HTML/CSS/React

Here is a **complete interview-focused question bank** for **HTML + CSS + React** exactly as requested.

These cover **basic → moderate → advanced → scenario + UI design based questions** — the kind usually asked for **1–3 YOE Full Stack Developer roles** like yours.

🔥 HTML Interview Questions

📌 Basics (Fundamentals)

✅ 1) What is HTML? Why is it used?

■ Explanation (simple, conceptual)

HTML stands for **HyperText Markup Language**.

It is the **standard markup language** used to create the **structure** of web pages.

HTML does not perform logic; it only defines *what elements appear and how they are arranged*.

HTML builds the skeleton of the webpage using tags like:

- headings
- paragraphs
- images
- links
- forms
- videos

The browser reads the HTML and renders it visually.

■ When/Why used

✔ To define page layout and content structure

✔ Acts as the foundation on which CSS and JavaScript work

✔ Essential for SEO, accessibility, and proper rendering

✔ Used in every web application including Angular/React templates

---

⬛ Example

```html
<h1>Welcome</h1>
<p>This is a simple web page.</p>
<img src="image.png" alt="Sample Image">
```

⬛ Summary line

HTML creates the structure of a web page and acts as the foundational layer for CSS & JavaScript.

✅ 2) Explain semantic HTML with examples.

⬛ Explanation (simple, conceptual)

Semantic HTML means using **meaningful tags** that describe the purpose of content instead of generic tags.

Browsers, screen readers, and search engines understand the meaning of these elements better.

Semantic tags improve:

- Accessibility
- SEO
- Readability
- Maintainability

⬛ When/Why used

✔ To help search engines understand webpage structure (SEO)

✔ To improve screen-reader accessibility

✔ To make code more readable and maintainable

✔ Required for modern frontend frameworks and standards

■ Examples

```
<header>Top navigation and logo</header>
<nav>Menu links</nav>
<main>Main page content here</main>
<section>One topic or block</section>
<article>Independent article/blog content</article>
<aside>Sidebar content</aside>
<footer>Copyright & social links</footer>
```

Compared to older non-semantic:

```
<div id="header">...</div>
<div class="nav">...</div>
```

■ Summary line

Semantic HTML uses meaningful tags like `<header>`, `<nav>`, `<section>` to improve SEO, accessibility, and readability.

✅ 3) What is DOCTYPE?

■ Explanation (simple, conceptual)

`<!DOCTYPE html>` is a declaration that tells the browser which **HTML version** the page is written in.

In modern HTML5:

```
<!DOCTYPE html>
```

It ensures the browser renders the page in **standards mode**, not "quirks mode".

Quirks Mode = old browser behavior → incorrect layouts

Standards Mode = correct HTML5 rendering

⬛ When/Why used

✔ Ensures consistent rendering across browsers

✔ Activates HTML5 parsing rules

✔ Prevents layout issues caused by quirks mode

⬛ Example

```
<!DOCTYPE html>
<html>
<head>
  <title>Sample</title>
</head>
<body>
  ...
</body>
</html>
```

⬛ Summary line

DOCTYPE tells the browser to use HTML5 standards mode for proper rendering.

✅ 4) Difference between `<div>` and `<span>`?

⬛ Explanation (simple, conceptual)

Both `<div>` and `<span>` are non-semantic tags but differ by display behavior:

🔵 `<div>`

- **Block-level element**

- Takes full width
- Always starts on a new line
- Used to group large sections

🔵 `<span>`

- **Inline element**
- Takes only required space
- Does not start new line
- Used to style small text portions

■ When/Why used

✔ `<div>` for layout structures, containers, sections

✔ `<span>` for applying CSS to part of a sentence

✔ Both often used with CSS classes

■ Example

```html
<div class="card">
  <h2>Product</h2>
  <p>Price: <span class="highlight">$20</span></p>
</div>
```

■ Summary line

`<div>` is block-level for layout; `<span>` is inline for styling small text pieces.

✅ 5) What are HTML attributes?

■ Explanation (simple, conceptual)

HTML attributes are **additional information** added inside tags to control element behavior, appearance, or identity.

Attributes are always written in:

```
name="value"
```

Common attributes:

- `id`
- `class`
- `href`
- `style`
- `src`
- `alt`
- `disabled`
- `placeholder`

■ When/Why used

✓ To uniquely identify elements (`id`)

✓ To apply CSS or JS (`class`)

✓ To store metadata (`data-*`)

✓ To control behavior (`required`, `readonly`)

✓ To specify paths (`src`, `href`)

■ Example

```html
<img src="logo.png" alt="Company Logo" width="200">
<a href="https://google.com" target="_blank">Google</a>
<input type="text" placeholder="Enter name">
```

■ Summary line

HTML attributes provide extra information to control behavior, styling, and identification of elements.

✅ 6) Self-closing tags list

◼️ Explanation (simple, conceptual)

Self-closing tags (also called *void elements*) are HTML tags that **do not need a closing tag** because they don’t wrap content.

They act alone and directly render a single element.

◼️ When/Why used

✔️ Used when the element doesn't contain inner text/content

✔️ Makes HTML cleaner and shorter

✔️ Often used to insert media, line breaks, metadata, etc.

◼️ Common Self-Closing Tags

```
<img>
<br>
<hr>
<input>
<meta>
<link>
<source>
<area>
<col>
<param>
<base>
```

◼️ Example

```html
<img src="logo.png" alt="Company logo">
<br>
<input type="text" placeholder="Enter name">
```

◼️ Summary line

Self-closing tags do not contain inner content and include elements like `<img>`, `<br>`, `<input>`, `<meta>`, `<link>`.

✅ 7) Difference between block-level and inline elements

⬛ Explanation (simple, conceptual)

HTML elements are categorized based on how they appear on the page.

🔵 Block-level elements

- Take full width of the container
- Always start on a new line
- Can contain both inline and block elements
- Examples: `<div>`, `<p>`, `<h1>`, `<section>`

🔵 Inline elements

- Take only as much width as required
- Do *not* start on a new line
- Typically contain only text or other inline elements
- Examples: `<span>`, `<a>`, `<strong>`, `<img>`

⬛ When/Why used

✔ Block elements are used for layout and page structure

✔ Inline elements are used for styling text or small content pieces

✔ Helps in making structured, readable UI

⬛ Example

```
<div>This is block element</div>
<span>This is inline</span> more inline text here.
```

⬛ **Summary line**

Block elements take full width and start new lines, while inline elements stay within the same line.

✅ 8) What are meta tags & where are they used?

⬛ **Explanation (simple, conceptual)**

Meta tags provide **metadata** about the HTML page.

They do not appear in the visible UI but help browsers, search engines, and social platforms understand the page.

Placed inside `<head>`.

⬛ **When/Why used**

✔ SEO (search engine optimization)

✔ Encoding and viewport settings

✔ Social media previews (Open Graph tags)

✔ Browser behavior control

✔ Page description, keywords

⬛ **Example**

```
<meta charset="UTF-8">
<meta name="description" content="Online shopping site">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta property="og:title" content="My Website">
```

⬛ **Summary line**

Meta tags provide metadata for SEO, browser instructions, encoding, and social sharing, placed in the `<head>` section.

✅ 9) What is the use of `<header>`, `<footer>`, `<aside>`, `<article>`?

■ **Explanation (simple, conceptual)**

These are **semantic HTML5 layout tags** that clearly represent page sections.

🔵 `<header>`

Top section of a webpage or section

Contains logo, navbar, title.

🔵 `<footer>`

Bottom section

Contains copyright, links, contact info.

🔵 `<aside>`

Sidebar content

Related but not primary content (ads, categories, filters).

🔵 `<article>`

Independent content block

A complete article, blog post, news item, or card.

■ **When/Why used**

✔ Improve semantic structure

✔ Enhance SEO

✔ Improve screen reader accessibility

✔ Make UI easier to maintain

■ **Example**

```
<header>
  <h1>Blog Site</h1>
```

```html
</header>

<article>
  <h2>Post Title</h2>
  <p>Blog content here...</p>
</article>

<aside>
  <p>Related posts or ads</p>
</aside>

<footer>
  <p>© 2025 My Website</p>
</footer>
```

■ Summary line

Semantic tags like `<header>`, `<footer>`, `<aside>`, `<article>` structure the page meaningfully for SEO & accessibility.

✅ 10) Explain `id` vs `class`

■ Explanation (simple, conceptual)

🔵 id

- Unique identifier
- Must NOT be repeated in the page
- Used for JavaScript, anchor links, forms

🔵 class

- Can be applied to multiple elements
- Used mainly for styling and reusable CSS
- Supports multiple classes per element

⬛ When/Why used

Attribute	When Used	Why
id	Unique element	JS targeting, anchor navigation
class	Reusable styling	Apply same CSS to multiple elements

⬛ Example

```
<div id="main-header">Header</div>

<p class="text-bold text-red">Hello</p>
<p class="text-bold">World</p>
```

CSS:

```
.text-bold { font-weight: bold; }
.text-red { color: red; }
```

⬛ Summary line

`id` is unique for identifying elements; `class` is reusable for styling multiple elements.

📌 Forms & Input Handling

✅ 1) What is the use of `<form>`?

⬛ Explanation (simple, conceptual)

`<form>` is an HTML element used to **collect user input** and **send it to the server**.

It wraps input fields like textboxes, checkboxes, dropdowns, buttons, etc.

The form defines:

- where to send data (`action`)
- which HTTP method (`method`)

- input handling

◼️ **When/Why used**

✔️ Login forms

✔️ Registration forms

✔️ Search boxes

✔️ File uploads

✔️ Feedback/contact forms

◼️ **Example**

```
<form action="/submit" method="POST">
  <input type="text" name="username">
  <button type="submit">Submit</button>
</form>
```

◼️ **Summary line**

`<form>` collects user input and sends it to the server using GET or POST requests.

✅ 2) Name different input types in HTML

◼️ **Explanation (simple, conceptual)**

HTML provides multiple types of `<input>` fields to collect different kinds of data.

◼️ **Common Input Types**

```
text
password
email
number
date
time
```

```
file
checkbox
radio
range
color
tel
url
search
hidden
submit
reset
button
```

⬛ When/Why used

✔ Different inputs give correct UI controls

✔ Enable browser-side validation

✔ Improve usability and accessibility

⬛ Example

```html
<input type="email" placeholder="Enter email">
<input type="date">
<input type="checkbox">
```

⬛ Summary line

HTML supports various input types to capture different forms of data easily.

✅ 3) What is `name` attribute used for?

⬛ Explanation (simple, conceptual)

The `name` attribute uniquely identifies a form field **for the backend/server**.

Without `name`, form data **won't be submitted**.

⬛ When/Why used

✔ To map input field → backend variable

✔ For form submission

✔ Required for radio/checkbox grouping

⬛ Example

```
<input type="text" name="email">
```

Backend receives:

```
email = value
```

⬛ Summary line

`name` attribute is used to send form field values to the backend during submission.

✅ 4) How does form submission work?

⬛ Explanation (simple, conceptual)

When a form is submitted:

1️⃣ Browser collects all input values

2️⃣ Converts them into key-value pairs

3️⃣ Sends them to the URL in `action`

4️⃣ Uses HTTP method from `method`

5️⃣ Server processes and responds

⬛ When/Why used

✔ Send user data to backend

✔ Perform authentication/registration

✔ Search or filter data

◼️ Example

```
<form action="/login" method="POST">
  <input name="user">
  <input name="password">
</form>
```

The browser sends:

```
POST /login
user=abc&password=123
```

◼️ Summary line

Form submission sends collected field values to the server using GET or POST methods.

✅ 5) Difference between GET & POST request in forms

◼️ Explanation (simple, conceptual)

🔵 GET

- Data is sent in URL (query params)
- Limited length
- Less secure
- Cached

🔴 POST

- Data sent in request body
- Not shown in URL
- More secure

- No length limit
- Used for sensitive operations

⬛ When/Why used

Type	When to use	Why
GET	Search, filters, navigation	Bookmarkable, cached
POST	Login, registration, file upload	Secure, large data

⬛ Example

GET:

```
/search?query=phone
```

POST:

```
body: username=abc&password=123
```

⬛ Summary line

GET sends data in URL; POST sends data in body — POST is more secure for sensitive operations.

✅ 6) What is `required` attribute?

⬛ Explanation (simple, conceptual)

`required` is an HTML validation attribute that makes an input **mandatory** before form submission.

Browser prevents submission until field is filled.

⬛ When/Why used

✔ Mandatory fields like email, name, password

✔ Client-side validation

✔ No need for JavaScript

⬛ **Example**

```
<input type="email" required>
```

⬛ **Summary line**

`required` ensures a form field must be filled before submission.

✅ 7) How do you validate HTML fields?

⬛ **Explanation (simple, conceptual)**

HTML provides **built-in validation** using attributes:

🔵 Validation Attributes

- `required`
- `min`, `max`
- `minlength`, `maxlength`
- `pattern`
- `type="email"`
- `type="number"`
- `step`

Browser shows automatic error without JavaScript.

⬛ **When/Why used**

✔ To validate input on client-side

✔ Reduce errors before sending data

✔ Lightweight and fast

✔ No JavaScript needed

⬛ **Example**

```html
<input type="email" required>
<input type="text" pattern="[A-Za-z]+" minlength="3">
<input type="number" min="1" max="100">
```

⬛ Short summary line

HTML validation uses built-in attributes like required, pattern, minlength, type to validate fields without JavaScript.

📌 Accessibility + SEO

✅ 1) What are ARIA labels?

⬛ Explanation (simple, conceptual)

ARIA stands for **Accessible Rich Internet Applications**.

ARIA labels are special HTML attributes used to improve **accessibility** for users with disabilities — especially those using **screen readers**.

They help describe:

- Buttons
- Icons
- Inputs
- Custom components (menus, modals, alerts)

Because some UI elements (icons, SVGs, custom components) don't have text, ARIA provides meaningful descriptions.

Common ARIA attributes:

- `aria-label`
- `aria-labelledby`
- `aria-hidden`
- `aria-expanded`

- `role="button"`, `role="alert"`

■ **When/Why used**

✔ To make web apps accessible (WCAG standards)

✔ When elements don’t have visible text (icon-only buttons)

✔ To improve usability for visually impaired users

✔ Required in many enterprise apps for compliance

■ **Example**

```
<button aria-label="Close Menu">
  <img src="close.png" alt="">
</button>

<input aria-label="Search product" type="text">
```

Screen readers will read:

- “Close Menu”
- “Search product”

■ **Summary line**

ARIA labels make non-text UI elements understandable for screen readers and improve accessibility.

✅ 2) How do you make HTML SEO friendly?

■ **Explanation (simple, conceptual)**

SEO-friendly HTML helps search engines like Google read, understand, and index your content better.

This improves ranking, visibility, and click-through rate.

◼ **When/Why used**

✔ For better search engine ranking

✔ To help crawlers understand page structure

✔ To improve metadata & content readability

✔ To support social media sharing

◼ **Techniques for SEO-friendly HTML**

🔵 1. Use semantic tags

```
<header>, <main>, <section>, <article>, <footer>
```

🔵 2. Proper heading structure

Use only one `<h1>` and logical `<h2>`, `<h3>` hierarchy.

🔵 3. Meta tags

```
<meta name="description" content="Product information and reviews">
<meta name="keywords" content="phones, electronics">
```

🔵 4. Meaningful URL + Title

```
<title>Best Laptops 2025 | Tech Reviews</title>
```

🔵 5. Add `alt` text to images

Helps Google Image Search index visuals.

🔵 6. Fast loading pages

(Use optimized images + caching)

🔵 7. Mobile Responsive HTML

Using:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

⬛ Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Angular Tutorial for Beginners</title>
  <meta name="description" content="Learn Angular with step-by-step tutoria
ls">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <article>
    <h1>Angular Full Guide</h1>
    <p>Complete tutorial...</p>
  </article>
</body>
</html>
```

⬛ Summary line

SEO-friendly HTML uses semantic tags, proper headings, meta tags, alt text, and responsive design to improve search ranking.

✅ 3) Importance of `alt` in images

⬛ Explanation (simple, conceptual)

The `alt` attribute provides a **text description** of an image.

It is displayed when the image cannot load, and is read aloud by screen readers.

It improves:

- Accessibility

- SEO
- User experience

⬛ **When/Why used**

✔ **Accessibility:** Helps visually impaired users

✔ **SEO:** Search engines use `alt` text to understand image content

✔ **Faster loading fallback:** Shows text when image fails

✔ **Image indexing:** Google Images ranking depends on alt text

---

⬛ **Example**

```
<img src="phone.jpg" alt="Android smartphone with 128GB storage">
```

If image fails:

```
Android smartphone with 128GB storage
```

Screen reader reads the same text aloud.

⬛ **Summary line**

`alt` text makes images accessible and SEO-friendly by describing their content to users and search engines.

🔥 CSS Interview Questions

📌 Core Fundamentals

✅ 1) What is CSS? Why do we use it?

⬛ **Explanation (simple, conceptual)**

CSS (**Cascading Style Sheets**) is used to control **presentation & styling** of web pages.

HTML creates the structure, CSS makes it **beautiful, responsive, and usable**.

CSS controls:

- colors
- fonts
- spacing
- layout
- animations
- responsiveness

■ When/Why used

✔ To separate content (HTML) from design (CSS)

✔ To create consistent UI across pages

✔ To make websites responsive

✔ To reuse styles across components

■ Example

```css
p {
  color: blue;
  font-size: 18px;
}
```

■ Summary line

CSS styles and layouts webpages, making them visually appealing and responsive.

✅ 2) Inline vs Internal vs External CSS

■ Explanation (simple, conceptual)

🔵 Inline CSS

Written inside HTML tag

```
<p style="color:red">Hello</p>
```

🔵 Internal CSS

Written inside `<style>` tag within `<head>`

```
<style>
p { color: red; }
</style>
```

🔵 External CSS

Separate `.css` file linked to HTML

```
<link rel="stylesheet" href="styles.css">
```

⬛ When/Why used

Type	When used	Why
Inline	Quick testing, single element	Highest priority
Internal	Page-specific styles	No extra files
External	Real projects, reusable styles	Better maintainability

---

⬛ Summary line

Inline = inside tag, Internal = inside `<style>`, External = in a separate `.css` file.

✅ 3) CSS Selectors — universal, class, id, attribute, pseudo

⬛ Explanation (simple, conceptual)

Selectors tell CSS **which elements** to apply styles to.

⬛ **When/Why used**

✔️ To target specific elements

✔️ To style dynamically based on state

✔️ To design reusable or specific UI elements

⬛ **Types of Selectors**

🔵 Universal Selector

Targets all elements

```css
* { margin: 0; padding: 0; }
```

🔵 Class Selector

Reusable styles

```css
.text-red { color: red; }
```

🔵 ID Selector

Unique element only

```css
#header { background: black; }
```

🔵 Attribute Selector

Targets based on attribute

```css
input[type="email"] { border: 1px solid blue; }
```

🔵 Pseudo Classes

State-based

```css
button:hover { background: yellow; }
```

🔵 Pseudo Elements

Parts of elements

```
p::first-letter { font-size: 30px; }
```

⬛ Summary line

Selectors define which HTML elements CSS styles apply to — classes, ids, attributes, pseudo-classes/elements.

✅ 4) What is specificity? CSS priority ranking

⬛ Explanation (simple, conceptual)

Specificity determines **which CSS rule wins** when multiple rules target the same element.

Higher specificity = more priority.

⬛ Specificity Ranking (Highest → Lowest)

Rank	Selector Type
1️⃣ Inline styles (`style=""`)	
2️⃣ ID selectors (`#id`)	
3️⃣ Class, pseudo-class (`.class` , `:hover`)	
4️⃣ Tag/element selectors (`p` , `h1`)	
5️⃣ Universal selector (`*`)	

⬛ When/Why used

✔️ To resolve style conflicts

✔️ To avoid unexpected styling issues

✔️ To control CSS override rules

⬛ Example

```
p { color: blue; }          /* low */
.text { color: green; }      /* higher */
#para1 { color: red; }       /* highest */
```

Result → **red**

⬛ **Summary line**

Specificity decides which CSS rule overrides others—inline > id > class > element.

✅ 5) Box model — margin, padding, border, content

⬛ **Explanation (simple, conceptual)**

Every HTML element is treated as a **box**.

Box model defines spacing and sizing around the element.

Structure:

```
┌─────────────────┐
│   Margin      │
│ ┌───────────────┐ │
│ │   Border   │ │
│ │ ┌───────────┐ │ │
│ │ │ Padding │ │ │
│ │ │ ┌───────┐ │ │ │
│ │ │ │Content│ │ │
```

⬛ **When/Why used**

✔ To control spacing inside/outside elements

✔ For layout design

✔ For responsive UI

■ Example

```css
.box {
  padding: 10px;
  border: 2px solid black;
  margin: 15px;
}
```

■ Summary line

CSS box model defines element spacing using content, padding, border, and margin.

✅ 6) Display types — block, inline, inline-block, flex, grid

■ Explanation (simple, conceptual)

🔵 block

- Full width
- New line
- Example: `div`, `p`, `section`

🔵 inline

- Only needed width
- Same line
- Example: `span`, `a`

🔵 inline-block

- Inline element but allows width/height control
- Useful for horizontal cards/buttons

🔵 **flex** (Flexbox)

- One-dimensional layout (row *or* column)
- Best for alignment & spacing

🔵 **grid**

- Two-dimensional layout (rows *and* columns)
- Best for complex layouts

⬛ When/Why used

Type	Use Case
block	layout sections
inline	text-level elements
inline-block	navbar buttons, small cards
flex	navbars, center alignment, lists
grid	dashboards, complex UI

⬛ Example

```
.container {
  display: flex;
  gap: 10px;
}
```

⬛ Summary line

Display property controls layout: block, inline, inline-block, flex (1D), grid (2D).

✅ 7) What is position? (static, absolute, relative, sticky, fixed)

⬛ Explanation (simple, conceptual)

`position` controls **how an element is placed** in the page.

🔵 static (default)

Normal document flow.

🔵 relative

Moves element *relative to its normal position*.

🔵 absolute

Position relative to nearest ancestor with `position: relative`.

🔵 fixed

Stays fixed relative to viewport (scrolling doesn't affect it).

🔵 sticky

Behaves like relative but sticks when scrolling reaches a threshold.

⬛ When/Why used

✔ sticky headers

✔ tooltips (absolute)

✔ fixed navbars

✔ overlays / modals

⬛ Example

```css
.header {
  position: sticky;
  top: 0;
}
```

⬛ Summary line

CSS position controls how elements float on the page—static, relative, absolute, fixed, sticky.

✅ 8) Z-index — what is it? Why needed?

⬛ Explanation (simple, conceptual)

`z-index` controls **stacking order** of elements on top of each other.

Higher z-index = appears on top.

Works only with positioned elements (`relative`, `absolute`, etc.).

⬛ When/Why used

✔️ To place dropdown over other elements

✔️ For modals, overlays, popups

✔️ For layered UI components

⬛ Example

```
.modal {
  position: absolute;
  z-index: 999;
}
```

⬛ Summary line

Z-index manages which element appears on top in overlapping layers.

📌 Responsive & Layout

✅ 1) What are media queries?

⬛ Explanation (simple, conceptual)

Media queries are CSS rules that apply styles **only when certain conditions are met**, such as screen size, device width, orientation (portrait/landscape), or resolution.

They allow the layout to **adapt** to:

- mobile screens

- tablets
- desktops
- large monitors

Media queries check the device’s characteristics and apply the matching CSS block.

■ When/Why used

✔ To create responsive web designs

✔ To adjust layout based on screen width

✔ To hide/show elements on smaller screens

✔ To reduce font sizes or adjust spacing for mobile

✔ To apply dark mode/light mode

■ Example

```css
/* For mobile screens (max width 600px) */
@media (max-width: 600px) {
  .container {
    flex-direction: column;
    padding: 10px;
  }
}

/* For tablets */
@media (min-width: 601px) and (max-width: 900px) {
  .container {
    padding: 20px;
  }
}
```

■ Summary line

Media queries detect screen/device size and apply responsive styles accordingly.

✅ 2) Flexbox full concepts — justify-content, align-items, wrap, gap

■ Explanation (simple, conceptual)

Flexbox is a **one-dimensional layout system** used to align items horizontally OR vertically within a container.

It focuses on **space distribution**, **alignment**, and **flexibility**.

🔵 Core Flexbox Concepts

1. justify-content (horizontal alignment in row direction)

Controls alignment **along the main axis** (row by default).

Values:

- `flex-start` → items start at left
- `center` → centered horizontally
- `flex-end` → items right aligned
- `space-between` → equal space between
- `space-around` → equal outer spacing
- `space-evenly` → equal spacing everywhere

---

2. align-items (vertical alignment in row direction)

Controls alignment **on cross-axis**.

Values:

- `stretch` (default)
- `center`
- `flex-start`
- `flex-end`

- `baseline`

---

3. flex-wrap

By default, flex items try to fit in one line.

`wrap` allows items to move to next row/column.

Values:

- `nowrap` (default)
- `wrap`
- `wrap-reverse`

---

4. gap

Sets uniform spacing between flex items without using margin.

■ When/Why used

✔ Navbars

✔ Card layouts

✔ Aligning items centrally

✔ Responsive row/column layouts

✔ Replacing floats/tables

■ Example

```css
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
  align-items: center;
  gap: 20px;
}
```

■ Summary line

Flexbox simplifies one-dimensional layouts using justify-content, align-items, wrap, and gap.

✅ 3) CSS Grid properties

■ Explanation (simple, conceptual)

CSS Grid is a **two-dimensional layout system** allowing layout in **rows and columns simultaneously**.

Perfect for complex, large-scale page structures.

🔵 Major Grid Properties

1. display: grid

Enables grid layout.

2. grid-template-columns / grid-template-rows

Defines number and size of rows/columns.

```css
grid-template-columns: 200px 1fr 1fr;
```

3. gap

Spacing between grid cells.

4. grid-template-areas

Define named areas — cleaner layout structure.

5. justify-items / align-items

Controls item alignment inside cells.

6. justify-content / align-content

Align entire grid within container.

7. grid-column / grid-row

Allows an item to span across multiple rows/columns.

⬛ When/Why used

✔️ Dashboards

✔️ Photo galleries

✔️ Page layouts with rows & columns

✔️ Complex card layouts

✔️ Template-style UI

⬛ Example

```css
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: auto;
  gap: 20px;
}
```

⬛ Summary line

CSS Grid handles two-dimensional layouts with easy control over rows, columns & spacing.

✅ 4) Difference between Flex vs Grid

⬛ Explanation (simple, conceptual)

Feature	Flexbox	Grid
Layout Direction	One-dimensional	Two-dimensional
Axes	Single (row OR column)	Both (rows AND columns)
Best For	Navbars, cards, alignment	Page layout, dashboards
Item Placement	Content-driven	Layout-driven
Complexity	Simple	Advanced

■ **When/Why used**

✔ Use **Flexbox** for aligning items in a single line

✔ Use **Grid** for full-page layout and multi-row/column structures

---

■ **Example**

Flex:

```
display: flex;
```

Grid:

```
display: grid;
grid-template-columns: repeat(3, 1fr);
```

■ **Summary line**

Flexbox is 1D (row/column) while Grid is 2D (rows + columns).

✅ 5) How to make UI responsive across devices?

■ **Explanation (simple, conceptual)**

Responsive design ensures UI adjusts across:

- mobile
- tablet
- desktop
- large screens

Achieved using flexible layouts and dynamic CSS rules.

■ **Techniques**

✔ 1. Media queries

Adjust layout per device width.

✔ 2. Flexible units

Use:

- `%`
- `vw`, `vh`
- `rem`, `em`
- `fr` (grid)

✔ 3. Flexbox & Grid

Build fluid layouts that adapt automatically.

✔ 4. Responsive images

```
<img src="..." style="max-width:100%;">
```

✔ 5. Mobile-first approach

Write mobile CSS → scale up.

✔ 6. Hide/show elements on small screens

```
@media(max-width:600px) { .desktop-only { display:none; } }
```

■ When/Why used

✔ Better user experience

✔ Required for modern devices

✔ Google ranking (SEO)

✔ Ensures UI doesn't break on small screens

■ Summary line

Responsive UI uses media queries, flexible units, and flex/grid layouts to adapt across devices.

✅ 6) Mobile-first vs Desktop-first approach

⬛ Explanation (simple, conceptual)

🔵 Mobile-first (recommended approach)

You design for mobile first, then expand for larger screens.

Use **min-width** media queries.

Example:

```
@media (min-width: 768px) { ... }
```

🔵 Desktop-first

Design desktop layout first, then shrink for mobile.

Uses **max-width** media queries.

Example:

```
@media (max-width: 768px) { ... }
```

⬛ When/Why used

Approach	When used	Why
Mobile-first	Modern apps	Most traffic is mobile; cleaner scaling
Desktop-first	Legacy apps	Desktop-heavy business tools

⬛ Summary line

Mobile-first starts with small screens using min-width; desktop-first starts big and scales down using max-width.

✅ 7) What is viewport?

⬛ Explanation (simple, conceptual)

Viewport is the **visible area of a webpage** inside the browser window.

Mobile screens have smaller viewports, so web pages must adapt.

To make responsive sites, we set viewport meta tag.

⬛ When/Why used

✔ Required for responsive design

✔ Controls scaling on mobile devices

✔ Ensures media queries behave correctly

⬛ Example (very important)

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Meaning:

- `width=device-width` → layout matches device screen
- `initial-scale=1.0` → no zoom on load

⬛ Summary line

Viewport is the visible browser area; setting viewport meta tag makes pages responsive on mobile.

📌 Styling & Effects

✅ 1) Pseudo elements — `::before` / `::after`

⬛ Explanation (simple, conceptual)

Pseudo-elements allow you to **insert virtual elements** before or after real HTML elements **without adding extra tags**.

Most commonly used pseudo-elements are:

🔵 `::before`

Inserts content **before** the element.

🔵 `::after`

Inserts content **after** the element.

They are treated as **child elements** created by CSS.

These pseudo-elements are powerful for:

- decorative icons
- background shapes
- tooltips
- labels
- custom UI elements
- styling without modifying HTML

⬛ When/Why used

✔ When you want to add decorative content without changing HTML

✔ To create shapes (triangles, overlays, ribbons)

✔ For adding icons before text

✔ Building custom UI components (badges, tags, counters)

✔ Cleaner HTML with styling handled by CSS only

⬛ Example

```
button::before {
  content: "🔥 ";
}

.card::after {
  content: "";
  position: absolute;
  width: 100%;
  height: 100%;
  background: rgba(0,0,0,0.1);
}
```

Here:

- The button displays a flame emoji before text
- The card gets a dark overlay using an `::after` pseudo-element

⬛ **Summary line**

`::before` and `::after` insert virtual elements for decoration without extra HTML.

✅ 2) Pseudo classes — `:hover`, `:active`, `:focus`

⬛ **Explanation (simple, conceptual)**

Pseudo-classes apply styles based on **element state**, not element type.

🔵 `:hover`

Applies when mouse hovers over element.

🔵 `:active`

Applies when element is **clicked or pressed**.

🔵 `:focus`

Applies when element gets **keyboard focus** (usually input fields).

⬛ **When/Why used**

✔ Improve button/link interactions

✔ Highlight focused input for accessibility

✔ Add animations on hover

✔ Improve UX on clickable elements

⬛ Example

```css
button:hover {
  background-color: orange;
}

button:active {
  transform: scale(0.95);
}

input:focus {
  border-color: blue;
  outline: none;
}
```

⬛ Summary line

Pseudo-classes style elements based on user interaction — hover, active click, or focus state.

✅ 3) CSS transitions vs animations

⬛ Explanation (simple, conceptual)

🔵 Transitions

Used for **smooth change between two states** (e.g., normal → hover).

Key rules:

- Needs a trigger (hover, active, click)
- Simple movement

🔵 Animations

Used for **complex, multi-step movements** without user interaction.

Key rules:

- Uses `@keyframes`
- Can loop, delay, repeat, bounce, fade, rotate, animate infinitely

⬛ When/Why used

Feature	Transitions	Animations
Trigger	User action	Auto-play
Complexity	Simple	Multi-step
Use case	Hover effects	Loaders, sliders
Keyframes	❌ No	✔️ Yes

⬛ Examples

Transition

```
button {
  transition: background 0.3s ease;
}
button:hover {
  background: green;
}
```

Animation

```
@keyframes bounce {
  0% { transform: translateY(0); }
  50% { transform: translateY(-20px); }
  100% { transform: translateY(0); }
}

.ball {
  animation: bounce 1s infinite;
}
```

■ **Summary line**

Transitions animate between two states; animations create complex, multi-step movement using keyframes.

✅ 4) What is rem vs em vs px?

■ **Explanation (simple, conceptual)**

🔵 px (Pixels)

- Absolute unit
- Does not scale with screen or parent font-size
- Precise but not responsive

🔵 em

- Relative to **parent element's** font-size
- If parent changes, child size changes too
- Can cause cascading/unexpected scaling

Example:

```css
.parent { font-size: 20px; }
.child { font-size: 2em; }   /* 40px */
```

🔵 rem (Root em)

- Relative to **root `<html>` font-size**
- Most predictable & recommended for responsiveness
- Doesn't depend on parent element

Example:

```css
html { font-size: 16px; }
button { font-size: 2rem; }  /* 32px */
```

⬛ When/Why used

Unit	When Used	Why
px	Pixel-perfect, small icons	Exact sizes
em	Scaling relative to parent	Flexible but tricky
rem	Responsive design	Most stable & widely used

⬛ Summary line

px is fixed, em is relative to parent, rem is relative to root — rem is best for responsive UI.

⚡ 5) How to center a div (multiple methods)

(Important interview question — asked often!)

⬛ Explanation (simple, conceptual)

Centering a div can mean:

- Horizontally
- Vertically
- Both

Each method works differently depending on layout.

✔️ Method 1: **Margin Auto (horizontal center)**

Works when element has width.

```css
.box {
  width: 200px;
  margin: 0 auto;
}
```

✓ Method 2: **Flexbox (horizontal + vertical center)**

Most common modern method.

```css
.container {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

✓ Method 3: **Grid (horizontal + vertical center)**

Simplest syntax.

```css
.container {
  display: grid;
  place-items: center;
}
```

✓ Method 4: **Absolute Position + Transform**

Useful for modals.

```css
.box {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

✔️ Method 5: **Text-align (center text inside div)**

Only for inline text.

```css
div {
  text-align: center;
}
```

✔️ Method 6: **Line-height (vertical center single line text)**

```css
div {
  height: 100px;
  line-height: 100px;
}
```

■ **When/Why used**

✔️ Flex/Grid for modern UI

✔️ Absolute positioning for modals/popup

✔️ Margin auto for simple layouts

✔️ transform method for pixel-perfect centering

■ **Summary line**

You can center a div using margin auto, flexbox, grid, transform, or text-align depending on layout needs.

📌 Practical/Scenario Based

Below are **three interview-ready UI implementations** with clean explanations, when/why notes, working example code you can copy, and a concise one-line summary you can speak in interviews. Each answer follows your exact structure.

■ **1) Create a responsive two-column layout**

■ **Explanation (simple, conceptual)**

A responsive two-column layout places content in two columns on larger screens and collapses to a single column on small screens. Use modern CSS (Flexbox or Grid) plus a media query so the layout adapts automatically without JS. Grid is ideal when you want strict column control; Flexbox is excellent for flow that can wrap.

■ **When/Why is it used**

Use this when you need a main content area and a sidebar (e.g., article + related widgets) or two panels that should sit side-by-side on desktop but stack on mobile for readability and usability.

■ **Example / Code snippet (HTML + CSS)**

This example uses CSS Grid (recommended for predictable two-column layouts). It is accessible (semantic tags), mobile-first, and includes gap + simple breakpoints.

```
<!-- responsive-two-column.html →
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width,initial-scale=1" />
<title>Two-column responsive layout</title>
<style>
```

```css
:root{
  --g: 1.25rem;
  --content-max: 1100px;
}

body{
  font-family: system-ui, -apple-system, "Segoe UI", Roboto, "Helvetica Neu
e", Arial;
  margin:0;
  padding: clamp(1rem, 2vw, 2rem);
  display:flex;
  justify-content:center;
  background:#f6f7fb;
}

.page{
  width:100%;
  max-width: var(--content-max);
  display:grid;
  gap: var(--g);
  grid-template-columns: 1fr; /* mobile-first: single column */
}

header.site-header{
  background:#ffffff;
  padding: 0.75rem 1rem;
  border-radius:8px;
  box-shadow: 0 1px 4px rgba(12,20,40,0.05);
}

/* Two-column at tablet+ */
@media (min-width: 768px){
  .page {
    grid-template-columns: 2fr 1fr; /* main : sidebar */
    align-items:start;
  }
}

main.article {
  background:#fff;
  padding:1rem;
  border-radius:8px;
  box-shadow: 0 1px 6px rgba(12,20,40,0.04);
}
aside.sidebar {
  background:#fff;
  padding:1rem;
  border-radius:8px;
  box-shadow: 0 1px 6px rgba(12,20,40,0.04);
}

.article h1{ margin-top:0; }
.card { padding:0.75rem; border-radius:6px; background:#fbfdff; }
</style>
</head>
<body>
<div class="page" role="document" aria-label="Sample two column layou
t">
  <header class="site-header" role="banner">
    <h2>My Site</h2>
  </header>

  <main class="article" role="main">
    <h1>Article Title</h1>
    <p>First paragraph — the main content appears here. On small screens thi
s stacks above the sidebar; on wider screens it sits left of the sidebar.</p>
    <section class="card">
      <h3>Section</h3>
      <p>Example content block inside main column.</p>
    </section>
  </main>
  <aside class="sidebar" role="complementary" aria-label="Related">
    <h3>Sidebar</h3>
    <p>Related links, widgets, or ads go here.</p>
  </aside>

</div>
</body>
</html>
```

■ Short summary line to speak in interview

“Use a mobile-first CSS Grid (or Flexbox) with a media-query breakpoint so two columns become one column on small screens — grid-template-columns: 1fr on mobile and 2fr 1fr on tablet/desktop.”

■ 2) Create a card UI (image + title + button)

■ Explanation (simple, conceptual)

A card UI groups an image, title, description and CTA into a compact, reusable component. Best practices include accessible markup (alt text, semantic tags), keyboard-focusable button, responsive image sizing, and a content hierarchy that is friendly to screen readers and search engines.

■ When/Why is it used

Use cards to present items such as products, articles, user profiles, or feature highlights on dashboards and list pages. Cards are modular and easy to repeat in lists or grids.

■ Example / Code snippet (HTML + CSS)

This card is responsive, accessible, and includes hover/focus affordances.

```
<!-- card-ui.html →
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width,initial-scale=1" />
```

```
<title>Card UI example</title>
<style>
 :root{
   --card-bg: #fff;
   --accent: #2563eb;
   --muted: #6b7280;
 }

 body{ font-family: system-ui, sans-serif; margin:2rem; background:#f3f4f6;
}

 .card {
   background: var(--card-bg);
   border-radius: 12px;
   overflow: hidden;
   box-shadow: 0 6px 18px rgba(11,20,50,0.06);
   max-width: 360px;
   transition: transform .18s ease, box-shadow .18s ease;
 }
 .card:focus-within,
 .card:hover {
   transform: translateY(-6px);
   box-shadow: 0 12px 28px rgba(11,20,50,0.12);
 }

 .card__media img{
   width:100%;
   height:200px;
   object-fit:cover;
   display:block;
 }

 .card__body {
   padding: 1rem;
 }
 .card__title {
   margin:0 0 .5rem 0;
   font-size:1.125rem;
   line-height:1.25;
 }
 .card__desc {
   margin:0 0 1rem 0;
   color: var(--muted);
   font-size:.95rem;
 }

 .card__actions {
   display:flex;
   gap:.5rem;
   padding: 0 1rem 1rem;
 }
 .btn {
   background: var(--accent);
   color: #fff;
   border: none;
   padding: .55rem .85rem;
   border-radius: 8px;
   cursor:pointer;
   font-weight:600;
   transition: transform .12s ease, box-shadow .12s ease;
 }
 .btn:focus { outline: 3px solid rgba(37,99,235,0.18); outline-offset: 2px; }
 .btn:hover { transform: translateY(-2px); box-shadow: 0 6px 12px rgba(37,99,
235,0.12); }

 /* responsive tweak */
 @media (max-width:420px){
   .card__media img { height:160px; }
   .card { max-width:100%; }
 }
</style>
</head>
```

```
<body>
  <article class="card" aria-labelledby="card1-title">
    <div class="card__media">
      <img src="https://picsum.photos/640/360?random=1" alt="Scenic mountain view" />
    </div>
    <div class="card__body">
      <h3 id="card1-title" class="card__title">Mountain adventure</h3>
      <p class="card__desc">Experience a 3-day guided hike with breath-taking views, local guides, and meals included.</p>
    </div>
    <div class="card__actions">
      <button class="btn" type="button">Book now</button>
      <button class="btn" type="button" style="background:#e5e7eb;color:#111;font-weight:600">Details</button>
    </div>
  </article>
</body>
</html>
```

Notes: replace the demo image URL with your asset and supply descriptive `alt` text for accessibility.

■ Short summary line to speak in interview

"Build a semantic `<article>` card with an image, accessible alt text, heading and CTA; use object-fit for responsive images and subtle hover/focus effects for affordance."

■ 3) Style a button hover animation

■ Explanation (simple, conceptual)

A polished hover animation improves perceived responsiveness and guides users. Keep animations short (100–300ms), hardware-accelerated (use transform/opacity), and ensure they are accessible (keyboard focus and reduced-motion respect).

■ When/Why is it used

Use hover animations to indicate clickable controls, improve micro-interactions, and draw attention to CTAs while ensuring motion is not distracting or harmful to motion-sensitive users.

■ Example / Code snippet (HTML + CSS)

This example includes: subtle scale, shadow on hover, an expanding pseudo-element underline, focus styles, and `prefers-reduced-motion` support.

```
<!-- button-animation.html →
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width,initial-scale=1" />
<title>Button Hover Animation</title>
<style>
 :root{ --accent:#0ea5e9; --accent-dark:#0284c7; --text:#fff; }

 body{ display:flex; align-items:center; justify-content:center; min-height:60vh; margin:0; font-family:system-ui, sans-serif; background:#0f172a; }

 .btn {
   --pad-x:1.05rem;
   --pad-y:.6rem;
   position:relative;
   display:inline-block;
   background:linear-gradient(180deg,var(--accent),var(--accent-dark));
   color:var(--text);
   padding:var(--pad-y) var(--pad-x);
   border-radius:10px;
   border:0;
   cursor:pointer;
   font-weight:700;
   letter-spacing:.2px;
   transform: translateZ(0); /* enable GPU composite */
   transition: transform .18s cubic-bezier(.2,.9,.3,1), box-shadow .18s ease;
```

```css
  box-shadow: 0 6px 18px rgba(2,6,23,0.55);
  overflow:hidden;
}

/* scale + lift on hover/focus */
.btn:hover,
.btn:focus {
  transform: translateY(-4px) scale(1.02);
  box-shadow: 0 14px 30px rgba(2,6,23,0.5);
}

/* animated underline using pseudo-element */
.btn::after {
  content: "";
  position: absolute;
  left: 50%;
  bottom: 8%;
  width: 0%;
  height: 3px;
  background: rgba(255,255,255,0.9);
  border-radius: 3px;
  transform: translateX(-50%);
  transition: width .28s cubic-bezier(.2,.9,.3,1);
}

.btn:hover::after,
.btn:focus::after {
  width: 70%;
}

/* focus visible for keyboard users */
.btn:focus-visible {
  outline: 3px solid rgba(14,165,233,0.18);
  outline-offset: 4px;
}
```

```
  /* respect user reduced motion preference */
  @media (prefers-reduced-motion: reduce) {
   .btn,
   .btn::after { transition: none !important; transform: none !important; }
  }
</style>
</head>
<body>
  <button class="btn" type="button">Primary action</button>
</body>
</html>
```

Accessibility tips:

- Ensure keyboard users see focus feedback (`:focus-visible` used above).
- Use `prefers-reduced-motion` to disable animations for users who prefer less motion.

◼️ Short summary line to speak in interview

"Animate buttons using transform/opacity for GPU-acceleration, add a pseudo-element for decorative effects, support keyboard focus, and respect `prefers-reduced-motion` for accessibility."

🔥 React.js Interview Questions

📌 Basics

✅ 1) What is React? Why use it?

◼️ Explanation (simple, conceptual)

React is a **JavaScript library** for building **fast, interactive user interfaces**, created by Facebook.

It uses a **component-based architecture**, a **virtual DOM**, and **one-way data flow**, which makes UI predictable and easier to maintain.

React builds UI by breaking the page into **reusable components** like buttons, forms, modals, cards, etc.

⬛ When/Why is it used?

✔️ To build **SPA (Single Page Applications)**

✔️ For highly interactive UI (dashboards, forms, live pages)

✔️ When performance matters (Virtual DOM makes rendering fast)

✔️ For modular, reusable UI components

✔️ Easy integration with backend APIs and modern tooling

✔️ Large ecosystem: hooks, Redux, routing, testing libraries

⬛ Example / Code snippet

```
function Hello() {
  return <h1>Hello React!</h1>;
}
```

⬛ Summary line

React is a fast, component-based UI library used to build interactive and scalable web applications.

✅ 2) What is SPA (Single Page Application)?

⬛ Explanation (simple, conceptual)

A SPA loads **one HTML page** and dynamically updates content using **JavaScript**, instead of loading a new page from the server each time.

Instead of:

```
Page reload → Server → New HTML
```

SPA does:

```
No reload → JS updates UI → API data fetch only
```

◼ **When/Why used?**

✔ Faster navigation (no full-page reload)

✔ Better user experience (like mobile apps)

✔ Efficient API communication

✔ Smooth state-based UI transitions

✔ Used by modern apps like Gmail, Facebook, Netflix

◼ **Example**

React Router makes SPA navigation possible:

```
<Route path="/home" element={<Home />} />
```

No page reload happens.

◼ **Summary line**

SPA loads one HTML page and updates UI dynamically without refreshing, giving fast app-like experience.

✅ 3) What is JSX?

◼ **Explanation (simple, conceptual)**

JSX (**JavaScript XML**) is a syntax that lets you write **HTML-like code inside JavaScript**.

```
const element = <h1>Hello World</h1>;
```

Browsers don’t understand JSX.

Babel compiles JSX → JavaScript like:

```
React.createElement("h1", null, "Hello World")
```

⬛ **When/Why used?**

✔️ Makes UI code readable and declarative

✔️ Allows mixing HTML with JavaScript logic

✔️ Avoids manual DOM manipulation

✔️ Helps create component-based structure

⬛ **Example**

```
const Welcome = () ⇒ <h2>Welcome User</h2>;
```

⬛ **Summary line**

JSX is a JavaScript syntax extension that lets you write HTML-like UI templates inside JS.

✅ 4) What are components? Types? (Functional vs Class)

⬛ **Explanation (simple, conceptual)**

Components are **independent, reusable UI pieces** in React—like Lego blocks.

Each component returns some UI and can manage its own data (state).

React components are of two types:

🟦 1. Functional Components (Modern & Preferred)

- Simple JavaScript functions
- Use **hooks** (useState, useEffect, etc.)
- Lightweight and fast
- Recommended for all new development

```
function Welcome() {
  return <h1>Hello!</h1>;
}
```

🟩 2. Class Components (Older)

- ES6 classes
- Have lifecycle methods (componentDidMount etc.)
- Mostly replaced by functional components

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello!</h1>;
  }
}
```

⬛ When/Why used?

✔ For reusable UI elements

✔ To manage component-specific state

✔ To separate UI into smaller, testable pieces

✔ Functional components are used for modern React apps

⬛ Summary line

Components are reusable UI elements; React has functional components (modern, hook-based) and class components (older).

✅ 5) Props vs State

⬛ Explanation (simple, conceptual)

🟦 Props (External Data)

- Passed *from parent to child*
- **Read-only**
- Used to configure a component
- Component cannot modify its own props

🟩 State (Internal Data)

- Managed *within the component itself*
- **Mutable** (changes with setState/useState)
- Used for dynamic UI behavior

⬛ When/Why used?

Concept	When used	Why
Props	Passing data down	Makes components reusable & configurable
State	Interactive features (input, toggles, counters, API data)	Allows components to update UI dynamically

---

⬛ Examples

Props Example

```
function Greeting(props) {
  return <h1>Hello {props.name}</h1>;
}
```

State Example

```
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() ⇒ setCount(count + 1)}>Count: {count}</button>;
}
```

⬛ Summary line

Props are read-only inputs passed from parent; State is mutable internal data used to manage dynamic UI.

✅ 6) Unidirectional Data Flow in React

⬛ Explanation (simple, conceptual)

React uses **one-way data flow**, meaning **data always flows from parent → child**.

Parents pass data through **props**, and children cannot directly modify parent data.

This makes React UI predictable because:

- data origin is clear
- state lives at a single source of truth
- debugging becomes easier

React *never* allows child-to-parent implicit data flow.

⬛ When/Why is it used?

✔ To keep UI predictable and easy to debug

✔ To ensure components are isolated

✔ To avoid unwanted side effects

✔ To maintain clarity about where data comes from

✔ Ideal for large applications where state complexity grows

⬛ Example

```
function Parent() {
  const [name, setName] = useState("Mohini");

  return <Child username={name} />;
}

function Child({ username }) {
  return <h2>Hello {username}</h2>;
}
```

Data moves only **Parent → Child**.

■ Short summary line

React uses one-way data flow, meaning data always moves from parent to child for predictable UI updates.

✅ 7) What are keys in React lists?

■ Explanation (simple, conceptual)

Keys are **unique identifiers** used when rendering lists.

React uses keys to detect:

- which items changed
- which items were added
- which items were removed

Keys help React efficiently update only the changed elements instead of re-rendering the whole list.

■ When/Why are keys used?

✔ To improve performance in lists

✔ To ensure stable component identity

✔ To avoid UI bugs when items reorder

✔ To help React's diffing algorithm

■ Example

```
const items = ["A", "B", "C"];

items.map((item, index) => (
  <li key={item}>{item}</li>
));
```

Avoid using **index** as a key unless:

- list never changes
- no reordering
- static content

◼ **Short summary line**

Keys uniquely identify list items so React can efficiently update and reorder them.

✅ 8) Virtual DOM vs Real DOM

◼ **Explanation (simple, conceptual)**

🔵 Real DOM

- Direct representation of UI in the browser
- Expensive and slow when updating large sections
- Small change → browser recalculates layout → repaints everything

🔵 Virtual DOM

- Lightweight JavaScript copy of the real DOM
- React compares old vs new virtual DOM using **diffing**
- Only updates the exact parts that changed

This is why React apps feel fast.

◼ **When/Why used?**

✔ Faster UI updates

✔ Efficient rendering

✔ Avoids full page redraw

✔ Better performance on dynamic UIs like dashboards

✔ Predictable updates through reconciliation

⬛ Example (conceptual diffing)

```
<div id="name">John</div>     // old virtual DOM
<div id="name">David</div>    // new virtual DOM
```

React detects only the changed text:

```
John → David
```

Only updates that node, not whole page.

⬛ Short summary line

Virtual DOM is a lightweight copy used by React to update only what changed, making UI fast and efficient.

✅ 9) Why do we need Hooks?

⬛ Explanation (simple, conceptual)

Hooks were introduced in React 16.8 to allow **state and lifecycle features inside functional components**.

Before hooks, only **class components** had state and lifecycle methods.

Hooks brought a simpler, cleaner way to manage:

- state (`useState`)
- side effects (`useEffect`)
- references (`useRef`)
- memoization (`useMemo`, `useCallback`)
- context (`useContext`)
- complex state logic (`useReducer`)

⬛ When/Why used?

✔️ Replace class components with cleaner functional components

✔️ Share logic across components through custom hooks

✔️ Reduce boilerplate (`constructor`, `this`, binding)

✔️ Improve readability and maintainability

✔️ Handle side effects easily

✔️ Local component state management

Hooks make modern React **simpler, faster, and more developer-friendly**.

⬛ Example

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("Count changed");
  }, [count]);

  return <button onClick={() => setCount(count + 1)}>Count {count}</button>;
}
```

⬛ Short summary line

Hooks enable state, side effects, and advanced logic inside functional components—making React simpler and more powerful.

📌 Hooks (Very Important For Interview)

✅ 1) `useState()` — how it works?

⬛ Explanation (simple, conceptual)

`useState()` is a React Hook that lets a **functional component store internal state**.

It returns **two values**:

1. current state
2. a function to update that state

React re-renders the component whenever the state changes — but **state updates are asynchronous and batched** for performance.

■ When/Why is it used?

✔ When a component needs to store dynamic data (inputs, toggles, counters)

✔ For updating UI on user actions

✔ To manage simple component-level state

■ Example

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() ⇒ setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

■ Summary line

`useState` stores component state and re-renders UI whenever the state updates.

✅ 2) `useEffect()` use cases — cleanup example

■ Explanation (simple, conceptual)

`useEffect()` lets you perform **side effects** in a functional component (API calls, subscriptions, timers, event listeners).

It runs **after rendering**.

It can also return a **cleanup function** that runs before component unmounts or before next effect run.

■ **When/Why is it used?**

✔ API calls (fetch data)

✔ Listening to browser events

✔ Setup timers/intervals

✔ Cleanup logic on unmount

✔ Syncing component with external systems

Cleanup Example: Timer unsubscribe

```
useEffect(() ⇒ {
  const timer = setInterval(() ⇒ {
    console.log("Timer running...");
  }, 1000);

  // cleanup function
  return () ⇒ {
    clearInterval(timer);
    console.log("Timer stopped");
  };
}, []);
```

■ **Summary line**

`useEffect` handles side effects like API calls and subscriptions; cleanup stops memory leaks.

✅ 3) `useRef()` — when do we use it?

■ **Explanation (simple, conceptual)**

`useRef()` gives you a **mutable object** that persists across renders **without causing re-renders**.

It can be used for:

- Accessing DOM elements
- Storing previous values
- Storing timers/intervals
- Keeping any mutable variable between renders

■ When/Why is it used?

✔ To access DOM nodes (`inputRef.current.focus()`)

✔ To store values without triggering re-render

✔ To track previous state

✔ To keep timeout/interval ids

■ Example

```
function InputFocus() {
  const inputRef = useRef(null);

  return (
    <>
      <input ref={inputRef} />
      <button onClick={() ⇒ inputRef.current.focus()}>
        Focus Input
      </button>
    </>
  );
}
```

■ Summary line

`useRef` **stores mutable values and DOM references without causing re-renders.**

✅ 4) `useMemo` vs `useCallback`

⬛ Explanation (simple, conceptual)

🔵 useMemo

Memoizes the **result of a function**.

Used for expensive calculations.

🔵 useCallback

Memoizes the **function itself**.

Used to prevent unnecessary re-renders when passing callbacks to child components.

⬛ When/Why used?

Hook	When used	Why
useMemo	Expensive calculations, derived values	Avoid recomputation
useCallback	Passing functions to children	Prevent unnecessary re-renders

⬛ Example

```
const expensiveValue = useMemo(() ⇒ heavyCalc(num), [num]);

const handleClick = useCallback(() ⇒ {
  console.log("clicked");
}, []);
```

⬛ Summary line

`useMemo` memoizes values; `useCallback` memoizes functions to avoid re-renders.

✅ 5) `useContext()` — global state sharing

⬛ Explanation (simple, conceptual)

`useContext()` allows you to share **global state** across components **without prop drilling** (passing props through multiple nested components).

Best for:

- themes
- user info
- language settings
- authentication status

■ When/Why used?

✔ To avoid passing props across many levels

✔ When different parts of app need same data (global state)

✔ For lightweight global state management (instead of Redux)

■ Example

```
const ThemeContext = createContext("light");

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Child />
    </ThemeContext.Provider>
  );
}

function Child() {
  const theme = useContext(ThemeContext);
  return <h1>Theme: {theme}</h1>;
}
```

■ Summary line

`useContext` **shares global data without prop drilling, ideal for app-wide settings.**

✅ 6) `useReducer()` — when to prefer it?

■ Explanation (simple, conceptual)

`useReducer()` is an alternative to `useState` for **complex state logic**.

It uses:

- a **state object**
- a **reducer function**
- a **dispatch(action)** function

Similar to Redux pattern but inside a component.

■ When/Why used?

✔ When state updates are complex

✔ When multiple actions change the same state

✔ When next state depends on previous state

✔ When you want clear "action → change" mapping

✔ Good for forms, multi-step flows, toggles, counters

■ Example

```
function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (
    <>
      <button onClick={() ⇒ dispatch({ type: "decrement" })}>-</button>
      <span>{state.count}</span>
      <button onClick={() ⇒ dispatch({ type: "increment" })}>+</button>
    </>
  );
}
```

■ Summary line

`useReducer` is preferred for complex state transitions or when multiple actions modify the same state.

📌 Advanced Concepts

✅ 1) What is lifting state up?

■ Explanation (simple, conceptual)

"Lifting state up" means **moving a piece of state to the closest common parent** so that multiple child components can share or update that same state.

Instead of:

- Child A having its own state
- Child B having its own state

We keep the shared state in **Parent**, and Children use props to interact with it.

■ When/Why used?

✔ When two or more components need the same data

✔ To sync UI between siblings (ex: input filter + list)

✔ To avoid duplicate or conflicting state in children

✔ Ensures a single source of truth

■ **Example**

```
function Parent() {
  const [value, setValue] = useState("");

  return (
    <>
      <Input value={value} onChange={setValue} />
      <Display value={value} />
    </>
  );
}

function Input({ value, onChange }) {
  return <input value={value} onChange={(e) ⇒ onChange(e.target.value)} />;
}

function Display({ value }) {
  return <h2>You typed: {value}</h2>;
}
```

■ **Summary line**

Lifting state up centralizes shared state in the parent so child components stay synchronized.

✅ 2) Controlled vs Uncontrolled components

■ **Explanation (simple, conceptual)**

Controlled Component

- React controls the form data
- Value stored in **state**

- UI always reflects state

```
<input value={name} onChange={(e) => setName(e.target.value)} />
```

Uncontrolled Component

- DOM controls the form data
- Uses `ref` to get value
- React does NOT track input value

```
<input ref={inputRef} />
```

■ When/Why used?

Type	When used	Why
Controlled	Form validation, dynamic inputs	Reliable, predictable, easy to validate
Uncontrolled	Simple forms, performance-critical	Less re-renders, simpler

---

■ Summary line

Controlled components use React state; Uncontrolled components rely on DOM and refs.

✅ 3) What are Higher Order Components (HOC)?

■ Explanation (simple, conceptual)

A Higher Order Component is a **function that takes a component and returns a new component** with enhanced behavior.

Pattern:

```
const EnhancedComponent = withFeature(OriginalComponent);
```

HOCs add reusable logic such as:

- authentication
- logging
- theming
- permissions
- data fetching

⬛ When/Why used?

✔️ To reuse common logic across many components

✔️ Before hooks, HOCs were used instead of custom hooks

✔️ Used in libraries like Redux (`connect`)

⬛ Example

```
function withLogger(WrappedComponent) {
  return function(props) {
    console.log("Component rendered");
    return <WrappedComponent {...props} />;
  };
}

const MyComponentWithLog = withLogger(MyComponent);
```

⬛ Summary line

HOCs wrap a component and return an enhanced component to reuse logic.

✅ 4) React Router basics — Routes, Navigate, Params

⬛ Explanation (simple, conceptual)

React Router enables **client-side routing** in SPAs.

No page refresh — only components change.

Key features:

🔵 Routes

Maps a URL path to a component.

```
<Routes>
  <Route path="/home" element={<Home />} />
</Routes>
```

🔵 Navigate

Redirect programmatically.

```
return <Navigate to="/login" />;
```

🔵 Params

Dynamic values in URL.

```
<Route path="/user/:id" element={<User />} />
```

Access params:

```
const { id } = useParams();
```

⬛ When/Why used?

✔ To create SPA navigation

✔ To handle dynamic pages (`/product/23`)

✔ For redirection after login/logout

✔ Cleaner URL-based navigation

⬛ **Summary line**

React Router provides client-side routing using Routes, Navigate for redirects, and Params for dynamic URLs.

✅ 5) Difference between SPA routing vs Server routing

⬛ **Explanation (simple, conceptual)**

🟦 Server Routing

- Browser sends request to server
- Server returns a full new HTML page
- Slow, heavy

Example:

```
/products → server responds with HTML page
```

🟩 SPA (Client-side) Routing

- Browser loads **single index.html**
- React updates UI based on URL
- Fast, no page reload
- Uses JavaScript + Virtual DOM

Example:

```
/products → React loads <Products/> component
```

⬛ **When/Why used?**

✔️ SPA routing gives fast, app-like experience

✔️ Server routing is used for SEO-heavy, static websites

⬛ **Summary line**

Server routing reloads whole pages; SPA routing loads components without page refresh.

✅ 6) What are portals in React?

⬛ **Explanation (simple, conceptual)**

Portals allow you to render a component **outside its parent DOM hierarchy** while still maintaining React component hierarchy.

Used to place UI elements at a different spot in the DOM — usually at the top level.

⬛ **When/Why used?**

✔ For modals

✔ For dropdowns

✔ For tooltips

✔ For popovers

✔ When an element must escape parent overflow/position

⬛ **Example**

```
function Modal({ children }) {
  return ReactDOM.createPortal(
    <div className="modal">{children}</div>,
    document.getElementById("modal-root")
  );
}
```

HTML:

```
<div id="modal-root"></div>
```

⬛ **Summary line**

Portals let you render components outside parent DOM hierarchy—perfect for modals and overlays.

✅ 7) Error Boundaries — usage

◼️ Explanation (simple, conceptual)

Error Boundaries are **React components that catch JavaScript errors** in their child component tree, preventing the whole app from crashing.

They catch:

- Rendering errors
- Lifecycle errors
- Errors inside constructors

They **DO NOT** catch:

- Event handler errors
- Async errors (setTimeout, Promises)
- Server errors

Error boundaries work like a global error UI wrapper.

◼️ When/Why used?

✔️ To prevent full UI crash

✔️ To show fallback UI ("Something went wrong")

✔️ To log errors to monitoring tools (Sentry, LogRocket)

✔️ For isolating failing components

✔️ Essential in production apps

◼️ Example

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.log("Error logged:", error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong.</h2>;
    }
    return this.props.children;
  }
}
```

Usage:

```
<ErrorBoundary>
  <MyComponent />
</ErrorBoundary>
```

■ Summary line

Error Boundaries catch render-time errors in components and display fallback UI without crashing the entire app.

✅ 8) Lazy loading & code splitting in React

■ Explanation (simple, conceptual)

Lazy loading and code splitting allow React to load **only the necessary code**, improving performance.

🔵 Code Splitting

Breaking app bundle into smaller chunks.

🔵 Lazy Loading

Loading components **only when needed**, instead of upfront.

React provides:

```
const Component = React.lazy(() ⇒ import('./Component'));
```

Lazy loading reduces:

- initial loading time
- bundle size
- bandwidth

⬛ When/Why used?

✔ Large applications with multiple pages

✔ When certain components are rarely used

✔ For performance optimization

✔ Faster initial load times

✔ Better user experience

⬛ Example

```
const Dashboard = React.lazy(() ⇒ import('./Dashboard'));

function App() {
  return (
    <Suspense fallback={<h3>Loading...</h3>}>
      <Dashboard />
    </Suspense>
  );
}
```

⬛ Summary line

Lazy loading loads components only when needed; code splitting reduces bundle size for faster performance.

✅ 9) What is Suspense?

■ Explanation (simple, conceptual)

`Suspense` is a React component used to **wrap lazy-loaded components** and show a **fallback UI** while waiting.

It suspends rendering until the lazy component is fully loaded.

Suspense also works with:

- Data fetching libraries like React Query
- Concurrent features
- Streaming rendering

■ When/Why used?

✔ When using `React.lazy()`

✔ For showing a loader/spinner

✔ To avoid blank screen while component downloads

✔ For smooth user experience

■ Example

```
<Suspense fallback={<Spinner />}>
  <Chart />
</Suspense>
```

■ Summary line

Suspense displays fallback UI while waiting for lazy-loaded components or async data.

✅ 10) What is reconciliation process?

⬛ Explanation (simple, conceptual)

Reconciliation is the **internal algorithm React uses to update the DOM** efficiently.

React compares:

- **Old Virtual DOM**
- **New Virtual DOM**

Then updates **only what changed** (diffing process).

Key rules in reconciliation:

1. Re-renders only affected components
2. Uses **keys** to detect moved/modified list items
3. Avoids recreating unchanged DOM nodes
4. Uses heuristics to compute minimal updates

⬛ When/Why used?

✔ To improve performance

✔ To avoid unnecessary DOM operations

✔ To provide smooth UI updates

✔ Used every time state or props change

⬛ Example (conceptual)

Old:

```
<li>A</li>
<li>B</li>
<li>C</li>
```

New:

```
<li>A</li>
<li>X</li>
```

```
<li>C</li>
```

React updates *only B → X* instead of re-rendering whole list.

⬛ Summary line

Reconciliation is React's diffing algorithm that updates only the changed elements for fast rendering.

📌 API + Integration

1) How `fetch` / `axios` works in React

⬛ Explanation (simple, conceptual)

Both `fetch` (built-in browser API) and `axios` (popular HTTP library) perform HTTP requests from the browser. In React you typically call them inside lifecycle hooks (e.g., `useEffect`) or event handlers to fetch/send data. `fetch` returns a `Promise` that resolves to a `Response` which you must parse (`res.json()`); `axios` returns a Promise that resolves to a response object with `data` already parsed. Both support cancellation (AbortController for fetch; AbortController or cancel token for axios).

⬛ When/Why is it used

✔️ To retrieve remote data (GET) or send changes (POST/PUT/DELETE) from UI.

✔️ Use `fetch` for a zero-dep, modern browser API; use `axios` when you want convenience features (automatic JSON parsing, interceptors, request/response transforms, timeout, easier error shape).

⬛ Example / Code snippet

`fetch` inside `useEffect` with `AbortController`:

```
import { useEffect, useState } from "react";

function UsersList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    const controller = new AbortController();
    const signal = controller.signal;

    async function load() {
      try {
        const res = await fetch("/api/users", { signal });
        if (!res.ok) throw new Error(`HTTP ${res.status}`);
        const data = await res.json();
        setUsers(data);
      } catch (err) {
        if (err.name === "AbortError") return; // request cancelled
        console.error("Fetch error", err);
      }
    }
    load();
    return () => controller.abort();
  }, []);

  return <ul>{users.map(u => <li key={u.id}>{u.name}</li>)}</ul>;
}
```

`axios` with interceptors and cancel via `AbortController` (axios v0.22+ supports `signal`):

```
import axios from "axios";
import { useEffect, useState } from "react";

axios.interceptors.response.use(
  r => r,
  err => {
    // global error handling (optional)
    return Promise.reject(err);
  }
);

function Users() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    const controller = new AbortController();
    axios.get("/api/users", { signal: controller.signal })
      .then(r ⇒ setUsers(r.data))
      .catch(err ⇒ {
        if (axios.isCancel?.(err)) return;
        console.error(err);
      });
    return () ⇒ controller.abort();
  }, []);

  return <div>{users.length} users</div>;
}
```

■ **Short summary line**

Use `fetch` for a no-dependency approach and `axios` when you want nicer defaults (auto JSON, interceptors); call them in hooks and cancel with AbortController to avoid memory leaks.

---

2) How to handle GET / POST API call

■ **Explanation (simple, conceptual)**

A GET request retrieves data and should be idempotent; a POST request sends data to create resources. In React, GET is often done in `useEffect` on mount; POST is triggered by a form submission handler. Handle loading, success, validation errors and server errors; always parse JSON and handle HTTP statuses.

■ **When/Why is it used**

✔ GET to populate UI on load (lists, details).

✔ POST to submit forms, create resources, or perform actions that change server state.

■ **Example / Code snippet**

GET (fetch) pattern:

```
useEffect(() ⇒ {
  let mounted = true;
  fetch("/api/items")
    .then(r => { if (!r.ok) throw r; return r.json(); })
    .then(data => { if (mounted) setItems(data); })
    .catch(err => console.error(err));
  return () => { mounted = false; };
}, []);
```

POST (axios) with optimistic UI and error handling:

```
import axios from "axios";
function CreateItem() {
  const [name, setName] = useState("");
  const [saving, setSaving] = useState(false);

  async function onSubmit(e) {
    e.preventDefault();
    setSaving(true);
    try {
      const response = await axios.post("/api/items", { name });
      // success: update UI or redirect
      console.log("Created", response.data);
    } catch (err) {
      // display server validation messages
      console.error("Create failed", err.response?.data || err.message);
    } finally {
      setSaving(false);
    }
  }

  return (
    <form onSubmit={onSubmit}>
      <input value={name} onChange={e => setName(e.target.value)} />
      <button type="submit" disabled={saving}>{saving ? "Saving..." : "Create"}
</button>
    </form>
  );
}
```

■ Short summary line

Perform GET in lifecycle hooks and POST in event handlers; always handle loading, success, and error states and parse/validate server responses.

---

3) How to show loading + error UI

■ Explanation (simple, conceptual)

UX requires clear feedback for network operations: show a loading indicator while a request is pending and a friendly error message if it fails. Maintain dedicated state for `loading`, `error`, and `data`. Consider boolean flags or a status enum (`idle|loading|success|error`) for clarity.

■ When/Why is it used

✔ To give users immediate feedback and avoid ambiguous UI.

✔ To prevent multiple submissions and indicate retry options.

✔ Improves perceived performance and accessibility (announce loaders).

■ Example / Code snippet

Minimal pattern with `status` enum and accessible spinner:

```
function DataFetcher() {
  const [status, setStatus] = useState("idle"); // idle | loading | success | error
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    let mounted = true;
    setStatus("loading");
    fetch("/api/data")
      .then(res => { if (!res.ok) throw new Error(res.statusText); return res.json();
})
      .then(d => { if (mounted) { setData(d); setStatus("success"); } })
      .catch(err ⇒ { if (mounted) { setError(err.message); setStatus("error"); } });
    return () ⇒ { mounted = false; };
  }, []);

  if (status === "loading") return <div role="status" aria-live="polite">Loadin
g...</div>;
  if (status === "error") return <div role="alert">Failed to load: {error}</div>;
  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}
```

■ Short summary line

Keep explicit `loading` / `error` state and render accessible loader and error UI so users always know what's happening.

---

4) What is CORS issue & solution

■ Explanation (simple, conceptual)

CORS (Cross-Origin Resource Sharing) is a browser security mechanism that restricts web pages from making requests to a different origin (protocol + host + port) unless the target server explicitly allows it via response headers. A typical CORS error happens when the browser blocks a cross-origin request because the server didn't return `Access-Control-Allow-Origin`.

There are two main request types:

- **Simple requests** — browser automatically checks response headers.
- **Preflighted requests** — browser sends an `OPTIONS` preflight when request has nonstandard headers or HTTP methods; server must respond with appropriate `Access-Control-*` headers.

■ When/Why is it used

✔ Prevents malicious websites from reading sensitive data on other origins.

✔ When your React dev server (e.g., `localhost:3000`) calls an API on another origin (e.g., `api.example.com`), you must enable CORS on the API.

■ Example / Solutions

Server-side fix (recommended): configure API to send CORS headers.

- **Node/Express**

```
// server.js
const express = require("express");
const cors = require("cors");
app.use(cors({ origin: "http://localhost:3000", methods: ["GET","POST"] }));
```

- **Spring Boot**

```
// @CrossOrigin or global WebMvcConfigurer
@CrossOrigin(origins = "http://localhost:3000")
@RestController ...
```

For preflight: ensure server responds to `OPTIONS` with `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers`.

Dev-time workaround (not for production):

- Use a proxy in dev (create-react-app `proxy` in package.json) so browser sees same origin.

```
// package.json (CRA)
"proxy": "http://localhost:8080"
```

- Or run a reverse proxy (nginx) or browser extension (dev only).

■ Short summary line

CORS is a browser-enforced origin policy; fix it by enabling appropriate `Access-Control-Allow-*` headers on the server or use a dev proxy — server-side configuration is the correct solution.

---

5) How to debounce API search calls

■ Explanation (simple, conceptual)

Debouncing delays invoking a function until a certain time has passed since the last call. For search APIs, debounce prevents firing a request on every keystroke and reduces request rate by only calling the API once the user pauses typing.

■ **When/Why is it used**

✔ Improve performance and reduce server load for live-search/autocomplete.

✔ Avoid stale/flooded network traffic.

✔ Improve UX by showing relevant results after user stops typing.

■ **Example / Code snippet**

Option A — custom hook using `setTimeout` + `useEffect`:

```
import { useState, useEffect } from "react";

// useDebouncedValue hook
function useDebounced(value, delay = 300) {
  const [debounced, setDebounced] = useState(value);
  useEffect(() => {
    const id = setTimeout(() => setDebounced(value), delay);
    return () => clearTimeout(id);
  }, [value, delay]);
  return debounced;
}

// Usage in component:
function Search() {
  const [q, setQ] = useState("");
  const debouncedQ = useDebounced(q, 400);
  const [results, setResults] = useState([]);

  useEffect(() => {
    if (!debouncedQ) { setResults([]); return; }
    let cancelled = false;
    (async () => {
      try {
        const res = await fetch(`/api/search?q=${encodeURIComponent(deboun
cedQ)}`);
        if (!res.ok) throw new Error("Search failed");
        const data = await res.json();
        if (!cancelled) setResults(data);
      } catch (err) {
        if (!cancelled) console.error(err);
      }
    })();
    return () => { cancelled = true; };
  }, [debouncedQ]);

  return (
    <>
      <input value={q} onChange={e => setQ(e.target.value)} placeholder="Sear
ch..." />
      <ul>{results.map(r => <li key={r.id}>{r.name}</li>)}</ul>
    </>
  );
}
```

Option B — `lodash.debounce` with `useCallback` (keeps function identity stable):

```
import { useMemo } from "react";
import debounce from "lodash.debounce";

function SearchWithLodash() {
  const [results, setResults] = useState([]);

  // create debounced version once
  const debouncedSearch = useMemo(() =>
    debounce(async (q) => {
      const r = await fetch(`/api/search?q=${encodeURIComponent(q)}`);
      setResults(await r.json());
    }, 350)
  , []);
  // cleanup on unmount
  useEffect(() => () => debouncedSearch.cancel(), [debouncedSearch]);

  return <input onChange={e => debouncedSearch(e.target.value)} />;
}
```

Notes:

- Cancel inflight requests when query changes (use AbortController).
- Prefer the custom `useDebounced` hook for simplicity and testability, or `lodash.debounce` when you want battle-tested debounce behavior.

⬛ Short summary line

Debounce search input (via a `useDebounced` hook or `lodash.debounce`) so API calls fire only after the user pauses typing — this reduces network load and improves UX.

📌 Performance Optimization

✅ 1) React.memo — when to use it?

⬛ Explanation (simple, conceptual)

`React.memo` is a **higher-order component** that prevents a component from re-rendering **unless its props change**.

It performs a **shallow comparison** of props.

If props are the same → React skips re-render → improves performance.

It is similar to **PureComponent** but for functional components.

⬛ When/Why is it used?

Use `React.memo` when:

✔ Child component receives props

✔ Props rarely change

✔ Child component is heavy (complex UI / expensive operations)

✔️ Parent re-renders frequently

✔️ Avoid unnecessary re-renders in list items

Avoid using it:

✖️ If the component is simple (overhead > benefit)

✖️ If props always change anyway

⬛ Example

```
const UserCard = React.memo(function UserCard({ user }) {
  console.log("Rendered");
  return <div>{user.name}</div>;
});

function App() {
  const [count, setCount] = useState(0);
  const user = { name: "Mohini" };

  return (
    <>
      <button onClick={() ⇒ setCount(count + 1)}>+</button>
      <UserCard user={user} />
    </>
  );
}
```

Without memo → UserCard re-renders every time count changes

With memo → UserCard renders **only once** because props didn't change.

⬛ Summary line

React.memo skips re-renders when props don't change, improving performance for heavy components.

✅ 2) Rerender issues — how to avoid?

■ Explanation (simple, conceptual)

React re-renders a component when:

- its **state** changes
- its **props** change
- its **parent** re-renders

Unnecessary re-renders slow down performance.

■ When/Why do we avoid them?

✔ To optimize performance

✔ To prevent slow UI (especially large lists, dashboards)

✔ To avoid wasted rendering work

■ How to avoid re-render issues?

1. Use `React.memo`

Prevents re-render if props are unchanged.

2. Use `useCallback`

Prevents functions from being recreated on every render.

```
const handleClick = useCallback(() ⇒ {}, []);
```

3. Use `useMemo`

Prevents expensive calculations from running repeatedly.

```
const sortedList = useMemo(() ⇒ sort(list), [list]);
```

4. Avoid recreating objects/arrays inline

Bad:

```
<Child data={{ name: "x" }} />
```

Good:

```
const data = useMemo(() ⇒ ({ name: "x" }), []);
<Child data={data} />;
```

5. Keep component small

Split UI into smaller components.

⬛ Summary line

Avoid re-renders using React.memo, useCallback, useMemo, stable props, and component splitting.

✅ 3) Why keys are important in list rendering?

⬛ Explanation (simple, conceptual)

Keys help React identify **which list items changed, moved, or were removed**.

React uses keys in the **reconciliation (diffing) algorithm**.

Without keys → React may re-render entire list incorrectly.

Keys must be **unique and stable**.

⬛ When/Why is it used?

✔ For dynamic lists

✔ To maintain component identity

✔ To avoid losing input states inside list items

✔ To improve performance

Example issue without keys:

- Reordering list → React confuses items and may show wrong data in wrong row.

⬛ Example

```
const items = ["A", "B", "C"];

items.map(item => <li key={item}>{item}</li>);
```

❌ Wrong key usage → index

```
<li key={index}>Item</li>
```

This breaks when:

- items reorder
- items get deleted

⬛ Summary line

Keys uniquely identify list items, enabling React to update only changed items and preserve correct UI behavior.

✅ 4) Webpack / Build bundling concept

⬛ Explanation (simple, conceptual)

Webpack is a **module bundler** used by React build tools (CRA, Vite alternatives).

It takes your:

- JS files
- CSS
- Images
- Assets

... and bundles them into **optimized build output**.

Bundling includes:

- minification
- tree-shaking
- dead code elimination
- splitting into chunks
- optimization of imports
- converting ES6+ to browser-compatible JS

⬛ When/Why is it used?

✔️ To reduce JS bundle size

✔️ To improve performance by loading required chunks

✔️ To support older browsers

✔️ To convert JSX → JS

✔️ To package entire app into `/build` or `/dist`

⬛ Example (conceptual)

Input:

```
src/
  index.js
  App.js
  utils/math.js
```

Webpack bundles into:

```
build/
  main.js (minified, tree-shaken)
  vendors~main.js
  css/main.css
```

In production:

- React code is optimized
- Comments removed
- White spaces removed
- Only necessary components included

■ Summary line

Webpack bundles and optimizes your React code into efficient production-ready files with tree-shaking and code splitting.

📌 UI Coding Based Interview Questions

Great — I'll cover each UI coding question with a clear design explanation, when/why to use the pattern, a practical example (React + minimal CSS / behavior), and a one-line interview summary. I'll keep the code copy-paste friendly and focused on interview-readiness.

■ Dashboard with sidebar + content

■ Explanation (simple, conceptual)

A dashboard layout typically has a persistent sidebar (navigation, filters) and a main content area (charts, cards, tables). Use CSS Grid or Flexbox for responsive layout; collapse the sidebar on small screens. Keep components small: `Sidebar`, `Topbar`, `Content`, `Widget/Card`.

■ When/Why is it used

✔ Admin panels, analytics dashboards, internal tools — when frequent navigation and data density are needed.

✔ Sidebar gives quick access; content area displays the current view.

■ Example / Code snippet (React + CSS using Grid, mobile-first)

```
/* Dashboard.jsx */
import React from "react";
import "./dashboard.css";
```

```
export default function Dashboard() {
 return (
  <div className="layout">
   <aside className="sidebar" aria-label="Main Navigation">
    <div className="brand">MyApp</div>
    <nav>
     <ul>
      <li className="active">Overview</li>
      <li>Reports</li>
      <li>Users</li>
      <li>Settings</li>
     </ul>
    </nav>
   </aside>

   <div className="main">
    <header className="topbar">
     <input placeholder="Search..." />
     <div className="profile">Mohini</div>
    </header>

    <section className="content">
     <div className="grid">
      <div className="card">Metric 1</div>
      <div className="card">Metric 2</div>
      <div className="card">Metric 3</div>
      <div className="card">Metric 4</div>
     </div>

     <div className="panel">
      <h3>Recent activity</h3>
      <p>List, chart or table goes here.</p>
     </div>
    </section>
   </div>
  </div>
  );
}
```

```
/* dashboard.css - mobile-first */
.layout { display: grid; grid-template-columns: 1fr; gap: 1rem; min-height:100vh; }
.sidebar { background:#0f172a; color:#fff; padding:1rem; }
.main { padding:1rem; background:#f8fafc; }
.topbar { display:flex; justify-content:space-between; gap:1rem; margin-bottom:1rem; }
.grid { display:grid; gap:0.75rem; grid-template-columns: 1fr; }
.card{ background:#fff; padding:1rem; border-radius:8px; box-shadow:0 4px 10px rgba(2,6,23,0.06); }

/* tablet+ → two columns: sidebar + main */
@media(min-width:768px){
  .layout { grid-template-columns: 220px 1fr; }
  .grid { grid-template-columns: repeat(4, 1fr); }
}
```

■ Short summary line to speak in interview

"Use a responsive Grid/Flex layout with a persistent sidebar and main content area; collapse or hide the sidebar on small screens for mobile usability."

■ Form with validation & submit API

■ Explanation (simple, conceptual)

A form should use controlled components for validation, show inline errors, disable submit while pending, and handle server-side validation messages. Use `useState` for values + `useEffect` or validation library (Yup + Formik or React Hook Form) for complex rules. Always show success/error to the user.

■ When/Why is it used

✔ For login, registration, edit forms, or any user input which must be validated and sent to backend.

✔ Ensures good UX (instant validation) and prevents invalid data reaching the API.

■ **Example / Code snippet (React + fetch, simple client-side validation)**

```
/* SimpleForm.jsx */
import React, { useState } from "react";

export default function SimpleForm() {
  const [form, setForm] = useState({ name: "", email: "" });
  const [errors, setErrors] = useState({});
  const [status, setStatus] = useState("idle"); // idle|loading|success|error

  const validate = (f) => {
    const e = {};
    if (!f.name.trim()) e.name = "Name is required";
    if (!/^\S+@\S+\.\S+$/.test(f.email)) e.email = "Invalid email";
    return e;
  };

  const handleChange = (e) => {
    setForm((s) => ({ ...s, [e.target.name]: e.target.value }));
  };

  const handleSubmit = async (ev) => {
    ev.preventDefault();
    const v = validate(form);
    setErrors(v);
    if (Object.keys(v).length) return;
    setStatus("loading");
    try {
      const res = await fetch("/api/submit", {
        method: "POST",
        headers: { "Content-Type":"application/json" },
        body: JSON.stringify(form)
      });
      if (!res.ok) throw new Error("Server error");
      setStatus("success");
    } catch (err) {
      setStatus("error");
      setErrors({ server: err.message });
    }
  };

  return (
    <form onSubmit={handleSubmit} noValidate>
      <div>
        <label>Name</label>
        <input name="name" value={form.name} onChange={handleChange}/>
        {errors.name && <div className="err">{errors.name}</div>}
      </div>

      <div>
        <label>Email</label>
        <input name="email" value={form.email} onChange={handleChange}/>
        {errors.email && <div className="err">{errors.email}</div>}
      </div>

      {errors.server && <div className="err">{errors.server}</div>}

      <button type="submit" disabled={status==="loading"}>
        {status==="loading" ? "Submitting..." : "Submit"}
      </button>

      {status==="success" && <div className="ok">Submitted successfully</
div>}
    </form>
  );
}
```

■ **Short summary line to speak in interview**

“Use controlled inputs with client-side validation, submit via async API call, show loading and server errors, and consider libraries like React Hook Form for large forms.”

⬛ Todo list with add / delete / edit

⬛ Explanation (simple, conceptual)

Todo list demonstrates basic CRUD in UI: add (create), list (read), edit (update), delete (remove). Use local state for small apps and API + optimistic updates for server-backed data. Each item needs a stable key; editing can be inline or modal-based.

⬛ When/Why is it used

✔ Common interview coding task demonstrating state management, list rendering, controlled inputs, and event handling.

⬛ Example / Code snippet (React, simple local state)

```
/* Todo.jsx */
import React, { useState } from "react";

export default function TodoApp() {
  const [todos, setTodos] = useState([]);
  const [text, setText] = useState("");
  const [editId, setEditId] = useState(null);

  const add = () => {
    if (!text.trim()) return;
    if (editId !== null) {
      setTodos((t) => t.map(it => it.id===editId ? {...it, text} : it));
      setEditId(null);
    } else {
      setTodos((t) => [...t, { id: Date.now(), text }]);
    }
    setText("");
  };
  const remove = (id) ⇒ setTodos(t ⇒ t.filter(x ⇒ x.id !== id));
  const startEdit = (item) ⇒ { setEditId(item.id); setText(item.text); };

  return (
    <div>
      <input value={text} onChange={e⇒setText(e.target.value)} placeholder
="Add todo"/>
      <button onClick={add}>{editId ? "Update" : "Add"}</button>

      <ul>
        {todos.map(todo ⇒ (
          <li key={todo.id}>
            <span>{todo.text}</span>
            <button onClick={()⇒startEdit(todo)}>Edit</button>
            <button onClick={()⇒remove(todo.id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

⬛ Short summary line to speak in interview

"Implement add/delete/edit by keeping todos in state, using stable keys and controlled inputs; switch to server calls with optimistic updates for production."

⬛ Table showing list of users with pagination & filter

⬛ Explanation (simple, conceptual)

A data table should support server-side pagination/filtering for large datasets. Client-side pagination only for small sets. Implement a `page`, `pageSize`, and `filter` state; request backend with those parameters; show loading, total count, and simple pagination controls. Use debounced filters.

⬛ When/Why is it used

✔ For admin UIs, user management, reporting — when dataset can be large and must be paginated/filterable for performance.

■ **Example / Code snippet (React, server-side pagination pattern)**

```
/* UsersTable.jsx */
import React, { useEffect, useState } from "react";

export default function UsersTable() {
  const [users, setUsers] = useState([]);
  const [page, setPage] = useState(1);
  const [pageSize] = useState(10);
  const [total, setTotal] = useState(0);
  const [q, setQ] = useState("");
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    let mounted = true;
    const fetchData = async () => {
      setLoading(true);
      try {
        const res = await fetch(`/api/users?page=${page}&size=${pageSize}&q=
${encodeURIComponent(q)}`);
        const payload = await res.json();
        if (!mounted) return;
        setUsers(payload.items);
        setTotal(payload.total);
      } catch (err) {
        console.error(err);
      } finally { if (mounted) setLoading(false); }
    };
    fetchData();
    return () => { mounted = false; };
  }, [page, pageSize, q]);

  const totalPages = Math.ceil(total / pageSize);
  return (
    <div>
      <input placeholder="Search..." value={q} onChange={e=>setQ(e.target.value)} />
      {loading ? <div>Loading...</div> :
        <table>
          <thead><tr><th>Name</th><th>Email</th></tr></thead>
          <tbody>
            {users.map(u => <tr key={u.id}><td>{u.name}</td><td>{u.email}</td></tr>)}
          </tbody>
        </table>
      }
      <div>
        <button onClick={()=>setPage(p => Math.max(1, p-1))} disabled={page===1}>Prev</button>
        <span>{page}/{totalPages}</span>
        <button onClick={()=>setPage(p => Math.min(totalPages, p+1))} disabled={page===totalPages}>Next</button>
      </div>
    </div>
  );
}
```

Backend expected response:

```
{ "items": [ ... ], "total": 234 }
```

■ **Short summary line to speak in interview**

“Prefer server-side pagination and filtering for large tables: send page/size/filter to API and render received page with proper keys and loading state.”

■ Build expense list component UI (related to your project)

■ **Explanation (simple, conceptual)**

Expense list shows user expenses with date, category, amount and actions (edit/delete). Use a list or compact table; support sorting, filtering by date/category, and optionally pagination. For better UX, show summary totals and a lightweight inline edit.

■ **When/Why is it used**

✔ Financial dashboards, budget apps — to present transaction-level details and allow edits or exports.

■ **Example / Code snippet (React functional component + minimal styles)**

```
/* ExpenseList.jsx */
import React, { useMemo } from "react";

export default function ExpenseList({ expenses = [], onDelete, onEdit }) {
  // compute total
  const total = useMemo(() ⇒ expenses.reduce((s,e) ⇒ s + e.amount, 0), [exp
enses]);

  return (
    <section>
      <h3>Expenses</h3>
      <div>Total: ₹{total.toFixed(2)}</div>
      <ul style={{ listStyle:'none', padding:0 }}>
        {expenses.map(exp ⇒ (
          <li key={exp.id} style={{ display:'flex', justifyContent:'space-between',
padding: '8px 0', borderBottom:'1px solid #eee' }}>
            <div>
              <div style={{ fontWeight:600 }}>{exp.title}</div>
              <div style={{ color:'#666', fontSize:12 }}>{exp.category} • {new Date
(exp.date).toLocaleDateString()}</div>
            </div>
            <div style={{ textAlign:'right' }}>
              <div style={{ fontWeight:700 }}>₹{exp.amount.toFixed(2)}</div>
              <div>
          <button onClick={() ⇒ onEdit(exp)}>Edit</button>
          <button onClick={() ⇒ onDelete(exp.id)}>Delete</button>
         </div>
        </div>
       </li>
      ))}
     </ul>
    </section>
  );
}
```

Usage hint: connect to API for fetch/update/delete; show optimistic UI or confirm deletion modal.

⬛ **Short summary line to speak in interview**

"Render an expense list with title/date/category/amount, compute total via useMemo, and provide edit/delete callbacks that call backend with optimistic UI patterns."

⬛ How to structure folder in scalable React project?

⬛ **Explanation (simple, conceptual)**

Use a feature-based (or domain-driven) structure for better scalability vs a flat type-based structure. Group components, hooks, services, pages, and styles by feature. Keep shared utilities and global state separate. Keep each folder cohesive and include tests next to code.

⬛ **When/Why is it used**

✔ Keeps repository maintainable as app size grows.

✔ Easier to onboard new developers.

✔ Encourages encapsulation and code ownership.

⬛ **Recommended folder structure (example)**

```
src/
  api/              # API clients (axios instances, endpoints)
  index.js
  users.js
app/              # app-wide config, routes, store init
  routes.jsx
  store.js
features/
  dashboard/
    components/
      MetricCard.jsx
    Dashboard.jsx
    dashboard.css
    dashboard.slice.js
    index.js
  expenses/
    ExpenseList.jsx
    ExpenseForm.jsx
    expenses.api.js
    expenses.slice.js
    styles.css
  users/
    UsersTable.jsx
    users.api.js
components/       # shared presentational components
  Button/
    Button.jsx
    Button.css
  Modal/
hooks/            # reusable hooks
  useDebounce.js
  useFetch.js
libs/             # utility functions
  date.js
  currency.js
pages/            # top-level pages (login, home)
  LoginPage.jsx
  NotFound.jsx
styles/            # global styles, variables
  variables.css
index.jsx
App.jsx
```

Key principles:

- Co-locate feature code (component + styles + tests).
- Keep `api/` for HTTP code & `hooks/` for reusable logic.
- Use `components/` for UI building-blocks.
- Use slices (Redux Toolkit) or feature-specific stores for state.
- Add tests next to components: `Component.test.jsx`.

■ Short summary line to speak in interview

"Use a feature-based folder structure (features/expenses, features/users) with shared components, hooks, and api clients separated — it scales well and improves ownership and testability."

If you prepare these questions thoroughly — **HTML/CSS/React panel is covered 100% for your interview level.**
