Alpine.js Course

Flavio Copes

Updated August 3, 2026

Contents

Add focused browser interactions with Alpine.js while keeping HTML, accessibility, progressive enhancement, and state ownership clear.

What you'll learn: Build an accessible issue board with local state, forms, dynamic lists, reusable components, persistence, and an explicit HTMX boundary.

Local state and directives

Choose Alpine-sized problems and bind local content, state, and visibility.

Choose Alpine-sized problems

Recognize interactions that need local browser state without a full component framework.

Before choosing a tool or writing more code, make the requirement concrete. The issue filter, disclosure, and edit form are local interactions around server-rendered content.

Alpine works best when HTML remains the center and JavaScript adds behavior close to the element that owns it. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to move routing, server authority, and a large shared data model into inline expressions. The happy path may still work, which makes the mistake easy to miss. write the no-JavaScript behavior first and give Alpine only the state needed for the local interaction. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Add a small local behavior to useful HTML without turning the whole page into a client application. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

List the board interactions and classify each as server-owned, URL-owned, page-local, or component-local. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Create an x-data scope

Declare local state and understand which descendants can read or change it.

Start from the behavior you can observe. The filter panel owns `query` and `status`, while each issue row owns only its own expanded state.

`x-data` creates an Alpine scope; nested scopes can shadow values, so placement establishes ownership. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to put one giant x-data object on body so every element can mutate every value. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to place state at the smallest common ancestor that needs it and name nested values to avoid accidental shadowing. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Add a small local behavior to useful HTML without turning the whole page into a client application. Record enough evidence that another developer can repeat the result without relying on your memory.

Move a value through three possible scope locations and observe which controls can access it. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Bind text, attributes, and visibility

Use x-text, x-bind, x-show, and x-cloak according to whether content, attributes, or display should change.

The board updates a result count, binds `aria-expanded`, hides a panel, and prevents pre-initialization flicker. That contrast gives us something useful to test instead of a rule to memorize.

Different directives change different parts of the DOM; choosing the right one keeps the HTML readable and predictable. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can use x-html for plain text or hide inaccessible state only with CSS and still get one successful demo. Push past the demo. prefer text binding for text, bind semantic attributes with state, and coordinate visual and accessibility state, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Add a small local behavior to useful HTML without turning the whole page into a client application. Save the before-and-after evidence now; it will make the final project review much more honest.

Build one disclosure and inspect its text, hidden state, and accessibility attributes before and after interaction. Save the evidence, then explain which requirement would force you to choose a different design.

Inspect Alpine state

Use browser tools and small expressions to trace scope, initialization, and DOM changes.

A temporary watcher shows exactly when filters change instead of guessing which directive failed. This is a small part of an accessible issue board that filters, edits, persists, and progressively enhances server-rendered HTML, but the boundary it creates affects everything that follows.

Alpine is still JavaScript manipulating the DOM, so ordinary console, elements, and event inspection remain useful. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to add more directives until the interface starts working by accident. It makes later failures harder to locate. Instead, reduce the component, inspect the owning scope, log transitions, and verify the resulting DOM state. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Add a small local behavior to useful HTML without turning the whole page into a client application. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Break one binding name intentionally, diagnose it from the scope and console, then remove the debugging code. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: local state and directives

Check progressive enhancement, x-data scope, expressions, text, attributes, visibility, and DevTools inspection.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Events and forms

Handle events, native controls, derived state, and asynchronous submission.

Handle events with intent

Use x-on and modifiers for a reason instead of attaching broad listeners everywhere.

Before choosing a tool or writing more code, make the requirement concrete. Escape closes the editor, click outside dismisses it, and the form—not an arbitrary button—owns submission.

Event modifiers encode propagation, default behavior, keyboard filters, timing, and listener targets. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to add prevent and stop to every handler until unexpected behavior disappears. The happy path may still work, which makes the mistake easy to miss. start from the browser event flow and add only the modifier required by the interaction contract. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Build a keyboard-usable issue editor whose state, validation, and submission lifecycle remain explicit. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Trace click, submit, Escape, and outside-click events and explain every modifier retained. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Bind form controls

Use x-model with text, checkbox, radio, select, and modifiers while preserving labels and native form behavior.

Start from the behavior you can observe. The issue editor keeps labeled native controls and can submit normally if the enhanced request path fails.

`x-model` synchronizes control values with state, but accessible HTML and browser submission semantics still matter. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to replace real controls with clickable div elements because binding looks shorter. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to bind native labeled controls, choose value types deliberately, and keep the form action meaningful. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Build a keyboard-usable issue editor whose state, validation, and submission lifecycle remain explicit. Record enough evidence that another developer can repeat the result without relying on your memory.

Bind all filter control types and inspect the state values, submitted values, and behavior without JavaScript. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Derive state instead of copying it

Compute validity, counts, and filtered views from source state instead of synchronizing duplicate fields.

The remaining-character count comes from the title, and the visible issue list comes from issues plus filters. That contrast gives us something useful to test instead of a rule to memorize.

Duplicated state drifts; a getter or expression can derive values from the authoritative inputs. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can update the source and several cached copies in separate event handlers and still get one successful demo. Push past the demo. keep one source of truth and derive secondary values at the point of use or in a named getter, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Build a keyboard-usable issue editor whose state, validation, and submission lifecycle remain explicit. Save the before-and-after evidence now; it will make the final project review much more honest.

Introduce a duplicated count, demonstrate how it drifts, then replace it with derived state. Save the evidence, then explain which requirement would force you to choose a different design.

Model async form states

Represent idle, submitting, success, and failure without allowing duplicate work or hiding errors.

The save button disables during one request, errors are announced, and the form returns to a usable state after failure. This is a small part of an accessible issue board that filters, edits, persists, and progressively enhances server-rendered HTML, but the boundary it creates affects everything that follows.

An enhanced form needs an explicit lifecycle and a final cleanup path even when fetch throws or the server rejects input. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to set a loading boolean before fetch and forget to clear it on exceptions. It makes later failures harder to locate. Instead, use a try/catch/finally flow, keep server errors visible, and make repeated submission behavior intentional. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Build a keyboard-usable issue editor whose state, validation, and submission lifecycle remain explicit. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Simulate slow success, validation failure, network failure, and double click; verify controls and messages each time. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: events and forms

Check event modifiers, keyboard behavior, x-model, derived state, validation, async status, and cleanup.

Answer each question before continuing. Read the explanation after every answer, including the

ones you get right.

Take this interactive quiz online

Lists, transitions, and the DOM

Preserve identity and focus through lists, conditional DOM, and transitions.

Render keyed lists

Use template, x-for, and stable keys so DOM identity follows the underlying record.

Before choosing a tool or writing more code, make the requirement concrete. Issue ids remain stable while titles and positions change, so edit state stays attached to the correct issue.

A stable key lets Alpine associate an existing DOM subtree with the same logical item after filtering or reordering. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to use the array index as identity for records that can be inserted, deleted, or sorted. The happy path may still work, which makes the mistake easy to miss. bind a unique stable record id as the key and test insertions, removals, and reordering. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Render dynamic issue rows without losing identity, focus, or respect for reduced-motion preferences. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Edit one row, reorder the issues, and verify focus and local state remain on that issue. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Choose x-show or x-if

Decide whether hidden DOM should remain mounted or be created and destroyed.

Start from the behavior you can observe. A frequently opened filter panel stays mounted, while an expensive one-time modal can be inserted only when needed.

`x-show` toggles display and preserves the subtree; `x-if` changes whether the subtree exists at all. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to swap the directives without considering initialization, focus, form values, or third-party widgets. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to choose from lifecycle and state requirements, then test repeated open and close behavior. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Render dynamic issue rows without losing identity, focus, or respect for reduced-motion preferences. Record enough evidence that another developer can repeat the result without relying on your memory.

Implement the editor both ways and record the differences in DOM, state, focus, and initialization. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Add purposeful transitions

Use motion to clarify state change while keeping duration modest and honoring reduced motion.

The issue panel fades briefly, and the stylesheet removes nonessential motion for reduced-motion preferences. That contrast gives us something useful to test instead of a rule to memorize.

A transition should help the user understand what appeared or disappeared, not delay access to information. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can animate every property and force long movement before content becomes usable and still get one successful demo. Push past the demo. transition cheap visual properties, keep timing short, and provide an equivalent low-motion experience, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Render dynamic issue rows without losing identity, focus, or respect for reduced-motion preferences. Save the before-and-after evidence now; it will make the final project review much more honest.

Test the interaction with normal and reduced-motion settings and verify focus never waits for animation. Save the evidence, then explain which requirement would force you to choose a different design.

Coordinate DOM updates and focus

Use refs and nextTick when code must act after Alpine has updated the DOM.

Opening the editor inserts or reveals it, then focus moves to the title through a named ref. This is a small part of an accessible issue board that filters, edits, persists, and progressively enhances server-rendered HTML, but the boundary it creates affects everything that follows.

State can change before the corresponding element exists or becomes visible; nextTick schedules work after the update. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to use arbitrary timers to guess when rendering has finished. It makes later failures harder to locate. Instead, change state, wait for Alpine's DOM update, and focus the intended stable reference. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Render dynamic issue rows without losing identity, focus, or respect for reduced-motion preferences. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Open and close the editor rapidly and prove focus lands correctly without time-based sleeps. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: lists, transitions, and the dom

Check x-for keys, x-if versus x-show, transitions, focus after DOM changes, refs, and nextTick.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Reusable Alpine

Extract data components, use stores sparingly, synchronize boundaries, and evaluate plugins.

Extract Alpine.data components

Move a repeated or complex component definition out of markup while keeping its API small.

Before choosing a tool or writing more code, make the requirement concrete. Every issue editor uses the same focused factory with explicit initial values rather than copying a large inline object.

`Alpine.data()` gives a component a named factory, initialization, methods, getters, and a testable JavaScript home. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to extract every two-line component and create a registry of abstractions nobody can navigate. The happy path may still work, which makes the mistake easy to miss. extract when behavior repeats or markup becomes hard to read, and pass only the initial inputs the component owns. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Extract reuse only where it clarifies the board, keeping local components local and global state rare. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Build two editor instances with different starting data and prove their state remains independent. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Use global stores sparingly

Reserve Alpine.store for genuinely shared browser state and keep component details local.

Start from the behavior you can observe. A global connection status may belong in a store; one issue row's menu state does not.

A store expands who can read and mutate a value, so it should represent page-wide state with a clear mutation model. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to move state to a store merely because two expressions reference it. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to identify the true owners and mutations before promoting state beyond component scope. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Extract reuse only where it clarifies the board, keeping local components local and global state rare. Record enough evidence that another developer can repeat the result without relying on your memory.

Audit every state value on the board and justify the few that need to be global. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Initialize, watch, and clean up

Use init and watchers for boundary synchronization without creating loops or forgotten external resources.

The filter initializes from the URL and writes deliberate changes back without watching a second copy of the same state. That contrast gives us something useful to test instead of a rule to memorize.

Initialization and watchers are appropriate for URL, storage, or third-party synchronization, but derived UI state should remain derived. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can watch values that update each other and leak listeners or timers across component lifecycles and still get one successful demo. Push past the demo. keep watchers one-directional, avoid redundant state, and return or invoke cleanup for external resources, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Extract reuse only where it clarifies the board, keeping local components local and global state rare. Save the before-and-after evidence now; it will make the final project review much more honest.

Create a URL-sync watcher, navigate backward, and verify there is no update loop or duplicate listener. Save the evidence, then explain which requirement would force you to choose a different design.

Evaluate plugins and CSP

Add official plugins or a CSP-compatible build only when the deployment and feature justify them.

The board adopts a focus helper only after native focus handling becomes genuinely complex and verifies the deployed CSP. This is a small part of an accessible issue board that filters, edits, persists, and progressively enhances server-rendered HTML, but the boundary it creates affects everything that follows.

Plugins extend Alpine with focused capabilities; stricter Content Security Policy can change how expressions are evaluated. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to install plugins preemptively or weaken CSP just to preserve convenient inline expressions. It makes later failures harder to locate. Instead, measure the need, read the plugin and CSP constraints, and test the exact production build and policy. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Extract reuse only where it clarifies the board, keeping local components local and global state rare. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

List the code and policy impact of one plugin and decide whether the native implementation remains simpler. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: reusable alpine

Check Alpine.data, stores, initialization, watchers, cleanup, plugins, and CSP-aware boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Accessible progressive UI

Keep semantics in sync, persist deliberately, coordinate HTMX, and test failure.

Keep semantics in sync

Update aria-expanded, hidden state, labels, errors, and focus together with visual state.

Before choosing a tool or writing more code, make the requirement concrete. The edit disclosure is a button controlling a named region and communicates its expanded state on every transition.

Reactive visibility is only one part of an accessible interaction; name, role, value, state, and focus must agree. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to change classes while leaving accessibility attributes and focus on stale elements. The happy path may still work, which makes the mistake easy to miss. start from the native interaction pattern and bind every relevant semantic state from the same source value. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Finish the issue board as a progressively enhanced interface that remains understandable with keyboard, assistive technology, slow networks, and JavaScript failure. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Inspect the accessibility tree and use the complete board without a mouse. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Persist only user-owned state

Store preferences and drafts deliberately without turning browser storage into an accidental database.

Start from the behavior you can observe. The filter preference can live locally; authoritative issue records still come from the server.

Persistence needs ownership, lifetime, validation, versioning, and a recovery path just like other state. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to serialize the whole component and trust it forever on every future page version. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to choose individual durable values, validate reads, version the format, and provide reset behavior. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Finish the issue board as a progressively enhanced interface that remains understandable with keyboard, assistive technology, slow networks, and JavaScript failure. Record enough evidence that another developer can repeat the result without relying on your memory.

Corrupt the saved filter, upgrade its shape, clear storage, and verify the board still starts safely. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Coordinate Alpine and HTMX

Let HTMX own server communication and HTML replacement while Alpine owns local interaction state.

HTMX swaps the issue list after a server action; Alpine controls the local filter panel and new rows initialize from their markup. That contrast gives us something useful to test instead of a rule to memorize.

The tools work well together when their ownership boundary is explicit and replaced DOM is expected to initialize again. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can let both tools issue the same request or mutate the same DOM subtree unpredictably and still get one successful demo. Push past the demo. assign network and DOM ownership, listen to lifecycle events only where coordination is necessary, and test swaps, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Finish the issue board as a progressively enhanced interface that remains understandable with keyboard, assistive technology, slow networks, and JavaScript failure. Save the before-and-after evidence now; it will make the final project review much more honest.

Replace the issue list while an editor is open and decide intentionally which state survives. Save the evidence, then explain which requirement would force you to choose a different design.

Finish and test the board

Run the final interface through keyboard, screen reader, JavaScript-off, slow-network, invalid-state, and replacement scenarios.

Core issue reading and submission work without Alpine; enhancement adds filtering, feedback, and convenience without blocking the task. This is a small part of an accessible issue board that filters, edits, persists, and progressively enhances server-rendered HTML, but the boundary it creates affects everything that follows.

Progressive enhancement is demonstrated by failure behavior and user outcomes, not by the presence of server-rendered HTML alone. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to test only the fully loaded happy path with a mouse. It makes later failures harder to locate. Instead, write a matrix across input methods, network states, reduced motion, storage state, and script availability. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Finish the issue board as a progressively enhanced interface that remains understandable with keyboard, assistive technology, slow networks, and JavaScript failure. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Complete the board and record evidence for each matrix row, including one limitation you chose not to hide. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: accessible progressive ui

Check semantics, focus, live regions, persistence, HTMX coordination, testing, and graceful failure.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/alpinejs/.