shadcn/ui Course

Flavio Copes

Updated August 3, 2026

Contents

Learn the open-code component model, compose accessible interfaces, build and validate forms, create themes, review registries, and maintain the code you own.

What you'll learn: Build an accessible, responsive, themed product interface with shadcn/ui components, a validated form, a focused client boundary, and a clear maintenance path.

Open-code foundations

Set up the current CLI, understand components.json, and inspect generated code.

What shadcn/ui is

Understand shadcn/ui as accessible component source code and a distribution system rather than a traditional runtime component package.

shadcn/ui gives you designed components, blocks, and the tools to copy their source into your own project. The top layer of component code becomes part of your application.

That changes the ownership model. You can inspect and adapt a Button or Dialog instead of fighting a distant package API, but updates are also your responsibility. Current shadcn/ui can generate components on different accessible primitive bases, including Base UI, Radix, and React Aria. Do not assume every project uses the same base or composition API.

The generated file is yours, but it can still import runtime packages for primitives, class composition, icons, and animation. Think of shadcn/ui as a source distribution layer over those dependencies. A registry update does not automatically patch your local Button, and upgrading a primitive package can still change its behavior.

This model is strongest when the team wants to shape a small component system around its product. It is weaker when nobody will own accessibility tests, local changes, and dependency upgrades. "Copy and paste" describes delivery, not the maintenance cost.

Base UI is the current default base, while Radix and React Aria remain supported choices. Read the source generated for this project. An example using `asChild`, `render`, or another composition mechanism may belong to a different base even when the visual result looks identical.

Compare this model with one component library installed from npm. Trace a generated Button from your file to its runtime imports, then list one advantage, one maintenance responsibility, and one upgrade path created by owning the source.

Create a current project

Start from the current shadcn CLI or add it to an existing React project without copying setup commands from an outdated tutorial.

The CLI evolves quickly. Use the current installation page to choose a supported template or initialize an existing Next.js or Vite project.

For this course, create a Next.js project so the final interface can connect to the preceding Next.js course. The CLI can scaffold a template directly, while `init` configures an existing project. Run commands from the application root and inspect every prompt rather than accepting a base, style, icons, and color you did not intend.

The current CLI lets you select a supported primitive base. That decision changes generated imports and composition APIs, so record it with the style and color choices. Do not combine a recent Base UI project with component code copied from an old Radix tutorial without translating the behavior.

```
# new Next.js template
npx shadcn@latest init -t next

# or inside an existing compatible project
```

```
npx shadcn@latest init
```

Before initialization, commit or snapshot the clean application. Run the command, then review `components.json`, global CSS, package changes, aliases, and every created file. Start the app and run its normal production build once; a styled demo page is not proof that aliases and server/client boundaries build correctly.

Create `project-board` with the current Next.js template or initialize the existing course project. Record the CLI version, base, style, icon library, and package changes so a later generated component can be reproduced and reviewed.

Read components.json

Understand the configuration that tells the CLI where to place code, which aliases and styles to use, and how to resolve registries.

`components.json` is a tool configuration file. It records choices the CLI needs when it resolves and writes component code.

Inspect the selected style and base, the CSS entry, aliases for components and utilities, icon choice, and configured registries. Paths must agree with your TypeScript or package import aliases. Changing the file affects future CLI operations; it does not magically rewrite every component you already customized.

The schema also records whether the project uses TypeScript/TSX, React Server Components, CSS variables, and the CSS or Tailwind configuration appropriate to the installed setup. Registry entries can be plain URLs or configured namespaces. Treat namespace names as project configuration because they determine where future code is fetched.

The selected style cannot be changed in place after initialization. More broadly, editing configuration is not a migration: existing source keeps its imports and markup while the next `add` command follows the new values. A careless alias or base change can leave two incompatible generations of components in the same folder.

Open `components.json` and trace every alias to TypeScript configuration and a real folder. Confirm the CSS path is loaded, identify the base used by an existing component, and predict the exact destination and imports for the next generated item before adding it.

Add and inspect a Button

Generate the first component, render its variants, and read the source and dependencies before treating the output as a black box.

The `add` command resolves a registry item, installs missing dependencies, and writes the component files into your project.

Use the CLI's `view` command before `add` when you want to inspect the resolved official item without writing it. After adding Button, review both the source diff and package diff. Notice how native button props are forwarded, how variants become classes, and how the class-merging helper is used. The exact code depends on the selected base and style, so reading your file is more useful than memorizing one screenshot.

```
npx shadcn@latest add button
```

A variant is a semantic promise, not merely a color. Destructive should identify a consequential

action, disabled must prevent interaction and expose state, and a pending button needs text or another accessible indication beyond a spinner. Inspect the final DOM: the result should still be a real button unless navigation semantics require a link.

Render the default, secondary, outline, and destructive variants plus disabled and pending examples. Activate each real button with Enter and Space, inspect focus and accessible names, and explain every dependency the command added.

Check your understanding: open-code foundations

Check the open-code model, CLI setup, components.json, component bases, and the files a generated component adds to your project.

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

Take this interactive quiz online

Compose components

Build product components, variants, semantic controls, and an accessible dialog.

Compose instead of wrapping everything

Combine focused component parts in application code and avoid building a second rigid abstraction layer over the generated source.

Many shadcn components expose related parts, such as CardHeader, CardContent, and CardFooter. You arrange those parts to match the interface you need.

Create application components for repeated product concepts, not one wrapper for every UI primitive. A ProjectCard can compose Card and Button because it represents a real object. A MyButton wrapper that only renames every Button prop usually creates indirection without adding meaning.

Use three levels deliberately. Primitive components own reusable interaction and visual contracts. Compositions arrange primitives for a recurring interface pattern. Domain components accept product data and actions, such as a project and an archive callback. Keeping these jobs distinct makes it clear where a change belongs.

Prop mirroring is a warning sign. If a wrapper accepts every Button prop plus several exceptions, callers must understand two APIs and upgrades must preserve both. A focused domain component can expose `project`, `onOpen`, and `onArchive` because those names describe the product rather than the underlying library.

Add Card and build a ProjectCard with a name, description, status, and actions outside `components/ui`. Change the underlying Card spacing and the product's archive wording separately; each edit should have one obvious owner and should not require forwarding arbitrary primitive props.

Work with variants and cn()

Use declared variants for repeated visual states and merge occasional caller classes without creating contradictory style combinations.

Generated components commonly use a variant utility and a `cn()` helper that combines conditional Tailwind classes and resolves conflicts.

Add a named variant when a visual state is reused and belongs to the component API. Pass `className` for a local layout adjustment. Do not make every one-off spacing choice a permanent variant, and do not let arbitrary classes erase required focus, disabled, or sizing behavior without review.

Variants work best as a small matrix of stable dimensions such as intent and size. If "compact destructive toolbar button on the dashboard" becomes one variant name, layout and product context have leaked into the primitive. Keep placement with the caller and meaning with the component.

Class merging is convenient because later conflicting utilities can win. That is also the risk: a caller can silently replace padding, foreground color, focus ring, or disabled opacity. Decide which classes are intentionally overridable and test the complete variant matrix after changing the merge order.

Inspect the Button variants and write down their public dimensions. Add one restrained `success` variant, use it twice, and use `className` only for surrounding layout. Render every size against every intent, including focus and disabled states, then try one conflicting class to see exactly what `cn()` resolves.

Preserve native semantics

Choose links, buttons, labels, inputs, and headings by behavior first, then apply components and visual styles without erasing meaning.

A polished component can still be inaccessible if the underlying element is wrong. Navigation is a link; an in-place action is a button.

Use the composition mechanism documented for your selected component base when another element must receive the component styles. Base UI, Radix, and React Aria may use different APIs. Verify the rendered DOM rather than assuming a prop named in an older example still applies.

Styling an anchor like a button does not turn it into a button, and that is good: it should still support copying the address, opening in a new tab, and modified clicks. Conversely, a button must not gain a fake `href` to perform an in-place mutation. Never nest one interactive element inside another to combine styles.

Composition can also change which element receives refs, event handlers, disabled state, and accessibility attributes. The source DOM is the evidence. Check the element name, role, accessible name, focus order, and keyboard behavior after composition, not only the React props before it.

Style a real Next.js Link as a button-like call to action using the documented API for your base. Inspect the final element, open it with a modified click, and compare it with an in-place button activated by Enter and Space. There should be no nested interactive nodes.

Build an accessible Dialog

Compose a modal interaction with a real trigger, title, description, focus management, and explicit close actions.

Dialog is an example of why accessible primitives matter. A useful modal must manage focus, keyboard dismissal, labelling, and the surrounding page.

Add the Dialog component and use the generated parts for its trigger, content, title, description, and close controls. Keep a visible title unless the accessible name is supplied another documented way. Test the primitive's default outside-click and Escape behavior before overriding it; changing dismissal can trap or surprise users.

Focus should enter the dialog at a useful control, remain within it while open, and return to the trigger on close. That trigger must still exist; opening from an item that is deleted during the interaction needs an explicit fallback focus target. Destructive confirmation should identify what will be lost rather than use a vague "Are you sure?"

When a dialog contains a save operation, controlled open state is often necessary. Do not close on submit before the server succeeds, because a failed request would hide the error and entered data. Keep the content usable on a short mobile viewport and with the on-screen keyboard visible.

```
npx shadcn@latest add dialog
```

Create an Edit project dialog with Cancel and Save. Open it by keyboard, tab through it, press Escape, and check focus return. Add a delayed failed save and verify the dialog stays open, values remain, the error is announced, and the content scrolls on a short viewport.

Check your understanding: compose components

Check component composition, variants, semantic controls, cards, dialogs, accessible names, and primitive-specific behavior.

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

Take this interactive quiz online

Forms and feedback

Create semantic fields, validate data, expose errors, and communicate mutation state.

Build a semantic form

Compose a card, labels, inputs, and a submit button while keeping the native form contract visible and usable.

Let's build a form before adding any validation library. This gives us a solid HTML foundation first.

Add the Button, Field, and Input components with the shadcn CLI. Then connect the label and input using `htmlFor` and `id`. The `name` attribute is what includes the value in `FormData`.

```
<form action={createProject}>
  <Field>
    <FieldLabel htmlFor='name'>Project name</FieldLabel>
    <Input id='name' name='name' />
  </Field>

  <Button type='submit'>Create project</Button>
</form>
```

Notice that shadcn/ui does not change how forms work. We still use a real `form`, a real input, and a submit button. The components give us a consistent design on top of those browser features.

The `name` is the server contract: an attractive input without one contributes nothing to `FormData`. Give fields useful autocomplete tokens and input types. Use `fieldset` and `legend` when several

controls answer one question, and connect help text with `aria-describedby`. Placeholder text can show an example, but it disappears during typing and cannot replace a label.

Start with the native submission path before adding a form library. It exposes incorrect names, missing button types, accidental nested forms, and Enter-key surprises early. Browser autocomplete and keyboard submission are useful product features, not details to disable for visual consistency.

Add a labelled description field with an appropriate name and autocomplete decision. Submit it to an action that prints only the received key names. Test Enter, Tab order, browser autocomplete, and a directly constructed `FormData` object before adding client validation.

Validate with React Hook Form and Zod

Connect a Zod schema to React Hook Form and infer one data shape without pretending client validation protects the server.

Our form accepts any project name right now, including an empty one. Let's fix that with a Zod schema.

The schema is the list of rules for our form data. We can also use it to create the TypeScript type, so the rules and the type cannot drift apart.

```
import { zodResolver } from '@hookform/resolvers/zod'
import { useForm } from 'react-hook-form'
import { z } from 'zod'

const projectSchema = z.object({
  name: z.string().trim().min(3).max(60),
  description: z.string().trim().max(280).optional(),
})

type ProjectForm = z.infer<typeof projectSchema>

const form = useForm<ProjectForm>({
  resolver: zodResolver(projectSchema),
  defaultValues: { name: '', description: '' },
})
```

Now React Hook Form can validate the fields before calling our submit handler. It also knows the exact shape of the valid data.

Be careful though. This only improves the form experience in the browser. Run the same schema in the Server Action too, because anyone can send a request without using our form.

Normalization is part of the contract. Decide whether a blank description becomes `undefined` or an empty string, and make that decision before persistence. Complete default values keep controls from switching between uncontrolled and controlled state and make failed submissions easier to restore. Choose validation timing deliberately; validating every keystroke can be noisy.

Use `safeParse` again on the server and map expected issues to a small serializable error shape. Do not return the entire Zod result, database exception, or rejected payload to the browser. Authorization is a separate check and must run even when the schema succeeds.

Try empty, whitespace-only, valid, and overlong values. Then forge a direct request and verify that

server validation agrees with the client while server authorization independently decides whether this user may create a project.

Render accessible field errors

Associate validation messages with their controls and expose invalid state without relying on red borders or placeholder text.

Validation is only useful if the user can understand what went wrong. A red border is not enough.

The Field component can connect the input to its error. We also add `aria-invalid`, so assistive technology can expose the invalid state.

```
<Controller
  name='name'
  control={form.control}
  render={({ field, fieldState }) => (
    <Field data-invalid={fieldState.invalid}>
      <FieldLabel htmlFor={field.name}>Project name</FieldLabel>
      <Input
        {...field}
        id={field.name}
        aria-invalid={fieldState.invalid}
      />
      {fieldState.invalid && (
        <FieldError errors={[fieldState.error]} />
      )}
    </Field>
  )
/>
```

Write errors that help the user fix the value. “Use at least 3 characters” is much better than “Invalid input”.

Give the message a stable ID and include it in the control’s `aria-describedby` together with any existing help text. This preserves the relationship in the accessibility tree instead of relying on visual proximity. Keep the label visible and never use color as the only signal.

On a short form, leaving focus on submit and announcing a summary may be enough. On a long form, focus a linked error summary after a failed submission so the user can jump to each field. Avoid moving focus on every client-side validation event, which interrupts typing.

Submit the empty form and inspect name, invalid state, description, and message in the accessibility tree. Navigate to every error without a mouse, then repeat with colors disabled or high contrast enabled. The problem and recovery instruction should remain clear.

Show pending, success, and failure

Prevent duplicate submissions and communicate asynchronous results in the interface without losing the user’s data or context.

The form does not finish the instant we click its button. While the request runs, we need to show that something is happening.

`useFormStatus()` gives a component the status of its parent form. We can use it to disable the submit button and change its text.

```
'use client'

import { useFormStatus } from 'react-dom'

export function SubmitButton() {
  const { pending } = useFormStatus()

  return (
    <Button type='submit' disabled={pending}>
      {pending ? 'Creating…' : 'Create project'}
    </Button>
  )
}
```

Put this component inside the form. `useFormStatus()` reads the nearest parent form, so it will not work if the button lives outside it.

Keep server errors visible near the form. A toast is fine for a small success message, but do not make a disappearing toast the only place where you explain a failure.

Pending UI reduces accidental repeats but does not guarantee exactly-once execution. Network retries, double submissions before state updates, and back-button resubmission still happen. Protect important operations with a unique constraint or idempotency key.

A network failure can leave the outcome unknown: the server may have committed before its response was lost. Do not claim "nothing was saved" unless you know that. Preserve entered values for correctable failures and offer a safe retry or refresh path. Success should update the durable interface or navigate to the created resource; a toast can reinforce that evidence.

Add success, known rejection, and unknown network-outcome modes to the delayed action. Submit twice and retry. The button should explain progress, values should survive failure, important messages should persist, and one idempotency key should create at most one project.

Check your understanding: forms and feedback

Check field semantics, FormData, schema validation, React Hook Form integration, accessible errors, pending states, and feedback messages.

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

Take this interactive quiz online

Themes and registries

Create themes, add dark mode, customize owned code, and assess third-party sources.

Understand theme tokens

Change semantic color and radius variables as a coherent system instead of scattering one-off utility overrides across components.

A shadcn theme maps semantic names such as background, foreground, primary, muted, destructive, border, and ring to CSS values.

Components consume the semantic tokens, so changing a token can update many surfaces consistently. Judge a theme as a system: foreground against background, text on primary, muted text readability, destructive actions, focus rings, and borders all need sufficient contrast.

Think in token pairs, not isolated colors. `primary` needs a readable `primary-foreground`; `destructive` needs its own foreground; popovers and cards need compatible surfaces and borders. A brand blue that works as a large background may fail as small text or a focus ring.

Semantic names also protect intent. Product code should request a muted surface or destructive action instead of knowing the raw brand palette. Change the token when the meaning changes everywhere; add a new semantic token when two meanings have genuinely diverged. One-off utility overrides make dark mode and future redesigns harder to reason about.

Radius, spacing, and typography participate in the same system even when represented differently. Review normal, hover, focus, active, disabled, selected, and error states rather than approving a static palette screenshot.

Create a restrained theme and place Button, Card, Input, selected control, disabled control, error message, and focus ring on one audit screen. Trace every visible value to a semantic token and test contrast plus forced-colors behavior.

Implement dark mode

Apply the documented dark-mode pattern for the framework and prevent an unreadable first render or a theme that ignores system preference.

Dark mode works by applying an alternate set of theme variables under a class or another documented selector.

Follow the current guide for Next.js or Vite because initialization and hydration differ by framework. Decide whether the interface follows the system only or offers light, dark, and system choices. Persist a manual choice and apply the initial theme early enough to avoid a flash of the wrong colors.

Treat light, dark, and system as three preference states. “System” is not another fixed palette: it resolves through `prefers-color-scheme` and must react when that preference changes. A manual light or dark choice should stop following the system until the user chooses system again.

Server-rendered HTML cannot read browser storage. The framework-specific provider pattern coordinates the first client render with the class placed on the document; use hydration suppression only where the documented theme mechanism requires it, not to hide unrelated mismatches. Watch the first paint on a throttled reload, not only the settled page.

Every semantic state needs a dark value. Charts, illustrations, code blocks, selection, disabled text, overlays, and focus rings often expose gaps that a background-and-foreground swap misses.

Implement the documented provider and a labelled three-state control. Reload each saved state under network throttling, change the operating-system preference while “system” is active, and audit every interactive state in both resolved themes.

Customize the code you own

Make a deliberate local component change and preserve the accessibility and behavior contracts supplied by its primitive base.

Owning the component source means you can change its markup, variants, classes, and behavior. It does not mean every change is harmless.

Keep product-specific composition outside the UI primitive when possible. When a change belongs in the primitive, document the reason and test its variants, ref forwarding, keyboard behavior, states, and types. A diff against a fresh registry version can help you understand future upstream changes without blindly overwriting yours.

Change the lowest owner of the decision. Product copy belongs in a domain component, repeated visual intent may belong in a variant, and behavior shared by every instance may belong in the primitive. Editing the primitive for one page creates a global exception that future callers cannot see.

Preserve the less visible contract: refs, prop types, data attributes used by CSS, primitive parts, accessible names, focus handling, and disabled behavior. A markup simplification that removes a primitive wrapper can silently break all of these while still looking correct with a mouse.

Record meaningful deviations near the component or in a maintenance note. When upstream changes, generate or view a fresh item separately and merge the relevant ideas. Re-adding over the owned file is not an update strategy.

Add a compact Button size or adjust Card spacing in the generated source. Compare it with a fresh registry version, then test types, refs, every variant, keyboard focus, disabled state, and data attributes. Record why the change belongs at this level.

Evaluate third-party registries

Treat registry items as incoming source code, inspect what a command will add, and avoid handing unreviewed components authority in your app.

The registry ecosystem can distribute components, hooks, pages, configuration, and dependencies. That is powerful because it distributes executable code.

Prefer known sources, inspect the registry item and generated diff, review new packages and scripts, and verify network, authentication, and environment access. Check accessibility yourself. A beautiful demo is not evidence that a component is maintained, safe, compatible with your selected base, or suitable for production.

Use `shadcn view` to inspect a resolved item before installation. Registry schema validation checks structure; it does not prove that the source is safe, licensed appropriately, or well maintained. An item can add code, dependencies, configuration, environment assumptions, and files outside the obvious component folder.

For a private registry, keep credentials in environment variables and send them only to the intended host. Review namespace configuration so a familiar item name cannot resolve from an unexpected source. Authentication proves access, not trustworthiness.

Install uncertain items in a disposable copy or isolated branch with no secrets loaded. Inspect lifecycle scripts, transitive packages, network calls, server/client directives, telemetry, and generated routes before moving selected code into the application.

Choose one item from the official directory and one third-party item. View both without adding them, then write an accept-or-reject review covering provenance, license, files, dependencies, scripts, authority, accessibility, base compatibility, and maintenance signals.

Check your understanding: themes and registries

Check CSS theme tokens, dark mode, source customization, registry trust, third-party code review, and update ownership.

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

Take this interactive quiz online

Build a product interface

Adapt blocks, test responsive behavior, place client boundaries, and audit the result.

Start from a block without surrendering the design

Use a larger block as an accelerator, understand the files it installs, and remove demo assumptions before integrating it into a real product.

Blocks combine many components into complete screens such as dashboards, login pages, and sidebars. They can save days of initial layout work.

A block is example application code, not a requirement. Inspect the command and diff, identify its component and data dependencies, then adapt navigation, copy, data, and information hierarchy. Delete demo files and controls you do not need. Do not preserve a complicated dashboard merely because it arrived in one command.

Inventory the block before integrating it: routes, components, hooks, mock data, images, dependencies, client boundaries, and environment assumptions. A large block may encode an authentication model or navigation hierarchy that does not match the product. Visual polish is not evidence that those decisions belong in your app.

Replace placeholder data from the outside in. First connect the page to the real domain shape, then remove dead cards, menus, and filters, and finally simplify abstractions that served only the demo. Verify permissions as real data is introduced; hiding an admin action in the interface is not authorization.

Keep the block's accessibility behavior only when you understand it. Test landmarks, heading order, keyboard navigation, focus, responsive overlays, and loading/empty/error states after removing pieces because composition changes can break relationships between parts.

Add a small dashboard or sidebar block to a disposable copy. List every added file and assumption, then transplant only what Project Board needs. The result should use real domain names, contain no unreachable demo control, enforce permissions on the server, and be simpler than the source block.

Make the layout responsive

Adapt cards, navigation, dialogs, and forms to narrow screens and content growth instead of testing only the polished desktop demo.

Component defaults cannot know your data length, language, or page layout. Responsive behavior emerges from the composition around them.

Start with one column, add wider layouts only when space permits, and let content wrap. Test long names, validation messages, zoom, touch targets, and open overlays. A Dialog that fits a sample sentence may overflow when real content or the on-screen keyboard appears.

Keep source order logical before using grid placement or visual ordering. A two-column desktop layout should collapse into the same reading and keyboard sequence a user expects. Do not hide a critical action merely because the narrow layout is difficult; move or simplify it.

Set sensible minimum sizes on grid children and allow long text to break. Tables may need a deliberate scroll container or a different small-screen representation. Fixed headers and footers inside a dialog must leave a scrollable content area, including when the mobile keyboard reduces the visual viewport.

Responsive testing includes interaction states. Open menus near viewport edges, show the longest validation message, focus controls at 200% zoom, and check that focus never lands behind an overlay. Device presets are useful starting points, not a substitute for dragging through the widths between them.

Test Project Board from 320px upward, at 200% zoom, with doubled text and a short viewport. Use keyboard and touch-sized controls while dialogs and errors are open. Fix overflow without truncating information or hiding actions users need.

Place Next.js client boundaries

Keep data loading and static composition in Server Components while isolating dialogs, menus, form hooks, and theme controls that need the client.

Many interactive shadcn components need client-side state. That does not require converting the entire page into a Client Component.

Let the server page load and shape data, then render focused client components for interactions. Pass serializable props and keep server actions in server files. Generated components that already contain `"use client"` establish their own boundary; importing one does not require adding the directive to every ancestor.

The directive creates a client module graph: the file and modules it imports are eligible for the browser bundle. Moving it to the page for convenience can pull static layout, formatters, and large dependency trees across the boundary. Inspect the bundle and module graph rather than counting directive lines.

Props crossing from server to client must be serializable. Send a shaped project record, not a database client, secret-bearing object, or arbitrary class instance. A Server Component can also be passed as `children` into a Client Component, letting the server render substantial static content while the client component controls a small interactive shell.

Keep actions in a dedicated server file when a Client Component must import them. The server directive protects execution location, while the action still needs authentication, authorization, and validation as a public mutation boundary.

Keep the Project Board page on the server and isolate its create dialog and form as a client island. Inspect browser modules, props, and rendered HTML. Then move the boundary to the page temporarily and compare what is shipped before restoring the smaller design.

Audit and maintain the interface

Finish Project Board with an accessibility, design, dependency, source-control, and update review that reflects the open-code ownership model.

A component page that looks good is not finished until its interactions, states, code ownership, and maintenance path are understood.

Audit behavior by state: initial, loading, empty, populated, invalid, pending, failed, successful, and unauthorized. Keyboard-test every action, inspect names and errors in the accessibility tree, check contrast and reduced motion, and review narrow screens, zoom, long content, and both themes. Test what happens when a dialog trigger disappears and when a network result is uncertain.

Audit ownership next. Separate generated primitives, local changes, domain compositions, registry sources, and runtime dependencies. Commit `components.json`, generated source, styles, and package changes together. Record the chosen base and style plus meaningful deviations, because a future maintainer cannot infer which upstream change is safe to merge.

Updates are source reviews. Use `view` or generate a fresh item separately, compare it with the owned file, and merge useful fixes while preserving intentional local behavior. Before any reinstall command, read CLI help and inspect the working tree so local changes are not silently replaced.

```
npx shadcn@latest --help
npx shadcn@latest add --help
npm run lint
npm run build
```

Complete Project Board with cards, a dialog form, schema validation, server-safe mutation states, theming, and responsive behavior. Write a maintenance note naming the base, style, registries, dependencies, customizations, test matrix, and update procedure. Another developer should be able to explain what is owned and reproduce the review without relying on the demo screenshot.

Check your understanding: build a product interface

Check blocks, responsive adaptation, Server and Client Component boundaries, upgrades, source control, and a complete UI review.

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/shadcn-ui/.