✅ Section 1: Svelte Essentials — Setup, Markup, and Reactivity

> 🧠 Focus: Svelte REPL usage, markup syntax, component structure, simple reactivity

- [ ] **Hello Svelte**
  Create a component that displays a message: `"Hello, Svelte!"`.
- [ ] **Dynamic Greeting**
  Create a `<script>` block with a `name` variable, and display `"Hello, {name}!"`.
- [ ] **Text Input Reflection**
  Add a text input bound to `name`, so that typing updates the greeting in real time.
  🧠 *Teaches:* `bind:value` *and reactivity*
- [ ] **Button Click Counter**
  Add a button that counts how many times it's clicked.
- [ ] **Conditional Message**
  Show `"You're awesome!"` only if the count is greater than 5.
- [ ] **Basic If/Else Toggle**
  Add a toggle button that switches between showing "Night" and "Day".
- [ ] **List Rendering**
  Display a list of fruits using `{#each}` syntax.
- [ ] **Empty List Handling**
  Use `{#if}` to display a message when the fruit list is empty.
- [ ] **List Filter UI**
  Filter the fruit list to show only those containing the letter "a".
- [ ] **Inline Style Binding**
  Use a slider to change the font size of some text in real time.

💡 Section 1 Challenge: *"Build a Smart Greeting Panel"*

> Create a small interface that:
>
> - Has a text input for `name`
> - Lets the user toggle between "casual" and "formal" tone
> - Displays one of four greetings:

- `"Hi, {name}!"` (casual + filled)
- `"Greetings, {name}."` (formal + filled)
- `"Hi there!"` (casual + blank)
- `"Welcome."` (formal + blank)

Bonus: Add a checkbox to hide/show the whole panel.

🧠 Requires: `{#if}`, `bind:value`, `on:click`, reactive vars, combining logic

Here is the fully updated:

✅ Section 2: Styling, Events, and DOM Interaction

🧠 Focus: inline and class-based styling, event handling, basic DOM manipulation (still REPL-only)

☐ **Style Some Text**
Use the `style` attribute to make text red, bold, and larger.

☐ **Class Binding Basics**
Add a `selected` class to a `<div>` when it is clicked. Toggle it on/off with each click.

☐ **Hover Highlight**
Change the background color of a box when hovered, using class binding and a reactive variable.

☐ **Mouse Tracker**
Show the current mouse coordinates inside a `<div>` using `on:mousemove`.

☐ **Click to Copy Message**
Clicking a message should replace it with `"Copied!"` for 2 seconds.

☐ **Text Color Changer**
Add a dropdown that lets users select a text color from a few options (e.g. red, blue, green).

☐ **Dark Mode Toggle**
Add a toggle switch that changes the background and text color of the entire component.

☐ **Double Click Detector**
Display a message only when an element is double-clicked, not single-clicked.

▢ **Input Character Counter**
Show how many characters have been typed in a textarea, updating live.

▢ **Random Style Generator**
Add a button that randomly changes the font size and color of a headline.

💡 Section 2 Challenge: *“Style Mixer Pad”*

Create a component with:

- A text input to enter any message
- A color picker (`<input type="color">`) to change text color
- A slider to adjust font size
- A checkbox that toggles italic style
- A button to randomize all styles at once

Bonus: Add a reset button to return to default styles.

🧠 Requires: reactive styling, DOM events, inputs, and binding — all combined interactively

✅ Section 3: Components and Props

🧠 Focus: Creating and using components, passing data with props, isolated composition (REPL-safe)

▢ **Make Your First Component**
Create a new file named `Box.svelte` that displays a colored box and use it in `App.svelte` .

▢ **Pass a Message Prop**
Pass a `message` prop into a child component and render it inside a `<p>` .

▢ **Reusable Colored Box**
Create a `ColorBox.svelte` that takes a `color` prop and sets background accordingly.

▢ **Multiple Instances with Different Props**
Render three `ColorBox` es with different colors.

- [ ] **Conditional Prop Behavior**
  Pass a `highlighted` boolean prop that changes a component's border if `true`.
- [ ] **Interactive Component with Props**
  Create a `Counter.svelte` that accepts a `start` prop and counts from that value.
- [ ] **Prop as Label Text**
  Create a `ButtonWithLabel` component that shows a label (via prop) above a button.
- [ ] **Default Prop Value**
  Set a default prop inside a component (e.g., default color is `"skyblue"`).
- [ ] **Prop-Based Class Toggle**
  Make a `Tag.svelte` component that uses a `selected` prop to apply a `selected` class.
- [ ] **Number Prop + Calculation**
  Pass a number to a component and display double its value using a reactive statement.

💡 Section 3 Challenge: *"Prop-Based Profile Card Generator"*

Create a `ProfileCard.svelte` component with the following props:

- `name`
- `avatarUrl`
- `bio`
- `highlight` (boolean)

The card should:

- Display the name, avatar image, and bio text
- Have a border color change when `highlight` is true
- Be used in `App.svelte` to render 3 cards with different data

🧠 Requires: reusable props, default values, conditional styling, and component composition

✅ Section 4: Component Events and Communication

🧠 Focus: Sending data from child to parent, custom events, event forwarding (all REPL-friendly)

- ▢ **Button Click Event in Child**
  Make a `Clicker.svelte` component with a button. Use it in `App.svelte` and log to console when it's clicked.
- ▢ **Emit Custom Event with `createEventDispatcher`**
  Inside `Clicker.svelte`, dispatch a `"count"` event every time the button is clicked.
- ▢ **Pass Event Data to Parent**
  Emit a `count` event with the current count value and handle it in the parent to show a running total.
- ▢ **Child Controls Parent Background**
  Dispatch a `"colorChange"` event from child with a color value. Parent should use it to change its background.
- ▢ **Form Submission with Event**
  Create a `LoginForm.svelte` that emits a `"submit"` event with the entered username.
- ▢ **Multiple Children Communicating Up**
  Create 3 `VoteButton.svelte` components. When clicked, each dispatches a `vote` event with `"like"`, `"love"`, or `"wow"`. The parent should count total reactions.
- ▢ **Forward Native Events with `on:click` Forwarding**
  Wrap a `<button>` in a component. Use `$$restProps` or `on:click` to forward the native click event.
- ▢ **Keyboard Event in Component**
  A `SearchBar.svelte` emits a `search` event when `Enter` is pressed inside an input.
- ▢ **Child Event Triggers Parent Toggle**
  Child emits `"toggle"` event; parent uses it to show/hide a box.
- ▢ **Event Cascade**
  Chain three components deep: `App.svelte → A → B`. Emit an event from `B`, catch and respond in `App`.

💡 Section 4 Challenge: *“Mini Poll Collector”*

Build a `PollOption.svelte` component that:

- Receives `label` as a prop

- Emits a `vote` event with its label when clicked

In `App.svelte`, render 4 such options and display live counts next to each label.

Bonus: Add a “Total Votes” display at the bottom.

🧠 Requires: props, custom events, event payloads, tracking state in parent from child events

✅ Section 5: Reactivity Deep Dive

🧠 Focus: `$:` reactive declarations, assignments, chained reactivity, computed values, reactive DOM

- [ ] **Basic Reactive Value**
  Create a number and a button that increments it using `$:` to automatically update a message.
- [ ] **Reactive Derived Value**
  Create two inputs: `length` and `width`, and use `$:` to compute `area = length * width`.
- [ ] **Chained Reactivity**
  Given a `counter`, define `double` and `quadruple` as reactive values based on it.
- [ ] **Reactive Style Binding**
  Create a slider that changes the size of a box using a reactive variable tied to style.
- [ ] **Reactivity With Dates**
  Show the number of seconds since the user clicked a button.
- [ ] **Conditional Reactivity**
  Use `$:` to change a message only when a certain variable exceeds a threshold.
- [ ] **Preventing Redundant Computation**
  Use a reactive block to log when a value changes and avoid re-running logic unnecessarily.
- [ ] **Reactive Input Mirror**
  Mirror the live value of a text input using a reactive statement.
- [ ] **Reactive Array Length Display**
  Add names to an array with a button, and display the length via `$:` reactivity.
- [ ] **Reactive Background Color**
  Change the page background using a dropdown of colors, with the value reactive.

💡 Section 5 Challenge: *“BMI Calculator with Reactive Warnings”*

Create two inputs for height (cm) and weight (kg).
Use reactive statements to calculate and display BMI.
Add a warning message if BMI > 30 or < 16.5 using reactivity alone.
Bonus: Color the BMI result text red if outside normal range.

🧠 Requires: multiple reactive declarations, conditional styling, chained computations

✅ Section 6: Stores and Global State

🧠 Focus: Writable stores, readable stores, derived stores, `$store` syntax, sharing state between components

- ▢ **Create a Writable Store**
  Define a `count` store using `writable(0)` and use it in a component with `$count`.
- ▢ **Increment Store from Button**
  Add a button that increments a writable store and reflects the change in real-time.
- ▢ **Multiple Components Share Store**
  Create a `Counter.svelte` component that uses a shared store. Use it twice in `App.svelte` and see synced updates.
- ▢ **Readable Clock Store**
  Make a readable store that emits current time every second using `setInterval`.
- ▢ **Store with Custom Methods**
  Build a store that tracks score and exposes `.increment()` and `.reset()` methods.
- ▢ **Derived Store: Full Name**
  Combine two writable stores (`firstName` , `lastName`) into a derived `fullName` store.
- ▢ **Custom Color Theme Store**
  Use a store to track theme color. Bind to CSS style or class to change background color.
- ▢ **Resettable Form with Store**
  Store the form state in a writable store. Provide a reset button that sets store back to defaults.
- ▢ **Store in Nested Component**
  Use a shared store inside a deeply nested component to update a value globally.

- ▢ **Toggle Store with Checkbox**
  Bind a checkbox input to a boolean store and toggle visibility of a div.

💡 Section 6 Challenge: *“Global Tab Tracker”*

Create a store that tracks the *currently selected tab*.
Render 3 components (`TabA`, `TabB`, `TabC`), each displaying "active" or "inactive" based on the current tab store.
Clicking a tab in one component should update the others instantly.

🧠 Requires: shared state, writable store, reactivity with `$`, conditional UI rendering

✅ Section 7: Lists and Each Blocks

🧠 Focus: `#each`, keyed blocks, dynamic lists, interactive lists, list manipulation, array reactivity

- ▢ **Render a Simple List**
  Given an array of animals, display each in an unordered list.
- ▢ **Numbered List with Index**
  Show a list of to-do items with their index using `#each item, i`.
- ▢ **List of Objects**
  Display a list of users (`{ name, age }`) showing both properties in a card layout.
- ▢ **Add to List Button**
  Use an input and button to append items to an array and reflect the change instantly.
- ▢ **Remove from List**
  Add a delete (❌) button to each list item that removes it when clicked.
- ▢ **Keyed Each Block**
  Use `#each users as user (user.id)` to preserve list identity when items change.
- ▢ **List Sorting**
  Create a list of numbers and a button that sorts them in ascending order.
- ▢ **Editable List Items**
  Allow the user to edit items inline (e.g. click an item → input appears).
- ▢ **List Filter**
  Use a text input to filter a list of items (e.g. search by name in a user list).

▢ **Multiple Lists from One Array**

Separate items into two categories (e.g. fruits vs vegetables) using `#each`.

💡 Section 7 Challenge: *“Interactive Ranked List”*

Display a ranked list of 5 items.
Each item has up/down arrows to change its position.
Ensure reordering works reactively using a keyed `#each`.

🧠 Requires: keyed blocks, index manipulation, reactive array updates, reordering logic

✅ Section 8: Event Handling & Interactivity

🧠 Focus: `on:click`, `on:input`, event forwarding, inline handlers, event modifiers

▢ **Handle a Click Event**

Create a button that says “Click Me” and updates a count each time it's clicked.

▢ **Log Input as You Type**

Use `on:input` to update a variable with what’s typed into a textbox, and display it below.

▢ **Pass Event to Function**

Create an `on:click={handleClick}` that logs `event.target` when clicked.

▢ **Inline Event Handler**

Make a button with an inline `on:click={() => alert('Inline!')}` event.

▢ **Multiple Events on Same Element**

Add `on:mouseenter` and `on:mouseleave` to change background color on hover.

▢ **Keyboard Event Listener**

Create an input that listens for `on:keydown` and displays the key pressed.

▢ **Event Modifier: preventDefault**

Add a form with a submit button that doesn't reload the page (using `on:submit|preventDefault`).

▢ **Forward Event to Parent**

In a component, use `createEventDispatcher()` to dispatch a `select` event and handle it in the parent.

▢ **Double Click Handler**

Add an `on:dblclick` to a box that changes its color on double-click.

▢ **Event with Data**

Pass an argument to your handler (e.g. `on:click={() => remove(item.id)}`) when a button is clicked.

💡 Section 8 Challenge: *“Custom Emoji Reaction Buttons”*

Create three emoji buttons (👍 😍 😡).
When any is clicked, increment its own counter.
Each button updates its label like “👍 4”, “😍 1”, “😡 2”.

🧠 Requires: event handlers with custom arguments, reactive state, inline logic

✅ Section 9: Forms & Bindings

🧠 Focus: `bind:value`, input controls, checkbox/radio binding, select menus, two-way data

▢ **Bind a Text Input**

Create an `<input>` and bind it to a variable so typing updates the screen in real-time.

▢ **Bind a Textarea**

Add a `<textarea>` for user feedback and show a live character count using `bind:value`.

▢ **Bind a Checkbox**

Use `bind:checked` to toggle a boolean and conditionally display a message.

▢ **Bind a Radio Group**

Create 3 radio buttons bound to the same variable to choose between “red”, “green”, “blue”.

▢ **Bind a Select Menu**

Bind a dropdown select to a variable and display the selected item below it.

▢ **Two Inputs, One Variable**

Bind two inputs (one range, one number) to the same value — changing one affects the other.

▢ **Form Submission with Bound Values**

Bind inputs in a mini form and display the collected data on submit (no backend or form reload).

▢ **Bind a List of Checked Items**

Create a checklist of items with checkboxes bound to an array — display selected items.

▢ **Editable Profile Form**

Bind fields like name, age, bio to inputs and show a live “preview” profile card.

▢ **Bind a Slider to a Visual**
Use a range slider bound to a number that controls the size of a circle on screen.

💡 Section 9 Challenge: *“Live Feedback Form with Validation”*

Create a form with name, email, and message fields.
Bind all inputs and show a live summary (e.g. “Hello, NAME!”).
Disable the submit button unless all fields are filled, and email includes “@”.

🧠 Requires: multiple `bind:`, validation, conditional UI, form reactivity

✅ Section 10: Advanced Reactivity

🧠 Focus: `$:`, reactive assignments, reactive statements, reactive arrays/objects

▢ **Basic Reactive Statement**
Create two number inputs and display their sum using `$:`.

▢ **Chain Reactive Values**
Use a reactive statement where one computed value depends on another (e.g., `c = a + b`, then `d = c * 2`).

▢ **Reactive Object Property Display**
Create an object with name/age, and reactively update a sentence like “NAME is AGE years old”.

▢ **Reactive Statement Logging**
Use `$:` to log when a variable changes, such as `count`.

▢ **Nested Reactive Statements**
Create a computed price from `pricePerItem * quantity`, then compute `taxedTotal = price * 1.18`.

▢ **Update Array Length Reactively**
Let users add/remove from an array, and show “You have N items” reactively.

▢ **Reactivity with Non-Primitive Types**
Bind an object to inputs (e.g., `user.name`) and use `$:` to compute `greeting`.

▢ **Avoid Unnecessary Reactivity**
Try updating an object property without reassigning the object — observe how it fails to react.

- ☐ **Use Spread to Trigger Reactivity**
  Fix the above case using `object = { ...object }` after mutating.
- ☐ **Track Derived State From User Input**
  Create a live character counter with color change if limit exceeded (e.g. 100+ characters = red).

💡 Section 10 Challenge: *"Real-Time Tip Calculator"*

Create inputs for bill amount, tip percent, and number of people.
Compute and display: total tip, grand total, per-person amount — all reactively.
Use `$:` and reactive chains only — no event handlers!

🧠 Requires: reactive chains, computed values, form binding, edge case handling

✅ Section 11: Component Props

🧠 Focus: passing data into components using `export let`, rendering dynamic content via props

- ☐ **Basic Prop Passing**
  Create a `<Greeting>` component that takes a `name` prop and displays "Hello, NAME!".
- ☐ **Number Prop for Calculations**
  Pass a `count` prop into a `<CounterSummary>` component that shows "You clicked COUNT times."
- ☐ **Boolean Prop for Toggling Behavior**
  Pass a `isAdmin` prop to conditionally render "Admin Access" or "User Access".
- ☐ **Styling via Props**
  Pass a `color` prop into a `<Box>` component and use it to set the background color.
- ☐ **Dynamic List via Props**
  Pass an array of tasks to a `<TodoList>` component that displays each task in a `<li>`.
- ☐ **Prop Default Values**
  Set a default value for a prop inside a component using `=`, e.g., `export let size = "medium"`.
- ☐ **Multiple Props Together**
  Pass `title`, `subtitle`, and `author` props into a `<Card>` component to render a blog preview.
- ☐ **Prop Type Gotchas**
  Experiment by passing the wrong prop type (e.g., a string instead of a number) and observe behavior.

- ▢ **Conditional Classes via Props**
  Create a `<Tag>` component that adds a `highlight` class if a prop `highlighted` is true.
- ▢ **Prop-Powered Icon Button**
  Pass an emoji or SVG string into a `<Button>` component as a prop and render it next to the label.

💡 Section 11 Challenge: *“Component Playground”*

> Build a `<ProfileCard>` component that takes props for `avatar`, `name`, `bio`, `isOnline`.
> Render it with different sets of props — one offline user, one online.
> Use `export let`, conditional styles, and slot a message inside the card.

🧠 Requires: multiple props, dynamic styling, conditional rendering, custom layout

✅ Section 12: Slots & Composition

> 🧠 Focus: `<slot>`, named slots, default content, wrapping components

- ▢ **Basic Slot Usage**
  Create a `<Card>` component with a `<slot>` and use it to wrap any custom content.
- ▢ **Default Slot Content**
  Add fallback content inside `<slot>` so something appears if no child is passed.
- ▢ **Named Slot Example**
  Create a layout component with `<slot name="header">`, `<slot>` (body), and `<slot name="footer">`.
- ▢ **Injecting Named Slots**
  Use the above component and fill the named slots with custom header/footer/body content.
- ▢ **Multiple Slot Wrappers**
  Make a `<Modal>` component with a named slot for the title and a default slot for content.
- ▢ **Slot for Actions**
  Build a `<Toolbar>` component with a slot for buttons passed from the parent.
- ▢ **Card Wrapper with Style**
  Create a `<FancyBox>` that styles the slot content with a colored border and padding.
- ▢ **Slot Props Awareness**
  Demonstrate how slot content is unaware of the parent component's variables.

- ▢ **Pass Interaction via Slot**
  Create a `<ConfirmDialog>` component that slots in a button, and shows a confirm box on click.
- ▢ **Use Multiple Instances with Different Slots**
  Render two `<Panel>` components with different slot content to demonstrate reuse.

💡 Section 12 Challenge: *“Dynamic Dashboard Widget”*

> Create a `<Widget>` component with named slots: `title`, `content`, and optional `footer`. Render 2–3 widgets (Weather, News, Notes) using this component. Each should have different slot content.

🧠 Requires: named slots, component reuse, visual layout design, thoughtful composition

✅ Section 13: Transitions & Animations

> 🧠 Focus: enter/leave transitions, `animate` directive, custom animations

- ▢ **Basic Fade In/Out**
  Use Svelte’s built-in `fade` transition to show/hide a message.
- ▢ **Slide Transition**
  Use `slide` to make a panel slide in from the side when toggled.
- ▢ **Scale Transition on Button Click**
  Animate a button growing/shrinking with the `scale` transition.
- ▢ **Custom Duration on Transitions**
  Use `{ duration: 800 }` to make a slow `fade` transition.
- ▢ **Fly In From Off-Screen**
  Use `fly` to animate an element entering from the bottom-right.
- ▢ **Combine Multiple Transitions**
  Combine `fade` and `slide` on an element to make entry more dynamic.
- ▢ **Animate a List on Reordering**
  Use `animate:flip` to smoothly reorder a list of numbers when shuffled.
- ▢ **Staggered List Transitions**
  Use `delay` to create staggered entry animations for a list of elements.

- [ ] **Conditional Visibility with Transitions**
  Animate a box appearing/disappearing with a toggle button.
- [ ] **Custom Transition Function**
  Write a simple custom transition that wobbles or rotates an element slightly on entry.

💡 Section 13 Challenge: *“Animated Notification Feed”*

> Build a mock notification list where new messages slide in and fade,
> and disappear when clicked (with animation). Add staggered entry for fun.

🧠 Requires: `in:`, `out:`, `animate:`, `delay`, list reactivity

✅ Section 14: Component Communication

> 🧠 Focus: props, events, bindings between parent and child components

- [ ] **Passing Props to Child**
  Create a `Greeting.svelte` component that accepts a `name` prop.
- [ ] **Using Props for Style**
  Pass a `color` prop to a `Tag.svelte` component to set its background color.
- [ ] **Child Emits Event to Parent**
  Have a `LikeButton` component emit a `liked` event when clicked, and handle it in the parent.
- [ ] **Emit With Payload**
  Emit an event with a payload (e.g. `{ id: 42 }`) and log it in the parent.
- [ ] **Component-Component Counter**
  Use a child component to increment a shared counter in the parent via events.
- [ ] **Prop Change Updates Child**
  Update a parent value and watch the child re-render automatically with the new value.
- [ ] **Child Controls Parent State**
  Make a `Switch` component that emits `onToggle` and updates the parent's boolean state.
- [ ] **Bind Prop Two-Way**
  Use `bind:value` to let a parent and child share and update the same variable.
- [ ] **Binding DOM Elements**
  Pass and bind an input field’s `value` to a parent using `bind:value`.

▢ **Intermediate Wrapper Component**
Pass props and events through an intermediate component (grandchild emits, parent handles).

💡 Section 14 Challenge: *"Mini Voting Booth"*

> Create a `VoteButton` component that takes a `label` prop, and emits `vote` events.
> In the parent, count how many times each button is clicked (e.g. "Cats" vs "Dogs").

🧠 Requires: props, event forwarding, payloads, parent state mutation

✅ Section 15: Advanced Stores & Derived State

> 🧠 Focus: custom stores, derived stores, reactive logic with `$store`

▢ **Writable Store Counter**
Create a `writable` store called `count` and update it with `increment()`.

▢ **Derived Store Example**
Create a `derived` store that returns "even" or "odd" based on the `count` store.

▢ **Store with LocalStorage Sync**
Write a store that syncs its value to `localStorage` and restores on page load.

▢ **Custom Store with API**
Create a `userStore` that wraps a `writable` and provides custom `login()` / `logout()` methods.

▢ **Derived Store from Two Sources**
Combine `a` and `b` into a `sum` store using `derived`.

▢ **Reactive UI from Store**
Create a toggle switch component that updates global store state and reflects instantly.

▢ **Multiple Components Reading One Store**
Use the same store in 3 different components and demonstrate live sync.

▢ **Chained Derivations**
Create a `total` store → derive `taxedTotal` → derive `finalTotal` with chained logic.

▢ **Async Derived Store**
Make a derived store that waits for a delay before updating (e.g. simulating debounce).

▢ **Resettable Store**
Create a custom store that can reset to its original default value with `reset()`.

💡 Section 15 Challenge: *“Reactive Calculator”*

Create a calculator UI with 3 inputs (price, quantity, discount), all bound to writable stores. Use derived stores to compute subtotal, discount amount, and final price. All values should update live.

🧠 Requires: writable, derived, custom logic, live reactivity, chaining

✅ Section 16: Keyboard, Mouse & Gestures

🧠 Focus: keyboard input, mouse tracking, event modifiers, and gestures

- ▢ **Keyboard Event Handler**
  Create an input that logs a message when you press the `Enter` key.
- ▢ **Prevent Default on Submit**
  Create a form with a submit button and prevent its default action using `on:submit|preventDefault`.
- ▢ **Mouse Coordinates Tracker**
  Track the mouse position (`x` , `y`) on the screen and display it in real-time.
- ▢ **Click Outside to Close**
  Show a dropdown that closes when the user clicks outside of it.
- ▢ **Escape Key to Close Modal**
  Open a modal on button click and close it on `Escape` key press.
- ▢ **Double Click Detector**
  Create a box that changes color only on a double-click.
- ▢ **Hover Tooltip**
  Show a tooltip when hovering over a specific word or button.
- ▢ **Keypress Shortcuts**
  Create keyboard shortcuts (e.g., `Ctrl+S` to simulate "save").
- ▢ **Drag Element Around**
  Make a circle that can be dragged around inside a box using mouse events.
- ▢ **Resizable Panel**
  Implement a resizable sidebar using `mousedown` , `mousemove` , and `mouseup` .

💡 Section 16 Challenge: *“Drag-to-Dismiss Notification”*

Display a dismissible notification box.
Let users **drag it horizontally**; if dragged beyond a threshold, it disappears.
Otherwise, it snaps back.

🧠 Requires: mouse events, conditional rendering, gesture logic, transitions

✅ Section 17: Transitions & Animations

🧠 Focus: built-in transitions, custom animations, and timing control

- ▢ **Fade In/Out a Box**
  Show/hide a box using Svelte’s built-in `fade` transition.
- ▢ **Slide In a Panel**
  Create a sidebar that slides in and out using the `slide` transition.
- ▢ **Scale on Mount**
  Display a component that gently scales into view with `scale` transition.
- ▢ **Custom Transition Duration**
  Modify the `fade` transition to last 2 seconds.
- ▢ **Transition with Easing**
  Use `fly` transition with an `easeInOut` function.
- ▢ **Crossfade Between Elements**
  Animate the switch between two boxes using `crossfade` .
- ▢ **Transition When Data Changes**
  Fade between different quotes (from a list) when you click a button.
- ▢ **Custom CSS Animation**
  Create a blinking text using a CSS `@keyframes` animation in a `<style>` block.
- ▢ **Animate Height on Expand**
  Make a details box that expands/collapses with animated height.
- ▢ **Staggered List Animation**
  Show list items one by one using `each` + delayed `fade` .

💡 Section 17 Challenge: *“Animated Card Shuffler”*

Create 4 cards in a row. On button press, shuffle them randomly.
Animate their movement with `crossfade` so the card rearrangement looks smooth.
Cards should not just instantly jump.

🧠 Requires: keyed each blocks, crossfade, transitions, and array manipulation

✅ Section 18: Accessibility, ARIA & Screenreader Considerations

🧠 Focus: writing inclusive UI with semantic HTML, ARIA roles, focus management, and screenreader support

- [ ] **Semantic HTML for Buttons**
  Create a styled `div` that acts like a button, then replace it with a real `<button>` for accessibility.
- [ ] **Add ARIA Labels**
  Use `aria-label` on an icon-only button (like a trash can) to describe its action to screen readers.
- [ ] **Keyboard Focus Visibility**
  Use `:focus` and `:focus-visible` styles to show when an element is focused.
- [ ] **Skip Link Navigation**
  Add a “Skip to Content” link that jumps the user directly to the main section when pressed.
- [ ] **Trap Focus in Modal**
  Create a modal that traps tab focus within itself using `tabindex` and JavaScript logic.
- [ ] **Toggle Button with ARIA Expanded**
  Create a dropdown toggle button that dynamically sets `aria-expanded`.
- [ ] **Role="alert" Live Region**
  Display a dynamically updated message (e.g., “Saved!”) using a `<div role="alert">`.
- [ ] **Accessible Toggle Switch**
  Build a custom toggle switch using `button`, and make sure it’s accessible via keyboard and screen readers.
- [ ] **ARIA Describedby for Context**
  Use `aria-describedby` to provide contextual help text for a form input.
- [ ] **Accessible Tab Panel**
  Create tabs with appropriate roles (`tablist`, `tab`, `tabpanel`) and keyboard navigation support.

💡 Section 18 Challenge: *“Accessible Notification Manager”*

Create a notification component that:

- Uses `role="status"` or `role="alert"`
- Announces itself when appearing
- Is dismissible by both keyboard (`Escape`) and screen reader users
- Uses semantic HTML and appropriate ARIA attributes

🧠 Requires: semantic elements, ARIA roles, event handling, and timing management

✅ Section 19: Advanced Component Patterns

🧠 Focus: slots, context API, advanced composition, and reusable abstractions

- [ ] **Basic `<slot>` Usage**
  Create a `Card` component that displays its content via `<slot>`.
- [ ] **Named Slots**
  Make a `Modal` component with `slot="header"`, `slot="body"`, and `slot="footer"` for flexible layout.
- [ ] **Fallback Slot Content**
  Add fallback text inside a slot that appears when nothing is passed in.
- [ ] **Component with Props and Slots**
  Build a `Notification` component that accepts a `type` prop and custom content via slot.
- [ ] **Render Props via Functions**
  Pass a function as a prop to a component and call it inside to render dynamic content.
- [ ] **Use `setContext` and `getContext`**
  Create a parent component that provides values to deeply nested children using context.
- [ ] **Context API for Theme**
  Set a "dark" or "light" theme at the top level and consume it in nested components with `getContext`.
- [ ] **Compound Components with Context**
  Make a `Tabs` component where `TabList`, `Tab`, and `TabPanel` communicate via shared context.
- [ ] **Polymorphic Component**
  Make a `Button` component that renders as `<button>`, `<a>`, or custom tag depending on a prop.

▢ **Dynamic Slot Forwarding**
Build a wrapper component that forwards its slot to another internal component's slot.

💡 Section 19 Challenge: *"Smart Accordion with Context & Slots"*

> Create an `AccordionGroup` component that manages multiple `AccordionItem`s.
> Use context to allow each item to register itself and only one to be open at a time.
> The items should support custom headers and bodies via named slots.

🧠 Requires: slot composition, context sharing, dynamic state, and reusable design

✅ Section 20: Performance, Optimization & Lazy Techniques

> 🧠 Focus: rendering performance, efficient reactivity, lazy loading, and large app strategies

▢ **Avoid Unnecessary Renders**
Create a parent component with a heavy child and prevent the child from rerendering unless a specific prop changes.

▢ **Use Reactive Statements Wisely**
Optimize a component by converting expensive computations into reactive declarations using `$:` only when needed.

▢ **Throttled Input Handling**
Build a text input that only updates a computed value after the user stops typing for 500ms (debounce/throttle).

▢ **Lazy-Load Components**
Dynamically import a component with `await import()` only when a button is clicked.

▢ **Virtual List (Large Dataset)**
Render only the visible portion of a 10,000-item list based on scroll position.

▢ **Memoization with Reactive Stores**
Create a derived store that caches and reuses computations across multiple components.

▢ **Use `bind:this` to Delay DOM Access**
Defer DOM-intensive work (like measuring height) until the element is mounted.

▢ **Unsubscribe from Stores Manually**
Create a component that manually subscribes to a store on mount and unsubscribes on destroy.

- [ ] **Code-Split Routes**
  Simulate a simple router that loads route components lazily to reduce initial bundle size.
- [ ] **Only Animate What’s Needed**
  Use `animate:flip` or `transition:*` only on changed items to reduce reflows.

💡 Section 20 Challenge: *“Lazy Virtualized Infinite Scroll List”*

> Create a scrollable list that:
>
> - Starts with 20 items
> - Loads 20 more when the user scrolls near the bottom
> - Only renders the visible items using virtualization
> - Uses lazy-loaded placeholder components for performance

🧠 Requires: list virtualization, scroll detection, lazy imports, and performance mindfulness

✅ Section 21: Animations & Transitions Mastery

> 🧠 Focus: using Svelte’s built-in transitions, animations, and custom motion logic

- [ ] **Basic `transition:fade`**
  Apply `transition:fade` to make a message appear and disappear smoothly.
- [ ] **`transition:fly` with Parameters**
  Use `transition:fly={{ y: 200, duration: 500 }}` to slide in a new element.
- [ ] **`transition:slide` on Conditional List**
  Show a list of items with `slide` transitions as they’re added/removed.
- [ ] **Custom Transition Function**
  Write a transition that changes `scale` and `opacity` together.
- [ ] **Animate Between Layout States**
  Use `animate:flip` to smoothly transition reordered items in a grid layout.
- [ ] **Keyed `{#each}` with Transition**
  Use a keyed list with `transition:fade` to ensure animations play when items are swapped.
- [ ] **Staggered Entry Animations**
  Display a series of elements with increasing delays for a staggered animation effect.

- [ ] **Transition Group Based on Index**
  Animate a list where each item flies in from a different direction based on its index.
- [ ] **Manual Animation Using `tick()`**
  Animate an element's position manually using `tick()` and a reactive loop.
- [ ] **Custom Motion with `requestAnimationFrame`**
  Animate a bouncing ball using physics-based motion logic instead of transitions.

💡 Section 21 Challenge: *"Morphing Menu: Staggered Transitions + Layout Flip"*

Build a vertical menu that:

- Staggers in each menu item with `fly`
- When toggled, morphs to a horizontal layout using `animate:flip`
- Reverts back with a smooth layout transition
- Should feel like a natural, unified motion experience

🧠 Requires: multiple transition types, layout flip, custom parameters, animation composition

✅ Section 22: Accessibility & UX Enhancements

🧠 Focus: building accessible, user-friendly interfaces with keyboard support and ARIA roles

- [ ] **Add Alt Text to Images**
  Display an image with appropriate `alt` text for screen readers.
- [ ] **Label Inputs Clearly**
  Use `<label for>` and `id` to associate text with form inputs.
- [ ] **Use `aria-*` Attributes**
  Add `aria-expanded` and `aria-controls` to a collapsible element.
- [ ] **Keyboard Navigation with `tabindex`**
  Create custom focusable elements using `tabindex`.
- [ ] **Trap Focus in a Modal**
  Build a modal where keyboard tabbing stays inside until it's closed.

- ▢ **Announce Dynamic Content with ARIA Live**
  Use `aria-live="polite"` to announce text that changes dynamically.
- ▢ **Accessible Toggle Button**
  Build a toggle button that updates `aria-pressed` and works with keyboard input.
- ▢ **Focusable Custom Elements**
  Create a custom dropdown where keyboard arrows navigate the list.
- ▢ **Role-Based Alerts**
  Build a toast/alert that announces itself with `role="alert"` for screen readers.
- ▢ **Highlight Focus Visibly**
  Style focus outlines clearly so users can see where the keyboard focus is.

💡 Section 22 Challenge: *“Keyboard-Accessible Accordion with ARIA Support”*

> Build an accordion that:
>
> - Supports keyboard navigation (Arrow keys and Enter)
> - Updates `aria-expanded` and `aria-controls` appropriately
> - Traps focus inside each open section
> - Announces changes to screen readers

🧠 Requires: `aria-*`, `tabindex`, keyboard events, and accessibility-first design

✅ Section 23: Component Composition Patterns

> 🧠 Focus: writing clean, reusable, and composable Svelte components using slots and context

- ▢ **Simple Slot Usage**
  Create a component that wraps a `<div>` and displays whatever is passed inside via `<slot>`.
- ▢ **Named Slots**
  Make a card component with `slot="header"` and `slot="footer"` for flexible layout.
- ▢ **Default Fallback Content in Slot**
  Use fallback text like `"No content provided"` when slot content is empty.

▢ **Slot Props**
Pass a value from the child to the parent via a `<slot let:...>` and render it conditionally.

▢ **Composition with Slot Components**
Create a `Modal.svelte` that uses slots for title and content, and can be reused easily.

▢ **Passing Props to Children**
Use `export let` on a child component and pass props from the parent dynamically.

▢ **Component Nesting and Reuse**
Create a `Layout` component that nests multiple components like `Sidebar`, `Header`, and `Main`.

▢ **Context API: `setContext()` + `getContext()`**
Share a theme color or a user ID from a parent component down the tree without props.

▢ **Scoped Context for Nested Components**
Use `getContext()` inside deeply nested components to access parent data without prop drilling.

▢ **Custom Component Renderer**
Create a wrapper that takes a component as a prop and renders it using `<svelte:component>`.

💡 Section 23 Challenge: *“Composable Notification System with Slots + Context”*

Create a notification component that:

- Accepts message text and type (error/success/info) via props
- Lets users inject a custom action button using a `slot`
- Uses context to expose a `dismiss()` method to the slotted action
- Can be reused across the app without prop drilling

🧠 Requires: slot composition, context API, scoped reusability

✅ Section 24: Building Mini UI Systems

🧠 Focus: implementing small, self-contained interactive components with advanced patterns

▢ **Build a Tab System**
Create a set of tabs that toggle content when clicked using local state and slot composition.

- ▢ **Dropdown Menu with Outside Click Detection**
  Close a dropdown menu when the user clicks outside the element using DOM events.
- ▢ **Star Rating Component**
  Let users hover and select a 1–5 star rating, with real-time visual feedback.
- ▢ **Carousel with Navigation Buttons**
  Build a basic carousel with Next/Prev buttons that cycle through images or content.
- ▢ **Progress Bar Component**
  Show an animated progress bar that fills based on a prop or state change.
- ▢ **Toast Notification Queue**
  Show multiple toasts in sequence, disappearing after a delay, using an array of notifications.
- ▢ **Copy-to-Clipboard Button**
  Build a button that copies text to the clipboard and shows feedback like "Copied!".
- ▢ **Dynamic Theme Switcher**
  Create a toggle to switch between dark and light mode using a reactive class or style.
- ▢ **Drag-and-Drop Reordering**
  Build a sortable list where items can be dragged and dropped to reorder.
- ▢ **Stepper Component (Multi-step Form UI)**
  Show different steps in a process using previous/next buttons and active step indication.

💡 Section 24 Challenge: *“Reusable Multi-Tab System with Keyboard Nav + Transitions”*

Build a fully accessible and keyboard-navigable tab system that:

- Uses slots for tab labels and content
- Supports arrow key navigation between tabs
- Animates tab content transitions
- Allows multiple instances on the same page

🧠 Requires: composition, keyboard events, animation, and advanced slot logic

✅ Section 25: Real-Time UI Interactions

> 🧠 Focus: mastering fine-grained reactivity and building interfaces that respond instantly to user behavior

- ▢ **Live Character Counter**
  Display the number of characters typed in a `<textarea>` and update it reactively.
- ▢ **Auto-Save Draft Timer**
  Simulate an auto-save draft label that updates every 5 seconds using `setInterval`.
- ▢ **Interactive Word Filter**
  Filter a list of words in real time as the user types into an input box.
- ▢ **Typing Indicator Simulation**
  Show "Typing..." after user input, and hide it after 2 seconds of inactivity.
- ▢ **Live Search Highlight**
  Highlight matching substrings in a block of text as a user types a search query.
- ▢ **Two-Field Auto Calculator**
  Update a result field instantly as two numeric inputs change (e.g., sum or multiply).
- ▢ **Dynamic Range Preview**
  Display a live value as the user drags a range slider.
- ▢ **Mouse Position Tracker**
  Show the current mouse X/Y coordinates inside a box in real-time.
- ▢ **Form Progress Tracker**
  Track completion of fields in a form and show a live completion percentage.
- ▢ **Live Tag Parser**
  Let users type `#tags` into a field and extract and display unique tags in real time.

💡 Section 25 Challenge: *“Live Code Previewer”*

> Build a two-pane component where:
>
> - The left pane is a `<textarea>` where a user types markdown
> - The right pane shows the rendered HTML preview (e.g., with bold, italics)
> - It updates live as the user types

- Bonus: throttle the updates to 100ms to avoid lag

🧠 Requires: real-time updates, text parsing, throttling, and two-way layout sync

✅ Section 26: Game-Like Interactions & Microgames

🧠 Focus: building playful, responsive UIs that feel alive and game-like

▢ **Click Counter with Combo Multiplier**
Increase a counter and multiply the value if the user clicks rapidly within a time window.

▢ **Simon Says Color Memory Game (Mini Version)**
Flash a simple 2-step color sequence and ask the user to repeat it via buttons.

▢ **Animated Timer with Countdown Ring**
Show a 10-second countdown with a circular SVG ring that shrinks as time passes.

▢ **Reaction Time Tester**
Random delay, then prompt the user to click as fast as possible — show reaction time in ms.

▢ **Drag-to-Complete Slider Puzzle**
Slide a puzzle block to complete an image, using drag events and snapping.

▢ **Click-and-Hold Progress Unlocker**
Require the user to press and hold a button to trigger an action after a few seconds.

▢ **Floating Score Particles**
When a user clicks, show a +1 floating number that animates upward and fades.

▢ **WASD Keyboard Movement Box**
Move a square inside a box using WASD keys; show live keypress feedback.

▢ **Random Prize Picker Wheel**
Build a spinning prize wheel that lands on a random slice.

▢ **Emoji Rain Animation on Button Click**
Trigger an animated "rain" of emojis across the screen using `setInterval`.

💡 Section 26 Challenge: *“Click Frenzy Mini Game with Levels”*

Create a button-clicking game where:

- The button changes position randomly on every click

- You track the number of successful clicks within 15 seconds
- You display the user’s level (e.g., "Beginner", "Speed Demon") based on score
- Use animations for the button's movement and level change

🧠 Requires: timers, animations, state management, and randomized behavior

✅ Section 27: Dynamic Lists, Grids & Layouts

🧠 Focus: deeply understanding `each` blocks, keyed updates, reordering, and layout-driven interactions

- [ ] **List of Names with Add/Remove Buttons**
  Create a list where users can dynamically add and remove names from an array.
- [ ] **Sortable List with Up/Down Buttons**
  Let users reorder items in a list using up/down arrows (no drag/drop yet).
- [ ] **Grid of Colored Boxes from Data**
  Display a responsive grid of colored boxes from a data array, changing colors with a click.
- [ ] **Expand/Collapse List Items**
  Each list item can be expanded to reveal more content when clicked.
- [ ] **Highlight on Hover in List**
  Highlight the currently hovered item in a list using dynamic CSS classes.
- [ ] **Filterable List by Category**
  Add a dropdown to filter a list by category (e.g., show only "Fruits").
- [ ] **Searchable List with Highlighted Matches**
  Type to filter a list and highlight matched substrings within items.
- [ ] **List Shuffler with Animation**
  Shuffle a list of cards and animate the transition using Svelte’s built-in transitions.
- [ ] **Keyed List with Count-Up Animation**
  Animate each item individually (e.g. count-up effect) based on their key.
- [ ] **Paginated Grid Display**
  Divide a long list into pages and add “Next”/“Previous” buttons to browse items.

💡 Section 27 Challenge: *“Interactive Inventory Grid”*

Create a 3x3 grid that represents an inventory where:

- Each cell can hold one item (with a name and color)
- Users can drag and drop items between cells
- Dropping an item into an occupied cell swaps the two
- Bonus: Animate the movement or swap

🧠 Requires: dynamic layouts, drag/drop logic, keyed state, and reactivity

✅ Section 28: Forms, Validation, and User Input Patterns

🧠 Focus: mastering controlled inputs, form reactivity, dynamic validation, and user feedback

▢ **Basic Text Input with Live Preview**
Bind a text input to a variable and show the typed text live below it.

▢ **Multi-Field Form with Submit Button**
Build a simple form (e.g. name + email) and log the input values on submission.

▢ **Checkboxes and Radio Buttons**
Create a survey-style input with checkbox groups and radio buttons.

▢ **Live Form Validation with Warnings**
Add live validation (e.g. name must be > 3 characters), showing error messages dynamically.

▢ **Disable Submit Until Valid**
Automatically disable the submit button unless the form is valid.

▢ **Character Count with Warning Color**
Track input length, show remaining characters, and change color when near max.

▢ **Dynamic Form Fields**
Allow users to add or remove input fields dynamically (e.g. add more skills).

▢ **Dropdown-Driven Form Changes**
Show different input fields based on a dropdown selection (e.g. contact type).

▢ **Form Data Preview Card**
As the user types, show a live preview of what the form submission would look like in a stylized card.

▢ **Form Reset and Undo Buttons**
Let users clear a form or undo the last change.

💡 Section 28 Challenge: *“Live CV Builder Form”*

Create a form that collects:

- Name, title, bio, and up to 5 skills
- Shows a live rendered CV card preview on the side
- Validates the name and bio length, requires at least 1 skill
- Lets the user reset or undo their last input

🧠 Requires: controlled inputs, dynamic lists, validation, undo state, and real-time preview rendering

✅ Section 29: User Feedback and Microinteractions

🧠 Focus: subtle animations, responsive state changes, tactile UI feel, and feedback loops

▢ **Button with Loading Spinner**
When clicked, disable the button and show a loading spinner for 1.5 seconds before re-enabling.

▢ **Like Button with Count and Animation**
Clicking a heart icon increases a like counter with a little pop animation.

▢ **Hover Tooltip on Icon**
Show a tooltip on hover over an icon with a delay and fade transition.

▢ **Snackbar Notification on Action**
After an action (e.g. submitting a form), show a temporary "toast" or snackbar message.

▢ **Undo Notification with Countdown**
After deleting something, show an “Undo” toast with a countdown timer.

▢ **Auto-Scrolling Notification List**
Add messages to a notification list that scrolls older ones away after a timeout.

▢ **Input with Live Validation Icons**
Show a green checkmark or red X beside a form input depending on validity.

▢ **Click Ripple Effect**
Add a ripple animation to a button when clicked, like Material Design.

- [ ] **Press and Hold Button with Timer**
  Only trigger an action if the button is pressed and held for 2 seconds (show progress).
- [ ] **Interactive Star Rating Component**
  Build a 5-star rating widget with hover preview, click selection, and reset.

💡 Section 29 Challenge: *“Animated Reaction Bar”*

> Build a row of 6 emoji reaction buttons (👍 ❤️ 😂 😮 😢 👎) that:
>
> - Show a tooltip on hover
> - Animate when selected (scale/bounce)
> - Track and update a reaction count per emoji
> - Let users undo or change their reaction
> - Bonus: fade out inactive reactions over time

🧠 Requires: transitions, user input state, feedback timing, hover events, and cleanup logic

✅ Section 30: Conditional Rendering and Reactive Blocks

> 🧠 Focus: using `{#if}`, `{#each}`, `{#await}`, and `$:` to build reactivity and dynamic displays

- [ ] **Conditional Message Display**
  Use `{#if}` to show a welcome message only if a name is entered.
- [ ] **Toggle Visibility with Button**
  Create a toggleable FAQ answer section using `{#if}` blocks.
- [ ] **List of Items with `{#each}`**
  Display a list of hobbies from an array using `{#each}`.
- [ ] **Conditionally Styled Items**
  Render items with different colors/styles based on a condition.
- [ ] **Show Loading State with `{#await}`**
  Simulate a delay with a Promise and show a loading message until resolved.
- [ ] **Error Handling with `{#await}`**
  Add an error block that renders if a Promise rejects.

- [ ] **Reactive Derived Value with `$:`**
  Show the doubled value of a number input using a reactive statement.
- [ ] **Reactive Class Toggle**
  Bind a class based on a variable (e.g. "active" if selected).
- [ ] **Dynamic List Filter**
  Let users search/filter through a list and show only matching results reactively.
- [ ] **Countdown Timer Using `$:`**
  Create a timer that updates every second using a reactive block.

💡 Section 30 Challenge: *“Reactive Weather Simulator”*

Create a simulated weather display that:

- Uses a dropdown to select a city
- Shows a loading spinner while "fetching" (simulated with timeout)
- Displays a weather report with emoji (🌧️☀️⛈️ etc.) based on selected city
- Changes background color based on weather type
- Has a live "last updated" time that auto-updates every 5 seconds

🧠 Requires: conditional rendering, `{#await}`, `$:`, reactive classes, timers, and component state

✅ Section 31: Local Storage, Persistence, and State Survival

🧠 Focus: saving and restoring UI state using `localStorage`, browser APIs, and persistent variables

- [ ] **Save Name to Local Storage**
  Create an input that remembers the entered name even after refresh.
- [ ] **Theme Toggle with Persistence**
  Toggle between light and dark mode and save the preference in `localStorage`.
- [ ] **Task List that Survives Reload**
  A todo list that stores its state in local storage (add/remove items).
- [ ] **Form Autofill from Local Storage**
  Pre-fill a form from previously saved values in `localStorage`.

- ▢ **Checkbox Group with Remembered Selection**
  Let users check options and preserve the checked ones between sessions.
- ▢ **Dynamic Background Color Remembered**
  Choose a background color with a color picker and save it to persist.
- ▢ **Dropdown Selection Persisted**
  Save the selected option of a dropdown across reloads.
- ▢ **Slider Value Memory**
  Use a slider input and store its value persistently.
- ▢ **Visit Counter Using Local Storage**
  Count how many times a user has visited the page.
- ▢ **Last Visit Timestamp**
  Show the date and time of the user's last visit using local storage.

💡 Section 31 Challenge: *"Mini Preferences Dashboard"*

Build a preferences panel that:

- Lets the user toggle theme (light/dark), font size (small/medium/large), and color theme
- Saves these to `localStorage`
- Restores them automatically on load
- Applies the settings to the page layout using classes and inline styles
- Bonus: Include a "Reset to Default" button that clears everything

🧠 Requires: local storage usage, reactive bindings, styling based on state, and persistence logic

✅ Section 32: Advanced Transitions and State-Driven Animation

🧠 Focus: controlling complex animations and transitions with state, conditions, and timing

- ▢ **Chained Transitions**
  Animate multiple elements entering one after the other in a chain.
- ▢ **Conditional Fade and Fly**
  Combine `fade` and `fly` transitions based on element type or state.

▢ **Custom Parameters for Transitions**
Use a `fly` transition with dynamic `delay` , `duration` , and `x/y` values.

▢ **Transition on List Add/Remove**
Animate items entering and leaving a list using `{#each}` and keyed transitions.

▢ **Staggered List Animation**
Animate a list with a delay between each item’s appearance.

▢ **Manual Transition Trigger**
Animate an element in/out with a button press, not a reactive `{#if}` .

▢ **Nested Element Transitions**
Transition parent and child elements with different effects.

▢ **Progress Bar with Spring Motion**
Create a loading bar using spring-like animation (use `tweened` store or CSS transitions).

▢ **"Bouncing" Effect on Button Click**
Animate a button to bounce slightly when clicked.

▢ **Pulse Animation Loop**
Make an icon or element pulse (scale up/down) repeatedly using timers and CSS animation.

💡 Section 32 Challenge: *“Animated Notification Center”*

Build a floating notification center that:

- Allows adding new messages with a “+” button
- Shows each message with a different entrance animation (e.g. fade, fly, scale)
- Removes messages after 5 seconds with an exit animation
- Messages should stagger their entry when multiple are added quickly
- Include a dismiss button that removes a message manually (with exit transition)

🧠 Requires: list transitions, timing coordination, conditional rendering, and user-driven animation triggers

✅ Section 33: Accessible & Semantic UI Patterns

🧠 Focus: writing accessible, semantic HTML with Svelte, including ARIA roles, keyboard support, and screen-reader-friendly UIs

- [ ] **Use of `<button>` vs `<div>` for Clickable Actions**
  Understand when to use semantic tags like `<button>` and why it's better than `<div>` for interactivity.
- [ ] **Accessible Labeling with `aria-label`**
  Add screen-reader-friendly labels to an icon-only button using `aria-label`.
- [ ] **Keyboard Navigation for a List**
  Allow arrow key navigation through a list of items (e.g. menu or links).
- [ ] **ARIA Roles for Modal Dialog**
  Add correct roles and keyboard trap (Escape key closes) for a modal.
- [ ] **Focusable Custom Component**
  Make a custom component focusable and keyboard-operable.
- [ ] **Semantic Form Elements**
  Use `<fieldset>`, `<legend>`, `<label>`, and `<input>` properly to build a survey form.
- [ ] **Skip to Content Link**
  Add a visually-hidden "Skip to Content" link that becomes visible on focus and jumps to main content.
- [ ] **Visually Hidden Text for Icons**
  Provide a hidden description for screen readers when using icons as buttons.
- [ ] **Tab Trap in Modal**
  Make sure Tab and Shift+Tab cycle focus within a modal and nowhere else.
- [ ] **Live Region Updates**
  Use `aria-live` to announce changes in a status area (e.g. “Form submitted!”).

💡 Section 33 Challenge: *“Accessible Keyboard-Only Menu”*

Build a dropdown menu that:

- Opens with keyboard (Enter or Space), closes with Escape
- Can be navigated using ArrowUp / ArrowDown keys
- Wraps around when reaching top/bottom
- Uses correct roles (`menu`, `menuitem`) and `aria-expanded`, `aria-controls`
- Visually shows current focus and selection with highlighting
- Works with screen readers

🧠 Requires: semantic HTML, ARIA roles, keyboard events, accessibility best practices

✅ Section 34: Advanced Component Composition & Slot Use

🧠 Focus: mastering `<slot>`, named slots, fallback content, component composition, and reusable UI structures

- ▢ **Basic Slot Usage**
  Create a `Card` component that accepts content via a default `<slot>`.
- ▢ **Named Slots**
  Add named slots (`<slot name="header">`, `<slot name="footer">`) to the `Card` component.
- ▢ **Fallback Content for Slots**
  Provide default text or markup when nothing is passed into a slot.
- ▢ **Slot Props (let:)**
  Pass data from the parent component into the child slot using `let:` syntax.
- ▢ **Scoped Slots for Rendering Lists**
  Use a `ListRenderer` component that uses slot props to render a list of data passed in.
- ▢ **Component-as-Layout**
  Create a reusable `PageLayout` component that accepts header, sidebar, and main content via slots.
- ▢ **Nesting Slots with Components**
  Nest components inside each other with multiple layers of slot content.
- ▢ **Conditional Slot Rendering**
  Render different slots conditionally depending on props passed to the component.
- ▢ **Dynamic Component Renderer**
  Build a component that takes another component and renders it dynamically via `<svelte:component>`.
- ▢ **Slot Styling from Parent**
  Style slotted content from outside the component using CSS selectors carefully.

💡 Section 34 Challenge: *"Customizable Toast System with Slots"*

Create a `Toast` component that:

- Accepts message content via default slot
- Accepts custom icons and action buttons via named slots
- Displays fallback "info" icon and close button if none is provided

- Can be reused in different ways across the app (error, warning, success)
- Is wrapped in a `ToastManager` that manages the stack of multiple `Toast` instances

🧠 Requires: default/named slots, fallback handling, reusable UI design with slot props and composition

✅ Section 35: Complex UI State and Reactive Patterns

🧠 Focus: deeply understanding reactivity, derived state, reactive statements, and efficient UI updates in Svelte

- ▢ **Basic Reactive Statement (`$:`)**
  Use a reactive declaration to compute a full name from two input fields.
- ▢ **Dependent Reactive Statements**
  Chain reactive variables where one depends on another (`$: b = a * 2` , etc).
- ▢ **Avoiding Infinite Reactive Loops**
  Demonstrate a loop that causes runaway reactivity, and fix it with proper conditions.
- ▢ **Reactive Object Updates (shallow vs deep)**
  Show that updating a nested object key doesn't trigger reactivity—use spread to fix it.
- ▢ **Derived State from Store**
  Use `$store` and a `$:` block to derive computed UI state (e.g. active item name).
- ▢ **Reactive Animation Trigger**
  Use a reactive statement to automatically animate a value when another value changes.
- ▢ **Reactive Array Filtering**
  Type into an input box and have a list automatically filter in real-time using a `$:` .
- ▢ **Chained `if` Conditions Reactively**
  Use several conditions in a chain of `$:` statements and watch their behavior.
- ▢ **Reactive Class Binding**
  Use a reactive expression to dynamically assign CSS classes (e.g. `"error"` , `"valid"`).
- ▢ **Reactive Counter with Side Effects**
  Add a reactive block that logs to console when a counter changes and crosses a threshold.

💡 Section 35 Challenge: *“Reactive Dashboard Cards”*

Create three UI cards that:

- Display a score, level, and rank—all computed reactively from one input (`points`)
- Dynamically update colors and badges based on current rank
- If `points` go over 1000, show a celebratory animation using a reactive trigger
- Allow typing into an input field to set points, and instantly update all derived values

🧠 Requires: derived state, nested reactivity, class binding, and conditional side effects

✅ Section 36: Keyboard & Input Interaction

🧠 Focus: building interactive elements that respond to keyboard input, focus states, and accessibility patterns

☐ **Keypress Alert**
Listen to `keydown` on the document and alert which key was pressed.

☐ **Arrow Key Movement**
Move a box on screen using arrow key events.

☐ **Keyboard Shortcuts**
Press “Ctrl + K” to show a hidden panel (like a search box or command bar).

☐ **Focus Management**
Programmatically set focus to an input field when a button is clicked.

☐ **Tab Index Order Demo**
Create a form and manipulate the `tabindex` to show keyboard navigation order.

☐ **Escape to Close Modal**
Build a modal that closes when the Escape key is pressed.

☐ **Input Validation on Enter**
On pressing Enter in a field, validate the input and show feedback.

☐ **Custom Select with Arrow Keys**
Create a dropdown list navigable by Up/Down arrows and Enter to select.

☐ **Toggle Class with Spacebar**
Press spacebar on a box to toggle its color (like a “checkbox” effect).

▢ **Simulate Text Adventure Navigation**
Use keys (WASD or arrows) to "navigate" through a grid of rooms and display position.

💡 Section 36 Challenge: *"Keyboard-Controlled Notification System"*

Build a system that:

- Listens for number keys 1–5, and shows a corresponding notification
- Pressing `Backspace` clears all notifications
- Pressing `Shift+N` toggles the entire notification center visibility
- Pressing `Escape` hides all notifications (but doesn't delete them)
- Every notification should be focusable and deletable with `Delete` key

🧠 Requires: multiple key events, state manipulation via keyboard only, conditional rendering, accessibility principles

✅ Section 37: Animating with State Machines & Timelines

🧠 Focus: managing animations using controlled logic (like simple state machines), delays, and timelines

▢ **Two-State Animation Toggle**
Animate a box between "open" and "closed" states with a toggle button.

▢ **Simple Visual State Machine**
Build a light that switches between "off", "dim", and "bright" with visual changes.

▢ **Animation Based on Time**
Show an object that fades in slowly, then moves after 2 seconds using `setTimeout`.

▢ **Manual Step Timeline Animation**
Use a sequence of `setTimeout` calls to animate one item in three steps (e.g., fade → grow → bounce).

▢ **Chain Two CSS Animations by State**
Animate a card flipping and then glowing—only when the flip is completed.

▢ **Replay Animation Button**
Create a way to trigger and restart an animation by resetting its key.

▢ **Color Cycle State Machine**
Click a box to cycle through three background colors: red → yellow → green → red…

▢ **Show Countdown Animation**
Display a countdown from 5 to 0 with a visual shrink or color transition at each step.

▢ **Trigger Animation on Scroll into View**
Animate a section when it enters the viewport (using `on:window:scroll`).

▢ **Pulse Animation with Pause/Resume**
Make a pulsing effect that can be paused/resumed with a button toggle.

💡 Section 37 Challenge: *“Four-State Wizard Card with Transitions”*

Build a “wizard card” that:

- Has 4 states: `start`, `info`, `confirm`, `done`, all controlled by “Next” and “Back” buttons
- Each state changes the card’s style and content with unique animations (slide, fade, flip…)
- Moving backward uses reverse animations
- Reset button brings the wizard back to the `start` state with a rewind effect

🧠 Requires: sequencing transitions, multiple states, animation chaining, and logic-controlled view flow

✅ Section 38: Advanced Component Patterns

🧠 Focus: reusable patterns, advanced prop handling, slots, context, and composition techniques

▢ **Slot-Powered Alert Box**
Create an `<AlertBox>` component with a default slot for message content.

▢ **Named Slots for Layout**
Build a card component with `header`, `body`, and `footer` named slots.

▢ **Default Fallback Slot Content**
Show a default message if no slot content is passed in.

▢ **Component with Forwarded Events**
Wrap a native button component and forward `click` and `mouseover` events.

▢ **Wrapper Component with Prop Passing**
Make a styled container that passes all received props to a child `<input>`.

▢ **Component Composition with Children**
Build a `<Panel>` component that takes a `<Panel.Title>` and `<Panel.Body>` inside it.

▢ **Using `$$restProps`**
Create a `<FancyButton>` component that supports any HTML attributes passed to it.

▢ **Context API: Provide/Consume**
Share a theme (dark/light) across unrelated components using context.

▢ **Dynamic Slot Rendering**
Build a `<TabGroup>` that renders different slots depending on the active tab.

▢ **Reusable Form Field with Props and Slots**
Create a `<FormField>` component that takes a label and displays an input slot.

💡 Section 38 Challenge: *"Fully Slotted Modal Dialog Component"*

Build a `<Modal>` component with:

- A dark background overlay
- Named slots for `title`, `body`, and `actions`
- Ability to close the modal via an `x` button or ESC key
- Prop to control visibility
- Slot fallback content if nothing is passed
- Bonus: use `createEventDispatcher()` to emit `on:close`

🧠 Requires: slot usage, event handling, context awareness, conditional rendering, and component design

✅ Section 39: Realistic UI Components

🧠 Focus: building real-world UI widgets using all your Svelte knowledge (state, props, slots, events, transitions, styling)

▢ **Toggle Switch Component**
Create a stylized toggle switch that changes appearance and state when clicked.

- [ ] **Accordion Section**
  Build an accordion with multiple collapsible panels.
- [ ] **Dropdown Menu with Click Outside to Close**
  Create a dropdown that closes if you click outside it.
- [ ] **Star Rating Component**
  Show 5 stars that highlight based on hover and click.
- [ ] **Tab Switcher with Dynamic Content**
  Build tabs that switch visible content based on selected tab.
- [ ] **Collapsible Sidebar**
  A sidebar that can collapse/expand with animation.
- [ ] **Tooltip on Hover**
  Show a small tooltip box when hovering over an icon or text.
- [ ] **Carousel with Manual Controls**
  Build a simple image carousel with next/prev buttons.
- [ ] **Auto-Expanding Textarea**
  A textarea that grows vertically as the user types more lines.
- [ ] **Notification Toast System**
  Create a system that shows dismissible temporary toast messages.

💡 Section 39 Challenge: *“Interactive Dashboard Widgets”*

Build a set of three mini-widgets for a dashboard, each doing one of the following:

- A live clock that updates every second
- A collapsible weather panel with fake temperature data
- A star rating widget that remembers your rating using `localStorage`

Make sure:

- Widgets are resizable using CSS
- Each is its own component with state and styling
- All can be placed side-by-side using Flexbox or Grid

🧠 Requires: layout, component composition, interaction, event handling, local persistence

✅ Section 40: Local Storage and Persistence

🧠 Focus: use `localStorage` and Svelte to persist UI state across browser sessions (no backend needed)

- ▢ **Counter with Remembered Value**
  Build a counter that stores its current value in `localStorage` so it stays the same after refresh.
- ▢ **Theme Toggle (Light/Dark) with Persistence**
  Create a theme toggle switch that remembers your preference in `localStorage`.
- ▢ **Saved Username Input**
  An input field where a user can type their name, and it autofills next time they visit.
- ▢ **Persisted Todo List**
  A basic todo list whose items persist across refreshes using `localStorage`.
- ▢ **Remembering Tab Selection**
  A tab interface where the last opened tab is restored after reload.
- ▢ **Modal Visibility Tracker**
  Show a "first-time user" modal only once, using `localStorage` to track if it's been dismissed.
- ▢ **Favorite Items Highlighter**
  Allow clicking items in a list to "favorite" them, and remember those selections persistently.
- ▢ **Local Storage Store Wrapper**
  Build a custom Svelte store that automatically syncs its values to/from `localStorage`.
- ▢ **Data Expiry Simulation**
  Save a message to localStorage that expires after 10 seconds using timestamps and logic.
- ▢ **Syncing Multiple Components with Same Local Storage Key**
  Two separate components use and update the same persisted data.

💡 Section 40 Challenge: *“Local Storage Power App”*

Build a mini productivity panel with:

- A todo list
- A theme toggle
- A persistent name input

All three parts must:

- Load saved data on startup
- Save any updates automatically
- Use one or more custom stores that interface with `localStorage` under the hood
- Work even if placed in totally different components

🧠 Requires: store creation, component communication, lifecycle hooks, DOM persistence

✅ Section 41: Accessible and Responsive Design

🧠 Focus: Learn how to make your UI accessible and responsive with minimal setup using only Svelte and CSS

- [ ] **Keyboard-Focusable Buttons**
  Ensure custom buttons (like divs styled as buttons) are focusable and work with Enter/Space keys.
- [ ] **Alt Text for Images**
  Display an image and provide meaningful `alt` text—explore how screen readers interpret it.
- [ ] **Label-Input Associations**
  Make sure all inputs are properly labeled for accessibility using `<label for="id">`.
- [ ] **ARIA Live Region Alert**
  Build a notification banner that announces messages via `aria-live`.
- [ ] **Accessible Modal with Keyboard Controls**
  Build a modal that:
  - traps focus,
  - can be closed with `Esc`,
  - and has proper ARIA roles.
- [ ] **Keyboard-Navigable Tab Interface**
  Create a tab component that allows switching tabs via keyboard arrows and Enter key.
- [ ] **Visually Hidden Instructions**
  Add a screen-reader-only label or help text using the `visually-hidden` CSS pattern.
- [ ] **Responsive Navigation Menu**
  Build a mobile-first nav menu that toggles with a hamburger icon on small screens.

▢ **Responsive Grid Gallery**
Build a simple image grid that adapts layout based on screen width using CSS Grid.

▢ **Responsive Component with `window.innerWidth`**
Show or hide parts of a component dynamically using the window size (tracked via reactive variables).

💡 Section 41 Challenge: *"Build an Accessible Survey"*

Design a 3-question mini survey that:

- Uses labeled form inputs
- Has ARIA attributes on all question containers
- Is fully keyboard-navigable
- Works well on both large and small screens
- Speaks the result of submission using a live region

🧠 Requires: form input mastery, ARIA usage, keyboard control, responsive layout, dynamic DOM manipulation

✅ Section 42: Advanced Animations and Transitions

🧠 Focus: Master the Svelte animation and transition system, including custom effects, staggered animations, and element coordination.

▢ **Staggered List Appearance**
Use `transition:fade` with a loop and delay to animate list items one-by-one.

▢ **Crossfade Between Elements**
Use `svelte/transition`'s `crossfade` to animate switching elements between two containers.

▢ **Custom Transition Function**
Create a custom transition that slides in and rotates an element on entry.

▢ **Parameterizing Transitions**
Build a reusable component with `fade` or `slide` where the duration and delay are props.

▢ **Simulated Loading Animation**
Use a loop of rectangles or dots that animate while "loading," using `@keyframes` and reactive state.

- ▢ **Slide-In Sidebar with Easing**
  Build a sidebar that animates in and out using `transition:slide` with a custom easing curve.
- ▢ **Spring-Driven Animation**
  Use Svelte’s `spring` store to move an object toward a target position with bounce.
- ▢ **Scale + Rotate Interactive Box**
  On hover or click, animate a box that scales and rotates simultaneously with `transition:scale`.
- ▢ **Chart Bar Grow Animation**
  Animate a bar chart where each bar grows from 0 to its height using `animate:flip`.
- ▢ **Presence Transition (In/Out Hooks)**
  Animate when an element both enters and leaves the DOM using `in:` and `out:` directives with different effects.

💡 Section 42 Challenge: *“Build a Coordinated Card Animator”*

Create 4 cards that:

- Appear one after the other with a staggered delay
- Can be clicked to remove themselves with a smooth collapse animation
- When one is removed, the others animate their movement using `animate:flip`
- Include a “Restore All” button that animates the cards back in

🧠 Requires: staggered transitions, animate:flip, keyed blocks, entry/exit coordination

✅ Section 43: Custom Stores and Derived State Logic

🧠 Focus: Go beyond writable stores — learn how to create custom stores, derived stores, and logic-based reactive state management.

- ▢ **Write a Custom Readable Store**
  Create a `clock` store that updates every second with the current time.
- ▢ **Custom Writable with Validation**
  Build a store for a numeric input that only allows even numbers.
- ▢ **Derived Store for Total Price**
  Combine a store of product prices and quantities into a derived store that gives a total.

- ▢ **Toggle Store (Encapsulated)**
  Write a `createToggle()` store that exposes `on`, `off`, and `toggle` methods.
- ▢ **Dynamic Greeting from Time Store**
  Use a derived store to display a greeting like "Good Morning", "Good Evening", etc., based on the `clock` store.
- ▢ **Undo-Redo History Store**
  Create a store with undo/redo functionality, storing a stack of previous states.
- ▢ **Poll Result Aggregator (Local)**
  Use derived stores to compute percentages from a local store of vote counts.
- ▢ **Store with Side Effects**
  Create a custom store that logs to the console whenever the value changes.
- ▢ **Chained Derived Stores**
  Combine multiple stores — e.g., `user`, `theme`, and `clock` — to derive a dashboard message.
- ▢ **Store-Driven Animation Trigger**
  Use a store to trigger a complex animation sequence when its value changes.

💡 Section 43 Challenge: *"Build a Reactive Score Tracker with Undo/Redo"*

Create a scoreboard UI where:

- Two teams can increment or decrement their score
- A "history" of all changes is stored in a custom undo/redo store
- The total score and leading team are shown using a derived store
- A reset button clears everything and smoothly animates the change

🧠 Requires: custom writable, derived store, encapsulated logic, UI+store sync, animation triggers

✅ Section 44: Simulated Backend Integration (Mock Data + Local Interactivity)

🧠 Focus: Simulate common backend interactions using static/mocked data to master asynchronous UI flows and frontend logic without needing a real server.

- ▢ **Simulate a Loading Spinner**
  Create a button that simulates a data fetch (2s delay) and shows a loading spinner.
- ▢ **Fake API Call on Mount**
  Use `onMount` to simulate fetching a user's profile data and displaying it.
- ▢ **Random Quote Generator**
  Load a random quote from a hardcoded array when a button is clicked.
- ▢ **Delayed Poll Results**
  Simulate fetching and displaying poll results after a delay with a fade-in animation.
- ▢ **Comment Fetch with "Retry"**
  Show an error message and "Retry" button if a simulated fetch fails randomly.
- ▢ **Paginated Data (Mocked)**
  Use buttons to paginate through a locally defined list of mock data.
- ▢ **Optimistic Voting UI**
  Add a vote button that instantly updates the vote count and "commits" it after a delay.
- ▢ **Loading Skeleton UI**
  Create a skeleton screen that is shown while mock data is "loading."
- ▢ **Simulated Search Field**
  Type in a box to filter a list of mock search results (with delay to simulate latency).
- ▢ **Fake Auth Login Flow**
  Build a login form that accepts only a hardcoded username/password combo and shows a dashboard on success.

💡 Section 44 Challenge: *"Mock a Social Feed with Refresh"*

Build a fake "social media feed" with the following features:

- Loads a list of mock posts on mount (simulated API)
- Clicking "Refresh" reloads new mock posts with a spinner
- Includes optimistic "like" buttons for each post
- Shows a toast notification if a mock network error is randomly triggered

🧠 Requires: async logic, state-driven UI, mock error handling, skeleton UI, and simulated interaction

✅ Section 45: Authentication UI Patterns (Frontend-Only, No Backend)

> 🧠 Focus: Practice common login and user flow patterns entirely in the frontend using mock data and local state.

- [ ] **Basic Login Form with Validation**
  Build a login form that validates the presence of username and password fields.
- [ ] **"Logged In" State Toggle**
  Simulate login/logout functionality by toggling a `loggedIn` boolean and showing conditional UI.
- [ ] **Login Form with Mock Credentials**
  Accept only a specific hardcoded username/password (e.g., `test / pass123`) and show a welcome message if correct.
- [ ] **Logout Button that Clears State**
  Add a logout button that resets the user state and brings the UI back to the login screen.
- [ ] **User Dashboard View Switch**
  Show different views depending on whether a user is logged in or not.
- [ ] **Signup Form with Field Validation**
  Create a signup form that checks for valid email, minimum password length, and confirmed password.
- [ ] **Redirect After Login (Simulated)**
  Simulate a route switch by conditionally rendering a dashboard after "login".
- [ ] **Remember Me (LocalStorage)**
  Store a "remember me" toggle that preserves login state across page reloads using `localStorage`.
- [ ] **Error Messages for Failed Login**
  Show appropriate messages when login fails due to missing/invalid input.
- [ ] **Fake Auth Delay with Loading State**
  Add a 2-second delay before login "succeeds", showing a spinner or loading message.

💡 Section 45 Challenge: *“Simulate a Full Auth Experience”*

> In the REPL, build a fully mock-authenticated app that includes:
>
> - A login form that accepts one hardcoded user (`sveltefan / learn123`)
> - Proper validation and error messages

- A dashboard visible only when logged in
- Logout functionality
- Optionally: a "remember me" checkbox that persists across reloads

🧠 Requires: state management, conditional rendering, basic form validation, and localStorage integration

✅ Section 46: Advanced Interactive UI (Polls, Voting, Charts)

🧠 Focus: Learn to build complex, reactive, and engaging interactive widgets that resemble real-world features—without any backend.

▢ **Like Button with Counter and Heart Icon**
Create a heart icon that toggles red on click and updates a like count visually.

▢ **Thumbs Up/Down Voting Widget**
Build a pair of buttons for upvote/downvote with visual feedback and totals.

▢ **Multiple Choice Poll UI**
Show a poll question with 3–4 answer buttons; allow one selection and visually highlight the choice.

▢ **Poll with Result Percentages**
Add a second view to the poll showing percentage results once a choice is made.

▢ **Poll Bar Graph Using Flexbox or CSS Widths**
Visualize vote results using colored bars that grow to show percentages.

▢ **Disable Voting After Selection**
Prevent multiple votes by disabling buttons after user has voted.

▢ **Animated Vote Reveal**
Use a fade or grow transition when revealing poll results.

▢ **Pie Chart-Like Display (CSS Only)**
Create a circular pie-like chart to visualize poll results using simple CSS tricks (no libs).

▢ **“You Voted” Badge with Conditional Rendering**
After submitting a vote, show a badge or label indicating the user has voted.

▢ **Emoji Reaction Strip**
Let users react with one of several emojis, each showing a total count that updates on click.

💡 Section 46 Challenge: *“Build a Multi-Option Poll with Animated Results”*

> Create an interactive poll component with:
>
> - 4 answer options
> - Ability to vote once
> - Result percentages shown in colored bars (animated width)
> - A badge showing “Thanks for voting!” after selection
> - Reset button to start over (resets vote and UI)

🧠 Requires: state handling, transitions, conditional rendering, interactive UI design, CSS-based visualization

✅ Section 47: Capstone Challenge Exercises (Disconnected Mastery Tasks)

> 🧠 Focus: These are standalone, advanced challenges that test your mastery of all Svelte frontend topics you've learned so far. Each can be done independently in the Svelte REPL.

- [ ] **Build a Toggleable Dark/Light Theme Switcher**
  Store the theme preference in `localStorage` and use reactive classes to switch themes across components.
- [ ] **Create a Custom Notification System with Dismiss Buttons**
  Notifications appear in a stack with fade-in/out and can be dismissed individually.
- [ ] **Simulate a Realtime Chat Feed**
  Use a store and `setInterval` to simulate new incoming messages, then animate them sliding into view.
- [ ] **Interactive Star Rating Component (1–5 Stars)**
  Display 5 stars, allow selection, hover preview, and store the selected rating.
- [ ] **Emoji Voting Panel with Live Percentages**
  Display a list of emoji options, show their % share of total votes using animated bars or widths.
- [ ] **Custom Dropdown Component with Keyboard Navigation**
  Fully accessible dropdown menu: arrow key navigation, enter/select support, escape to close.
- [ ] **Build a Resizable Panel (Drag-to-Resize)**
  Allow the user to click and drag a divider to resize two sections of a panel horizontally.

- ▢ **Mini Drawing Canvas Using Mouse Events**
  Use mouse events to create a small drawing tool on an HTML `<canvas>` or div grid.
- ▢ **Auto-Save Notes App Using localStorage and Stores**
  Build a textarea with autosave feature every few seconds, using Svelte stores and localStorage.
- ▢ **Visual Timer with Pause/Reset Controls**
  Make a visual countdown timer with a progress bar and buttons to pause, resume, and reset.

💡 Final Challenge: *“Build a Fully Interactive Voting Poll Gallery”*

In a single Svelte REPL app, implement:

- A list of 3–4 disconnected poll components (each with its own question)
- Each poll allows only one vote per user (per poll)
- Results appear after voting, animated
- Use `localStorage` to remember votes across page reloads
- A button to clear all votes and reset all polls

🧠 This challenge brings together everything: interactivity, state handling, local storage, animation, conditional rendering, and modular design — without needing any backend or shared state.

🎉 **Congratulations!** You now have full frontend Svelte mastery for interactive, animated, media-rich UI — and you’re ready to build your personal digital vlog site from scratch.