Intermediate Web Development

Outline

Foundations: HTML & Semantics

Why semantics matter

- Improves accessibility (screen readers, keyboard navigation).
- Boosts SEO and crawler understanding.
- Separates structure from presentation for maintainability.

Document skeleton and best practices

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
      initial-scale=1">
  <title>App  Feature</title>
  <link rel="stylesheet" href="styles.css">
  <script defer src="main.js"></script>
</head>
<body>
  <header>...</header>
  <main>...</main>
  <footer>...</footer>
</body>
```

CSS: Layouts, Architecture, and Patterns

Box model recap and sizing strategy

- Content — padding — border — margin.
- Use `box-sizing: border-box` for predictable sizes.
- Prefer relative units for responsive sizing: `rem`, `%`, `vw`/`vh`.

Practical reset and base

```
/* Reset-ish base */
* { box-sizing: border-box; }
html, body { height: 100%; margin: 0; font-family:
    system-ui, -apple-system; }
:root {
  --gap: 1rem;
  --radius: 6px;
  --bg: #fafafa;
  --text: #111;
}
body { background: var(--bg); color: var(--text);
    line-height: 1.5; }
```

Flexbox vs Grid — when to use each

- Flexbox: 1D layout — rows or columns (align and distribute space).
- Grid: 2D layout — complex multi-row/column designs.
- Combine both: Grid for page, Flex for header/toolbars/components.

Example responsive grid + flex

```
.container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 16px;
}
.header { grid-column: 1 / -1; display:flex; justify-
    content:space-between; }
.main { grid-column: 1 / 9; }
.sidebar { grid-column: 9 / -1; }
@media (max-width: 900px) {
  .main, .sidebar { grid-column: 1 / -1; }
  .header { flex-direction: column; gap: 0.5rem; }
}
```

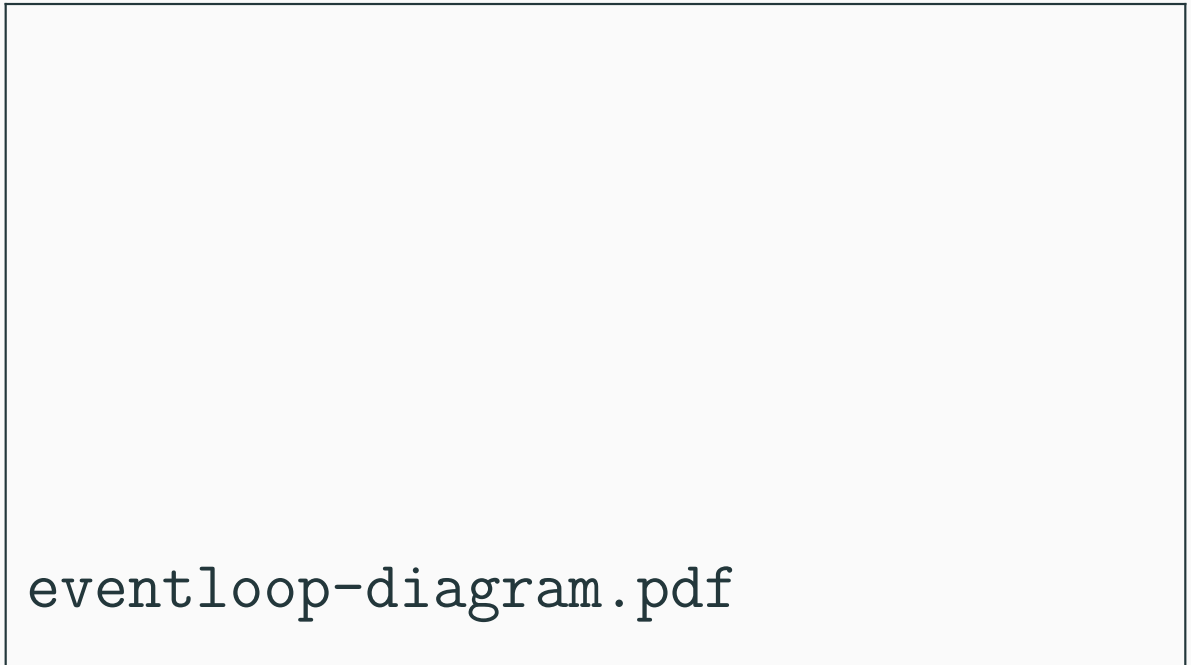
CSS Architecture: BEM, componentization, and variables

- BEM: block__element–modifier for predictable selectors.
- CSS custom properties work at runtime (theming).
- Use preprocessor (SCSS) for complex logic, but prefer native features where possible.

JavaScript: Core Concepts and Browser Internals

JavaScript in the browser: runtime overview

- Single-threaded JS engine (V8, SpiderMonkey, JavaScriptCore).
- Web APIs provide async capabilities (fetch, timers, DOM events).
- Event loop coordinates call stack, microtask queue, and task queue.



Event loop: tasks vs microtasks

- **Tasks** (macrotasks): `setTimeout`, user events, I/O callbacks.
- **Microtasks**: `Promise.then`, `queueMicrotask`; they run after the current task completes and before the next task.
- Ordering matters: microtasks can starve tasks if abused.

Example: microtask vs macrotask

```
console.log('start');

setTimeout(() => console.log('timeout'), 0);

Promise.resolve().then(() => console.log('promise'));

console.log('end');
// Output:
// start
// end
// promise
// timeout
```

Async/await best practices

```
async function loadAll(urls) {
  // serial (slower if independent)
  for (const url of urls) {
    const res = await fetch(url);
    console.log(await res.json());
  }

  // parallel
  const promises = urls.map(u => fetch(u));
  const responses = await Promise.all(promises);
  // then process responses
}
```

- Use `Promise.all` for independent requests.
- Use `Promise.allSettled` if you need partial success

Memory and performance considerations

- Avoid large DOM trees and frequent reflows — batch DOM reads/writes.
- Use document fragments for bulk DOM insertions.
- Monitor memory leaks: detached DOM nodes, forgotten timers, or global caches.

DOM batch update pattern

```
// Bad: repeatedly touching DOM in a loop
items.forEach(item => {
  const el = document.createElement('div');
  el.textContent = item;
  list.appendChild(el);
});

// Better: use a fragment
const frag = document.createDocumentFragment();
items.forEach(item => {
  const el = document.createElement('div');
  el.textContent = item;
  frag.appendChild(el);
});
list.appendChild(frag);
```

Rendering, Critical Path, and Progressive Enhancement

Critical rendering path — what to optimize

- Browser parses HTML $\rightarrow$ builds DOM; parses CSS $\rightarrow$ builds CSSOM.
- JS can block parsing if included in the head without `defer` or `async`.
- Minimize initial CSS/JS to reduce Time to First Paint (TTFP) and Largest Contentful Paint (LCP).

Example: optimize script delivery

```
<!-- Blocking -->
<script src="bundle.js"></script>

<!-- Non-blocking (defer: preserves execution order)
    -->
<script src="bundle.js" defer></script>

<!-- Async: executes as soon as available (order not
    preserved) -->
<script src="analytics.js" async></script>
```

Server-side rendering (SSR) vs Client-side rendering (CSR)

- SSR: HTML generated on the server — faster first paint and better SEO.
- CSR: initial HTML lightweight, JS hydrates UI — better for highly interactive apps.
- Hybrid: pre-render or use streaming SSR for best of both worlds.

Modules, Tooling, and Build Systems

ES Modules and code structure

- Use ESM (‘import‘ / ‘export‘) for clean dependency graphs.
- Tree-shaking works best when modules are statically analyzable.
- Prefer named exports for clarity and tooling support.

Small module example

```
// utils/math.js
export function clamp(v, a, b) { return Math.max(a,
    Math.min(b, v)); }

// components/widget.js
import { clamp } from '../utils/math.js';
export function createWidget(el) {
  // ...
}
```

Bundlers: Webpack, Rollup, Vite — when to pick

- **Vite**: very fast dev server, uses native ESM in dev.
- **Webpack**: feature-rich ecosystem; complex configuration for big apps.
- **Rollup**: great for libraries and tree-shaking.

Example npm scripts and basic Vite setup

```
# package.json scripts
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

Notes:

- Keep build steps simple and reproducible.
- Use environment variables for different build targets.

Testing, Linting, and CI

Quality tools: linting and types

- ESLint for code style and finding bugs early.
- Prettier for consistent formatting.
- TypeScript for type safety and better refactoring.

Testing pyramid

- Unit tests: fast, isolated logic tests (Jest).
- Integration tests: how modules work together.
- End-to-end tests: full user flows (Playwright, Cypress).

Simple GitHub Actions CI snippet

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with: { node-version: 18 }
      - run: npm ci
      - run: npm run build
      - run: npm test
```

Performance Tuning

Critical metrics and tools

- Core Web Vitals: LCP, FID (or INP), CLS.
- Tools: Lighthouse, WebPageTest, Chrome DevTools Performance panel.
- Focus: reduce main-thread blocking work and large layout shifts.

Strategies: code splitting, lazy loading, and caching

- Code-splitting: split by route or heavy components.
- Lazy-load images (native ‘loading="lazy"‘) and media.
- Use service workers for offline-first patterns and caching strategies (stale-while-revalidate).

Example: dynamic import (code-splitting)

```
// only load heavy charting library when needed
async function showChart(data) {
  const { Chart } = await import('./vendor/chart-lib.
      js');
  new Chart('#canvas', { data });
}
```

Security: XSS, CSP, and auth basics

XSS and sanitization

- Different types: reflected, stored, DOM-based.
- Sanitize or escape user input on render; do server-side validation.
- Use templating engines or safe libraries; avoid ‘innerHTML‘ with untrusted data.

Content Security Policy (CSP) example

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' https://apis.example.com;
  style-src 'self' 'unsafe-inline';
  img-src 'self' data:;
```

- CSP reduces XSS risk by limiting allowed sources.
- Start in report-only mode to measure impact before enforcement.

Authentication session security

- Use secure, httpOnly cookies or token-based auth (JWT) with care.
- Protect against CSRF (tokens, SameSite cookies).
- Rotate keys, enforce least privilege on APIs.

Accessibility (a11y) in detail

Foundations: semantics, keyboard, and focus management

- Proper semantic markup is the first step.
- Ensure logical tab order and visible focus styles.
- Manage focus when opening dialogs or navigating content dynamically.

Accessible modal pattern (simplified)

```
<div role="dialog" aria-modal="true" aria-labelledby=
    "dlg-title">
  <h2 id="dlg-title">Dialog title</h2>
  <button class="close">Close</button>
  <p>Content...</p>
</div>
```

Key points:

- Trap focus inside the dialog while open.
- Return focus to the triggering control when closed.

Testing accessibility

- Automated tools (Axe, Lighthouse) catch common issues.
- Manual testing: keyboard-only, screen readers (NVDA/VoiceOver), color contrast checks.
- Involve users with disabilities when possible — real-world testing matters most.

Deployment, CI/CD, and Observability

Hosting options and tradeoffs

- Static hosting (Vercel, Netlify): fast, cheap, great for JAMstack.
- Server-hosted/containers: more control, backend integration.
- Edge functions for low-latency personalization and SSR.

CI/CD pipeline essentials

- Automate tests and builds on PRs.
- Run linting, type-checks, and security scanning.
- Deploy artifacts with versioning and canary/blue-green strategies for production.

Monitoring, logging, and user telemetry

- Capture client-side errors (Sentry), performance metrics (RUM).
- Set alerts for regressions (LCP spikes, error rate increases).
- Use feature flags to control rollouts and mitigate failures.

Patterns, examples, and a mini-project outline

Project structure recommendation

- `src/` — components, pages, utilities
- `public/` — static assets
- `tests/` — unit and e2e tests
- `build/` — generated output

Component example (vanilla JS web component)

```
class MyCard extends HTMLElement {
  constructor() {
    super();
    const root = this.attachShadow({ mode: 'open' });
    root.innerHTML = `<style> :host{ display:block }</
        style>
      <slot></slot>`;
  }
}
customElements.define('my-card', MyCard);
```

- Web Components provide encapsulation without frameworks.
- Shadow DOM prevents style leakage.

Design tokens and theming

- Store design values centrally: colors, spacing, type scale.
- Use CSS variables for runtime theming and easy overrides.
- Keep tokens atomic and consistent for scalable systems.

Resources and further learning

Canonical resources

- MDN Web Docs (HTML/CSS/JS references).
- Web Fundamentals and Lighthouse docs.
- JavaScript.info, CSS-Tricks, A11y Project.

Summary

- HTML structures content; CSS styles it; JS adds behavior.
- Intermediate topics: event loop, rendering pipeline, SSR vs CSR, security, accessibility.
- Tooling and processes (testing, linting, CI/CD) are essential for production apps.

Q & A

Questions? Tell me which section you'd like expanded into a demo or speaker notes.