Web Forms Course

Flavio Copes

Updated August 8, 2026

Contents

Learn to build accessible forms, validate input, submit data and files, use FormData, and handle user input safely on the server.

What you'll learn: Build accessible forms and understand the complete browser-to-server submission path.

What a form does

Controls, name-value pairs, methods, and browser submission.

What a form does

Understand how a form collects named values and turns a user action into an HTTP request the server can process.

A form is an agreement between the page and the server. The page collects named values. The browser turns those values into an HTTP request. The server decides whether to accept them.

Start with a form that works without JavaScript:

```
<form action="/subscribe" method="post">
  <label for="email">Email address</label>
  <input id="email" name="email" type="email" required>
  <button type="submit">Subscribe</button>
</form>
```

When you submit it, the browser reads the successful controls inside the form. Here it finds one named value, such as `email=hello@flaviocopes.com`.

The `action` tells the browser where to send the request. The `method` tells it to use POST. The server at `/subscribe` still has to parse and validate the value before storing it or sending email.

Notice the different jobs:

- `id` connects the input to its label
- `name` creates the key sent to the server
- `type` gives the browser input and validation behavior
- `required` blocks an empty native submission

An input without a `name` can still appear and accept text, but normal form submission leaves it out.

Open the Network panel, submit the form, and inspect the request method, URL, and payload. That request is the real output of the form.

Form action and method

Choose the destination URL and GET or POST method, then predict where the browser places the submitted name-value pairs.

`action` chooses the request URL. `method` chooses how the browser sends the named values.

Use GET when submitting the form reads information:

```
<form action="/search" method="get">
  <label for="query">Search</label>
  <input id="query" name="q">
  <button type="submit">Search</button>
</form>
```

Entering `forms` navigates to a URL such as `/search?q=forms`. That URL can be copied, bookmarked, revisited, and indexed. This makes GET a good fit for searches and filters.

Use POST when the request changes server state:

```
<form action="/account/email" method="post">
  <label for="email">New email address</label>
  <input id="email" name="email" type="email" required>
  <button type="submit">Update email</button>
</form>
```

The browser puts the encoded values in the request body. The URL alone no longer describes the operation.

POST is not automatically private or secure. HTTPS protects both URLs and bodies while they travel across the network. Server logs and application code can still expose carelessly handled data.

Do not use GET for an action such as deleting a record. Browsers, crawlers, previews, and caches may follow GET links because GET is expected to be safe.

Submit one GET form and one POST form with DevTools open. Compare the request URL and payload.

Which controls are submitted

Identify successful form controls and understand why disabled, unnamed, unchecked, or unselected controls may be absent from submitted data.

A submitted form does not include every control inside it. The browser builds the data set from **successful controls**.

Consider this form:

```
<form action="/preferences" method="post">
  <input name="displayName" value="Ada">
  <input value="not sent">
  <input name="accountId" value="42" disabled>

  <label>
    <input name="newsletter" type="checkbox" value="yes">
    Join the newsletter
  </label>

  <button name="action" value="save">Save</button>
</form>
```

The browser sends `displayName=Ada`. It does not send the unnamed input. It omits the disabled input and the unchecked checkbox. It includes `action=save` only when that submit button initiated submission.

This matters on the server. Missing `newsletter` might mean "not selected," but a missing required `displayName` is an error. Treat absence according to the endpoint contract instead of turning every missing value into an empty string.

Do not use `disabled` to protect an important value. A visitor can change the HTML or construct a request by hand. If the server already knows the account ID from the authenticated session or route, use that trusted value rather than accepting it from the form.

`readonly` behaves differently from `disabled`: a read-only text control is normally submitted. It can be useful for a visible value, but the server must still verify it.

Try checking the newsletter box and submitting with the Network panel open. Compare both payloads.

Submit buttons

Use button types deliberately so primary submission, secondary actions, and reset behavior do not surprise the visitor.

A `button` inside a form submits by default. I still write the type because it makes the intent clear and prevents surprises when markup moves.

```
<button type="submit">Publish</button>
<button type="button" id="preview">Preview</button>
```

The first button submits the form. The second does nothing until JavaScript gives it behavior.

Several submit buttons can send different actions to one endpoint:

```
<button type="submit" name="action" value="draft">Save draft</button>
<button type="submit" name="action" value="publish">Publish</button>
```

Only the submitter that started the submission contributes its `name` and `value`. The server can read `action`, then check whether that action is allowed for the current user and record.

Do not trust the button value as authorization. Someone can send `action=publish` without clicking the visible button.

Avoid `type="reset"` in most forms. It restores the initial values, which can erase several minutes of work without saving anything. A clearly labeled "Clear form" action may be justified, but it should be hard to trigger accidentally.

Also test keyboard submission. Pressing Enter in a field can submit without a pointer click, which is why form-level submit handling matters more than button click handling.

Text and choice controls

Choose and combine the native controls users need.

Text controls

Choose text, email, URL, password, search, telephone, and multiline controls according to the value a person needs to enter.

Use the native control that matches what the person is entering.

```
<label for="email">Email address</label>
<input id="email" name="email" type="email" autocomplete="email">

<label for="website">Website</label>
<input id="website" name="website" type="url">

<label for="message">Message</label>
<textarea id="message" name="message" rows="6"></textarea>
```

The type affects browser behavior. An email input can show a suitable mobile keyboard and perform basic format validation. A URL input understands URL-shaped values. A `textarea` accepts multiple lines.

The type does not prove that an address exists, that a URL is safe to fetch, or that a message is acceptable. The server receives strings and must apply the real contract.

Use `type="password"` when text should be visually obscured. It does not encrypt the value. HTTPS protects the request in transit, and the server must store passwords using a suitable password-hashing function rather than plain text.

`type="search"` and `type="tel"` mainly improve input behavior. Telephone formats vary, so do not apply a narrow pattern unless your service truly accepts only one format.

Add `autocomplete` values where the browser can safely help. They reduce typing and make forms easier to complete.

Test the controls on a phone or with device emulation. Look at the keyboard and validation behavior, not only their desktop appearance.

Numbers, dates, and ranges

Use numeric and date-related controls without assuming every value that looks numeric should use a number input.

Use `type="number"` for a quantity that can be compared, increased, or decreased.

```
<label for="seats">Number of seats</label>
<input id="seats" name="seats" type="number" min="1" max="12" step="1">
```

The browser can provide stepper controls and reject values outside the declared range. The submitted value is still text in the request. The server must parse it, reject invalid numbers, and check the range again.

Postal codes, card numbers, and telephone numbers are identifiers. Arithmetic makes no sense for them. They may contain leading zeroes or formatting characters, so use a text-like input with an appropriate `inputmode` when a numeric keyboard helps.

Date controls submit normalized strings such as `2026-07-30`, but displaying and parsing dates still needs care. Decide whether the value represents a calendar date, a local time, or an instant. Do

not silently invent a time zone.

A range input is useful when an approximate position matters more than typing an exact value:

```
<label for="volume">Volume</label>
<input id="volume" name="volume" type="range" min="0" max="100" value="50">
```

Show the current value when people need to know it. A slider without a visible value can be hard to use precisely.

Submit boundary values such as `0`, `12`, `13`, and an empty value. Confirm the server rejects what the HTML says it rejects.

Checkboxes and radio buttons

Model independent boolean choices with checkboxes and one choice from a set with radio buttons that share a name.

A checkbox represents an independent choice. Radio buttons represent one choice from a set.

```
<label>
  <input name="newsletter" type="checkbox" value="yes">
  Send me the weekly newsletter
</label>
```

When checked, this sends `newsletter=yes`. When unchecked, it sends no `newsletter` field at all. The server should turn that absence into `false` only if that matches the contract.

Use a shared name for radio buttons:

```
<fieldset>
  <legend>Delivery speed</legend>

  <label>
    <input name="delivery" type="radio" value="standard" checked>
    Standard
  </label>

  <label>
    <input name="delivery" type="radio" value="express">
    Express
  </label>
</fieldset>
```

The shared `name` makes one group. Selecting Express clears Standard and sends `delivery=express`.

Give every option an explicit, stable value. The visible label can change without changing what the server stores.

The server must allowlist the expected values. A person can send `delivery=teleport` even though that option does not appear in the page.

Use separate checkboxes when several answers can be true. Controls sharing one checkbox name can produce repeated values, so read them with an API that preserves all entries.

Submit each state and inspect the payload. Pay special attention to what is absent.

Select menus

Build a native select control with meaningful option values, an optional prompt, and multiple selection only when the interaction warrants it.

A `select` offers a known set of choices and submits the value of the selected `option`.

```
<label for="country">Country</label>
<select id="country" name="country" required>
  <option value="">Choose a country</option>
  <option value="dk">Denmark</option>
  <option value="it">Italy</option>
</select>
```

The visible label is for people. The value is the server contract. Selecting Italy sends `country=it`.

The empty first option works as a prompt. With `required`, the browser asks for another choice before native submission. The server must make the same decision because a handcrafted request can still send an empty or unknown value.

Do not use a huge select merely because the data is a list. Searching hundreds of options is difficult. An autocomplete or another focused control may be better.

Add `multiple` only when several choices are allowed. A multiple select can be difficult to operate, and it submits the same name more than once:

```
topic=html&topic=css
```

The server must use a method such as `getAll('topic')` rather than reading only the first value.

Change the visible country labels without changing their values. Submit again and confirm the request contract stays stable.

File inputs

Accept files with the native file control, restrict suggested formats, and understand that server-side checks remain essential.

Use `type="file"` to let a visitor choose a local file. The browser does not send the file until the form is submitted.

```
<form action="/profile/photo" method="post" enctype="multipart/form-data">
  <label for="photo">Profile photo</label>
  <input id="photo" name="photo" type="file" accept="image/png,image/jpeg">
  <button type="submit">Upload photo</button>
</form>
```

The `name` identifies the file part. `multipart/form-data` lets the request carry binary file bytes alongside ordinary text fields.

`accept` guides the file picker. It can reduce mistakes, but it is not a security boundary. A custom client can upload any bytes with any filename and declared media type.

The server must set a request-size limit, check that a file exists, inspect its actual type, and decide whether the format is allowed. It should generate its own storage name rather than using the original filename as a path.

Add `multiple` only when the endpoint and interface support several files. The request then contains

repeated file parts under the same name, and every file needs its own validation.

Do not automatically display a chosen filename as trusted HTML. Treat filenames as user input.

Try submitting the form with no file, a valid image, and a renamed non-image file. The browser may guide the first two cases. Only the server can make the final decision.

Check your understanding: native form controls

Check form structure, labels, control types, validation attributes, submission methods, and how browsers encode common values.

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

Take this interactive quiz online

Accessible form structure

Labels, instructions, focus, and useful error messages.

Label every control

Connect visible labels to controls so the form remains understandable, clickable, and navigable with assistive technology.

Every form control needs an accessible name. For visible fields, a real `label` is usually the clearest way to provide it.

Connect the label with matching `for` and `id` values:

```
<label for="email">Email address</label>
<input id="email" name="email" type="email">
```

Clicking “Email address” now focuses the input. Assistive technology can also announce the label when the control receives focus.

You can wrap a control instead:

```
<label>
  Email address
  <input name="email" type="email">
</label>
```

Both patterns work. I prefer explicit `for` and `id` values when the label and control are separate layout elements.

A placeholder is not a label. It disappears during typing, can have weak contrast, and often contains an example rather than the field's meaning.

The `name` has a different job. It creates the key sent to the server. A control can submit correctly while still having no accessible name, so you need both.

Icon-only controls need a name too. Prefer visible text when space allows. Otherwise provide a concise accessible name, such as `aria-label="Search"`.

Test the form by clicking every label. Then inspect each control in the Accessibility pane and check

its computed name.

Group related controls

Use fieldset and legend to give checkbox and radio groups a question that remains available to screen-reader users.

Individual labels name individual controls. A related set of controls also needs a group name.

Use `fieldset` and `legend` for a question with several choices:

```
<fieldset>
  <legend>Preferred contact method</legend>

  <label>
    <input name="contact" type="radio" value="email">
    Email
  </label>

  <label>
    <input name="contact" type="radio" value="phone">
    Phone
  </label>
</fieldset>
```

The legend provides the question. Each label provides an answer. A screen reader can announce both when someone moves through the radio buttons.

Visual proximity alone does not communicate this relationship. A row of radio buttons may look grouped on screen while sounding like unrelated choices.

Use a fieldset for meaningful groups, not as a wrapper around every form section. Contact details with independent text fields can often use a heading and normal labels. A set of radios or related checkboxes usually benefits from a legend.

If the whole group is invalid, place the error near the legend and connect it to the fieldset when needed. Do not repeat the same error after every option.

Navigate the group with a keyboard. Tab should enter the radio group, and arrow keys should move between its options.

Write useful instructions

Place concise help where people encounter it and connect control-specific instructions with aria-describedby when needed.

Tell people about a requirement before they submit. An error should not be the first place they learn the password length or accepted file type.

Put general instructions near the form heading. Put field-specific help beside the field:

```
<label for="username">Username</label>
<p id="username-help">Use 3-20 letters, numbers, or hyphens.</p>
<input
  id="username"
```

```
  name="username"
  aria-describedby="username-help"
  minlength="3"
  maxlength="20"
  pattern="[A-Za-z0-9-]+"
>
```

`aria-describedby` lets assistive technology associate the extra instruction with the input. The visible paragraph still helps everyone else.

Keep help concise and concrete. “Enter a valid value” explains nothing. “Use 3–20 letters, numbers, or hyphens” tells the person what will pass.

Do not put essential instructions only in a tooltip or placeholder. They can disappear, be difficult to reach with a keyboard, or never be discovered.

The server must enforce the same contract. If the page says 20 characters but the endpoint allows only 16, the interface creates avoidable failures.

Read the form without touching it. Can you predict every required format and limit before submitting? If not, move the missing instruction next to the relevant field.

Keyboard and focus behavior

Preserve logical source order, visible focus, native keyboard interaction, and a predictable destination after submission or validation.

A form should work without a mouse. Native controls already provide most of the required keyboard behavior.

Press Tab to move through interactive controls. Use Space to toggle a checkbox. Use arrow keys inside a radio group. Press Enter to submit where the browser supports implicit submission.

Keep DOM order aligned with visual order. CSS can rearrange controls, but the keyboard still follows the document. A layout that looks correct while focus jumps around is not correct.

Never remove a focus outline without replacing it with an equally visible indicator:

```
:focus-visible {
  outline: 3px solid currentColor;
  outline-offset: 3px;
}
```

Do not assign positive `tabindex` values to force an order. Fix the HTML order instead. Use `tabindex="-1"` only when script needs to focus a normally non-focusable status or error summary.

After validation fails, keep the entered values. If the form is long or has several errors, focus an error summary that links to the invalid fields. For a short form, the browser's native focus behavior may already be enough.

Avoid moving focus after every small update. Unexpected focus movement can be more confusing than helpful.

Complete the entire form using only the keyboard. Make sure every control is reachable, the focus indicator is visible, and submission reaches a useful result.

Write accessible error messages

Identify the field, explain the problem, preserve entered values, and make errors available without relying only on color.

A useful error identifies the field, explains the problem, and tells the person how to fix it.

"Email is invalid" is vague. "Enter an email address in the format name@example.com" is actionable.

Place the message next to the field and connect it:

```
<label for="email">Email address</label>
<input
  id="email"
  name="email"
  type="email"
  value="ada@"
  aria-invalid="true"
  aria-describedby="email-error"
>
<p id="email-error">Enter an email address in the format name@example.com.</p>
```

`aria-invalid` marks the current state. `aria-describedby` connects the explanation. The visible text means the error does not depend on a red border or icon.

When several fields fail, add a summary near the top:

```
<div role="alert" tabindex="-1">
  <h2>There are 2 problems</h2>
  <a href="#email">Enter a complete email address</a>
</div>
```

Preserve valid values when returning the form. Re-entering unrelated information is a punishment, not validation.

Do not echo sensitive values such as passwords. Escape submitted text before rendering it into HTML.

Try errors with color disabled, at 200% zoom, and with a screen reader if available. You should still be able to find the problem and its field.

Browser validation

Declare input constraints and explain validation errors.

Required values and length constraints

Declare required fields and sensible minimum or maximum lengths while keeping optional fields genuinely optional.

HTML can express simple constraints without JavaScript.

```
<label for="message">Message</label>
<textarea
  id="message"
  name="message"
```

```
  required
  minlength="20"
  maxlength="1000"
></textarea>
```

`required` rejects an empty value. `minlength` and `maxlength` define the accepted text length. The browser can block native submission and focus the invalid control.

Choose constraints from the real operation. A 20-character minimum may make sense for a support request, but not for a first name. Arbitrary limits create frustration and can reject legitimate data.

Do not mark every field required. Ask only for what the current task needs. A shorter form is easier to complete, easier to validate, and safer to store.

Whitespace needs a decision. A value containing only spaces may pass a length check. The server can trim where appropriate, then check whether meaningful content remains. Do not trim passwords or other values where spaces may be intentional.

The browser constraint is user feedback, not the final authority. A request can bypass the page. The server must check presence and length again before saving.

Test the empty value, 19 characters, 20 characters, and 1001 characters. Confirm both browser and server use the same boundaries.

Patterns and formats

Use native type validation and regular-expression patterns only when the accepted format is narrow, stable, and clearly explained.

Start with a native type. Email and URL inputs already provide basic format validation and suitable input behavior.

Use `pattern` only when the accepted format is narrow and stable. For example, a project code might be exactly three uppercase letters followed by four digits:

```
<label for="project">Project code</label>
<p id="project-help">Use the format ABC-1234.</p>
<input
  id="project"
  name="project"
  pattern="[A-Z]{3}-[0-9]{4}"
  aria-describedby="project-help"
  required
>
```

The pattern should describe the whole accepted value. Explain it in normal language before the person submits.

Avoid complicated regular expressions for names, postal addresses, telephone numbers, and email addresses. Real-world values are more varied than they first appear. A narrow rule often blocks legitimate people while doing little for security.

Format is only one kind of validation. `ABC-1234` may match the pattern but refer to no existing project. The server must perform that semantic check.

The server must also enforce the same syntax. Browser validation can be skipped, and clients can

submit values that never appeared in the form.

Write down five valid values and five invalid values before adding a pattern. If the rule is hard to explain or rejects plausible input, use a simpler constraint and validate the business meaning on the server.

The Constraint Validation API

Read validity states, set a custom message, and trigger or report browser validation without rebuilding the browser's validation system.

The Constraint Validation API lets JavaScript inspect and extend the browser's native validation.

Every form control exposes a `validity` object. It tells you whether a required value is missing, a number is out of range, a pattern does not match, or another constraint failed.

Use `checkValidity()` to test without showing the browser message. Use `reportValidity()` when the browser should also show its feedback.

Some rules depend on more than one field. A confirmation value is a good example:

```
const form = document.querySelector('form')
const password = form.elements.password
const confirmation = form.elements.confirmation

function validateConfirmation() {
  const matches = confirmation.value === password.value
  confirmation.setCustomValidity(matches ? '' : 'Passwords do not match')
}

password.addEventListener('input', validateConfirmation)
confirmation.addEventListener('input', validateConfirmation)
```

An empty string clears the custom error. If you forget that step, the field remains invalid even after the person fixes it.

Prefer declarative attributes such as `required`, `min`, and `pattern` for rules HTML already understands. Add JavaScript for relationships or richer presentation, not to rebuild the entire validation system.

This API still runs in the client. The server must repeat the rule before changing data.

Break the confirmation once, fix it, and call `form.checkValidity()` after each edit. Confirm the state changes back to valid.

Validate on both client and server

Use browser validation for immediate feedback and server validation as the final authority over every submitted value.

Client and server validation solve different problems.

Client validation gives fast feedback. It can catch an empty required field before a request, point to the field, and preserve the person's place in the form.

It cannot protect the server. A client can remove attributes, disable JavaScript, change hidden

values, or send a request without using the page.

Imagine a form for reserving workshop seats:

```
<input name="seats" type="number" min="1" max="4" required>
<input name="workshopId" type="hidden" value="42">
```

The browser can check the visible number. The server must make the final decisions:

1. Parse `seats` as an integer.
2. Reject values below 1 or above 4.
3. Load workshop 42 from trusted storage.
4. Check that it exists and still has enough space.
5. Check that the current user may reserve it.
6. Create the reservation without allowing a race to oversell it.

Changing the hidden ID to another workshop is easy. Passing `seats=4` does not prove four places remain. Format validation and business validation are different layers.

Return field errors in a shape the form can display beside the correct controls. Preserve safe values so the person can correct the request. Use a general error for a server failure rather than pretending the input was wrong.

Keep the client and server rules derived from the same contract where your stack allows it, but never skip the server check.

Use DevTools to remove `max="4"`, submit `seats=100`, and confirm the server rejects it. That is the test that matters.

Submission formats

URL encoding, multipart data, FormData, and file uploads.

URL-encoded form data

Understand the default application/x-www-form-urlencoded body and how names, values, spaces, and repeated keys are encoded.

A normal POST form uses `application/x-www-form-urlencoded` unless you choose another encoding.

```
<form action="/profile" method="post">
  <input name="displayName" value="Ada Lovelace">
  <input name="topic" value="HTML & CSS">
  <button type="submit">Save</button>
</form>
```

The request body resembles a query string:

```
displayName=Ada+Lovelace&topic=HTML+%26+CSS
```

Each name and value is encoded so spaces, ampersands, and other special characters do not break the structure. The server's form parser reverses that encoding.

This format works well for ordinary text fields. It does not carry file bytes, so forms with file inputs use multipart encoding instead.

Repeated names are valid:

```
topic=html&topic=css
```

A parser that turns the body into a simple object may keep only one value. Use an API that preserves every value when the form contract permits repetition.

Encoding is not validation or escaping. A decoded value is still untrusted input. Validate it before storing it, and escape it for the context where you later display it.

Submit a value containing spaces, `&`, `+`, and a non-ASCII character. Inspect the raw payload and the server's decoded values.

Multipart form data

Use multipart/form-data when uploading files and let the browser generate the boundary required by the request body.

Use `multipart/form-data` when a form sends files.

```
<form action="/support" method="post" enctype="multipart/form-data">
  <input name="subject">
  <input name="screenshot" type="file" accept="image/png,image/jpeg">
  <button type="submit">Send request</button>
</form>
```

The request body contains separate parts. Each part has headers and content. The text field is one part and the selected file is another.

The `Content-Type` request header includes a generated boundary:

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary...
```

That boundary tells the server where one part ends and the next begins.

When sending a `FormData` object with `fetch()`, do not set the `Content-Type` header yourself:

```
await fetch(form.action, {
  method: 'POST',
  body: new FormData(form)
})
```

The browser adds the correct header and matching boundary. Manually writing `multipart/form-data` without a boundary creates a body the server may not parse.

Multipart encoding does not make uploads safe. The server still needs total request limits, per-file limits, type checks, generated storage names, and authorization.

Inspect one multipart request in DevTools. Find the ordinary field, the file part, and the boundary in the request header.

The FormData object

Read a form into JavaScript, inspect its entries, preserve repeated values, and append or replace fields before submission.

`FormData` gives JavaScript the same name-value model used by native form submission.

Create it from the form. Include the submitter when a button started the submission:

```
const data = event.submitter
  ? new FormData(form, event.submitter)
  : new FormData(form)

for (const [name, value] of data) {
  console.log(name, value)
}
```

The entries follow successful-control rules. Unnamed and disabled fields are missing. Unchecked checkboxes are missing. File inputs produce `File` values. Passing the submitter preserves the activated button's name and value. `event.submitter` can be `null` when no button triggered the submission, so the fallback uses the one-argument constructor.

Use `get()` for a field with one value. Use `getAll()` when repeated controls share a name:

```
const topics = data.getAll('topic')
```

`append()` adds another value. `set()` replaces all current values under that name. This difference matters when building multi-select or checkbox data.

Do not convert `FormData` directly with `Object.fromEntries()` unless repeated keys are impossible. An object can hold only one property per key, so earlier values may disappear.

You can send the object directly with `fetch()`. Let the browser create the multipart header. If the endpoint expects JSON instead, convert values deliberately and remember that files need another plan.

Print the entries from a form containing a disabled input, unchecked checkbox, two checked topics, and one file. Compare the output with what you see on screen.

Redirect after a successful POST

Return a redirect after changing server state so refreshing the result page does not submit the same form again.

After a successful POST, redirect to a GET page. This is often called the Post/Redirect/Get pattern.

Imagine a form that creates a support request:

```
POST /support/requests
```

The server validates the fields, creates request 842, then responds:

```
HTTP/1.1 303 See Other
Location: /support/requests/842
```

The browser follows with:

```
GET /support/requests/842
```

The address bar now contains the result URL. Refreshing repeats the GET instead of asking to repeat the POST. The result can also be bookmarked or shared if access rules allow it.

Redirect only after the change succeeds. When validation fails, return the form with field errors and a suitable non-success status. Redirecting errors often throws away the submitted values and messages unless you store them elsewhere.

A `303 See Other` explicitly tells the client to retrieve the location with GET. Other redirect statuses have different method rules, so choose deliberately.

The server must still handle duplicate POST requests safely. A person can double-click, retry after a timeout, or resend the request outside the browser. A redirect improves navigation; it is not duplicate protection.

Submit the form, inspect the POST and following GET in the Network panel, then refresh the result page. Confirm no second record is created.

Check your understanding: forms over HTTP

Check methods, safe and idempotent behavior, success responses, redirects, common client errors, and server failures around form submission.

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

Take this interactive quiz online

JavaScript and forms

Handle events and submit data with fetch.

Handle the submit event

Listen for form submission at the form level, prevent the default only when needed, and preserve native behavior when JavaScript fails.

Listen to the form's `submit` event, not only a button click.

```
const form = document.querySelector('form')

form.addEventListener('submit', event => {
  event.preventDefault()

  const data = event.submitter
    ? new FormData(form, event.submitter)
    : new FormData(form)
  console.log([...data])
})
```

The submit event covers pointer clicks, keyboard submission, and different submit buttons. A click listener on one button misses some of those paths.

Call `preventDefault()` only when JavaScript will complete the submission another way. Without it, the browser follows the form's normal `action` and `method`. That native behavior is a useful baseline.

Keep the HTML complete:

```
<form action="/feedback" method="post">
  <!-- labeled controls -->
  <button type="submit">Send feedback</button>
</form>
```

If the script fails to load, the form still reaches the server. JavaScript can enhance the interaction with inline results, but it should not be the only place that knows the endpoint.

Do not skip server validation because `form.checkValidity()` passed. The event handler runs in a client the visitor controls.

Add two named submit buttons and log `event.submitter`. Then submit once by click and once with the keyboard. Check which button value appears in `FormData`.

Submit a form with fetch

Send FormData or JSON with fetch, inspect the HTTP status, and update the interface only after the server confirms the result.

Use `fetch()` when you want to submit in the background and update the current page.

Start from a working native form, then enhance its submit event:

```
form.addEventListener('submit', async event => {
  event.preventDefault()

  const data = event.submitter
    ? new FormData(form, event.submitter)
    : new FormData(form)

  const response = await fetch(form.action, {
    method: form.method,
    body: data,
    headers: {
      Accept: 'application/json'
    }
  })

  if (!response.ok) {
    throw new Error(`HTTP ${response.status}`)
  }

  const result = await response.json()
  console.log(result)
})
```

`fetch()` resolves when it receives an HTTP response, even when the status is 400 or 500. Check `response.ok` or `response.status` before treating the operation as successful.

The response format is part of the endpoint contract. If the client asks for JSON, the server should return JSON for both success and expected validation failures. Do not call `response.json()` on an HTML error page and hide the resulting parse error from the user.

Network failures reject the promise. Handle them separately from validation errors because the person may need to retry without changing any fields.

While the request is pending, disable the relevant submitter and show a clear status. Restore it in a `finally` block so an error does not leave the form stuck.

The server must still validate, authorize, and protect against duplicates. `fetch()` changes the interface, not the trust boundary.

Test one success, one validation response, one server error, and offline mode in DevTools.

Pending, success, and error states

Design explicit submission states so a visitor knows whether the form is working, succeeded, or needs attention.

A JavaScript-enhanced form has at least four states: idle, pending, successful, and failed.

Make those states visible in the interface:

```
<button type="submit">Send message</button>
<p id="form-status" aria-live="polite"></p>
```

When submission starts, change the button label to "Sending…" and disable that submitter. Put a short message in the live region so the state is not communicated only through a spinner.

Wait for the server response before showing success. Receiving a network response does not always mean the operation succeeded, so inspect the status and expected response body.

Validation failure is different from network failure:

- A validation response should identify fields and preserve their values.
- A network failure should explain that the request could not be completed and allow a retry.
- A server failure should avoid blaming the person's input.

Do not clear the form when the request starts. If it fails, the person would lose their work. Reset only after confirmed success and only when an empty form makes sense.

Use a `finally` block to restore controls. Otherwise one thrown error can leave the button disabled forever.

Disabling a button reduces accidental double-clicks, but it does not make the endpoint duplicate-safe. Requests can be retried by the browser, network, or user.

Throttle the connection in DevTools and watch every transition. Then force an error and confirm the form returns to a usable state.

Reset after success

Clear a completed form at the right time and understand the difference between current values, default values, and calling reset.

Call `form.reset()` only after the server confirms success and when clearing the form matches the task.

```
const response = await submitForm()

if (response.ok) {
  form.reset()
}
```

Reset restores every control to its initial value from the HTML. It does not always produce an empty form:

```
<input name="country" value="Italy">
<input name="newsletter" type="checkbox" checked>
```

After `reset()`, the text returns to `Italy` and the checkbox becomes checked again.

This is useful for a "send another message" form. It is wrong for an account settings form where the current saved values should remain visible.

Do not reset after a validation or network error. Preserve the values so the person can fix one field or retry.

Custom interface state may need separate cleanup. A character counter, preview, custom error message, or `aria-invalid` attribute does not necessarily reset itself. Listen for the form's `reset` event or clear that state in the same success path.

File inputs are also cleared by reset. Confirm that behavior before resetting a form containing a large upload a person may not want to select again.

Set default values in a small test form, edit them, call `form.reset()`, and note what returns. Then decide whether reset matches the real task.

Check your understanding: asynchronous form code

Check promises, async functions, rejection handling, task ordering, concurrent work, and the common mistakes behind unreliable form submissions.

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

Take this interactive quiz online

Server-side safety

Validate input and protect form endpoints from abuse.

Never trust submitted input

Treat every field, filename, header, and hidden value as untrusted data and validate it against the server's own rules.

The browser interface is not a security boundary. The server receives a request, not proof that someone used your form as intended.

A client can:

- omit required fields
- repeat a field several times
- add fields that do not exist in the page
- change hidden and disabled values
- skip browser validation
- send a different content type
- upload arbitrary bytes under a familiar filename

Start each endpoint with an explicit contract. For a profile update, it might accept `displayName` as a string of 1–80 characters and `timezone` as one value from a known list. Everything else is ignored or rejected.

Validate in layers:

1. Check that the request body is small enough to parse safely.
2. Accept only the expected content type.
3. Parse with a maintained parser.
4. Check required fields and data types.
5. Check lengths, ranges, and allowed values.
6. Check relationships and business rules.
7. Check that the authenticated user may perform the operation.

Syntax and meaning are different. `2026-07-30` may be a valid date string but still be unavailable for a booking. A numeric `productId` may exist but belong to another account.

Validation also does not replace safe output and database APIs. Escape untrusted text for the HTML context where it appears. Use parameterized database queries rather than joining input into SQL.

Do not accept a trusted value from the form when the server already knows it. Read the current user from the authenticated session. Read an item's price from the database. Treat a submitted price or owner ID as a claim to verify, not a command.

Return useful field errors for ordinary mistakes. Log repeated or structurally impossible failures without recording secrets.

Take one endpoint and write its accepted fields, types, limits, allowed values, and authorization rule. Then send one request that violates each rule.

Protect cookie-authenticated forms from CSRF

Understand cross-site request forgery and combine SameSite cookies, CSRF tokens, and origin checks according to the application's needs.

Cross-site request forgery matters when the browser attaches authentication automatically, usually through cookies.

Imagine you are signed in to `bank.test`. Another site can create a form that posts to `https://bank.test/transfers`. The browser may attach matching bank cookies even though the form came from another site.

The request is authenticated, but the user did not intend it.

Start with the HTTP contract:

- GET and other safe requests must not change data.
- State-changing requests use POST or another appropriate method.
- Session cookies use a suitable `SameSite` policy.
- Sensitive endpoints require a CSRF defense chosen for the application's architecture.

A common server-rendered pattern uses a random token tied to the user's session:

```
<form action="/account/email" method="post">
  <input name="csrfToken" type="hidden" value="random-session-token">
  <!-- email field and submit button -->
</form>
```

The server compares the submitted token with the expected token before changing the email. The

attacker can submit a form, but cannot read a correctly protected page to obtain the token.

The token must be unpredictable, checked on the server, and handled without leaking it into logs or external URLs. A hidden input is not secret from the person using the page; its value is useful because another origin cannot normally read it.

Origin checks can add another layer. Verify the `Origin` header, or carefully fall back to `Referer` when your design requires it. Do not accept a missing or mismatched origin blindly on sensitive operations.

`SameSite` cookies reduce risk but are not a reason to ignore the rest of the design. Cross-origin requirements, older clients, and application architecture affect the right combination.

CSRF defenses do not fix cross-site scripting. Script running in your own origin may be able to read tokens and submit requests.

Try posting to a protected endpoint without a token, with a wrong token, and with the correct token. Only the last request should change state.

Limit abuse and automated submissions

Add rate limits, honeypots, challenge widgets, and monitoring without making the form unnecessarily hostile to real people.

Public forms attract spam, repeated submissions, and automated abuse. No single control separates every person from every bot.

Start by defining what abuse costs. A contact form may send email. A password-reset form may send messages and reveal account behavior. A file form consumes storage and processing time. Protect the expensive operation, not only the HTML page.

Apply server-side rate limits using signals that fit the endpoint. An IP address is useful but imperfect because many people can share one address and attackers can rotate addresses. For authenticated actions, an account or user ID is often a stronger key. Sensitive flows may need both.

When a limit is exceeded, return `429 Too Many Requests` and a generic message. Keep enough server-side logging to see patterns without storing submitted secrets.

A honeypot can catch simple automation:

```
<div hidden>
  <label for="company-website">Leave this field empty</label>
  <input id="company-website" name="companyWebsite" tabindex="-1" autocomplete="off">
</div>
```

The server quietly rejects or discards submissions that fill it. Do not rely on a honeypot alone; capable bots can detect it.

Add a challenge such as Turnstile only when simpler controls are not enough. A challenge adds friction and can fail for real people, so use it where the abuse cost justifies it.

Other useful layers include request-size limits, duplicate detection, minimum completion time used as a signal, account verification, and queues for expensive work.

Do not block solely because a request is fast or uses an unfamiliar browser. Signals can be wrong. Combine them and monitor the result.

Test a burst of submissions, repeated identical content, a filled honeypot, and a valid slow submission.

Confirm the endpoint limits abuse without losing ordinary messages.

Handle file uploads safely

Limit upload size, verify content, generate server-side filenames, isolate storage, and avoid serving untrusted files as executable content.

A file upload crosses several trust boundaries at once. The request contains user-controlled bytes, a user-controlled filename, and a user-controlled declared media type.

No single check makes that safe. Use layers.

Limit the request early

Set a total request limit before buffering the complete body. Set a per-file limit too. A 20 MB compressed archive can expand far beyond 20 MB, so avoid accepting archive formats unless the product truly needs them and you can inspect them safely.

Allow only required formats

Start from a small allowlist. If the feature needs profile photos, accepting JPEG and PNG is easier to defend than accepting every file type.

Do not trust `file.type`, the multipart `Content-Type`, or the filename extension. They are hints supplied by the client. Inspect the file signature and parse the content with a suitable maintained library. For images, decoding and re-encoding can produce a clean output for the formats you support.

Control storage

Generate a random storage name. Keep the original name only as untrusted metadata when the interface needs to display it.

Store uploads outside executable application directories. Ideally, serve public files from an isolated origin or through a download handler that maps a record ID to the stored object.

Set a correct response `Content-Type`. For content that should not render in the browser, use a download response. Active formats such as HTML and SVG need extra care because they can execute or reference other content when served incorrectly.

Check permission and lifecycle

Verify that the current user may upload and later access the file. Scan files when the risk warrants it. Define retention, deletion, and storage quotas so abandoned uploads do not grow forever.

Process uploads in a restricted environment. A parser bug should not grant access to application secrets or the main filesystem.

Test an oversized file, a renamed executable, a valid image with a strange filename, two files with the same original name, and an unauthorized download. The visible file picker cannot cover these server-side cases.

Build a complete feedback form

Combine accessible HTML, browser validation, FormData, HTTP handling, server validation, security controls, and useful result states.

Build a feedback form that works from the browser all the way to a server decision.

The form collects:

- name
- email address
- category
- message
- optional screenshot

Start with native HTML

Give every control a visible label and stable name. Group related choices with `fieldset` and `legend`. Add concise help before the fields that need it.

Use `/feedback` as the action and POST as the method. Add multipart encoding for the screenshot. The form must work when JavaScript is unavailable.

Define the server contract

Write down the accepted fields before implementing the endpoint.

Validate required values, lengths, the category allowlist, and screenshot limits. Treat the original filename and media type as untrusted. Generate a storage name and keep the upload outside executable application files.

Return field errors without erasing valid values. After a successful native POST, redirect to a GET confirmation page.

Add CSRF protection if the form uses cookie authentication. Add request-size and rate limits. Do not log the message body, token, or uploaded bytes by default.

Add JavaScript as an enhancement

Intercept the form's submit event and send `FormData` with `fetch()`. Show idle, pending, success, validation-error, and network-error states.

Disable the active submitter while the request is pending, but make the endpoint safe if the request arrives twice. Reset only after confirmed success.

If the script fails, the native submission path must still work.

Test the failures

Complete the form with only a keyboard. Then test:

- empty and overlong values
- a category not present in the page
- a large file and a renamed non-image
- a missing or invalid CSRF token when required
- repeated submission
- offline mode and a server error
- refresh after success
- 200% zoom and narrow screens

You are done when each failure produces an understandable result, preserves safe work, and leaves the form usable.

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