SVELTE

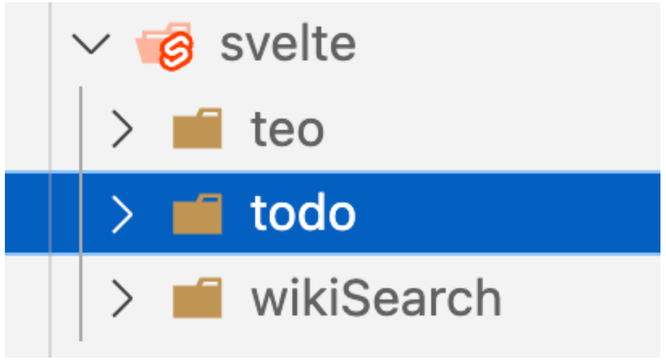
A declarative interface framework

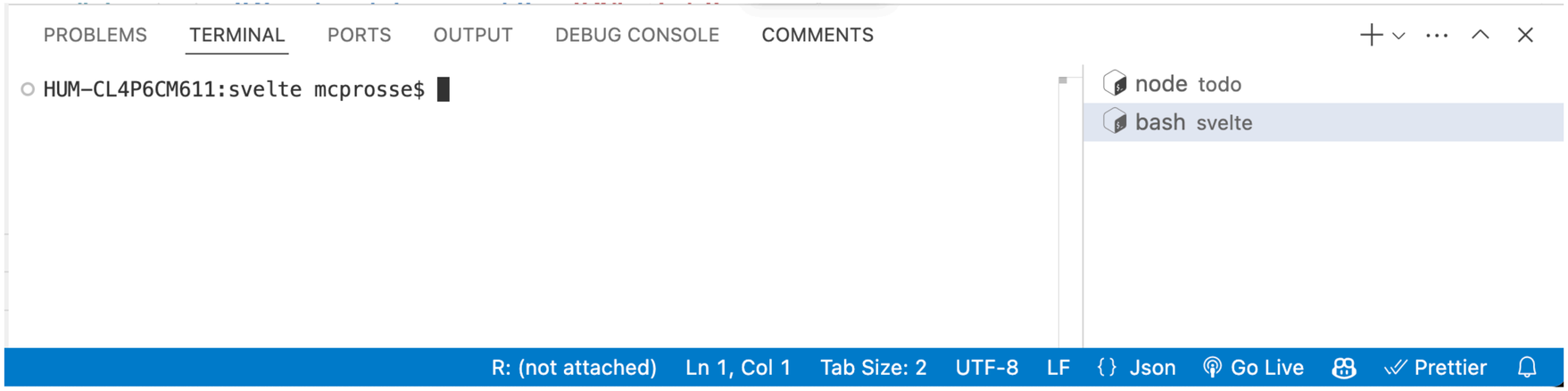
FIRST PROJECT

A To Do App using SvelteKit

CONFIGURE VS CODE

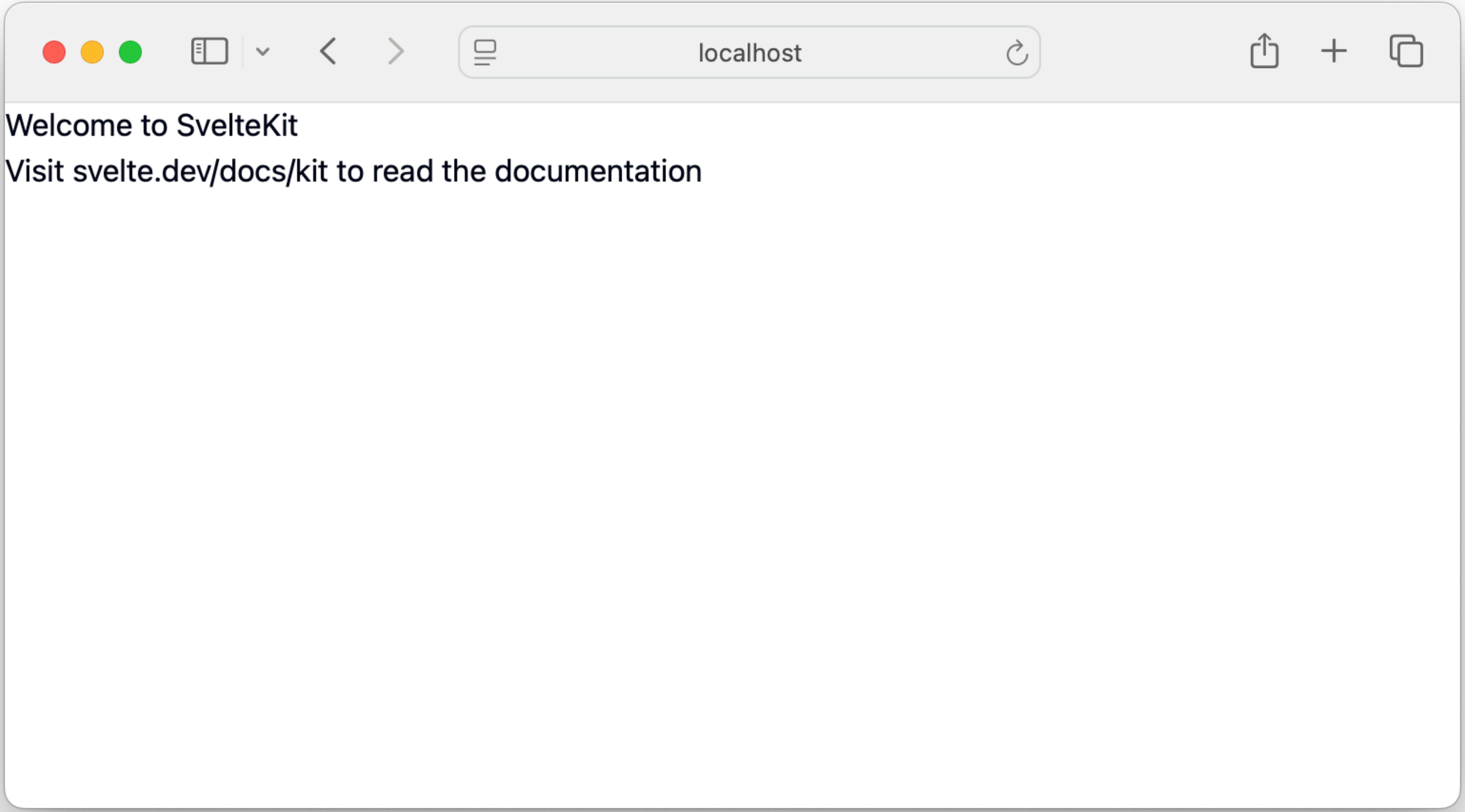
- Organize directories for various apps
- Create a svelte directory, for example
- Don't create your todo app directory manually
- I use two terminals when developing
- Navigate into the svelte directory
- Next, install…





INSTALL SVELTE AND INITIATE SHADCN

- Using SvelteKit CLI (double check sv@<version>)
  - Docs: https://next.shadcn-svelte.com/docs/installation/sveltekit
  - **pnpm dlx sv@0.6.18 create digs-todo**
    - SvelteKit minimal
    - Yes, using Typescript syntax
    - prettier, eslint, tailwindcss (typography and forms)
    - Install with pnpm
  - CHANGE INTO digs-todo DIRECTORY!
  - **pnpm i**
  - **pnpm dlx shadcn-svelte@next init**
    - Chose your style and base color: either Default or New York
  - **pnpm run dev**

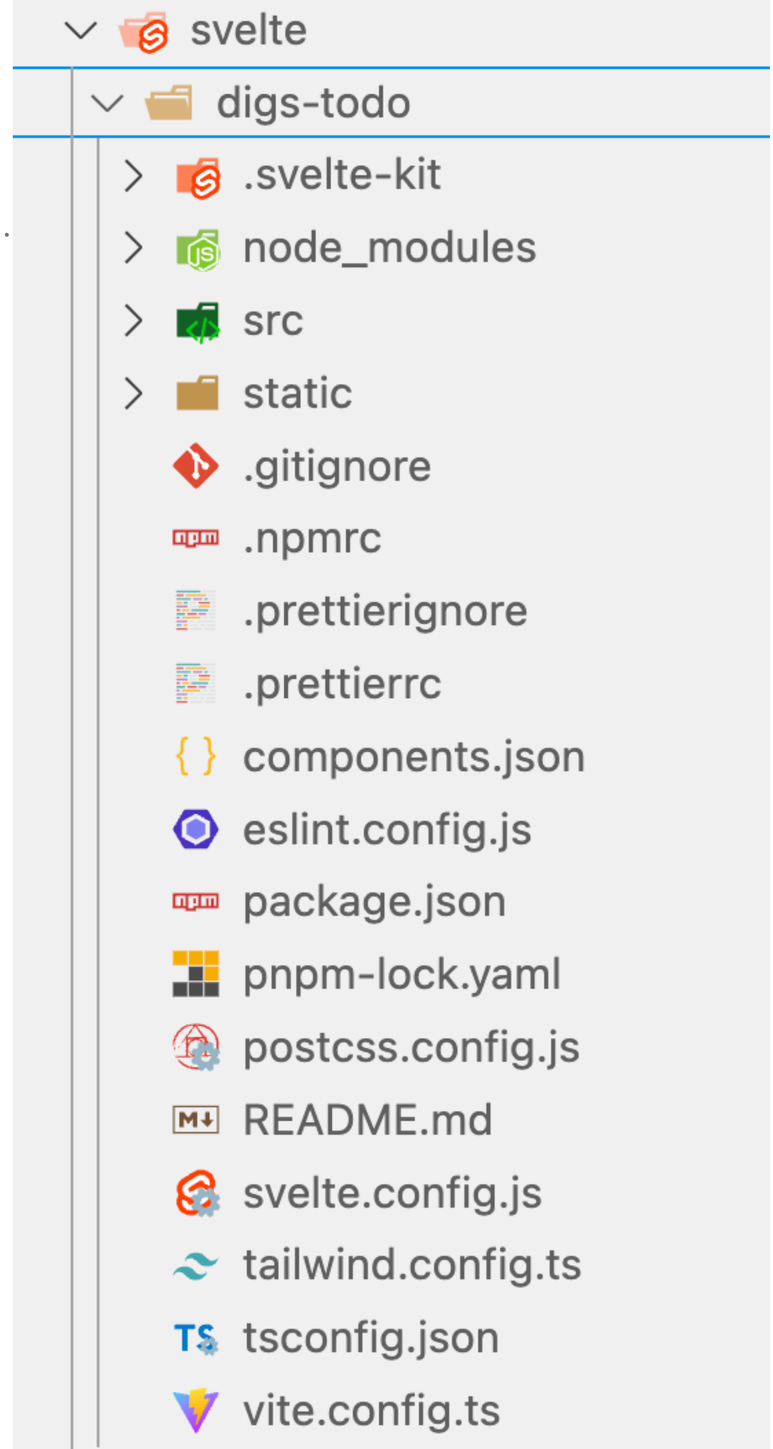

localhost
Welcome to SvelteKit
Visit svelte.dev/docs/kit to read the documentation

SVELTE AND SVELTEKIT

- Svelte
  - Write and implement UI components
    - Buttons, menus, headers, etc.
- SvelteKit
  - A framework
  - Routing
  - App development

PROJECT STRUCTURE

- src
  - lib/$lib
  - routes
    - +layout.svelte
    - +page.svelte
  - app.html
- static
  - favicon.png



+PAGE AND +LAYOUT

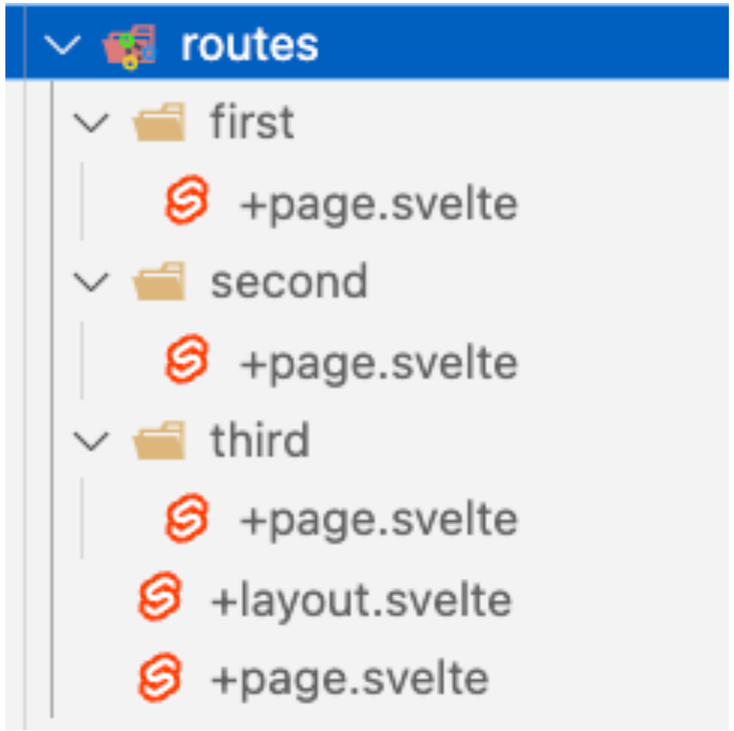
- +layout applies globally to the app
- Each +page is a separate page on the app
- Edit the +layout file

```
<script lang="ts">
  import '../app.css';
  let { children } = $props();
</script>

<main class="container mx-auto">
  {@render children()}
</main>
```

ROUTING

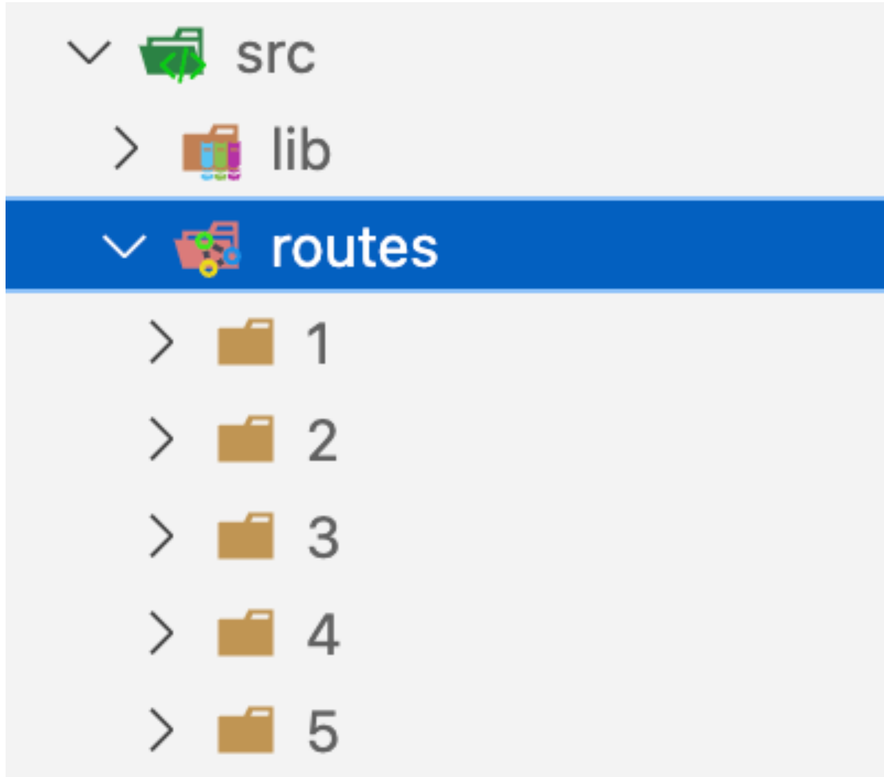
- A route is a path in your directory structure
- A route can create a new page
- Routes can be created manually or generated programmatically

  - website.com
  - website.com/first
  - website.com/second
  - website.com/third



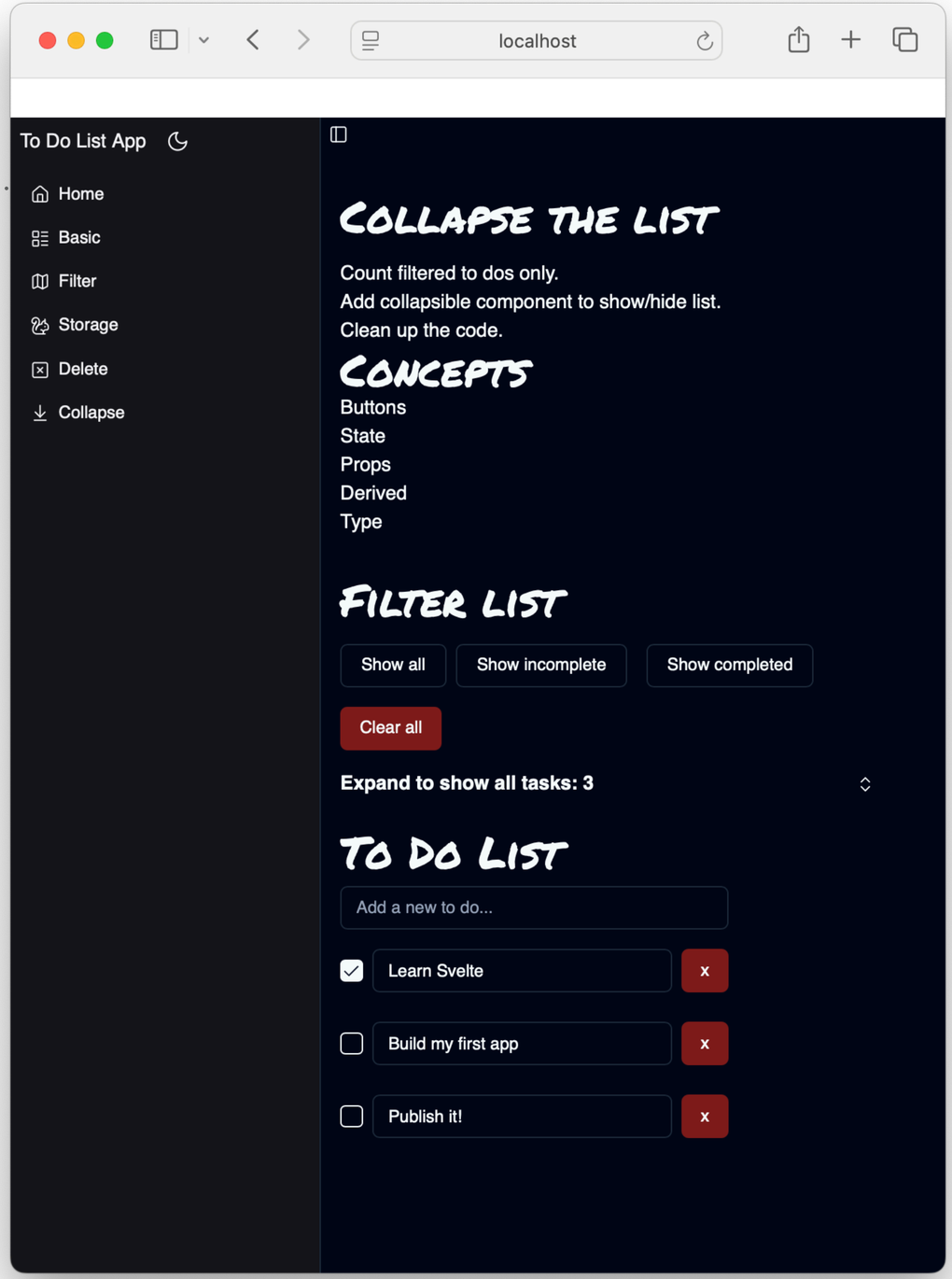
ROUTING

- Create five new routes
- We will fill them in later
- +page at the highest level is the home page
- Each route folder will have its own +page.svelte file



WHAT ARE WE GOING TO BUILD

- To Do app
- Learn the basic concepts
- Explore and experiment
- Make it your own



{#EACH}

AN ITERATION EXERCISE

- Edit src/routes/+page.svelte
- Add <script lang="ts"> section at the top
- Create a string array with at least four tasks:
- let todoItems = ['Add a sidebar.', 'Tweak the layout.', 'Create routes.', 'Add items to a To Do list.']

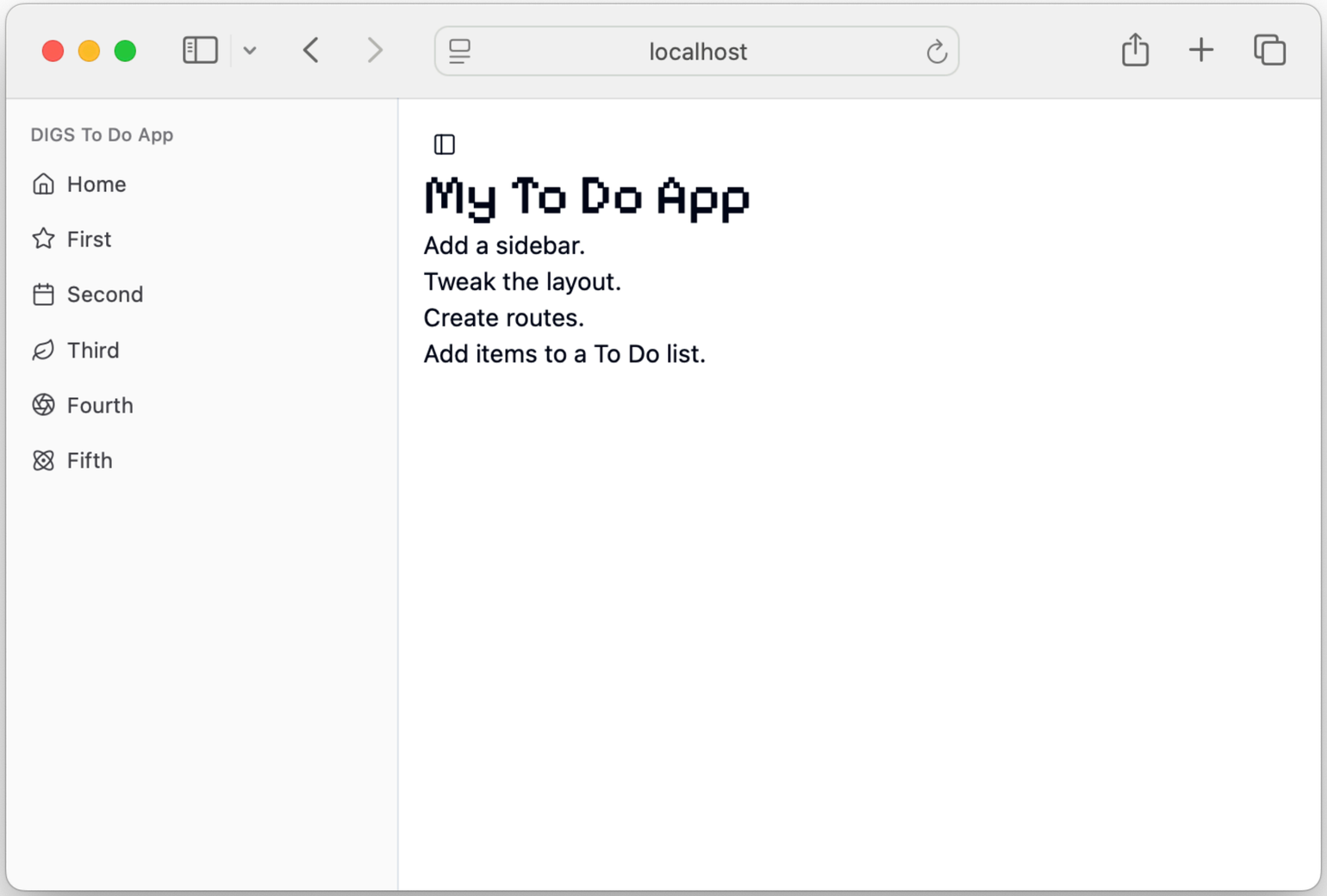
AN ITERATION EXERCISE

- Add an ordered list to your +page using
- Use {#each} to iterate over the todoItems array

```
<ol>
  {#each todoItems as item}
    <li>{item}</li>
  {/each}
</ol>
```

TO DO APP — FIRST STEP

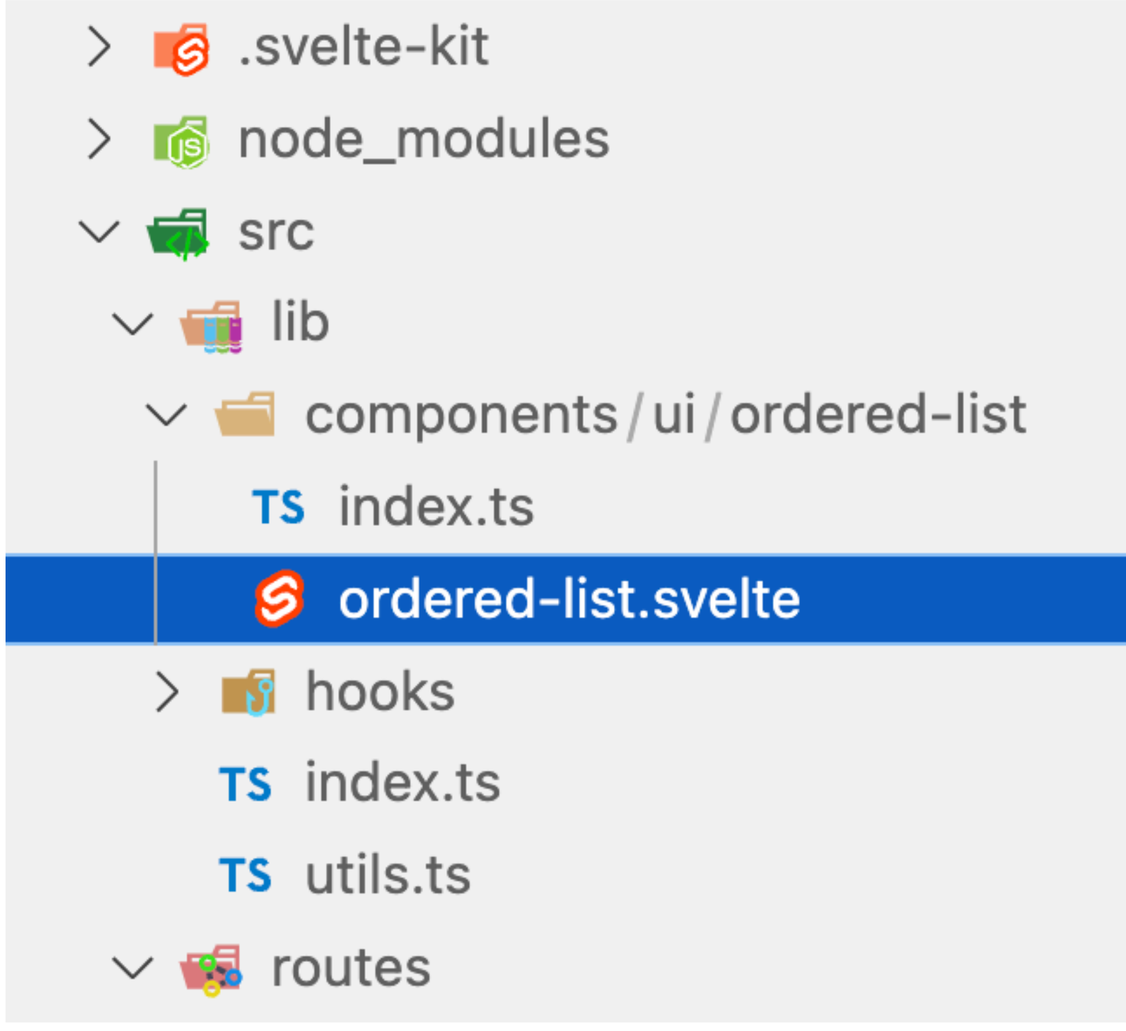
Note: screenshots show features will add later.



COMPONENTS

CREATE YOUR FIRST COMPONENT

- Convert the ordered list from an <ol> element to a component.
- Step 01
  - Go to src/lib/components/ui
  - Create a dir called ordered-list
  - Create ordered-list.svelte
  - Create index.ts



INDEX.TS

- This file exports the component to make it available to use.

```
import Root from "./ordered-list.svelte";

export {
  Root, Root as OrderedList,
};
```

CREATE THE COMPONENT — STEP ONE

- Edit the ordered-list.svelte file.
- Components have $props().

```
<script>
  let { items } = $props();
</script>

<ol>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ol>
```

USE THE COMPONENT ON SRC/ROUTES/+PAGE.SVELTE

```
<script lang="ts">
  import {OrderedList} from "$lib/components/ui/ordered-list";
  let todoItems = [
    'Add a sidebar.',
    'Tweak the layout.',
    'Create routes.',
    'Add items to a To Do list.'
    ]
</script>

<h1 class="text-4xl">My To Do App</h1>

<OrderedList items={todoItems} />
```

USE THE COMPONENT

+page.svelte ● | ordered-list.svelte

GitHub > svelte > digs-test > src > routes > +page.svelte > ...

Legacy mode

```svelte
<script lang="ts">
  import { OrderedList } from "$lib/components/ui/ordered-list";

  let todoItems = [
  'Learn Svelte',
  'Learn TypeScript',
  'Build a To Do App',
  'Profit'
  ]
</script>

<h1>DIGS To Do App</h1>
<OrderedList items={todoItems} listStyle="lower-alpha"/>
```

EXTRA FANCY — ADD LIST STYLE OPTION TO THE COMPONENT

- Edit the ordered-list component file.
- Assign the lang="ts" and define the types allowed for items and listStyle.
- Don't forget to add the list-style-type option to the <ol> style attribute.

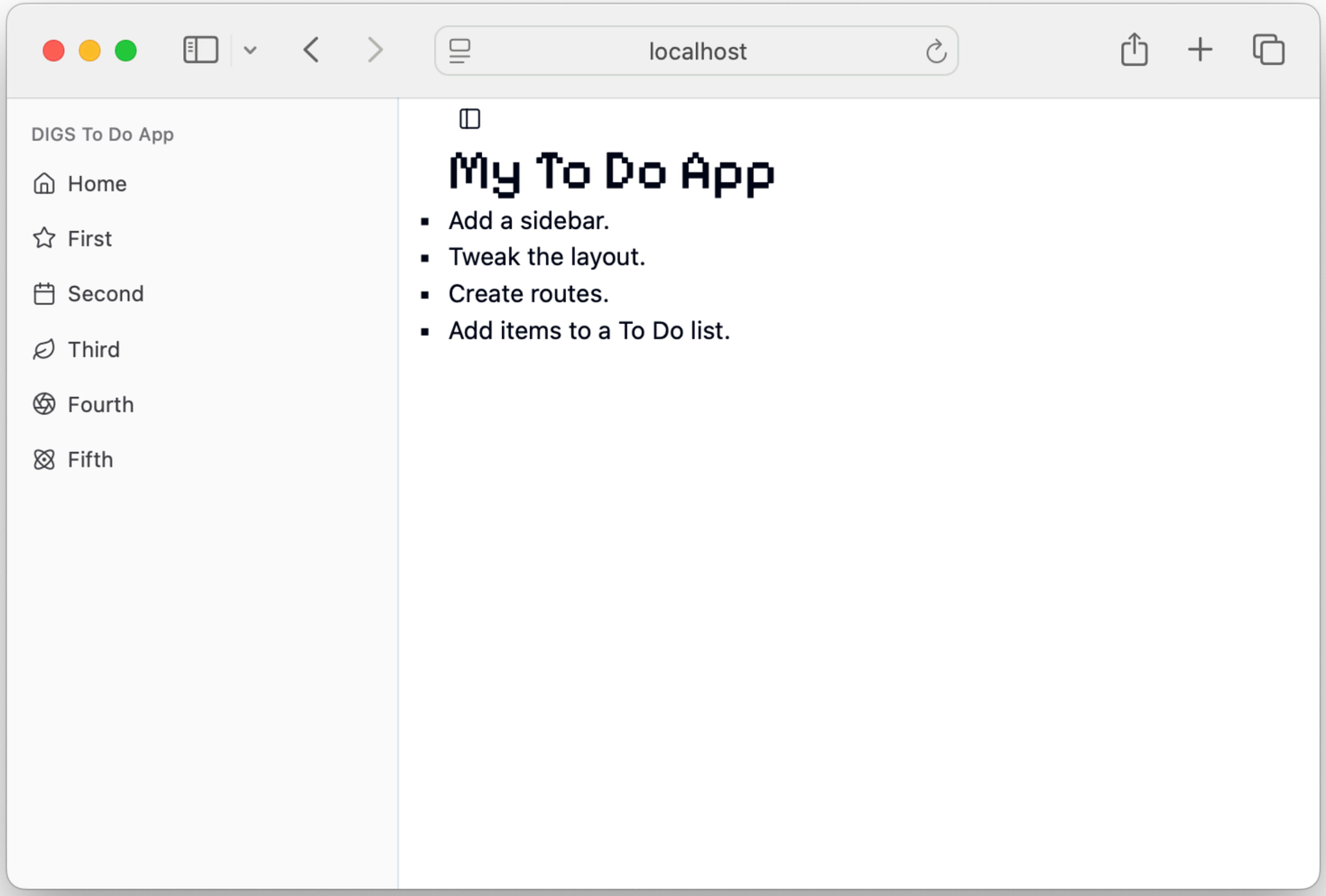
```
<script lang="ts">
  let { items, listStyle = 'decimal' }: {
    items: string[],
    listStyle?: 'disc' | 'circle' | 'square' | 'decimal' | 'lower-roman' |
'upper-roman' | 'lower-alpha' | 'upper-alpha'
    } = $props();
</script>

<ol style="list-style-type: {listStyle}">
  {#each items as item}
    <li>{item}</li>
  {/each}
</ol>
```

EXTRA FANCY — ADD LIST STYLE OPTION TO THE COMPONENT

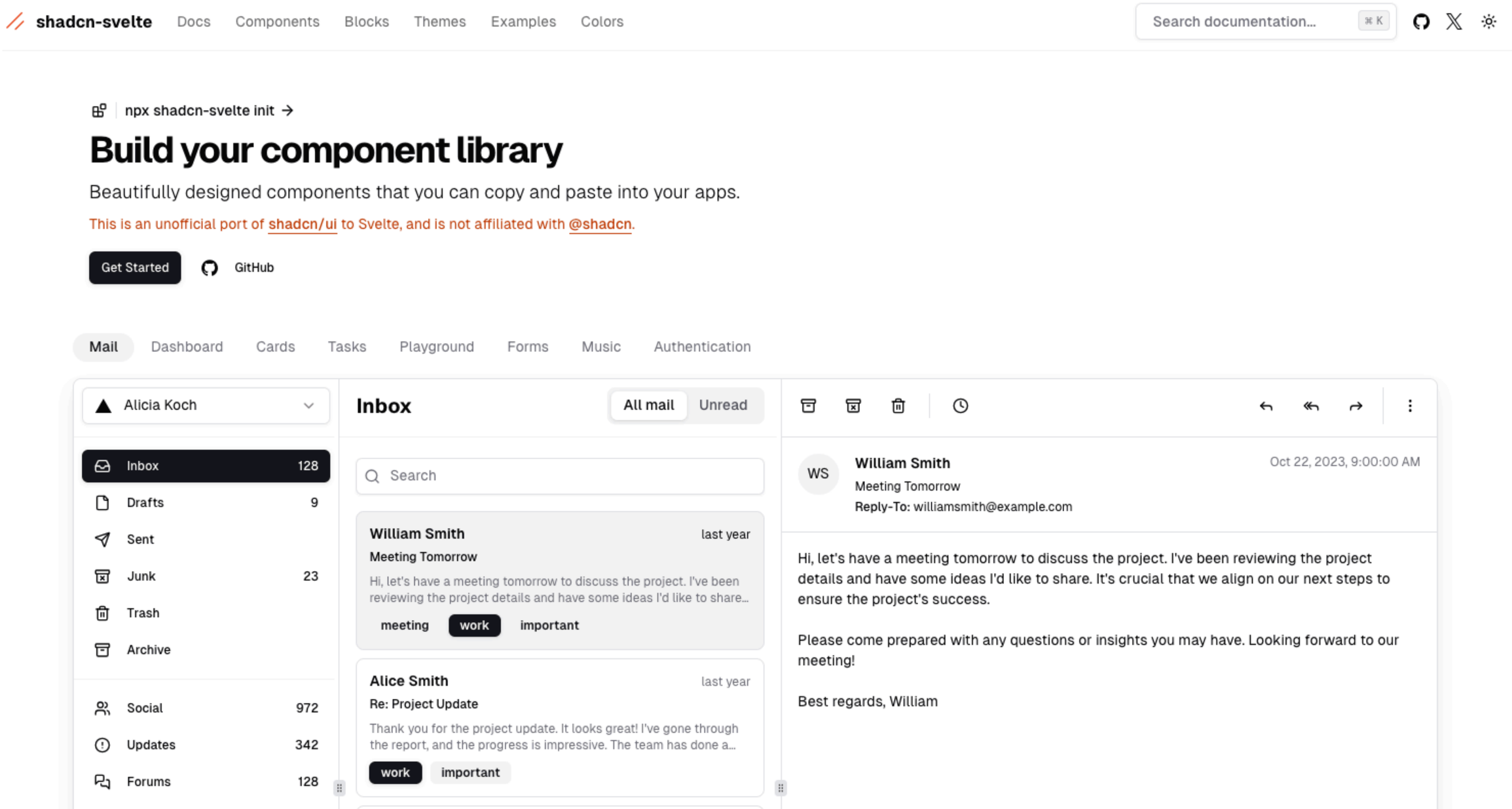
```
<script lang="ts">
  let { items, listStyle = 'decimal' }: {
    items: string[],
    listStyle?: 'disc' | 'circle' | 'square' | 'decimal' | 'lower-
roman' | 'upper-roman' | 'lower-alpha' | 'upper-alpha'
    } = $props();
</script>

<ol style="list-style-type: {listStyle}">
  {#each items as item}
    <li>{item}</li>
  {/each}
</ol>
```

EXTRA FANCY — ADD LIST STYLE OPTION

```
<OrderedList items={todoItems} listStyle="square"/>
```



https://next.shadcn-svelte.com/



Button

Displays a button or a component that looks like a button.

Docs API Reference Component Source <>

Preview Code

Style: New York

Button

Installation

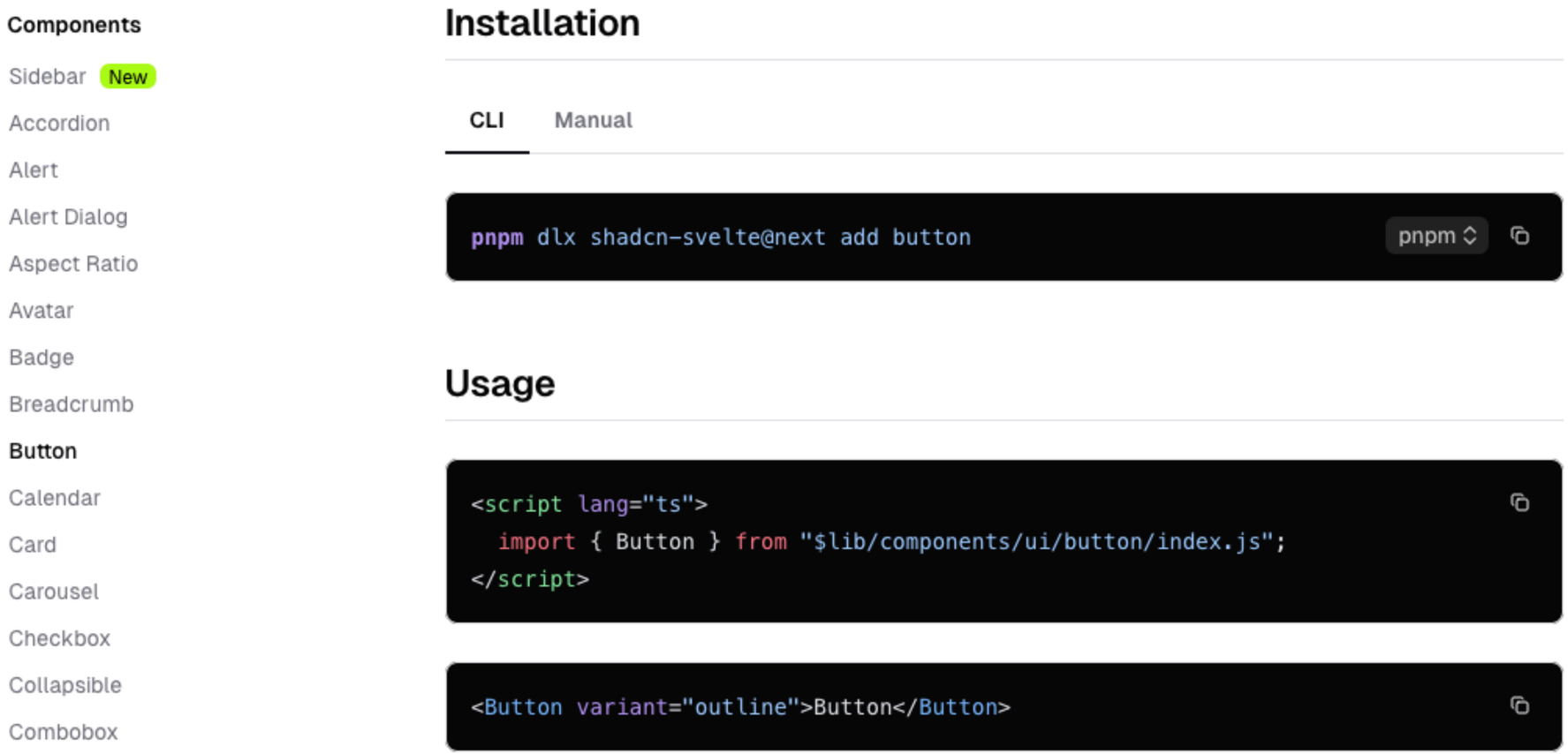
CLI Manual

```
pnpm dlx shadcn-svelte@next add button
```

pnpm

USE SHADCN-SVELTE COMPONENTS

- How to add components from Shadcn
  - Where to install in your app: in +layout or in a route +page?



BUILD THE FOUNDATION

- Add a sidebar (https://next.shadcn-svelte.com/docs/components/sidebar)
  - Install: **pnpm dlx shadcn-svelte@next add sidebar**
  - Create the following file manually:
    - src/lib/components/app-sidebar.svelte
    - See the "menu" sample in the shadcn docs

Copy this "menu" sample to the app-sidebar.svelte file

(https://next.shadcn-svelte.com/docs/components/sidebar)

Getting Started

- Introduction
- Installation
- components.json
- Theming
- Dark mode
- CLI
- Typography
- Figma
- Changelog
- Migration v5
- About

Components

- Sidebar New
- Accordion
- Alert
- Alert Dialog
- Aspect Ratio
- Avatar
- Badge
- Breadcrumb
- Button

src/lib/components/app-sidebar.svelte

```svelte
<script lang="ts">
  import Calendar from "lucide-svelte/icons/calendar";
  import House from "lucide-svelte/icons/house";
  import Inbox from "lucide-svelte/icons/inbox";
  import Search from "lucide-svelte/icons/search";
  import Settings from "lucide-svelte/icons/settings";
  import * as Sidebar from "$lib/components/ui/sidebar/index.js";

  // Menu items.
  const items = [
    {
      title: "Home",
      url: "#",
      icon: House,
    },
    {
      title: "Inbox",
      url: "#",
      icon: Inbox,
    },
    {
      title: "Calendar",
      url: "#",
      icon: Calendar,
    },
    {
```

BUILD THE FOUNDATION

- Import the component to +layout.svelte file
  - `import * as Sidebar from "$lib/components/ui/sidebar/index.js";`
  - `import AppSidebar from "$lib/components/app-sidebar.svelte";`

+layout.svelte ● app-sidebar.svelte

GitHub > svelte > digs-test > src > routes > +layout.svelte > ...

Runes mode

```svelte
<script lang="ts">
  import '../app.css';
  import * as Sidebar from "$lib/components/ui/sidebar/index.js";
  import AppSidebar from "$lib/components/app-sidebar.svelte";
  let { children } = $props();
</script>

<Sidebar.Provider>
  <AppSidebar />
  <main class="container mx-auto">
    <Sidebar.Trigger />
    {@render children?.()}
  </main>
</Sidebar.Provider>

```

BUILD THE FOUNDATION

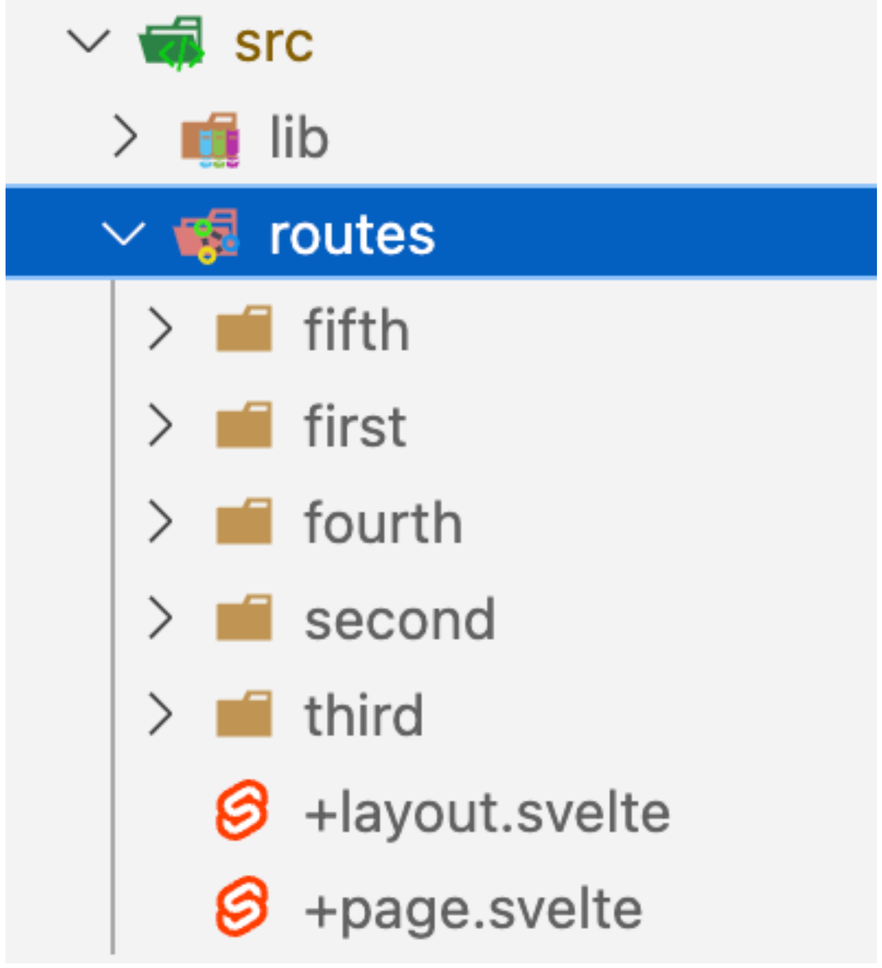
- Install the Lucide Svelte dependency
  - https://lucide.dev/guide/packages/lucide-svelte
  - **pnpm add lucide-svelte**
- In addition to Home, create and configure five additional routes
- Pick some icons to display on the sidebar

+layout.svelte | app-sidebar.svelte

GitHub > svelte > digs-test > src > lib > components > app-sidebar.svelte > script

Legacy mode

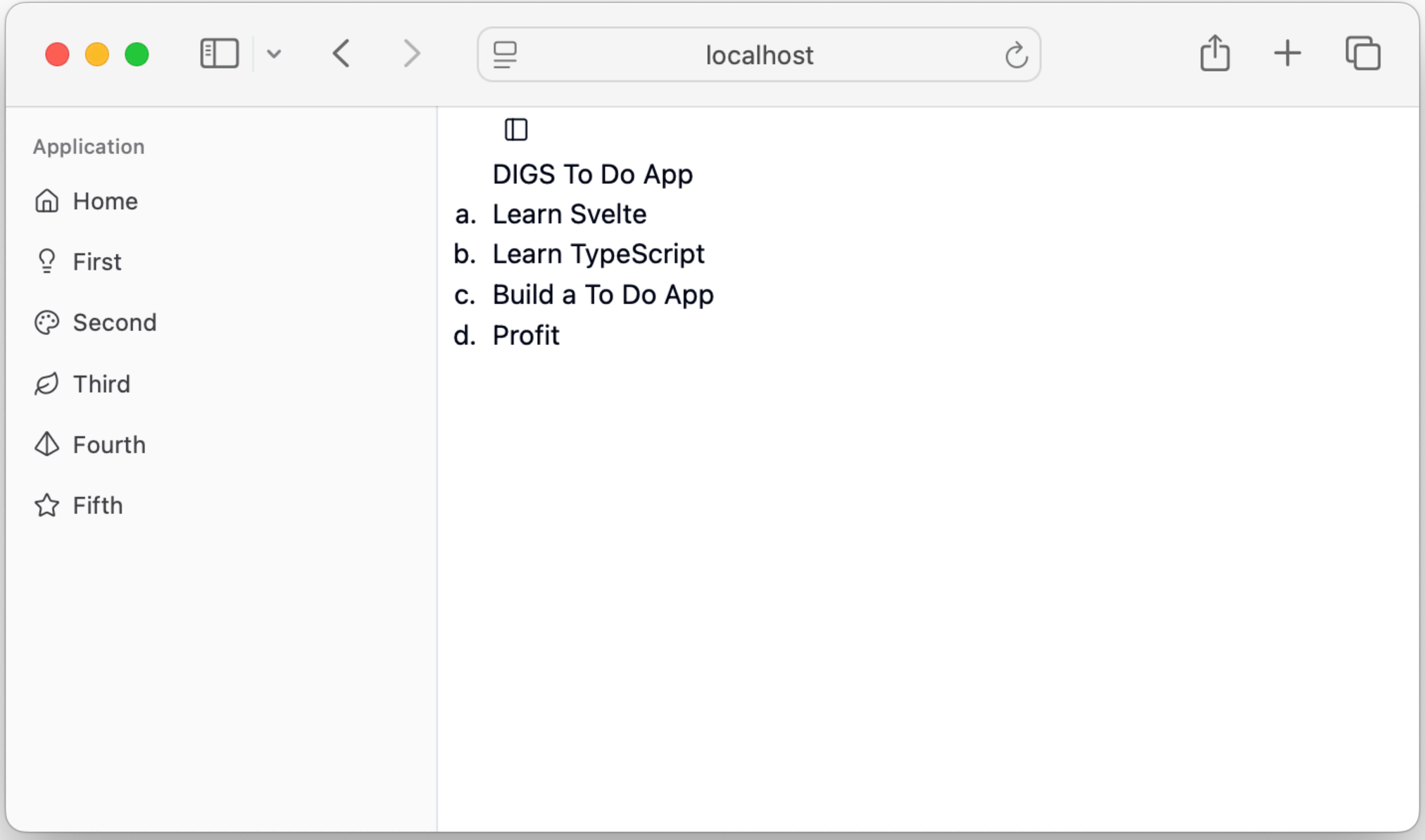
```svelte
<script lang="ts">
import { Palette, House, Lightbulb, Leaf, Pyramid, Star } from "lucide-svelte";
import * as Sidebar from "$lib/components/ui/sidebar/index.js";

// Menu items.
const items = [
  {
    title: "Home",
    url: "/",
    icon: House,
  },
  {
    title: "First",
    url: "first",
    icon: Lightbulb,
  },
  {
    title: "Second",
    url: "second",
    icon: Palette,
  },
```



+layout.svelte | app-sidebar.svelte

GitHub > svelte > dt > src > lib > components > app-sidebar.svelte > Sidebar.Root > Sidebar.Content > Sidebar.Group > Sidebar.GroupLabel

```svelte
<Sidebar.Root>
 <Sidebar.Content>
  <Sidebar.Group>
   <Sidebar.GroupLabel>DIGS To Do App</Sidebar.GroupLabel>
   <Sidebar.GroupContent>
    <Sidebar.Menu>
     {#each items as item (item.title)}
      <Sidebar.MenuItem>
       <Sidebar.MenuButton>
        {#snippet child({ props })}
         <a href={item.url} {...props}>
          <item.icon />
          <span>{item.title}</span>
         </a>
        {/snippet}
       </Sidebar.MenuButton>
      </Sidebar.MenuItem>
     {/each}
    </Sidebar.Menu>
   </Sidebar.GroupContent>
  </Sidebar.Group>
 </Sidebar.Content>
</Sidebar.Root>
```


localhost
Application
Home
First
Second
Third
Fourth
Fifth
DIGS To Do App
a. Learn Svelte
b. Learn TypeScript
c. Build a To Do App
d. Profit

WEB FONTS

BUILD THE FOUNDATION

- Add a global font from Fontsource
  - Browse: https://fontsource.org/?variable=true
  - Install docs: https://fontsource.org/docs/getting-started/variable
  - CLI: **pnpm add @fontsource-variable/geist**
  - Import to +layout
  - Configure app.css to change the font of <h1> to your new font
  - Add an <h1> to your main +page

+layout.svelte | +page.svelte | app.css 6

GitHub > svelte > digs-test > src > routes > +layout.svelte > ...

Runes mode

```svelte
<script lang="ts">
  import '../app.css';
  import * as Sidebar from "$lib/components/ui/sidebar/index.js";
  import AppSidebar from "$lib/components/app-sidebar.svelte";
  // Supports weights 100-900
  import '@fontsource-variable/geist';
  let { children } = $props();
</script>

<Sidebar.Provider>
  <AppSidebar />
  <main class="container mx-auto">
    <Sidebar.Trigger />
    {@render children?.()}
  </main>
</Sidebar.Provider>

```

+layout.svelte | +page.svelte | app.css 6

GitHub > svelte > digs-test > src > app.css > ...

```
68  @layer base {
69    * {
70      @apply border-border;
71    }
72    body {
73      @apply bg-background text-foreground;
74    }
75    h1 {
76      @apply text-2xl font-bold;
77      font-family: 'Geist Variable', sans-serif;
78    }
79  }
80
```

+layout.svelte | +page.svelte | app.css 6

GitHub > svelte > digs-test > src > routes > +page.svelte > ...

Legacy mode

```svelte
<script lang="ts">
  import { OrderedList } from "$lib/components/ui/ordered-list";

  let todoItems = [
  'Learn Svelte',
  'Learn TypeScript',
  'Build a To Do App',
  'Profit'
  ]
</script>

<h1>DIGS To Do App</h1>
<OrderedList items={todoItems} listStyle="lower-alpha"/>

```

TO DO APP

TO DO APP — FIRST STEP

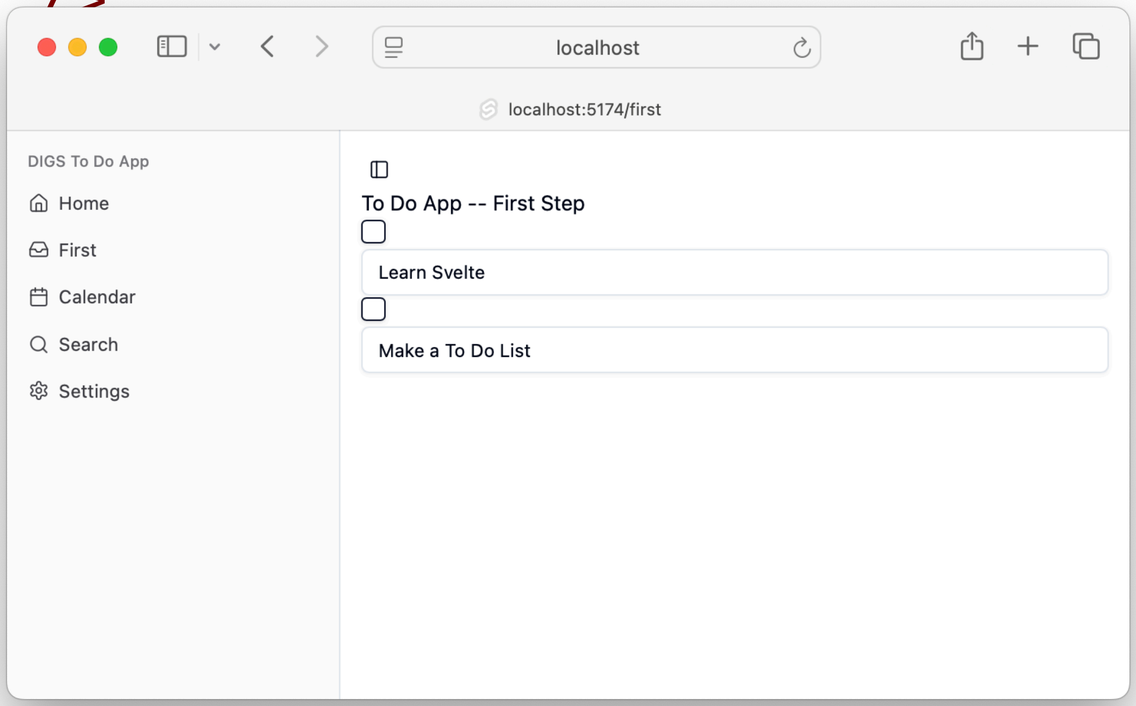
- Create **route/first/+page.svelte**
- Install and import components from Shadcn: Checkbox and Input
  - Input may already be installed as part of Sidebar
- Inside <script lang="ts">, create an array of objects to store the tasks
- Use the $state rune. What is it?

```
// Create a store to hold the todos and provide two sample todos
  let todos = $state([
  {text: 'Learn Svelte', done: true},
  {text: 'Make a To Do List', done: false},
  ])
```

TO DO APP — FIRST STEP

- Add the following to your first route +page.svelte file.
- Use {#each} to loop over the todos array.

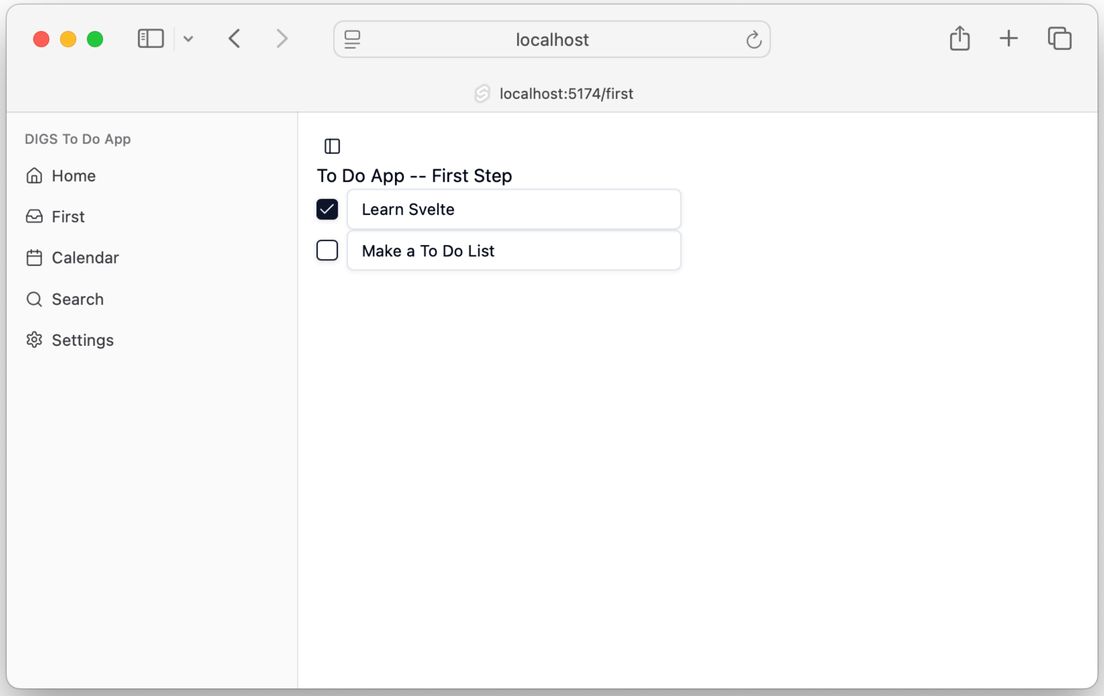
```
{#each todos as todo}
  <div>
    <Checkbox checked={todo.done} />
    <Input value={todo.text} />
  </div>
{/each}
```



TO DO APP — FIRST STEP

- Let's make the components look a little better with Tailwind classes

```
{#each todos as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox checked={todo.done} />
    <Input value={todo.text} />
  </div>
{/each}
```

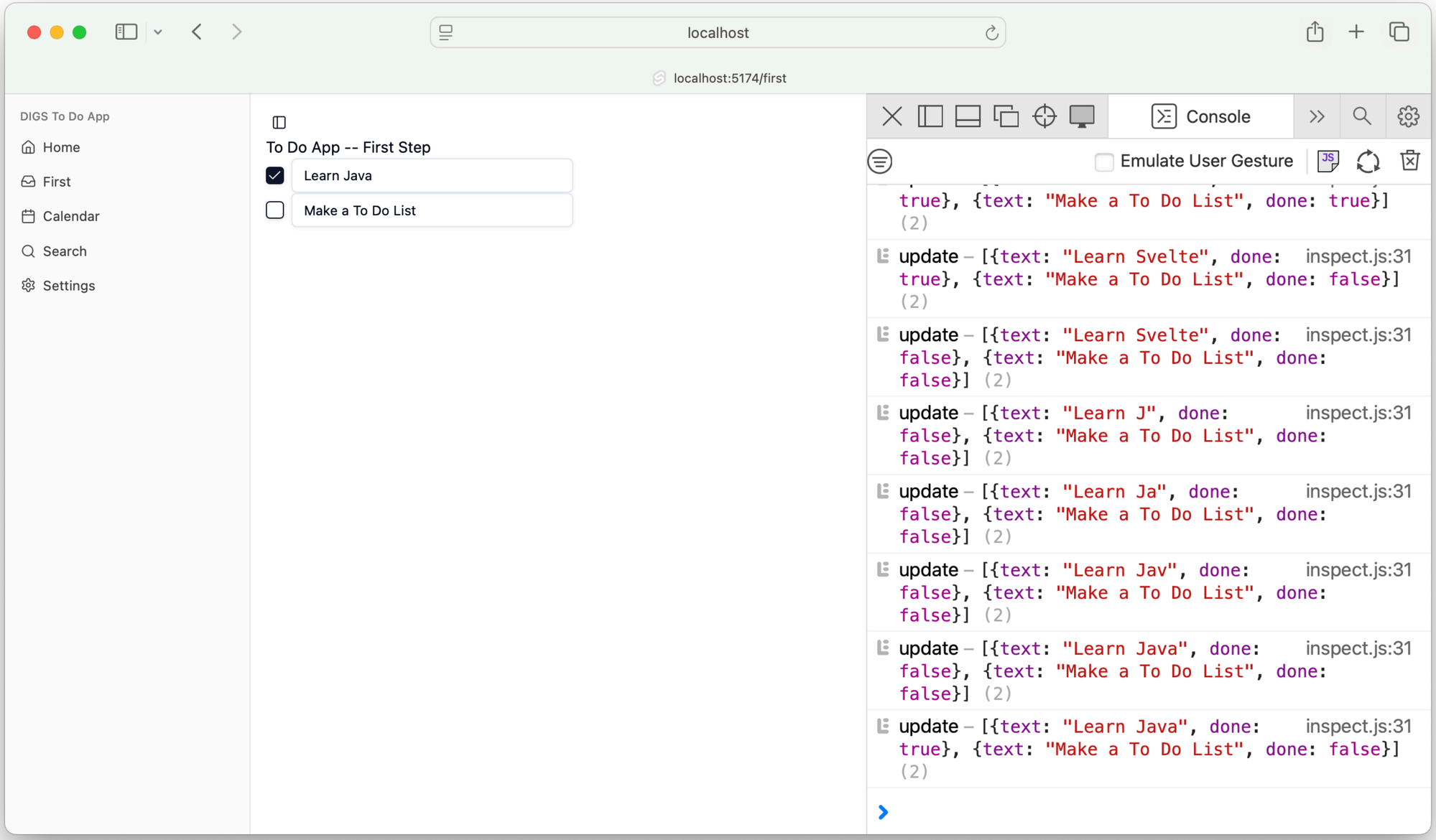


WATCH THE ARRAY CHANGE

- Add $inspect(todos) to <script>
- Use bind: to connect the components with the array bidirectionally.
  - Without bind: data can only go from parent to child (array to component)

```
{#each todos as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox bind:checked={todo.done} />
    <Input bind:value={todo.text} />
  </div>
{/each}
```

WATCH THE ARRAY CHANGE



ADD A NEW TASK

- Add a new Input component above the {#each} loop.

```
<Input placeholder="Add a new todo" />
```

- Why doesn't it do anything? We have to tell it what to do.
  - When the user hits Enter, add the value of Input as a new task in todos.
- Two steps: write the function, invoke the function.

FUNCTION ADD NEW TASK

- New function in <script> of routes/first/+page.svelte
- What should the function do?
  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.
  - Update the todos array to include this new task.
  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.

```
function addTask(event: KeyboardEvent) {}
```

  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.
  - Update the todos array to include this new task.
  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.
  - If the user hits any key other than enter, stop the function.

```
if (event.key !== 'Enter') return
```

  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.
  - Update the todos array to include this new task.
  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.
  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.

```
const input = event.currentTarget as HTMLInputElement
const text = input.value.trim()
```

  - Set the default value of the task as false.
  - Update the todos array to include this new task.
  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.
  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.

```
const done = false
```

  - Update the todos array to include this new task.
  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.
  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.
  - Update the todos array to include this new task.

```
todos = [...todos, {text, done}]
```

  - Clear out the Input box.

FUNCTION ADD NEW TASK

- Steps
  - Create the function framework.
  - If the user hits any key other than enter, stop the function.
  - When the user hits Enter, get the value of Input and trim off any extra spaces.
  - Set the default value of the task as false.
  - Update the todos array to include this new task.
  - Clear out the Input box.

```
input.value = ''
```

FUNCTION ADD NEW TASK

- The addTask function:

```
function addTask(event: KeyboardEvent) {
    // Only add a todo if the user pressed the Enter key
    if (event.key !== 'Enter') return
    // Get the input element
    const input = event.currentTarget as HTMLInputElement
    const text = input.value.trim()
    const done = false
    // Spread the props and add the new one
    todos = [...todos, {text, done}]
    // Clear the input
    input.value = ''
  }
```

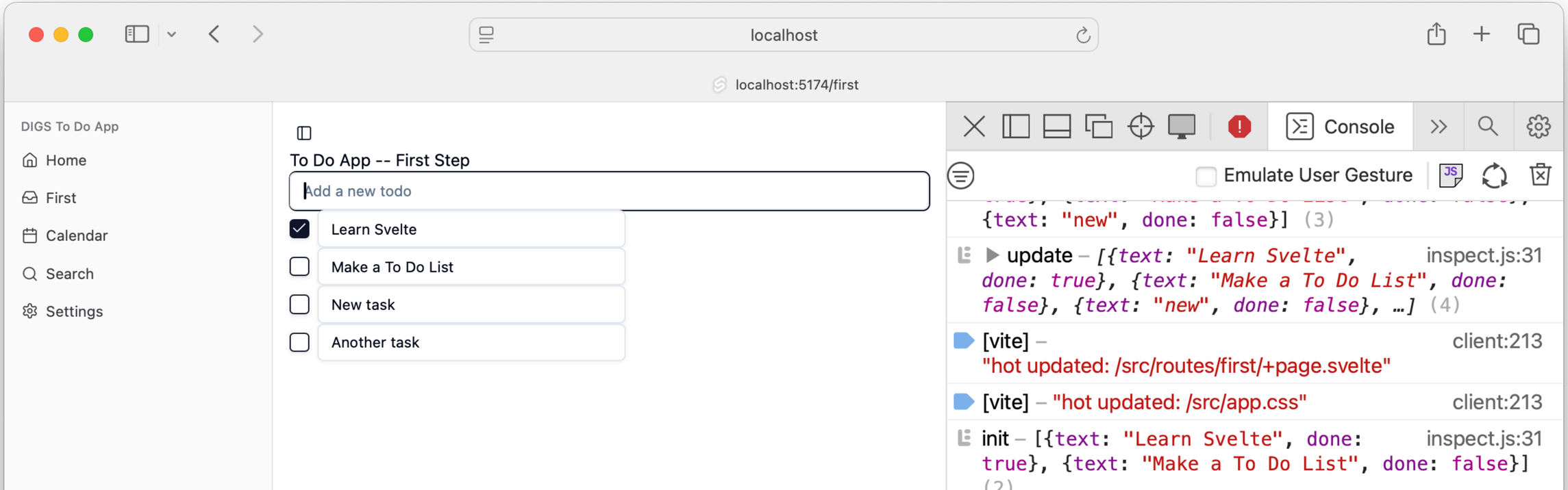
FUNCTION ADD NEW TASK

- Alternate version:

```
function addTask(event: KeyboardEvent) {
    // Only add a todo if the user pressed the Enter key
    if (event.key === 'Enter') {
      // Get the input element
      const input = event.currentTarget as HTMLInputElement
      const text = input.value.trim()
      const done = false
      // Spread the props and add the new one
      todos = [...todos, {text, done}]
      // Clear the input
      input.value = ''
    }
  }
```

INVOKE THE FUNCTION

- Update the Input component
  - `<Input onkeydown={addTask} placeholder="Add a new todo" />`
  - OR: `<Input onkeydown={(event) => addTask(event)} placeholder="Add a new todo" />`
  - Now you can add new tasks!



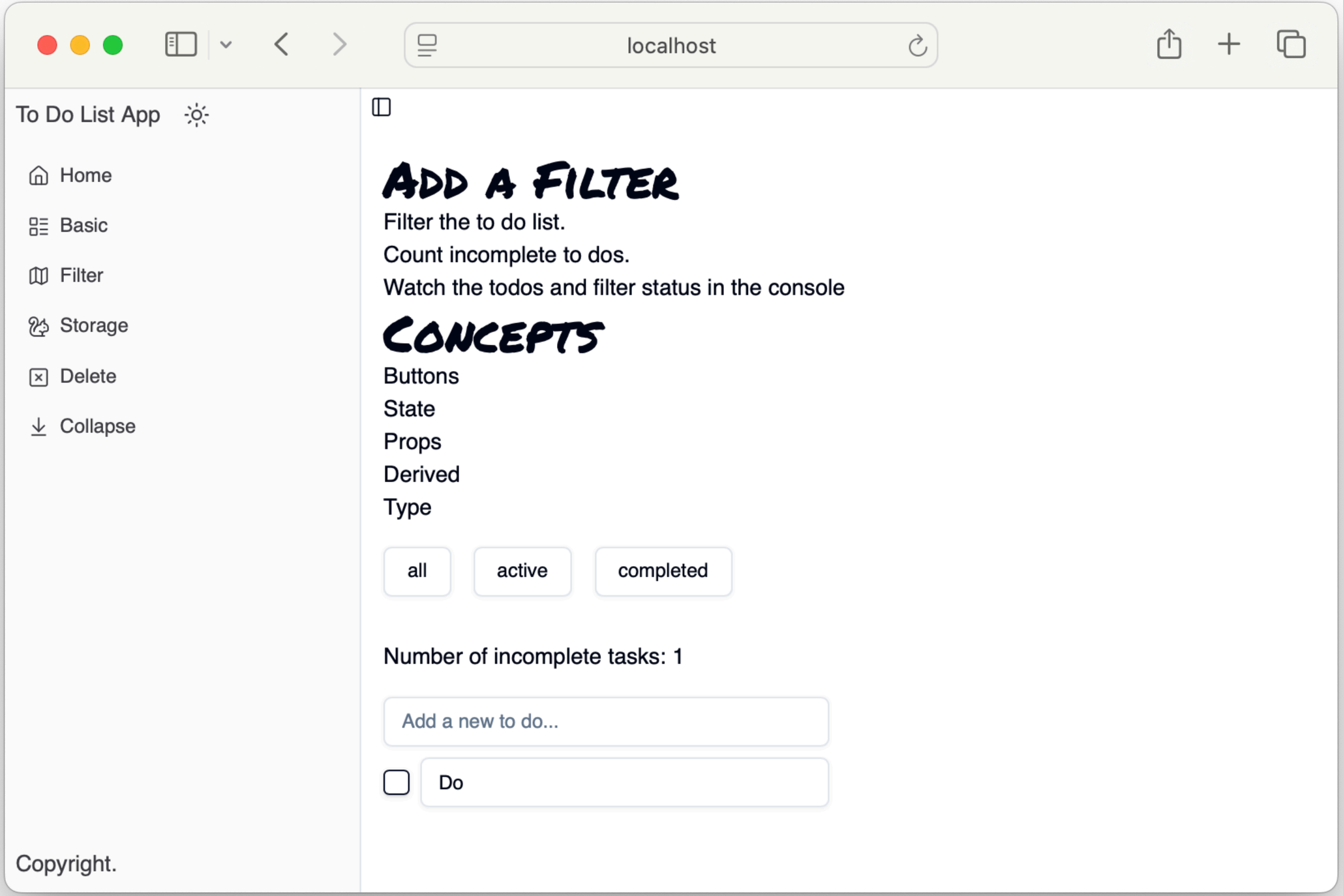
QUICK REVIEW

- Quick concept review:
  - Understand how the **todos** array is getting updated by passing state.
  - Understand what …**todos** does in the function.
    - The spread syntax enumerates the properties of todos and reassigns the key-value pairs to the object.
  - Can you edit a task? Why or why not?

STEP TWO

PREPARING TO MOVE ON

- Create the rest of your routes and update your sidebar if you haven't already.
- Copy the +page.svelte file from the First route to the Second route.
- In step two
  - Count the number of incomplete tasks.
  - Filter the task list based on completion status.


localhost
To Do List App
Home
Basic
Filter
Storage
Delete
Collapse
Copyright.
Add a Filter
Filter the to do list.
Count incomplete to dos.
Watch the todos and filter status in the console
Concepts
Buttons
State
Props
Derived
Type
all
active
completed
Number of incomplete tasks: 1
Add a new to do...
Do

ADD THE BUTTONS

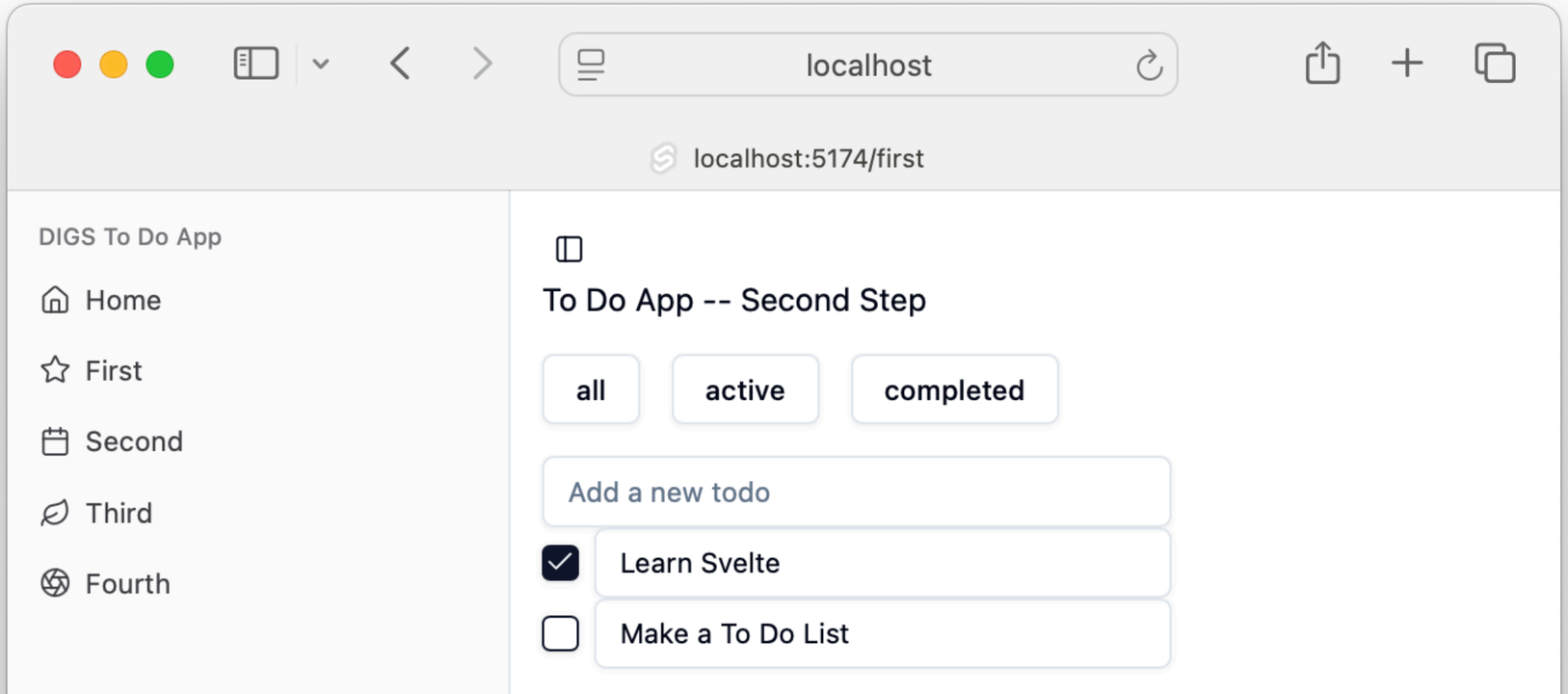
- Install and import the Button component (it might already be there from Sidebar)
  - Note: I couldn't style buttons until I reinstalled!
- We want three buttons:
  - Show all tasks
  - Show active tasks
  - Show completed tasks
- How should we add three buttons?

```
<p>To Do App -- Second Step</p>
<div class="filters space-x-4 py-4">
  <Button variant="outline" class="max-w-s ">Button</Button>
</div>
```

ADD THE BUTTONS

- Let's use {#each} to create 3 buttons.

```
<p>To Do App -- Second Step</p>
<div class="filters space-x-4 py-4">
  {#each ['all', 'active', 'completed'] as f}
    <Button variant="outline">{f}</Button>
  {/each}
</div>
```



WHAT DO THE BUTTONS DO?

- We need to write two functions:
  - One **setFilter()**
    - Set the filter status based on which button was clicked.
    - The buttons will invoke this function.
  - Two **filteredList()**
    - Create a list of tasks based on which button was clicked.
    - If 'active' is clicked, then the list become only tasks where todo.done=false.
    - We will use this list to create the task list.

BUT FIRST...

- Edit the first/+page.svelte file.
- Create a variable array to define the types of filters.

```
const filterVals = ['all', 'active', 'completed']
```

- Update the {#each} loop to use this variable.

```
<div class="filters py-4 space-x-4">
  {#each filterVals as f}
  <Button variant="outline">{f}</Button>
  {/each}
</div>
```

BUT FIRST…

- Specify Types for our stateful variables.
- Create type declarations in <script>

```
const filterVals = ['all', 'active', 'completed']
type Filters = (typeof filterVals)[number];
type Todo = {text: string; done: boolean};
```

BUT FIRST...

- Delete the previous let todos statement.
- Use types in these new statements.

```
// Create a store to hold the todos and provide two sample todos
   let todos = $state<Todo[]>([])
   let filter = $state<Filters>('all')
```

UPDATE THE BUTTON TEMPLATE TO INVOKE THE FIRST FUNCTION

- When the user clicks the button, invoke the function and pass the filter status.

```
{#each ['all', 'active', 'completed'] as f}
    <Button variant="outline" onclick={() => setFilter(f as Filters)}>{f}</Button>
{/each}
```

- Break down the onclick event
- () =>This is an anonymous function, "run what comes next"
- setFilter() We will write this function.
- (f as Filters) We pass a parameter to the function.
  - f will be 'all' on the first time through
  - The first button would invoke setFilter('all')

WRITE THE FUNCTION TO SET THE FILTER

- Defining a function: function setFilter(newFilter: Filters){…}
  - Notice the Type
- Invoking a function: setFilter(f as Filters)

```
function setFilter(newFilter: Filters){

  }
```

WRITE THE FUNCTION TO SET THE FILTER

- Set the filter to be the value passed to the function from the button.
- Why? We will use this to create a filtered list to display on the screen.

```
// Set the filter based on the button clicked
function setFilter(newFilter: Filters){
  filter = newFilter
}
```

WRITE A FUNCTION TO CREATE FILTERED LISTS

- The framework:

```
function filteredList(){

 //Three possibilities: all, active, completed...

}
```

WRITE A FUNCTION TO CREATE FILTERED LISTS

- Option #1, ternary operation

```
function filteredList(){
    return filter === 'all' ? todos :
           filter === 'active' ? todos.filter(todo => !todo.done) :
           todos.filter(todo => todo.done)
  }
```

WRITE A FUNCTION TO CREATE FILTERED LISTS

- Option #2, switch

```
function filteredList(){
   switch(filter){
     case 'all':
       return todos
     case 'active':
       return todos.filter(todo => !todo.done)
     case 'completed':
       return todos.filter(todo => todo.done)
   }
}
```

WHY DOESN'T THE LIST UPDATE?

- The list is currently showing the full todos array.
- We need it to show the output of the filteredList() function.
- Update this:

```
{#each todos as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox bind:checked={todo.done} />
    <Input bind:value={todo.text} />
  </div>
{/each}
```

WHY DOESN'T THE LIST UPDATE?

- The list is currently showing the full todos array.
- We need it to show the output of the filteredList() function.
- Update to this:

```
{#each filteredList() as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox bind:checked={todo.done} />
    <Input bind:value={todo.text} />
  </div>
{/each}
```

LIST CAN BE FILTERED NOW

- Add some tasks.
- Mark a task as completed.
- Click the buttons to show the different lists.
- What is happening.
  - Not really filtering.
  - Regenerating the list with each click.

- Next, let's count how many tasks are incomplete.

ADD A <DIV> FOR THE SUMMARY STATEMENT

- We will need to write a function to count the incomplete tasks.
- But here is the

```
<div class="py-4">Number of incomplete tasks: {taskCount()}</div>
```

THE FUNCTION

- The function will evaluate the version of todos output from filteredList().
- The list is an array, so we can use the .length method to count the items.
- We will return this count back to the .
- We will leverage the .done property of the todo.
- In English: filter the array for todos not done. If not done tasks is zero…

```
function taskCount(){
  return todos.filter(todo => !todo.done).length === 0 ? 'All clear!' :
         todos.filter(todo => !todo.done).length;
}
```

ADD TASKS, SEE COUNT

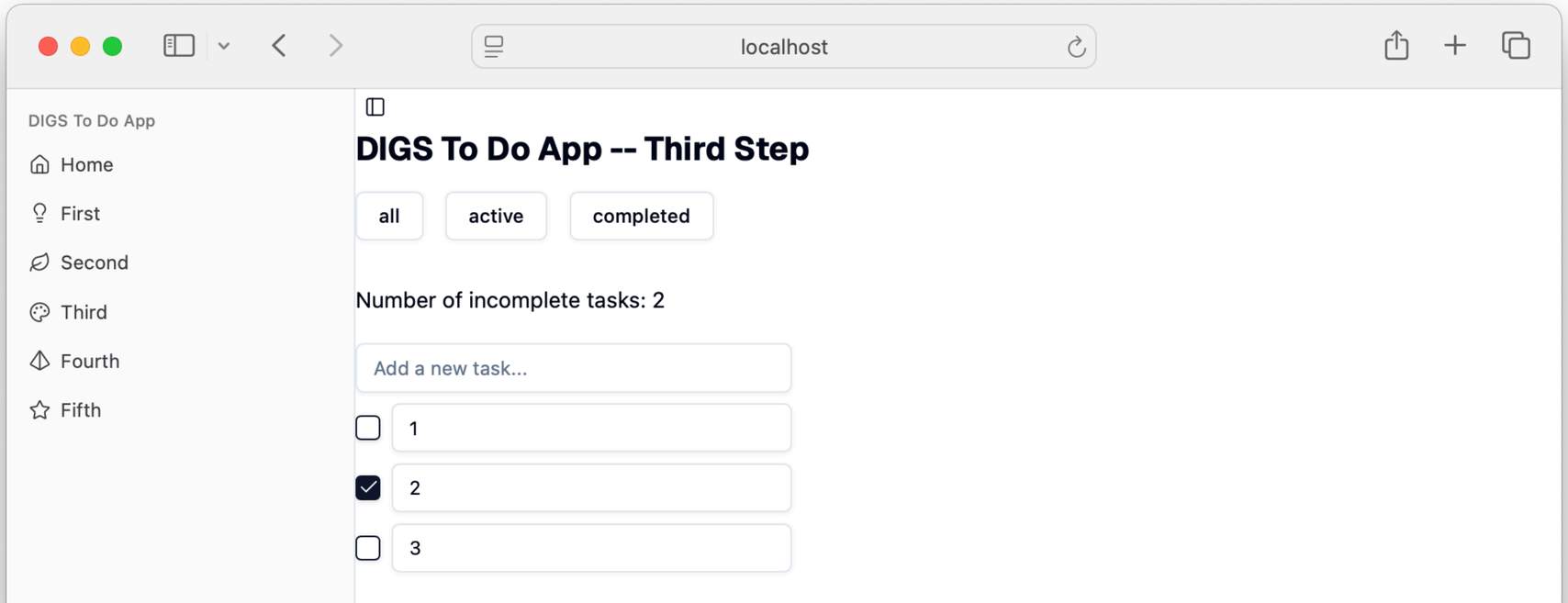
- Now your task list will count incomplete tasks.
- Add some tasks.
- Mark some complete.
- Filter with buttons

- Done with the second version of the app!
- Next, let's make it possible for the app to remember the task list.

LOCAL STORAGE

PART THREE, STORAGE

- Copy +page.svelte from the second to the third route
- Two features covered in this section:
  - We will check the browser local storage for saved todos and load them.
  - When the list changes, we will update the local storage.



PART THREE, STORAGE

- Checking the browser local storage for saved todos and loading them.
- Add an $effect rune to the <script> section of +page.
- Ternary operation (it works, but probably not the best strategy)

```
// Load the todos from local storage
  $effect(() => {
    const savedTodos = localStorage.getItem('todos')
    savedTodos ? todos = JSON.parse(savedTodos) : todos = []
  })
```

PART THREE. STORAGE

- First, check the browser local storage for saved todos and load them.
- The $effect rune, limited usage
- Try the logical AND operator: &&
- If savedTodos is truthy, then run the next statement (no else statement)

```
// Load the todos from local storage
  $effect(() => {
    const savedTodos = localStorage.getItem('todos')
    savedTodos && (todos = JSON.parse(savedTodos))
  })
```

PART THREE. STORAGE

- Second, update the local storage when todos changes.
- Local storage wants JSON.
- We need to convert todos to and from string.

```
// Local storage of todos
  $effect(() => {
    localStorage.setItem('todos', JSON.stringify(todos))
  })
```

PART THREE, STORAGE

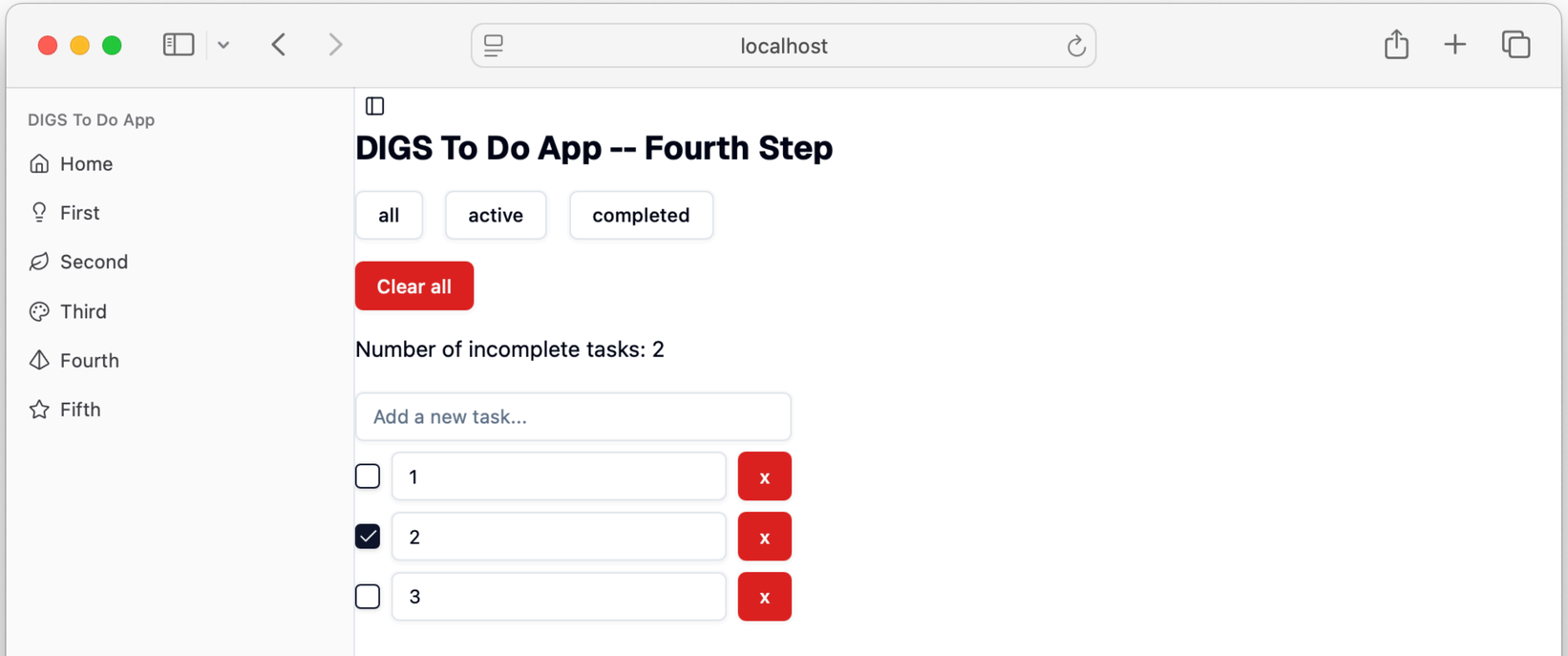
- Review
- Using the browser's local storage to .getItem and .setItem

```
// Load the todos from local storage
  $effect(() => {
    const savedTodos = localStorage.getItem('todos')
    savedTodos && (todos = JSON.parse(savedTodos))
  })

  // Local storage of todos
  $effect(() => {
    localStorage.setItem('todos', JSON.stringify(todos))
  })
```

DELETE TASKS

PART FOUR — DELETE TASKS

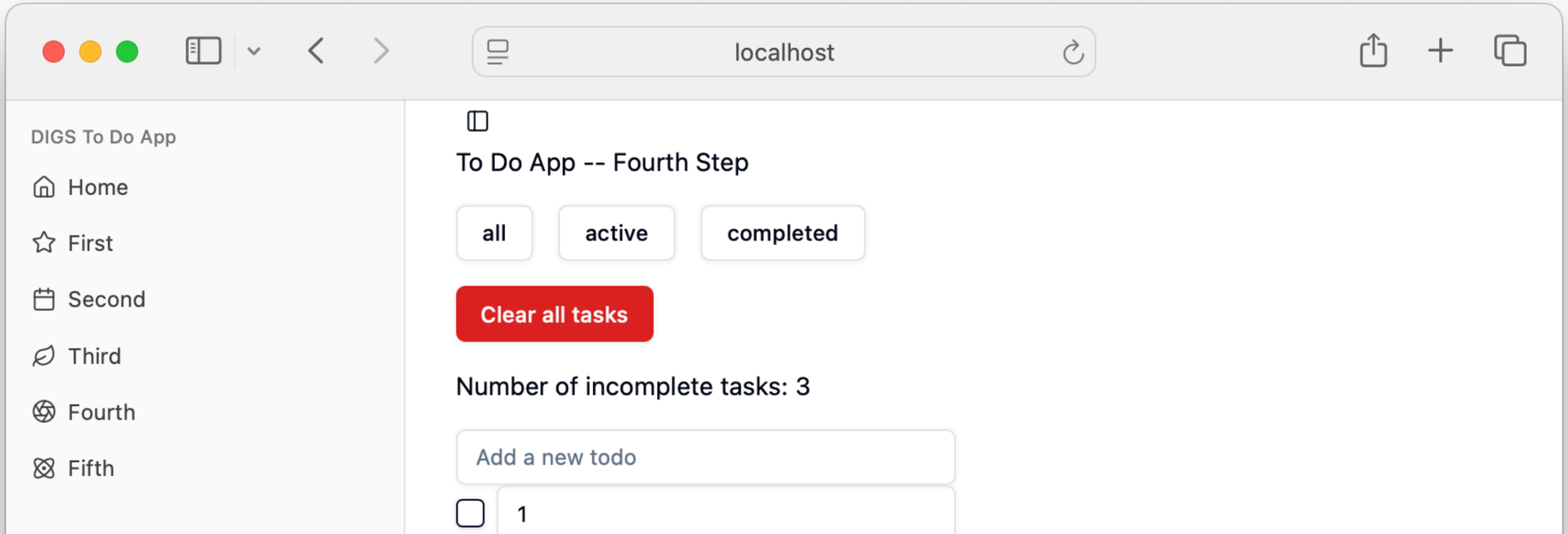
- Copy your +page to the Fourth route.
- Objective
  - Create a button that clears the entire list
  - Use buttons to delete individual tasks



FIRST STEPS, STRUCTURE

- Place the buttons on the page.
- No new functions are needed!

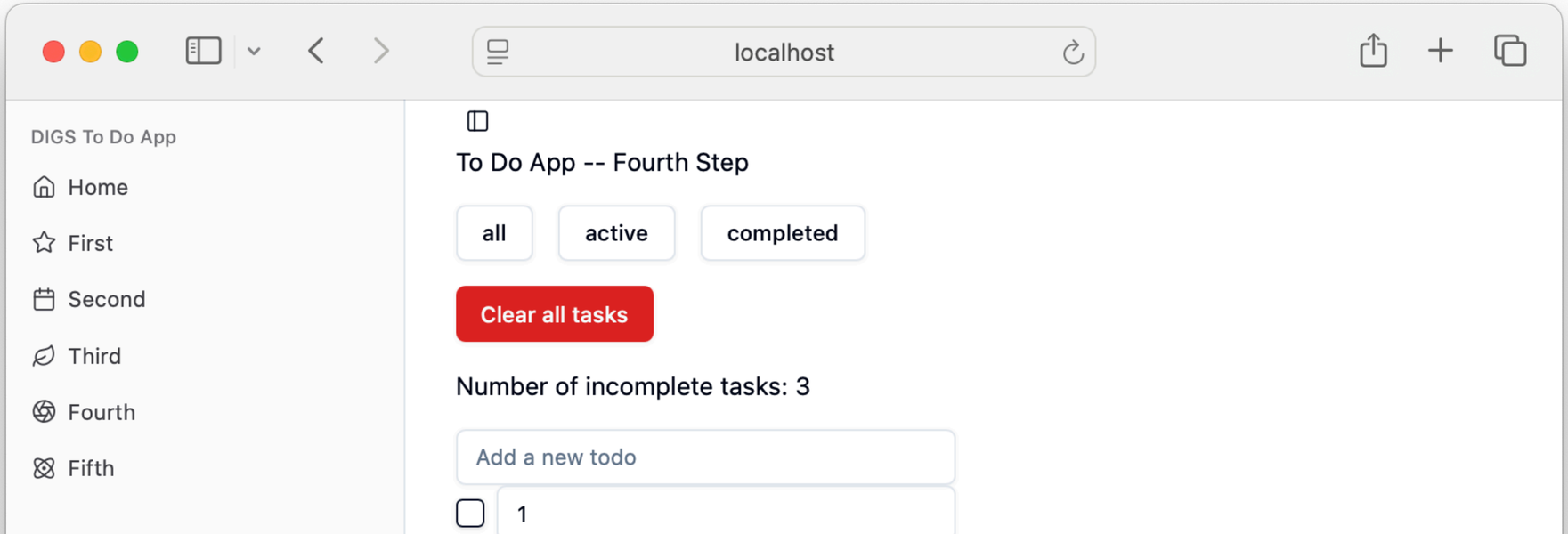
```
<div>
  <Button variant="destructive" onclick={WHAT TO DO HERE?}>Clear all tasks</Button>
</div>
<div class="py-4">Number of incomplete tasks: {taskCount()}</div>
```



FIRST STEPS, STRUCTURE

- Place the buttons on the page.
- No new functions are needed!

```
<div>
  <Button variant="destructive" onclick={() => todos = []}>Clear all tasks</Button>
</div>
<div class="py-4">Number of incomplete tasks: {taskCount()}</div>
```



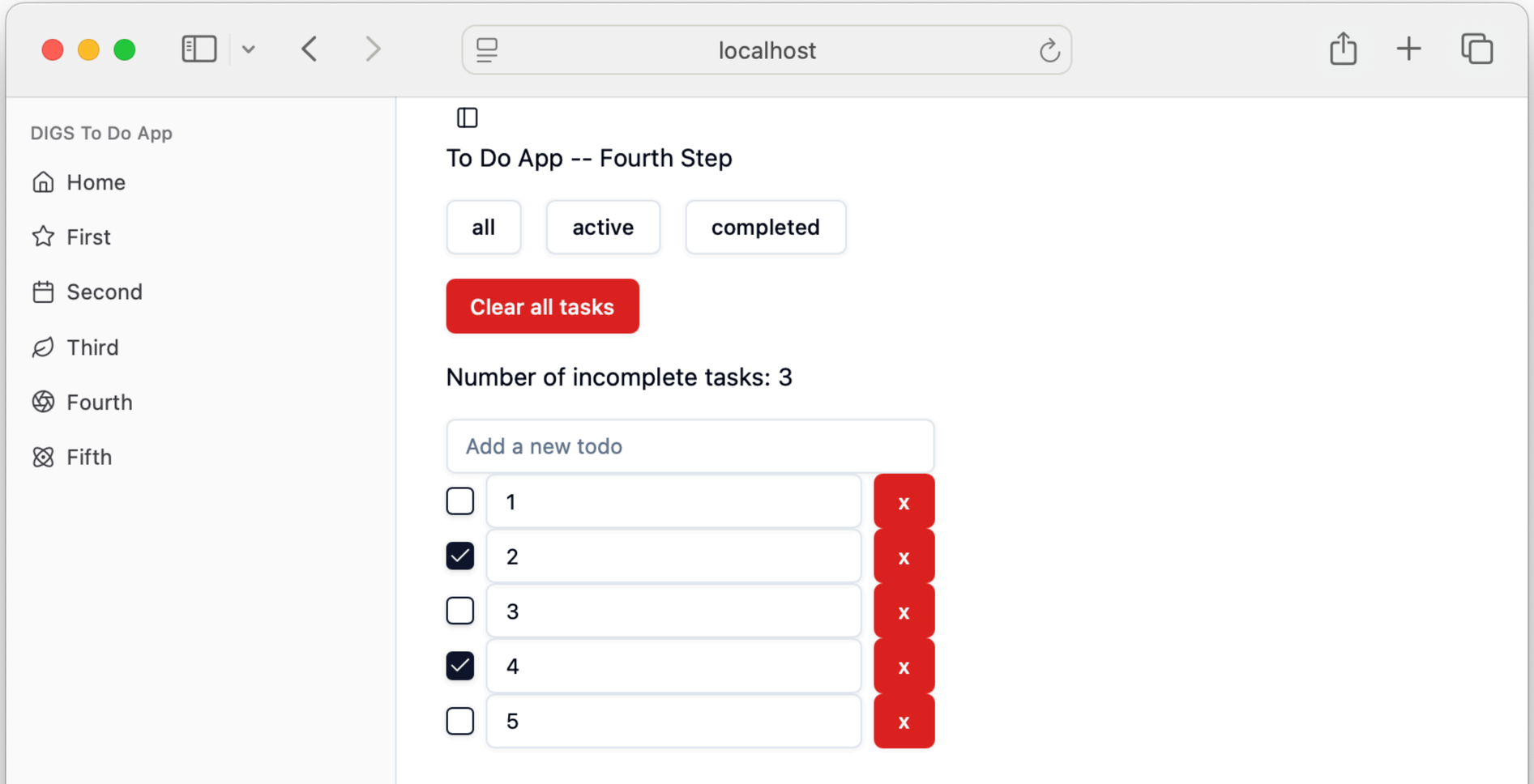
CLEAR THE ENTIRE LIST

- This button should reset the todos array to blank/empty.
- Using an anonymous inline function:
  - () =>
  - todos = []

```
<div>
    <Button variant="destructive" onclick={() => todos = []}>Clear all tasks</Button>
</div
```

DELETE A SINGLE TASK

- How did I create these little 'x' buttons?



DELETE A SINGLE TASK

- How did I create these little 'x' buttons?

```
{#each filteredList() as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox bind:checked={todo.done} />
    <Input bind:value={todo.text} />
    <Button variant="destructive" class="max-w-s " onclick={}>x</Button>
  </div>
{/each}
```

DELETE A SINGLE TASK

- What should each button's onclick event do?

```
{#each filteredList() as todo}
  <div class="flex items-center space-x-2 max-w-xs">
    <Checkbox bind:checked={todo.done} />
    <Input bind:value={todo.text} />
    <Button variant="destructive" class="max-w-s " onclick={}>x</Button>
  </div>
{/each}
```

DELETE A SINGLE TASK

- What should each button's onclick event do?
- Onclick event
  - {() =>} is an anonymous function
  - todos = we are going to reassign the todo list to exclude the current task
  - todos.filter() we are going to filter the todos to exclude the current task
  - (todo, index) => index !== i) filter out the current index

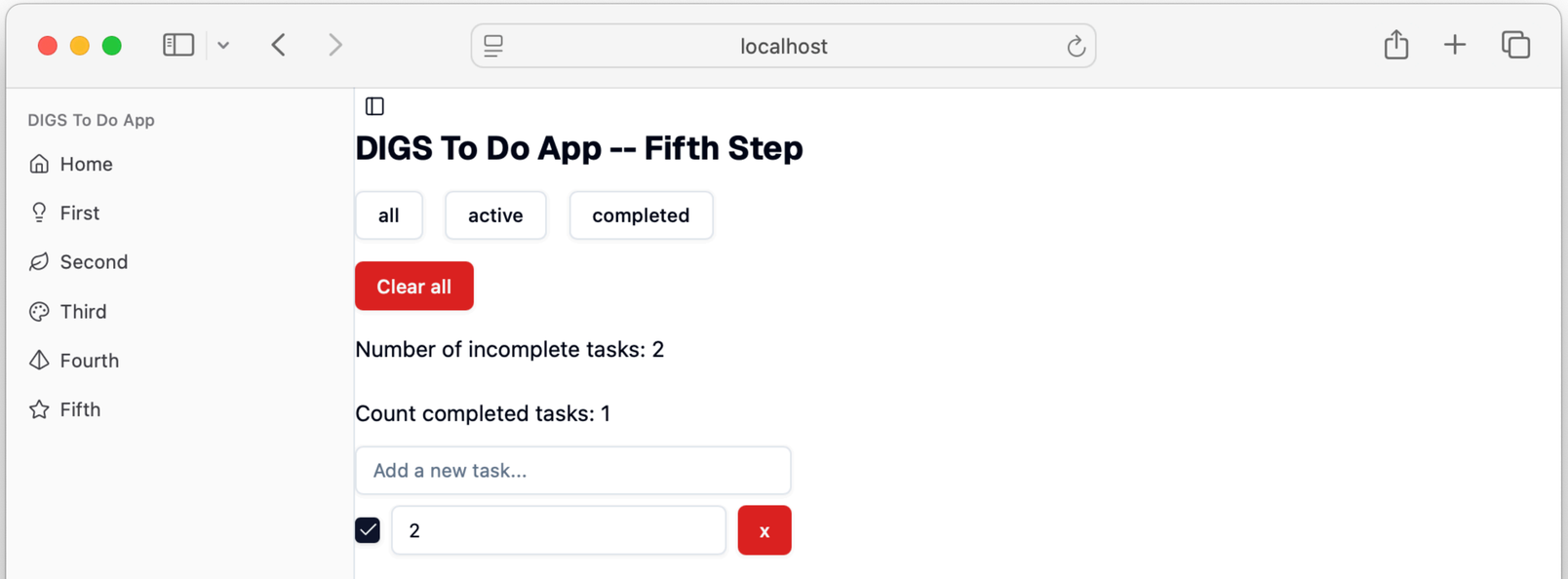
DELETE A SINGLE TASK

- Onclick event
- onclick={() => todos = todos.filter((todo, index) => index !== i)}
- OR Better: onclick={() => todos = todos.filter((_, index) => index !== i)}
  - Underscore because we don't need the todo item at this point, only the index.
- Narrative: when the button is clicked, reassign the value of the todos array to equal the todos array when it is filtered to exclude the current item.
- BUT! To make this work, we need to add the i value to the {#each} loop.
- {#each filteredList() as todo, i}

COUNT BY FILTER

FINAL STEP

- Copy +page to the Fifth route
- Objectives
  - Instead of counting incomplete tasks, count based on filter.
  - Make final clean up and layout choices.
  - Add dark mode?

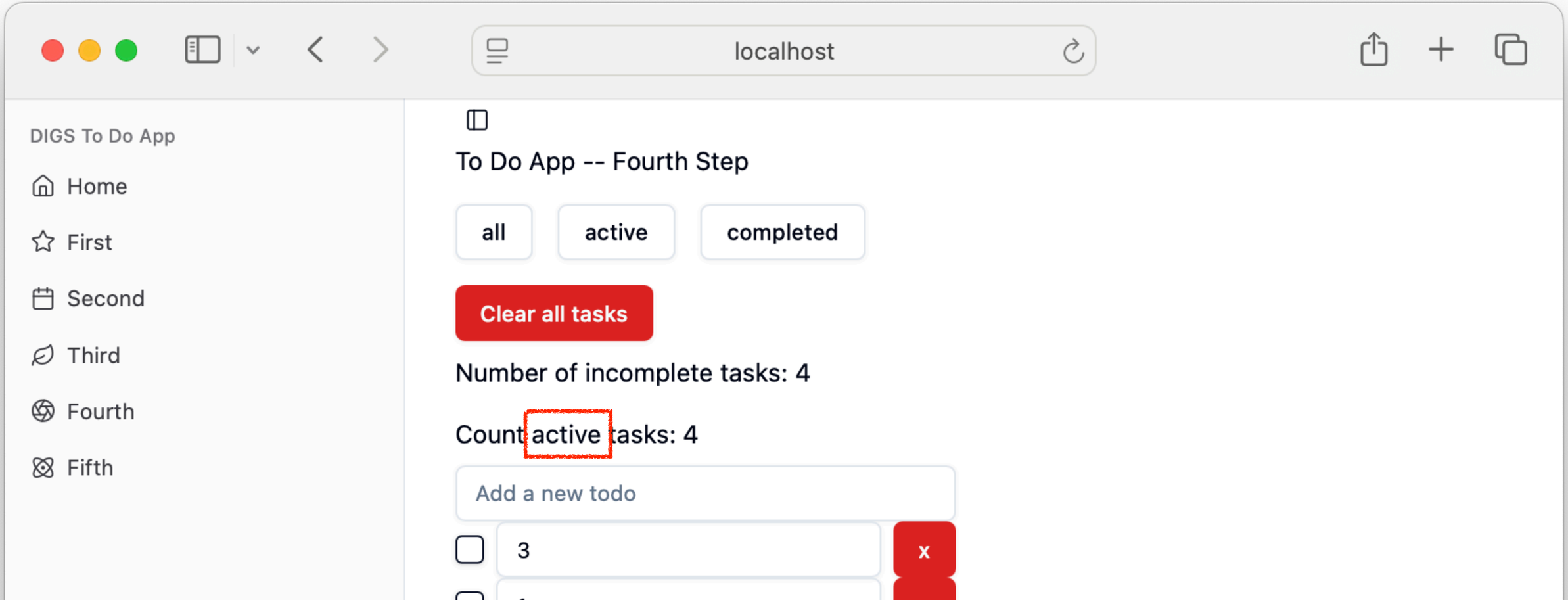


COUNTING FILTERED ITEMS

- Add a new below the current count
- `<div class="font-bold py-2">Count tasks: {filterCount()}</div>`
- Two steps:
  - How can we change the label to say the following based on button clicked?
    - "Count all tasks"
    - "Count incomplete tasks"
    - "Count completed tasks"
  - What does the filterCount() function do?

COUNTING FILTERED ITEMS

- `<div class="font-bold py-2">Count {SOMETHING} tasks: {filterCount()}</div>`



COUNTING FILTERED ITEMS

- The filterCount() function.
- If the filtered list has zero items, then say 'All clear!'
- Otherwise return the length of the array.

```
function filterCount(){

}
```

COUNTING FILTERED ITEMS

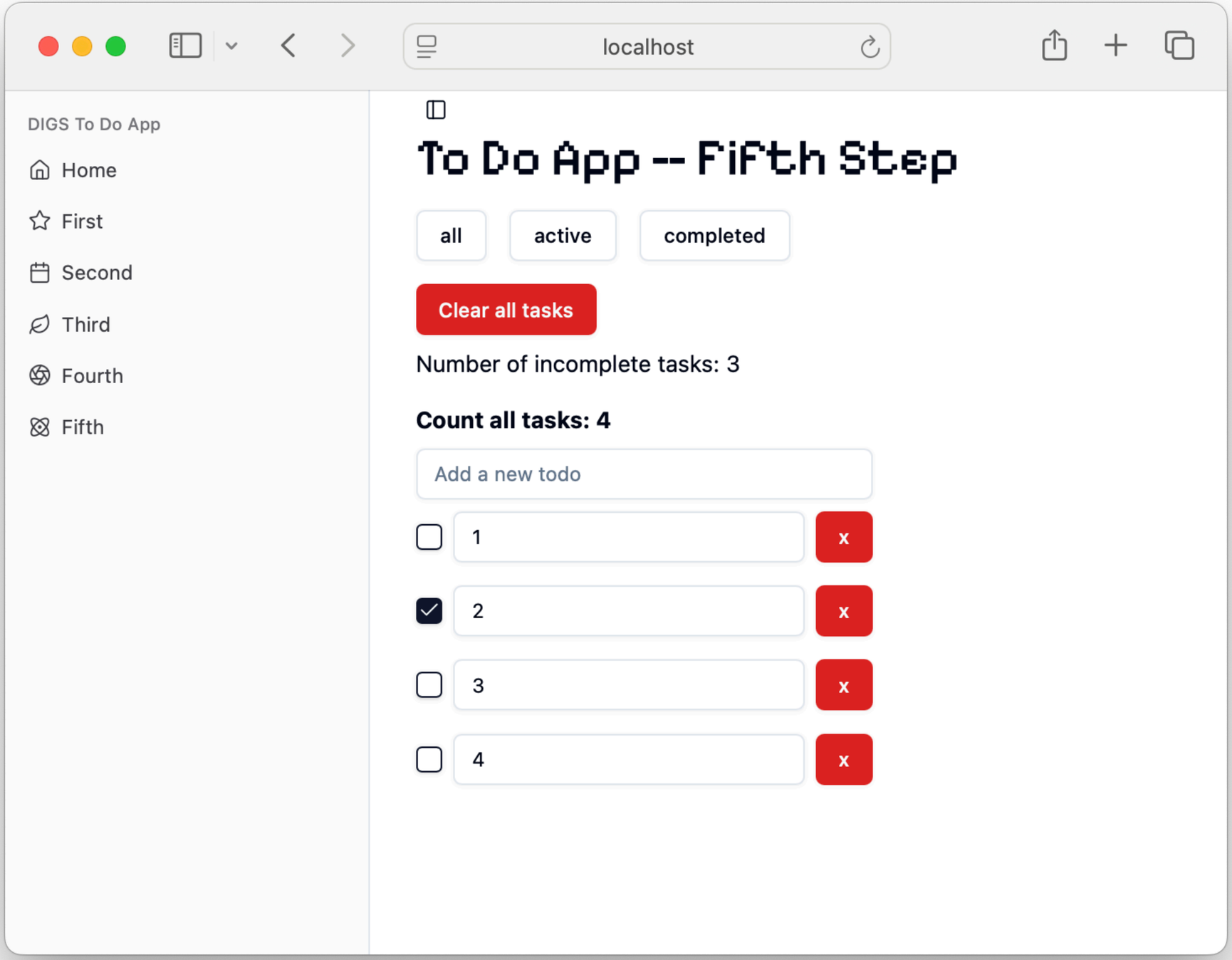
- Will this work?

```
function filterCount(){
  return todos.length === 0 ?
   'All clear!' :
   todos.length;
}
```

COUNTING FILTERED ITEMS

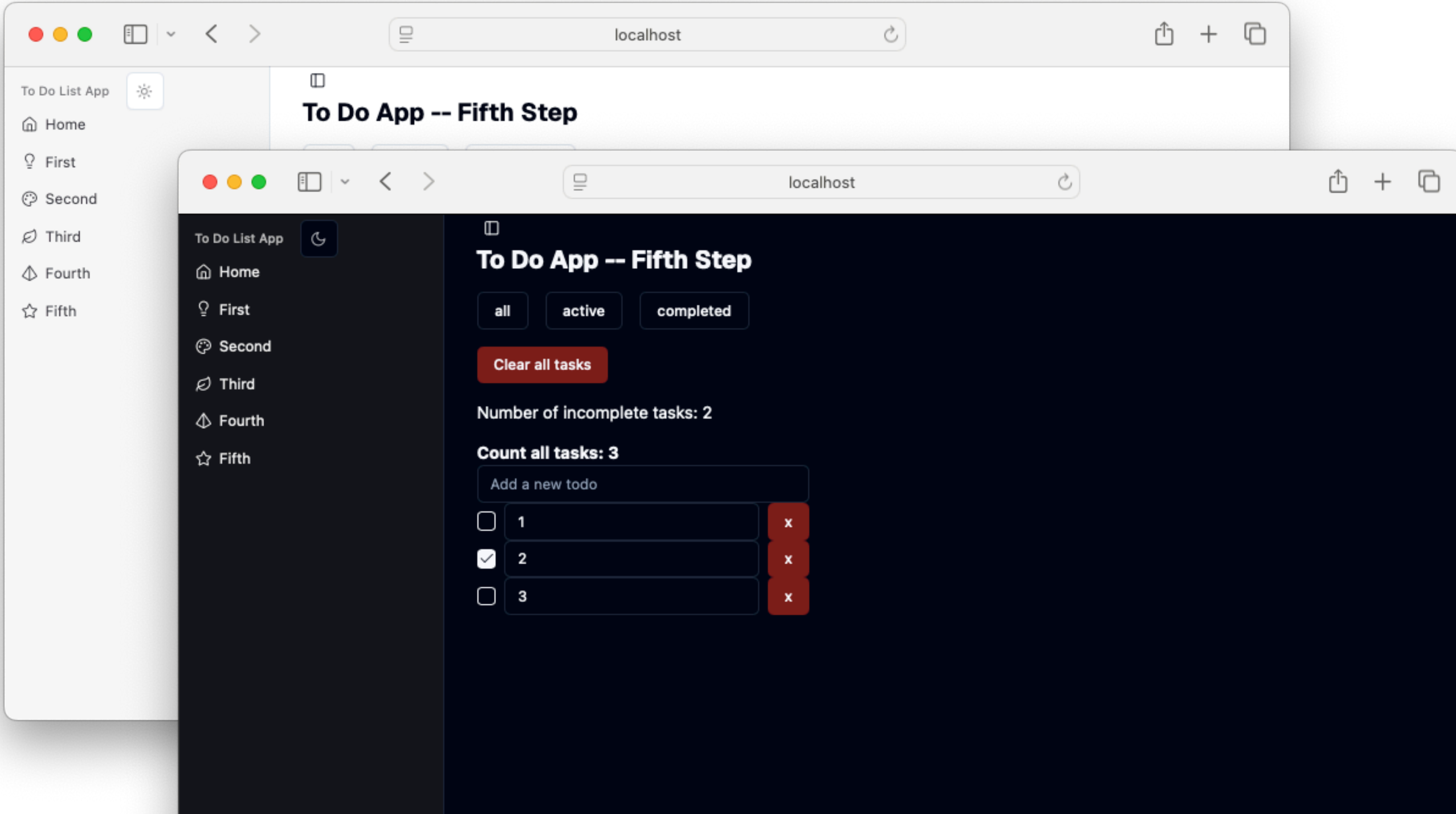
- We need to count the filtered list.
- We already wrote a function to do the filtering, so no need to repeat that code.
- Let's reuse the filteredList() function.

```
function filterCount(){
   return filteredList().length === 0 ?
    'All clear!' :
    filteredList().length;
 }
```


localhost
DIGS To Do App
Home
First
Second
Third
Fourth
Fifth
To Do App -- Fifth Step
all
active
completed
Clear all tasks
Number of incomplete tasks: 3
Count all tasks: 4
Add a new todo
1
x
2
x
3
x
4
x

CHALLENGE

- Add the ability to toggle between light and dark mode in your app.
- Here's where to find the shadcn documentation:
  - https://next.shadcn-svelte.com/docs/dark-mode/svelte


localhost
To Do List App
Home
First
Second
Third
Fourth
Fifth
To Do App -- Fifth Step
localhost
To Do List App
Home
First
Second
Third
Fourth
Fifth
To Do App -- Fifth Step
all
active
completed
Clear all tasks
Number of incomplete tasks: 2
Count all tasks: 3
Add a new todo
1
2
3
x
x
x