State management in Next.js

React State Hook

`useState` is a *hook*. A hook is a special type of function that allows us to "hook into" React internals.

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

  return React.createElement(
    'button',
    { onClick: () => setCount(count + 1) },
    'Value: ',
    count
  );
}
```

When this code runs, `React.createElement` produces a React element, a plain JS object. React elements are essentially descirptions of the UI we want.

We're saying in this case that we want a button that contains the text “`Value: 0`".

```
{
  type: 'button',
  key: null,
  ref: null,
  props: {
    onClick: () => setCount(count + 1),
    children: 'Value: 0',
  },
  _owner: null,
  _store: { validated: false }
}
```

Our React element, that JS object, is describing this DOM structure. React takes that description and turns it into the real thing. It creates a `<button>` DOM node and appends it to the page.

Whenever a state variable is updated, it triggers a re-render.

The new React element describes this DOM structure:

```
<button>Value: 1</button>
```

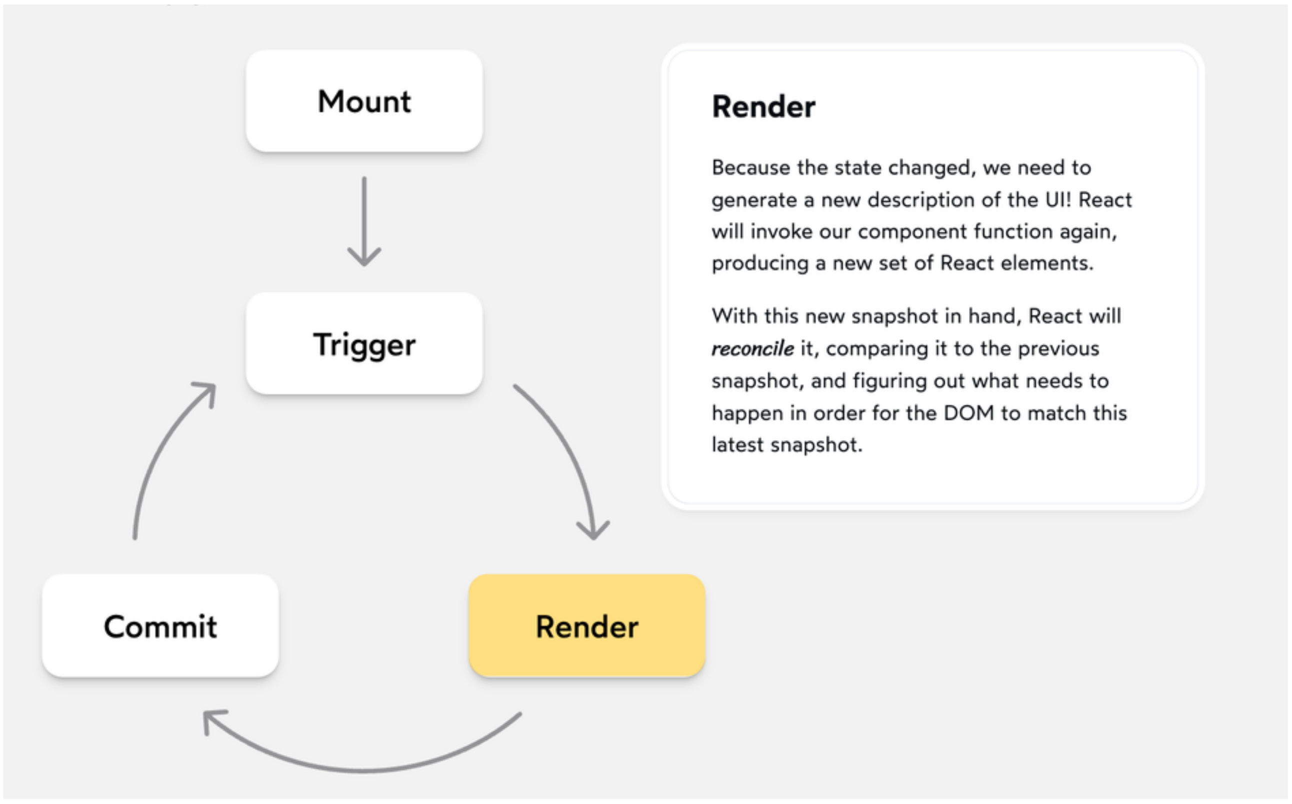
React deals with JS objects that describe this markup. React now must figure out how to *update* the DOM, so that it matches this latest snapshot in that component state.

This process is known as ***reconciliation***. An optimized algorithm, React figures out what's changed. It sees that the button's text content has changed from "`Value: 0`" to "`Value: 1`".

Once React worked out what's different, it will need to ***commit*** these changes **locally**. With surgical precision, it updates the DOM, taking care to only tweak the things that need to be tweaked.

This is the fundamental "flow" of React, the core loop.

React Core Loop



Because the state changed, we need to generate a new description of the UI! React will invoke our component function again, producing a new set of React elements.

With this new snapshot in hand, React will *reconcile* it, comparing it to the previous snapshot, and figuring out what needs to happen in order for the DOM to match this latest snapshot.

Mount: When we render the component for the first time, there is no previous snapshot to compare to. And so, React will create all the necessary DOM nodes from scratch, and inject them into the page.

Trigger: Eventually, something happens that trigger state change, invoking "`set`" function (`setCount`). We're telling React that the value of a state variable has just been updated.

Render: Because the state changed, we need to generate a new description of the UI! React will invoke our component function again, producing a new set of React elements.

With this new snapshot in hand, React will ***reconcile*** it, comparing to the previous snapshot, and figuring out what needs to happen for the DOM to match this latest snapshot.

Commit: If any DOM updates are required, React will perform those mutations (changing the text content of a DOM node, creating new nodes, deleting removed nodes, etc). Once the changes have been **committed**, React goes idle, and waits for the next trigger, the next state change.

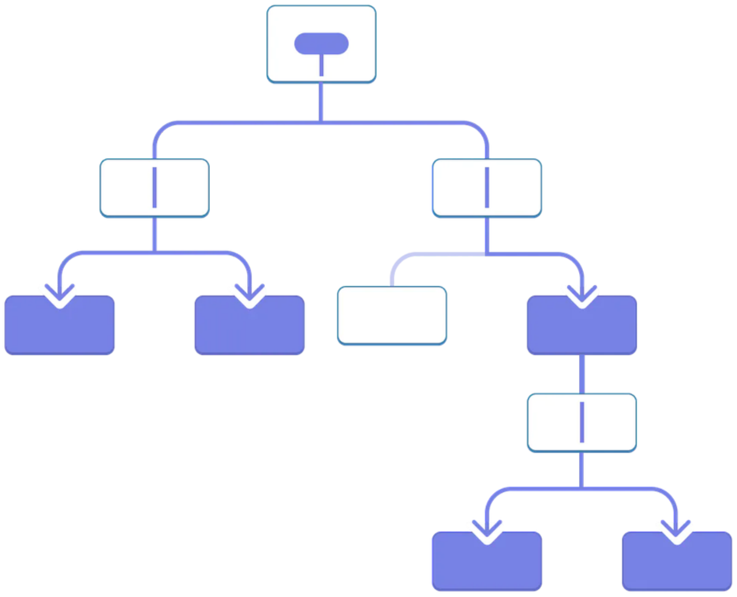
Not all re-renders require re-paints! If nothing has changed between snapshots, React won't edit any DOM nodes, and nothing will be re-painted. Understand, when "re-rendering" we **not** saying that we should throw away the current UI and re-build everything from scratch.

React tries to keep the re-painting to a minimum, because re-painting is slow. Instead of generating a bunch of new DOM nodes from scratch (lots of painting), it figures out what's changed between local **snapshots**, and makes the required tweaks with surgical precision.

React Context API

Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

As passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop.



The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called "**prop drilling**". In such situations, React's Context API can be a savior. The process is broadly categorized into three steps:

Create a Context (`createContext`)

Call `createContext` outside of any components to create a context.

```
import { createContext } from 'react';
const ThemeContext = createContext('light'); // default value
```

`defaultValue`: The value that you want the context to have when there is no matching context provider in the tree above the component that reads context.

If you don't have any meaningful default value, specify `null`. The default value is meant as a "last resort" fallback. It is static and never changes over time.

The return value from `createContext` is a context object.

The context object itself does not hold any information. It represents *which* context other components read or provide. Typically, you will use `SomeContext.Provider` in components to specify context value and call a `useContext(SomeContext)` hook in components to read it.

The context object has a few properties:

- `SomeContext.Provider` lets you provide the context value to components.
- `SomeContext.Consumer` an alternative way to read the context value but preferred is `useContext`.

This is creating a "**container**" that will hold the data you want to share. This "**container**" can be visualized as a conduit that runs through your component tree, making data available where needed.

```
// store/context/pokemonStore.tsx
'use client';

import { useMemo, useState, createContext, useContext, ReactNode } from 'react';

interface Pokemon {
  id: number;
  name: string;
  image: string;
}

// STEP 1: CREATE A CONTEXT
interface ContextProps {
  filter: string;
  setFilter: (filter: string) => void;
  filteredPokemon: Pokemon[];
}

const PokemonContext = createContext<ContextProps | undefined>({
  filter: '',
  setFilter: () => {},
  filteredPokemon: [],
});
```

`ContextProps` describes the shape of the context.

After defining these, a new context called `PokemonContext` is created using `createContext`.

A default value(s) is provided to this context.

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.

Provide the Context (`Provider`)

For the data to flow through this conduit, there needs to be a source or starting point.

```
// store/context/pokemonStore.tsx
// STEP 2: CREATE A PROVIDER
```

```tsx
interface ProviderProps {
  pokemon: Pokemon[];
  children: ReactNode;
}

export const PokemonProvider = ({ pokemon, children }: ProviderProps) => {
  const [filter, setFilter] = useState('');
  const filteredPokemon = useMemo(
    () => pokemon.filter((p) => p.name.toLowerCase().includes(filter)),
    [filter, pokemon]
  );

  return (
    <PokemonContext.Provider value={{ filter, setFilter, filteredPokemon }}>
      {children}
    </PokemonContext.Provider>
  );
};
```

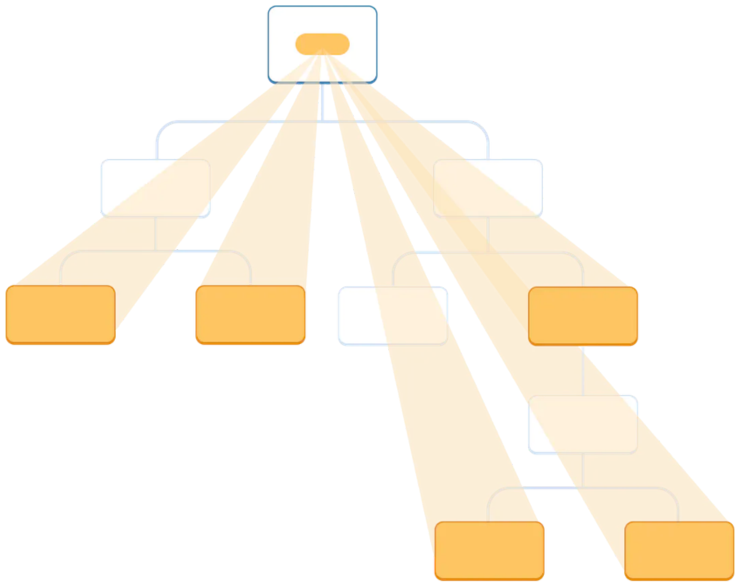
The `Provider` does a couple of things:

1. It takes in a prop `pokemon` which is an array of Pokémon.
2. It initializes a local state for `filter` that's used to input a filter to the Pokémon list.
3. It computes `filteredPokemon` based on the current value of the `filter` and the `pokemon` list.
4. It renders the `PokemonContext.Provider` component and provides the current context value (`filter`, `setFilter`, `filteredPokemon`) to all child components.
5. This is done using and passing into the providers `value` prop.

Next, we wrap a part of your component tree with a special component called a `Provider`.

```tsx
// app/layout.tsx
const getData = async () => {
  const resp = await fetch(
    'https://jherr-pokemon.s3.us-west-1.amazonaws.com/index.json'
  );
  const pokemon: Pokemon[] = await resp.json();
  return pokemon;
};

export default async function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  const pokemon = await getData();
  return (
    <html lang='en'>
      <body className={`${karla.className} bg-slate-300 p-5`}>
        <PokemonProvider pokemon={pokemon}>{children}</PokemonProvider>
      </body>
    </html>
  );
```

This pumps the data into the context so that any child "consumer" inside it can access the data.



The Context API is like a broadcasting system.

The Provider broadcasts data, and any component within its scope can tune in to receive this data, bypassing the traditional prop-passing method. This makes it easier to share global states, configurations, or any form of shared data across a large tree of React components.

Consume the Context (`useContext`)

Components that need access to this shared data can tap into (or "**consume**") this conduit to get the data they need without it being explicitly passed down through props.

```
// Create a custom hook that uses the context
export const usePokemon = () => {
  const context = useContext(PokemonContext);
  if (context === undefined) {
    throw new Error('usePokemon must be used within a PokemonProvider');
  }
  return context;
};
```

A custom hook called `usePokemon` is also defined. This hook allows any component to easily access the context provided by `PokemonProvider`.

1. Inside the `usePokemon` hook, the `useContext` hook is used to get the current value of the context.
2. If the `context` is `undefined`, this means the hook is being used outside of the provider.
3. If everything is in order, it returns the context, allowing any component that uses this hook to access `filter`, `setFilter`, and `filteredPokemon` values.

```
export default function PokemonFilter() {
  const { filter, filteredPokemon, setFilter } = usePokemon();
  return (
    <>
      <input
        type='text'
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        className='p-2 rounded-lg shadow-md placeholder:text-sm'
        placeholder='Filter pokemon by name'
      />
      {/* GRID CONTAINER */}
      <div className='grid grid-cols-4 gap-8'>
        {filteredPokemon.map((p) => (
          <PokemonCard key={p.id} pokemon={p} />
        ))}
      </div>
    </>
  );
}
```

In summary, Context lets a component provide some information to the entire tree below it.

- Create and export it with `export const MyContext = createContext(defaultValue)`.
- Pass it to the `useContext(MyContext)` hook to read it in any child component, no matter how deep.
- Wrap children into `<MyContext.Provider value={...}>` to provide it from a parent.
- Context passes through any components in the middle.
- Context lets you write components that “adapt to their surroundings”.
- Before you use context, try passing props or passing JSX as `children`.

When values in a context change, any component that consumes that context may re-render, depending on how it consumes the context and other optimization mechanisms in place.

Considerations

Components & Re-renders

Context Consumers: Any component that uses `Context.Consumer` or the `useContext` hook to access context values will re-render when the context value changes.

Components outside the Provider: Components that are outside the hierarchy of a particular context provider won't re-render when that context's value changes, as they don't have access to the context.

Child Components of Consumers: If a context-consuming component renders child components and passes the context value down as a prop, those child components will also re-render.

Recommendations

If a context object is very large, and only one property of that object changes, all consumers of that context will still re-render, even if they only use a different property of that object. To mitigate this!

Break Up Contexts: Instead of a large context, have multiple smaller contexts for separate concerns.

Optimize Context Value: Use memorization techniques, such as the `useMemo` hook, to ensure that the context value object doesn't change unless necessary.

Too Many Consumers: If many components in various parts of the app are consuming a context, and the context value changes frequently, you might have excessive re-renders.

Restructuring your component tree to have fewer context consumers. Not all state needs to be global. If only a small part of the application needs certain state, use local component state instead of context. If a functional component re-renders often and you're certain that the render output is the same for the same props, you can wrap the component with `React.memo` to memoize the render output against the props.

If you're doing complex computations in the render method of a context consumer, consider using the `useMemo` hook to cache the results of these computations.

Zustand with Immer

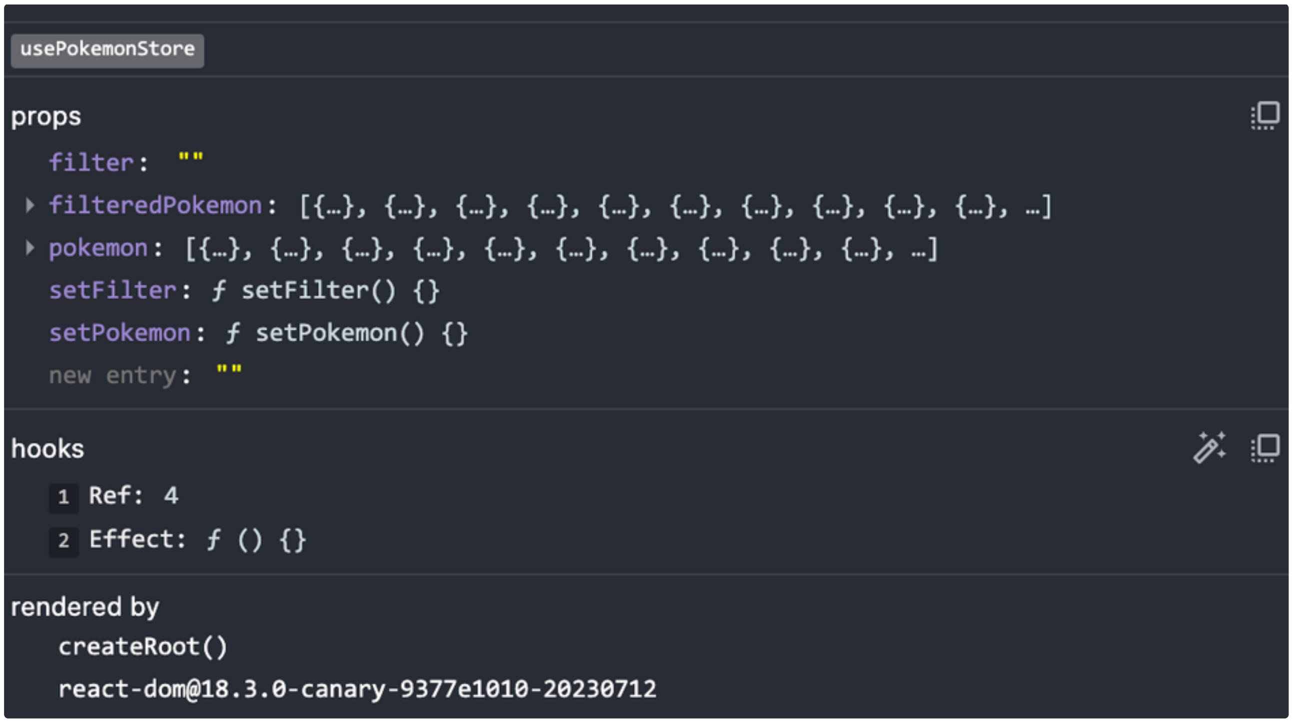
A small, fast, and scalable state management solution that has a comfortable API based on hooks.

`create` function from Zustand:

This is the primary function you use to define and create a Zustand store. It initializes the state and provides mechanisms to update it.

```
import { Pokemon } from '@/app/page';
import { create } from 'zustand';
import { mountStoreDevtool } from 'simple-zustand-devtools';
import { produce } from 'immer';

interface PokemonStore {
  pokemon: Pokemon[];
  filteredPokemon: Pokemon[];
  filter: string;
  setPokemon: (pokemon: Pokemon[]) => void;
  setFilter: (filter: string) => void;
}

const filterPokemon = (pokemonList: Pokemon[], filter: string) => {
  const lowercasedFilter = filter.toLowerCase();
  return pokemonList.filter((pokemon) =>
    pokemon.name.toLowerCase().includes(lowercasedFilter)
  );
};

export const usePokemonStore = create<PokemonStore>((set) => ({
  pokemon: [],
  filteredPokemon: [],
  filter: '',

  setPokemon: (pokemon: Pokemon[]) =>
    set(
      produce((draft) => {
        draft.pokemon = pokemon;
        draft.filteredPokemon = pokemon;
      })
    ),

  setFilter: (filter: string) =>
    set(
      produce((draft) => {
        draft.filter = filter;
        draft.filteredPokemon = filterPokemon(draft.pokemon, filter);
      })
    ),
}));

if (process.env.NODE_ENV === 'development') {
  mountStoreDevtool('usePokemonStore', usePokemonStore);
}
```

This function, provided by `simple-zustand-devtools`, is used to integrate the Zustand store with React development tools for easier debugging and state inspection.


```
usePokemonStore

props
  filter: ""
▸ filteredPokemon: [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, …]
▸ pokemon: [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, …]
  setFilter: ƒ setFilter() {}
  setPokemon: ƒ setPokemon() {}
  new entry: ""

hooks
  1 Ref: 4
  2 Effect: ƒ () {}

rendered by
  createRoot()
  react-dom@18.3.0-canary-9377e1010-20230712
```

`set` function invoked in the callback provided to `create`.

With `set` we update the state just like the setter function from the React state hook. We can provide an object to merge with the current state or a function to produce the new state based on previous state.

`produce` is the primary function you use with **Immer**. It takes two arguments:

The current state and a function in which you can make changes to that state. The function provides you with a `draft` of the state. Inside `produce`, we pass a function that we are given a `draft` of the current state. You can modify this **draft** as if you were working with a regular mutable object. Once the function completes, Immer produces a new, **immutable state** based on the changes you made to the `draft`.

Why Immer?

Immutable updates with mutable syntax: Immer allows you to write code as if you're directly modifying the state object, but under the hood, it produces immutable updates to your object reference types.

```
const state = {
  user: {
    name: 'Alice',
    age: 30,
    address: {
      city: 'Wonderland',
      zip: '12345'
    }
  }
};
// Without Immer:
const updatedState = {
  ...state,
  user: {
    ...state.user,
    address: {
      ...state.user.address,
      city: 'New Wonderland'
    }
  }
};
```

This makes the code more intuitive and readable.

```
const newState = produce(state, draft => {
  draft.user.address.city = 'New Wonderland';
});
```

Safety: With Immer, you don't have to worry about accidentally mutating the state directly, because if you mutate an object, React will be unaware of the change and avoid the render.

```
const [state, setState] = useState({ count: 0 });

function increment() {
  state.count += 1;
  setState(state);
}
```

When managing state in React (or in many other frameworks), it's crucial to avoid direct mutations. Direct mutations can lead to unpredictable behavior, especially in React, which uses a **reconciliation** algorithm to determine when and what to re-render based on a snapshot. As reconciliation relies on reference just equality checks to quickly determine if parts of the virtual DOM need updating. Again, if you mutate an object directly, React can be unaware of the change, and avoid a needed render.

If you mutate a state object directly, these checks can fail, leading to the UI not updating correctly.

```
import produce from 'immer';

const [state, setState] = useState({ count: 0 });

function increment() {
  const newState = produce(state, draft => {
    draft.count += 1;
  });
  setState(newState);
}
```

It ensures all updates are immutable.

Immutability ensures that we don't accidentally introduce side effects or bugs by mutating state directly. However, writing immutable code, especially for nested structures, can be verbose and hard to read.

Important, In the JavaScript (and by extension, React), objects are **reference types**. This means that when you create an object, what you're really getting is a reference to a location in memory where that object's data is stored. When you compare two objects using `===`, you're comparing their references, not their content.

So, even if two objects have the exact same properties and values, they are considered different if they reference separate locations in memory. This is why `{id: 1} === {id: 1}` returns `false`: each object literal creates a new object in memory, and thus, a new reference.

When comparing two objects using `===`, their references (i.e., memory locations) are compared, not their content. Therefore, even if two objects have identical properties and values, they are considered distinct if they were created separately. This is why `{id: 1} === {id: 1}` returns `false`, as each object literal results in a new, distinct memory reference.

When you mutate a property of an object in React state directly, you're changing the value inside the object, but the reference to the object itself remains unchanged. Reacts reactivity system relies on reference comparisons to determine if something has changed to re-render. If the reference hasn't changed, React assumes the object hasn't changed either, even if internal properties have.

> This behavior can lead to bugs and unexpected behavior in your application because the UI won't reflect the updated state. This is why it's crucial in React to always update state immutably. When you need to update an object in the state, you should create a new object with the updated values. This ensures that React sees a new reference and knows to re-render the component.
>
> `setData(prevData => ({ ...prevData, id: 2 }))`
>
> Immer abstracts away the need for manual immutable updates using spread operators or other verbose methods. Instead, it allows you to write code that looks like you're directly mutating the state, but under the hood, it produces a new, immutable state for you.

Reduced Boilerplate: Especially with nested structures, making immutable updates can require a lot of spread operators and can become verbose. Immer reduces this boilerplate.

Efficiency: Immer uses a technique called structural sharing. This means that only the parts of the state that change is copied and updated, while the rest are shared between the old and new state.

Modifying store state (`setState`)

Zustand is minimalistic state management that doesn't rely on React context API, unlike other libraries. This design choice gives Zustand some advantages when it comes to (SSR) in frameworks like Next.

Isolation from React Lifecycle: Zustand stores are independent of React component lifecycle. They are just JS objects and functions. This means they don't have issues related to mounting or unmounting in an SSR. Using `setState` in RSC works, because it's not a hook, unlike its `useStore` hook.

```
import PokemonFilter from '@/components/PokemonFilter';
import { usePokemonStore } from '@/store/zustand/usePokemonStore';

export interface Pokemon {
  id: number;
  name: string;
  image: string;
}

const getData = async () => {
  const resp = await fetch(
    'https://jherr-pokemon.s3.us-west-1.amazonaws.com/index.json'
  );
  const pokemon: Pokemon[] = await resp.json();
  return pokemon;
};

export default async function Home() {
  const pokemon = await getData();
  usePokemonStore.setState({ pokemon: pokemon });
  return (
    <main className='flex flex-col gap-8'>
      <PokemonFilter pokemon={pokemon} />
    </main>
  );
}
```

No React context dependency: Since Zustand doesn't rely on context API for its core functionality, you don't need to wrap your app in a `Provider` component.

This eliminates a common source of errors in SSR environments.

No direct dependency on browser API: Zustand doesn't inherently depend on browser-specific APIs for its core functionality. Its more SSR-friendly, as it won't break when code is executed on the server.

That said, while the core of Zustand works well with SSR in Next.js, there are considerations:

Initial state on the server: If you want your Zustand store to have an initial state derived from server-side data in Next.js, you'll need to set up a mechanism to do that. This often involves populating the store on the server and then passing that state to the client.

Select state to consume only: you select specific pieces of state or functions from the store using a selector function, is often called **state selection** or **selector pattern**.

Zustand supports this pattern, allowing you to pick only the parts of the state (or associated actions) that you're interested in. This can help optimize re-renders in React components.

```
xport default function PokemonFilter({ pokemon }: { pokemon: Pokemon[] }) {
  const { filter, filteredPokemon, setFilter, setPokemon } = usePokemonStore(
    (state) => ({
      filter: state.filter,
      filteredPokemon: state.filteredPokemon,
      setFilter: state.setFilter,
      setPokemon: state.setPokemon,
    })
  );
  useEffect(() => {
    setPokemon(pokemon);
  }, [pokemon]);
  return (
    <>
      <input
        type='text'
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        className='p-2 rounded-lg shadow-md placeholder:text-sm'
        placeholder='Filter pokemon by name'
      />
      {/* GRID CONTAINER */}
      <div className='grid grid-cols-4 gap-8'>
        {filteredPokemon.map((p) => (
          <PokemonCard key={p.id} pokemon={p} />
        ))}
      </div>
    </>
  );
}
```

The benefit of this pattern is that your component will only re-render when the pieces of state selected change. If other parts of the Zustand store change, but they weren't selected in your consuming component, it won't cause a re-render. It can improve performance optimizations, especially in larger applications with frequently updating state.

Synchronizing Client Stores

In the context of Next.js, especially with the introduction of React Server Components (RSC), managing state synchronization between the server and client becomes crucial.

The pattern described below offers a solution to this challenge using Zustand.

Initial State on the Server:
The Zustand store (`usePokemonStore`) is first initialized on the server, specifically within the `Home` page component, as it's an RSC component.

Client-side Initialization:
The store isn't initialized on the client-side until a `useEffect` hook is triggered.

This effectively mounts a client-side version of the same store. The data persisted in this client-side version ensures that the state remains consistent across both server and client.

So, can we improve this experience? Here we have a component just for that reason.

```
'use client';

import { useEffect, useRef } from 'react';
import { usePokemonStore } from '@/store/zustand/usePokemonStore';

const STORE_MAP = {
  pokemon: usePokemonStore,
  // ... add other stores as needed
};

interface Props {
  storeName: keyof typeof STORE_MAP;
  data: Record<string, any>;
}

export default function StoreInitializer({ storeName, data }: Props) {
  const initialized = useRef(false);
  const store = STORE_MAP[storeName];
  useEffect(() => {
    // only initialize the store once with the initial data :)
    if (!initialized.current && store) {
      store.setState({ ...data });
      initialized.current = true;
    }
  }, [store, data]);

  return null;
}
```

One of the significant advantages of this pattern is the granular control it offers. With the advent of Next.js version 13, developers can decide at a granular level which state pieces are essential for the client. This ensures that only relevant data is sent, optimizing performance, and reducing unnecessary data transfer.

The use of `useRef` in the `StoreInitializer` component ensures that the Zustand store is initialized only once. This prevents potential issues like unnecessary re-renders or overwriting of state.

```
export default async function Home() {
  const initialData = await getData();
  return (
    <main className='flex flex-col gap-8'>
      <StoreInitializer
        storeName='pokemon'
        data={{ pokemon: initialData, filteredPokemon: initialData }}
      />
      <PokemonFilter initialData={initialData} />
    </main>
  );
}
```

Server Components vs. Client Components:
It's crucial to understand the distinction between React Server Components and Client Components in the Next.js framework. While Server

Components facilitate efficient server-side rendering without sending associated JavaScript to the client, Client Components operate on the client, allowing for interactivity. This clear separation is foundational for state synchronization patterns like the one described above.

Advantages

Small & fast: Zustand doesn't require much setup. Here's how you can create a store and use it:

```
import create from 'zustand';

const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 }))
}));

// In a component:
const count = useStore(state => state.count);
```

Scalable: You can easily split your stores as your app grows:

```
// userStore.js
export const useUserStore = create(set => ({
  user: null,
  setUser: user => set({ user })
}));

// themeStore.js
export const useThemeStore = create(set => ({
  theme: 'light',
  toggleTheme: () => set(state => ({ theme: state.theme === 'light' ? 'dark' : 'light' }))
}));
```

Flux principles: Zustand maintains a unidirectional data flow:

```
const useStore = create(set => ({
  todos: [],
  addTodo: todo => set(state => ({ todos: [...state.todos, todo] }))
}));

// Action -> Store -> View
useStore(state => state.addTodo)('New Todo'); // Action
```

Hooks-based API: Zustand integrates seamlessly with React components:

```
function Counter() {
  const count = useStore(state => state.count);
  const increment = useStore(state => state.increment);

  return (
    <div>
      {count}
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Not lots of boilerplate: Compared to Redux:

```
// Redux:
const incrementAction = { type: 'INCREMENT' };
```

```javascript
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT': return state + 1;
    default: return state;
  }
}

// Zustand:
const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 }))
}));
```

Why Zustand over Redux?

Simple and un-opinionated: Redux has a specific way of handling actions, reducers, and middleware. Zustand lets you suit your application without enforcing a specific pattern.

```javascript
// Redux:
const incrementAction = { type: 'INCREMENT' };
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT': return state + 1;
    default: return state;
  }
}

// Zustand:
const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 }))
}));
```

Hooks are the primary for consuming state: Zustand leverages hooks as the primary way to interact with state, making it feel more integrated with modern React patterns.

```javascript
function Counter() {
  const count = useStore(state => state.count);
  return <div>{count}</div>;
}
```

Doesn't wrap your App in providers: Redux and context requires you to wrap your app in a `<Provider>` to make the store available. Zustand doesn't have this requirement, making the setup cleaner.

```javascript
// Redux:
<Provider store={store}>
  <App />
</Provider>

// Zustand:
// No provider needed!
<App />
```

Inform components transiently (without any render): Zustand allows for in-place updates, which means you can update state without causing any re-render. This can be useful for non-visual updates.

```javascript
const useStore = create((set, get) => ({
  logs: [],
  addLog: (log) => set(state => ({ logs: [...state.logs, log] }), false) // The second argument as "false" prever
}));
```

Why Zustand over Context?

Less Boilerplate: React Context often requires creating context, providers, and consumers. Zustand simplifies this with a single hook.

```
// Context:
const CountContext = React.createContext();
<CountContext.Provider value={count}>
  ...
</CountContext.Provider>

// Zustand:
const useStore = create(set => ({ count: 0 }));
```

Renders components only on changes: React Context will re-render consumers even if the part of the context they're interested in hasn't changed.

Zustand ensures components only re-render when the specific state they're subscribed to changes.

```
// With Zustand, if `count` doesn't change, this component won't re-render:
function Counter() {
  const count = useStore(state => state.count);
  return <div>{count}</div>;
}
```

Centralized, Action-Based State Management: Zustand allows for a centralized store with clear actions, like Redux but without the boilerplate. This can make state updates more predictable.

```
const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 })),
  decrement: () => set(state => ({ count: state.count - 1 }))
}));
```

In summary, while Redux and context have their own strengths and use cases, Zustand offers a simpler, more modern, and efficient approach in many scenarios.

Why Zustand over Flux?

Flux is an architectural pattern introduced by Facebook to address some challenges in building client-side web applications. The core idea behind Flux is to have a unidirectional data flow.

Zustand shares some similarities with Flux.

Unidirectional data flow:

In Flux, the data flow is unidirectional: `Actions -> Dispatcher -> Store -> View`. This ensures that the system remains predictable as it scales. Zustand, while not enforcing a strict flow, encourages a similar pattern where actions update the store, and the store updates the view.

Centralized store:

Flux applications have centralized stores that contain the application's entire state. Zustand also promotes a centralized store, though it's flexible enough to allow for multiple smaller stores if needed.

State is read-only:

In Flux, the state in the store is read-only. Changes to the state are made via actions.

Zustand also encourages this pattern. You don't modify the state directly. Instead, you use setter functions or actions to produce a new state.

Actions:

In Flux, actions are dispatched to signal a state change. These actions are processed by the store to produce a new state. In Zustand, you can define functions (akin to actions) within the store that, when called, update the state. Example to illustrate the Flux-like nature of Zustand:

```
// Flux Action:
const action = { type: 'INCREMENT' };

// Flux Store:
class CounterStore extends Flux.Store {
  constructor() {
    super(dispatcher);
    this.state = { count: 0 };
  }
  reduce(state, action) {
    switch (action.type) {
      case 'INCREMENT':
        return { count: state.count + 1 };
      default:
        return state;
    }
  }
}
```

While Zustand is influenced by Flux principles, it's worth noting that it doesn't strictly enforce them. In both cases, you have a clear action (or function in Zustand) that updates the state.

```
// Zustand Store:
const useCounterStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 }))
}));
```

It provides a more flexible and lightweight approach, making it easier to set up and use while still benefiting from the predictability and structure that Flux introduced.

React Query

Both prefetching approaches, using `initialData` or `<Hydrate>`, are available within the `app` directory.

Prefetching Approaches:

React Query in Next.js offers two primary prefetching approaches:

1. **Using** `initialData`:
   - The Next.js approach of prefetching data in a RSC and passing it down to RCC via props.
   - May require prop drilling through multiple layers of Client Components using the same query.
   - Query re-fetching is based on the page load time, not when the data was prefetched on the server.
2. **Using** `<Hydrate>`:
   - Prefetch a query on the server, `dehydrate` cache and `rehydrate` it on the client with `<Hydrate>`
     - Requires slightly more setup up front
     - No need to prop drill the query results

- Query re-fetching is based on when the query was prefetched on the server

Note: Both approaches need a `<QueryClientProvider>`.

Setting up the provider (`QueryClientProvider`)

The hooks from the `react-query` package must retrieve a `QueryClient` from their context. Ensure you wrap your component tree with `<QueryClientProvider>`.

```
import { QueryClient } from '@tanstack/react-query';

'use client';

import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import React from 'react';

export default function QueryProviders({ children }: any) {
  const [queryClient] = React.useState(() => new QueryClient());

  return (
    <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
  );
}
```

First create a request-scoped singleton instance of `QueryClient`. This ensures that data is not shared between different users and requests, while still only creating the `QueryClient` once per request. Additionally, we have single place where we can modify global configuration for the client.

```
import { QueryClient } from '@tanstack/query-core';
import { cache } from 'react';

export const getQueryClient = cache(
  () =>
    new QueryClient({
      defaultOptions: {
        queries: {
          refetchOnMount: false,
          refetchOnWindowFocus: false,
          refetchOnReconnect: false,
          retry: false,
        },
      },
    })
);
```

Remember to update your (`QueryProviders`) to use the singleton.

Wrapping your main (`RootLayout`)

```
import QueryProviders from '@/utils/QueryProviders';
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang='en'>
      <body>
        <QueryProviders>
          {children}
          <ReactQueryDevtools initialIsOpen={false} />
        </QueryProviders>
      </body>
    </html>
  );
}
```

Using your RSC (`initialData`)

You can still follow the same fetch of your initial data in a (**RSC**) Server Component higher up in the component tree. We then can pass it to the Client Component as a prop.

```
export interface Pokemon {
  id: number;
  name: string;
  image: string;
}

export const getData = async () => {
  const resp = await fetch(
    'https://jherr-pokemon.s3.us-west-1.amazonaws.com/index.json'
  );
  const pokemon: Pokemon[] = await resp.json();
  return pokemon;
};

export default async function Home() {
  const initialData = await getData(); // initial server-side data
  return (
    <main className='flex flex-col gap-8'>
      <Hydrate state={dehydratedState}>
        <PokemonFilter pokemon={initialData} />
      </Hydrate>
    </main>
  );
}
```

We then can supply our (**RSC**) initial data as props to the `useQuery` on the client.

```
export default function PokemonFilter(props: { pokemon?: Pokemon[] }) {
  const { data } = useQuery({
    queryKey: ['pokemon'],
    queryFn: getData,
    initialData: props.pokemon,
  });
  const [filter, setFilter] = useState('');
  const filteredPokemon = useMemo(
    () => data?.filter((p) => p.name.toLowerCase().includes(filter)),
    [filter, data]
  );
  return (
    <>
      <input
```

```
        type='text'
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        className='p-2 rounded-lg shadow-md placeholder:text-sm'
        placeholder='Filter pokemon by name'
      />
      {/* GRID CONTAINER */}
      <div className='grid grid-cols-4 gap-8'>
        {filteredPokemon?.map((p) => (
          <PokemonCard key={p.id} pokemon={p} />
        ))}
      </div>
    </>
  );
}
```

Why `useQuery` on the client-side after fetching data in a Server Component (Home page component)?

This may seem redundant at first, but there are several reasons why this approach is beneficial:

Client-side Re-fetching: Even if initial data is fetched on the server, there may be situations where you need to re-fetch after the initial render. Like in response to user interaction or when data changes.

Data Synchronization: `useQuery` provides a consistent way to sync data between the client and the server. If data changes on the server after an initial fetch, `useQuery` helps keep client data up to date.

Caching & Stale Data: `useQuery` auto manages caching. When `initialData` prefetched data is used as the initial cache, but subsequent requests (if any) benefit from client cache. `useQuery` can mark data as stale and re-fetch if needed, ensuring UI always reflects most recent data.

Loading & Error States: `useQuery` provides `isLoading`, `isError`, and other utility states out-of-the-box. Even if you have prefetched data, these states can be useful for subsequent interactions.

Consistency in Data Fetching Patterns: Using `useQuery` consistently across components, whether the initial data comes from the server or not, provides a uniform data-fetching pattern. This can make the codebase easier to understand and maintain.

Optimistic Updates & Mutations: If you use `react-query` for mutations (`useMutation`), it pairs very well with `useQuery` for things like optimistic updates. If you already have your data fetching set up with `useQuery`, integrating mutations becomes more straightforward.

Above, by using `useQuery` in the `PokemonFilter` component, you're setting up a pattern that allows for all the advantages mentioned above. Even if the initial render doesn't require a client-side fetch because of the prefetched `initialData`, any subsequent data needs (like re-fetching, synchronization, etc.) can leverage the established `useQuery` pattern.

Taking it a step further with (`<Hydrate>`)

So, we fetched data in the Server Component **higher up in the component tree** than the Client Components, can now use these prefetched queries.

Hydration in the context of Next.js and `react-query` is a process where the data that was fetched on the server (and is now part of your server-rendered page) gets "hydrated" or absorbed into the client-side application so that you don't need to refetch it when the React client takes over. Here we use `dehydrate` and `<Hydrate>` functionality from `react-query` to provide a seamless experience for this.

```
import PokemonFilter from '@/components/PokemonFilter';
import { getQueryClient } from '@/utils/getQueryClient';
import { Hydrate, dehydrate } from '@tanstack/react-query';

export interface Pokemon {
  id: number;
  name: string;
  image: string;
}

export const getData = async () => {
  const resp = await fetch(
    'https://jherr-pokemon.s3.us-west-1.amazonaws.com/index.json'
  );
  const pokemon: Pokemon[] = await resp.json();
  return pokemon;
};

export default async function Home() {
  const initialData = await getData(); // initial server side data
  const queryClient = getQueryClient();
  await queryClient.prefetchQuery(['prefetched-hydrated-list'], getData);
  const dehydratedState = dehydrate(queryClient);
  return (
    <main className='flex flex-col gap-8'>
      <Hydrate state={dehydratedState}>
        <PokemonFilter pokemon={initialData} />
      </Hydrate>
    </main>
  );
}
```

You can fetch inside multiple Server Components, and you can use `<Hydrate>` in multiple places

When discussing hydration, it's all about synchronizing server-side fetched data with the client-side, particularly within frameworks that support (SSR) or (SSG) like Next.js. It's a way to make the data already fetched on the server immediately available to the client-side without a new network request.

Why would Hydration be necessary?

Immediate Availability: When a user first loads the page, the HTML delivered is generated from the server-side. This means that all the data is present, and the page can be displayed instantly without waiting for additional network requests.

Client-side Navigation: As users navigate through a part of your React components that make data requests, it's more efficient for the client-side code to have immediate access to a cache of data that's already been fetched. This avoids unnecessary repeated network requests as a user interacts.

Consistency Between Server and Client: Without hydration, there's a potential mismatch between what the server rendered (`initialData`) and what the client sees after it initializes. This mismatch can lead to "flashes" of old content or unexpected re-renders as the client-side code "catches up" to server state.

Hydration ensures that the first client-side render matches the server's render exactly.

Optimization and Caching on Client: Even though the server has its caching mechanism, client-side libraries like `react-query` provide benefits, such as:

1. Stale data re-fetching
2. Data synchronization across components
3. Retries on failed requests
4. Optimistic updates

To leverage these features, the client needs to be aware of the data. Hydration provides the initial dataset to the client, and from there, the client can handle the data as it sees fit.

Changes Over Time: The data on the server might become stale over time, especially if it's a long-lived page or if the user has kept the page open for an extended period. By having the data in the client's cache (thanks to hydration), tools like `react-query` can smartly refetch when necessary, ensuring a user sees up-to-date data without having to manually refresh the page.

In summary, while the server provides the initial data and does so efficiently, once the control is handed over to the client, it's the client's responsibility to manage and update this data.

Hydration allows a client to start with the exact **snapshot** of data the server used, and then manages cache, and updates that data as needed.

Let's break down the hydration process step by step:

Retrieve from the Singleton Instance
You'll get this from the utility function (like `getQueryClient`) that we set up. This instance is used both on the server-side to prefetch data, and on the client side to access the cache.

```
const queryClient = getQueryClient();
```

Prefetch Data
On the server, before rendering your component, prefetch data using the `prefetchQuery` method.

```
await queryClient.prefetchQuery(['yourQueryKey'], fetchDataFunction);
```

Dehydrate the State
Once your data is fetched, "`dehydrate`" or serialize client's state, preparing it to be sent to the client.

```
const dehydratedState = dehydrate(queryClient);
```

Wrap a Component or Component Tree
In your component that gets sent to the client, wrap the portion of the component tree that needs access to the prefetched data in the `<Hydrate>` component.

```
<Hydrate state={dehydratedState}>
  <YourComponent />
</Hydrate>
```

By passing `dehydratedState` to `<Hydrate>` state prop, we are telling `react-query` on the client side: "Here's data that we fetched on the server, use this first before trying to re-fetch anything."

Multiple Server Components and Multiple Hydrate Usages:
It's completely viable to have multiple (**RSC**) server components each doing its own data fetching and then hydrating parts of your client-side. Each component would have its own `dehydrated` state. The key is to ensure that each `<Hydrate>` component is provided the correct dehydrated state.

Once your page is on the client, `react-query` will check its cache before making any new fetch calls. If the data is already in the cache (**thanks to the hydration process**), it won't re-fetch. If the data isn't in the cache or is considered stale, then it'll trigger a re-fetch as needed.

Accessing Hydrated Data (`useQuery`)

During server rendering, nested `useQuery` calls within the `<Hydrate>` Client Component can now access prefetched data provided in the state property.

The advantage is that `useQuery` does not care if the date is prefetched or not. If it has been already fetched then it will be avaiable to `useQuery` straight away, otherwise it will do the fetch at that point in time and cache the result at the `setQueryClient` parameters.

```tsx
export default function PokemonFilter(props: { pokemon?: Pokemon[] }) {
  // Prefetched Data: Using the initialData prop, the useQuery hook can start
  // with data that was fetched server-side and passed down to the client. This
  // is a useful pattern for speeding up perceived load times because the client
  // has data to display immediately while it verifies the freshness of that
  // data in the background and fetches any update changes.
  const { data } = useQuery({
    queryKey: ['prefetched-hydrated-list'],
    queryFn: getData,
    initialData: props.pokemon,
  });

  // Client-only Fetch: The second useQuery doesn't prefetch the data and will
  // start fetching it only when the component mounts on the client. This is
  // typical for data that isn't as crucial for the initial render or for when
  // SSR (Server-Side Rendering) isn't being used.
  const { data: otherData } = useQuery({
    queryKey: ['client-only-hydrated-list'],
    queryFn: getData,
    initialData: props.pokemon,
  });

  const [filter, setFilter] = useState('');
  const filteredPokemon = useMemo(
    () => data?.filter((p) => p.name.toLowerCase().includes(filter)),
    [filter, data]
  );
  return (
    <>
      <input
        type='text'
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        className='p-2 rounded-lg shadow-md placeholder:text-sm'
        placeholder='Filter pokemon by name'
      />
      {/* GRID CONTAINER */}
      <div className='grid grid-cols-4 gap-8'>
        {filteredPokemon?.map((p) => (
          <PokemonCard key={p.id} pokemon={p} />
        ))}
      </div>
    </>
  );
}
```

SSR | TanStack Query Docs