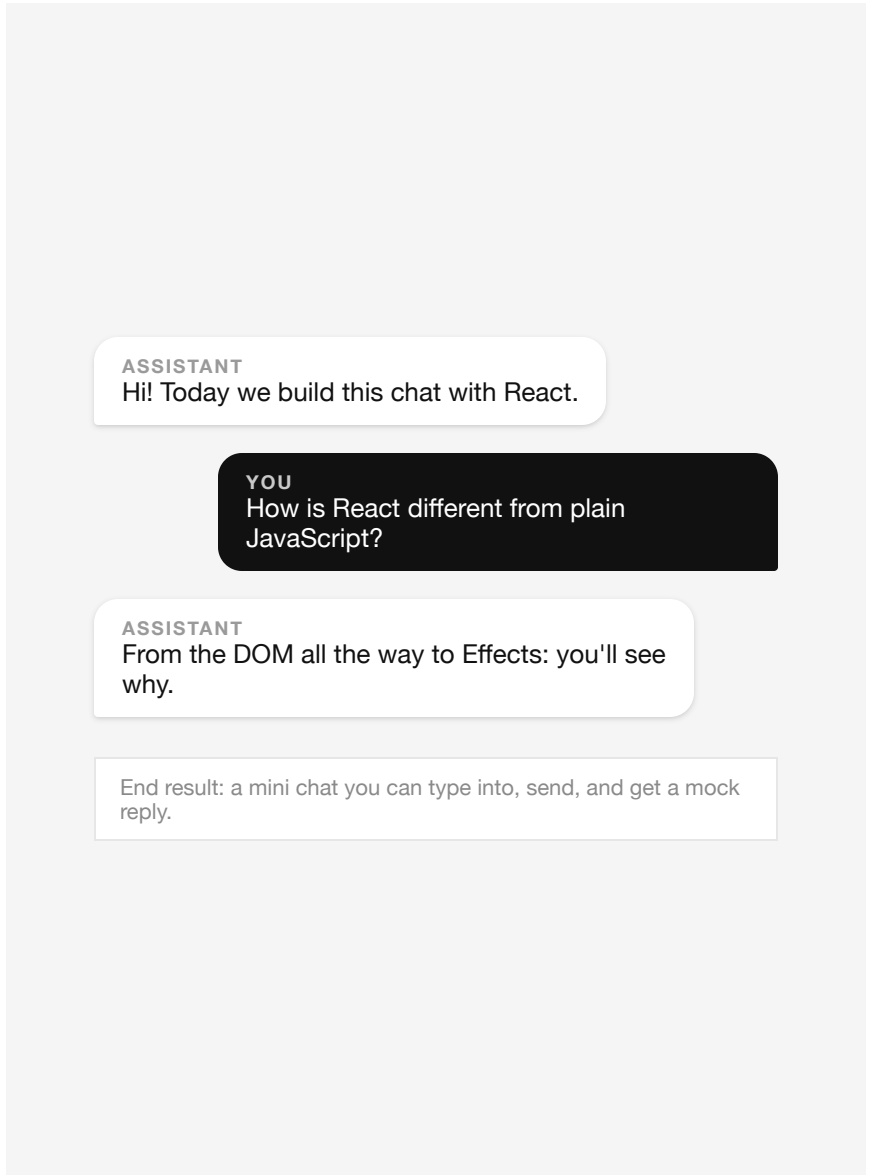


WEB DEVELOPMENT ACADEMY

React Basics

Components, props, state and effects — one chatbox assembled piece by piece

JSX | Components | Props | State | Effects

LESSON ROADMAP

What we build today

01	**Web, DOM and React**	Recap HTML/CSS/JS · how pages became apps · manual DOM updates · why React exists · installation and project structure
02	**Components, JSX & Props**	JSX syntax · first component · imports · props · conditional rendering · lists · Labs 1–4
03	**State & Events**	State triggers render and commit · controlled input · props vs state · array state · Labs 5–7
04	**Lifecycle, Effects & Lists again**	Mount / update / unmount · useEffect with cleanup · Lab 8: auto-scroll · how every topic meets back at list rendering
05	**Wrap-up**	Exit ticket: DOM or state first? · sort work into render / event / effect · the road to a real chatbot

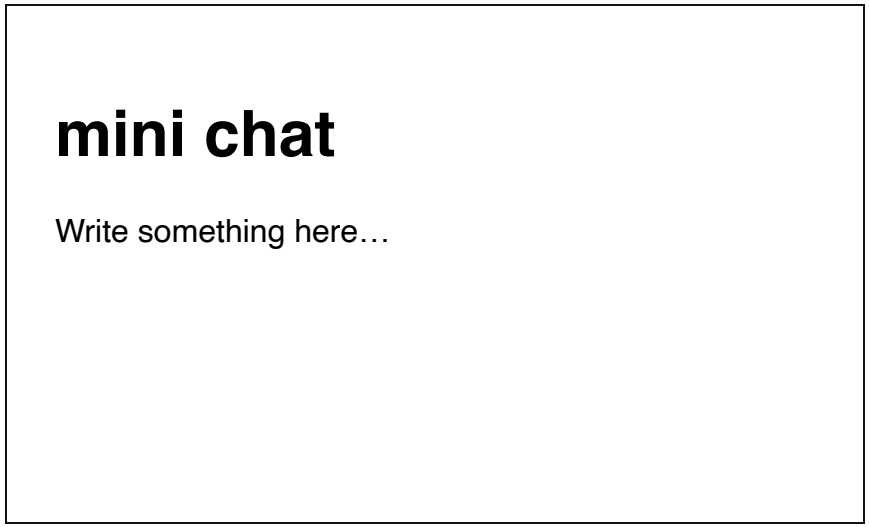
Before React: build this by hand

chat-header.html — edit it, the preview reruns as you type

```
<style>
  body {
    margin: 0;
    font-family: -apple-system, sans-serif;
    background-color: #fffbe8; /* try a different color */
  }
  .card {
    padding: 24px;
  }
</style>

<div class="card">
  <h1>mini chat</h1>
  <p>Write something here…</p>
</div>
```

preview — live, from your code



- change `background-color` — the whole card follows
- edit the <h1> and <p> text — still just tags
- add a second <p> — nothing updates itself, you type every line

the point: every line above was typed by a human. The rest of today: teach the screen to type them for you, from data.

Read the data before you render it

chat.js — intellisense pops up while you type

```
1  const messages = [
2    { id: "m1", role: "assistant", text: "Hi! This is a mock chat" },
3    { id: "m2", role: "user", text: "How is React different?" },
4    { id: "m3", role: "assistant", text: "We describe the screen from data"
5  ];
6
7  console.log(messages);
8  console.log(messages.length);
9  console.log(messages[0]);
10 console.log(messages[0].text);
11
12 // your turn: log the role of the last message
```

console — live output, reruns as you type

```
[{"id":"m1","role":"assistant","text":"Hi! This is a mock chat"},{"id":"m2","role":"user","text":"How is React different?"},{"id":"m3","role":"assistant","text":"We describe the screen from data"}]
3
{"id":"m1","role":"assistant","text":"Hi! This is a mock chat"}
Hi! This is a mock chat
```

- `messages` is an **array of objects** — same shape you'll use in every lab today
- `messages[0]` reads **one item**; `.text` reads **one field** off it
- **console.log before you build the screen**: check the shape of your data first

Try it: `console.log(messages[3])` — there is no fourth message. What prints, and is that an error?

Loop through data, one index at a time

loop.js — intellisense pops up while you type

```
1 const names = ["Ada Lovelace", "Grace Hopper", "Katherine Johnson
2
3 for (let index = 0; index < names.length; index++) {
4   const name = names[index];
5   console.log(index + " " + name);
6 }
7
8 // your turn: also console.log(names.length) before the loop starts
```

console — live output, reruns as you type

```
0 Ada Lovelace
1 Grace Hopper
2 Katherine Johnson
```

- `index` starts at **0**, stops *before* `names.length` — three names means indexes 0, 1, 2
- `names[index]` is how the loop reads **one item** — same bracket notation from last class
- add a fourth name to the array — the loop needs **zero changes**: it already reads `.length`

hold that thought: a `for` loop is one way to visit every item — later today, `.map()` does the same job, but returns the screen instead of a console line.

The web begins as documents

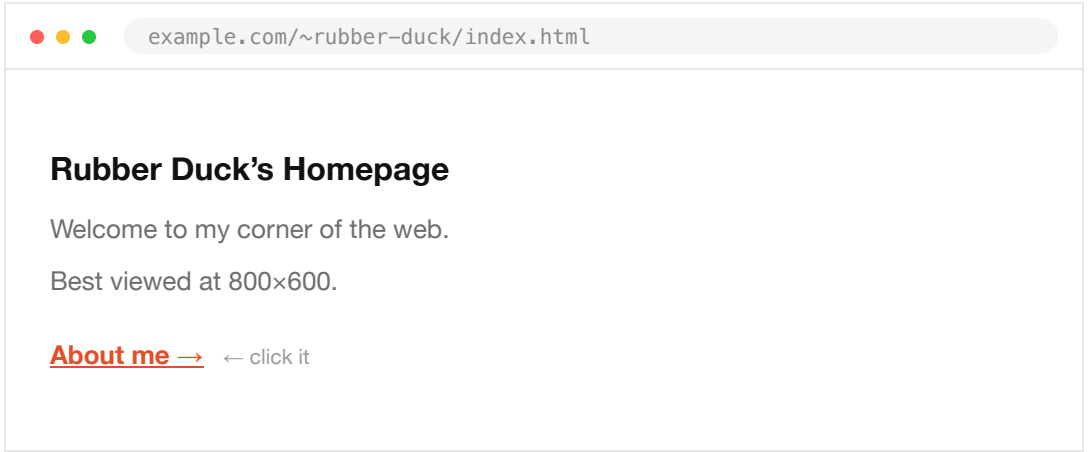
1990s **Documents** — 1995 **CSS + JS** — 2000s **Web apps** — 2013 **React**

1 / 4 · THE 1990s

Documents

The web begins as a library of pages: HTML documents sitting on a server, linked together.

- one tool: HTML writes the content **and** the look
- every click asks the server for a **whole new page**
- the browser is a viewer: it renders what arrived, then waits



Click the link: the server sends a **whole new page** — the browser redraws every part of it.

The wall: nothing on the page can react — the server decides everything.

Style and behavior split off

1990s — Documents
1995 — CSS + JS
2000s — Web apps
2013 — React

2 / 4 · 1995–1996

Style & behavior

Browsers add two companions to HTML: CSS for the look, JavaScript for behavior.

- CSS moves the styling **out of the markup**
- JS can change the page — by commanding the DOM **one call at a time**
- suddenly parts of the page can respond **without a reload**

HTML · STRUCTURE + CSS · STYLE + JS · BEHAVIOR

guestbook.js — one change, one command

```
document.querySelector("#visits")
  .textContent = "3rd visit!";
```

Guestbook

Rubber Duck was here.

visits: —

run the script | Reset

The page can now change **without a reload** — but the script must name and command every node itself.

The wall: every change is a hand-written command — find the node, tell it what to become.

The web becomes apps

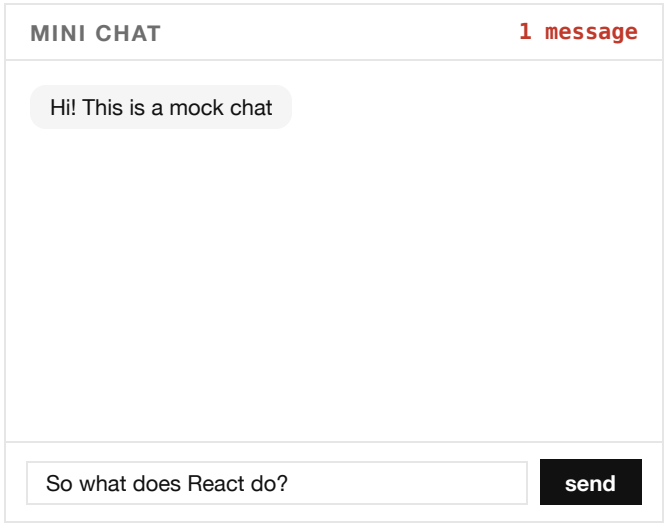
1990s — Documents
1995 — CSS + JS
2000s — Web apps
2013 — React

3 / 4 · THE 2000s

Web apps

Mail, chat, maps, feeds: pages become software people live in.

- one tap changes **many parts** of the screen at once
- those parts all read the **same data** — they must change together
- hand-written DOM code must **remember every spot**



Press send — count the commands

1 · append the bubble
2 · counter + 1
3 · clear the input
4 · add the reply
5 · counter + 1 again

Five updates, every send — all remembered by hand.

Reset

The wall: keep every part in sync by hand — and every feature adds more spots to remember.

The page on screen is the DOM, not your HTML

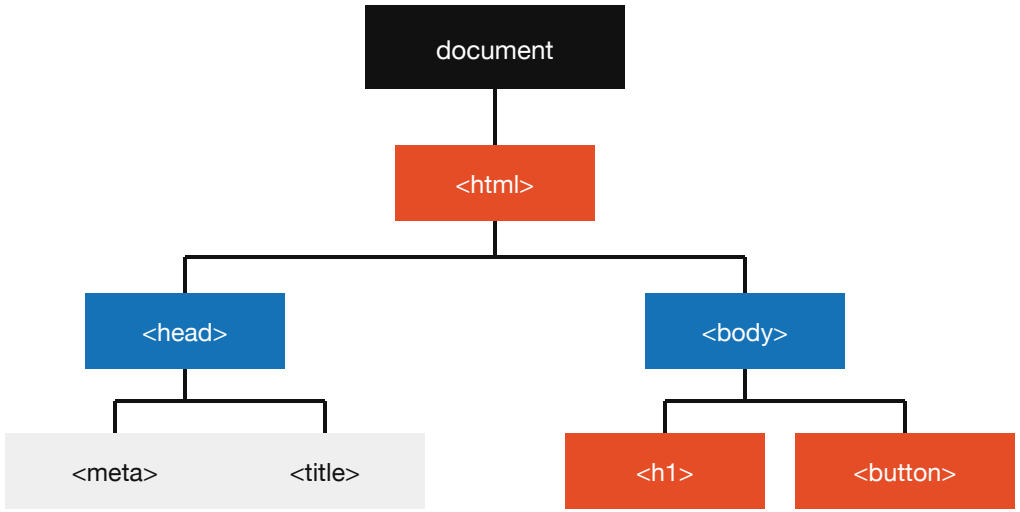


FIG. 06 — THE BROWSER READS YOUR HTML INTO THIS LIVING TREE

HTML — the file we write — the browser reads it once, at load time

DOM — the tree the browser builds in memory — what the screen renders and programs change

JAVASCRIPT — listens to the user, then adds / removes / edits nodes in the tree

append.js · one new message, by hand

```
const item = document.createElement("li");  // 1 create the node
item.className = "me";
item.textContent = message;                 // 2 fill it in
document.querySelector("#log").append(item); // 3 attach it
```

- Three commands **per message** — and every spot you touch by hand must remember its siblings: the counter, the input, the reply…
- What changes is **the DOM, never the HTML file**: DevTools → Elements shows the tree as it is **right now**, not the source

Your HTML is read once — after that, every change is a DOM change. Next slide: you drive this by hand — and feel why one piece of data beats five commands.

Command the DOM by hand

append.js: the header count is your job too

```
// The header shows a message count: append must keep it honest
// too, or the screen lies. Every spot you must remember by
// hand is pain — hold that feeling for the next slide.
// TODO: append a few messages, then make the count follow

function appendMessage(text) {
  const item = document.createElement("li");
  item.className = "me";
  item.textContent = text;
  document.querySelector("#log").append(item);
}

appendMessage("first message, by hand");
```

preview: mini chat (manual DOM)

MINI CHAT · MANUAL DOM · 1 MESSAGES

Hi!

first message, by hand

typed in the form

Add message

React: the screen as a function of data

1990s — Documents
1995 — CSS + JS
2000s — Web apps
2013 — React

4 / 4 · 2013

React arrives

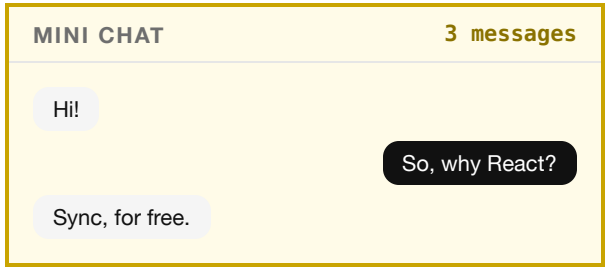
Facebook open-sources React — a library **on top of** HTML, CSS and JS, not instead of them.

- you **describe the screen from data**: UI = f(data)
- React works out **which DOM nodes** must change
- today's labs rebuild this chat **the React way**

```
the data — one array
const messages = [
  { me: false, text: "Hi!" },
  { me: true, text: "So, why React?" },
  { me: false, text: "Sync, for free." },
];
```

UI = f(data)

the component describes the screen from data



The same screen, now **derived from the array**: change the data in one place and **React** works out every DOM update — even the counter.

The answer: React doesn't replace HTML, CSS or JavaScript — describe the screen from data, and the DOM sync work is **no longer yours**.

Derive the screen from one data source

MANUAL DOM the author must remember every spot

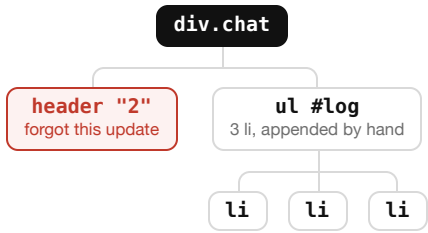
MINI CHAT 2 messages

Hi! This is a mock chat

How is React different?

We describe the screen from data

Three messages were sent, but the counter still says **2**: the code that appended the li forgot the header.



UI FROM DATA React derives every part from one array

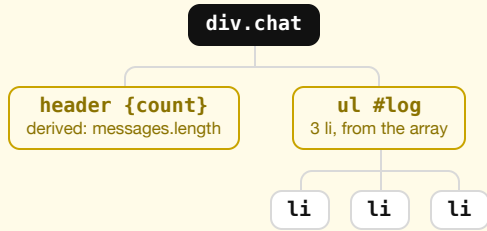
MINI CHAT 3 messages

Hi! This is a mock chat

How is React different?

We describe the screen from data

The list and the counter both read the same `messages` array: change the data in one place and every reader follows.



data (messages) → components compute "what the screen should be" → React touches only the DOM that needs it

Ask: if the counter is derived from the same array as the list, do you still have to remember to update it?

COMPONENT LIFECYCLE `mount` → `update` → `unmount`

every re-render you just saw is an **update**: same component, new data. Full lifecycle + Effects: later today.

One command, a dev server with hot reload

STEP 1

Install Node.js

LTS build from nodejs.org. It ships `npm`, the package manager every command below runs through.

STEP 2

Scaffold the project

`npm create vite@latest` — pick a project name, then choose **React** as the framework.

STEP 3

Install dependencies

`cd` into the new folder, then `npm install` — downloads React and every package Vite scaffolded into `node_modules/`.

STEP 4

Start the dev server

`npm run dev` — open the printed localhost address. Save a file → the browser updates itself.

Check yourself: edit the text inside <App /\>, save, and watch it change without a manual refresh. That loop — save, see it instantly — is what every lab today runs on.

terminal

```
$ npm create vite@latest my-app
✔ Select a framework: › React
✔ Select a variant: › JavaScript

$ cd my-app
$ npm install
$ npm run dev

  VITE ready in 320 ms
  ➜  Local:   http://localhost:5173/
```

No install today? Every lab in this deck runs its own sandboxed React preview right in the browser — the same App you'll write locally.

Every file has one job

```
my-app/
my-app/
├── node_modules/   ← installed packages
├── public/
│   └── vite.svg    ← static files, served as-is
├── src/
│   ├── assets/     ← images, fonts
│   ├── App.jsx     ← the root component
│   ├── App.css
│   ├── main.jsx    ← entry point
│   └── index.css
├── index.html      ← the one real HTML page
├── package.json    ← scripts + dependencies
└── vite.config.js
```

- `index.html` is nearly empty: one `<div id="root"></div>` and a `<script>` tag pointing at `main.jsx`
- `main.jsx` finds that `#root` div and mounts `<App />` into it — this is the only file that touches the DOM directly
- `App.jsx` is a component like any other: everything you build today lives in `src/`, usually one component per file
- `node_modules/` is generated by `npm install` — never edit it, never commit it

The chain: `index.html` → `main.jsx` renders `<App />` → `App.jsx` renders whatever you build — every lab today is really editing one link in this chain.

One card, three ways

HTML: WRITTEN AHEAD OF TIME

```
<div class="card">
  <img class="avatar" src="ada.png">
  <p>Ada Lovelace</p>
</div>
```

Frozen: the text can't change, you write every card by hand

MANUAL DOM: COMMANDED STEP BY STEP

```
const card = document.createElement("div");
card.className = "card";
const img = document.createElement("img");
img.className = "avatar";
card.append(img);
card.append(p);
```

Flexible but long, and you must remember every update yourself

JSX: DESCRIBED FROM DATA

```
function AvatarCard() {
  return (
    <div className="card">
      <img className="avatar" src="ada.png" />
      <p>Ada Lovelace</p>
    </div>
  );
}
```

Short, and when the data changes, the screen follows

JSX is not an HTML string and not a new language: it's a way of **writing JavaScript that contains markup**: still a function, still a `return`, and the markup gets compiled to JavaScript before the browser runs it

Predict first: what does the JSX we write arrive at the browser as: a new HTML file, JavaScript, or the DOM directly?

Three rules the compiler enforces

BROKEN.JSX 3 mistakes — find them

```
function Bio() {
  return (
    <h1>Ada Lovelace</h1>
    ①<p ②class="intro">First programmer</p>
    <img src="ada.png"③>
  );
}
```

FIXED.JSX it compiles

```
function Bio() {
  return (
    <>
      <h1>Ada Lovelace</h1>
      <p className="intro">First programmer</p>
      <img src="ada.png" />
    </>
  );
}
```

① **return one root element**: siblings need a wrapper `<div>` — or an empty `<>…</>` fragment

②③ **close every tag**: `<img />` has no children · attributes are **camelCase**: `class` → `className`, `stroke-width` → `strokeWidth`

✓ the browser never sees JSX: it's compiled to JavaScript first — got HTML to convert? `transform.tools/html-to-jsx`

Predict: `<img CLASS="x" />` — which rule does it break, and what would you write instead?

Curly braces: a window into JavaScript

Profile.jsx · markup that reads JavaScript

```jsx
const person = {
  name: "Gregorio Y. Zara",
  theme: { background: "#111", color: "#fff" },
};

function Profile() {
  return (
    <section ①style={person.theme}>
      <h1>②{person.name}'s todos</h1>
      <img
        className="avatar"
        ③src={getImageUrl(person)}
        alt={person.name}
      />
    </section>
  );
}
```

① **attributes too**: {…} replaces the quotes — `style` takes a whole object

② **text**: the value of the expression is rendered in place

③ **any expression fits**: a variable, `{2 + 2}`, a function call `{formatDate(today)}`

✗ **statements don't**: `if` and `for` are not expressions — branching gets its own slide next

The quotes are the boundary: `person.name` prints the literal text — `{person.name}` prints **Gregorio Y. Zara**. Rendering logic and content live in the same place.

Predict: why does `{ if (done) "✔" }` break the build, and how would you write the same idea correctly?

A component is a JavaScript function

Profile.jsx · the smallest useful component

```jsx
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      </section>
  );
}
```

a plain function · the name starts with a**capital letter**: `Profile`, never `profile` — a lowercase tag is read as an HTML element, not as your function

it returns markup: that description *is* the screen — plain JS commands the DOM, React describes it (the syntax is JSX: two slides ahead)

one definition, many instances: `<Profile />` three times = three avatars on screen — the markup was written once

components render components: Gallery renders Profile — the nesting grows into a **component tree** (last slide of this section)

In plain JavaScript the screen is the **side effect of commands**. In React the screen is the **return value of a function**: you describe the UI, React makes it so.

Split it up: export, import, render

APP.JSX

```jsx
import Gallery from "./Gallery.js";

export default function App() {
  return <Gallery />;
}
```

GALLERY.JSX

```jsx
import Profile from "./Profile.js";

export default function Gallery() {
  return (
    <section>
      <Profile />
      <Profile />
    </section>
  );
}
```

PROFILE.JSX

```jsx
export default function Profile() {
  return <img src="…" alt="Katherine Johnson" />;
}
```

one component per file · `export default` = "this is the one thing this file offers"

`import Profile from "./Profile.js"`: the path is relative to this file — then use it like any tag

why split: one huge file gets hard to navigate — and the **import chain (App → Gallery → Profile) is a tree** that bundlers walk to build your JavaScript (last slide)

Predict: two files both export a `Profile`: what does `import Profile from "./B.js"` pick — the one from B, both of them, or whichever loads first?

Your first component: MessageBubble

index.html | main.jsx | App.jsx | index.css | App.css

```
1  import "./App.css";
2
3  // Lab 1: your first component
4  // A component is a JavaScript function that returns markup.
5  // TODO 1: put the message text between <p> and </p>
6  // TODO 2: the tag in App is broken: it must match the function name
7  //         exactly, starting uppercase → <MessageBubble />
8  // TODO 3: render one more bubble: add a second <MessageBubble />
9
10 function MessageBubble() {
11   return (
12     <li className="bubble them">
13       <span className="label">Assistant</span>
14       <p className="text"></p>
15     </li>
16   );
17 }
18
19 export default function App() {
20   return (
21     <div className="chat">
22       <header>mini chat</header>
```

preview: `mini chat`

MINI CHAT

← broken: lowercase = an unknown HTML tag

SUCCESS CRITERIA

Show solution | Check answers

- ○ two bubbles render: one per <MessageBubble />
- ○ the message text is between the tags: markup goes inside
- ○ no lowercase tag left: components start uppercase

Complete the TODOs, then press Check answers

your move: ① put the message text between and ② fix the broken tag in App so it renders `<MessageBubble />` ③ add a second `<MessageBubble />`, then press **Check answers**.

props: the data coming in

the parent decides, the child renders

```jsx
function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      width={size}
    />
  );
}

function Profile() {
  return (
    <Avatar
      person={{ name: "K. Saruhashi", imageId: "YfeOqp2" }}
      size={100}
    />
  );
}
```

PROPS CAN BE JSX ITSELF: CHILDREN

```jsx
function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
  );
}

// whatever the caller nests inside is children
<Card>
  <Avatar person={katsuko} size={100} />
</Card>
```

Every JavaScript value fits in a prop: strings, numbers, objects, arrays, functions — and JSX. The destructuring `{ person, size }` is a plain function parameter: one object, unpacked.

— props flow **one way, down**: the caller passes them, the component only reads them — a **snapshot** it never edits

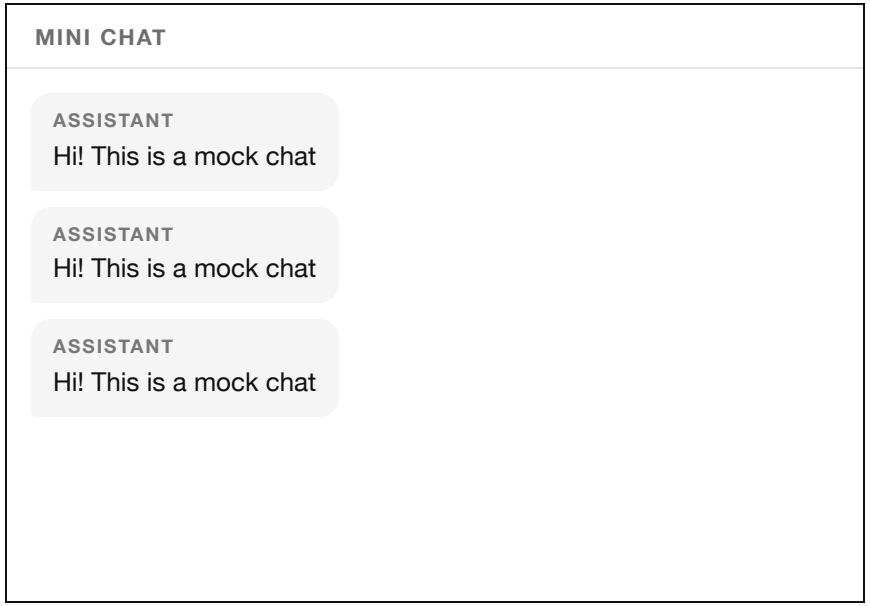
Predict: `<Avatar size="100" />` vs `<Avatar size={100} />` — what is the type of `size` in each, and why does it matter for `width={size}`?

Pass props to MessageBubble

index.html | main.jsx | App.jsx | index.css | App.css

```
1  import "./App.css";
2
3  // Lab 2: pass props to the component
4  // App below already passes props: name and text.
5  // TODO 1: catch them in the signature:
6  //   function MessageBubble({ name, text }) {
7  // TODO 2: render each one with curly braces, replacing the hardcoded
8  //   label and text: {name} and {text}
9  // props flow one way: read them inside, never edit them
10
11 function MessageBubble() {
12   return (
13     <li className="bubble them">
14       <span className="label">Assistant</span>
15       <p className="text">Hi! This is a mock chat</p>
16     </li>
17   );
18 }
19
20 export default function App() {
21   return (
22     <div className="chat">
```

preview: `mini chat`



SUCCESS CRITERIA

Show solution | Check answers

- ○ three bubbles: one per <MessageBubble /> in App
- ○ every bubble shows its text prop: curly braces render it
- ○ the label comes from the name prop: Assistant / You / Assistant

Complete the TODOs, then press Check answers

your move: ① catch `{ name, text }` in the signature ② render each one with curly braces instead of the hardcoded label and text ③ **Check answers**: three calls, three different bubbles.

Branching JSX: if, ternary, &&

EARLY RETURN

```jsx
function Item({ name, isPacked }) {
  if (isPacked) {
    return <li>{name} ✔</li>;
  }
  return <li>{name}</li>;
}
```

one `return` per branch — clearest when the JSX is big

TERNARY: ONE OF TWO

```jsx
<li>
  {isPacked
    ? name + " ✔"
    : name}
</li>
```

an expression, so it fits inside `{…}` — pick A or B inline

RENDER ONLY WHEN TRUE

```jsx
<li>
  {name} {isPacked && "✔"}
</li>
```

hide / show with one symbol: false renders **nothing at all**

the && trap

`{count && <Badge />}` with `count = 0` prints a lonely **0**: 0 is falsy, but it's still a number React will happily render. Guard the comparison: `{count > 0 && <Badge />}`

Conditional rendering is not a React feature: it **is** JavaScript. The ternary is what Lab 3 uses to pick a bubble's side: `role === "user" ? "me" : "them"`

Predict: `isPacked = false` — what does the ternary render? What does the && version render?

filter(), then map(): data becomes the list

coming up in Lab 3: read it ahead

```
const people = [
  { id: 0, name: "Ada Lovelace", field: "computing" },
  { id: 1, name: "K. Saruhashi", field: "geochemistry" },
  { id: 2, name: "Grace Hopper", field: "computing" },
];

// 1. filter: pick the rows you want
const chemists = people.filter(
  (person) => person.field === "computing"
);

// 2. map: transform every row → an element
const items = chemists.map((person) => (
  <li key={person.id}>{person.name}</li>
));

return <ul>{items}</ul>;
```

`id: 0` → `id: 2` n rows of data = n elements

a key is each item's ID card

tells React **which item is which** when the list changes

use an **id from the data**: stable, never regenerated on every render

append a row → React knows the old ones are intact and adds only the new one

- `map` = transform every item → a new array of elements (the array-methods slide)
- `filter` comes first when you only want some rows: select, then render
- adding an item = adding an **object to the array**: no markup written by hand

Predict: the same name is in the data twice and you use the name as the key — what breaks, and why does the id fix it?

Build the conversation from data

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { MESSAGES } from "./messages.js";
2  import "./App.css";
3
4  // Lab 3: from data to a conversation
5  // MESSAGES is an array of objects the system prepared for you.
6  // TODO 1: render the text from props in every bubble
7  // TODO 2: use role for the sender:
8  //   className: "bubble " + (role === "user" ? "me" : "them")
9  //   label:      role === "user" ? "You" : "Assistant"
10
11 function MessageBubble({ role, text }) {
12   return (
13     <li className="bubble them">
14       <span className="label">Assistant</span>
15       <p className="text"></p>
16     </li>
17   );
18 }
19
20 // App is provided: the map() example: data becomes bubbles
21 export default function App() {
22   return (
```

preview: `mini chat`

MINI CHAT · 3 MESSAGES

ASSISTANT

ASSISTANT

ASSISTANT

SUCCESS CRITERIA

Add message | Show solution | Check answers

- ○ bubbles match the data: map covers every item
- ○ every bubble shows its prop text
- ○ sender label matches role: You / Assistant

Complete the TODOs, then press Check answers

your move: ① render `{text}` in the empty <p> ② use `role` for label + side (ternary, pattern in the TODO): then press **Add message** and watch the map keep up.

A component must not keep secrets

IMPURE touches the outside world

```
let guest = 0;

function Cup() {
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
    </>
  );
}
```

the output depends on **when** it ran, not on its input

PURE input in, JSX out

```
function Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

function TeaSet() {
  return (
    <>
      <Cup guest={1} />
      <Cup guest={2} />
      <Cup guest={3} />
    </>
  );
}
```

the count comes through **props**: render it twice, get the same screen

① **minds its own business**: it never changes variables that existed before it was called

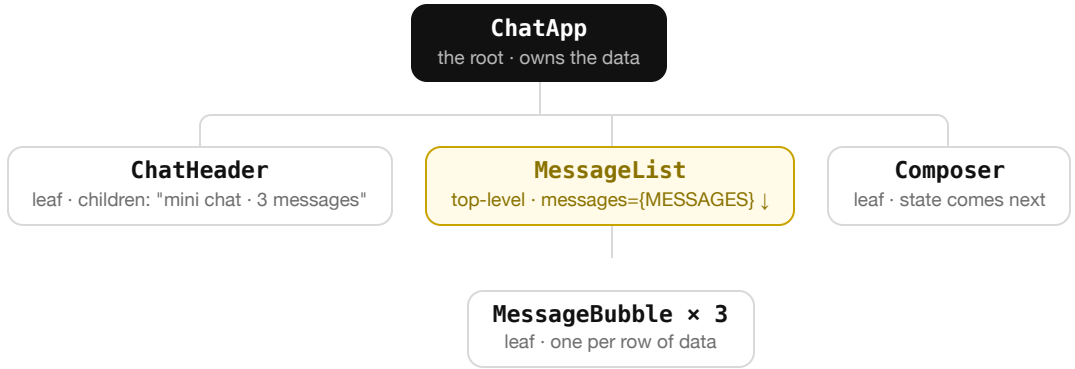
② **same inputs → same output**: given the same props, the JSX is identical

? why care: React may render a component **twice** (StrictMode) to catch impurity — impure code shows 3, 4 where you meant 1, 2

Predict: the impure `TeaSet` — what do the two cups show on the **first** render? And after a re-render?

The chatbox is a tree of components

THE RENDER TREE OF LAB 4 · EVERY EDGE = "RENDERS"



reads it in two directions:	parent → child: data flows down as props · child → parent: requests travel through callbacks; ownership stays above

parent/child is per edge, not per lineage: `ChatApp` is the **parent** of `MessageList`; `MessageList` is in turn the **parent** of every `MessageBubble` — `ChatApp` is *not* called the parent of `MessageBubble`, even though it's above it in the tree (that's a grandparent relationship: same idea, one edge further)

the render tree: what renders what — root at the top, leaves at the bottom, one edge per "renders"

top-level vs leaf: top-level components hold data and change often; leaves are small, reusable, stable

a child does not edit parent state directly: `ChatApp` passes values down as props; it can also pass a callback so a child can request an update while the parent stays the owner

when these components move into separate files, imports form a **module dependency tree**. The bundler follows that tree to build your JavaScript; today's lab keeps the small example in App.jsx

this is the exact tree **you assemble in Lab 4** — after that, state makes it come alive

Predict: a new `TypingIndicator` rendered inside `MessageList` — is it a leaf or top-level? And which file must import it?

Assemble the chatbox

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```jsx
import { MESSAGES } from "./messages.js";
import "./App.css";

// Lab 4: assemble the chatbox
// Every piece is built (Labs 1–3): now compose them into one tree:
//   App → ChatHeader + MessageList + Composer
// TODO 1: MessageList: accept { messages }, then map() every message to a
//   MessageBubble with key={message.id} (the pattern from Lab 3)
// TODO 2: ChatHeader: render {children} between the header tags
// TODO 3: App: pass the data down — messages to MessageList, and the
//   header text as children:
//   <ChatHeader>mini chat · {MESSAGES.length} messages</ChatHeader>

function MessageBubble({ role, text }) {
  return (
    <li className={"bubble " + (role === "user" ? "me" : "them")}>
      <span className="label">
        {role === "user" ? "You" : "Assistant"}
      </span>
      <p className="text">{text}</p>
    </li>
  );
```

preview: the assembled chatbox

Type a message…

SUCCESS CRITERIA

Add message | Show solution | Check answers

- ○ every message renders: the map inside MessageList covers the data
- ○ bubbles live inside the log: MessageList composes MessageBubble
- ○ the header counts from the data: text passed as children
- ○ the composer is part of the tree

Complete the TODOs, then press Check answers

your move: ① `MessageList`: map `messages` → `MessageBubble` with a `key` ② `ChatHeader`: render `{children}` ③ wire `ChatApp`: pass the data down, header text as children.

State changes trigger a re-render

predict first: does "hello" show up on screen?

```
function App() {
  let draft = "";            // a plain variable

  function handleInput() {
    draft = "hello";         // changed... the screen doesn't care
  }

  return (
    <p className="preview">Typing: {draft}</p>
  );
}
```

The component already rendered with "": silently reassigning a variable never **asks React to re-render**. You need **`useState`**: `const [draft, setDraft] = useState("")` (current value · setter that requests an update · initial value)

SIMULATED: TYPING ONE CHARACTER

input | event | setState | render | commit

`draft = "" screen: Typing:`

Step through: from the user's event to the commit.

Next step | Reset

React works in three beats

1 · TRIGGER

something gives React a reason

Data changed (say, state was set): so React starts calculating.

→

2 · RENDER

React calls your components

It asks "given this data, what should the screen be?": components answer with JSX.

→

3 · COMMIT

only the changed DOM is touched

React compares against the previous screen and updates exactly the nodes that differ.

one piece of data, read in many places

```
<header>{messages.length} messages</header>
<ul>
  {messages.map(m => <MessageBubble key={m.id} {...m} />)}
</ul>
```

- Both `header` and `ul` read the same `messages`: there is no spot you must remember to update separately
- **A re-render ≠ rebuilding the whole DOM**: the commit touches only the nodes whose values changed
- You don't need the internal algorithm: your whole job is to **describe the screen from the data, correctly**

A controlled input reads and updates state

```
these two lines are the heart of Lab 5
const [draft, setDraft] = useState("");

<input
  value={draft}
  onChange={(event) => setDraft(event.target.value)}
/>
```

type → event → `setDraft` → render → commit the same cycle as the simulation

a controlled input = state owns the value

`value={draft}`: what's displayed always comes from state

`onChange`: every keystroke asks for a state update

End of cycle: screen and state **always agree**: one place to look after

- **Snapshot:** the `draft` you read inside a handler is the value of that render: the setter doesn't change the variable on the very next line
- Why draft and messages are separate: one is the **message still being typed**, the other is the **sent history** : different lifetimes
- Lab 5: add the "Typing: …" preview and a Clear button calling `setDraft("")`

Build the composer

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { useState } from "react";
2  import { MESSAGES } from "./messages.js";
3  import "./App.css";
4
5  // Lab 5: state React remembers
6  // MessageBubble is done (Lab 3 solved): the composer is yours
7  // TODO 1: state: const [draft, setDraft] = useState("");
8  // TODO 2: bind the input:
9  //   value={draft}
10 //   onChange={(e) => setDraft(e.target.value)}
11 // TODO 3: a Clear button next to the input:
12 //   <button onClick={() => setDraft("")}>Clear</button>
13 // TODO 4: make the preview follow: {draft || "…"}
14
15 function MessageBubble({ role, text }) {
16   return (
17     <li className={"bubble " + (role === "user" ? "me" : "them")}>
18       <span className="label">
19         {role === "user" ? "You" : "Assistant"}
20       </span>
21       <p className="text">{text}</p>
22     </li>
```

preview: `mini chat`

MINI CHAT · 3 MESSAGES

ASSISTANT
Hi! This is a mock chat

YOU
How is React different from plain JS?

ASSISTANT
We describe the screen from data

Type a message…

Typing: …

SUCCESS CRITERIA

Show solution | Check answers

- ○ typing updates the preview: input bound to state
- ○ Clear empties both the input and the preview
- ○ draft never leaks into the chat history

Complete the TODOs, then press Check answers

your move: add the `draft` state, bind `value` + `onChange`, add a `Clear` button, and make the preview follow (`{draft || "…"}`). A plain variable changes: state re-renders.

Input from outside, or memory of your own

PROPS the input, handed down

```
// the parent decides, the child renders
<MessageBubble role="user" text="How?" />

function MessageBubble({ role, text }) {
  return <p className="text">{text}</p>;
}
```

- comes **from the parent**: an input the component receives
- a **read-only snapshot**: the component never edits its props
- new props = **the parent re-rendered**: change arrives from above

STATE the memory, owned here

```
// the component remembers; nobody else can touch it
const [draft, setDraft] = useState("");

setDraft(e.target.value); // the only way in
```

- **born inside** the component: private memory (`useState`)
- **survives re-renders**, dies when the component unmounts
- only **the setter** changes it: asking React, never reassigning

`props ↓`
`state ←` → **same ending, different owner:** either changes → re-render → the UI follows. The design question is never "props or state?" but **who owns this data?**

Lab 5's lesson in one line: `draft` and `messages` were two states because they are two lifetimes — being typed vs already sent.

Predict: a sent message's text lives in `messages` in App and arrives in MessageBubble as … what? While typing, the same words live in `draft` — why two homes?

Send button and Enter are the same thing

```
the most-missed line: preventDefault
<form onSubmit={handleSubmit}>
  <input value={draft} onChange={...} />
  <button type="submit">Send</button>
</form>

function handleSubmit(event) {
  event.preventDefault();  // no page reload
  // ... send the message
}
```

In a plain browser: submitting a form = **reloading the whole page** (all state lost): React still has to tell the browser "no" with preventDefault

- **Enter in the input** and **clicking the submit button** fire the same event (`submit` on the form): write one handler, get both paths
- Events come from **one specific action**: code that responds to an action belongs in a handler, not in an Effect (more on that in the lifecycle section)
- The Lab 6 sending order: **preventDefault → trim → reject empty → update the array → clear the draft**

Predict: if you forget preventDefault and hit send: what happens to all state? And where is the half-finished conversation?

Create a new array: never edit the old one

the heart of Lab 6: the updater function

```
setMessages((previous) => [...previous, yours, reply]);
setDraft("");

// provided: unique ids for every message (even duplicates)
let nextId = messages.length + 1;
const newId = () => "m" + nextId++;
```

`[...previous, ...]` = a **new array holding all the old items + the new ones at the end**: the old array is untouched, and the mock reply appears immediately (no timers/async yet)

- React compares old/new arrays **by identity**: silently editing the old one may look like "nothing changed" to React
- `(previous) => ...` always receives the latest value: safer than reading `messages` directly (snapshot!)
- Sending the same text twice is fine, because every send gets a **fresh id**: keys never collide (predict: what if text were the key?)

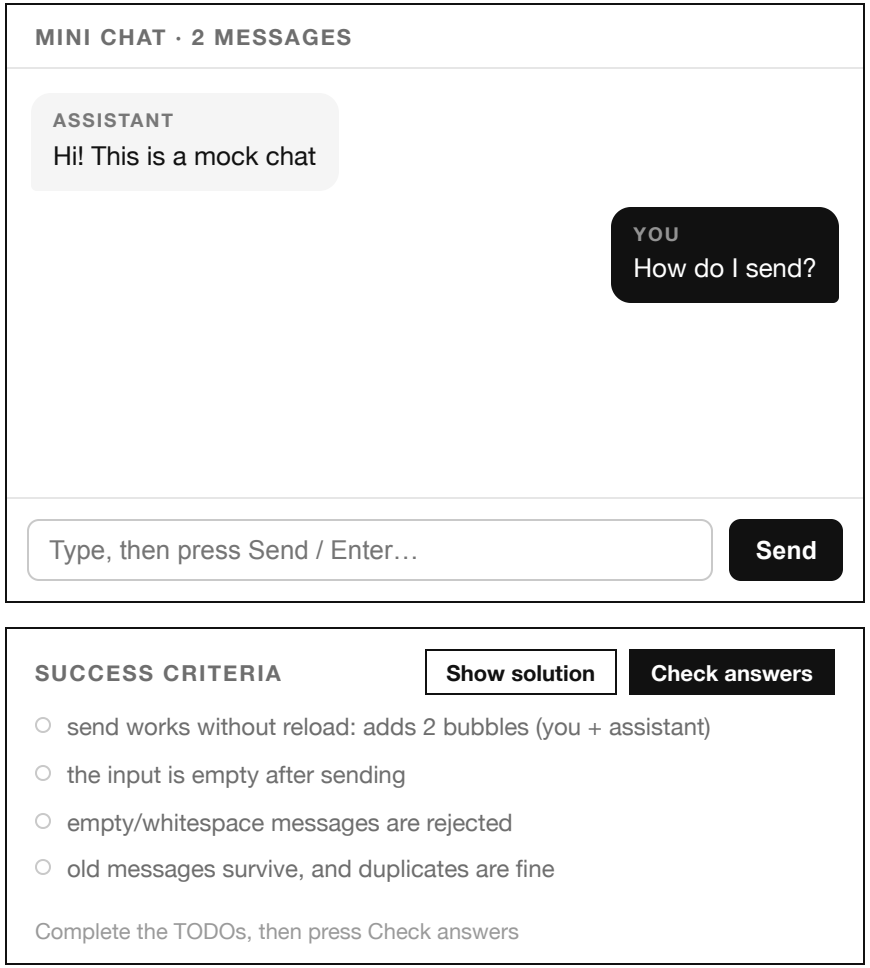
Take-away from the lab: sending is caused by the user's event: that work lives in an **event handler**, no Effect involved

Assemble: send + reply

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { useState } from "react";
2  import { MESSAGES } from "./messages.js";
3  import "./App.css";
4
5  // Lab 6: send a message, get a mock reply
6  // The composer works (Lab 5 solved): wire sending into the log
7
8  function MessageBubble({ role, text }) {
9    return (
10     <li className={"bubble " + (role === "user" ? "me" : "them")}>
11       <span className="label">
12         {role === "user" ? "You" : "Assistant"}
13       </span>
14       <p className="text">{text}</p>
15     </li>
16   );
17 }
18
19 // provided: unique ids for new messages + a mock reply (no real API yet)
20 let nextId = MESSAGES.length + 1;
21 const newId = () => "m" + nextId++;
22 const mockReply = (text) =>
```

preview: `mini chat`



your move: finish `handleSubmit`: prevent, trim, stop on empty, then `setMessages(p => [...p, yours, reply])` and clear the draft. Button and Enter share one `onSubmit`.

Render empty and disabled states

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { useState } from "react";
2  import { MESSAGES } from "./messages.js";
3  import "./App.css";
4
5  // Lab 7: empty and disabled states
6  // Sending works (Lab 6 solved): but this chat starts EMPTY, and the scree
7  // lies twice: a blank log says nothing (broken? loading?), and Send looks
8  // alive even with an empty input. Conditional rendering is how the UI
9  // follows the state it is derived from.
10 // TODO 1: disable Send unless the draft holds real text:
11 //   <button type="submit" disabled={!draft.trim()}>Send</button>
12 // TODO 2: when messages is empty, say so inside the <ul>:
13 //   {messages.length === 0 && (
14 //     <li className="empty">No messages yet</li>
15 //   )}
16
17 function MessageBubble({ role, text }) {
18   return (
19     <li className={"bubble " + (role === "user" ? "me" : "them")}>
20       <span className="label">
21         {role === "user" ? "You" : "Assistant"}
22       </span>
```

preview: `mini chat` (starts empty!)

MINI CHAT · 0 MESSAGES

Type, then press Send / Enter…

Send

SUCCESS CRITERIA

Show solution | Check answers

- ○ a fresh log speaks: the empty note shows
- ○ Send is honest about a blank input: disabled at the start
- ○ whitespace is not text: spaces keep Send disabled, a word wakes it
- ○ sending works, the note leaves, and Send locks again

Complete the TODOs, then press Check answers

your move: the blank log says nothing and Send looks alive on empty input — the screen lies. Fix both with `disabled={!draft.trim()}` and the empty state.

Mount, update, unmount

MOUNT

joins the UI tree

Opening the chat → `ChatRoom` appears for the first time: state is born, effects start running

UPDATE

re-computed when there's a reason

Typing a message / receiving new props: the **same** component re-renders, it isn't born again

UNMOUNT

leaves the UI tree

Closing the chat → `ChatRoom` is removed: state is gone, effects must clean up

Common mix-ups

an update is **not** an unmount followed by a mount: same component, different beat

hiding with CSS (`display:none`) is **not** unmounting: it's still in the tree, just invisible

```
{open && <ChatRoom roomId={roomId} />}

// open:   false → true  = mount
// typing / switching roomId = update
// open:   true → false = unmount
```

Predict first: in the next demo: does typing a message make the connection reconnect?

Predict, then press: watch the event log

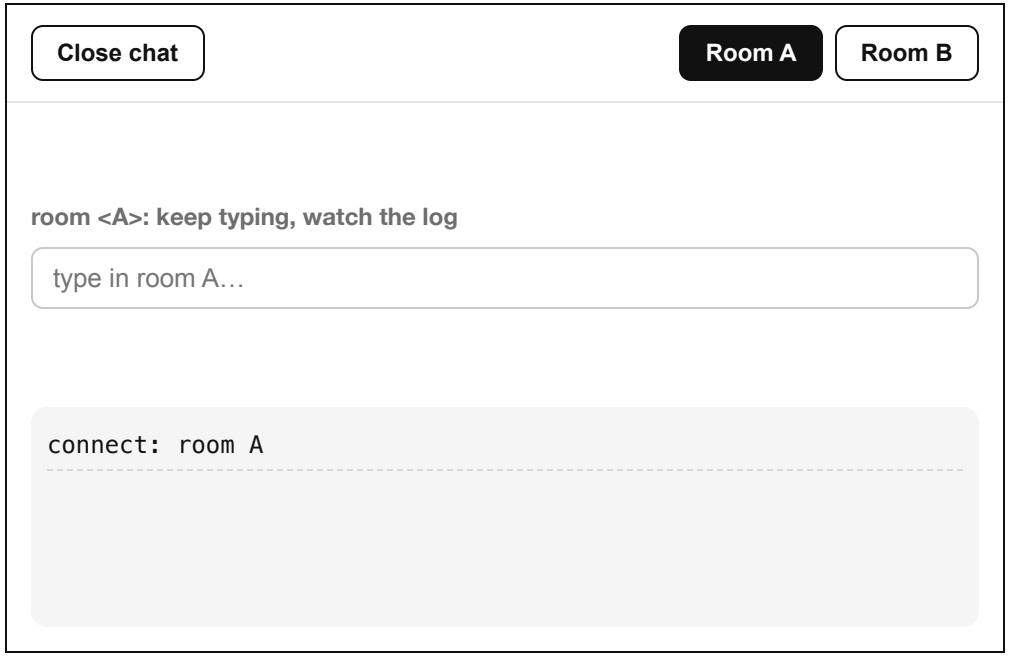
ChatRoom.jsx + App: complete, no TODOs

```jsx
// setLog is stable; roomId is the dependency that changes
useEffect(() => {
  const connection = createConnection(roomId, (line) =>
    setLog((previous) => [...previous, line])
  );
  connection.connect();
  return () => connection.disconnect();
}, [roomId, setLog]);
```

The Effect lists every reactive value it reads. React keeps `setLog` stable, so only a changed `roomId` restarts the connection · `createConnection` is a mock · the log lives in App, so cleanup on unmount stays visible

Prediction table: open room A → ? · type a message → ? · switch to B → ? · close the chat → ?: guess before pressing, then compare with the event log

demo: `ChatRoom` (real React)



Answers to check for: typing → a quiet update, no reconnect · switching rooms → cleanup of the old room, then setup of the new one (**no unmount**) · closing → unmount + cleanup of the last room

The Effect lifecycle

▶ Play | Next action | Reset | Open room A | Type a message | Switch to room B | Type again | Close the chat

COMPONENT	**mount**	update	update · same instance	update	**unmount**
EFFECT	setup · connect A	quiet	cleanup A → connect B	quiet	cleanup · disconnect B

LOG

```
connect: room A
disconnect: room A
connect: room B
disconnect: room B
```

Every action re-renders the **component**, but the **Effect** only restarts when its dependency, `roomId`, changes. Quiet steps are where typing does **not** touch the connection.

The rhythm of cleanup

runs before every new setup when a **dependency changes**: not just at unmount

at unmount: connections/timers get tidied away

Strict Mode (dev) may run an extra setup/cleanup cycle: don't read render counts from logs as real behavior

WHICH WORK GOES WHERE

render	computes the screen from props/state: no side effects
event handler	responds to one action, like hitting send: calling an API because the user sent belongs here
Effect	syncs with an external system after commit, like a connection, not for "reacting to a changed value"

Don't map hooks one-to-one onto class lifecycle methods: think in the Effect's setup/cleanup cycle instead

Your first Effect

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { useEffect, useRef, useState } from "react";
2  import { MESSAGES } from "./messages.js";
3  import "./App.css";
4
5  // Lab 8: follow the newest message
6  // The chat works (Lab 7 solved): but when the log overflows, the newest
7  // message lands below the fold. Where would the scroll code even live?
8  //   In render?  forbidden — components must stay pure.
9  //   In Send?    it only covers sends: a reply arrives on its own too.
10 // This is a job for an Effect: after the commit, sync a DOM node with sta
11 // Provided: logRef — a ref is React's door to one DOM node (more next les
12 // TODO 1: add the effect below the state: when messages change, scroll th
13 //   log to the bottom: logRef.current.scrollTo({ top: logRef.current.scrc
14 // TODO 2: pass the dependency array so it re-runs when messages changes -
15 //   and only then: typing must not yank the log back down
16
17 function MessageBubble({ role, text }) {
18   return (
19     <li className={"bubble " + (role === "user" ? "me" : "them")}>
20       <span className="label">
21         {role === "user" ? "You" : "Assistant"}
22       </span>
```

preview: `mini chat` (overflows on purpose)

MINI CHAT · 6 MESSAGES

ASSISTANT
Hi! This is a mock chat

YOU
The log got long while I was reading

ASSISTANT
New replies land below the fold

YOU
So somebody must scroll for me

Type, then press Send / Enter…

Send

SUCCESS CRITERIA

Show solution | Check answers

- ○ on load the log already shows the newest message
- ○ reading is safe: typing must not yank the log back down
- ○ sending scrolls the newest message into view

Complete the TODOs, then press Check answers

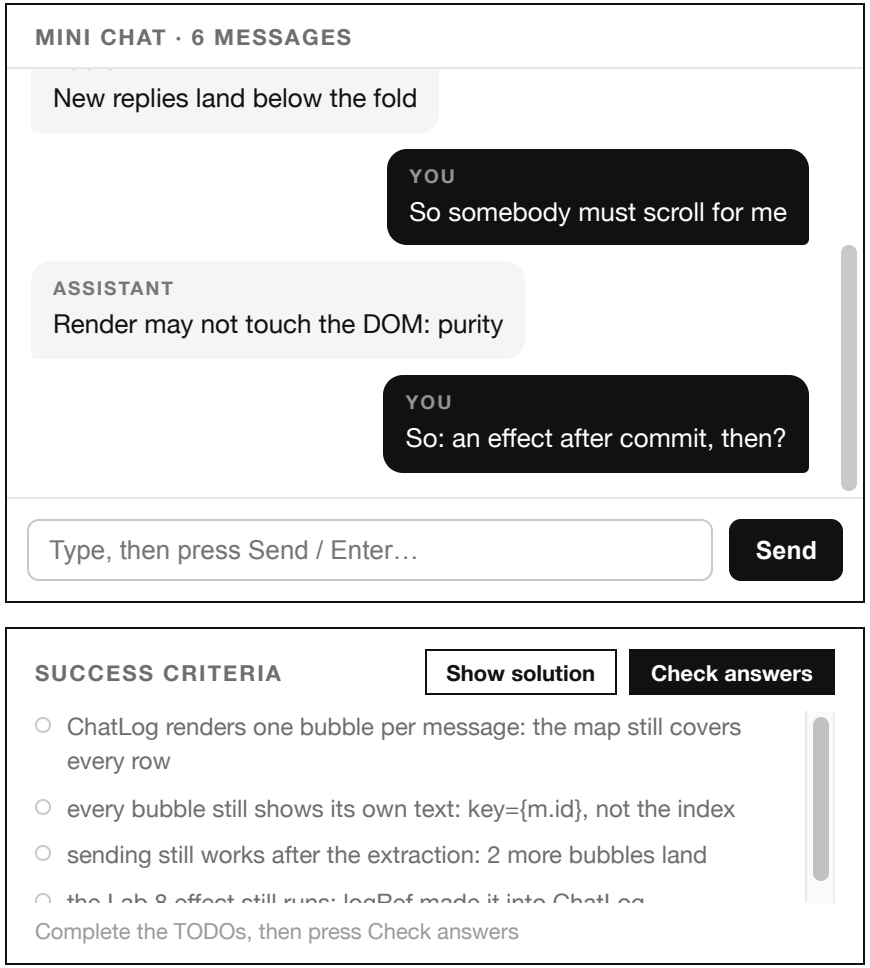
your move: render may not scroll (purity), Send only covers sends: write the `useEffect` with `[messages]` deps. Typing must not move the log.

Extract ChatLog: the map() from Lab 3, one more time

index.html | main.jsx | App.jsx | messages.js | index.css | App.css

```
1  import { useEffect, useRef, useState } from "react";
2  import { MESSAGES } from "./messages.js";
3  import "./App.css";
4
5  // Lab 9: pull the list out into its own component
6  // Everything below already works (Labs 1–8 solved) — except the log is
7  // still inline in App. TODO: extract it into ChatLog.
8  // TODO 1: write ChatLog({ messages, logRef }): a <ul className="log"
9  //   ref={logRef}> that maps over messages, one
10 //   <MessageBubble key={m.id} role={m.role} text={m.text} /> per item
11 //   (the exact pattern from Lab 3 — now living in its own component)
12 // TODO 2: replace the inline <ul> in App with
13 //   <ChatLog messages={messages} logRef={logRef} />
14
15 function MessageBubble({ role, text }) {
16   return (
17     <li className={"bubble " + (role === "user" ? "me" : "them")}>
18       <span className="label">
19         {role === "user" ? "You" : "Assistant"}
20       </span>
21       <p className="text">{text}</p>
22     </li>
```

preview: `mini chat` (overflows on purpose)



your move: write `ChatLog({ messages, logRef })` — same `map()` + `key` as Lab 3, now behind its own component. Stuck? press **Show solution**.

You've been rendering a list the entire time

chat.jsx: ChatLog, extracted in Lab 9

```jsx
function ChatLog({ messages, logRef }) {
  return (
    <ul className="log" ref={logRef}>
      {messages.map((m) => (
        <MessageBubble key={m.id} role={m.role} text={m.text} />
      ))}
    </ul>
  );
}
```

one component (`MessageBubble`, Lab 1–2), rendered once per row of **state** (Lab 5–6), re-run on every commit (Lab 7–8) — and in Lab 9 it moved into its own component without changing shape: a list render is where every topic today meets.

Predict: delete the `key` prop — the app still renders. What breaks later, and when would you actually notice?

- **props** gave the bubble its data per row — `{m.role}`, `{m.text}` (Lab 2)
- **state** is what `messages` actually is: a new array on every send, never edited in place (Lab 5–6)
- **events** are what grow the array — Send appends a row, the list re-renders with one more item (Lab 6)
- **effects** react to the list changing — auto-scroll runs because the array got longer, not because a user clicked (Lab 8)

the rule, one more time

n rows of data in → n elements out, each with a stable `key`. Everything else you learned today — props, state, events, effects — exists to decide what goes *into* that array.

EXIT TICKET · BEFORE YOU LEAVE

The screen is wrong: what do you open first?

THE MENTAL-MODEL CHECK · CLICK YOUR ANSWER

You open the running app and a bubble shows the **wrong text**. You get one first move:

Inspect the DOM | **Inspect state & props**

Pick one — then be able to say why.

THEN ANSWER IN WRITING · LMS FORM OR PAPER

Q1 What does React take off your hands, compared with vanilla DOM manipulation?

Q2 Props vs state: who owns each, and who is allowed to change each?

Q3 `setDraft("hi")` runs: what happens, in order, until the pixels change?

Q4 Why split a screen into components at all?

Q5 Which part of today is still the fuzziest for you?

Q6 Confidence building a component alone, 1–5:

1 | 2 | 3 | 4 | 5

1 = only with the slides open … 5 = from a blank file

Component = block · props = input · state = memory · event = trigger → **state change → render → new UI**

If this strip feels obvious, the lesson did its job — Next.js, Zustand and TanStack Query are all built on top of it.

Next

From mini chat to a real chatbot

What this lesson must leave behind: being able to say: **which part renders data · which part handles user actions (event handlers) · which part syncs with the outside world (Effects)** before adding AI and network complexity.

Next lesson: call a real backend from the submit handler with **loading, error and retry**, then layer in domain data, conversation history, and answer evaluation for the chatbot.

HOMEWORK

- Finish labs 1–8 with every check green
- Add a "clear history" button: which state does it touch?
- Sketch, no code: a room list joins the chat — draw the tree, then decide which component owns rooms, activeRoom, messages and draft, and why
- Read ahead: react.dev/learn/synchronizing-with-effects