DOM manipulation best practices

DOM manipulation is a critical part of front-end development, but inefficient handling can lead to performance issues, poor user experience, and maintenance difficulties. Below are **best practices** for working with the DOM efficiently and cleanly.

🔧 1. Minimize Direct DOM Access

Why:

Accessing the DOM is **expensive** in terms of performance.

Best Practice:

- Cache DOM queries instead of querying repeatedly.

```
// ❌ Bad
document.getElementById("btn").addEventListener("click", () ⇒ {
  document.getElementById("btn").style.color = "red";
});

// ✅ Good
const btn = document.getElementById("btn");
btn.addEventListener("click", () ⇒ {
  btn.style.color = "red";
});
```

🛑 2. Avoid Reflows and Repaints

Why:

Modifying layout properties (e.g., width, height) triggers reflows/repaints which are costly.

Best Practice:

- Batch DOM reads and writes.
- Use `documentFragment` for multiple changes.

```
// ✅ Using DocumentFragment
const list = document.getElementById("list");
const fragment = document.createDocumentFragment();

for (let i = 0; i < 100; i++) {
  const li = document.createElement("li");
  li.textContent = `Item ${i}`;
  fragment.appendChild(li);
}
list.appendChild(fragment);
```

⚡ 3. Use Class Manipulation over Style Changes

Why:

Keeps your JavaScript clean and separates concerns.

Best Practice:

- Modify classes instead of styles directly.

```
// ❌ Avoid
element.style.backgroundColor = "blue";

// ✅ Do
element.classList.add("highlight");
```

```css
/* CSS */
.highlight {
  background-color: blue;
}
```

📌 4. Use Event Delegation

Why:

Reduces the number of event listeners and improves performance.

Best Practice:

- Attach a single event listener to a parent and manage children via `event.target`.

```
document.getElementById("list").addEventListener("click", (e) => {
  if (e.target.tagName === "LI") {
    console.log("Clicked", e.target.textContent);
  }
});
```

🎨 5. Avoid Layout Thrashing

What:

When you interleave reads and writes to the DOM, causing forced reflows.

Example of Thrashing:

```
element.style.width = element.offsetWidth + 1 + "px"; // forces reflow
```

Best Practice:

- Group **reads** first, then **writes**.

🔄 6. Use requestAnimationFrame for Animations

Why:

It syncs with the browser's refresh rate (60fps) for smoother performance.

```
function animate() {
  element.style.left = element.offsetLeft + 1 + "px";
  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

🧹 7. Clean Up Event Listeners

Why:

Prevent memory leaks, especially in Single Page Applications (SPAs).

```
function handleClick() {
  console.log("Clicked");
}

button.addEventListener("click", handleClick);

// Later
button.removeEventListener("click", handleClick);
```

🧩 8. Use Template Literals and `innerHTML` Carefully

- Fast but risky: can lead to **XSS attacks** if user input is involved.
- Prefer `textContent` or `createElement` when rendering user data.

```
// ❌ Unsafe if `username` is from user input
element.innerHTML = `<div>${username}</div>`;
```

```
// ✅ Safer
const div = document.createElement("div");
div.textContent = username;
element.appendChild(div);
```

🧪 9. Use DevTools to Profile DOM Performance

- Use Chrome DevTools → Performance tab to inspect:
  - Layout/reflow costs
  - Repaints
  - Long tasks

✅ Summary

Best Practice	Why it matters
Cache DOM references	Improves performance
Batch DOM operations	Reduces reflows
Use classes for styling	Separation of concerns
Use event delegation	Fewer listeners, better performance
Avoid layout thrashing	Prevents jank and reflows
Use `requestAnimationFrame`	Smooth animations
Clean up event listeners	Prevents memory leaks
Sanitize user data before using `innerHTML`	Prevents XSS vulnerabilities

Would you like examples of good vs. bad DOM manipulation in a sample app (like a todo list)?