REACTJS

ADVANCED PROGRAMMING

```
class ReactAdvanced extends React.Component {
  constructor(props) {
    super();
  }
  render() {
    let name = "Obi-Wan Kenobi";
    return <h1>Programming like {name}</h1>;
  }
}
```

DOCDOKU
digital enterprise applications

PRESENTATION

- Trainer
- **DocDoku**

ABOUT YOU

- Experience with React
- What you expect from the training
- Your upcoming projects using React

TERMS

- Schedules : 9h -> 17h30
- Breaks : 15 minutes morning and afternoon
- Lunch : 1h30

EDUCATIONAL OBJECTIVES

- Understand advanced ReactJS **concepts**
- Optimize **performance** of ReactJS applications
- Improve product code **quality**
- Integrate the various essential external **libraries**
- Improve user experience with advanced **Redux** features

AGENDA

- [1] Introduction
- [2] Module Federation with React & Next.js
- [3] Advanced techniques and design patterns
- [4] Advanced Redux features
- [5] Performance Optimization
- [6] Animations/transitions

[1] INTRODUCTION

- Reminders about ES6+ and modules.
- The key principles of React: VirtualDOM, JSX, One-way Data Flow.
- Discover the ecosystem of ReactJS tools.

ES6 KEY FEATURES

- Modules
- Destructuring
- Spread Operator
- Arrow functions
- Template Literals
- Generator functions

ES6 MODULES

Modules are a way to **structure** and **organize** code by breaking it into different **independent** files.

```
// main.js
import MyClass from "my-class";
MyClass.hello();
```

```
// my-class.js
export default class MyClass {
  static hello() {
    console.log("Hello world");
  }
}
```

DESTRUCTURING

Shortcut syntax that allows to **unpack** data by assigning it to distinct **variables**

```
let jedis = [
  { name: "Obi-Wan Kenobi", cristal_color: "Blue" },
  { name: "Luke Skywalker", cristal_color: "Green" },
];

/// [{object 1},{object 2}]
let [{ name }, { cristal_color }] = jedis;
console.log(name, cristal_color); // Obi-Wan Kenobi, Green
```

This is equivalent to :

```
let name = jedis[0].name;
let cristal_color = jedis[1].cristal_color;
```

SPREAD OPERATOR

Handy operator that allows to **transform** an **array** into an **argument list**

```
function add(a, b) {
  return a + b;
}

let numbers = [1, 2];
add(...numbers);
```

Copying an array:

```
let numbers_copy = [...numbers];
```

ARROW FUNCTIONS

Shortcut syntax for a traditional **function**

```
// Simplification steps of a square function:
function square(x) {
  return x * x;
}
const square = function (x) {
  return x * x;
};
const square = (x) => {
  return x * x;
};
const square = (x) => x * x;
```

Limitations: **super**, **this** and **arguments** keywords are not bound to an arrow function.

TEMPLATE LITERALS

Also known as **template strings**, commonly used to **interpolate** expressions.

```
let some_string = `this is a simple string`;
let some_multiline_string = `this is first line
this is second line`;

// interpolation
let name = "Joe";
let string = `Hello my name is ${name}`;
```

TEMPLATE LITERALS AND TAG TEMPLATES

Advanced function that can **parse** the template literal and **transform** it.

```
function myTag(strings, arg1) {
  return strings[0].replace("Bonjour", "Hello") + arg1;
}

let data = "world";
let string = myTag`Bonjour ${data}`; // Hello world
```

FUNCTION*

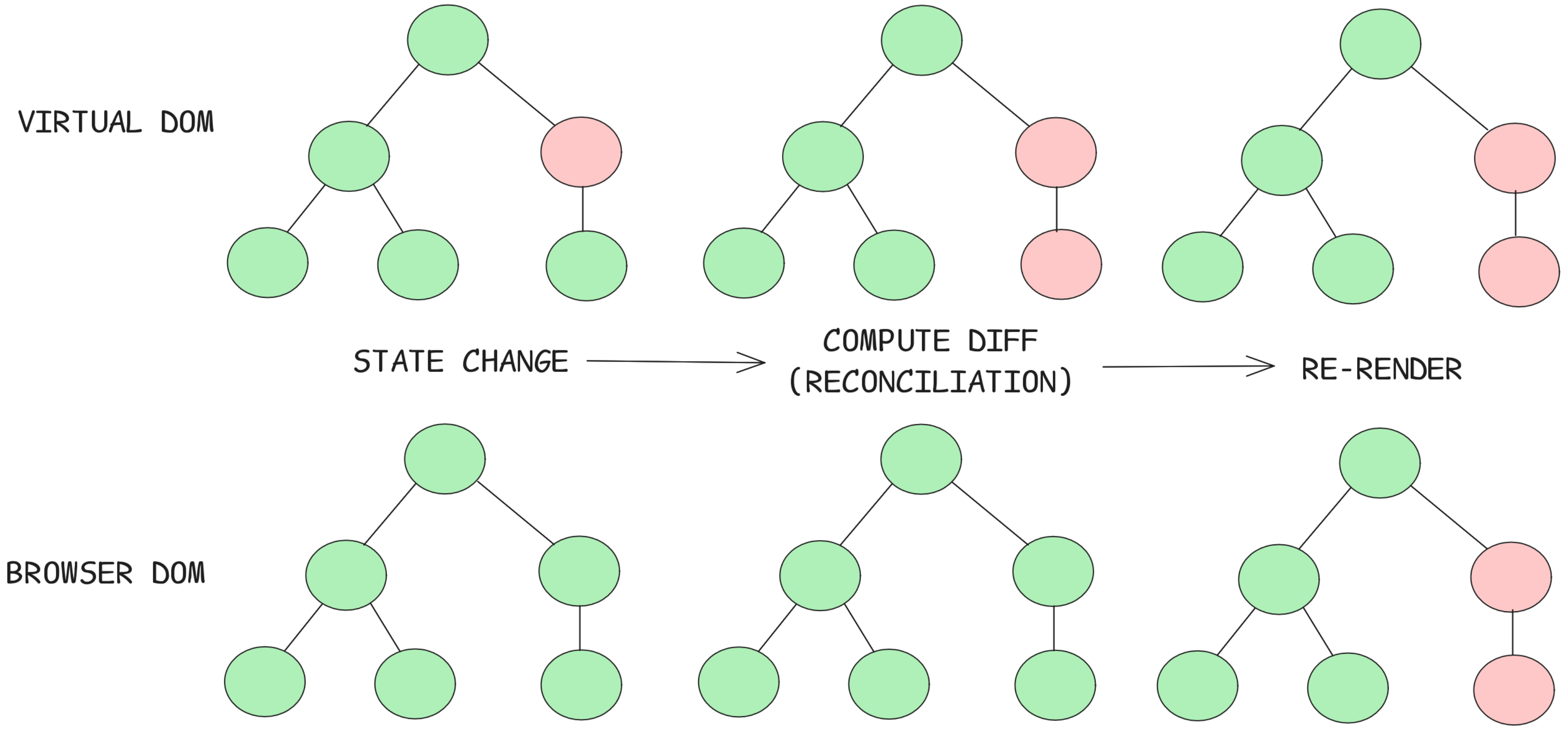
The `function*` declaration creates a binding of a new generator function.

The **yield** operator is used to pause and resume a generator function.

```
function* generator(i) {
  yield i;
  yield i + 1;
}

const gen = generator(0);
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // undefined
console.log(gen.next()); // Object { value: undefined, done: true }
```

REACT VIRTUAL DOM

Programming concept that is the **representation** of the UI in memory, and **synced** with the "real" DOM



JSX

Syntax **extension** for JavaScript

Allows to write HTML inside the JS code.

Basic example:

```
let my_string = (
  <div>
    <h1>Hello world</h1>
  </div>
);
```

JSX: SINGLE ROOT ELEMENT

JSX does not allow to have more than one root element

```
// this won't compile
let data = (
    <p>Hello</p>
    <p>World</p>
)
```

Either wrap it inside a `<div>` element or an empty tag `<></>`

```
let data = (
  <>
    <p>Hello</p>
    <p>World</p>
  </>
);
```

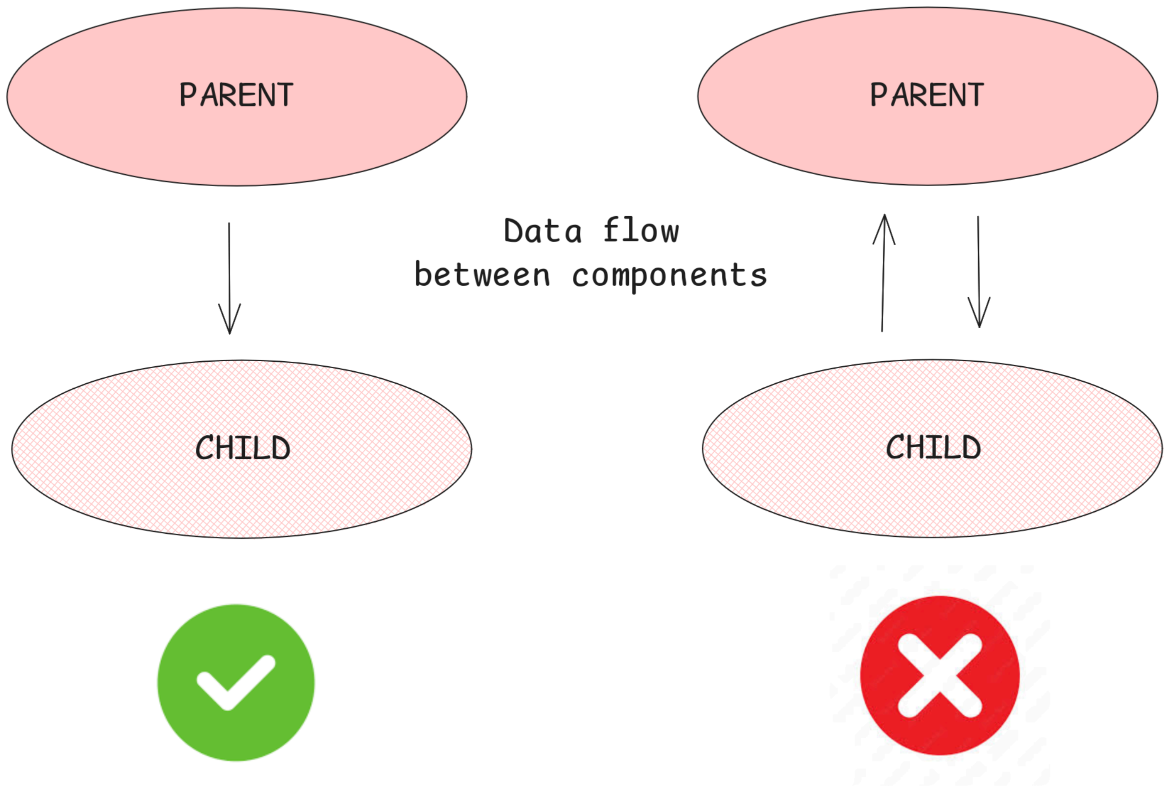
JSX: INTERPOLATION

Usage of curly braces `{some_variable}`

```
export default function MyFunction() {
  const name = "Obi-Wan Kenobi";
  const color = "blue";
  return (
    <>
      <h1>{name} is a jedi.</h1>
      <h2>His color is {color}</h2>
    </>
  );
}
```

ONE WAY DATA FLOW

The data flow in React is **unidirectional**. From the parent component to the child component.



READ ONLY PROPS

The child component can **access** the data passed by the parent but cannot modify it.

Here, the parent passes a jedi name as a prop (name) to the child. The child can **render** the data but cannot change it.

```
function Parent() {
  const jedi_name = "Obi-Wan Kenobi";
  return <Jedi name={jedi_name} />;
}

function Jedi({ name }) {
  return <p>{name}</p>;
}
```

DEVELOPING WITH REACT

Prerequisites

- NodeJS/NPM https://nodejs.org
- A code editor https://code.visualstudio.com/
- Some editor extensions (eslint, prettier, React Native Tools)
- Some editor configuration https://code.visualstudio.com/docs/nodejs/reactjs-tutorial
- Debugger tools https://react.dev/learn/react-developer-tools

PRACTICE

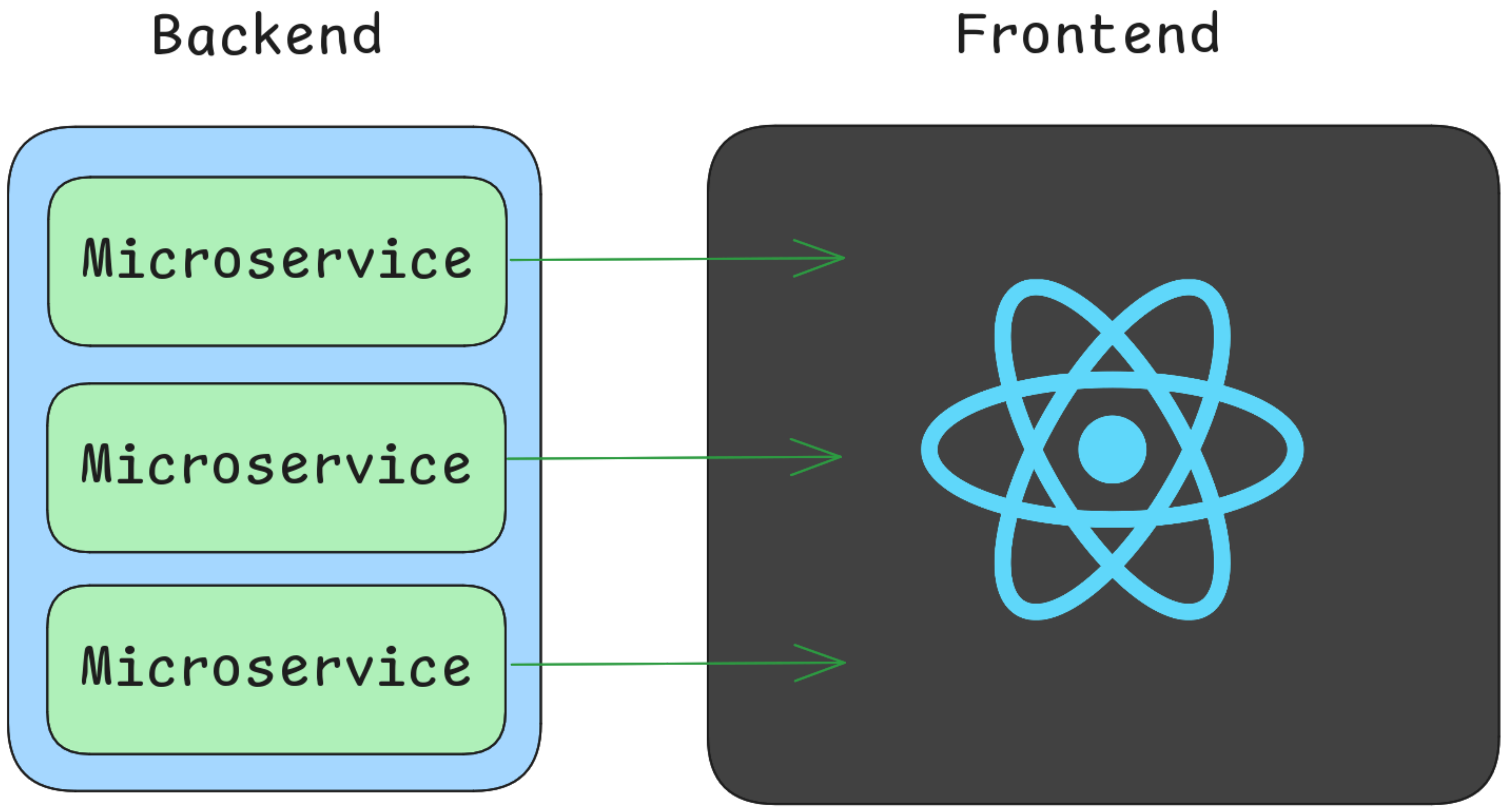
Setting up a development environment optimized for React and a first web application which will serve as a common thread for the following chapters.

[2] MODULE FEDERATION

- The concept and its role in micro-frontends architecture
- Explanation of remote and host applications, and dynamic module loading
- Webpack configuration: key setup steps using ModuleFederationPlugin
- Integration, sharing code and dependencies
- Key advantages

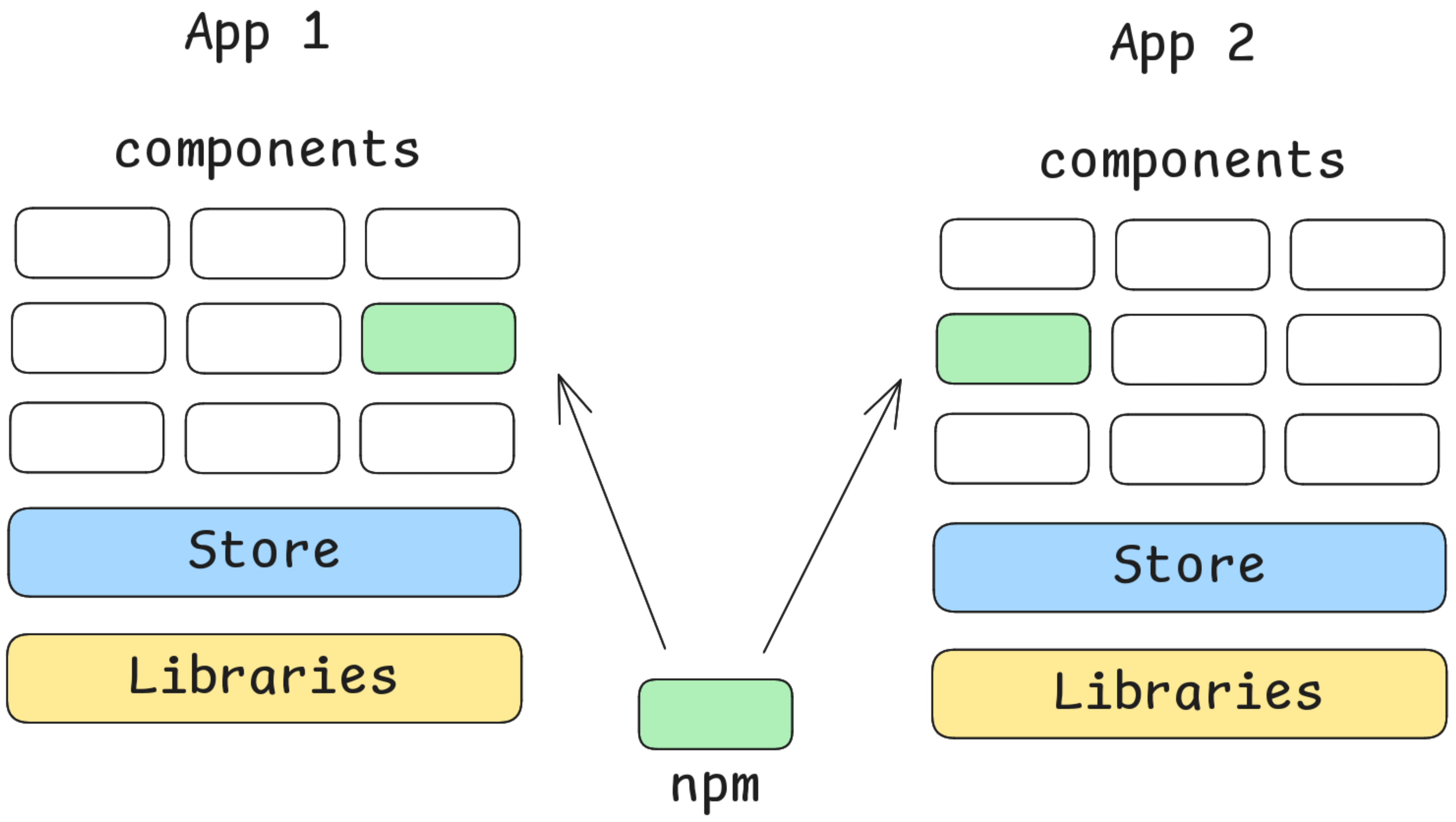
TRADITIONAL APP

Monolithic architecture, before module federation.



SHARING DEPS

Multiple push/updates to perform to **re-deploy**.



CONS OF A MONOLOTHIC FRONTEND

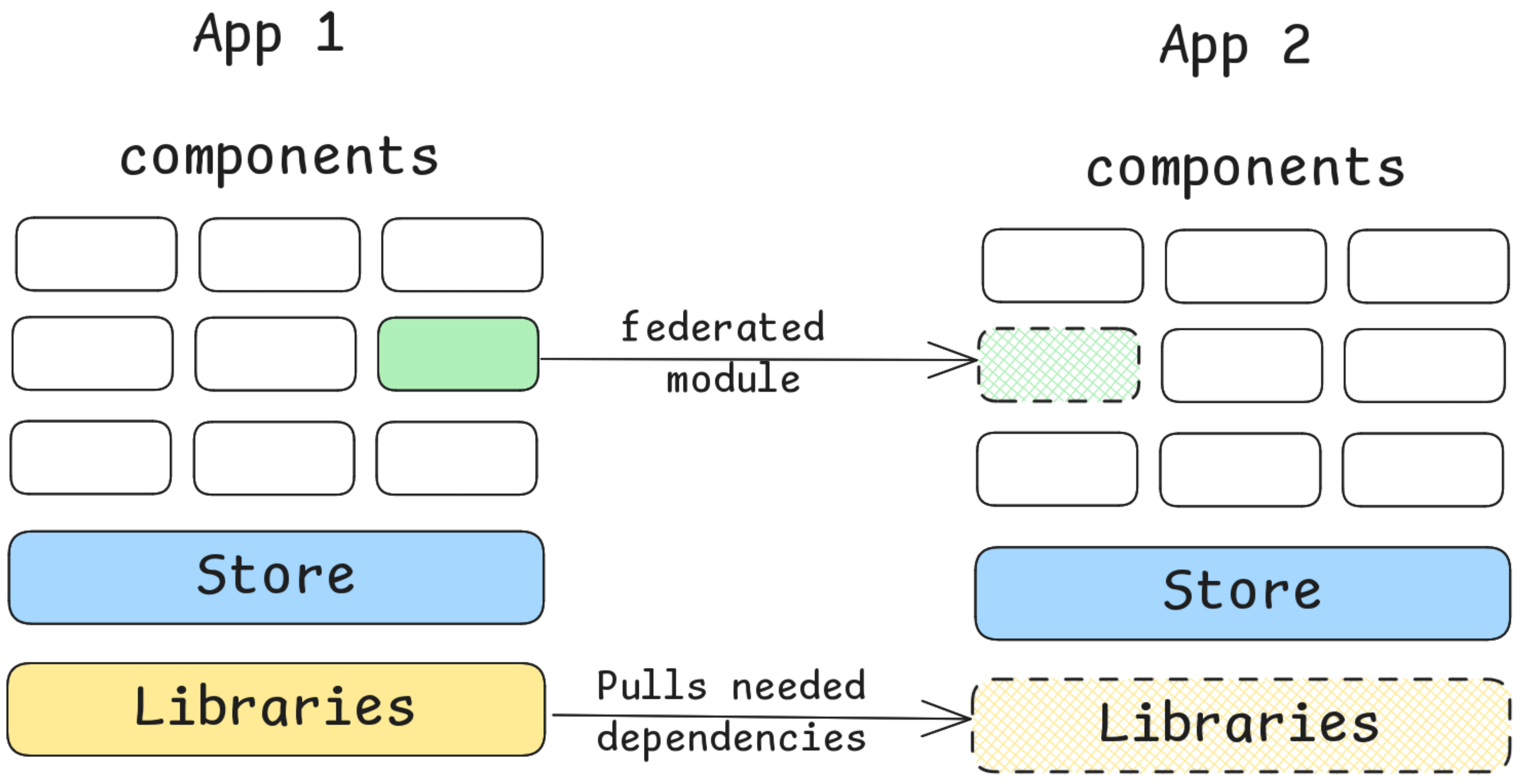
- Technology lock
- Process overhead

Now, what if we have 2 or more apps?

Push to npm, update all apps, redeploy all apps...

FEDERATED MODULE

Pull at **runtime**



WEBPACK.CONFIG.JS

Case study, 3 apps :

- App1 is served on localhost:3001
- App2 is served on localhost:3002
- Nav is served on localhost:3003

WEBPACK.CONFIG.JS - APP 1

In "remotes", we define the dependencies we import from

```
module.exports = {
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "app1",
      filename: "remoteEntry.js",
      remotes: {
        nav: "nav@http://localhost:3003/remoteEntry.js",
      },
      //...
    }),
  ],
};
```

WEBPACK.CONFIG.JS - APP 2

In "remotes", we define the dependencies we import from

```
module.exports = {
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "app2",
      filename: "remoteEntry.js",
      remotes: {
        app1: "app1@http://localhost:3001/remoteEntry.js",
        nav: "nav@http://localhost:3003/remoteEntry.js",
      },
      //...
    }),
  ],
};
```

WEBPACK.CONFIG.JS - NAV

In "exposes", we define the dependencies to expose to other apps

```
module.exports = {
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "nav",
      filename: "remoteEntry.js",
      remotes: {},
      exposes: {
        "./Header": "./src/Header",
      },
      //...
    }),
  ],
};
```

KEY ADVANTAGES

- Code sharing becomes easier
- Same look and feel accross different apps
- Automatic dependencies reload via "exposes"
- Deploy one app to update others

PRACTICE

Setup a module federation based app.

[3] ADVANCED TECHNIQUES & DESIGN PATTERNS

- Higher-Order Components (HOC) pattern
- Rendering into remote DOM elements with Portals
- Dependency injection with Contexts
- React Hooks: useEffect, useState
- Custom hooks and custom Logic

HIGHER-ORDER COMPONENTS

Function that takes a **component** as **argument**, and **returns** a **new component**.

- Wraps the original component
- Can add additional functionality
- Can accept more than one argument so it can be fully customizable

```
const MyWrapper = (SomeComponent, arg1, arg2) => {
  // some code...
  return (props) => {
    return <SomeComponent {...props} />;
  };
};
```

HOC USE CASES

Higher-order components can be used to implement cross-cutting concerns in your application such as

- authentication
- error handling
- logging
- performance tracking
- and many other features..

HOC BENEFITS

- **Reusability**: Reuse component logic across multiple components.
- **Flexibility**: Flexible way to add functionality to your components.
- **Separation of concerns**: Encapsulating functionality in a separate component.
- **Composition**: HOCs can be composed together to create more complex functionality.

PORTALS

Technique that allows you to render a part of a component oustide its own tree.

```
<!DOCTYPE html>
<html lang="en">
  <head></head>
  <body>
    <div id="root"></div>
    <div id="my-portal-1"></div>
    <div id="my-portal-2"></div>
  </body>
</html>
```

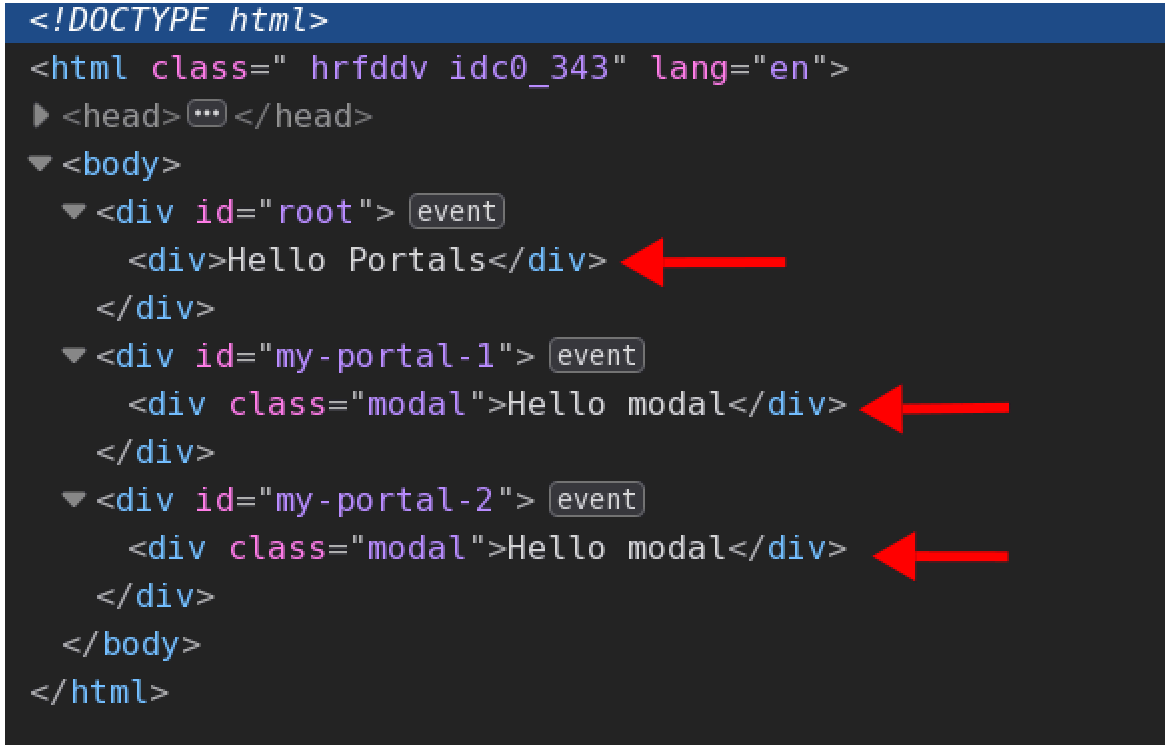
CREATEPORTAL

Method from react-dom

```
import ReactDOM from "react-dom";

function App() {
  const byId = document.getElementById;

  return (
    <>
      <div>Hello Portals</div>

      {ReactDOM.createPortal(<Modal />, byId("my-portal-1"))}
      {ReactDOM.createPortal(<Modal />, byId("my-portal-2"))}
    </>
  );
}
```

PORTAL: REAL DOM

At runtime, the generated DOM will be the following:



```
<!DOCTYPE html>
<html class=" hrfddv idc0_343" lang="en">
 <head>…</head>
 <body>
   <div id="root"> event
     <div>Hello Portals</div>
   </div>
   <div id="my-portal-1"> event
     <div class="modal">Hello modal</div>
   </div>
   <div id="my-portal-2"> event
     <div class="modal">Hello modal</div>
   </div>
 </body>
</html>
```

PORTAL USE CASES

Basically usefull for

- Moving out of the parent's z-index
- Dialogs
- Modals
- Tooltips
- Loading indicators
- ...

DEPENDENCY INJECTION WITH CONTEXTS

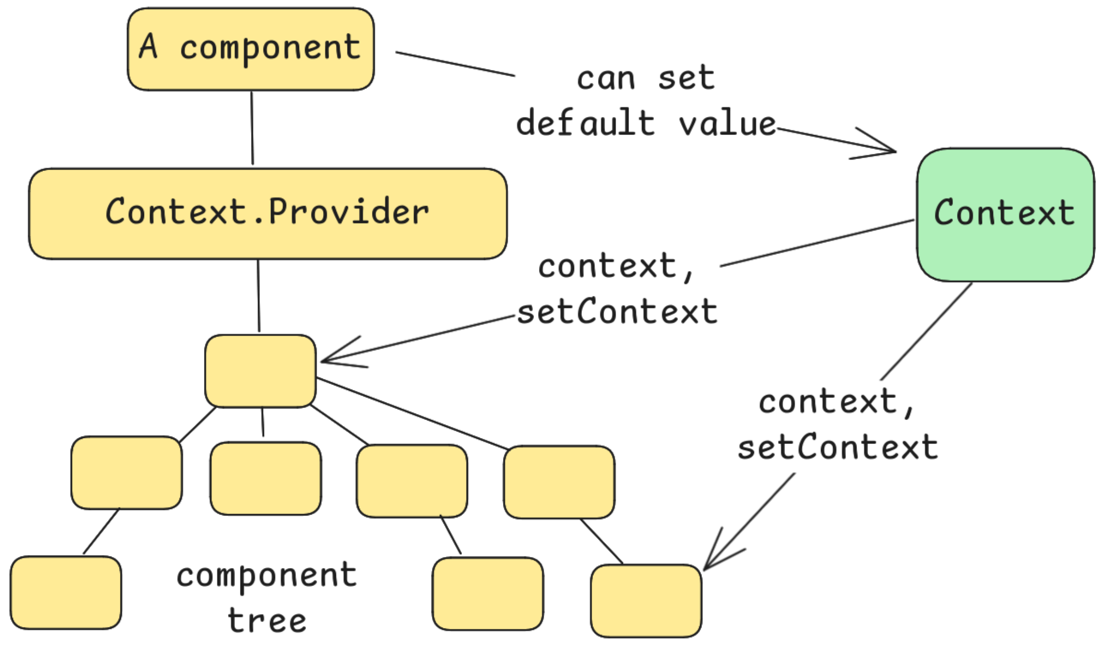
Passing props through a whole component tree is verbose and not that user friendly.

Context lets a parent component **provide data** to the **entire tree** below it.

Goal: automate changes

INJECT ANYWHERE

Once the **Context.Provider** has been defined, the data can be injected anywhere in the sub tree.



CONTEXT.PROVIDER

The tree has to be wrapped inside a special tag:

```
import { MyDataContext } from "./MyDataContext";

function App() {
  const [state, setState] = useState("Hello world");

  return (
    <>
      <MyDataContext.Provider value={[state, setState]}>
        <div>Hello Contexts</div>
        <Level1 />
      </MyDataContext.Provider>
    </>
  );
}
```

USECONTEXT

Inside the children nodes we can be provided with the context and a function to update the context:

```
import { useContext } from "react";
import { MyDataContext } from "./MyDataContext";

const Level1 = () => {
  let [ctx, setCtx] = useContext(MyDataContext);

  return (
    <>
      <p>Data: {ctx} </p>
      <Level2 />
    </>
  );
};
```

REACT HOOKS

React comes with different built-in hooks

- **State** hooks
- **Context** hooks
- **Ref** hooks
- **Effects** hooks
- **Performance** Hooks
- And also allows you to write custom ones

STATE HOOK

useState lets you add a **state variable** to your component

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

const State = () => {
  let [state, setState] = useState(0);

  let increment = () => {
    setState(state + 1);
  };

  return <button onClick={increment}>Count: {state}</button>;
};

export default State;
```

EFFECT HOOK

useEffect lets you synchronize a component with an external system

For example

- Websokets
- SSE
- RxJS (observables, ...)
- Any asynchronous message system

USEEFFECT

Synopsis:

```
useEffect(setup, dependencies?)
```

The **setup** argument is a method that will handle **connection** and **disconnection**.

The **dependencies** arguments is an array of variables that mostly comes from a **state** or a **context**, can be an empty array.

USEEFFECT EXAMPLE

In this example, we connect to a websocket and clear the connection (when unmounted)

```
export default function App() {
  const ref = useRef(null);

  useEffect(() => {
    // connection
    ref.ws = new WebSocket("wss://my-websocket.io/");

    // disconnection
    return () => {
      ref.ws.close();
    };
  }, []);

  return <div>Hello useEffect</div>;
}
```

CUSTOM HOOKS

A custom hook is a **function** that mainly combines one or more basic hooks to **encapsulate logic** that you would like to reuse.

Key advantages:

- Reusability
- Modularity
- Making components simpler/readable

CUSTOM HOOK USE CASES

With custom hooks, we can move out stuff from the component logic and automate things such as

- HTTP calls
- Forms validation
- On-screen notifications
- Loggers, Trackers
- Idle detection, ...

Let's study a few ones :

- https://github.com/sergeyleschev/react-custom-hooks
- https://github.com/uidotdev/usehooks

PRACTICE

Write a few custom hooks

[4] ADVANCED REDUX

- Redux core
- Review and comparison of different Redux libraries
- Simplifying and optimizing form management
- Redux persist
- Custom Redux Middleware

REDUX LIBRARIES

Redux is a predictable **state container**.

It helps manage the **application state** in a centralized location called the redux **store**.

React lets you choose between using the native **implementation** (Core) or middlewares:

Toolkit, Saga, Thunk, ...

REDUX CORE

Official bindings for react: https://react-redux.js.org/

Wraps your app within a store provider tag

```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import store from "./store";
import App from "./App";

const rootEl = document.getElementById("root");

ReactDOM.createRoot(rootEl).render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

REDUX CORE HOOKS

Redux cores comes with built-in custom hooks that helps manipulate the store

- **useSelector** reads a value from the store state and subscribes to updates
- **useDispatch** returns the store's dispatch method to let you dispatch actions (functions that you'll have to implement)

REDUX TOOLKIT

Redux is powerful, but setting it up and writing boilerplate code for actions and reducers **can be time-consuming**.

Redux Toolkit is a **set of utilities**, including a standardized way to write reducers, create actions, and configure the Redux store.

It is designed to **simplify** the development process and promote **best practices**.

REDUX TOOLKIT CREATESLICE

A simple slice for a counter

```
import { createSlice } from "@reduxjs/toolkit";
import type { PayloadAction } from "@reduxjs/toolkit";

export const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    increment: (state) => (state.value += 1),
    decrement: (state) => (state.value -= 1),
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

USE THE SLICE

In our component, we retrieve the counter and actions

```
import React from "react";
import type { RootState } from "../../app/store";
import { useSelector, useDispatch } from "react-redux";
import { decrement, increment } from "./counterSlice";

export function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <>
      <button onClick={() => dispatch(decrement())}> - </button>
      <span>{count}</span>
      <button onClick={() => dispatch(increment())}> + </button>
    </>
  );
}
```

REDUX SAGA

"An intuitive Redux side effect manager"

- Built on top of redux core
- ES6 generator functions (`function*`)
- Asynchronous handy functions
- Declarative calls

REDUX SAGA EXAMPLE

Create a reducer

```
export const INCREMENT = "INCREMENT";
export const DECREMENT = "DECREMENT";
export const INCREMENT_ASYNC = "INCREMENT_ASYNC";

export function reducer(state = 0, action) {
  if (action.type == INCREMENT) return state + 1;
  if (action.type == DECREMENT) return state - 1;
  return state;
}
```

REDUX SAGA GENERATOR

```
import { put, takeEvery, all, delay } from "redux-saga/effects";
import { INCREMENT, DECREMENT, INCREMENT_ASYNC } from "./reducer";

function* incrementAsync() {
  yield put({ type: INCREMENT });
  yield delay(1000);
  yield put({ type: DECREMENT });
  yield delay(1000);
  yield put({ type: INCREMENT });
}

function* watchIncrementAsync() {
  yield takeEvery(INCREMENT_ASYNC, incrementAsync);
}
export function* rootSagas() {
  yield all([watchIncrementAsync()]);
}
```

SAGA STORE

Now that we have created the reducer and the sagas root for our counter, we can create a store:

```
import { createStore, applyMiddleware } from "redux";
import createSagaMiddleware from "redux-saga";

import reducer from "./counter-reducer";
import rootSagas from "./counter-sagas";

const sm = createSagaMiddleware();
export const store = createStore(reducer, applyMiddleware(sm));
sagaMiddleware.run(rootSagas);
```

REDUX THUNK

Thunk allows writing **functions** with **logic** inside that can **interact** with a Redux store's **dispatch** and **getState** methods.

What is a "thunk"?

"thunk" is a programming term that means "a piece of code that does some **delayed** work"

=> Asynchronous

https://github.com/reduxjs/redux-thunk

REDUX THUNK

Enable thunk middleware

```
import { createStore, applyMiddleware } from "redux";
import { thunk } from "redux-thunk";
import rootReducer from "./reducers/index";

const store = createStore(rootReducer, applyMiddleware(thunk));
```

THUNK - API CALL EXAMPLE

We define the service

```
import { createApi } from "@reduxjs/toolkit/query/react";
import { fetchBaseQuery } from "@reduxjs/toolkit/query/react";

export const usersApi = createApi({
  reducerPath: "usersApi",
  baseQuery: fetchBaseQuery({ baseUrl: "/api/v1/" }),
  endpoints: (builder) => ({
    getPokemonById: builder.query({
      query: (id: string) => `users/${id}`,
    }),
  }),
});

export const { useGetUserByIdQuery } = usersApi;
```

THUNK - API CALL EXAMPLE - 2

We configure the redux store

```
import { configureStore } from "@reduxjs/toolkit";
import { setupListeners } from "@reduxjs/toolkit/query";
import { usersApi } from "./services/users";

export const store = configureStore({
  reducer: {
    [usersApi.reducerPath]: usersApi.reducer,
  },
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware().concat(usersApi.middleware);
  },
});
```

THUNK - API CALL EXAMPLE - 3

And use we it inside our component

```
import { useGetPokemonByNameQuery } from "./services/pokemon";

export default function User() {
  const { data, error, isLoading } = useGetUserByIdQuery(123);

  if (isLoading) return <p>Loading..</p>;
  if (error) return <p>Ooops..</p>;
  return <p>{data}</p>;
}
```

REDUCE THUNK - ASYNC

```
const INCREMENT_COUNTER = "INCREMENT_COUNTER";

function increment() {
  return {
    type: INCREMENT_COUNTER,
  };
}

function incrementAsync() {
  return (dispatch) => {
    setTimeout(() => {
      dispatch(increment());
    }, 1000);
  };
}
```

REDUX FORM MANAGEMENT

Library that provides reducers, HOC, components

https://redux-form.com

- Simplifies and optimizes form management
- Simplifies custom components
- Uses the redux store to save the form state

CREATE REDUX STORE

Create a redux store for the redux form

```
import { createStore, combineReducers } from "redux";
import { reducer as formReducer } from "redux-form";

// combine all reducers to create a redux store
const reducers = combineReducers({ form: formReducer });

// create redux store using above combined reducers
const store = createStore(reducers);

// export redux store to use it in the application
export default store;
```

REGISTER REDUX FORM

Now create the form

```
import React from "react";
import { Field, reduxForm } from "redux-form";

const MyForm = (props) => {
  const { handleSubmit, pristine, reset, submitting } = props;

  return (
    <form onSubmit={handleSubmit}>
      <Field name="name" component="input" type="text" />
      <Field name="email" component="input" type="email" />
    </form>
  );
};

// a unique identifier for this form
export default reduxForm({ form: "myForm" })(MyForm);
```

USE THE FORM

We can insert our form inside the app

```
import React, { Component } from 'react';
import { Provider } from "react-redux";
import store from "./store";
import MyForm from "./my-form";

function App {
  const handleSubmit = (values) => {
    console.log(values);
  }
  return (
    <Provider store={store}>
        <MyForm onSubmit={handleSubmit} />
    </Provider>
  );
}

export default App;
```

REDUX PERSIST

State management tool that allows the state in a Redux store to persist across browser and app sessions

https://github.com/rt2zz/redux-persist

Uses **localStorage** as default storage (also supports **sessionStorage** and **indexedDB**)

- Data pre-loading
- Network errors handling

REDUX PERSIST

Basic setup, store configuration

```
import { createStore } from "redux";
import { persistStore, persistReducer } from "redux-persist";
import storage from "redux-persist/lib/storage";
import rootReducer from "./my-reducers";

const persistConfig = { key: "root", storage };

const persistedReducer = persistReducer(persistConfig, rootReducer);

export default () => {
  let store = createStore(persistedReducer);
  let persistor = persistStore(store);
  return { store, persistor };
};
```

REDUX PERSIST - PRELOAD

Automatic preload can be done using the PersistGate component

```
import { PersistGate } from "redux-persist/integration/react";

const App = () => {
  return (
    <Provider store={store}>
      <PersistGate loading={<MyLoader />} persistor={persistor}>
        <RootComponent />
      </PersistGate>
    </Provider>
  );
};
```

Pass a custom loader indicator to the `loading` prop, or null.

CUSTOM REDUX MIDDLEWARE

Middleware use cases:

- Create side effects for actions
- Modify or cancel actions
- Modify the input accepted by dispatch.

https://redux.js.org/usage/writing-custom-middleware

CUSTOM MIDDLEWARE EXAMPLE

Let's write a middleware that logs previous and next state for every action dispatched.

See **./demos/custom-middleware**

PRACTICE

Use redux core to share state between components.

[5] PERFORMANCE OPTIMIZATION

- React API for optimization: React.Suspense, React.Lazy, Concurrent, React.Cache
- Server-Side Rendering (SSR) with NextJS
- Component lifecycle optimization
- Immutability for speed and simplicity
- Pure Components: React.PureComponent and React.memo
- Production deployment and optimization

REACT.SUSPENSE

Shows a custom component until the children have finished loading

```
<Suspense fallback={<Loading />}>
  <SomeComponent />
</Suspense>
```

Can be nested

```
<Suspense fallback={<Loader1 />}>
  <SomeComponent />
  <Suspense fallback={<Loader2 />}>
    <SomeOtherComponent>
  </Suspense>
</Suspense>
```

REACT LAZY

`lazy` lets you defer loading component's code until it is rendered for the first time

```
import React, { lazy } from "react";
import Loader from "./Loader";

const MyComponent = lazy(() => import("./MyComponent"));

function App() {
  return (
    <React.Suspense fallback={<Loader />}>
      <MyComponent />
    </React.Suspense>
  );
}

export default App;
```

CONCURRENT MODE

Feature that enables React to **render multiple versions** of your UI **simultaneously**

Enable Concurrent Mode in package.json

```
{
  "dependencies": {
    "react": "^19.0.0",
    "react-dom": "^19.0.0"
  },
  "concurrent": true
}
```

CONCURENT MODE BENEFITS

Benefits of Concurrent Mode

- **Improved responsiveness**: ensures that your UI remains responsive even when there are long-running tasks or network requests.
- **Smoother animations**: React can render multiple frames of an animation concurrently, results in a more polished and user-friendly experience.
- **Efficient resource utilization**: Prioritize and schedule updates based on their importance and the available resources.

REACT CACHE

React's cache function helps prevent a function from being executed repeatedly with the same arguments.

```
import { cache } from "react";
import veryLongProcess from "./my-lib";
const data = [1, 2, 3 /* ... */, 99999999999];

const getResult = cache(veryLongProcess);

function MyComponent1() {
  const result = getResult(data);
  // ...
}
function MyComponent2() {
  const result = getResult(data);
  // ...
}
```

NEXTJS

React **framework** for building **full-stack** web applications.

- React Components to build user interfaces,
- Next.js for additional features and optimizations.

NEXTJS FEATURES - ROUTER

- **Routing**:
  - File-system based router
  - Server Components
  - Layouts
  - Nested routing
  - Loading states
  - error handling

NEXTJS FEATURES - RENDERING

- **Rendering**
  - Client-side and Server-side Rendering with Client and Server Components.
  - Further optimized with Static and Dynamic Rendering on the server with Next.js.
  - Streaming on Edge and Node.js runtimes.

NEXTJS FEATURES - DATA

- **Data Fetching**
  - Simplified data fetching with async/await in Server Components
  - Extended fetch API for request memoization
  - Data caching and revalidation.

NEXTJS FEATURES - OTHERS

- **Styling** Support for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS
- **Optimizations** Image, Fonts, and script optimizations
- **TypeScript** Improved support for TypeScript

SERVER-SIDE RENDERING

Also referred to as "**SSR**" or "**Dynamic Rendering**".

The page HTML is generated on each request.

Demo/code study: `demos/nextjs-ssr`

LIFECYCLE OF COMPONENTS

Each component in React has a **lifecycle** which you can monitor and manipulate during its **three** main **phases**.

The three phases are:

- **Mounting**
- **Updating**
- **Unmounting**

MOUNTING PHASE

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

The **render()** method is **required** and will always be called, the **others** are **optional** and will be called if you define them.

PRACTICE

Use memo and useCallback to prevent useless re-renders

[6] ANIMATIONS AND TRANSITIONS

- Animating components "manually" with CSS animations and transitions
- Simplifying work with React Transition Group
- Going further with the main animation libraries: comparison and implementation example

CSS TRANSITION PROPETY

Shortcut for

- transition-property
- transition-duration
- transition-timing-function
- transition-delay
- transition-behavior

```css
.my-class {
    transition:
        width 1s ease-out,
        opacity 500ms linear;
}
```

ONTRANSITIONEND

The **TransitionEnd** event is fired when the transition of the element ends.

```
<SomeComponent className="my-class"
  onTransitionEnd={transitionEndHandler}>
  This element fades out.
</SomeComponent>
```

Event properties: **propertyName**, **elapsedTime**, **pseudoElement**.

CSS KEYFRAMES

CSS Keyframes animations work the same.

Event handler: **onAnimationEnd**. Properties:

- animationName
- elapsedTime
- pseudoElement

```
<Button onAnimationEnd={handleAnimationEnd} className={classes}>
   ...
</Button>
```

ANIMATIONS: CODE STUDY

Code study: **./demo/animations** folder

ANIMATION LIBRARIES

There are several libriraies available

- React spring
- React reveal
- Framer Motion
- React Transition Group
- React pose
- React motion
- Auto animate
- and many more..

REACT TRANSITION GROUP

This library provides **4 components**:

- Transition
- CSSTransition
- SwitchTransition
- TransitionGroup

These components will **update the DOM** (classes, elements, group of elements).

RTG: TRANSITION

Allows you to manage css styles from the JavaScript code.

```
<Transition nodeRef={nodeRef} in={inProp} timeout={duration}>
  {state => (
    <div ref={nodeRef} style={{
      ...defaultStyle,
      ...transitionStyles[state]
    }}>
      I'm a fade Transition, current state: {state}!
    </div>
  )}
</Transition>
```

RTG: TRANSITION

We have to provide the 4 states (Entering, entered, exiting, exited):

```
const transitionStyles = {
  entering: { opacity: 1 },
  entered:  { opacity: 1 },
  exiting:  { opacity: 0 },
  exited:  { opacity: 0 },
};
```

The "in" property accepts a boolean that will trigger enter and exit states.

RTG: CSSTRANSITION

Built upon the Transition component.

WIll **automatically add CSS classes** based on the root class name. We just have to define them:

```
.my-el-enter {  opacity: 0; }
.my-el-enter-active {  opacity: 1;  transition: opacity 200ms;}
.my-el-exit {  opacity: 1;}
.my-el-exit-active {  opacity: 0;  transition: opacity 200ms;}
```

```
<CSSTransition nodeRef={nodeRef} in={inProp} timeout={200}
  classNames="my-el">
  <div ref={nodeRef}>
    Will receive my-el-* class names
  </div>
</CSSTransition>
```

RTG: SWITCHTRANSITION

Wraps a Transition or CSSTransition component. Will wait for the child to finish its animation and will render a new one.

```
<SwitchTransition mode="in-out">
  <CSSTransition
    key={state}
    nodeRef={nodeRef}
    addEndListener={endListenerHandler}
    classNames='my-class-name'>
    <button ref={nodeRef} onClick={toggleState}>
      {state ? "State is true" : "State is false"}
    </button>
  </CSSTransition>
</SwitchTransition>
```

RTG: TRANSITIONGROUP

Used in lists, and wraps a CSSTransition or a Transition component.

```
<TransitionGroup className="todo-list">
{items.map(({ id, text, nodeRef }) => (
    <CSSTransition
        key={id} nodeRef={nodeRef} timeout={500}
        classNames="my-class-name">
        <li ref={nodeRef}>
            <button onClick={removeItemHandler}>
                Click me to remove
            </button>
        </li>
    </CSSTransition>
))}
</TransitionGroup>
```

RTG: CODE STUDY

Code study: **./demo/rtg** folder

PRACTICE

Add animations to an existing app

END OF TRAINING

Thank you for your participation