Web Accessibility Course

Flavio Copes

Updated August 3, 2026

Contents

Find and remove barriers with semantic HTML, keyboard and focus work, accessible forms, restrained ARIA, and repeatable testing.

What you'll learn: Audit, repair, test, and document an event-registration page across keyboard, screen-reader, zoom, motion, and failure conditions.

Accessibility foundations

Start from user tasks, standards, the accessibility tree, and native semantics.

Start from user tasks

Define who must accomplish which task under which input, perception, and cognitive constraints.

Before choosing a tool or writing more code, make the requirement concrete. Registration must work without sight, without a mouse, with zoom, with reduced motion, and with clear error recovery.

Accessibility work begins with people and tasks, then uses standards and tools to find barriers. 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 treat accessibility as a final score or a checklist for one imaginary user. The happy path may still work, which makes the mistake easy to miss. write critical tasks, varied interaction conditions, and observable success criteria before auditing components. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Frame accessibility around successful user tasks and inspect what browsers expose to assistive technology. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Describe the complete registration task for keyboard-only, screen-reader, low-vision, and cognitive-load scenarios. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Understand conformance and reality

Use WCAG as a shared testable baseline while recognizing that conformance does not guarantee a good experience.

Start from the behavior you can observe. The team maps failures to criteria and also records confusing behavior that no automated score explains.

Standards organize important requirements, but usability, language, platform differences, and real assistive technology still need human judgment. 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 chase a perfect automated score and stop when the badge turns green. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to combine criterion-based checks, task testing, and user evidence, then prioritize barriers by impact. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Frame accessibility around successful user tasks and inspect what browsers expose to assistive technology. Record enough evidence that another developer can repeat the result without relying on your memory.

Take one automated finding and one manual usability problem and show how each affects the registration task. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Inspect the accessibility tree

See how names, roles, values, states, and relationships are derived from HTML and attributes.

A native button exposes a role, accessible name, disabled state, and keyboard behavior without custom scripting. That contrast gives us something useful to test instead of a rule to memorize.

Assistive technologies work from platform accessibility information, not directly from the visual pixels. 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 assume visible text and CSS alone produce the intended programmatic meaning and still get one successful demo. Push past the demo. inspect the computed accessibility properties and fix the source semantics that create them, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Frame accessibility around successful user tasks and inspect what browsers expose to assistive technology. Save the before-and-after evidence now; it will make the final project review much more honest.

Compare a button element with a clickable div in the accessibility inspector and keyboard. Save the evidence, then explain which requirement would force you to choose a different design.

Prefer native semantics

Choose the HTML element whose built-in meaning and behavior match the interaction before adding ARIA.

Links navigate, buttons perform actions, headings structure content, and labels name controls. This is a small part of a deliberately flawed event-registration page that is audited, repaired, and retested with real evidence, but the boundary it creates affects everything that follows.

Native elements carry semantics, focus behavior, keyboard activation, forms integration, and platform support. 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 rebuild a native control from generic elements and repair each missing behavior manually. It makes later failures harder to locate. Instead, start with native HTML, add ARIA only for a missing semantic, and never contradict the host element. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Frame accessibility around successful user tasks and inspect what browsers expose to assistive technology. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Replace three generic interactive elements on the flawed page and list the code you could delete. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: accessibility foundations

Check disability models, user tasks, standards, semantic HTML, accessibility trees, and progressive enhancement.

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

Take this interactive quiz online

Structure, keyboard, and focus

Create a navigable document and predictable keyboard and focus behavior.

Build a meaningful page outline

Use title, language, landmarks, headings, lists, and regions to make the page navigable.

Before choosing a tool or writing more code, make the requirement concrete. The event page has one clear main heading, logical subsections, a main landmark, and a named registration region.

Structure gives users multiple ways to understand and move through a long page. 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 choose heading levels for visual size or wrap every container in a generic div. The happy path may still work, which makes the mistake easy to miss. write the content outline first and use semantic landmarks and ordered heading levels to express it. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Make page structure and every interaction understandable and operable without a mouse. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Navigate the page by headings and landmarks in an accessibility tool and repair every ambiguous stop. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Make every action keyboard operable

Support standard Tab, Shift+Tab, Enter, Space, arrow, and Escape behavior appropriate to each control.

Start from the behavior you can observe. Buttons activate with Enter and Space, links with Enter, and the date choice uses a documented native control rather than invented keys.

Keyboard support follows interaction semantics; different widgets have different expected keys. 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 add keydown handlers to generic elements without focus, role, or standard behavior. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to use native controls first and test the expected key model without a mouse. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Make page structure and every interaction understandable and operable without a mouse. Record enough evidence that another developer can repeat the result without relying on your memory.

Complete registration with the keyboard, including error correction, and record every extra or missing keystroke. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Protect focus order and visibility

Keep DOM order logical, focus indicators visible, and off-screen or hidden content out of the sequence.

The visual card layout follows the same DOM order, and sticky content does not cover the focused element. That contrast gives us something useful to test instead of a rule to memorize.

Focus order should follow the task, while a visible indicator shows exactly where keyboard input will go. 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 repair layout with positive tabindex values or remove outlines without an equivalent and still get one successful demo. Push past the demo. keep source order meaningful, use natural focus, and style a high-contrast indicator that remains unobscured, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Make page structure and every interaction understandable and operable without a mouse. Save the before-and-after evidence now; it will make the final project review much more honest.

Tab at 200 percent zoom and in high-contrast settings; capture every place focus becomes hidden or surprising. Save the evidence, then explain which requirement would force you to choose a different design.

Manage focus after change

Move focus only when context changes require it and return it to a sensible trigger afterward.

Opening a modal focuses its heading or first useful control, traps interaction appropriately, and returns focus to the opener. This is a small part of a deliberately flawed event-registration page that is audited, repaired, and retested with real evidence, but the boundary it creates affects everything that follows.

Dynamic updates can leave focus on removed elements or make new context impossible to discover. 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 move focus for every small update or leave it inside removed DOM. It makes later failures harder to locate. Instead, define the focus destination and return path as part of the interaction design. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Make page structure and every interaction understandable and operable without a mouse. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Open, submit, fail, close, and reopen the modal using only the keyboard and verify focus at every step. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: structure, keyboard, and focus

Check landmarks, headings, keyboard operation, focus order, focus visibility, dialogs, and skip links.

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

ones you get right.

Take this interactive quiz online

Forms, content, and perception

Repair labels, errors, alternatives, contrast, reflow, and motion.

Label and group form controls

Give every control a persistent name, group related choices, and provide instructions before they are needed.

Before choosing a tool or writing more code, make the requirement concrete. Contact preference radios sit inside a fieldset with a legend, and address fields use appropriate autocomplete tokens.

Labels and grouping make form purpose available visually and programmatically; placeholders are not durable labels. 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 placeholder text as the only name or repeat vague labels such as “value”. The happy path may still work, which makes the mistake easy to miss. associate explicit labels, group related controls, and add purpose metadata supported by the browser. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Repair the registration content so input and information remain clear across perception and reading conditions. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Inspect the accessibility name and autocomplete behavior for every registration field. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Make errors recoverable

Identify the field, explain the problem, preserve input, and move or announce attention appropriately.

Start from the behavior you can observe. The summary links to invalid fields, each field has a specific message, and server validation preserves valid values.

An error is useful only if the user can find it, understand it, and correct it without restarting. 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 change the border color and return a generic “invalid form” message. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to provide text, programmatic association, an error summary for multiple failures, and a clear correction path. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Repair the registration content so input and information remain clear across perception and reading conditions. Record enough evidence that another developer can repeat the result without relying on your memory.

Submit five distinct errors and repair them with keyboard and screen reader without losing entered data. Repeat it once with an invalid or hostile condition and write down the boundary the failure

revealed.

Provide equivalent content

Write useful alternative text and provide captions, transcripts, or text equivalents according to media purpose.

The event logo has concise purpose, a decorative flourish has empty alt text, and a speaker video has reviewed captions. That contrast gives us something useful to test instead of a rule to memorize.

An alternative conveys the purpose and information of content, not a mechanical inventory of pixels. 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 repeat nearby text in alt attributes or omit the information carried by a meaningful chart and still get one successful demo. Push past the demo. classify each image or medium by purpose and provide the shortest equivalent that preserves the user task, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Repair the registration content so input and information remain clear across perception and reading conditions. Save the before-and-after evidence now; it will make the final project review much more honest.

Audit every image and media item, justify its alternative, and test the page with images unavailable. Save the evidence, then explain which requirement would force you to choose a different design.

Support visual preferences and reflow

Preserve content and controls under zoom, narrow viewports, high contrast, color changes, and reduced motion.

The registration page reflows without two-dimensional scrolling for ordinary text and never uses color as the only status signal. This is a small part of a deliberately flawed event-registration page that is audited, repaired, and retested with real evidence, but the boundary it creates affects everything that follows.

Users change text size, colors, contrast, motion, and viewport to make content perceivable and comfortable. 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 lock layout dimensions and disable user scaling to preserve the design. It makes later failures harder to locate. Instead, use flexible layout, redundant cues, sufficient contrast, visible focus, and reduced-motion alternatives. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Repair the registration content so input and information remain clear across perception and reading conditions. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Test at 400 percent zoom, narrow width, forced colors, and reduced motion; record any lost content or operation. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: forms, content, and perception

Check labels, instructions, errors, autocomplete, images, color, contrast, zoom, reflow, motion, and media alternatives.

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

Take this interactive quiz online

Dynamic content and ARIA

Use custom semantics, names, live regions, and tables with restraint.

Learn the ARIA contract

Treat an ARIA role as a promise to implement the matching states, properties, focus, and keyboard behavior.

Before choosing a tool or writing more code, make the requirement concrete. A custom tab interface implements the complete tab pattern rather than adding role=tab to arbitrary links.

ARIA changes accessibility semantics; it does not add behavior, styling, focusability, or validation automatically. 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 a role to silence an audit while leaving interaction behavior unchanged. The happy path may still work, which makes the mistake easy to miss. use an established pattern, implement its complete keyboard and state contract, and test across target technology. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Use ARIA and dynamic announcements only where native HTML cannot express the required relationship or state. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Choose one custom widget, list every behavior its role promises, and compare that list with the implementation. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Compute names and descriptions

Understand how text, labels, aria-label, aria-labelledby, and aria-describedby combine or override one another.

Start from the behavior you can observe. The registration button keeps its visible name while a description explains the cancellation deadline.

Accessible names identify controls; descriptions add supporting information, and careless ARIA can hide useful visible text. 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 add aria-label everywhere and accidentally replace clearer visible labels. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to prefer associated visible text, inspect the computed name, and add descriptions only for useful supplementary context. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Use ARIA and dynamic announcements only where native HTML cannot express the required relationship or state. Record enough evidence that another developer can repeat the result without relying on your memory.

Inspect ten interactive elements and make every programmatic name match the visible intent. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Announce dynamic updates

Use restrained live regions for important asynchronous changes without creating repeated or interrupting noise.

A polite status reports successful registration; validation errors use the form error pattern rather than a stream of competing alerts. That contrast gives us something useful to test instead of a rule to memorize.

Live regions can reveal updates that do not move focus, but priority, timing, and changed text affect what is announced. 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 wrap the whole application in an assertive live region and still get one successful demo. Push past the demo. announce concise outcome changes, initialize regions predictably, and avoid duplicating information already reached by focus, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Use ARIA and dynamic announcements only where native HTML cannot express the required relationship or state. Save the before-and-after evidence now; it will make the final project review much more honest.

Test success, loading, validation, and network failure announcements and count what is spoken. Save the evidence, then explain which requirement would force you to choose a different design.

Make data tables understandable

Use captions, header cells, scopes, and simple structure so schedule relationships remain available.

The event schedule has a caption, column headers, and row headers where session names identify each row. This is a small part of a deliberately flawed event-registration page that is audited, repaired, and retested with real evidence, but the boundary it creates affects everything that follows.

A data table encodes relationships between headers and cells; layout tables and div grids do not provide those relationships. 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 style a div grid to look tabular or merge cells until header relationships become ambiguous. It makes later failures harder to locate. Instead, use a real table for relational data, keep it simple, and associate headers explicitly when complexity requires it. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Use ARIA and dynamic announcements only where native HTML cannot express the required relationship or state. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Navigate the schedule by table commands and verify the right headers are announced for every data cell. Hand the result to someone else and see whether the behavior is clear without an oral

explanation.

Check your understanding: dynamic content and aria

Check widget patterns, names and descriptions, live regions, hidden state, tables, custom controls, and ARIA limits.

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

Take this interactive quiz online

Audit, test, and maintain

Combine automation and human testing, prioritize barriers, and prevent regressions.

Run automated checks

Use validators, linters, browser audits, and test libraries to catch deterministic problems early.

Before choosing a tool or writing more code, make the requirement concrete. The pipeline catches missing labels and invalid relationships, while manual testing judges focus order and error clarity.

Automation is excellent at repeated machine-testable rules and poor at deciding whether a task is understandable or an alternative is meaningful. 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 treat tool output as complete coverage or ignore it because it cannot catch everything. The happy path may still work, which makes the mistake easy to miss. automate reliable rules, review each result in context, and record which important checks remain manual. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Turn the repaired page into a repeatable accessibility practice rather than a one-time cleanup. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run two tools, deduplicate findings, verify false positives, and map uncovered tasks to manual tests. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Test with keyboard and assistive technology

Use a small repeatable matrix across browsers, screen readers, zoom, voice, and input modes.

Start from the behavior you can observe. The release matrix covers keyboard-only, high zoom, one desktop screen reader and browser pair, and one mobile pair.

Compatibility varies, so test the combinations important to your audience without claiming one setup represents everyone. 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 perform random screen-reader exploration without tasks or expected outcomes. That trades a clear decision for an assumption you will eventually have to debug. The stronger move

is to define tasks, supported combinations, expected announcements, and evidence before running the test. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Turn the repaired page into a repeatable accessibility practice rather than a one-time cleanup. Record enough evidence that another developer can repeat the result without relying on your memory.

Register, find errors, correct them, and confirm success in each chosen combination. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Prioritize and prevent regressions

Rank barriers by user impact and reach, then add the cheapest reliable guard that stops recurrence.

An impossible registration path is fixed before a minor redundant announcement, and the label regression gains an automated test. That contrast gives us something useful to test instead of a rule to memorize.

Not every issue has equal severity, and a fix that is not protected can disappear in the next component refactor. 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 sort issues only by tool severity or fix count and still get one successful demo. Push past the demo. combine task blockage, affected users, frequency, workaround, and confidence, then add review or automation, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Turn the repaired page into a repeatable accessibility practice rather than a one-time cleanup. Save the before-and-after evidence now; it will make the final project review much more honest.

Triage the flawed page findings and defend the first three fixes plus their regression guard. Save the evidence, then explain which requirement would force you to choose a different design.

Finish the accessibility report

Document scope, methods, environments, findings, fixes, remaining risks, and ownership for the repaired page.

The report links evidence to user tasks, records assistive technology versions, and assigns remaining work with retest dates. This is a small part of a deliberately flawed event-registration page that is audited, repaired, and retested with real evidence, but the boundary it creates affects everything that follows.

A credible report says what was tested and what was not, rather than declaring the product accessible forever. 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 publish a perfect score without scope, test conditions, or known limitations. It makes later failures harder to locate. Instead, write an honest evidence-based report and integrate accessibility into design, implementation, review, and release. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Turn the repaired page into a repeatable accessibility practice rather than a one-time cleanup. A short observation with concrete evidence is more useful than a confident

sentence with no reproduction path.

Compare the original and repaired registration task, then hand the report and regression matrix to another tester. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: audit, test, and maintain

Check automated and manual testing, assistive technology, issue prioritization, regression tests, process, and final evidence.

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/web-accessibility/.