```
const form = document.querySelector("#all-form");
const input = document.querySelector("#all-input");
const list = document.querySelector("#all-list");
const filterButtons = document.querySelector(".filter");

let todos = [];
```

Esse trecho de código está **selecionando elementos do HTML** e preparando uma variável para armazenar as tarefas da lista.

📌 1. const form = document.querySelector("#all-form");

- Usa document.querySelector() para **selecionar um elemento do HTML**.
- O parâmetro "#all-form" é um **seletor CSS** — nesse caso, está procurando um elemento com **id="all-form"**.
- O resultado (o elemento HTML) é armazenado na constante form.

📌 2. const input = document.querySelector("#all-input");

- Está selecionando um campo <input> com **id="all-input"**.
- Esse é o campo onde o usuário digita a nova tarefa.

📌 3. const list = document.querySelector("#all-list");

- Seleciona o elemento com id all-list, que provavelmente é uma <ul> ou <ol>, onde as tarefas vão aparecer.

📌 4. const filterButtons = document.querySelector(".filter");

- Aqui ele usa .filter com ponto (.), o que significa que está buscando uma **classe CSS**, não um id.
- **Atenção:** querySelector() pega **apenas o primeiro** elemento com essa classe.
- Se você tiver **vários botões de filtro**, o ideal é usar querySelectorAll() para pegar todos:

```
const filterButtons = document.querySelectorAll(".filter");
```

📌 5. let todos = [];

- Isso declara uma variável chamada todos, que é um **array vazio** no início.
- Ele vai guardar as tarefas que o usuário digitar, no formato de objetos (ex: { id, text, completed }).

💡 Resumo:

Esse trecho está preparando seu script para:

- Saber **onde o usuário digita** (input)
- Saber **quando o formulário é enviado** (form)
- Saber **onde mostrar as tarefas** (list)
- Saber **quais botões de filtro existem** (filterButtons)
- Guardar **as tarefas digitadas** (todos)

```
form.addEventListener("submit", (e) => {
  e.preventDefault();
```

🧠 O que está acontecendo aqui?

1. `form.addEventListener("submit", ...)`

- Isso está **escutando um evento** no formulário (`form`).
- O evento que ele está ouvindo é o **"submit"** — ou seja, quando o usuário clica no botão de "Adicionar" ou pressiona Enter.
- Quando isso acontece, ele executa a função que está entre parênteses: `(e) => { ... }`

2. `(e) => { ... }`

- Isso é uma **função arrow** (setinha).
- O **e** é um parâmetro que representa o **evento** que aconteceu (nesse caso, o "submit").

3. `e.preventDefault();`

- Essa linha **impede o comportamento padrão** do formulário.
- O comportamento padrão de um `<form>` ao ser enviado é **recarregar a página**.
- Como estamos criando tudo com JavaScript, não queremos que a página recarregue — então usamos `preventDefault()` pra **evitar isso**.

📌 Em resumo:

Esse trecho está dizendo:

“Quando o formulário for enviado, **não recarregue a página**, e sim execute o código que vamos escrever aqui dentro.”

```
const text = input.value.trim();
if (text !== " ") {
  const todo = {
    id: Date.now(),
    text,
    completed: false,
  };
  todos.push(todo);
  input.value = " ";
  renderTodos();
  saveToLocalStorage();
}
```

Esse trecho é uma parte super importante do funcionamento da sua To-Do List. Ele **cria uma nova tarefa e a adiciona à lista**.

📌 `const text = input.value.trim();`

- Pega o que o usuário digitou no campo de input.
- `.value` acessa o valor digitado.
- `.trim()` remove **espaços em branco** do começo e do fim.
  - Exemplo: `" Tarefa 1 "` vira `"Tarefa 1"`.

📌 `if (text !== " ") {`

- **Essa linha está com um pequeno erro!** 🛑

A verificação `text !== " "` só funciona se o valor for **exatamente um espaço**.

- **Melhor usar:** `if (text !== "")` ou `if (text)` — para garantir que só tarefas **não vazias** sejam adicionadas.

📌 Correto seria:

```
if (text !== "") {
```

ou:

```
if (text) {
```

📌 `const todo = { ... };`

Aqui é criado um **objeto de tarefa**, com três propriedades:

```
{
  id: Date.now(),        // Um ID único baseado na data/hora atual
  text: text,            // O texto digitado pelo usuário
  completed: false       // A tarefa começa como "não concluída"
}
```

- `Date.now()` gera um número único com o horário atual (excelente pra usar como ID).
- `text` vem da constante criada na primeira linha.
- `completed: false` indica que a tarefa ainda não foi marcada como feita.

📌 `todos.push(todo);`

- Adiciona esse novo objeto à lista de tarefas `todos`.

📌 `input.value = " ";`

- Isso limpa o campo de input depois que o usuário adiciona uma tarefa.
- **Dica:** em vez de `" "`, o ideal é usar `""` (string vazia), assim:

```
input.value = "";
```

📌 `renderTodos();`

- Atualiza a tela para exibir a nova lista com a tarefa que acabou de ser adicionada.

📌 `saveToLocalStorage();`

- Salva a lista atual de tarefas no navegador usando `localStorage`, pra que ela continue existindo mesmo se a página for recarregada.

🔁 Resumo:

Esse bloco faz:

1. Pega o texto digitado pelo usuário
2. Verifica se não está vazio
3. Cria uma tarefa com ID único
4. Adiciona à lista
5. Limpa o campo
6. Atualiza a tela
7. Salva no `localStorage`

```
function renderTodos(filter = "all") {
  list.innerHTML = "";

  const filteredTodos = todos.filter((todo) => {
    if (filter === "all") return true;
    if (filter === "active") return !todo.completed;
    if (filter === "completed") return todo.completed;
  });

  filteredTodos.forEach((todo) => {
    const li = document.createElement("li");
    li.classList.toggle("completed", todo.completed);

    const span = document.createElement("span");
    span.textContent = todo.text;
    span.addEventListener("click", () => toggleTodo(todo.id));

    const btn = document.createElement("button");
    btn, (textContend = "❌");
    btn.addEventListener("click", () => removeTodo(todo.id));

    li.append(span, btn);
    list.appendChild(li);
  });
}
```

Essa função é **responsável por exibir as tarefas na tela**, com base no filtro selecionado (todas, pendentes ou concluídas).

🧠 1. function renderTodos(filter = "all")

- Cria uma função chamada renderTodos.
- O parâmetro filter tem valor padrão "all", ou seja, se nada for passado, ele mostra **todas as tarefas**.
- Você pode passar "active" (pendentes) ou "completed" (concluídas) para mudar o que será exibido.

🖌️ 2. list.innerHTML = "";

- Limpa o conteúdo atual da <ul> (lista).
- Isso evita que tarefas sejam duplicadas quando atualizamos a lista.

🔍 3. Filtragem das tarefas

```
const filteredTodos = todos.filter((todo) => {
  if (filter === "all") return true;
  if (filter === "active") return !todo.completed;
  if (filter === "completed") return todo.completed;
});
```

- Essa parte cria um **novo array** com as tarefas filtradas:
  - "all" → mostra todas
  - "active" → só as **não concluídas**
  - "completed" → só as **concluídas**
- .filter() percorre o array todos e **retorna apenas os que correspondem ao filtro.**

🧱 4. Loop para criar e exibir cada tarefa

```
filteredTodos.forEach((todo) => {
  const li = document.createElement("li");
  li.classList.toggle("completed", todo.completed);
```

- Para cada tarefa filtrada, cria um <li> (item da lista).
- Se a tarefa estiver concluída (completed: true), adiciona a **classe CSS completed**, que normalmente deixa o texto com text-decoration: line-through.

🖊️ 5. Criando o texto da tarefa

```
 const span = document.createElement("span");
  span.textContent = todo.text;
  span.addEventListener("click", () => toggleTodo(todo.id));
```

- Cria uma para mostrar o texto da tarefa.

- Quando o usuário clica no texto, chama a função toggleTodo() que **marca como concluída ou não.**

❌ 6. Criando o botão de remover

```
const btn = document.createElement("button");
 btn, (textContend = "❌");
 btn.addEventListener("click", () => removeTodo(todo.id));
```

⚠️ **Atenção aqui:** tem um erro de digitação nessa linha!

Você escreveu:

```
btn, (textContend = "❌");
```

🔧 **Correção:**

```
btn.textContent = "❌";
```

- Isso cria um botão com o símbolo ❌.
- Quando clicado, chama a função removeTodo(todo.id), que remove a tarefa.

📌 7. Montando o item da lista

```
li.append(span, btn);
 list.appendChild(li);
```

- Adiciona o span (texto) e o btn (botão) dentro da <li>.
- E então adiciona essa <li> à <ul> (a lista inteira de tarefas).

☑ Resumo Geral

A função renderTodos() faz:

1. Limpa a lista na tela
2. Filtra as tarefas com base no filtro selecionado
3. Para cada tarefa:
    a. Cria um item de lista
    b. Mostra o texto
    c. Permite marcar como concluída
    d. Permite excluir
4. Exibe tudo no HTML

```
function toggleTodo(id) {
  todos = todos.map((todo) =>
    todo.id === id ? { ...todo, completed: !todo.completed } : todo
  );
  renderTodos(currentFilter);
  saveToLocalStorage();
}
```

Essa função toggleTodo(id) serve para **alternar o estado de uma tarefa entre "concluída" e "pendente".**

🧠 1. function toggleTodo(id)

- Define uma função chamada toggleTodo.
- Ela recebe um **id** como parâmetro — que é o identificador da tarefa que o usuário clicou para marcar ou desmarcar como concluída.

🔁 2. Atualizando as tarefas:

```
todos = todos.map((todo) =>
  todo.id === id ? { ...todo, completed: !todo.completed } : todo
);
```

Essa linha é poderosa, então vamos por partes:

todos.map(...)

- .map() percorre o array todos e **retorna um novo array**, com possíveis modificações.

todo.id === id ? ... : ...

- Para cada todo, ele verifica: o id da tarefa é igual ao id recebido?
  - **Se for igual**, isso significa que é a tarefa que o usuário clicou → então **ela deve ser atualizada.**
  - **Se não for**, a tarefa é mantida como está.

{ ...todo, completed: !todo.completed }

- Se a tarefa for a correta, ele retorna um **novo objeto**, copiando todos os dados com ...todo, **mas invertendo o valor de completed**:
  - Se completed era false, vira true.
  - Se era true, vira false.

💡 Isso é chamado de **imutabilidade** — ao invés de modificar diretamente o objeto original, você cria uma cópia modificada.

🔄 3. renderTodos(currentFilter);

- Depois de atualizar a tarefa, essa linha **re-renderiza a lista** na tela.
- currentFilter é a variável que diz qual filtro está ativo ("all", "active" ou "completed"), pra manter a visão do usuário igual ao que ele estava vendo.

💾 4. saveToLocalStorage();

- Atualiza o armazenamento local com o novo estado das tarefas.
- Assim, se a página for recarregada, a tarefa marcada como feita ainda estará feita.

☑️ Resumo

A função toggleTodo(id) faz:

1. Encontra a tarefa com aquele id
2. Inverte o valor de completed
3. Atualiza o array todos com a nova versão
4. Atualiza a interface com renderTodos()
5. Salva tudo no localStorage

```
function removeTodo(id) {
  todos = todos.filter((todo) => todo.id !== id);
  renderTodos(currentFilter);
  saveToLocalStorage();
}
```

A função `removeTodo(id)` serve para **remover uma tarefa da lista** com base no seu ID.

🧠 1. `function removeTodo(id)`

- Cria uma função chamada `removeTodo`.
- Recebe um **`id`** como parâmetro — que é o identificador da tarefa que o usuário clicou no botão ❌ para remover.

🖌️ 2. `todos = todos.filter((todo) => todo.id !== id);`

- Aqui está o coração da função.

Quebra da lógica:

- `.filter()` cria um **novo array**, **removendo** itens que **não** atendem à condição.
- A condição `todo.id !== id` significa:
  - **"Mantenha todos os itens que *não* têm esse id"**
- Resultado: a tarefa com aquele `id` **é excluída da lista**.

✅ Exemplo: Suponha que você tenha 3 tarefas com os IDs: `101`, `102`, `103`

Se você passar `102` como parâmetro, a linha acima **vai remover apenas** a tarefa com ID 102.

🔄 3. `renderTodos(currentFilter);`

- Re-renderiza a lista na tela usando o filtro atual (por exemplo: todas, ativas ou concluídas).
- Isso faz com que a tarefa desapareça da interface imediatamente após ser excluída.

💾 4. `saveToLocalStorage();`

- Atualiza o armazenamento local do navegador com a nova lista (sem a tarefa removida).
- Isso garante que, mesmo se a página for recarregada, a tarefa deletada **não volte**.

☑️ Resumo

A função `removeTodo(id)`:

1. Remove a tarefa com o ID informado
2. Atualiza a interface
3. Salva a lista atualizada no `localStorage`

```
function saveToLocalStorage() {
  localStorage.setItem("todos", JSON.stringify(todos));
}
```

A função `saveToLocalStorage()` é bem simples, mas super importante! Ela serve para **salvar as tarefas no navegador**, de forma que **não se percam ao recarregar a página**.

🔐 O que é `localStorage`?

- É um recurso do navegador que permite armazenar dados **localmente**, no próprio navegador do usuário.
- Os dados **ficam salvos mesmo depois que a página é fechada ou o computador é desligado**.
- Ele armazena os dados em formato de **string (texto)**.

🧠 1. `localStorage.setItem("todos", ...)`

- `setItem` é o método que salva uma **chave/valor** no `localStorage`.
- `"todos"` é a **chave** que estamos usando para salvar o array de tarefas.
  - Você pode imaginar como o nome de uma "gaveta" onde estamos guardando os dados.
- O valor precisa ser uma **string**, então...

🔄 2. `JSON.stringify(todos)`

- O `todos` é um array de objetos JavaScript.
- `JSON.stringify()` transforma esse array em uma **string JSON**, que pode ser armazenada no `localStorage`.

🔎 Exemplo:

```
let todos = [
  { id: 1, text: "Ler", completed: false },
  { id: 2, text: "Estudar", completed: true }
];
```

Depois de aplicar:

```
JSON.stringify(todos)
```

O resultado é:

```
"[{\"id\":1,\"text\":\"Ler\",\"completed\":false},{\"id\":2,\"text\"
:\"Estudar\",\"completed\":true}]"
```

Ou seja, um texto formatado que representa os dados.

☑ Resultado final:

Toda vez que a função `saveToLocalStorage()` for chamada:

1. Ela **pega o array `todos`**
2. Converte para string
3. Salva no navegador com a chave `"todos"`

💡 E depois?

Quando o app for carregado (ex: quando você abre a página de novo), normalmente você usaria:

```
const stored = localStorage.getItem("todos");
todos = JSON.parse(stored) || [];
```

Assim você **recupera os dados salvos** e os transforma de volta em um array.

```
function loadFromLocalStorage() {
  const data = localStorage.getItem("todos");
  if (data) {
    todos = JSON.parse(data);
  }
}
```

essa função `loadFromLocalStorage()` faz — ela é **a outra metade da mágica** de manter as tarefas salvas no navegador.

🧠 Objetivo geral:

A função serve para **carregar os dados salvos no navegador (localStorage)** e colocar de volta no array `todos`, assim que o site ou app for aberto.

🔍 Linha por linha:

`function loadFromLocalStorage() {`

- Declara uma função chamada `loadFromLocalStorage`.

`const data = localStorage.getItem("todos");`

- `localStorage.getItem("todos")`:
  - Pega o **valor salvo anteriormente** com a chave `"todos"` — que foi armazenado com `saveToLocalStorage()`.
  - Esse valor é uma **string JSON**.
- `data` agora é uma string, tipo:

`"[{\"id\":1,\"text\":\"Estudar JS\",\"completed\":false}]"`

`if (data) {`

- Verifica se **existe algum dado salvo**.
- Se `data` for `null` (ou seja, ainda não tem nada salvo), ele **não tenta carregar nada** (evita erro).

```
todos = JSON.parse(data);
```

- `JSON.parse(data)` transforma a string JSON de volta em **array de objetos JavaScript**.
- Atribui isso ao array `todos`, ou seja: agora o app tem de volta a lista de tarefas salvas.

☑ Resumo

A função faz isso:

1. **Busca no navegador** os dados salvos com a chave `"todos"`.
2. Se existir, **transforma a string em um array de objetos**.
3. **Atualiza a variável `todos`** com esses dados.

💡 Exemplo prático:

Se o localStorage tiver salvo isto:

```
"[{\"id\":101,\"text\":\"Lavar roupa\",\"completed\":false}]"
```

Depois de `loadFromLocalStorage()` rodar, `todos` será:

```
[
  { id: 101, text: "Lavar roupa", completed: false }
]
```

E tudo volta como estava! ✨

```
let currentFilter = "all";

loadFromLocalStorage();
renderTodos();

filterButtons.forEach(button => {
  button.addEventListener("click", () => {
    filterButtons.forEach(btn => btn.classList.remove("active"));
    button.classList.add("active");
    currentFilter = button.dataset.filter;
    renderTodos(currentFilter);
  });
});
```

essa parte — ela é responsável por **mostrar os filtros (Todos, Ativos, Completos)** funcionando corretamente no app.

🧠 1. `let currentFilter = "all";`

- Cria uma variável chamada `currentFilter` e define o valor inicial como `"all"` (ou seja, mostrar **todas as tarefas**).
- Isso é usado para controlar **qual filtro está ativo** no momento.

💾 2. `loadFromLocalStorage();`

- Chama a função que você já viu:
  - Ela **carrega os dados salvos no navegador** e os coloca no array `todos`.

💻 3. `renderTodos();`

- Mostra as tarefas na tela usando o filtro atual (que neste momento é `"all"`).
- Isso significa que, assim que a página carrega, **todas as tarefas salvas aparecem**.

⭕ 4. Filtro dos botões:

```
filterButtons.forEach(button => {
```

- `filterButtons` representa todos os **botões de filtro** da interface (por exemplo: "Todos", "Ativos", "Concluídos").
- Esse `.forEach` vai **adicionar um ouvinte de clique em cada botão**.

💡 5. Ao clicar em um botão:

```
button.addEventListener("click", () => {
```

Quando o usuário clica em um botão de filtro:

🧹 *Remove classe "active" de todos os botões:*

```
filterButtons.forEach(btn => btn.classList.remove("active"));
```

- Limpa o visual → todos os botões perdem o estilo de "selecionado".

☑️ *Adiciona "active" ao botão clicado:*

```
button.classList.add("active");
```

- Marca visualmente o botão que foi clicado (estilo "ativo").

🔄 *Atualiza o filtro atual:*

```
currentFilter = button.dataset.filter;
```

- `dataset.filter` acessa o valor do atributo `data-filter` do botão (HTML).
    - Exemplo: `<button data-filter="active">Ativos</button>`
- Assim, o filtro atual muda para: `"all"`, `"active"` ou `"completed"` — dependendo do botão clicado.

🔄 *Atualiza a lista na tela:*

```
renderTodos(currentFilter);
```

- Re-renderiza a lista de tarefas usando o novo filtro.
- Exibe **apenas as tarefas correspondentes** (todas, pendentes ou concluídas).

☑️ Resumo geral:

Essa parte do código:

1. Carrega os dados salvos (`loadFromLocalStorage`)
2. Mostra todas as tarefas (`renderTodos`)
3. Adiciona comportamento de clique aos botões de filtro
4. Sempre que um botão for clicado:
    a. Troca a aparência do botão selecionado
    b. Atualiza o filtro atual
    c. Atualiza a lista de tarefas mostrada