JavaScript Reference

Use supporting JavaScript for interaction, filtering, accessible updates and data presentation.

- Connecting JavaScript and Selecting Elements
- Handling Events and User Input
- Filtering and Sorting Displayed Data
- Updating Interfaces Accessibly
- Presenting PHP Data with JavaScript Charts

Connecting JavaScript and Selecting Elements

JavaScript can add interaction to an HTML interface. Keep the page usable when supporting scripts fail.

Connect a script

Place this before the closing `body` tag:

```
<script src="js/app.js"></script>
```

Or use `defer` in the page head:

```
<script src="js/app.js" defer></script>
```

Select an element

```
<p id="status">Waiting for an action.</p>
```

```
const statusMessage = document.querySelector("#status");

if (statusMessage) {
  statusMessage.textContent = "JavaScript loaded.";
}
```

Use meaningful identifiers and check that an element exists before using it.

Check

- [ ] Script path is correct.

- [ ] Browser console has no error.
- [ ] Text is inserted with `textContent` when HTML is unnecessary.
- [ ] Core information remains available without the script.

Handling Events and User Input

Events allow a script to respond to user actions.

```
<label for="team-filter">Filter teams</label>
<input id="team-filter" type="search">
<p id="filter-status" role="status"></p>
```

```
const filter = document.querySelector("#team-filter");
const status = document.querySelector("#filter-status");

filter.addEventListener("input", () => {
  const value = filter.value.trim();
  status.textContent = value
    ? `Filtering by: ${value}`
    : "Showing all teams";
});
```

Do not use JavaScript as the only validation for server-side operations. PHP must independently validate submitted values.

Accessible interactions

Use native buttons for actions, retain keyboard behaviour, give controls labels and announce important updates through an appropriate status region.

Check

- [ ] Event is attached to the intended control.
- [ ] Empty input is handled.
- [ ] Keyboard users can perform the action.

- [ ] Server-side validation still exists.
- [ ] Status changes are understandable.

Filtering and Sorting Displayed Data

Client-side filtering can help users explore data already present in the browser. Database filtering remains preferable for large or permission-sensitive datasets.

```
const rows = [...document.querySelectorAll("[data-team-row]")];
const search = document.querySelector("#team-search");

search.addEventListener("input", () => {
  const query = search.value.trim().toLowerCase();
  let visibleCount = 0;

  for (const row of rows) {
    const team = row.dataset.team.toLowerCase();
    const visible = team.includes(query);
    row.hidden = !visible;
    if (visible) visibleCount++;
  }

  document.querySelector("#result-count").textContent =
    `${visibleCount} results shown`;
});
```

This demonstrates iteration and selection, but JavaScript itself is supporting technology. Explain where authoritative data processing occurs.

Check

- [ ] Original data is safely rendered.
- [ ] Case and empty searches are handled.
- [ ] Result count updates.
- [ ] Hidden content is not treated as access control.

- [ ] Large datasets are processed server-side.

Updating Interfaces Accessibly

Dynamic interfaces must communicate changes without removing keyboard access or context.

Status message

```
<p id="save-status" role="status" aria-live="polite"></p>
```

```
const status = document.querySelector("#save-status");
status.textContent = "Changes saved.";
```

Use `role="alert"` for urgent errors, not routine success messages.

Toggle a region

```
<button id="details-button" aria-expanded="false"
        aria-controls="details-panel">Show details</button>
<section id="details-panel" hidden>...</section>
```

```
const button = document.querySelector("#details-button");
const panel = document.querySelector("#details-panel");

button.addEventListener("click", () => {
  const opening = panel.hidden;
  panel.hidden = !opening;
  button.setAttribute("aria-expanded", String(opening));
  button.textContent = opening ? "Hide details" : "Show details";
});
```

Check

- [ ] Native controls are used.
- [ ] State is conveyed programmatically and visibly.
- [ ] Focus is not unexpectedly moved.
- [ ] Interface works with keyboard input.
- [ ] Essential content is not available only through animation or colour.

Presenting PHP Data with JavaScript Charts

PHP can query and process database data, then pass a small prepared result to JavaScript for presentation.

```
<?php
$labels = array_column($rows, "team_name");
$points = array_map("intval", array_column($rows, "points"));
?>
<script>
const labels = <?= json_encode($labels) ?>;
const points = <?= json_encode($points) ?>;
</script>
```

Use a chart library or your own display code with these arrays. Do not construct JavaScript by concatenating unescaped database text.

Preserve meaning

Include:

- descriptive chart title
- labels and units
- adequate contrast
- predictable category order
- a table or textual alternative
- an empty-data message

The chart should visualise a result that responds to a user need or success criterion. JavaScript is one presentation option, not the evidence by itself.

Check

- ☐ PHP output uses `json_encode()`.

- [ ] Data is already filtered or aggregated appropriately.
- [ ] Chart has a non-visual alternative.
- [ ] Empty and extreme values are tested.
- [ ] Script errors do not hide all useful output.