</span>

{isOverdue && (
  <span className="text-red-400 bg-red-500/10 border border-red-500/30 rounded-full">
    Em atraso
  </span>
)}
```

> **Por que calcular isso no frontend e não guardar um campo "atrasado" no banco?**
> Porque "atrasado" depende do dia de hoje, que muda a cada instante — se guardássemos esse status no banco, teríamos que atualizá-lo constantemente. É mais simples e mais correto calcular isso "na hora", sempre que a tela é desenhada, comparando a data salva com a data atual.

TaskList.jsx

Decide entre duas situações: se não há tarefas, mostra o estado vazio (com o botão "+" central, igual ao mockup); se há tarefas, percorre a lista com .map() criando um TaskItem para cada uma.

5.5 TaskModal — formulário reutilizável de criar/editar

Em vez de ter um modal para criar e outro, quase idêntico, para editar, criamos um único componente que recebe um mode ("create" ou "edit"):

```
export default function TaskModal({ open, mode = 'create', initialTask, onClose,
onSubmit }) {
  const [title, setTitle] = useState('');
  const [dueDate, setDueDate] = useState('');

  useEffect(() => {
    if (open) {
      setTitle(initialTask?.title || '');
      setDueDate(initialTask?.due_date || '');
    }
  }, [open, initialTask]);

  const isEdit = mode === 'edit';
  // ... renderiza título "Create Task" ou "Edit Task" conforme isEdit
}
```

- o hook `useEffect` roda toda vez que open ou initialTask mudam. Assim, quando o modal é aberto para editar, ele "pré-preenche" os campos com os dados da tarefa clicada; quando é aberto para criar, initialTask é nulo e os campos começam vazios;
- o operador `?.` (optional chaining) evita erro caso initialTask seja null — ele simplesmente resulta em undefined em vez de quebrar o programa;
- o mesmo botão "Accept" e a mesma validação servem para os dois modos — quem decide o que fazer com os dados é o componente pai, através da função onSubmit.

5.6 App.jsx — o cérebro da aplicação

É aqui que vive o estado da aplicação inteira e a comunicação com a API. Vamos entender por partes.

Código completo para digitar

```
import { useEffect, useState, useCallback } from 'react';
import Header from './components/Header.jsx';
import SearchBar from './components/SearchBar.jsx';
import Stats from './components/Stats.jsx';
import TaskList from './components/TaskList.jsx';
import TaskModal from './components/TaskModal.jsx';
import { api } from './api.js';

export default function App() {
  const [tasks, setTasks] = useState([]);
  const [total, setTotal] = useState(0);
  const [completed, setCompleted] = useState(0);
  const [search, setSearch] = useState('');
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  // Modal de criar/editar: { open, mode: 'create' | 'edit', task }
  const [modal, setModal] = useState({ open: false, mode: 'create', task: null });

  const loadTasks = useCallback(async (query) => {
    try {
      setLoading(true);
      const data = await api.listTasks(query);
      setTasks(data.tasks);
      setTotal(data.total);
      setCompleted(data.completed);
      setError(null);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }, []);

  useEffect(() => {
    const timeout = setTimeout(() => loadTasks(search), 300);
    return () => clearTimeout(timeout);
  }, [search, loadTasks]);

  const openCreateModal = () => setModal({ open: true, mode: 'create', task: null });
  const openEditModal = (task) => setModal({ open: true, mode: 'edit', task });
  const closeModal = () => setModal((prev) => ({ ...prev, open: false }));

  const handleSubmitModal = async ({ title, dueDate }) => {
```

```
    if (modal.mode === 'edit' && modal.task) {
      await api.updateTask(modal.task.id, { title, dueDate });
    } else {
      await api.createTask(title, dueDate);
    }
    closeModal();
    loadTasks(search);
  };

  const handleToggle = async (id, completedValue) => {
    setTasks((prev) =>
      prev.map((t) => (t.id === id ? { ...t, completed: completedValue } : t))
    );
    await api.toggleTask(id, completedValue);
    loadTasks(search);
  };

  const handleDelete = async (id) => {
    setTasks((prev) => prev.filter((t) => t.id !== id));
    await api.deleteTask(id);
    loadTasks(search);
  };

  return (
    <div className="min-h-screen bg-base">
      <Header />

      <main className="max-w-3xl mx-auto px-4 pb-16">
        <div className="bg-panel border border-border rounded-2xl px-6 py-8 sm:px-10">
          <h2 className="text-center text-2xl font-semibold text-white mb-6">Suas taferas</h2>

          <SearchBar value={search} onChange={setSearch} />

          <Stats total={total} completed={completed} />

          {error && (
            <p className="text-center text-red-400 text-sm mt-6">
              {error} — verifique se o backend está rodando.
            </p>
          )}

          {!error && loading && tasks.length === 0 ? (
            <p className="text-center text-gray-500 text-sm py-16">Loading...</p>
          ) : (
            <TaskList
              tasks={tasks}
              onToggle={handleToggle}
              onDelete={handleDelete}
              onEdit={openEditModal}
              onOpenCreate={openCreateModal}
            />
          )}
        </div>
      </main>

      <button
        type="button"
        onClick={openCreateModal}
        aria-label="Add task"
        className="fixed bottom-8 right-8 w-14 h-14 rounded-full bg-primary hover:bg-
primaryHover shadow-lg shadow-primary/30 flex items-center justify-center text-white transition-
colors"
```

```
      >
        <svg width="22" height="22" viewBox="0 0 24 24" fill="none">
          <path d="M12 5v14M5 12h14" stroke="currentColor" strokeWidth="2.5"
strokeLinecap="round" />
        </svg>
      </button>

      <TaskModal
        open={modal.open}
        mode={modal.mode}
        initialTask={modal.task}
        onClose={closeModal}
        onSubmit={handleSubmitModal}
      />
    </div>
  );
}
```

Explicando os trechos

Estado (useState)

```
  const [tasks, setTasks] = useState([]);
  const [total, setTotal] = useState(0);
  const [completed, setCompleted] = useState(0);
  const [search, setSearch] = useState('');
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  const [modal, setModal] = useState({ open: false, mode: 'create', task: null });
```

Cada useState guarda um pedaço de informação que, ao mudar, faz o React redesenhar a tela automaticamente. O modal, por exemplo, guarda em um único objeto se está aberto, em qual modo, e qual tarefa (se houver) está sendo editada.

Buscando dados (useCallback + useEffect)

```
  const loadTasks = useCallback(async (query) => {
    try {
      setLoading(true);
      const data = await api.listTasks(query);
      setTasks(data.tasks);
      setTotal(data.total);
      setCompleted(data.completed);
      setError(null);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }, []);

  useEffect(() => {
    const timeout = setTimeout(() => loadTasks(search), 300);
    return () => clearTimeout(timeout);
  }, [search, loadTasks]);
```

- `useCallback` evita recriar a função loadTasks a cada nova renderização, o que é uma boa prática quando essa função é usada dentro de um useEffect;
- o `useEffect` roda sempre que search muda — ou seja, toda vez que o usuário digita algo na busca;

- o truque do `setTimeout` de 300 milissegundos, combinado com o "cancelamento" (`clearTimeout`) no retorno do efeito, cria o chamado debounce: se o usuário está digitando rápido, cada nova tecla cancela a busca anterior agendada, e só a última, 300ms depois de parar de digitar, é realmente enviada ao servidor. Isso evita disparar uma requisição a cada letra digitada.

Abrindo o modal em modo criar ou editar

```
const openCreateModal = () => setModal({ open: true, mode: 'create', task: null });
const openEditModal = (task) => setModal({ open: true, mode: 'edit', task });
const closeModal = () => setModal((prev) => ({ ...prev, open: false }));

const handleSubmitModal = async ({ title, dueDate }) => {
  if (modal.mode === 'edit' && modal.task) {
    await api.updateTask(modal.task.id, { title, dueDate });
  } else {
    await api.createTask(title, dueDate);
  }
  closeModal();
  loadTasks(search);
};
```

Repare como handleSubmitModal decide, em um único lugar, se deve chamar updateTask ou createTask — o TaskModal nem precisa saber disso, ele só entrega { title, dueDate } de volta para quem o abriu.

Atualização otimista (handleToggle e handleDelete)

```
const handleToggle = async (id, completedValue) => {
  setTasks((prev) => prev.map((t) => (t.id === id ? { ...t, completed: completedValue
} : t)));
  await api.toggleTask(id, completedValue);
  loadTasks(search);
};
```

> **Conceito importante: atualização otimista**
>
> Antes mesmo de a resposta do servidor chegar, já atualizamos a tela localmente (setTasks muda o estado na hora). Isso faz a interface parecer instantânea. Em seguida, disparamos a chamada real à API e, quando ela responde, recarregamos a lista (loadTasks) para garantir que tudo — inclusive os contadores — fique 100% sincronizado com o banco de dados.

Renderização (o "return" do componente)

Por fim, o App organiza visualmente tudo isso: o Header no topo, o painel "Your Tasks" com SearchBar, Stats e TaskList, o botão flutuante "+" fixo no canto da tela, e o TaskModal, que só aparece quando modal.open é verdadeiro.

Capítulo 6 — Juntando as Peças: Fluxos Completos

Agora que conhecemos cada arquivo separadamente, vamos acompanhar o caminho completo de três ações, do clique do usuário até o banco de dados e de volta.

6.1 Fluxo: criar uma tarefa com data limite

1. Usuário clica no botão "+"

O App chama openCreateModal(), que muda o estado modal para { open: true, mode: "create", task: null }.

2. O TaskModal aparece

Como initialTask é null, os campos de nome e data começam vazios.

3. Usuário digita o nome e escolhe uma data, clica em "Accept"

O modal chama onSubmit({ title, dueDate }).

4. App decide o que fazer

Como modal.mode é "create", handleSubmitModal chama api.createTask(title, dueDate).

5. api.js envia a requisição

Um POST para /api/tasks com o corpo { title, dueDate } em JSON.

6. Fastify recebe e valida

A rota confere se o título não está vazio, insere no MySQL e devolve a tarefa criada com status 201.

7. App fecha o modal e recarrega a lista

closeModal() e loadTasks(search) buscam os dados atualizados, e a nova tarefa aparece na tela.

6.2 Fluxo: uma tarefa fica "em atraso"

Diferente dos outros fluxos, este não depende de nenhuma ação do usuário — ele acontece "sozinho", com o passar do tempo:

- a tarefa foi criada com due_date igual a, por exemplo, "2026-09-01";
- nenhuma ação foi tomada e a tarefa nunca foi marcada como concluída;
- ao carregar a tela em qualquer dia depois de 1º de setembro de 2026, o TaskItem calcula getTodayISO() (a data de hoje) e compara com due_date;
- como due_date é menor (mais antiga) que hoje, e completed é falso, isOverdue vira true;
- a tela então troca as classes CSS: o título fica vermelho e aparece o selo "Em atraso" — sem que nada tenha mudado no banco de dados.

6.3 Fluxo: editar uma tarefa

1. Usuário passa o mouse sobre uma tarefa e clica no lápis

O TaskItem chama onEdit(task), repassado pelo TaskList, chamando openEditModal(task) no App.

2. Estado do modal muda

modal vira { open: true, mode: "edit", task: <a tarefa clicada> }.

3. O TaskModal é reaberto, mas agora com dados

O useEffect detecta que initialTask não é nulo e preenche os campos com o título e a data já existentes.

4. Usuário altera algo e confirma

onSubmit({ title, dueDate }) é chamado novamente.

5. Desta vez, o modo é "edit"

handleSubmitModal chama api.updateTask(modal.task.id, { title, dueDate }) — um PATCH para /api/tasks/:id.

6. Fastify atualiza apenas os campos enviados

Graças à montagem dinâmica do SQL que vimos no Capítulo 4, apenas title e due_date são alterados; completed permanece como estava.

Capítulo 7 — Rodando o Projeto do Zero

Chegou a hora de colocar tudo para funcionar. Siga esta ordem exata.

7.1 Banco de dados

```
mysql -u root -p < backend/schema.sql
```

Digite a senha do MySQL quando solicitado. Isso cria o banco todolist e a tabela tasks.

7.2 Backend

```
cd backend
cp .env.example .env
# edite o .env e informe DB_USER, DB_PASSWORD, etc.
npm install
npm run dev
```

Se tudo estiver certo, o terminal mostrará algo como "Servidor rodando em http://localhost:3333".

7.3 Frontend

Abra um novo terminal (deixe o backend rodando no anterior):

```
cd frontend
cp .env.example .env
npm install
npm run dev
```

Acesse http://localhost:5173 no navegador. Você deve ver a tela "Your Tasks", pronta para uso.

7.4 Checklist de problemas comuns

- Erro de conexão com o banco: confira usuário/senha no .env do backend e se o serviço do MySQL está ativo;
- Erro de CORS no navegador: confirme que CORS_ORIGIN no .env do backend aponta para a porta correta do frontend (padrão 5173);
- Tela em branco no frontend: abra o console do navegador (F12) e veja se há mensagens de erro; confirme que o backend está rodando antes do frontend fazer as requisições;
- "Tabela tasks não existe": rode novamente o schema.sql ou apenas reinicie o backend, que cria a tabela automaticamente.

Capítulo 8 — Exercícios Propostos

A melhor forma de fixar o aprendizado é modificar o projeto. Tente os desafios abaixo, do mais simples ao mais avançado.

Nível Iniciante

- Adicione um contador de tarefas em atraso no componente Stats (ex.: "Em atraso: 2");
- Mude a cor do botão flutuante "+" para outra cor de sua escolha, usando classes do Tailwind;
- Adicione uma mensagem de confirmação (window.confirm) antes de excluir uma tarefa.

Nível Intermediário

- Crie um filtro (Todas / Pendentes / Concluídas) usando o parâmetro status que a rota GET /api/tasks já aceita, mas que o frontend ainda não utiliza;
- Adicione um campo de prioridade (baixa/média/alta) à tabela tasks e exiba um selo colorido correspondente na interface;
- Implemente paginação na listagem de tarefas, usando LIMIT e OFFSET no SQL.

Nível Avançado

- Adicione autenticação de usuários (login/senha), de forma que cada pessoa veja apenas suas próprias tarefas;
- Crie testes automatizados para as rotas do Fastify usando a própria API de testes do framework (app.inject());
- Publique o backend em um serviço de nuvem (Render, Railway, etc.) e o frontend em outro (Vercel, Netlify), conectando os dois em produção.

Capítulo 9 — Glossário

- **API (Interface de Programação de Aplicações):** conjunto de rotas que um sistema expõe para que outros sistemas conversem com ele.
- **API REST:** estilo de API que usa o protocolo HTTP e seus métodos (GET, POST, PATCH, DELETE) para representar ações sobre recursos.
- **Componente (React):** pedaço reutilizável de interface, escrito como uma função que retorna JSX.
- **CORS:** mecanismo de segurança do navegador que precisa ser liberado explicitamente pelo servidor para permitir requisições vindas de outra origem (outro domínio ou porta).
- **CRUD:** sigla para Create, Read, Update, Delete — as quatro operações básicas sobre dados.
- **Debounce:** técnica para atrasar a execução de uma ação até que o usuário pare de gerar eventos (ex.: parar de digitar) por um curto período.
- **Estado (state):** dado que, quando alterado, faz o React redesenhar a tela automaticamente.
- **Fastify:** framework Node.js para criação de servidores e APIs HTTP.
- **Hook (React):** função especial do React (como useState, useEffect, useCallback) que permite usar estado e outros recursos dentro de componentes funcionais.
- **JSX:** extensão de sintaxe do JavaScript que permite escrever elementos parecidos com HTML dentro do código.
- **Pool de conexões:** conjunto de conexões com o banco de dados mantidas abertas e reaproveitadas, para melhorar a performance.
- **Props:** abreviação de "properties"; são os dados passados de um componente pai para um componente filho em React.
- **SQL Injection:** tipo de ataque em que um usuário mal-intencionado insere comandos SQL em campos de formulário; evitado com consultas parametrizadas.
- **Vite:** ferramenta de build e desenvolvimento para projetos frontend modernos, usada aqui para rodar o React.

Fim da apostila — bons estudos e bom código!