Front-End Development with React

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

ABOUT ME

Jonas Bandi
jonas.bandi@ivorycode.com
Twitter: @jbandi

- Freelancer, in den letzten 9 Jahren vor allem in Projekten im Spannungsfeld zwischen modernen Webentwicklung und traditionellen Geschäftsanwendungen.
- Dozent an der Berner Fachhochschule seit 2007
- In-House Kurse & Beratungen zu Web-Technologien im Enterprise: UBS, Postfinance, Mobiliar, AXA, BIT, SBB, Elca, Adnovum, BSI ...

letsboot.ch
swiss dev training

digicomp

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

JavaScript / Angular / React / Vue / Vaadin
Schulung / Beratung / Coaching / Reviews

jonas.bandi@ivorycode.com

Hooks

Advantages of Hooks

Hooks *embrace JavaScript* closures and avoid introducing React-specific APIs where JavaScript already provides a solution.

Reduce complexity in components.

Enable reuse:
Logic can easily be decoupled from components and shared among components.
Hooks are composable: new Hooks can be created by composing other Hooks.

Favoring *composition* over inheritance.

Why Hooks (2023)?

Because the React ecosystem has embraced Hooks.

All modern React libraries expose their API via Hooks:

- ReactRouter / ReactLocation
- MaterialUI
- MobX, ReactRedux, Recoil ...
- ReactQuery
- react-i18next
- ...

Basic Hooks

useState
useEffect
useRef
useContext
useReducer

useImperativeHandle
useSyncExternalStore

useTransition
useDeferredValue

useMemo
useCallback

useLayoutEffect
useInsertionEffect
useDebugValue
useId

https://react.dev/reference/react/hooks
https://react.dev/reference/react-dom/hooks

The Rules of Hooks

https://react.dev/reference/rules/rules-of-hooks

https://en.wikipedia.org/wiki/Fight_Club_(novel)

The Rules of Hooks

(a part of the *Rules of React*)

Hooks are JavaScript functions, but you need to follow two rules when using them.

1. Only Call Hooks at the Top Level
   (don't call Hooks inside loops, conditions, or nested functions)
2. Only Call Hooks from React Functions
   (from function components or custom hooks)

Use the ESLint plugin to enforce these two rules:
https://www.npmjs.com/package/eslint-plugin-react-hooks

Rules of Hookes: https://react.dev/reference/rules/rules-of-hooks
Rules of React: https://react.dev/reference/rules
https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e

EXERCISE

Exercises: Hooks - 1 to 5

Drawbacks of Hooks

- Stale Closure Problems
  https://epicreact.dev/how-react-uses-closures-to-avoid-bugs/

- The "Rules of Hooks" (it's not "just a function" when there are rules ...)

- Hooks can be "viral": Hooks can only be called from Hooks
  -> calling library APIs only from Hooks => finally everything is a Hook

DOM Access with `useRef`

React abstracts the real DOM behind the component tree and the virtual DOM. But sometimes you need access to real DOM elements.

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  function buttonClicked() {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={buttonClicked}>Focus the input</button>
    </>
  );
}
```

`useRef()` combined with the `ref` attribute can be used to access DOM nodes.

https://react.dev/reference/react/useRef#examples-dom

However, `useRef()` can be used for keeping any mutable value for the full lifetime of the component (a "replacement" for instance fields in classes)

https://react.dev/reference/react/useRef

forwardRef

... enables a custom component to expose a wrapped DOM node to its parent

```
interface MyInputProps{ ... }

function MyInputImpl(props: MyInputProps, ref: Ref<HTMLInputElement>) {
  return <input ref={ref} />;
});

const MyInput = forwardRef<HTMLInputElement, MyInputProps>(MyInputImpl);
```

`forwardRef` and TypeScript is sometimes tricky ...

Note: in React v19 a `ref` can be passed as a `prop`.
`forwardRef` will not be needed any more and will be deprecated in the future.

https://react.dev/reference/react/forwardRef
https://react.dev/blog/2024/04/25/react-19#ref-as-a-prop

useImperativeHandle

useRef can't give access to custom function components since they are functions not instances ...
But with useImperativeHandle a custom component can expose imperative methods to its parent

```
function MyInputImpl(props: MyInputProps, ref: Ref<MyInputRef>) {
  const inputRef = useRef<HTMLIputElement>(null);

  useImperativeHandle(ref, () => {
    return {
      focus() {
        inputRef.current.focus();
      },
    };
  }, []);

  return <input ref={inputRef} />;
});
const MyInput = forwardRef<MyInputRef, MyInputProps>(MyInputImpl);
```

https://react.dev/reference/react/useImperativeHandle

DEMO:

- `forwardRef` for exposing inner DOM node
- `ref` to custom component (does not work)
- `useImperativeHandle` for exposing imperative methods

React Context & useContext

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Providing a context

```
export const MyContext = React.createContext('my-context');
```

```
import {MyContext} from '../MyContext';
...
const contextObject = ...
<MyContext.Provider value={contextObject}>
    <App />
</MyContext.Provider>
```

an object is provided somewhere in the component tree

consuming a context

```
import React, {useContext} from 'react';
import {MyContext} from '../MyContext';
...
function MyComponent(){
    const contextObject = useContext(MyContext);
    // do something with the context object
}
```

the object can be consumed deeper down in the component tree

Note: in React v19 a context can be consumed with use (a new API which is not a Hook) https://react.dev/reference/react/use

https://react.dev/learn/passing-data-deeply-with-context
https://react.dev/learn/scaling-up-with-reducer-and-context

Note: historically there have been other ways to consume a context via render props and "injection" https://reactjs.org/docs/context.html

Demo: Context

useReducer

https://react.dev/reference/react/useReducer

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

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

https://react.dev/learn/extracting-state-logic-into-a-reducer#comparing-usestate-and-usereducer

Background: Reduce

```
var a = [1,2,3,4,5];

var result = a.reduce(
  // reducer function
  (acc, val) => {
    const sum = acc.sum + val;
    const count = acc.count + 1;
    const avg = sum/count;
    return {sum, count, avg};
  },
  // state object
  {sum:0, count:0, avg:0}
);

console.log('Statistics:', result);
```

The reducer function is a pure function.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

State Reducer Pattern

State changes are modelled with a reducer function.
With this pattern state updates can be consolidatedoutside of components.

(old state , action) => new state

aka: "reducer function"

Redux made this pattern popular: https://redux.js.org/

EXERCISE

Exercise: Hooks

Custom Hooks

"Hooks are just functions."
"Hooks can be composed from other hooks."

DEMO:

- simple wrapper around useState
- useToggle
- useState with localStorage persistence
- useTodoId in ToDo app router solution

Hook Libraries

https://github.com/streamich/react-use
https://usehooks.com/
https://github.com/antonioru/beautiful-react-hooks
https://usehooks-ts.com/
https://github.com/palmerhq/the-platform
https://github.com/rehooks/awesome-react-hooks
https://github.com/alibaba/hooks
https://nikgraf.github.io/react-hooks/
https://observable-hooks.js.org/
https://crutchcorn.github.io/rxjs-use-hooks/
...

Examples of custom Hooks:

- https://rangle.io/blog/simplifying-controlled-inputs-with-hooks/
- https://overreacted.io/making-setinterval-declarative-with-react-hooks/
- https://upmostly.com/tutorials/using-custom-react-hooks-simplify-forms/

3rd Party Hook Demos

useInterval

https://github.com/streamich/react-use/blob/master/docs/useInterval.md

useImmer

https://github.com/immerjs/use-immer

useAxios

https://github.com/simoneb/axios-hooks

Custom Hook

Hooks are just function. Custom Hooks can call other hooks.

usage in component:

```
function App() {
  const [name, setName] = useLocalStorageState('name');
  const handleChange = event => setName(event.target.value);
 return <input value={name} onChange={handleChange} id="name" />
}
```

custom hook:

```
function useLocalStorageState(key, defaultValue = '') {
  const [state, setState] = React.useState(
    () => window.localStorage.getItem(key) || defaultValue
  );

  React.useEffect(() => {
    window.localStorage.setItem(key, state);
  }, [key, state]);

  return [state, setState];
}
```

Custom Hook

custom hook:

```
function useForm(initialValues, submitHandler){

  const [formValues, setFormValues] = useState(initialValues);

  function handleChange(event) {
    const target = event.target;
    const value = target.type === "checkbox" ? target.checked : target.value;
    const name = target.name;
    setFormValues({...formValues, [name]: value});
  }

  function handleSubmit(e) {
    e.preventDefault();
    submitHandler(formValues)
  }

  return {formValues, handleChange, handleSubmit};
}
```

usage in component:

```
function AppComponent() {
  const initialValues = {isGoing: true, numberOfGuests: 0};
  let {formValues, handleChange, handleSubmit} = useForm(initialValues, submitForm);
  ...
```

https://reactjs.org/docs/hooks-custom.html

Custom Hooks for Data Fetching

```
function useUsers(){

  const [data, setData] = useState(initialValue);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
      let cancelled = false;
      async function loadData(){
        try {
          const res = await fetch('https://jsonplaceholder.typicode.com/users');
          const resJson = await res.json();
          if (!cancelled) setData(resJson);
        } catch (err) {
          if (!cancelled) setError(err);
        } finally {
          if (!cancelled) setLoading(false);
        }
      }
      loadData();
      return () => { cancelled = true }
    }

  return {loading, data, error};
}
```

custom Hook

```
function AppComponent() {
  const {loading, data, error} = useUsers();

  if (loading) return <Spinner/>;
  if (error) return <>...</>;
  ...
  return <>...</>
}
```

Consider using `axios-hooks`, `use-http`, `react-fetch-hook` or *TanStack Query* or *SWR* instead of implementing the low-level fetch.

EXERCISE

Exercise: Custom Hooks

Component Composition

Container vs. Presentation Components

"Separtion of Concerns"

Application should be decomposed in container- and presentation components:

Container	**Presentation**
Little to no markup	Mostly markup
Pass data and actions down	Receive data & actions via props
typically stateful / manage state	mostly stateless
	better reusability

aka: Smart- vs. Dumb Components

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Separation of Concerns

Separation of concerns is not equal to separation of file types!

Keep things together that change together.

You can split a component into a controller and a view:

```
import {View} from './View';

export function Controller {
  ... // state & behavior
  return (
    <View data={...}
          onEvent={...} />
  );
}
```

Controller.js

```
export function View({data, onEvent}){
  return (
    <div>
      {data.message}
      <button onClick={()=>onEvent()}>
        Go!
      </button>
    </div>
  );
}
```

View.js

https://codesandbox.io/s/NxqMqyxlD

https://medium.com/styled-components/component-folder-pattern-ee42df37ec68

Controlling Component Lifecycle with a Key

React is preserving state in a component "instance".
By passing a **key** to a component, you can control the lifecycle of the "instance".

Example:

```
export default function ProfilePage({ userId }) {
  const [comment, setComment] = useState('');

  // 🔴 Avoid: Resetting state on prop change in an Effect
  useEffect(() => {
    setComment('');
  }, [userId]);
  // ...
}
```

```
export default function ProfilePage({ userId }) {
  return (
    <Profile
      userId={userId}
      key={userId}
    />
  );
}

function Profile({ userId }) {
  // ✅ This and any other state below will reset on key
change automatically
  const [comment, setComment] = useState('');
  // ...
}
```

https://react.dev/learn/you-might-not-need-an-effect#resetting-all-state-when-a-prop-changes

Composition with `props.children`

```
function WrapperComponent(props) {
    return (
      <div className="container">
        <hr/>
        {props.children}
        <hr/>
      </div>
    )
  }
}
```

```
function App() {
    return (
      <WrapperComponent>
        <ChildComponent>
      </WrapperComponent>
    );
}
```

TypeScript: There are several possibilities to type children. Popular choices are:

- React.ReactNode
- JSX.Element

https://www.totaltypescript.com/jsx-element-vs-react-reactnode

https://react.dev/learn/passing-props-to-a-component#passing-jsx-as-children

Components as props

```
function Layout({leftComponent, middleComponent, rightComponent}) {
  return (
    <div className="row">
      <div>{leftComponent}</div>
      <div>{middleComponent}</div>
      <div style={{display: 'flex', flexDirection: 'column'}}>
        {[1,2,3].map((i) => (
          <div key={i}>
            {React.createElement(rightComponent)}
          </div>)
        )}
      </div>
    </div>
  );
}
```

Passing component instance

passing component type

```
function App() {
  return (
    <Layout
      leftComponent={<Navigation/>}
      middleComponent={<MainContent/>}
      rightComponent={Advertisement}/>
  );
}
```

Higher Order Components

```
function withBackground(Component){
  return () => (
    <div className="container">
      <hr/>
        <Component/>
      <hr/>
    </div>
  )
}
```

Higher Order Component: takes a component as argument & returns a new component

```
function Content() {
    return (
      <div>Test!</div>
    )
}
const WrappedComponent = withBackground(Content);
```

applying the higher order component

`lazy()` and `memo()` are two examples of higher order components.

Note: HOCs were very popular before hooks. Hooks did replace many scenarios for HOCs!

Recompose: Higher-Order Component Library
https://github.com/acdlite/recompose (not maintained any more)

https://reactjs.org/docs/higher-order-components.html
https://www.robinwieruch.de/react-hooks-higher-order-components

Render Props

"render props" is a technique for sharing code beteween components.

```
function App() {
  return (
    <WrapperComponent render={(context) => {
      return <Content data={context.data}/>
    }}/>
  );
}
```

```
function WrapperComponent({render}) {
  const context = ...
  return (
    <div>
      <hr/>
      <div className="container">
        {render(context)}
      </div>
      <hr/>
    </div>
  )
}
```

Note: many use-cases for render props can be replaced with hooks in a more elegant way!

(but hooks can't render anything, cant' set values on a context and cant' implement error boundaries)

https://reactjs.org/docs/render-props.html
https://frontarm.com/james-k-nelson/hooks-vs-render-props/

Lazy Loading & Suspense

Lazy Loading of Components

React makes it very easy to load components on demand with
`React.lazy` & `<Suspense>`

https://react.dev/reference/react/lazy#suspense-for-code-splitting

```
import React, {lazy, Suspense} from 'react';
const OtherComponent = lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </Suspense>
  );
}
```

`<Suspense>` will render the `fallback` if a contained component is not yet loaded.

Note: `React.lazy()` is built on top of dynamic `import()` specified in ECMAScript 2020:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import

React.lazy()

`React.lazy()` is built on top of dynamic `import()`

Dynamic imports are part of ES2020:

```
import('./second.js')
    .then(m => m.sayHello());
```

function call

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#Dynamic_Imports
https://caniuse.com/#feat=es6-module-dynamic-import

Dynamic import is supported by modern bundlers (webpack, rollup, parcel): Based on `import` a seperate JavaScript bundle is created at build time, that can be lazyly-loaded by the browser when needed. The goal is a small initial bundle. This is also referred to as *code-splitting*, *lazy-loading* or *chunks*.

https://webpack.js.org/guides/code-splitting/
https://parceljs.org/code_splitting.html
https://rollupjs.org/guide/en#experimental-code-splitting

traditional

JavaScript Bundling

The complete source code is statically bundled at build-time.

Dynamic Module Loading

Dynamic imports are part of ES2020:

```
import('./second.js')
    .then(m => m.sayHello());
```

function call

Runtime: supported in all modern browsers.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#Dynamic_Imports
https://caniuse.com/#feat=es6-module-dynamic-import

Build-Time / Bundling:

Dynamic imports are supported by modern bundlers (webpack, rollup, parcel):
Based on **`import()`** a separate JavaScript bundle is created at build time, that can be asynchronously loaded by the browser when needed.
This is typically used to keep the initial bundle small. This is also called: *code-splitting or lazy-loading of chunks*.

https://webpack.js.org/guides/code-splitting/
https://parceljs.org/code_splitting.html
https://rollupjs.org/guide/en#experimental-code-splitting

Lazy Loading

JavaScript dynamically loads a chunk at *runtime*. The code is (partially) generated by the bundler.
Frameworks have additional abstractions (Angular Router, React.lazy ...)

All the application parts (source, npm-packages ...) must be available at build time.

One build creates a *single* deployment artifact consisting of several chunks.

Route Splitting

In most cases it makes sense to lazy-load top-level route components, just because it is so simple.

```
const ToDoScreen = lazy(() => import('./components/ToDoScreen'));
const DoneScreen = lazy(() => import('./components/DoneScreen'));
```

```
createBrowserRouter([{
   ...
      { path: "/", element: <ToDoScreen /> },
      { path: "/done", element: <DoneScreen /> },
   ...
]});
```

Note: React Router v7, will probably provide "automatic" route splitting for top level routes

Note: If a central state-management is used (i.e. Redux) you should also think about "chunking" it by route ...

Suspense

https://react.dev/reference/react/Suspense

```
function App() {
    return (
        <>
            <Suspense fallback={<div>Loading...</div>}>
                <Content/>
            </Suspense>
        </>
    )
}
```

```
let loaded = false;
const promise = new Promise((resolve) => {
    setTimeout(() => {
        loaded = true;
        resolve(loaded);
    }, 1000);
});


function Content() {
    if (!loaded) {
        throw promise;
    }
    return <h1>Content</h1>;
}
```

Suspense

https://github.com/pmndrs/suspend-react

```
function App() {
    return (
        <>
            <Suspense fallback={<div>Loading...</div>}>
                <Content/>
            </Suspense>
        </>
    )
}
```

```
async function loadData() {
    const promise: Promise<string> = new Promise((resolve) => {
        setTimeout(() => {
            resolve("Test");
        }, 1000);
    });
    return promise;
}

function Content() {
    const data = suspend(loadData);
    return <h1>{data}</h1>;
}
```

Error Boundaries

Error Boundary

An Error Boundary is a component that can catch errors that happen during render or lifecycle methods in the components below them in the tree.

```
class ErrorBoundary extends React.Component {

  static getDerivedStateFromError(error) {
     // return a state update object with error data
     // next render can then use that error data
  }

  componentDidCatch(error, info) {
     // perform an action based on the error
  }

  render() {
   //
  }
}
```

"pure" function, no side-effects, just updating the state

side-effects like logging are executed here

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

Errors in render or during the component lifecycle, can't be handled with try/catch since React is calling these functions, based on declarative JSX.

https://react.dev/reference/react/Component#catching-rendering-errors-with-an-error-boundary

react-error-boundary

Small wrapper library providing a JSX element for an error boundary.
(so you don't need to write a class component)

```
npm install react-error-boundary
```

```
<ErrorBoundary
    onError={myErrorHandler}
    fallback={<CustomErrorComponent/>}>

  <ComponentThatMayError />

</ErrorBoundary>
```

https://github.com/bvaughn/react-error-boundary

React Error Handling

imperative vs. declarative code paths

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

  function increaseCount() {
    try {
     const val = maybeThrowError();
     return () => setCount((count) => count + val);
    } catch(e) { console.log('Error!') }
  }

  const val = maybeThrowError();

  return (
    <div>
      <div>Display of Counter: {val}</div>
      <button onClick={increaseCount()}>count is {count}</button>
    </div>
  );
}
```

event handling is imperative
=> handling with try-catch

rendering is declarative
=> handling with error-boundary

```
function App() {
  return (
    <div className="App">
        <ErrorBoundary fallback={<h1>Error ...</h1>}>
          <Counter />
        </ErrorBoundary>
    </div>
  );
}
```

React Error Handling

bridging from imperative to declarative error handling

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

  function increaseCount() {
    try {
     const val = maybeThrowError();
     return () => setCount((count) => count + val);
    } catch(e) { e => setError(e); }
  }

  if (error) throw error;

  return (
    <div>
      <div>Display of Counter: {val}</div>
      <button onClick={increaseCount()}>count is {count}</button>
    </div>
  );
}
```

EXERCISE

Exercise: Component Composition

Performance

React Performance

How fast is a "render-cycle"?
How many render cycles do we have?

Mechanisms from React:

- `memo`: higher order component ro prevent re-renders by memoizing a component
  https://reactjs.org/docs/react-api.html#reactmemo
- `useMemo`: hook to memoize a value to prevent expensive calculations on every render
  https://reactjs.org/docs/hooks-reference.html#usememo
- `useCallback`: hook to memoize a function to make callbacks
  https://reactjs.org/docs/hooks-reference.html#usecallback

https://kentcdodds.com/blog/fix-the-slow-render-before-you-fix-the-re-render
https://reactjs.org/docs/profiler.html

React Compiler

"under construction"

In the future the "React Compiler" will hopefully make manual optimization with memo, useMemo and useCallback obsolete ...

https://react.dev/learn/react-compiler

EXERCISE

Exercise: Performance

Data Fetching / Remote State

Proper Data Fetching with `useEffect`

https://react.dev/learn/synchronizing-with-effects#fetching-data

```
useEffect(() => {
  let ignore = false;

  async function startFetching() {
     try {
        const json = await fetchTodos(userId);
        if (!ignore) {
           setTodos(json);
        }
     } catch (e: any) {
        setError('An error occured: ' + e.getMessage());
     }
  }

  startFetching();

  return () => {
    ignore = true;
  };
}, [userId]);
```

effect function can't be async!

avoiding race conditions by ignoring "outdated" fetches

error handling

cleanup function:
- ignoere the result
- alternative: abort the fetch

potential for further improvement:
- caching / dedupiung
- loading state

Proper data fetching verbose and complicated:

- consider using a library
- encapsulate the logic into custom hooks

https://react.dev/learn/synchronizing-with-effects#what-are-good-alternatives-to-data-fetching-in-effects

Custom Hooks for Data Fetching

```
function useUsers(){

  const [data, setData] = useState(initialValue);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
      let cancelled = false;
     (async () => {
       try {
         const res = await fetch('https://jsonplaceholder.typicode.com/users');
         const resJson = await res.json();
         if (!cancelled) setData(resJson);
       } catch (err) {
         if (!cancelled) setError(err);
       } finally {
         if (!cancelled) setLoading(false);
       }
     })();
      return () => { cancelled = true }
    }

  return {loading, data, error};
}
```

custom Hook

```
function AppComponent() {
  const {loading, data, error} = useUsers();

  if (loading) return <Spinner/>;
  if (error) return <>...</>;
  ...
  return <>...</>
}
```

Consider using **`axios-hooks`**, **`use-http`** or *TanStack Query* or *SWR* instead of implementing the low-level fetch.

https://github.com/simoneb/axios-hooks
https://github.com/ava/use-http
https://swr.vercel.app/
https://tanstack.com/query

Data Fetching Libraries

If you are using "plain" axios or fetch, then you have to care about cancellation, loading & error state, caching, deduplication, refetching ...

Consider a data-fetching library:

- TanStack Query https://tanstack.com/query/
- SWR https://github.com/vercel/swr
- RTK Query (only if already using Redux): https://redux-toolkit.js.org/rtk-query/overview

Consider data loading and mutation via the router:

- ReactRouter: https://reactrouter.com/en/main/start/overview#data-loading
- TanStack Router https://tanstack.com/router/latest/docs/framework/react/guide/data-loading

... or a more-lightweigth hook

- use-http: https://use-http.com
- axios-hooks: https://github.com/simoneb/axios-hooks

Libraries for Managing remote State

The patterns:

- get rid of "self managed" state and lifecycle
- "stale-while-revalidate" pattern
- client state is just a "cached snapshot" of server state

Demo:

- getting rid of state with react router
- getting rid of state with tanstack-query

EXERCISE

Exercise: Data Fetching

Type-Safe Backend Access

https://twitter.com/donavon/status/1430219301500358658

```
interface ToDoGetResponse {
  data: ToDoData[];
}
```

```
export interface ToDoData {
  id: string;
  title: string;
  completed: boolean;
}
```

using Angular HttpClient

```
getTodos(completed?: boolean): Observable<ToDo[]> {
  return this.http.get<ToDoGetResponse>(BACKEND_URL);
}
```

using axios with TypeScript

```
export async function loadToDos(
  const serverResponse = await axios.get<ToDosGetResponse>(BACKEND_URL);
  ...
```

The type information is only for the TypeScript compiler at build time. At runtime there is no guarantee or check that the network call really returns objects of the specified type.

You should only use `interface` or `type` as *type arguments* for backend access.

TypeScript Generators

Open API Generator: https://openapi-generator.tech/

https://github.com/OpenAPITools/openapi-generator

```
npx @openapitools/openapi-generator-cli \
    generate -i https://petstore.swagger.io/v2/swagger.json -g typescript-angular -o .
```

Generated code is very different for: `typescript-angular`, `typescript-fetch`, `typescript-axios` ...

Alternative: https://swagger.io/tools/swagger-codegen/ resp. https://github.com/swagger-api/swagger-codegen

Nx Plugin for Open API Generator: https://github.com/trumbitta/nx-trumbitta/tree/main/packages/nx-plugin-openapi

Orval - Restful Client Generator https://orval.dev/overview

Alternative simple generator for Java: https://github.com/vojtechhabarta/typescript-generator

```
public class Person {
    public String name;
    public int age;
    public boolean hasChildren;
    public List<String> tags;
    public Map<String, String> emails;
}
```

```
interface Person {
    name: string;
    age: number;
    hasChildren: boolean;
    tags: string[];
    emails: { [index: string]: string };
}
```

Very easy to configure in a Maven or Gradle build.
Can detect DTOs automatically for JAX-RS or DTOs can be specified via pattern.
Somehow configurable (mapping, naming ...)
Limitation: One single generated file / module containing all the types.

Graphql:
- https://graphql-code-generator.com/
- https://github.com/apollographql/apollo-tooling

.NET:
- NSwag: https://github.com/RicoSuter/NSwag, Swashbuckle: https://github.com/domaindrivendev/Swashbuckle.AspNetCore
- TypeWriter Visual Studio Extension: https://frhagn.github.io/Typewriter/
- "Handmade": https://github.com/lmcarreiro/cs2ts-example

"Full Stack" TypeScript:
- https://nestjs.com/
- https://remix.run/
- https://nextjs.org/
- https://blitzjs.com/

TypeScript Generators

Open API Generator: https://openapi-generator.tech/

https://github.com/OpenAPITools/openapi-generator

```
npx @openapitools/openapi-generator-cli \
    generate -i https://petstore.swagger.io/v2/swagger.json -g typescript-angular -o .
```

Generated code is very different for: `typescript-angular`, `typescript-fetch`, `typescript-axios` ...

Alternative: https://swagger.io/tools/swagger-codegen/ resp. https://github.com/swagger-api/swagger-codegen

Alternative simple generator for Java: https://github.com/vojtechhabarta/typescript-generator

```
public class Person {
    public String name;
    public int age;
    public boolean hasChildren;
    public List<String> tags;
    public Map<String, String> emails;
}
```

```
interface Person {
    name: string;
    age: number;
    hasChildren: boolean;
    tags: string[];
    emails: { [index: string]: string };
}
```

Very easy to configure in a Maven or Gradle build.
Can detect DTOs automatically for JAX-RS or DTOs can be specified via pattern.
Somehow configurable (mapping, naming ...)
Limitation: One single generated file / module containing all the types.

Graphql:

- https://graphql-code-generator.com/
- https://github.com/apollographql/apollo-tooling

"Full Stack" TypeScript:

- https://nestjs.com/
- https://remix.run/
- https://nextjs.org/
- https://blitzjs.com/

.NET:

- NSwag: https://github.com/RicoSuter/NSwag, Swashbuckle: https://github.com/domaindrivendev/Swashbuckle.AspNetCore
- TypeWriter Visual Studio Extension: https://frhagn.github.io/Typewriter/
- "Handmade": https://github.com/lmcarreiro/cs2ts-example

Runtime Type-Checking

Runtime type-checks can only happen via JavaScript.

Integration with TypeScript can happen by code-generation or by deriving TypeScript types from schema definitions at compile time.

Zod: describe types (schemas) in TS code: https://zod.dev/
→ infer TS types from schemas

Valibot: describe types (schemas) in TS code: https://github.com/fabian-hiller/valibot
→ infer TS types from schemas / a smaller and modular alternative to Zod

AJV: JSON Schema Validator https://ajv.js.org/
→ deriving TS types from AJV definitions: https://ajv.js.org/guide/typescript.html
→ generating json schema from TS types: https://github.com/YousefED/typescript-json-schema

joi: Data Validator for JavaScript https://joi.dev/
→ generating TS types from joi schemas: https://github.com/mrjono1/joi-to-typescript

io-ts: describe types in TS code: https://github.com/gcanti/io-ts
→ derive TS types: https://github.com/gcanti/io-ts/blob/master/index.md#typescript-integration

TypeScript-JSON: https://github.com/samchon/typescript-json
→ generate runtime type checks from TS types via TypeScript custom transformer

Zod Demo

"runtim type-checking"

fetching data supposedly typesafe, but the type definitions are not correct:

```
npm install ky
```

ky is a lightweight alternative to axios

```
interface ToDo { name: string, completed: boolean }
```

```
const todos = ky.get('http://localhost:3456/todos').json<ToDo[]>();
console.log(todos[0].name); // <= undefined
console.log(todos[0].name.toUpperCase()); // kaboom!!!
```

runtime type-check with Zod, compile-time types are derived ...

```
npm install zod
```

```
const ToDoSchema = z.object({
  id: z.number(),
  name: z.string(),
  completed: z.boolean(),
});

const ToDoResponseSchema = z.object({
  result: z.array(ToDoSchema),
});

type ToDoResponse = z.infer<typeof ToDoResponseSchema>;
```

```
const todos = ky.get('http://localhost:3456/todos').json<ToDo[]>();

// ToDoResponseSchema.safe(response); // <= throws errors
const validationResult = ToDoResponseSchema.safeParse(response);

if (!validationResult.success) {
  console.error(validationResult.error.issues);
  return;
} else {
  const todos = response.result;
  console.log("TODOS", todos);
  const firstTodo = todos[0];
  console.log("First todo", firstTodo.title.toUpperCase());
}
```

State

"Out-of-the-Box" State Management

Hooks: useState, useRef, useReducer & Context

"Minimal State Mindset":
Anything that can be derived from the application state, should be derived. Automatically.
- MobX Introduction

Derived state is calculated during rendering.
Optimization with Memoization (useMemo).

Libraries offer their own concepts: Redux with Reselect, MobX, Recoil …)

Basic Patterns for State in a Component Tree

Lifting State Up

(properties down / events up)

https://react.dev/learn/sharing-state-between-components

Container Components / Presentation Components

aka Smart- vs Dumb Components

https://www.patterns.dev/posts/presentational-container-pattern/

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

https://toddmotto.com/stateful-stateless-components

Component Composition

https://react.dev/learn/passing-data-deeply-with-context#before-you-use-context

https://react.dev/learn/passing-props-to-a-component#passing-jsx-as-children

Thinking in React: Lifting State Up

State should be modeled as a single source of truth.
Duplicated state should be avoided! Sometimes duplication can be avoided by *deriving* dependent state from existing state.

https://react.dev/learn/sharing-state-between-components
https://react.dev/learn/thinking-in-react

Lifting State up

https://react.dev/learn/sharing-state-between-components

Unidirectional Data-Flow

State should be explicitly owned by one component.
State should *never be duplicated* in multiple components.
(sometimes it is tricky to detect the *minimal state* from which other state properties can be *derived*)

- A parent component passes state to children as properties. Children are re-rendered when these properties change.
- Children should not edit state of their parent
- Children "notify" parents (events, callbacks …)

React formalises **unidirectional data-flow** via `props`:
passing data and callbacks to child components

Container vs. Presentation Components

"Separtion of Concerns"

Application should be decomposed in container- and presentation components:

Container	**Presentation**
Little to no markup	Mostly markup
Pass data and actions down	Receive data & actions via props
typically stateful / manage state	mostly stateless
	better reusability

aka: Smart- vs. Dumb Components

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Separation of Concerns

Separation of concerns is not equal to separation of file types!

Keep things together that change together.

You can split a component into a controller and a view:

```
import {View} from './View';

export function Controller {
  ... // state & behavior
  return (
    <View data={...}
          onEvent={...} />
  );
}
```

Controller.js

```
export function View({data, onEvent}){
  return (
    <div>
      {data.message}
      <button onClick={()=>onEvent()}>
        Go!
      </button>
    </div>
  );
}
```

View.js

https://codesandbox.io/s/NxqMqyxlD

https://medium.com/styled-components/component-folder-pattern-ee42df37ec68

Data Flow & Component Architecture

React Data Flow Critique

In React the data flow is heavily coupled to the component structure and the re-rendering of the component tree.

The idea behind "lifiting" state up is, that the component tree matches the data-flow tree. This is a abstraction that might become a problem for more complicated applications.
At a certain point though, "lifting state up" will hurt the "Composability and Reusablity" principles

"Fine-grained reactivity" is not possible in React. Changes only propagate if a component-tree is re-rendered.
Some 3rd party solutions propose to "lift state out" instead of "lifting state up".

Vue, Solid.js, Qwik, Svelte and soon also Angular 8swith signals) offer fine grained reactivity.

https://podrocket.logrocket.com/usewat-david-khourshid
https://medium.com/@nimelrian/thinking-in-react-a-paradox-statement-33c19e2eb9e2
https://react.dev/reference/react/useSyncExternalStore

Prop Drilling

From the component architecture follows the pattern of "*lifting state up*": if several components need to reflect the same changing data, then the shared state should be liftet up to their closest common ancestor.

https://reactjs.org/docs/lifting-state-up.html

"Prop Drilling" is the process you have to go through to pass data and events through the component tree.

Prop drilling can be a *good thing*: it makes the data-flow very explicit!
Prop drilling can be a *bad thing*: passing data from its holder to a consumer via serveral intermediates is tedious and makes changing the component tree more difficult.

https://blog.kentcdodds.com/prop-drilling-bb62e02cb691

Basic State Options

simple state

```
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
```

structured state with objects

```
const [state, setState] = useState({
  firstName: '',
  lastName: ''
});
```

```
setState({...state, firstName: 'updated'})
```

useReducer to change structured state

```
const [state, dispatch] = useReducer(reducerFn, {
  firstName: '',
  lastName: ''
});
```

```
dispath({
  type: 'udpateFirstName', payload:'updated'
})
```

derived state:

```
const fullName = `${firstName} ${lastName}`;
```

```
const fullName = useMemo(() => `${firstName} ${lastName}`, [firstName, lastName]);
```

structured state with immer

```
const [state, updateState] = useImmer({
  firstName: '',
  lastName: ''
});
```

```
updateState(draft => {
  draft.firstName = 'updated'
})
```

Immutability

- the author of immer.js

https://twitter.com/mweststrate/status/1310575600990642177

User interfaces are the quintessential place you find mutable state in any application—literally, the user is mutating the state of the app by clicking and typing things. So any good UI framework will not insist on immutability but rather, make it easy to work with state at whatever level is appropriate for your situation.

https://blog.plan99.net/reacts-tictactoe-tutorial-in-kotlin-javafx-715c75a947d2

The bottom line is that its harder to write immutable code and easier to break it, plus there is little point having an immutable front-end if you don't have a back-end to support it. Fashion aside, I'd rather see immutability for what it is, ie a tool to plug a gap when your choice of framework does not handle state intuitively.

Immutability in JavaScript: A Contrarian View: http://desalasworks.com/article/immutability-in-javascript-a-contrarian-view/

Tania Rascia
@taniarascia

How do you deal with updating nested objects? I'm tired of writing this all the time:

```
newState = {
  ...oldState,
  prop: {
    ...oldState.prop,
    nestedProp: newValue
  }
}
```

5:26 PM · Jan 8, 2020 · Twitter Web App

https://twitter.com/taniarascia/status/1214946400582041600

Immer.js

Immer is a tiny package that allows you to work with immutable state in a more convenient way. It is based on the copy-on-write mechanism.

```
import produce from "immer"

const baseState = [
    { todo: "Learn typescript", done: true },
    { todo: "Try immer", done: false}
]

const nextState = produce(baseState, draftState => {
    draftState.push({todo: "Tweet about it"})
    draftState[1].done = true
})
```

Integration with React Hooks: https://github.com/immerjs/use-immer
Immer is included in Redux Toolkit.

https://immerjs.github.io/immer/

A single component

Managing state in a component is simple:

Application with a State Container

"Lifting state out"

A state container extracts the shared state out of the components, and manages it in a global singleton.

The component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

Managing State with a State Container

State can be managed outside the components.
Components can be bound to state.

State Management: State Container

Redux: https://redux.js.org/
MobX: https://mobx.js.org/
https://recoiljs.org/
https://jotai.org/
https://github.com/pmndrs/zustand
https://github.com/pmndrs/valtio
https://atlassian.github.io/react-sweet-state/#/
https://xstate.js.org/docs/

Demo: Global State with Jotai

```
npm i jotai
```

outside of components

```
import { atom } from 'jotai'

const countAtom = atom(0);
const doubleCountAtom = atom((get) => get(countAtom) * 2);
```

inside components

```
const [count, setCount] = useAtom(countAtom);
const increment = () => setCount((count) => count + 1);
```

```
const count = useAtomValue(doubleCountAtom);
```

Integration and ecosystem: https://jotai.org/docs/introduction#extensions

The "Wild West" of React State Management

The React State Museum: https://github.com/GantMan/ReactStateMuseum

Redux: https://redux.js.org/
Redux Toolkit: https://redux-toolkit.js.org/
MobX: https://mobx.js.org/
MobX State Tree: https://github.com/mobxjs/mobx-state-tree
XState: https://xstate.js.org/docs/
Satchel: https://github.com/Microsoft/satcheljs
RxJS & Recompose: https://github.com/acdlite/recompose
Rematch: https://github.com/rematch/rematch
react-stateful: https://github.com/didierfranc/react-waterfall (based on Context API)
Unstated: https://github.com/jamiebuilds/unstated
Unstated Next: https://github.com/jamiebuilds/unstated-next
Cerebral: https://github.com/cerebral/cerebral
Recoil: https://recoiljs.org/
Zustand: https://github.com/pmndrs/zustand
Jotai: https://github.com/pmndrs/jotai
Valtio: https://github.com/pmndrs/valtio
React Tracked: https://react-tracked.js.org/
React Sweet State: https://atlassian.github.io/react-sweet-state/#/
...

The Next Generation ...

Cory House
@housecor

My take on global state in React:
1. Use react-query for remote state (HTTP calls).
2. Use context for low frequency updates.

If I have other global state that changes often, I consider:
3. Jotai (Recoil simplified)
4. Zustand (Redux simplified)
5. Valtio (Mobx simplified)

2:36 PM · Aug 29, 2022 · Twitter Web App

https://twitter.com/housecor/status/1564230512486629381

Evaluating State Management Libraries

Important Characteristics:

- Immutable vs Mutable State
- Implementing Derived State
- Rerendering Performance and Optimization Techniques

URL State

URL State

The URL represents a persistent state of your application that is controlled by the user.

Keep the browser-refresh in mind:
What state does the user expect to "survive" the refresh?

URL > client state

Do not sync URL with client state ...
... but use the URL as primary source.

Why you should lift component state up to the URL:
https://www.youtube.com/watch?v=sFTGEs2WXQ4

nuqs

https://nuqs.47ng.com/

```
npm install nuqs
```

```
<NuqsAdapter> ... </NuqsAdapter>
```

```
import { parseAsInteger, useQueryState } from "nuqs";
export function Demo() {
  const [hello, setHello] = useQueryState("hello", { defaultValue: "" });
  const [count, setCount] = useQueryState(
    "count",
    parseAsInteger.withDefault(0),
  );
  return (
    <>
      <button onClick={() => setCount((c) => c + 1)}>Count: {count}</
button>
      <input
        value={hello}
        placeholder="Enter your name"
        onChange={(e) => setHello(e.target.value || null)}
      />
      <p>Hello, {hello || "anonymous visitor"}!</p>
    </>
  );
}
```

Router as State Manager?

TanStack Router BETA

SEARCH PARAM APIS TO MAKE YOUR STATE-MANAGER JEALOUS

Instead of throwing you to the wolves, TanStack Router outfits you with state-manager-grade search param APIs. With **schemas, validation, full type-safety and pre/post manipulation** you'll wonder why you're not storing everything in the URL.

https://tanstack.com/router

Delayed Startup

React: Delay initial render

```
async function startApp() {

    const config = await initStuff();

    ReactDOM.render(
        <React.StrictMode>
            <App config={config}/>
        </React.StrictMode>,
        document.getElementById('root')
    );
}
startApp().catch(console.error);
```

Styling React

Styling React Components

- Traditional CSS

  Optional with a CSS preprocessor: SASS, Less, Stylus

- Inline Styles

```
<h1 style={{color: 'red'}}> Test </h1>
```

  Are Inline Styles Faster than CSS? https://danielnagy.me/posts/Post_tsr8q6sx37pl

- CSS Modules

  CSS classes are scoped to components

- CSS-in-JS Library

  Generate styles with JavaScript

  - Emotion: https://github.com/emotion-js/emotion
  - Styled Components: https://github.com/styled-components/styled-components
  - Stitches: https://stitches.dev/
  - Vanilla Extract: https://vanilla-extract.style/
  - TSS React: https://www.tss-react.dev/

- Tailwind: https://tailwindcss.com/

https://nextjs.org/docs/app/building-your-application/styling/css-in-js

```
npm i node-sass @emotion/core @styled-components
```

```
/** @jsx jsx */
import React from 'react';
import {css, jsx} from '@emotion/core'
import styled from 'styled-components'
import styles from './Greeter.module.scss';

const Title = styled.h1`
  color: brown;
`;

export default function Greeter() {
  return (
    <div>
      <h1 className={styles.title}>Styled with CSS module</h1>
      <h1 css={css`
        color: pink;
      `}>Styled with Emotion</h1>
      <Title>Styled with Styled components</Title>
    </div>
  )
}
```

Note: typically it does not make sense to use different styling libraries!

TailwindCSS

utility-first CSS framework

```
<button class="px-4 py-1 text-sm text-purple-600 font-semibold rounded-full border
                  border-purple-200 hover:text-white hover:bg-purple-600
                  hover:border-transparent focus:outline-none focus:ring-2
                  focus:ring-purple-600 focus:ring-offset-2">
     Message
</button>
```

https://tailwindcss.com/docs/utility-first#why-not-just-use-inline-styles

Installation for any framework:
https://tailwindcss.com/docs/installation/framework-guides

Tailwind is traditionally strong for styling raw html elements.
Typically it can't be used to style a traditional component library.
But the rise of headless component libraries open a new usage-scenario for Tailwind.

https://tailwindcss.com/

Tailwind is very controversial

Tailwind CSS is the worst:
https://www.youtube.com/watch?v=lHZwlzOUOZ4

The Tailwind CSS Drama Your Users Don't Care About
https://www.builder.io/blog/the-tailwind-css-drama-your-users-don't-care-about

Why I don't like Tailwind:
https://www.aleksandrhovhannisyan.com/blog/why-i-dont-like-tailwind-css/

Component Libraries

Component Libraries

Material UI

https://mui.com/

- Chackra UI:
  https://chakra-ui.com/
- Ant Design of React
  https://ant.design/docs/react/introduce
- Semantic UI
  https://react.semantic-ui.com/

And more:
Rainbow UI, Cloudscape Design System, react-bootstrap, reactstrap ...

- KendoReact
  https://www.telerik.com/kendo-react-ui/
- PrimeReact
  https://www.primefaces.org/primereact/
- Infragistics / Ignite UI:
  https://www.infragistics.com/products/ignite-ui-react
- DevExtreme
  https://js.devexpress.com/
- Syncfusion:
  https://www.syncfusion.com/react-ui-components
- jQWidgets:
  https://www.jqwidgets.com/react/react-js-components.htm
- agGrid
  https://www.ag-grid.com/

https://github.com/brillout/awesome-react-components

Headless Components

Headless Components

Components with minimal or no UI.

Radix https://www.radix-ui.com/

React Aria https://react-spectrum.adobe.com/react-aria/index.html

headless UI https://headlessui.com/

Often combined with Tailwind: https://tailwindcss.com/

shadcn/ui https://ui.shadcn.com/

TanStack Table: https://tanstack.com/table
ReactRanger: https://github.com/tannerlinsley/react-ranger

TanStack Form: https://tanstack.com/form
HouseForm: https://houseform.dev/

```
npm install @mui/material @emotion/react @emotion/styled
```

```
import Button from "@mui/material/Button";

<Button variant="contained" onClick={increment}>Click Me!</Button>

<Button sx={{ color: "red" }} onClick={increment}>Click Me!</Button>
```

Radix

first install tailwind: https://tailwindcss.com/docs/guides/vite

```
npm install @radix-ui/react-switch
```

unstyled:

```
<div className=" items-center">
  <div>Switch</div>
  <div>
    <Switch.Root>
      <Switch.Thumb />
    </Switch.Root>
  </div>
</div>
```

styled:

```
<Switch.Root className="w-[42px] h-[25px] bg-blackA6 rounded-full relative shadow-[0_2px_10px]
                shadow-blackA4 focus:shadow-[0_0_0_2px] focus:shadow-black data-[state=checked]:bg-
                black outline-none cursor-default">
  <Switch.Thumb className="block w-[21px] h-[21px] bg-white rounded-full shadow-[0_2px_2px]
            shadow-blackA4 transition-transform duration-100 translate-x-0.5 will-change-transform
            data-[state=checked]:translate-x-[19px]" />
</Switch.Root>
```

Have Fun with React!

JavaScript / Angular / React / Vue / Vaadin

Schulung / Beratung / Coaching / Reviews

jonas.bandi@ivorycode.com