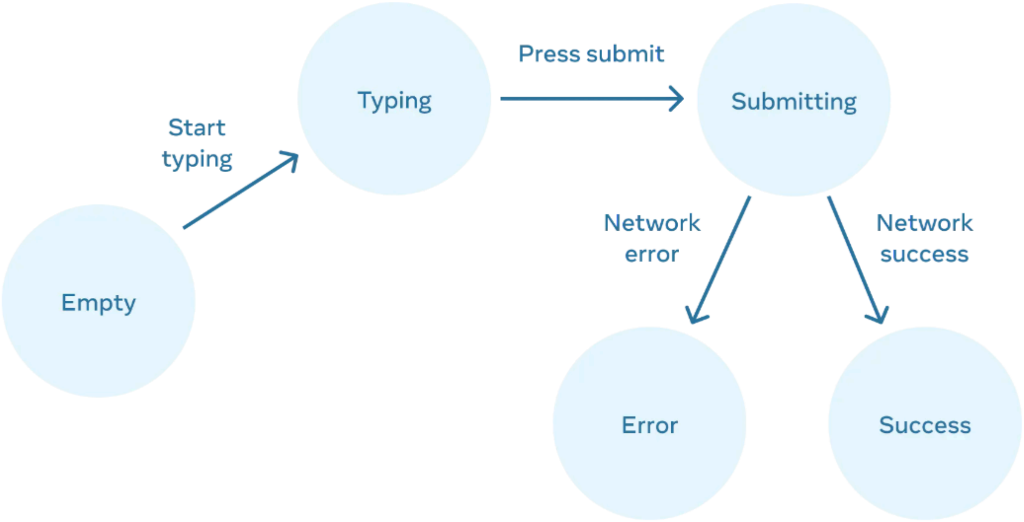
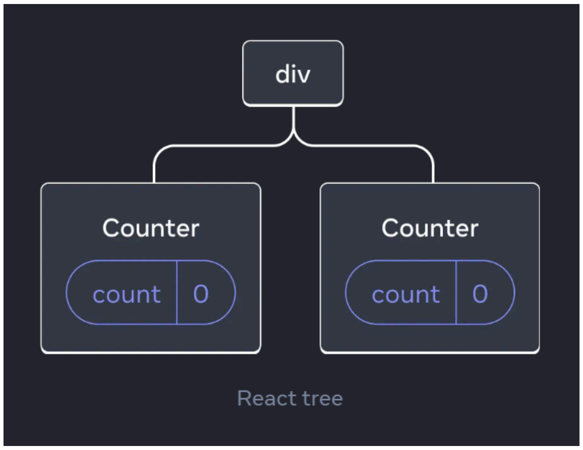
- ডিক্লেয়ারেটিভ এ স্টেপ বাই স্টেপ instruction দেয়ার প্রয়োজন হয় না। যেমনঃ রিয়্যাক্টে স্টেট আপডেট হলে UI ও একা একাই আপডেট হয়ে যায়। আমাদের ম্যানুয়ালি ডম ধরে ধরে আপডেট করা লাগে না। অপরদিকে ইম্পেরেটিভ এর ক্ষেত্রে ভ্যারিয়াবল এর ভ্যালুও আপডেট করা লাগে, ডম ধরে ধরে সেই ভ্যালু আপডেট ও করা লাগে।
- ইম্পেরেটিভ প্রোগ্রামিং এর ক্ষেত্রে প্রোগ্রামে সব কিছু অনেক precisely বলে দিতে হয় যে, কোনটির পরে কোনটি করবে।
- ডিক্লেয়ারেটিভ ও ইম্পেরেটিভ প্রোগ্রামিং এর পার্থক্য বোঝার জন্য একটা উদাহরণ নেয়া যেতে পারে। যেমন ধরুন আপনি আপনার উবার ড্রাইভারকে বললেন মিরপুর ১০ যাবেন। এর সাথে আর কিছু বলেননি। তাহলে আপনার ড্রাইভার আপনাকে আপনার গন্তব্যে পৌঁছে দিবেন। এক কথাতেই শেষ। কিন্তু আপনি গাড়িতে চড়ে আপনার গন্তব্যের কথা না বলে এভাবে বলা শুরু করলেন যে, ডানে যান, এবার বামে যান, তারপর আবার ডানে যান। এভাবে যতক্ষণ না আপনি গন্তব্যে পৌঁছাচ্ছেন ততক্ষণ এভাবেই বলতে থাকলেন। এভাবেও আপনি পৌঁছাতে পারবেন। কিন্তু এক্ষেত্রে আপনাকে অনেক অনেক বেশি দিকনির্দেশনা দেয়া লাগবে। প্রথম উপায়টি হচ্ছে ডিক্লেয়ারেটিভ এবং পরের বেশি বেশি নির্দেশনা দেয়ার উদাহরণটি ইম্পেরেটিভ প্রোগ্রামিং এর উদাহরণ।
- রিয়্যাক্ট কাজ করে ডিক্লেয়ারেটিভ পদ্ধতিতে।
- UI কে ডিক্লেয়ারেটিভলি চিন্তা করতে চাইলে আমরা পাঁচটি ধাপ অনুসরণ করতে পারি। ধাপগুলো নিচে উল্লেখ করা হলো।
- কম্পোনেন্টের বিভিন্ন ভিজুয়্যাল অবস্থাকে চিহ্নিত করতে হবে। প্রথমে আমরাকে আগাম চিন্তা করে নিতে হবে যে, একটা কম্পোনেন্ট কি কি শর্তাধীনে কি কি ধরনের আউটপুট দিতে পারে। ধরা যাক, একটা কম্পোনেন্ট আছে যেটা কিনা একটা টেক্সটবক্স শো করে এবং ওই বক্সে কিছু লিখে সাবমিট বাটনে চাপ দিলে বক্সের কনটেন্ট অনুযায়ী কিছু একটা মেসেজ প্রিন্ট করে দেয়। তো আমাদের পাঁচটি ধাপের প্রথমটিতে এটা আগাম কল্পনা বা ডিজাইন করে নিতে হবে যে, টাইপ করলে UI টা কেমন দেখাবে, টাইপ না করলে কেমন দেখাবে, টাইপ করতে করতে আবার সবকিছু মুছে দিলে কেমন দেখাবে, আনএক্সপেক্টেড কিছু টাইপ করলে কেমন দেখাবে, সাবমিট বাটনে চাপ দেয়ার পর UI কেমন দেখাবে ইত্যাদি। সব ধরনের কেইসে UI কেমন হবে তার সংকলনকে living styleguides বা storybooks বলে।
- দ্বিতীয় ধাপে আমরাকে চিহ্নিত করতে হবে যে, কি কি কারণে UI পরিবর্তন হতে পারে। সেটা ইউজারের ক্লিকের কারণে হতে পারে, ইনপুট ফিল্ডে কিছু লেখার কারণে হতে পারে ইত্যাদি। আবার কম্পিউটারের কোন কাজের কারণেও হতে পারে, এই যেমন নেটওয়ার্ক রেসপন্সের কারণে হতে পারে, বা একটা ছবি হয়তো লোড হচ্ছে, বা কোন টাইমআউট কমপ্লিট করছে ইত্যাদি। মানুষের দেয়া বিভিন্ন ইনপুটকে হ্যান্ডেল করার জন্য বেশিরভাগ সময়েই event handlers প্রয়োজন পড়ে।



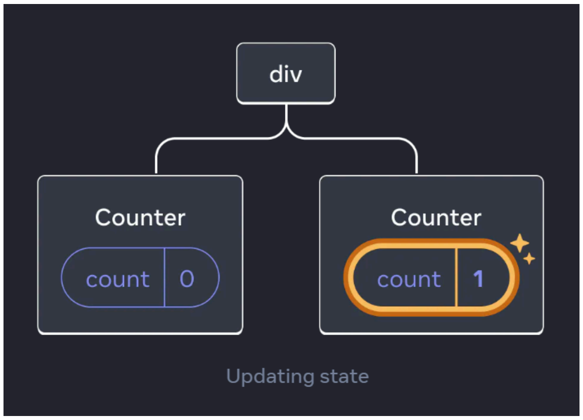
Form states

- তৃতীয় ধাপে ভিজুয়্যাল স্টেটগুলোকে রিঅ্যাক্টের ইন মেমোরি ব্যবস্থা useState ব্যবহার করে ধরে রাখতে হবে।
- চতুর্থ ধাপে আমাদের চেষ্টা করতে হবে যত কম সংখ্যক state ব্যবহার করে কাজ সারা যায় তত ভালো। অযথা সবগুলো ভিজুয়্যাল স্টেটের জন্য আলাদা আলাদা state নেয়ার দরকার নেই। এমনও হতে পারে একটা state দিয়েই দুইটা বা তারও বেশি সংখ্যক ভিজুয়্যাল স্টেটের কাজ সেরে ফেলা গেল। নিজেকে কয়েকটা প্রশ্ন করে আমরা সিদ্ধান্ত নিতে পারি যে, state সংখ্যা কিভাবে কমানো যায়। যেমন, একটা state যদি false হয় তাহলে দেখা গেল যে আরেকটা state কখনোই false পারেনা। তাহলে এই দুটোকে কম্বাইন করে একটা state দিয়েই কাজ চালানো যায়। আবার একটা state এর তথ্য যদি অলরেডি আমরা অন্য একটি state থেকে পেতে পারি তাহলে সেক্ষেত্রে একাধিক state এর কি দরকার!
- পঞ্চম এবং শেষ ধাপে আমরা সবগুলো state কে একে একে event handlers সাথে যুক্ত করে দিতে পারি।
- এবার আমরা কথা বলবো অনেকগুলো state কে কিভাবে গ্রুপিং করা যায় বা স্ট্রাকচার করা যায়। এক্ষেত্রে কিছু নিয়মকানুন আছে যেগুলো মেনে চললে প্রোগ্রামে বাগস কম হবে এবং ওয়েল স্ট্রাকচারড হবে। চলুন নিয়মগুলো এক নজর দেখে নেয়া যাক।
- যদি এমন দেখা যায় যে, দুই বা ততোধিক state কে আমরা সবসময় একইসাথে আপডেট করছি। তাহলে তাদেরকে মার্জ করে একটা state বানিয়ে ফেলা যায়। অনেক সময় প্রোগ্রামে আমরা কো অর্ডিনেট বা স্থানাঙ্ক নিয়ে কাজ করি। বেশিরভাগ সময়েই x ও y এর মান একসাথে আপডেট করা লাগে। তাহলে এক্ষেত্রে x এর জন্য আলাদা এবং y এর জন্য আলাদা state না বানিয়ে তাদেরকে মার্জ করে নেয়া যায়। একাধিক state কে একটা অবজেক্ট বানিয়ে একটা state এ রাখা হয় আরেকটা ক্ষেত্রে, সেটা হলো যখন আমরা অনিশ্চিত যে কত সংখ্যক state আমাদের প্রয়োজন হতে পারে।

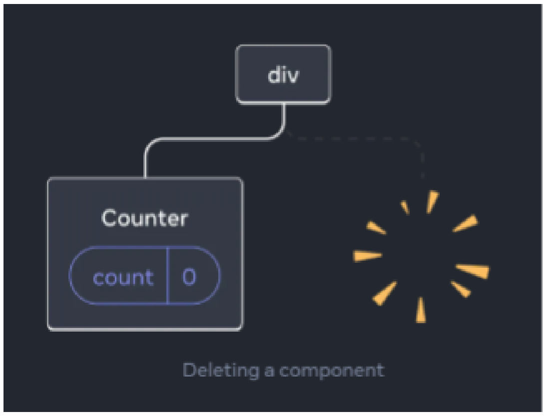
- state এর মধ্যে কন্ট্রাডিকশন এড়িয়ে চলতে হবে অর্থাৎ একই state এর দুইটি আলাদা তথ্য যদি একে অপরের সাথে কন্ট্রাডিক্ট বা ডিসঅ্যাগ্রি করে তাহলে এরকম পরিস্থিতি এড়াতে হবে।
- এমন কোন তথ্য যেটা আমরা কম্পোনেন্টের মধ্যেই হিসাব-নিকাশ করে ফেলতে পারছি বা অন্য কোন state থেকে পেয়ে যাচ্ছি তাহলে সেই তথ্যকে আলাদা করে অন্য একটা state এ রাখার দরকার নেই। অর্থাৎ state রিডানডেন্সি কমাতে হবে। যেমন, যদি একটি state থাকে যেটা নামের প্রথম অংশ এবং দ্বিতীয় কোন state নামের শেষ অংশ ধারণ করে তাহলে আমরা এই দুটি state ব্যবহার করেই কিন্তু পুরো নাম ডিরাইভ করতে পারবো। এক্ষেত্রে পুরো নাম ধরে রাখার জন্য তৃতীয় কোন state রাখলে সেটি হবে রিডানডেন্সি।
- একই ডাটা যখন একাধিক state এ ঘোরাফেরা করে তখন তাকে আপ টু ডেট রাখা মুশকিল হয়ে যায়, এরকম পরিস্থিতিতে ডুপলিকেশন বাদ দিতে হবে।
- ডিপলি নেস্টেড state এর দরকার নেই। এরকম state আপডেট করার জন্য ভালো না। সেজন্য আমরা ডাটাকে নরমালাইজ বা ফ্ল্যাট করে নিবো। এসব নিয়ম কানুনের উদ্দেশ্য হচ্ছে কোন ধরনের ভুল ভ্রান্তি ছাড়া state কে সবসময় আপডেট উপযোগী রাখা।
- Derived State বলতে অন্য কোন State বা Props থেকে যেই ভ্যারিয়াবল বা State তৈরি হয় তাকে বোঝায়।
- যতটুকু সম্ভব Props Mirror করা উচিত নয়, তবুও যদি এমন কোন সিচুয়েশন আসে যখন প্যারেন্ট এর সাথে চাইল্ড State এর synchronization এর কোন প্রয়োজন নেই অথবা শুধু একবার প্যারেন্ট এর State ব্যবহার করতে হবে শুধু তখন আমরা চাইলে Props mirror করতে পারি।
- Props চাইল্ড কম্পোনেন্টে পাঠিয়ে সেটা আবার চাইল্ডে স্টেট হিসেবে রেখে কাজ করলে প্যারেন্ট ও চাইল্ড কম্পোনেন্টে একই ডাটা নিয়ে কাজ করা হয়, তখন তাকে Props Mirror হয়।
- অনেক সময় এরকম হয় যে, আমরা দুটি কম্পোনেন্টের state সবসময় একইসাথে পরিবর্তন করে থাকি। এক্ষেত্রে দুটি কম্পোনেন্ট থেকে state সরিয়ে তাদের নিকটবর্তী প্যারেন্ট কম্পোনেন্টে নিয়ে গিয়ে প্রপস আকারে চাইল্ডে পাঠালে কাজের সুবিধা হয়। এই বিষয়টিকে lifting state up বলে। প্রথম অবস্থাকে বলা হয় uncontrolled component এবং পরের অবস্থাকে controlled component বলা হয়। কেননা দ্বিতীয় ক্ষেত্রে একটা প্যারেন্টের হাতে পুরো নিয়ন্ত্রণ থাকছে। এই অবস্থাকে আরও একটি ফ্রেজ দিয়ে সংজ্ঞায়িত করা হয়, a single source of truth for each state.
- state কম্পোনেন্টের একান্ত সম্পত্তি। ভিন্ন ভিন্ন কম্পোনেন্টে state গুলো আলাদা সত্ত্বা বজায় রেখে চলে। কোন্ state কোন্ কম্পোনেন্টের সাথে জড়িত এটা খুব ভালোভাবে রিঅ্যাক্ট মনে রাখে এবং UI ট্রিতে তা সংরক্ষণ করে। State is tied to a position in the render tree.
- এটা মনে হতে পারে যে, state গুলো তাদের নির্দিষ্ট কম্পোনেন্টের মধ্যে বসবাস করে কিন্তু তা সঠিক নয়। state গুলো মূলত রিঅ্যাক্টের মধ্যেই থাকে, রিঅ্যাক্ট নিজ দায়িত্বে রেন্ডার ট্রি বানানোর সময় state গুলোকে তাদের সঠিক কম্পোনেন্টের সাথে জুড়ে দেয়।

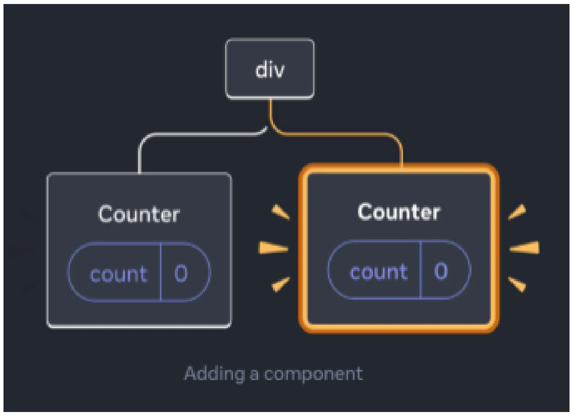


- এখানে দুটি আলাদা কম্পোনেন্ট (Counter) দেখা যাচ্ছে। এরা একই কম্পোনেন্ট হলেও দুজনের ফাংশনালিটি আলাদা হবে কারণ এরা রেন্ডার ট্রি এর আলাদা জায়গায় অবস্থান করছে।

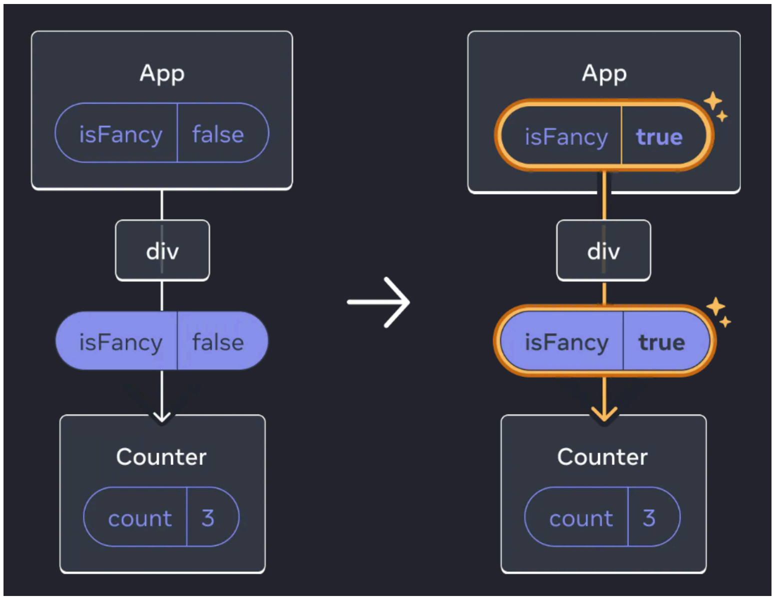


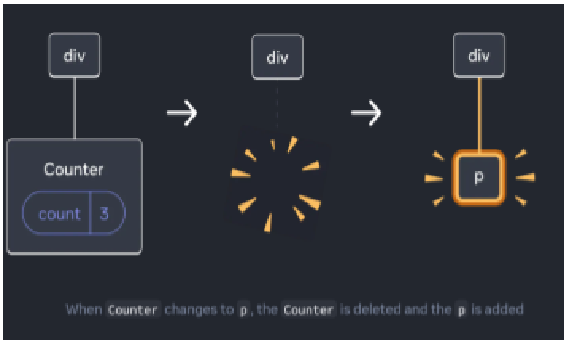
- একজনকে আপডেট করলে আরেকজনের ওপর কোন প্রভাব পড়বেনা।

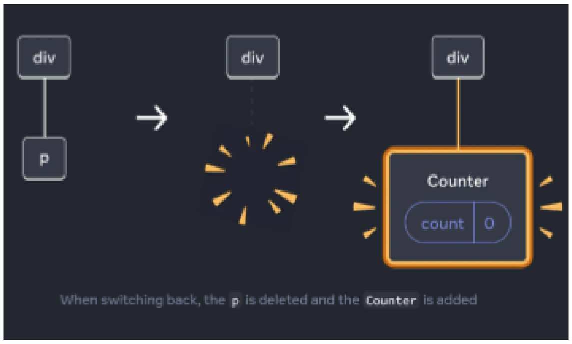




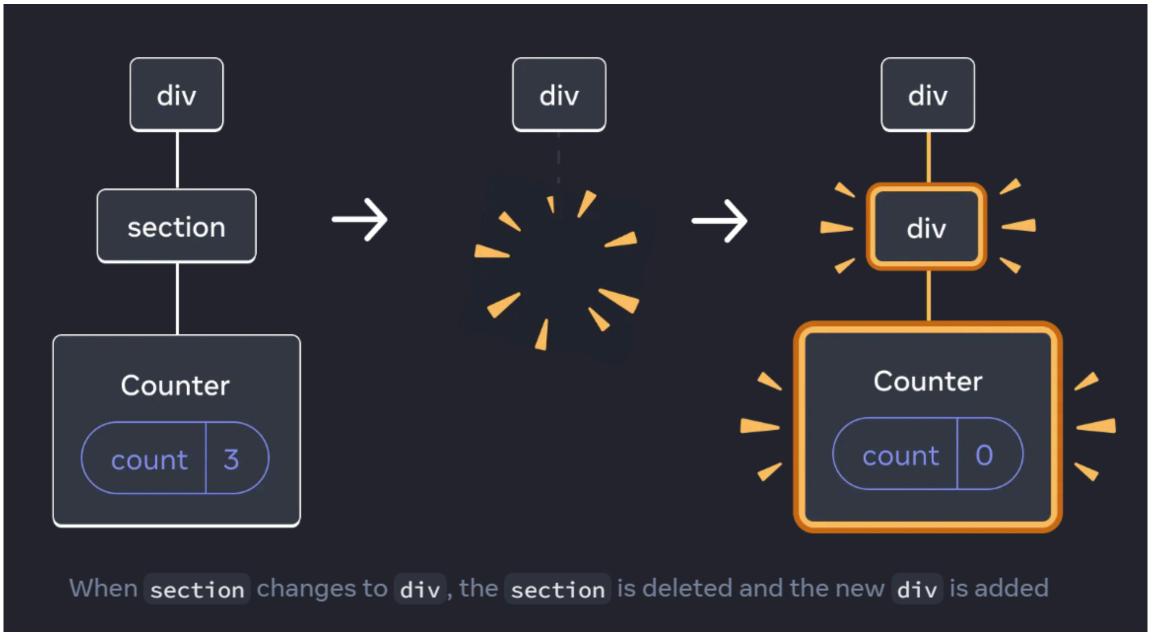
- কিন্তু কোন একটি কম্পোনেন্টকে ডিলিট করে দেয়া হয় তাহলে সেটি তার state সহ গায়েব হয়ে যাবে। আবার ফেরত নিয়ে আসলে তার state রিসেট হয়ে ফেরত আসে।
- রিঅ্যাক্ট ততক্ষণই একটা কম্পোনেন্টের state কে সংরক্ষণ করে রাখে যতক্ষণ সেটা রেন্ডার ট্রি এ তার নিজের জায়গায় বহাল থাকে। যদি সেটা সেখান থেকে সরিয়ে নেয়া হয় বা অন্য কোন কম্পোনেন্ট সেই জায়গায় রেন্ডার করা হয় তাহলে state হারিয়ে যায়। Same component at the same position preserves state.



- এখানে Counter নামক কম্পোনেন্টের state কে সংরক্ষণ করা হয়েছে কারণ এটি রেন্ডার ট্রি এ তার নিজস্ব জায়গায় রয়েছে কোন ধরনের পরিবর্তন ছাড়া। যদিও স্টাইলে কিছু পরিবর্তন এনে পরীক্ষা করা হয়েছে কিন্তু সেটা রেন্ডার ট্রি তে কম্পোনেন্টের অবস্থানে কোন পরিবর্তন আনেনা। It's the same component at the same position, so from React's perspective, it's the same counter. Remember that it's the position in the UI tree—not in the JSX markup—that matters to React!
- 

When `Counter` changes to `p`, the `Counter` is deleted and the `p` is added


When switching back, the `p` is deleted and the `Counter` is added
- Different components at the same position reset state, এখানে আমরা রেন্ডার ট্রি এর যে জায়গায় Counter নামক কম্পোনেন্ট ছিল সেখানে p ট্যাগ নিয়ে এসেছি, যখন আবার আগের কম্পোনেন্টকে ফেরত আনা হবে তখন তার state রিসেট হয়ে যাবে। Also, when you render a different component in the same position, it resets the state of its entire subtree.



When `section` changes to `div`, the `section` is deleted and the new `div` is added

- এখানে আরও একটি উদাহরণ দেয়া হলো ট্যাগ পরিবর্তনের।
- As a rule of thumb, if you want to preserve the state between re-renders, the structure of your tree needs to “match up” from one render to another. If the structure is different, the state gets destroyed because React destroys state when it removes a component from the tree. This is why you should not nest component function definitions.
- যদিও রিঅ্যাক্টের ডিফল্ট আচরণ হচ্ছে রেন্ডার ট্রি তে কম্পোনেন্ট একই থাকলে সে state প্রিজার্ভ করে কিন্তু মাঝেমাঝে আমাদের state রিসেট করা দরকার হতে পারে। যেমনঃ একটা কম্পোনেন্টে একজন প্লেয়ারের নাম শো করবে, তার খেলা হয়ে গেলে দ্বিতীয় খেলোয়াড়ের নাম আসবে। প্রপস পাস করে আমরা খেলোয়াড়ের নাম পরিবর্তন করতে পারি, কিন্তু কাজগুলো তো একটি কম্পোনেন্টের মধ্যেই হবে। তাই এক্ষেত্রে state প্রিজার্ভড থাকবে। কিন্তু ভিন্ন খেলোয়াড়ের জন্য তো আলাদা আলাদা স্কোর হওয়ার কথা যেটা আমরা state এ ধরে রাখতে চাই।

Initial state

Clicking “next”

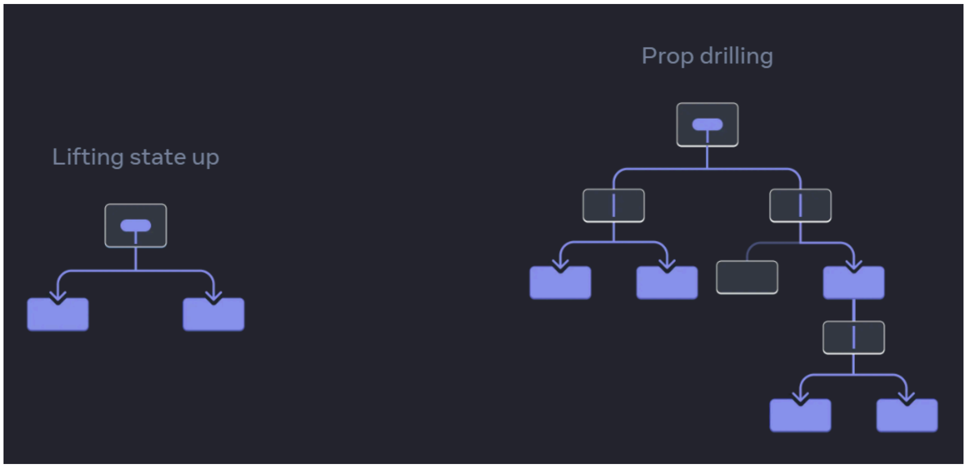
Clicking “next” again

- এক্ষেত্রে আমরা দুইটা পদ্ধতি অনুসরণ করতে পারি। প্রথমটি হচ্ছে কম্পোনেন্টকে ভিন্ন পজিশনে রেন্ডার করা। এখানে আমরা PlayerA কে একবার খেলাচ্ছি, তারপর আবার next চেপে

PlayerB খেলাচ্ছি। ট্রি এর ভিন্ন ভিন্ন জায়গায় তাদের রেন্ডার হচ্ছে। This solution is convenient when you only have a few independent components rendered in the same place.

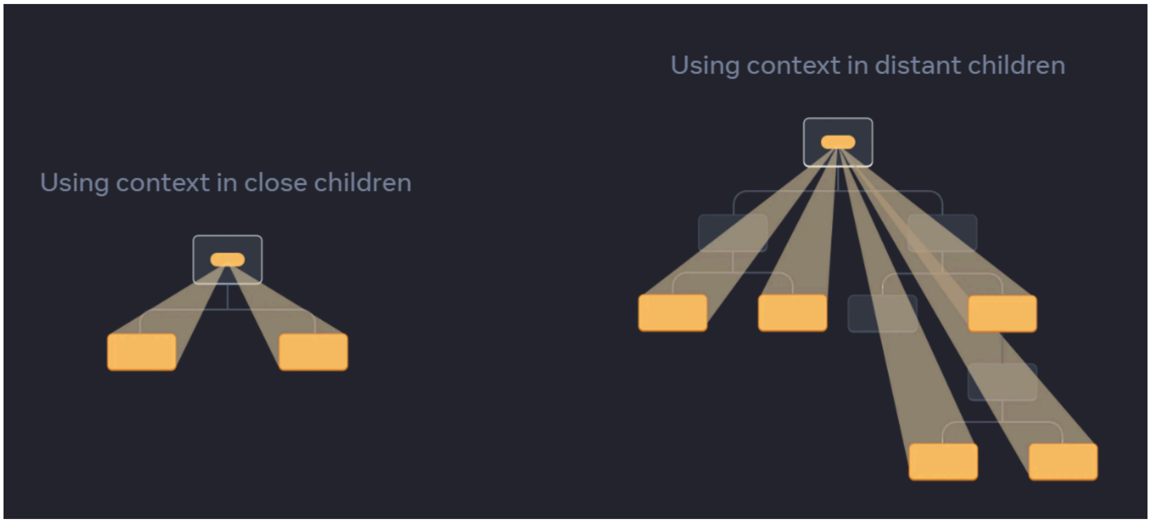
- দ্বিতীয় পদ্ধতি হচ্ছে আমরা কম্পোনেন্টগুলোকে আলাদাভাবে চেনার জন্য key ব্যবহার করতে পারি। Remember that keys are not globally unique. They only specify the position within the parent.
- যদি আমরা রিমুভড কম্পোনেন্টের state প্রিজার্ভ করতে চাই তাহলে দুটি বুদ্ধিতে সেটা করা যায়। প্রথমত, আমরা কোন কম্পোনেন্টকে প্রকৃতপক্ষে রিমুভ করবোইনা বরং CSS দিয়ে তাদেরকে ঢেকে রাখবো, তাহলে তাদের state হারিয়ে যাবেনা। তবে এই বুদ্ধি ছোটখাটো ট্রি এর ক্ষেত্রে প্রযোজ্য। আরেকটা বুদ্ধি হচ্ছে আমরা lifting the state up করে state গুলোকে প্যারেন্টে নিয়ে গিয়ে রেখে দিবো, তাহলে চাইল্ড হারিয়ে গেলেও state হারাবেনা। localStorage এও তথ্য ধরে রাখা যায়।
- Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer. Reducers are a different way to handle state.
- আমরা তিন ধাপে useState থেকে useReducer তে শিফট করতে পারি। নিচে সেগুলোর বিবরণ দেয়া হলো।
- প্রথম ধাপ হচ্ছে “Move from setting state to dispatching actions.” useState ব্যবহারকালে আমরা setState এর মধ্যে আমরা পুরো বিজনেস লজিক দিয়ে দিই। কিন্তু useReducer ব্যবহারকালে আমরা এরকম করবোনা। আমরা শুধু বলে দিবো কি কি ঘটনা ঘটেছে তার একটা সারমর্ম (অবজেক্ট আকারে) এবং দ্বিতীয় ধাপে বিজনেস লজিক আমরা কম্পোনেন্টের বাইরে রিডিউসার ফাংশনের মধ্যে লিখে আসবো। তৃতীয় ধাপে কম্পোনেন্টের মধ্যে useReducer কে ইমপোর্ট করতে হবে।
- useReducer হুক মূলত দুইটি জিনিস রিটার্ন করে, একটি হল Stateful Value যেটি ঠিক useState এর মত কাজ করে, অন্যটি হল dispatch ফাংশন যেটি setState এর মত state এ ভ্যালু সেট করতে সাহায্য করে ।
- useReducer হুক তৈরি করতে আর্গুমেন্ট হিসেবে প্রথমে reducer function দিতে হয় এবং এর পরে initialState দিতে হবে। অনেকটা এভাবে - useReducer(reducer, initialState).
- রিডিউসার ফাংশন লিখার সময় দুটি বিষয় খেয়াল রাখতে হবে, এটাকে অবশ্যই পিউর ফাংশন হতে হবে এবং Each action describes a single user interaction, even if that leads to multiple changes in the data. For example, if a user presses “Reset” on a form with five fields managed by a reducer, it makes more sense to dispatch one `reset_form` action rather than five separate `set_field` actions.

- যখন একটি বা একাধিক State কে পরিবর্তন করার জন্যে অনেক গুলো handler রয়েছে অথবা State আপডেট অনেক কমপ্লেস হবে, সে সময় useReducer ব্যবহার করা ভালো হবে। useState দিয়ে মূলত খুব কম কোড লিখেই State Manage করা যায়।
- Props Drilling বলতে Props কে এক কম্পোনেন্ট থেকে অন্য কম্পোনেন্ট এ পাঠানো। তবে, রিয়্যাক্ট এর ডেটা ফলো শুধু উপর থেকে নিচে যায়, অর্থাৎ আপনি প্যারেন্ট থেকে চাইল্ডে শুধু ডেটা পাঠাতে পারবেন।
- সাধারণত আমরা প্যারেন্ট কম্পোনেন্ট থেকে চাইল্ডে তথ্য পাস করি প্রপস এর মাধ্যমে। কিন্তু ঝামেলাটা বাধে তখন যখন একটা প্যারেন্ট কোন তথ্য তার কোন ডিপলি নেস্টেড চাইল্ডে পাঠাতে চায়। তখন মাঝের অনেকগুলো কম্পোনেন্টের মাধ্যমে ট্রি এর একদম নিচের কোন চাইল্ডে তথ্য পাঠাতে গিয়ে মাঝের কম্পোনেন্টগুলোতেও প্রপসগুলো পাঠাতে হয়, যেটা হয়তো একেবারেই অপ্রয়োজনীয়। মাঝের কম্পোনেন্টগুলোর সেই প্রপস একদমই দরকার নাও হতে পারে। এই সমস্যা আরেকটা ক্ষেত্রেও হতে পারে যখন অ্যাপ্লিকেশনের অনেকগুলো কম্পোনেন্টের একই তথ্য দরকার পড়ে।
- Context আমাদের এই সমস্যা সমাধান করতে পারে প্রয়োজনীয় তথ্যগুলোকে কম্পোনেন্টের বাইরে কোথাও স্টোর করে প্রয়োজনমতো সাপ্লাই দিয়ে।



- "Lifting state up" in React refers to the practice of moving the state from a child component to its parent component. This is done to share the state among multiple components or to manage the state in a higher-level component that has a broader scope.
- Prop drilling in React refers to the process where you pass down props through multiple layers of nested components in order to provide data or functionality to a deeply nested child component. Prop drilling can become a challenge as your component tree grows, making it less maintainable and leading to potential code smell.
- Context is an alternative to passing props. Context lets a parent component provide data to the entire tree below it.
- তিন ধাপে আমরা Context আমাদের কোড বেইজে ইন্ট্রোডিউস করতে পারি।

- Create a context, use that context from the component that needs the data and provide that context from the component that specifies the data.



- Context lets a parent—even a distant one!—provide some data to the entire tree inside of it.
- `useContext` is a Hook. Just like `useState` and `useReducer`.
- মূলত, Props Drill করে করে এক কম্পোনেন্ট থেকে অন্য কম্পোনেন্ট এ পাঠানোর সমস্যার সমাধান করতেই Context API ব্যবহার করা হয়।
- Context API, createContext ফাংশন এর দ্বারা initial value নিয়ে থাকে, initial value হতে পারে কোন Object, Array, String, Number। এমনকি আপনি কোন State কেও initial value হিসেবে দিতে পারেন।
- You can insert as many components as you like between the component that provides context and the one that uses it.
- **Context lets you write components that “adapt to their surroundings” and display themselves differently depending on where (or, in other words, in which context) they are being rendered.**
- How context works might remind you of CSS property inheritance. In CSS, you can specify `color: blue` for a `<div>`, and any DOM node inside of it, no matter how deep, will inherit that color unless some other DOM node in the middle overrides it with `color: green`. Similarly, in React, the only way to override some context coming from above is to wrap children into a context provider with a different value.
- In CSS, different properties like `color` and `background-color` don’t override each other. You can set all `<div>`’s `color` to red without impacting `background-color`. Similarly, different React contexts don’t override each other. Each context that you make with `createContext()` is

completely separate from other ones, and ties together components using and providing that particular context. One component may use or provide many different contexts without a problem.

- Context is very tempting to use! However, this also means it's too easy to overuse it. **Just because you need to pass some props several levels deep doesn't mean you should put that information into context.**
- In general, if some information is needed by distant components in different parts of the tree, it's a good indication that context will help you.
- আমরা useState দিয়েও Simple State Manage করতে পারি । কিন্তু যখন Complex State এবং অসংখ্য Event Handler Manage করার সিচুয়েশন আসে তখন useReducer হুক এ সব কিছু অনেক সহজ করে দেয় ।
- createContext ফাংশন টি কল করার মাধ্যমে একটি Global Store তৈরি হয় যেখানে সকল State বা Data থাকে । আমরা চাইল্ড কম্পোনেন্ট থেকে সেই ডেটা গুলো access করতে পারি ।