# React Basics

## 🟣 أولًا: What is React?

### ✅ تعريف بسيط:

الReact هي **مكتبة JavaScript** (مش فريموورك) بتساعدنا نبني واجهات المستخدم (User Interfaces) بطريقة ذكية ومنظمة، وبتخلي تجربة البرمجة في الويب أسهل وأكتر كفاءة.

### ✅ مين عمل React؟

تم تطويرها من **فيسبوك** (Meta دلوقتي)، عشان يحلوا مشكلة الأداء وتنظيم الكود لما الموقع يكبر.

### ✅ ليه نستخدم React؟

لأنها بتوفر لنا:

1. ✅ **Component-Based**
   بتقسّم الصفحة لأجزاء (components) مستقلة وسهلة الإعادة والاستخدام.
2. ✅ **Virtual DOM**
   مش بتتعامل مع الصفحة مباشرة، لكن بتعمل نسخة وهمية منها وتحدّث اللي اتغير بس (وده أسرع بكتير React).
3. ✅ **Declarative**
   بدل ما تقول لصفحتك "اعمل كذا وكذا"، انت بتقول "أنا عايز الشكل النهائي يكون كده"، و React تهتم بالتنفيذ.

### ✅ مثال سريع:

بدل ما تكتب HTML وتعمل DOM يدويًا، React بتخلي ده أوتوماتيك.

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

### ✅ القصة من البداية:

### 📌 زمان (قبل React):

لما المستخدم يضغط على زر أو يطلب صفحة جديدة (مثلاً من رابط)، المتصفح كان:

1. يبعت **طلب جديد** (Request) للسيرفر.
2. السيرفر يرد بصفحة HTML جديدة بالكامل.
3. المتصفح يعمل **Refresh كامل للصفحة**.
4. المستخدم يحس إن الصفحة بتترعش أو بطيئة.

وده اسمه **Multi-Page App (MPA)**

## ✅ دلوقتي (مع React و SPA):

ال React بتشتغل في حاجة اسمها **Single Page Application (SPA)**

- يعني: المستخدم يدخل على صفحة واحدة بس.
- بعد كده أي تغيير أو انتقال بيتم **من غير ما الصفحة تتعملها Reload**.
- ال React بتحدث الأجزاء اللي اتغيرت بس من خلال **Virtual DOM**.

### 🧠 إزاي ده بيحصل؟

1. أول ما المستخدم يدخل الموقع، السيرفر بيبعت صفحة HTML بسيطة فيها JavaScript.
2. ال JavaScript يشغل React، وReact تبدأ ترسم الواجهة.
3. لما المستخدم يضغط أو يعمل تغيير، React:
   - **ما تبعتش طلب جديد للسيرفر لصفحة HTML.**
   - **تحدث الجزء المطلوب من الصفحة فقط.**

## ✅ العلاقة بين Client و Server:

| الطرف | الدور |
|---|---|
| 💻 Client (المتصفح) | يعرض الصفحة، يتعامل مع React، يبعت طلبات (API Requests) للبيانات |
| ☁ Server (الخادم) | يرد على الطلبات بالبيانات فقط (مش الصفحة كلها)، وغالبًا بيبقى Backend زي Node.js أو PHP |

### 📌 في SPA:

- الواجهة بتتعامل مع السيرفر من خلال **API** (يعني طلبات JSON).
- مثال:
  - ال React يطلب من السيرفر: "هاتلي بيانات المستخدم".
  - السيرفر يرد بـ `{ name: "Ayaat", age: 22 }` :JSON
  - ال React تعرضها جوا الصفحة.

## ✅ مثال عملي على الفرق:

### 🟥 زمان ده كان (MPA):

- تضغط على "الصفحة التالية" ← الصفحة كلها تعيد التحميل.

### 🟩 دلوقتي مع React (SPA):

- تضغط على "الصفحة التالية" ← جزء معين يتغير بس في نفس الصفحة (من غير Refresh).

### 🧠 ليه ده مهم جدًا؟

لأن ده بيخلي الموقع:

- أسرع.
- أكثر سلاسة.
- شكله احترافي.

## 🟣 الفرق بين SPA و MPA

| المقارنة | SPA (Single Page Application) | MPA (Multi Page Application) |
|---|---|---|
| ✅ المعنى | تطبيق صفحة واحدة | تطبيق يحتوي على صفحات كثيرة |
| ✅ طريقة العمل | المستخدم يفتح صفحة واحدة فقط، وكل التنقلات داخلها تتم بدون Reload | كل صفحة لها ملف HTML منفصل، وكل ما تنتقلي يحصل Reload |
| ✅ الاستخدام | React, Vue, Angular | مواقع قديمة، PHP, WordPress |
| ✅ الأداء | أسرع بعد التحميل الأول (لأن التنقل داخلي) | أبطأ شوية لأن كل مرة الصفحة كلها بتتجدد |
| ✅ البيانات | البيانات بتتحمل من API (JSON) | السيرفر يبعت صفحة HTML كاملة في كل مرة |
| ✅ تجربة المستخدم | سلسة، بدون اهتزاز أو انتظار طويل | ممكن المستخدم يحس بتقطيع أو تأخير عند التنقل |
| ✅ مثال | Gmail, Facebook, Twitter (بعد ما بقوا SPA) | مواقع الجرائد، أو المنتديات القديمة |

## 🎯 مثال واقعي:

### 1. 🟥 MPA:

تخيل إنك بتروح لمكتب حكومي، وكل خدمة في مكتب مختلف:

- عايز تغير الاسم → تروح مكتب 1.
- عايز تجيب شهادة → تروح مكتب 2.

كل مرة تخرج من باب وتدخل باب جديد (Reload كامل).

### 2. 🟩 SPA:

نفس المكتب، لكن:

- الموظف نفسه يغيرلك الخدمة جوه نفس المكان.
- تغير الاسم، وتاخد شهادة، وتستلم البطاقة وكل ده وانت في نفس المكان.

(يعني React تحدث الواجهة بس، وانت مش حاسس إنك "غيرت صفحة").

## 🧠 ليه React تستخدم SPA؟

لأن:

- المستخدم ما يحسش بتقطيع.
- الأداء يكون أسرع.
- الشيفرة تكون أنظف وأسهل في التحكم.

## 🟣 يعني إيه Virtual DOM؟

### ✅ أولًا: يعني إيه DOM أصلاً؟

DOM = Document Object Model

يعني: **نموذج لشجرة الصفحة**، React أو JavaScript بيشوفوا صفحة HTML على إنها شجرة من العناصر (div, p, h1...)

مثلاً:

```
<body>
  <h1>Hello</h1>
  <button>Click</button>
</body>
```

دي عند React شجرة:

- body
  - h1
  - button

## 🧠 طيب فين المشكلة؟

### المشكلة 👇

كل مرة تغير عنصر في الصفحة، زي مثلًا تغيّر نص أو لون أو عدد، المتصفح لازم:

1. يروح يدور على العنصر ده في الشجرة الحقيقية.
2. يغيره.
3. يعيد ترتيب أو رسم باقي الصفحة أحيانًا.

وده بطيييء جدًا لما الصفحة تكبر (زي فيسبوك أو تويتر).

## 💡 هنا ييجي دور Virtual DOM

### ✅ ال Virtual DOM = نسخة وهمية من DOM، موجودة في الذاكرة بس.

ال React بتستخدمها كالتالي:

1. أول مرة بتعرض الصفحة → React تنشئ نسخة من الشجرة (Virtual DOM).
2. لما يحصل تغيير (زي ما المستخدم يضغط زر)
   - ال React **بتعمل نسخة جديدة من Virtual DOM بعد التغيير**.
   - وتقارن النسخة القديمة والجديدة سطر بسطر.
   - وتشوف بالضبط إيه اللي اتغير.
3. بعد المقارنة → React تروح تغير **الجزء ده بس** في الشجرة الحقيقية (Real DOM).

## 🎯 مثال واقعي:

تخيّل عندك ورقة فيها 1000 سطر، وعدّلت كلمة واحدة في سطر رقم 500.

- **بدون Virtual DOM**: هتطبع الورقة كلها من جديد.
- **مع Virtual DOM**: هتقارن النسخة الجديدة والقديمة، وتطبعي السطر 500 بس.

وده معناه:
✅ **أداء أسرع**
✅ **موقع أسرع**
✅ **تجربة مستخدم أذكى**

## ✅ مثال عملي في React:

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

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

كل مرة تضغط الزر:

- ال React **مش بتعيد بناء الصفحة كلها**.
- بتقارن الـ Virtual DOM القديم والجديد.
- وتغير الرقم بس في الزر.

# 🟣 What are Components in React?

## ✅ التعريف:

ال **Component = جزء مستقل من واجهة المستخدم (UI)**
يعني زي قطعة من الصفحة، ممكن نعيد استخدامها في أماكن مختلفة.

📦 تخيل الـ Component زي "علبة" فيها:

- شكل (HTML)
- منطق (JavaScript)

- وتقدر تتغير أو تتفاعل (state أو props)

## 🧠 ليه بنستخدم Components؟

1. ✅ **إعادة الاستخدام (Reusability)**
بدل ما نكتب نفس الكود 10 مرات، نكتبه مرة واحدة بس ونستدعيه في كذا مكان.
2. ✅ **تنظيم الكود (Separation)**
كل حاجة ليها Component خاص بيها. مثلًا: زر – نموذج – شريط علوي...
3. ✅ **سهولة التطوير**
نقدر نختبر وتطور كل جزء لوحده.

## 🏗️ تشبيه بسيط:

تخيل بتبني موقع = زي ما تبني بيت:

- **البيت = الصفحة**
- **الباب، الشباك، المطبخ = Components**
- الباب تقدر تركبيه في أكتر من أوضة ➤ **Reusable**

## 🔷 أنواع Components:

### 1. Function Component (الأكثر استخدامًا)

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

### 2. Class Component (قديم شوية، نادر الاستخدام دلوقتي)

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

💡 إحنا بنستخدم دلوقتي **Function Components** مع **Hooks**.

## ✅ مثال واقعي:

### تخيل عندك صفحة موقع متجر:

- ✅ Navbar → Component
- ✅ ProductCard → Component

- ✅ Button → Component
- ✅ Footer → Component

```
function App() {
  return (
    <>
      <Navbar />
      <ProductCard />
      <Footer />
    </>
  );
}
```

## 🔁 مميزات Components:

| الميزة | الشرح |
|---|---|
| Reusable ♻️ | ممكن أستدعي نفس الـ Component في أماكن مختلفة |
| Composable 🧷 | أركّب Components داخل بعض |
| ذكية 🧠 | تقدر تاخد بيانات وتتفاعل مع المستخدم |

# Imperative code V.S Declarative code

## ✅ أولًا: يعني إيه Imperative و Declarative؟

### ببساطة:

| النمط | المعنى |
|---|---|
| **Imperative** | بتقول للكمبيوتر *إزاي يعمل الحاجة خطوة خطوة* |
| **Declarative** | بتقول للكمبيوتر *عايزة النتيجة تبقى إيه*، وهو يشوف إزاي يعملها |

## 🧱 تشبيه بسيط:

تخيل عايز تعمل بيت:

- 🟥 **Imperative:**
  تمسك الطوب بإيدك وتحطه بنفسك.
  (بتقول الخطوات بالتفصيل: حط طوبة هنا، بعديها طوبة تانية...)
- 🟩 **Declarative:**
  تجيب مقاول وتقوله: "أنا عايز بيت فيه ٣ أوض، مطبخ، وصالة"

وهو يشوف يركّبهم إزاي.

## 💡 طيب، هل JavaScript Imperative؟

### ✅ الإجابة:

**نعم، JavaScript هي لغة Imperative بشكل عام.**

يعني لما تكتب جافاسكريبت، بتكتب خطوات واضحة للكمبيوتر يعمل إيه.

### ✅ مثال على الكتابة بأسلوب Imperative:

```
let total = 0;
for (let i = 0; i < 5; i++) {
  total += i;
}
console.log(total); // 0 + 1 + 2 + 3 + 4 = 10
```

أنتي هنا كتبت الخطوات:

- ابدأ من i = 0
- زوّد i لحد ما توصل 5
- اجمع كل القيم في total

### ✅ طيب هل جافاسكريبت فيها Declarative كمان؟

أيوه، **جوا جافاسكريبت** ممكن تكتب بعض الأكواد بأسلوب Declarative
مثلاً لما تستخدم `map()` أو `filter()` أو `forEach()`، دي style أقرب لـ declarative.

### مثال:

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8]
```

أنت هنا ما قلتش للكمبيوتر "لف على العناصر و اضربهم فـ 2"، أنت بس قلت:
**"أنا عايز نسخة جديدة من الأرقام مضروبة ×2"**

وده اسلوب declarative داخل جافاسكريبت.

## 🎯 طيب React بقى؟ هي Imperative ولا Declarative؟

### ✅ال React مكتوبة بجافاسكريبت، لكن هي Declarative

- ال React بتخليك تقول: "أنا عايز الصفحة تبقى شكلها كده"

- وهي (React) تتولى إدارة الـ DOM والتحديثات.

```
function App() {
  function handleClick() {
    alert('Button clicked!');
  }

  return (
    <button onClick={handleClick}>Click me</button>
  );
}
```

هنا: إنت بتقول "أنا عايز الزرار لما أضغط عليه يطلعلي alert"، وسيب React تتصرف.

# create react project

ال browser بيفهم 3 لغات فقط وهم HTML, CSS, JS فا لو كنت بتستخدم اي لغة تانية هايتم ترجمة اللغة دي ال 3 لغات اللي المتصفح بيفهموا بستخدام Code Builder .

في ريأكت المتصفح ميقدرش يفهم ال jsx و عشان كده بيتم ترجمته باستخدام Code Builder مثل / webpack , vite ... etc

بمإن الريأكت هي مكتبة ومن الممكن تحتاج تستخدم مكتبات تانية معاها فا في package manager تسمي NPM بنستخدمها لتحميل المكتبات .

ال Node.js هيRuntime evironment "بيئة تشغيل" تحتوي على npm , و بتخليك تكتب كود يعمل على server side .

## ✅ أولاً: متطلبات قبل إنشاء المشروع

لازم يكون عندك على الجهاز:

- ✅ Node.js (تلقائيًا npm بنثبّت معاه)
- ✅ terminal شغّال (زي PowerShell أو Command Prompt أو Git Bash)

تتأكد إنهم متثبتين كده:

```
node -v
npm -v
```

لو طلعلك رقم الإصدار يبقى تمام 💪

## ✅ ثانيًا: إنشاء مشروع React باستخدام Vite أو CRA

### الطريقة الأولى (المفضلة الآن): باستخدام Vite

أسرع وأخفّ من create-react-app

```
npm create vite@latest my-react-app -- --template react
```

- ال `my-react-app` : اسم المشروع، ممكن تغيره.
- ال `--template react` : معناه مشروع React عادي (ممكن تعمله بـ TypeScript كده:
- `--template react-ts` )

بعد كده:

```
cd my-react-app
npm install
npm run dev
```

هتفتحلك صفحة على المتصفح غالبًا على:

`http://localhost:5173`

## ✅ ثانيا: تركيب شكل المشروع بعد إنشاؤه

في ملفات Vite مثلاً هتلاقي:

```
my-react-app/
├─ index.html
├─ package.json
├─ vite.config.js
└─ src/
    ├─ main.jsx
    ├─ App.jsx
    └─ assets/
```

- `main.jsx` : نقطة دخول التطبيق.
- `App.jsx` : المكون الرئيسي اللي بيبدأ منه كل حاجة.
- `index.html` : الصفحة الوحيدة في الـ SPA وبيتم حقن الـ React app فيها.

## ✅ مثال بسيط داخل App.jsx:

```
function App() {
  return (
    <div>
      <h1>Hello Ayaat 👋</h1>
      <p>Welcome to your first React project!</p>
    </div>
```

```
  );
}

export default App;
```

## 🔷 What is JSX?

**JSX** اختصار لـ **JavaScript XML**
وهو عبارة عن **امتداد** للغة JavaScript يسمح لك بكتابة **HTML داخل JavaScript**.

يعني بدل ما تكتب كود HTML في ملف لوحده، وكود JavaScript في ملف تاني، JSX بيخليك تكتب الاتنين مع بعض بطريقة تشبه HTML داخل كود React.

### ✅ مثال بسيط:

```
const element = <h1>Hello, Ayaat!</h1>;
```

السطر اللي فوق ده شكله HTML، لكن هو في الحقيقة JavaScript!
لما React يترجم الكود ده، بيحوّله لحاجة اسمها **React.createElement()**.

### 🎯 طيب ليه بنستخدم JSX؟

- **علشان شكله مألوف** زي HTML، فبيكون سهل نكتبه ونقرأه.
- **بيدمج بين منطق الواجهة (UI logic) والتصميم** في نفس المكان.
- بيساعد React تفهم إيه اللي المفروض يتعرض على الشاشة.

### ✅ مثال عملي في React:

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

لو نديتيه في الصفحة:

```
<Welcome /> // Welcome, Ayaat!
```

### 💡 ملاحظات مهمة عن JSX:

1. **لازم دايمًا ترجع Element واحد بس** من الدالة:
   يعني لو عايز ترجع أكثر من عنصر، حوّطهم بـ `<div>` أو Fragment (<>):

   ✅ صحيح:

```
    return (
  <div>
    <h1>Hello</h1>
    <p>How are you?</p>
  </div>
);
```

2. **الكلمات المحجوزة في JavaScript بتتكتب بشكل مختلف:**

- `class` → `className`
- `for` → `htmlFor`

```
<div className="box">Content</div>
```

3. **تقدر تكتب JavaScript داخل JSX** بين `{}` :

```
const name = "Ayaat";
return <h1>Hello, {name}</h1>;
```

### 🤔 هل JSX إجباري؟

لا، تقدرِ تكتب React من غير JSX، بس بيكون صعب ومش واضح. شوف الفرق:

🔴 بدون JSX:

```
const element = React.createElement('h1', null, 'Hello, Ayaat');
```

✅ بـ JSX:

```
const element = <h1>Hello, Ayaat</h1>;
```

# How React Work?

### 🔷 أولاً: الفكرة العامة

ال React بتخلينا نقدر نعمل صفحات تفاعلية بسهولة من غير ما نحتاج نحدث الصفحة كلها كل شوية.
اللي بيخلي ده ممكن هو إن React بتعتمد على:

- **Virtual DOM**
- **Components**
- **One-way data flow**

## 🔶 خطوات عمل React تحت الغطاء:

### 1. إنت بتكتب JSX

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

ده مش HTML حقيقي، هو **JSX** – شبيه بالـ HTML لكنه بيتحول لجافاسكريبت.

### 2. الReact بتحوّله لـ JavaScript باستخدام Babel

```
React.createElement("h1", null, "Hello Ayaat!");
```

### 3. الReact تبني Virtual DOM

- ال Virtual DOM هو نسخة في الذاكرة من الصفحة.
- الReact بتحفظ نسخة من الـ DOM وبتقارنها بأي تغييرات بتحصل بعد كده.

### 4. لما البيانات تتغير (State/Props):

- ال React بتقارن بين النسخة القديمة والجديدة من الـ Virtual DOM.
- بتعرف إيه اللي اتغير بالظبط (مش بتشيّك كل حاجة).

### 5. ال React بتعمل تحديث حقيقي (Real DOM) فقط للجزء اللي اتغير

يعني بدل ما تعمل refresh للصفحة كلها، بتغيّر بس العنصر اللي محتاج يتغيّر.

🟢 النتيجة: أداء أعلى + تجربة مستخدم أفضل + كود منظم.

### 🧠 مثال واقعي:

لو عندك **عداد** زي كده:

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

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Add</button>
    </div>
```

```
  );
}
```

- أول مرة بيظهر `count = 0`
- لما تضغط الزر:
  - ال React تعمل setState
  - ال Virtual DOM يتحرك يقارن العدد الجديد بالقديم
  - يلاقي إن `<h1>` اتغير بس
  - يحدث الـ `<h1>` فقط في الصفحة

**كده وضحت ليه React قوية !**

# Build custom component

## ✅ أولًا: يعني إيه "custom component"؟

كلمة "custom" معناها مكوّن خاص بيك، مش جاهز من React.
ال React بيتيحلك تعمل Components بنفسك، زي مثلًا:

- `Navbar`
- `Footer`
- `ProductCard`
- `Button`

كل واحد منهم عبارة عن function أو class (بس إحنا حاليًا بنشتغل بالفنكشن).

## ✅ مثال عملي: بناء Component لبطاقة منتج

### 1. نفتح ملف جديد داخل `src` ، وليكن `ProductCard.js` :

```
function ProductCard() {
  return (
    <div>
      <h3>اسم المنتج: قميص رجالي</h3>
      <p>السعر: 250 جنيه</p>
    </div>
  );
}

export default ProductCard;
```

### 2. نروح على `App.js` ونعرضه:

```
import ProductCard from "./ProductCard";

function App() {
  return (
    <div>
      <h1>أهلاً بيك في المتجر</h1>
      <ProductCard />
    </div>
  );
}

export default App;
```

## ✅ أولًا: إزاي نضيف ستايل لـ component؟

في React فيه ٣ طرق رئيسية نقدر نعمل بيها **styling**:

## 🟢 1. Inline Styling

يعني تكتب الـ CSS جوه الـ component نفسه كـ object.

```
function Welcome() {
  const style = {
    color: 'blue',
    fontSize: '20px',
    backgroundColor: 'lightgray'
  };

  return <h1 style={style}>Welcome, Ayaat!</h1>;
}
```

### ✅ ليه تستخدمه؟

- لما يكون الستايل بسيط أو ديناميكي حسب شرط معين.

### ⚠️ ملحوظة:

- لازم تكتب أسماء الخصائص بنظام **camelCase**:
  مثلًا: `backgroundColor` بدل `background-color`.

## 🟢 2. CSS File عادي (External CSS)

يعني تكتب CSS في ملف خارجي وتستدعيه.

### 📌 خطوات:

1. تعمل ملف `App.css` مثلًا:

```
import './App.css';

function Welcome() {
  return <h1 className="title">Welcome, Ayaat!</h1>;
}
```

### ✅ ليه تستخدمه؟

- لو عندك أكتر من عنصر بيشارك نفس الستايل.
- لو بتحب تفصل بين الكود والستايل.

## 🟢 3. CSS Modules

لو عايز تعمل **ستايل خاص لكل component** ومايأثرش على باقي الصفحة.

### 📌 خطوات:

1. تعملي ملف CSS بالاسم:
`Welcome.module.css`

```
import styles from './Welcome.module.css';

function Welcome() {
  return <h1 className={styles.title}>Hello from Module!</h1>;
}
```

### ✅ الميزة:

- كل class name بيبقى ليه اسم خاص (unique)، فمفيش تعارض بين المكونات.

# Binding data to JSX

في React، **ربط البيانات بـ JSX (Data Binding)** يعني إنك بتستخدم متغيرات أو قيم ديناميكية جوا الكود اللي بيرجع JSX (يعني اللي بيتعرض على الشاشة). وده بيديلك المرونة إنك تعرض بيانات جاية من الـ state أو الـ props أو حتى متغيرات عادية في مكونك.

### 📌 مثال بسيط لفهم ربط البيانات (Data Binding) في JSX:

```
function Welcome() {
  const name = "Ayaat";
```

```
  const age = 23;

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>You are {age} years old.</p>
    </div>
  );
}
```

## ✅ شرح:

- `{name}` و `{age}` دول مثال على Data Binding.
- القوسين `{}` معناهم "أنا دلوقتي عايز أستخدم JavaScript جوا JSX".
- يعني تقدر تكتب جوه الأقواس:
  - متغير ( `name` )
  - عملية حسابية ( `age + 5` )
  - دالة ( `getGreeting()` )
  - أو أي تعبير (expression).

## 🎯 أمثلة أكتر لتثبيت المعلومة:

### 1. ربط دالة بالـ JSX:

```
function getGreeting(name) {
  return `Hello, ${name}`;
}

function App() {
  return <h1>{getGreeting("Ayaat")}</h1>;
}
```

### 2. ربط عملية حسابية:

```
const price = 100;
const discount = 20;

function App() {
  return <p>Final price: {price - discount} EGP</p>;
}
```

## ⚠️ ملاحظات مهمة:

- لا تستخدم الجمل الكاملة (statements) جوا `{}` في JSX، زي `while` , `for` , `if` ... دول مينفعوش.
- لازم التعبير يكون expression يعني يرجع قيمة.

مثال خطأ:

```
{/* خطأ */}
// {if(true) { return "Hi" }}
```

## ✅ فايدة ربط البيانات في JSX:

- **الديناميكية**: بدل ما تكتب الكلام بنفسك، React تقدر تغيره حسب الحالة أو البيانات.
- **التفاعل**: لما الـ state أو الـ props تتغير، React بيعمل re-render ويحدث البيانات تلقائيًا.

# Pass data to components

لما تحب **تمر بيانات (Data) من مكون (Component) لمكون آخر** في React، غالبًا بيكون السيناريو كالتالي:

## ✅ من الأب إلى الابن (Parent → Child)

دي الطريقة الأساسية اللي React بتشتغل بيها، وهي إن المكون الأب بيبعت بيانات للابن عن طريق الـ **props**.

كلمة **Props** في React هي اختصار لـ **"Properties"**، وبتُستخدم عشان تمرر **بيانات من المكون الأب (Parent Component) إلى المكون الابن (Child Component)**.

## ✅ ليه بنستخدم props؟

لأن React مبنية على فكرة **المكونات (Components)**، وكل مكون بيكون مستقل بذاته، ولو عايزين مكون يعرف بيانات من مكون تاني، بنمررها له عن طريق **props**.

## 🎯 حاجات مهمة عن props:

- **Read-only**: يعني ماينفعش تغير قيمتها جوه المكون الابن.
- **مش object عادي**: هو شبه object، لكن React بتتعامل معاه بذكاء، وكل مرة تتغير فيه البيانات، بتعمل re-render.
- **تُستخدم لعمل مكونات مرنة وقابلة لإعادة الاستخدام.**

## 🎯 الفكرة ببساطة:

1. الأب يعرف بيانات.
2. يمرر البيانات كمُلكية (prop) للابن.
3. الابن يستقبل البيانات من `props` .

## 📌 مثال عملي وواقعي:

لو بتبني **مكون لبطاقة مستخدم (User Card)**، وعاوز تستخدمه لناس كتير:

```
// child componet
function UserCard(props) {
  return (
    <div>
      <h2>{props.username}</h2>
      <p>{props.email}</p>
    </div>
  );
}
// parent componet
// استخدام المكون:
function App() {
  return (
    <div>
      <UserCard username="Ayaat" email="ayaat@email.com" />
      <UserCard username="Suzan" email="suzan@email.com" />
    </div>
  );
}
```

### 💡ملحوظات مهمة:

- لازم دايمًا البيانات تمشي من فوق لتحت (من parent ل child).
- الـ **read-only** props يعني الطفل ماينفعش يغير القيمة.
- لو عايز الطفل يبعت حاجة للأب → لازم تمرر له **دالة من الأب** ينادي عليها. (هانفهم الخطوة دي قدام!)

### 🧠 فـ ليه نستخدم props؟

لأن دي الطريقة الوحيدة اللي بنخلي بيها مكونات React تتكلم مع بعضها — الأب يعرف كل حاجة، والابن يستقبل ويعرض.

# Adding js code to component

يعني إيه "adding JS code to component" في React؟
المقصود بيها هو **كتابة كود JavaScript داخل مكون** (Component) **React** علشان نستخدمه في:

1. **تنفيذ عمليات منطقية** (Logic).
2. **تخزين بيانات مؤقتة** (variables/constants).
3. **إجراء حسابات** أو **جلب بيانات**.
4. **التفاعل مع البيانات اللي جاية من الـ props**.

### 💡فين نكتب الكود ده؟

✅ داخل الـ Function (المكون)

```
function Welcome(props) {
  const name = props.name; // JavaScript variable
  const greeting = `Hello, ${name}!`; // JS code to create string

  return <h1>{greeting}</h1>; // JSX with embedded JS
}
```

### 💡فين كمان ممكن نكتب JavaScript جوه JSX؟

عن طريق الأقواس `{}`
يعني تقدر تحط أي كود JS جوه الـ JSX بين `{}`

```
function AgeChecker(props) {
  return <p>{props.age >= 18 ? "Adult" : "Child"}</p>;
}
```

ده اسمه **expression** – يعني سطر بيرجع قيمة.

مثال عملي:

```
function Temperature(props) {
  const temp = props.value;
  const status = temp > 30 ? "Hot" : "Cold";

  return (
    <div>
      <p>Temperature: {temp}°C</p>
      <p>Status: {status}</p>
    </div>
  );
}
```

### 📌 خلاصة:

- بنكتب JavaScript جوه المكون (قبل `return` ) أو جوه `{}` داخل JSX.
- ينفع تكتب `objects` , `arrays` , `functions` , `if` , `let` , `const` ... إلخ.
- اللي جوه `{}` لازم يكون expression بترجع قيمة، مش statement زي `if` أو `for` .

## component splitting

ال**Component Splitting** في React معناها إننا بنقسم الكود بتاعنا لمكونات صغيرة (components) بدل ما نكتب كل حاجة في ملف واحد كبير. ده بيخلّي المشروع منظم وسهل الصيانة وإعادة الاستخدام.

## 💡 ليه نعمل Component Splitting؟

تخيل إنك بتعمل موقع كبير فيه صفحات مختلفة (رئيسية، تواصل، خدمات...)، وكل صفحة فيها أجزاء زي:

- Header
- Navbar
- Main content
- Footer

لو كتبت كل ده في ملف واحد (زي `App.js` ) الكود هيبقى معقد جدًا وصعب تتبع الأخطاء أو تعمل تحديث على جزء معين. لكن لما تقسميهم لـ Components، تقدر:

- تكتب كود نظيف ومنظم
- تعيد استخدام أجزاء من الموقع (زي الـ Header مثلاً)
- تعمل Testing لكل جزء لوحده

## 🧩 مثال بسيط على التقسيم:

### 🔷 ملف `App.js` :

```
import Header from './Header';
import Footer from './Footer';
import Content from './Content';

function App() {
  return (
    <div>
      <Header />
      <Content />
      <Footer />
    </div>
  );
}

export default App;
```

## ✨ نصيحة:

ابدئي دايمًا بتقسيم أي مشروع React حسب أجزاء الصفحة: كل جزء يعبر عن component مستقل. ومع الوقت هتتعلم إزاي تعمل تنظيم ملفات أقوى (زي folders لكل جزء).

## 🌟 يعني إيه `children prop` ؟

في React، **كل حاجة بنكتبها جوا وسم (Tag) للكمبوننت، React بتبعتها كـ `children`** .

يعني لو عندك Component بالشكل ده:

```
<MyBox>
  <p>Hello</p>
  <button>Click me</button>
</MyBox>
```

اللي جوا `<MyBox>` (يعني `<p>` و `<button>` ) بيتبعت للـ MyBox component كـ `props.children` .

## ✅ إمتى نحتاج `children` ؟

لما نكون عايزين نخلي الـ Component عبارة عن **صندوق يحوي محتوى ممكن يتغير حسب الاستخدام**.

## 🎯 مثال عملي

تعالي نعمل Component اسمه `Card` ، وده وظيفته يعرض المحتوى اللي جوّاه بشكل منسّق:

```
// Card.js
function Card(props) {
  return (
    <div style={{ border: '1px solid gray', padding: '16px', borderRadius: '8px'
}}>
      {props.children}
    </div>
  );
}
```

وبعدين نستخدمه كده :

```
// App.js
import Card from './Card';

function App() {
  return (
    <div>
      <Card>
        <h2>Title</h2>
        <p>This is the content inside the card.</p>
      </Card>

      <Card>
        <button>Click me</button>
      </Card>
```

```
    </div>
  );
}
```

## 🧠 الفكرة ببساطة:

- `Card` مش عارفة إيه المحتوى اللي جواها.
  - أي محتوى تحطيه بين الوسمين `<Card> ... </Card>`، بيتبعت كـ `props.children`.
  - كده نقدر نعيد استخدام Card في كل مكان بطريقة مرنة وجميلة.

## 📌 ليه `children` مفيد؟

- يخليك تبني **layout** عام reusable.
- يوفر عليك تكرار الكود.
- يعطيك تحكم أكتر في شكل الصفحة بدون ما تعيد كتابة نفس التصميم.

### ال **props.children** :

هي خاصية تستخدم لتمرير عناصر الطفل (child elements) الى ال component تعتبر props.children احد ال props المدمجة و المميزة في React.
يتم استخدامها عندما تحتاج الى امكانية ادراج عناصر داخل component دون الحاجة الى تحديدها بشكل صريح في الجزء العلوي من الكود .

*تقدر تكتب الكمبونت في React بأكثر من شكل باستخدام أنواع مختلفة من **الدوال**، وكل شكل ليه استخداماته، ودي الأنواع:*

1. ✅ **Arrow Function** (الأكثر شيوعًا الآن)
2. ✅ **Regular Function Declaration**
3. ✅ **Function Expression**

## ✅ يعني إيه Listening to Events؟

يعني "أسمع أو أراقب الأحداث" اللي بتحصل في الصفحة، زي لما المستخدم:

- يضغط على زر (Button)
- يكتب حاجة في input
- يعمل hover أو يخرج بالماوس
- يرفع ملف... وهكذا.

ودي بنسميها **events**.

## ✅ طب في React أتعامل إزاي مع الأحداث؟

بتضيف **event handler** (يعني دالة بتتنفذ لما يحصل الحدث) عن طريق JSX كده:

```
function MyButton() {
  const handleClick = () => {
    alert('Ayaat clicked the button!');
  };

  return <button onClick={handleClick}>Click me</button>;
}
```

### ✅ ملاحظات مهمة:

1. اسم الحدث بيبدأ بـ `on` زي: `onSubmit` , `onChange` , `onClick` .
2. لازم تكتبيه بـ **CamelCase** (يعني أول حرف صغير والباقي كل كلمة أولها حرف كبير).
3. بتحط الدالة كـ **مرجع** بدون أقواس:

- ✅ `onClick={handleClick}`
- ❌ `onClick={handleClick()}` ← دي بتشغل الدالة على طول مش لما يضغط.

### ✅ مثال تاني مع input:

```
function MyInput() {
  const handleChange = (event) => {
    console.log('You typed:', event.target.value);
  };

  return <input type="text" onChange={handleChange} />;
}
```

# introduction to state managment

### 💡 تعريف بسيط لـ State Management:

هي الطريقة اللي بنستخدمها **لإدارة وتخزين البيانات (state)** في التطبيق، **ومشاركة البيانات بين أكثر من مكون (Component)**.

### 🎯 مثال بسيط يوضح الفكرة:

تخيل إن عندك تطبيق فيه:

- مكون لكتابة اسم المستخدم.
- ومكون تاني بيعرض الاسم ده في مكان تاني في الصفحة.

لو كل مكون عنده الـ state الخاصة به، مش هيعرفوا يشاركوا البيانات مع بعض.

لكن باستخدام طريقة لإدارة الحالة (state management)، ممكن نخليهم يشتركوا في نفس البيانات ويحدثوها بسهولة.

## 🧠 أنواع إدارة الحالة:

1. **Local State (الحالة المحلية):**
   - هي اللي بتكون داخل Component معين.
   - بنستخدم `useState` أو `useReducer` لإدارتها.
   - مناسبة للحاجات البسيطة (زي: input, modal, checkbox).
2. **Global State (الحالة العامة):**
   - بنستخدمها لما نحتاج نشارك البيانات بين أكثر من Component.
   - بنستخدم أدوات زي:
     - **Context API** ✅ (أداة مدمجة في React)
     - **Redux** 🔄 (مكتبة خارجية قوية جدًا)
     - **Zustand, Recoil, MobX** (مكتبات بديلة)

## 🔄 الفرق بين local و global:

| النوع | مكان الاستخدام | مناسبة لـ |
|---|---|---|
| Local | داخل Component واحد | input, modal |
| Global | بين عدة Components | login info, cart, theme |

## 👇 باختصار:

> ال **State Management = تنظيم وتحديث البيانات داخل التطبيق بطريقة تخلّي كل مكون يعرف يتصرف صح حسب البيانات.**

## ✅ يعني إيه `useState`؟

ال `useState` هو **Hook** من React بنستخدمه علشان نحفظ ونتحكم في **البيانات المتغيرة** داخل الـ Component.
بمعنى تاني، لو عندك حاجة في الصفحة بتتغير (زي الاسم اللي المستخدم بيكتبه، أو عداد، أو حالة زرار)، هتحتاج تستخدمي `useState`.

## ✅ شكله العام:

```
const [state, setState] = useState(initialValue);
```

- `state` : ده اسم القيمة (زي متغير).
- `setState` : دي دالة بتغير القيمة.
- `useState(initialValue)` : بنحدد فيها القيمة الأولية (initial value).

## ✅ مثال واقعي:

خلينا نفترض إن عندك تطبيق فيه زرار، كل ما تدوس عليه، العداد يزيد 1.

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

function Counter() {
  const [count, setCount] = useState(0); // 0 القيمة الأولية

  const increase = () => {
    setCount(count + 1); // 1 لما ندوس على الزرار، نزود
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={increase}>Increase</button>
    </div>
  );
}

export default Counter;
```

## 📌 شرح:

- بدأنا العداد بـ 0.
- لما المستخدم يضغط الزرار، بنشغل `increase`.
- ال `setCount(count + 1)` بتغير القيمة، وده بيخلي React **تعيد عرض** (re-render) الـ component.

## ✅ استخدامات شائعة لـ `useState`:

- حفظ النص اللي المستخدم كتبه في input.
- تتبع هل عنصر معين ظاهر ولا مخفي.
- تبديل بين ثيم فاتح وداكن.
- التحكم في فتح/إغلاق مودال أو سايد بار.
- عداد زي المثال اللي فوق.

## مهم جدا

ال `setState` في React (سواء في الكلاسيك `class components` أو Hook زي `useState` في `function components`) بيشتغل **بشكل غير متزامن (Asynchronous)**، وده بيخلّي سلوكه مختلف شوية عن توقعاتنا أحيانًا.

## 🧠 يعني إيه "غير متزامن" هنا؟

لما تكتب:

```
setName("Sara");
console.log(name); // ده مش هايظهر "Sara"
```

المتغير `name` مش هايتم تحديثه فورًا بعد React . `setName` بيأجل التحديث لحد ما يخلص **render cycle** كامل. فـ `console.log(name)` هايظهر **القيمة القديمة**، مش اللي لسه حطّتيها.

## 🧩 ليه React بيعمل كده؟

علشان React يقدر:

- **يجمع أكتر من تحديث** ويعالجهم مرة واحدة (batching).
- يقلل عدد الـ renders، وده بيخلّي التطبيق أسرع.
- يحافظ على الأداء والكفاءة.

## 🎯 مثال حقيقي بيوضح القصة:

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

  const handleClick = () => {
    setCount(count + 1);
    console.log("Current count:", count);
  };

  return <button onClick={handleClick}>Click me</button>;
}
```

لو ضغط على الزرار، هايظهر في الكونسول:

```
Current count: 0
```

حتى لو اتحسبت القيمة الجديدة داخليًا، React لسه ما حدثش القيمة لغاية ما يخلص الـ render.

## 💡 طب أعمل إيه لو محتاج القيمة الجديدة فورًا؟

لو عايز تستخدم القيمة الجديدة لحظة التحديث، ممكن تكتب الشكل ده:(هايوضح الجزء بالتفصيل قدام)

```
setCount(prevCount => {
  console.log("Current count:", prevCount);
  return prevCount + 1;
});
```

في الحالة دي React بيديلك **القيمة الأحدث من الـ state**، وده مفيد لما تعتمد على القيم السابقة.

### 💡 إيه اللي بيحصل تحت الغطا؟

ال React لما يشوف `useState` :

1. بيخزن القيمة في مكان داخلي عنده.
2. لما تنادي `setCount(5)` مثلًا:
   - مش بيغيّر القيمة فورًا!
   - بيضيف الطلب في "قائمة انتظار".
   - بعد ما الكمبوننت يخلص شغله، React يعيد **render** بالـ state الجديد.

> 🎯 عشان كده بنقول التحديث "غير متزامن" (Asynchronous).

## 🧩 1. Handling User Inputs

أول حاجة بنبدأ بيها لما نعمل فورم في React هي إننا نستقبل مدخلات المستخدم.

✅ وده بيتم غالبًا باستخدام الـ event `onChange` .

مثال:

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

ده بيسجل اللي المستخدم بيكتبه ويحدث الـ state.

## 🧩 2. Single State vs Multiple States

لما بيكون عندنا فورم فيه أكتر من حقل (مثلاً: الاسم، الإيميل، الباسورد)، بيكون عندنا اختيارين:

- ✅ **Single state:**
  تخزن كل القيم في object واحد:

```
const [formData, setFormData] = useState({ name: "", email: "" });
```

✅ **Multiple states:**

تعمل لكل حقل state منفصلة:

```
const [name, setName] = useState("");
const [email, setEmail] = useState("");
```

🟡 الأفضلية: لو البيانات مرتبطة ببعض (زي فورم)، يبقى الـ Single State أنسب وأسهل في التعامل.

## 🧩 3. Update State from Previous State

لو عايز تحدث حالة بناءً على حالتها القديمة (زي عداد أو إضافة عنصر لقائمة):

```
setCount(prevCount => prevCount + 1);
```

ده مهم علشان تحديث الـ state في React **غير متزامن** (Asynchronous)، فلو استخدمت القيمة مباشرة بدون الرجوع للحالة السابقة ممكن تطلع النتيجة غلط.

## 🧩 4. Two-Way Binding

ده معناه إن المستخدم يكتب في الـ input، واللي كتبه يظهر في المكان اللي إحنا بنعرض فيه البيانات، ولو غيرنا الـ state يدويًا تتحدث القيمة في الـ input.

يعني العلاقة دايمًا شغالة في الاتجاهين:

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

## 🧩 5. Adding from Input + Form Submit (old way)

في الطرق القديمة (أو التقليدية)، بنحط البيانات داخل input، وبعدين نستخدم زر submit:

```
<form onSubmit={handleSubmit}>
  <input type="text" value={task} onChange={(e) => setTask(e.target.value)} />
  <button type="submit">Add Task</button>
</form>
```

والدالة `handleSubmit` بتستخدم `()preventDefault` علشان تمنع الصفحة من إعادة التحميل:

```
const handleSubmit = (e) => {
  e.preventDefault();
  // Add task logic
};
```

## 🧩 6. **Using `<form>` Instead of `<div>` for Accessibility

استخدام `<form>` بدل `<div>` له فوائد خاصة بذوي الاحتياجات:

- ✅ المتصفحات والأدوات المساعدة (screen readers) بتتعامل مع الفورمات بطريقة أوضح.
- ✅ تقدر تعمل Submit بالضغط على Enter.
- ✅ بيكون مفهوم أكتر لمحركات البحث والمستخدمين.
- ✅ تقدر تراقب كل حرف بيكتبه المستخدم و تحدث ال state بناءًا عليه باستخدام onChange{} في ال div مقدرش استخدم onSubmit .
- ✅ نعمل ارسال للبيانات (form submission): لما المستخدم يضغط submit نقدر نمسك الحدث onSubmit و نعالجه في ال JS بدل ما النموذج ينعاد تحميله زي زمان !

💡 مثال واقعي بسيط يربط كل دا:

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

function TodoForm() {
  const [formData, setFormData] = useState({ task: "", note: "" });

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

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Submitted:", formData);
    setFormData({ task: "", note: "" });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        name="task"
        value={formData.task}
        onChange={handleChange}
        placeholder="Enter task"
      />
      <input
        name="note"
        value={formData.note}
        onChange={handleChange}
        placeholder="Note"
      />
      <button type="submit">Add</button>
    </form>
  );
}

export default TodoForm;
```

## 🎯 يعني إيه "Send child information to parent"؟

في React، الطبيعي إن الـ **Parent Component** (يعني مكوّن الأب) هو اللي بيبعت بيانات للـ **Child Component** (يعني الابن) عن طريق الـ props.

لكن لو عايز **الابن يبعت حاجة للأب؟** ساعتها بنمشي بالعكس شوية:

- **الأب بيبعت دالة للابن كـ prop،**
- والابن لما يحب يبعتله حاجة (زي لما يضغط على زرار أو يكتب حاجة)،
- **ينادي على الدالة دي ويمرر البيانات فيها،**
- فالدالة دي بتشتغل في الأب وتستقبل البيانات، وتقدر تعمل بيها اللي هي عايزاه (زي إنها تحطها في `useState` مثلاً).

## 🛠️ إزاي نعمل كده؟ الخطوات:

1. ✅ الأب يعرف دالة تستقبل البيانات.
2. ✅ يبعت الدالة للابن عن طريق الـ props.
3. ✅ الابن ينادي الدالة دي ويديها الداتا.

## 💡 أهم معلومة لازم تفهمها:

> ✅ **الـ Parent هو اللي دايمًا بيتحكم في البيانات (state)، حتى لو اللي عدّلها كان Child.**

## 📦 مثال عملي مبسط (بالشرح جنب الكود):

### 🧐 Parent.js

```
import React, { useState } from 'react';
import Child from './Child';

function Parent() {
  const [childData, setChildData] = useState(""); // 🟡 هنا بنحط الداتا اللي جاية من الابن

  const handleChildData = (dataFromChild) => {
    setChildData(dataFromChild); // 🟢 كده استقبلنا الداتا من الابن وخليناها في state
  };

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Child says: {childData}</p>
      <Child sendDataToParent={handleChildData} /> {/* 🟠 بعتنا الدالة للابن */}
    </div>
  );
}
```

```
export default Parent;
```

## 👶 Child.js

```
function Child(props) {
  const sendData = () => {
    const data = "Hello from Child!"; // 🔵 أي بيانات احنا عايزين نبعتها
    props.sendDataToParent(data); // 🔴 بننادي الدالة اللي جت من الأب وبنبعت
الداتا فيها
  };

  return (
    <div>
      <h2>Child Component</h2>
      <button onClick={sendData}>Send Data to Parent</button>
    </div>
  );
}

export default Child;
```

## 📍 شرح اللي بيحصل خطوة بخطوة:

1. الأب عنده دالة اسمها `handleChildData` بتستقبل داتا.
2. بعتها للابن على هيئة prop اسمها `sendDataToParent` .
3. الابن عنده زرار، أول ما نضغط عليه، بيشغل دالة `sendData` اللي فيها سطر بينادي على `()props.sendDataToParent` وبيبعتلها الداتا.
4. الدالة دي بتتنفذ جوه الأب، وبالتالي يقدر يحتفظ بالداتا دي جوه state أو يعمل بيها أي حاجة.

## ⭐ معلومة مهمة جدًا (هنعلم عليها):

🔶 تقدر تبعت مش بس Strings،
✅ ممكن تبعت **Object**، أو **Array**، أو حتى **Function** من الابن للأب!

# 🟦 "Controlled vs Uncontrolled Components"؟

## ✅ Controlled Components

يعني React هي اللي ماسكة زمام الأمور.
هي اللي عارفة القيمة الحالية بتاعة input، وبتتحكم في كل تغيير بيحصل فيه.

**المستخدم يكتب = React تعرف = React تغير القيمة = ترجع تعيد رسم الـ UI.**

```
import React, { useState } from 'react';

function ControlledInput() {
  const [inputValue, setInputValue] = useState("");

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={handleChange}
      />
      <p>You typed: {inputValue}</p>
    </>
  );
}
```

🗣️ شرح بالعربي:
ده input مرتبط بـ useState. يعني لما المستخدم يكتب حرف، React بتحدث القيمة عن طريق setInputValue، واللي بيخلّي React ترندر الـ UI من تاني بالقيمة الجديدة.

## ✅ Uncontrolled Components

يعني React سايبة العنصر يعمل اللي هو عايزه بنفسه.
العنصر (input مثلًا) بيحتفظ بقيمته داخله، وReact مش بتتابع التغييرات دي لحظة بلحظة، لكن ممكن نوصله بـ `ref` لما نحتاج نعرف قيمته.

```
import React, { useRef } from 'react';

function UncontrolledInput() {
  const inputRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Input value is:", inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
```

```
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

### 🗣️ شرح بالعربي:

هنا React مش بتتدخل في القيمة، المستخدم يكتب براحتُه، ولما نضغط "Submit"، بنستخدم `ref` علشان نقرأ القيمة مرّة واحدة من الـ DOM.

### ✨ توضيح عام أكتر:

تخيل إن عندك طفل (input):

- **في الـControlled**: إنت (React) واقف وراه كل لحظة، بتسجل كل حاجة بيقولها وبتعدل معاه لو قال حاجة غلط.
- **في الـUncontrolled**: سايبه يقول اللي عايز، ولما يخلص كلامه، تيجي تسأله "إنت قلت إيه؟" وتشوف رده.

## 🟨 إمتى أستخدم Controlled؟ وإمتى Uncontrolled؟

### 🟢 Controlled لما:

- تعمل form validation لحظي
- تتبع القيم لحظة بلحظة
- تعمل Auto-Suggestions أو تعديل مباشر

### 🔵 Uncontrolled لما:

- تعوز تجمع بيانات من فورم كبير من غير تبعيات كتير
- تعمل integration سريع
- الأداء مهم جدًا ومفيش validation

# Rendering List of Data

### 🧠 الفكرة ببساطة:

أنت عندك **بيانات كتير** (زي وجبات، منتجات، تعليقات،... إلخ)،
وعايز **تعرضهم كلهم في الصفحة**، بس من غير ما تكتب نفس كود HTML مكرر عشرين مرة.

الحل؟
نخزن البيانات دي في **مصفوفة (Array)**،
ونستخدم طريقة `map()` علشان نمر على كل عنصر في المصفوفة، ونرجّع لكل عنصر شكل العرض الخاص بيه.

### 📃 إزاي بيحصل ده خطوة خطوة:

## 1. عندك مصفوفة فيها بيانات:

```
const records = [
  {
    date: new Date(2023, 7, 20),
    meal: "Breakfast",
    content: "Eggs",
    calories: 340,
  },
  {
    date: new Date(2023, 7, 21),
    meal: "Dinner",
    content: "Beans",
    calories: 600,
  },
];
```

كل عنصر داخل المصفوفة اسمه **object**،
وبيمثل وجبة، وكل وجبة فيها: التاريخ + نوع الوجبة + الأكل + السعرات.

## 2. لما تيجي تعرض البيانات دي:

```
<ul>
  {records.map((record) => (
    <li key={record.date.toString()}>
      <p>Calories: {record.calories}</p>
      <p>Meal: {record.meal}</p>
      <p>Content: {record.content}</p>
      <p>Date: {record.date.toString()}</p>
    </li>
  ))}
</ul>
```

أنت هنا بتقول:

> "يا React، خُد كل عنصر في records، وارجعلي عنصر `<li>` يحتوي على بيانات الوجبة".

ودي طريقة **ديناميكية جدًا** وسهلة جدًا، لأنك مش هتكرر الكود.

## ❗ ليه بنستخدم `key` هنا؟

ال React لما بتعرض list، لازم تفرق بين العناصر، فبنستخدم خاصية `key` علشان:

- ال React تقدر تعرف كل عنصر لوحده.

- ولما يحصل تحديث في العنصر، تعمله Update بس، مش تعيد بناء الـList كلها.

🔑 وعلشان كل وجبة ليها تاريخ مختلف، استخدمنا:

```
key={record.date.toString()}
```

## ✅ ليه الطريقة دي أفضل؟

1. **بتوفّر وقت ومجهود**: بدل ما تكرر كود HTML لكل عنصر.
2. **سريعة جدًا**: React تقدر تحدث عنصر واحد بدل القائمة كلها.
3. **ديناميكية**: لو زودت عنصر جديد في المصفوفة، هيظهر تلقائيًا.
4. **نظيفة ومنظمة**: الكود بيكون بسيط وسهل القراءة والصيانة

## 🎯 يعني إيه stateful list في React؟

يعني بدل ما تكتب بياناتك في مصفوفة عادية (const)، بتكتبها جوا `useState` ، وده بيخلي React يعرف يتابع التغييرات اللي بتحصل عليها ويعيد رسم الصفحة أوتوماتيك لما حاجة تتغير.

يعني مثلاً عندك لستة مهام (to-do list)، كل مرة تضيف مهمة أو تحذف واحدة، React يلاحظ التغيير ويحدث الواجهة.

## 🧠 طيب ليه ما نستخدمش array عادي؟

لأن الـ array العادي لو عدلت عليه، React مش هيعرف إنه فيه تغيير، ومش هيحدث الصفحة.

لكن لو بتحطه في `useState` ، كل ما تعدله بـ `setTasks` مثلاً، React هيعيد رسم الجزء ده من الصفحة أوتوماتيك.

## 🧪 مثال بسيط كده:

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

function TodoList() {
  const [tasks, setTasks] = useState([
    { id: 1, text: "أذاكر React" },
    { id: 2, text: "أخرج أتمشى شوية" }
  ]);

  const addTask = () => {
    const newTask = { id: Date.now(), text: "مهمة جديدة" };
    setTasks([...tasks, newTask]); // بنضيف المهمة الجديدة
  };

  return (
    <div>
      <button onClick={addTask}>أضف مهمة</button>
```

```
      <ul>
        {tasks.map((task) => (
          <li key={task.id}>{task.text}</li>
        ))}
      </ul>
    </div>
  );
}
```

### 💡 يعني اللي بيحصل هنا إيه؟

1. بنبدأ بـ لستة فيها مهمتين.
2. لما تدوس على الزرار، بيضيف مهمة جديدة.
3. ال `setTasks` بتخلي React يعرف إن فيه تغيير، فيعيد رسم الليستة.

### ⚠️ خد بالك من حاجتين مهمين:

- ماينفعش تستخدم `push()` على الـ array، لازم تعمل نسخة جديدة وتضيف عليها ( `[tasks, newTask...]` ).
- لازم كل عنصر في الـ map يكون ليه `key` مميز (زي `id` )، عشان React يعرف يفرّق بينهم.

## 🧠 يعني إيه Lists في React؟

في React، ساعات كتير بنكون محتاجين نعرض بيانات جاية من Array، زي لستة ناس، أو منتجات، أو وجبات... إلخ.

فبدل ما نكتب الكود كذا مرة، بنستخدم `map()` علشان نكرر عنصر معين بالشكل اللي احنا عايزينه، والبيانات بتتغير جوه كل عنصر حسب محتويات الـ Array.

### 💡 طب ليه الـ Key مهمة في اللستات؟

لما React تيجي تعرض عناصر من Array، هي مش بتعرضهم مره وخلاص! لأ، React ذكية، وبتتابع العناصر دي علشان لما يحصل تعديل أو حذف أو إضافة، تعرف إيه اللي اتغير بالظبط وما تعيدش كل اللستة من الأول.

هنا بقى بييجي دور الـ `key` 🔑.

### 📌 تخيلي كده:

عندك لستة فيها 100 اسم. واحد اتشال من النص.
لو مفيش Key، React مش هتعرف مين اللي اتشال، فهتقول: "يلا نعيد رسم اللستة كلها من جديد!" 😅

لكن لو كل عنصر ليه `key` مميز، React هتبص على الـ keys وتقول:

> "آه، العنصر اللي كان رقمه 7 اتشال! طب خلاص، أنا هشيل ده بس والباقي زي ما هو."

وده يخلي الأداء أسرع بكتير 👈، ويخلي التطبيق يشتغل بسلاسة أكتر وميهنجش.

### 👀 إزاي نختار الـ Key؟

- لازم يكون **فريد** لكل عنصر.
- يفضل نستخدم `id` لو متاح (زي المثال بتاعك).
- **ما تستخدمش** الـ index بتاع الـ map (إلا لو مضطر ومفيش `id` فعلاً).

## 🧪 مثال :

```
import React from 'react';
const PeopleList = () => {
    const people = [
    { id: 1, name: 'أحمد', age: 25 },
    { id: 2, name: 'سارة', age: 30 },
    { id: 3, name: 'محمد', age: 28 },   ];
    const renderPeople = () => {
        return people.map((person) => (
            <div key={person.id}>
                <p>الاسم: {person.name}</p>
                <p>العمر: {person.age}</p>
            </div>
        ));
        };
    return <div>{renderPeople()}</div>; };
export default PeopleList;
```

هنا كل شخص له id مختلف (1، 2، 3...) فـ React هتقدر تتابع مين اللي اتغير بسهولة لما يحصل update.

## ✅ خلاصة:

- الـ `key` مهمة علشان React تعرف تتابع العناصر جوه القايمة وتحدثها بكفاءة.
- من غير React ، `key` ممكن تعيد رسم كل العناصر حتى لو حاجة بسيطة اتغيرت.
- لازم يكون الـ `key` فريد وثابت.

## 📌 يعني إيه Conditional Behavior؟

في React، ممكن تعرض جزء من الصفحة **بس لو شرط معين اتحقق**.

يعني مثلاً:

- لو المستخدم سجل دخوله ← نعرضله اسمه.
- لو مافيش بيانات ← نعرض "لا توجد بيانات".
- لو حصل Error ← نعرض رسالة الخطأ.

## 📘 طيب بنستخدم إيه؟

عندك كذا طريقة:

### 1. باستخدام `if / else` :

```
function MyComponent() {
  const isLoggedIn = false;
  let message;

  if (isLoggedIn) {
    message = "أهلاً بيك!";
  } else {
    message = "من فضلك سجل الدخول";
  }

  return <div>{message}</div>;
}
```

📌 **الشرح:**
بتجهز الرسالة حسب الشرط، وبعدين تعرضها كلها مرة واحدة.

## ✅ 2. الاستخدام بـ `&&` (الـ Logical AND)

ده لما تحب **تظهر حاجة بس لما الشرط يكون true**، ومش محتاج else.

```
function MyComponent() {
  const isLoggedIn = true;

  return (
    <div>
      {isLoggedIn && <p>أهلاً بيك!</p>}
    </div>
  );
}
```

📌 **الشرح:**
لو `isLoggedIn` طلعت true ← هيتعرض `<p>أهلاً بيك!</p>`
لو false ← مش هيعرض أي حاجة (هيتجاهل السطر ده).

## ✅ 3. استخدام switch (لو عندك حالات كتيرة)

لو مش حالة واحدة true/false، لأ، عندك حالات متعددة:

```
function StatusMessage({ status }) {
  let message;
```

```
  switch (status) {
    case 'loading':
      message = 'جاري التحميل...';
      break;
    case 'success':
      message = 'تم بنجاح!';
      break;
    case 'error':
      message = 'حصل خطأ!';
      break;
    default:
      message = 'حالة غير معروفة';
  }

  return <div>{message}</div>;
}
```

📌 **الشرح:**
بتشيك على قيمة `status` وتعرض الرسالة المناسبة ليها.

## ✅ 4. التغليف في دالة صغيرة

أحياناً لو الشرط معقد أو طويل، بنحطه في دالة مستقلة:

```
function renderMessage(isLoggedIn) {
  if (isLoggedIn) {
    return <p>أهلاً بيك!</p>;
  }
  return <p>سجل الدخول من فضلك</p>;
}

function MyComponent() {
  return (
    <div>
      {renderMessage(true)}
    </div>
  );
}
```

## ✅ 5. باستخدام شرط داخل JSX (الـ ternary operator):

```
function MyComponent() {
  const showContent = true;
```

```
  return (
    <div>
      {showContent ? 'عرض المحتوى هنا' : 'لا يوجد محتوى للعرض'}
    </div>
  );
}
```

💬 **شرحها بالبلدي:**
هنا بنقول في السطر ده:

> لو `true` = `showContent` ⟵ هنعرض "عرض المحتوى هنا"
> لو `false` ⟵ هنعرض "لا يوجد محتوى للعرض"

يعني كأنك بتقول له: **اختار واحدة حسب الشرط**.

## ✅ ليه مهم أوي الموضوع ده؟

علشان:

- تحكم في **إيه اللي يظهر وامتى**.
- تخل التطبيق **ديناميكي** ويتغير حسب حالة المستخدم.
- تظهر أو تخفي محتوى بسهولة من غير ما تعيد كتابة كود كتير.

## 🎯 امتى أستخدم أي طريقة؟

| الطريقة | تستخدميها لما... |
|---|---|
| `? :` | عندك اختيارين بس (true / false) |
| `&&` | عايز تعرض حاجة بس لما الشرط true |
| `if` / `switch` | عندك أكتر من حالة، أو عايز تكتب الكود بشكل أوضح |
| دالة منفصلة | الشرط كبير أو معقد وعايز تنظم الكود |

## ✅ ما معنى Dynamic Style Change؟

هو ببساطة:
**تغيير شكل العنصر (زي اللون، الحجم، الخط...) أثناء تشغيل الصفحة، حسب حالة معينة أو حدث معين.**

يعني بدل ما تكتب لون معين وثابت في الـ CSS، تقدر تخليه يتغير حسب شرط معين باستخدام JavaScript داخل React.

## 🎯 ليه نستخدمه؟

عشان نخلي الصفحة **تتفاعل** مع المستخدم!
مثلاً:

- لو حصل خطأ، الكلام يظهر بلون أحمر.
- لو المستخدم ضغط على زر، نغير شكل الكلام.
- لو عنصر "مميز" نخليه يظهر بلون مختلف وخط أكبر.

## 📘 المثال الأول مع زر:

```
import React, { useState } from 'react';

const MyComponent = () => {
  const [isSpecial, setIsSpecial] = useState(false); // حالة للتغيير

  // تحديد الـ style حسب القيمة
  const styles = isSpecial
    ? {
        color: 'blue',
        fontSize: '20px',
        fontWeight: 'bold',
      }
    : {};

  return (
    <div>
      <div style={styles}>
        Welcome to you in my world
      </div>
      <button onClick={() => setIsSpecial(!isSpecial)}>
        Change Style
      </button>
    </div>
  );
};
```

## 💡 شرح عميق للمثال:

- عندك حالة `isSpecial` نوعها Boolean (صح أو غلط).
- لما المستخدم يضغط على الزر، القيمة تتغير ( `false` ↔ `true` ) باستخدام `setIsSpecial` .
- لو القيمة `true` ، يتم تطبيق الـ style اللي فيه لون أزرق وخط كبير.
- لو القيمة `false` ، مش هنحط أي style.

## ✅ المثال الثاني بشرط مباشر:

```
const isError = true;

<div
  style={{
    color: isError ? 'red' : 'black',
    fontSize: isError ? '18px' : '14px',
    fontWeight: isError ? 'bold' : 'normal',
  }}
>
  Welcome to you in my world
</div>
```

### 💡 شرح المثال:

- بنقول: لو في خطأ ( `isError = true` ) ⟵ خليه يظهر بلون أحمر وخط كبير وغامق.
- لو مفيش خطأ ( `isError = false` ) ⟵ اللون عادي، والخط صغير.

## 📌 طرق تانية لتغيير الـ Style:

### 1. ✅ باستخدام `className`:

لو عندك CSS خارجي:

```
const isSpecial=true;

<div className={isSpecial ? 'special' : 'normal'}>
  Welcome to you
</div>
```

وفي ملف CSS:

```
.special {
  color: blue;
  font-weight: bold;
}

.normal {
  color: gray;
}
```

### 2. ✅ باستخدام مكتبات زي `styled-components` :

وفي كمان package تانية غيرها ب syntax تاني اسمها stitches styling , لكن styled-components افضل و خصوصا انها بتستخدم syntax مش غريب على JS وهو ال tagged string template literals تقدر تبحث عنه و تفهمه اكتر من اللينك ده https://wesbos.com/tagged-template-literals

## ✅ يعني إيه Styled Components؟

ببساطة، دي طريقة نقدر بيها نكتب CSS جوه كود React نفسه، من غير ما نحتاج نفتح ملف CSS خارجي.
يعني بدل ما تكتب className وتروح تعرف الـ CSS في ملف تاني، لأ... هتكتب كل حاجة في نفس الملف، بس بـ **شكل منظم ومحمي**.

## ✅ طيب ليه أستخدمها بدل CSS العادي؟

1. **كل Component ليه ستايله الخاص:**
   مش هتقلق من إن class يبوظ شكل حاجة تانية، لأن كل ستايل خاص بالـ component ده بس.
2. **تقدر تستخدم المتغيرات بسهولة:**
   مثلاً تغير لون الزرار لو المستخدم ضغط عليه، بسهولة عن طريق props أو state.
3. **الشيفرة أوضح وأسهل في الصيانة:**
   كل حاجة في ملف واحد. React + CSS = ❤️

## ✅ إزاي بنستخدمها؟

```
import styled from 'styled-components';
```

ده الاستدعاء الأول.

بعد كده بنعرّف عنصر Styled خاص بينا بالشكل ده:

```
const StyledButton = styled.button`
  color: white;
  font-size: 16px;
  padding: 10px 20px;
  background-color: blue;
`;
```

هنا إحنا أنشأنا زرار بلونه وحجمه وخلفيته.

## ✅ وبعدين نستخدمه إزاي؟

زي أي React Component عادي:

```
function MyComponent() {
  return (
    <div>
      <StyledButton>Click Me</StyledButton>
```

```
    </div>
  );
}
```

## ✅ يعني إيه "props" في Styled Components؟

يعني تقدر تبعت بيانات (زي لون – حجم – حالة) من الـ component بتاعك للـ Styled Component، وهو يغيّر شكله حسب الـ props دي.

يعني مثلاً لو بعتله `primary={true}` ، يغير اللون ويبقى زرار رئيسي، ولو مش بعتله حاجة، يبقى زرار عادي.

## ✅ لو عايز أغير اللون مثلاً حسب حالة الزرار؟

تقدر تبعت props:

```
const StyledButton = styled.button`
  background-color: ${(props) => (props.primary ? 'blue' : 'gray')};
  color: white;
`;
```

وبعدين في الاستخدام:

```
<StyledButton primary>Primary Button</StyledButton>
<StyledButton>Secondary Button</StyledButton>
```

لو الزار عليه attribute اسمها primary هايكون blue غير كده هايكون gray
لو بعت `primary={false}` صراحةً كده، هيبقى اللون **gray** برضو، لأن القيمة false.

## ✅ فائدة كبيرة كمان: ديناميكية التغيير

تقدر تستخدم state وتغير الستايل في لحظة، مثلاً:

```
const MyComponent = () => {
  const [isActive, setIsActive] = useState(false);

  return (
    <div>
      <StyledButton
        onClick={() => setIsActive(!isActive)}
        primary={isActive}
      >
        {isActive ? 'Active' : 'Inactive'}
      </StyledButton>
```

```
    </div>
  );
};
```

## 🔚 الخلاصة:

- ال Styled Components بتخلي الشكل مع الكود في نفس المكان.
- بتحميك من الـ CSS اللي بيبوظ بعضه.
- بتخلي الكود أنظف وأسهل في القراءة والتعديل.
- تقدر تستخدم فيه props و state بسهولة علشان تعمل ستايلات متغيرة.

## 📝 ملاحظة مهمة عن Styled Components:

أي **CSS** بتكتبه جوه `styled-components` هو نفس شكل الـ **CSS العادي**،
بس في اختلافات بسيطة:

1. ✅ **بتكتبه جوه JavaScript** باستخدام backticks `` ` ``, مش في ملف `.css`.
2. ✅ تقدر تستخدم **JavaScript logic** جواه، زي `props`:

```
color: ${(props) => (props.primary ? 'blue' : 'gray')};
```

3. ✅ تكتب الـ **Media Queries** بنفس طريقة CSS العادية:

```
@media (max-width: 768px) {
  font-size: 14px;
}
```

يعني بكل بساطة:
**نفس CSS اللي متعود عليه، لكن جواه JavaScript وبتقدر تضيف منطق (Logic) بسهولة.**

## 💡 إيه هي CSS Modules؟

الـ **CSS Modules** هي طريقة منظمة لكتابة الـ CSS في React،
بتخلي كل مكون (Component) ليه تنسيقه الخاص،
وده يمنع إن الكلاسات تتلخبط مع بعض أو يحصل تعارض (conflict) في الأسماء.

## ✅ إزاي نستخدمها خطوة بخطوة؟

### 🟢 1. إنشاء ملف CSS خاص بالمكون

يعني كل كومبوننت تعملي له ملف CSS باسمه،
بس تسمي الملف بـ `module.css.[اسم]` عشان React تفهم إنه CSS Module.

مثلاً:

```
Button.module.css
```

وتكتب جواه التنسيقات عادي جدًا:

```
/* Button.module.css */
.button {
  background-color: #3498db;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.button:hover {
  background-color: #2980b9;
}
```

### 🟢 2. استيراد الملف في الكومبوننت

يعني في ملف الكومبوننت `Button.js` تعمل import بالشكل ده:

```
import styles from './Button.module.css';
```

وهنا `styles` هي object فيها كل الكلاسات اللي كتبتها في الملف.

### 🟢 3. استخدام الكلاسات

بدل ما تكتب الكلاس كده:

```
<button className="button">
```

هتكتبه كده:

```
<button className={styles.button}>
```

✅ كده React هتربط الزرار بالكلاس `button.`
بس كمان هتضيف عليه اسم عشوائي مخصوص زي:

```
class="Button_button__3T65H"
```

وده بيمنع التعارض مع أي كود CSS تاني في التطبيق.

## 💡 طيب ليه نستخدم CSS Modules؟

- 🛡️ **تمنع التعارض** بين الكلاسات لما المشروع يكبر.
- 🧹 **تنظيم الكود**: كل مكون ليه ملف الـ CSS بتاعه.
- 🧠 **أسهل في التعديل**: تعرف بسرعة التنسيقات الخاصة بكل مكون.

# ✅ التركات المهمة في CSS Modules (Dynamic Styling)

## 🔷 1. استخدام أكثر من كلاس مع بعض

لو عايز تطبق أكثر من كلاس على عنصر واحد، تعمل كده:

```
<div className={`${styles.box} ${styles.active}`}></div>
```

> 🔶 ممكن كمان تستخدم مكتبة `clsx` أو `classnames` لو هتكرر ده كتير.

## 🔷 2. كلاس ديناميكي حسب شرط (Conditional class)

```
<div className={isActive ? styles.active : styles.inactive}>
  Hello
</div>
```

## 🔷 3. استخدام `template literals` لو هتبدلي بين أكثر من حالة:

```
<div className={`${styles.button} ${isPrimary ? styles.primary :
styles.secondary}`}>
  Click me
</div>
```

## 🔷 4. استخدام مكتبة `clsx` أو `classnames` (نصيحة!)

عشان تسهل كتابة الكلاسات الديناميكية، نزل مكتبة `clsx` :

```
npm install clsx
```

واستخدمها كده:

```
import clsx from 'clsx';
import styles from './Button.module.css';

const Button = ({ isPrimary }) => {
  return (
```

```
    <button className={clsx(styles.button, isPrimary && styles.primary)}>
      Click me
    </button>
  );
};
```

> 🔶 ال `clsx` بتوفرلك كتابة نظيفة بدل ما تكتب if-else أو ternary كتير.

## 🔷 5. استخدام كلاس داخل كلاس (Nesting)

لو بتشتغل بـ **Sass modules** (مثلاً `.module.scss`) تقدر تعمل Nesting:

```
/* Button.module.scss */
.button {
  background: red;

  &:hover {
    background: darkred;
  }

  &.active {
    border: 2px solid green;
  }
}
```

## 🔷 6. لو عندك كلاس فيه Dash ( - ) في الاسم

مثال: `button-primary`

تستخدمه بالشكل ده:

```
<div className={styles["button-primary"]}></div>
```

## 🔷 7. تغيرات على حسب Props

مثلاً:

```
const Button = ({ variant }) => {
  return (
    <button className={styles[variant]}>
      Click
    </button>
```

```
  );
};
```

لو عملت:

```
<Button variant="primary" />
```

ال React هيربط الكلاس `styles.primary`