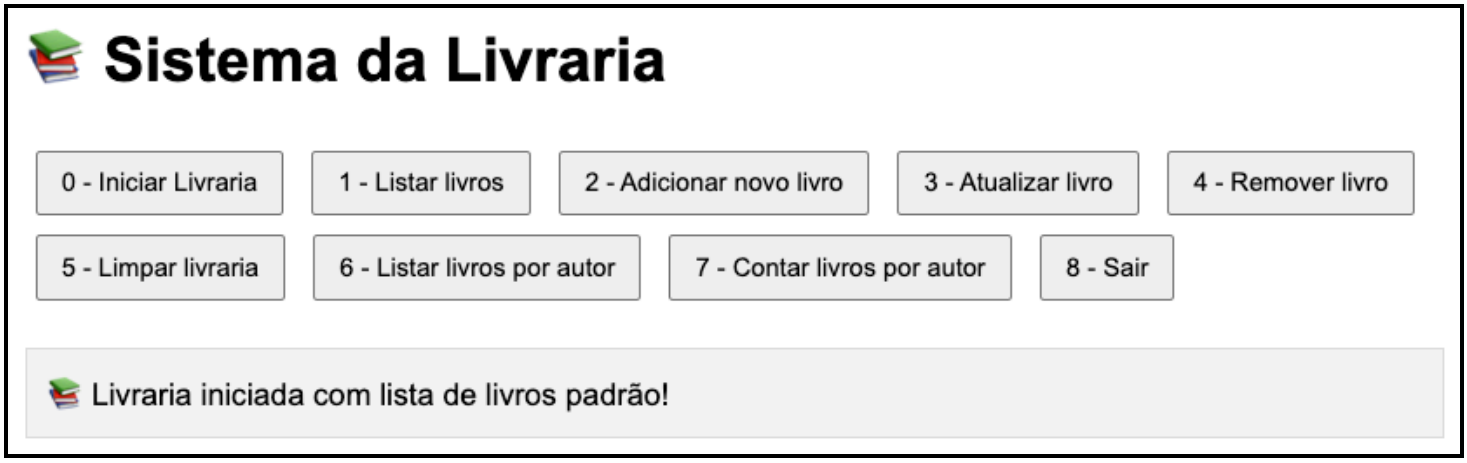


A. ARQUIVOS DO CRUD

1. Estrutura geral dos arquivos

- **`index.html`**
  - Define a estrutura estática da interface: `<header>`, `<main>`, `<form>`, `<table>`, `<canvas>` para gráficos etc.
  - Carrega os scripts **na ordem correta** (primeiro `lib.js`, depois `ui.js` e por fim `Chart.js`), garantindo que as funções e objetos estejam disponíveis quando chamados.
  - O HTML sozinho **não tem lógica**; ele fornece o **DOM inicial** que os scripts vão manipular.
- **`lib.js`**
  - Contém funções de **lógica de negócio e persistência**, como:
    - `getBooks()`, `saveBooks()`, `resetBooks()` → leitura/escrita em `localStorage`.
    - `addBook()`, `deleteBook()` → atualização da coleção.
  - Não conhece a interface (DOM). Ele é um **módulo de dados**.
- **`ui.js`**
  - É responsável pela **camada de apresentação e interação**:
    - Escuta eventos do DOM (`onsubmit`, `onclick`, etc.).
    - Chama funções de `lib.js` para manipular dados.
    - Atualiza elementos do DOM dinamicamente (`innerHTML`, `appendChild`, etc.).
    - Chama `Chart.js` para atualizar os gráficos a partir dos dados.
- **`Chart.js` (biblioteca externa)**
  - Renderiza os dados recebidos em forma de **gráficos interativos** (canvas API).
  - Não sabe nada sobre `localStorage` nem sobre HTML; ele só recebe dados **pré-processados** do `ui.js`.

2. Papel do DOM

O **DOM (Document Object Model)** é a "ponte" entre o HTML estático e os scripts JS:

- O navegador **transforma o HTML em uma árvore de nós (DOM Tree)**.
- `ui.js` obtém referências a esses nós usando `document.getElementById`, `querySelector`, etc.
- Quando o usuário interage (ex.: clica em "Adicionar Livro"):
  - O evento é capturado no DOM.
  - O `ui.js` processa a entrada → chama `lib.js` → atualiza os dados no `localStorage`.
  - Depois, o `ui.js` **modifica o DOM** para refletir a mudança (nova linha na tabela, gráfico atualizado).

Ou seja:

```
Usuário → DOM Event → ui.js → lib.js (dados) → localStorage
             ↑                                        ↓
             └──────────── Atualização do DOM ←────────
```

3. Fluxo de dados

1. **Carregamento da página**
   - `index.html` monta a estrutura básica do DOM.
   - `lib.js` é carregado primeiro → funções de persistência ficam disponíveis.
   - `ui.js` é carregado depois → inicializa interface (ex.: chama `renderBooks()` para preencher a tabela com `getBooks()`).
   - `Chart.js` é carregado por último, permitindo que `ui.js` já tenha os dados prontos para renderização.
2. **Interação do usuário (CRUD)**
   - Usuário insere livro no formulário → `ui.js` captura evento.
   - `ui.js` chama `addBook()` em `lib.js`.
   - `lib.js` salva em `localStorage`.
   - `ui.js` atualiza DOM (adiciona a linha na tabela) e redesenha gráfico com `Chart.js`.
3. **Persistência automática**
   - Mesmo ao fechar/reabrir o navegador, os dados permanecem em `localStorage`.
   - Ao carregar novamente, `ui.js` lê via `getBooks()` e reconstrói a interface.

4. Separação de responsabilidades

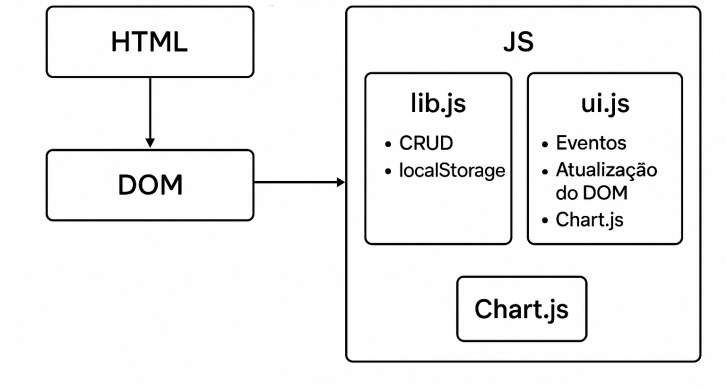
- **HTML (estrutura estática)** → define onde os dados vão aparecer.

- **DOM (estrutura dinâmica)** → representação viva do HTML manipulável pelo JS.
- **`lib.js` (dados)** → CRUD sobre `localStorage`.
- **`ui.js` (apresentação)** → manipula DOM + orquestra dados e eventos.
- **Chart.js (visualização avançada)** → transforma arrays de dados em gráficos.

Em resumo:

- O **DOM é o elo vivo** entre o que o usuário vê e os dados internos.
- O `lib.js` fornece a **lógica de persistência**.
- O `ui.js` é o **tradutor** entre DOM, `lib.js` e Chart.js.
- Essa arquitetura reflete o padrão clássico **MVC (Model-View-Controller)**:
  - **Model** → `lib.js + localStorage`
  - **View** → HTML + Chart.js
  - **Controller** → `ui.js`

MVC aplicado ao CRUD da Livraria



B. PASSO A PASSO PARA EXECUTAR O CRUD

1. Pré-requisitos

- Ter **Python** instalado na máquina
  Teste no terminal:

```
python --version
```

ou

```
python3 --version
```

- Ter o **VSCode** instalado (recomendo a extensão `CodeRunner` para facilitar a edição de HTML/JS).

2. Abrir o projeto no VSCode

1. Crie uma pasta chamada `livraria` (ou outro nome que preferir).
2. Coloque dentro dela os arquivos:
   - `index.html`
   - `ui.js`
   - `lib.js`
3. Abra essa pasta no VSCode (`File > Open Folder`).

3. Abrir o terminal integrado do VSCode

- No VSCode, vá em: **Terminal > New Terminal**.

4. Subir um servidor HTTP local

No terminal do VSCode, digite **um desses comandos**:

- Para Python 3 (mais comum hoje em dia):

```
python3 -m http.server 8000
```

- Para algumas instalações do Windows (onde `python` aponta para o Python 3):

```
python -m http.server 8000
```

⚠️ Se já tiver algo rodando na porta 8000, pode trocar por outro número, ex.: `8080` .

5. Acessar no navegador

- Abra o navegador (Chrome, Edge ou Firefox).
- Digite na barra de endereços:

```
http://localhost:8000/index.html
```

6. Usando o sistema

- A tela inicial mostrará os **botões de ação**.
- Clique em **"Iniciar Livraria"** para carregar os livros iniciais no `localStorage` .

- Depois, pode:
  - Listar livros,
  - Adicionar novos,
  - Atualizar,
  - Remover,
  - Contar por autor,
  - etc..

7. Finalizando

- Para parar o servidor, volte ao terminal e pressione:
  Ctrl + C.

Resumo em 3 comandos no terminal:

```
cd livraria
python3 -m http.server 8000
```

Depois é só abrir no navegador:

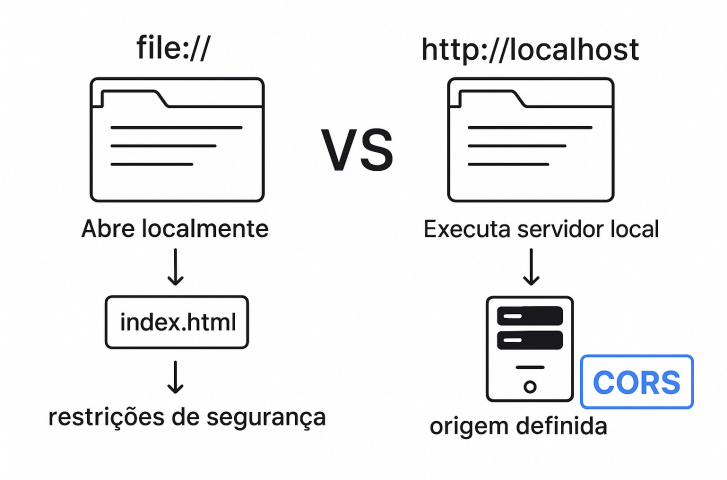
```
http://localhost:8000/index.html
```

C. POR QUE NÃO SIMPLESMENTE ABRIR O HTML LOCALMENTE?

Quando você abre um arquivo **index.html** diretamente no navegador (via `file://`), ele funciona para HTML e CSS simples, mas apresenta limitações importantes para **JavaScript moderno**.

- **Regras de segurança do navegador**: navegadores bloqueiam certas operações de JS quando o arquivo é aberto localmente, como requisições `fetch` , importações de módulos (`import/export`) e acesso a APIs modernas.
- **CORS (Cross-Origin Resource Sharing)**: controla quais sites ou origens podem acessar recursos de outro site/origem. Se você tenta fazer uma requisição JavaScript sem servidor, a origem do arquivo é considerada `null` e o navegador bloqueia a operação por segurança.
- **localStorage**: funciona em arquivos locais, mas algumas funções dependem do contexto de origem (origin), que se comporta de forma diferente sem servidor. Isso pode gerar inconsistências ou problemas ao carregar recursos externos.

- **Módulos ES6**: ao usar `type="module"` no `<script>`, o navegador exige que o arquivo seja servido por um servidor HTTP, caso contrário ocorrerá erro de CORS.
- **Simulação de ambiente real**: aplicações web normalmente rodam em servidores; usar um servidor local permite testar a aplicação de forma mais próxima do ambiente de produção, incluindo rotas, AJAX, APIs e armazenamento persistente.



Resumindo: executar um servidor local (mesmo simples, como `python -m http.server`) garante que JavaScript moderno funcione corretamente, evita erros de CORS e simula o comportamento real da aplicação.