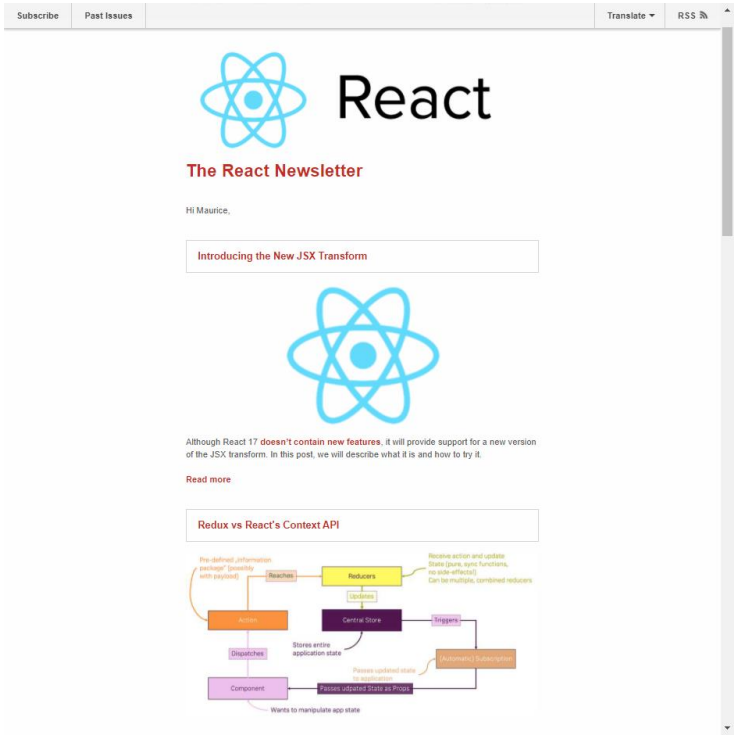
Getting started with Suspense & Concurrent Rendering

Maurice de Beijer - @mauricedb

- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Twitter: @mauricedb
- Web: http://www.TheProblemSolver.nl
- E-mail: maurice.de.beijer@gmail.com

The React Newsletter



© ABL - The Problem Solver

Workshop goal

- Use **<Suspense />**
  - Bundle splitting and lazy loading
  - Data resources
- UI **approaches**
  - Fetch from render
  - Start fetching before render
- **Concurrent mode**
  - Enabling Concurrent mode
  - Orchestrating <Suspense /> boundaries using <SuspenseList />
  - Using useTransition() to prioritize work

Type it out by hand?

"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"

© ABL - The Problem Solver

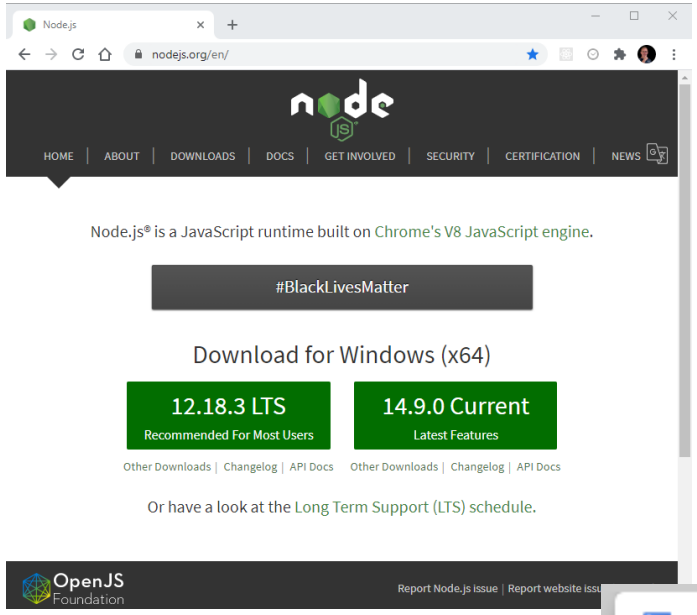
Prerequisites

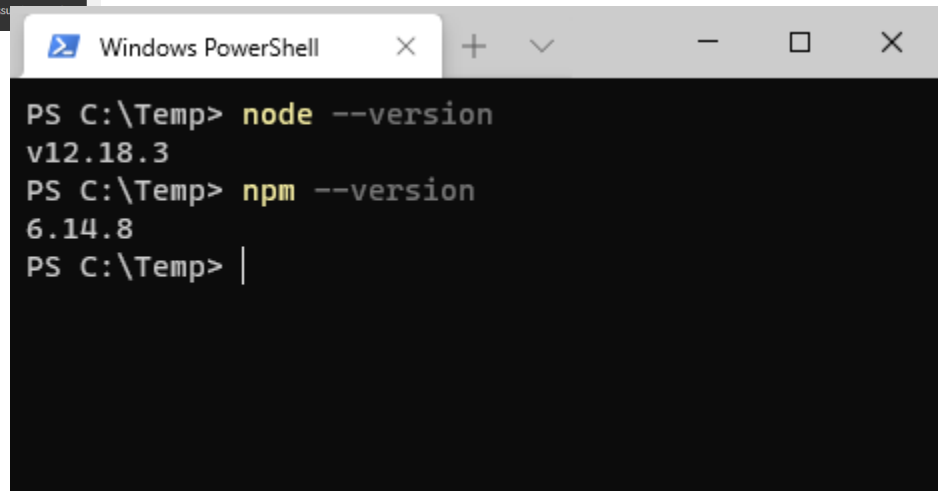
Install Node & NPM

Install the GitHub repository

© ABL - The Problem Solver

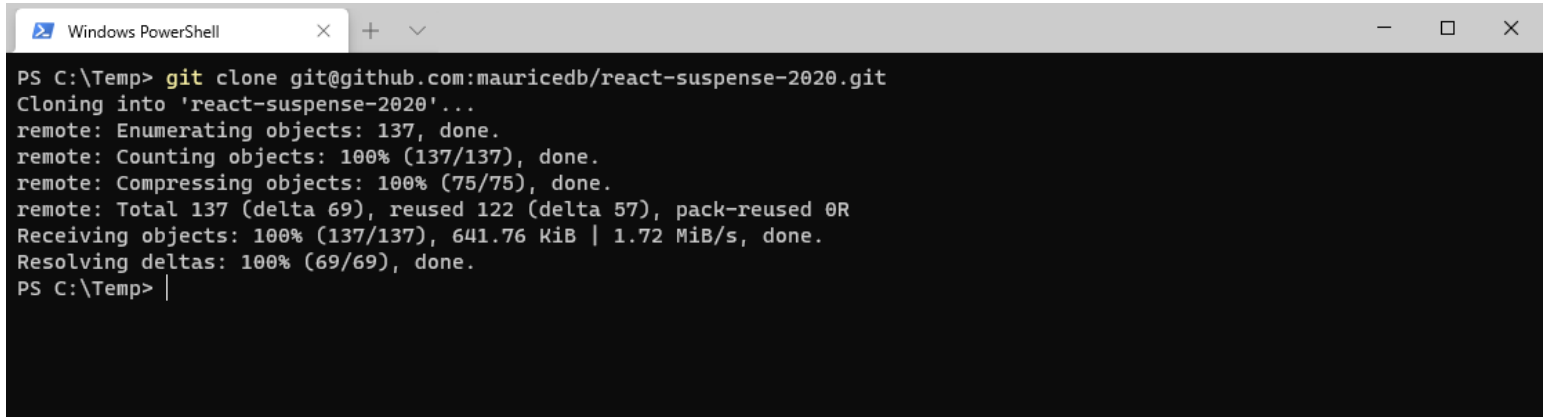
Install Node.js & NPM





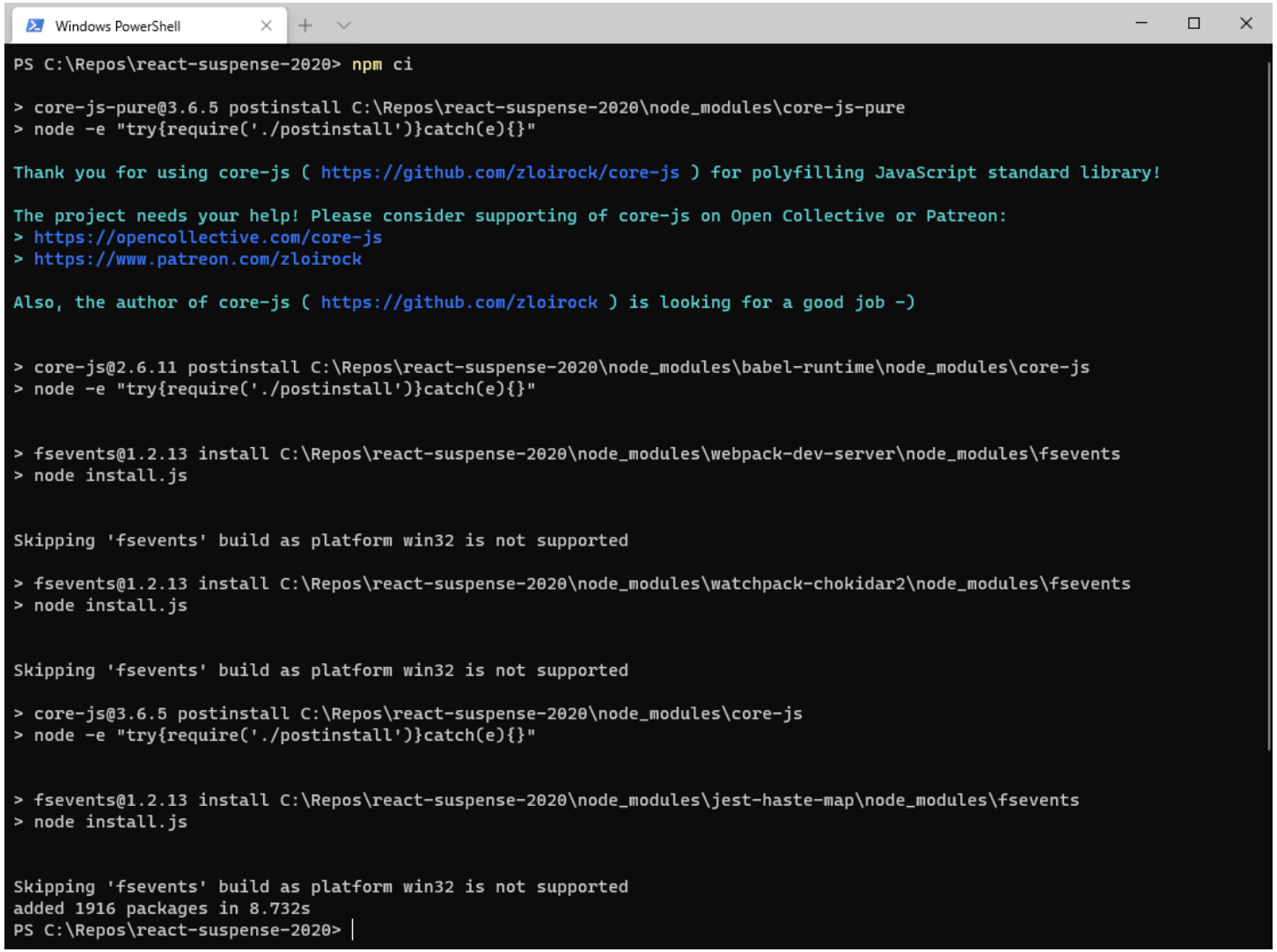
© ABL - The Problem Solver

Clone the GitHub Repository



© ABL - The Problem Solver

Install NPM Packages



© ABL - The Problem Solver

Following Along

- Repo: http://bit.ly/suspense-2020
- Slides: http://bit.ly/suspense-2020-pdf

© ABL - The Problem Solver

Bundle Splitting & Lazy Loading

Bundle Splitting & Lazy Loading

- By default CRA puts all application code in a **single code bundle**
  - All code is loaded when the application first starts
- Many **components may not be rendered** until much later
  - Or maybe never at all
- Splitting the code into multiple bundles makes **the startup faster**
  - Only load additional parts when they are actually needed
- Using **a bundle per route** is a good start
  - 👉 Beware that shared code could end up in multiple bundles
  - Import in the main bundle to really share the code
- **Using React.lazy()** automatically creates a new bundle

React.lazy()

TS App.tsx

src > TS App.tsx > ...

You, 4 days ago | 1 author (You)

```tsx
import React from "react";
import { BrowserRouter, Route, Switch } from "react-router-dom";

import NavBar from "./components/NavBar";
const Movies = React.lazy(() => import("./components/Movies"));
const Users = React.lazy(() => import("./components/Users"));
const UserDetails = React.lazy(() => import("./components/UserDetails"));

const App: React.FC = () => (
  <BrowserRouter>
    <div className="container">
      <NavBar />
      <Switch>
        <Route path="/movies">
          <Movies />
        </Route>
```

<Suspense />

<Suspense />

- Allows React to **"suspend"** rendering a component subtree
  - Used when a (grand) child component is not ready to be rendered
    - ECMAScript bundle containing the component isn't loaded yet
    - The data needed for a component to render isn't available yet
- The **"fallback" component** will be rendered instead
  - Replaces the complete children component tree
- Rendering is suspended when **a promise is thrown**
  - And resumed when the promise resolves

<Suspense />

index.tsx

src > index.tsx

```tsx
ReactDOM.render(
  <React.StrictMode>
    <React.Suspense fallback={<Loading />}>
      <ErrorBoundary
        fallbackRender={({ error, componentStack }) => (
          <ErrorDisplay error={error} componentStack={componentStack} />
        )}
      >
        <App />
      </ErrorBoundary>
    </React.Suspense>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Nesting <Suspense />

Nesting <Suspense />

- Multiple suspense components **can be nested**
- React will use **the closest parent** <Suspense /> component
  - Very useful to control what part of the UI is replaced by a fallback

Nesting <Suspense />

App.tsx

src > App.tsx > ...

```tsx
const App: React.FC = () => (
  <BrowserRouter>
    <div className="container">
      <NavBar />
      <Suspense fallback={<Loading />}>
        <Switch>
          <Route path="/movies">
            <Movies />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/user/:userId/movie/:movieId">
            <UserDetails />
          </Route>
        </Switch>
      </Suspense>
    </div>
  </BrowserRouter>
```

Data Resources

Data Resources

- **Data resources** are functions that
  - Return data that is available
  - Throw a promise when data is not available yet
    - Resolve when that data becomes available
    - Reject is there is some failure
- How the data is requested is not important
  - Just an implementation detail
- Several data loading libraries already **support this approach**
  - useSWR(), react-async
- Workshop **createResource()** based on gist by **Ryan Florence**

Creating a Resource

TS resource.ts ×

src > components > TS resource.ts > ...

You, 4 days ago | 1 author (You)

```ts
import { createFetchResource } from "../utils/createResource";

const resource = createFetchResource();

export default resource;
```

Using a Resource

Implement the Resource

- **Replace the useAbortableFetch() hook** and apply resource.read() in:
  - .\src\components\Movies.tsx
  - .\src\components\Users.tsx
  - .\src\components\UserMovieDetails.tsx
  - .\src\components\UserUserDetails.tsx

<Suspense /> & Errors

<Suspense /> & Errors

- If a suspense resource **fails** to load an **error it thrown**
  - When requesting it during the render()
- Catch the error using an **ErrorBoundary**
  - Just like other runtime errors in React lifecycle functions
- Error boundaries **can be nested**
  - Just like suspense boundaries

<Suspense /> & Errors

TS index.tsx ×

src > TS index.tsx

```tsx
ReactDOM.render(
  <React.StrictMode>
    <ErrorBoundary
      fallbackRender={({ error, componentStack }) => (
        <ErrorDisplay error={error} componentStack={componentStack} />
      )}
    >
      <App />
    </ErrorBoundary>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Parallel <Suspense />

Parallel <Suspense />

- Multiple suspense boundaries can **suspend in parallel**
  - React will suspend them all and show multiple fallback components
- If you want to **render a component while others are still loading**
- **Multiple suspending components** in a **single <Suspense/>** is also fine
  - Will resume when all resource promises are resolved

Parallel <Suspense />

```tsx
// UserDetails.tsx
// src > components > UserDetails.tsx > ...
6   const UserDetails: React.FC = () => (
7     <>
8       <Suspense fallback={<Loading />}>
9         <UserUserDetails />
10      </Suspense>
11      <Suspense fallback={<Loading />}>
12        <UserMovieDetails />
13      </Suspense>
14    </>
15  );
```

User Interface Approaches

Fetch from render

- Typically a component **requests it's data when it first renders**
  - Then displays a fallback component
  - And re-renders when the data is available

Start fetching before render

- Sometimes we can **predict data requirements** before render
  - And start the request before the component is first rendered
  - Possibly the data is already loaded when the component renders
- For example when a **user hovers over a navigation link**

- 👉 Beware: Can lead to extra AJAX requests
  - Not every hover is followed by a click on the link

Start fetching before render

NavBar.tsx

src > components > NavBar.tsx > ...

```tsx
const NavBar: React.FC = () => (
  <nav className="navbar navbar-light bg-light navbar-expand">
    <Link to="/" className="navbar-brand mb-0 h1">
      Home
    </Link>
    <div className="navbar-nav">
      <resource.NavLink
        to="/movies"
        className="nav-link"
        activeClassName="active"
        cacheKeys="https://the-problem-solver-sample-data.azurewebsites.net/top-rated-movies"
      >
        Movies
      </resource.NavLink>
      <resource.NavLink
        to="/users"
        className="nav-link"
        activeClassName="active"
        cacheKeys="https://the-problem-solver-sample-data.azurewebsites.net/accounts?sleep=1000"
      >
        Users
      </resource.NavLink>
    </div>
  </nav>
);
```

Start fetching before render

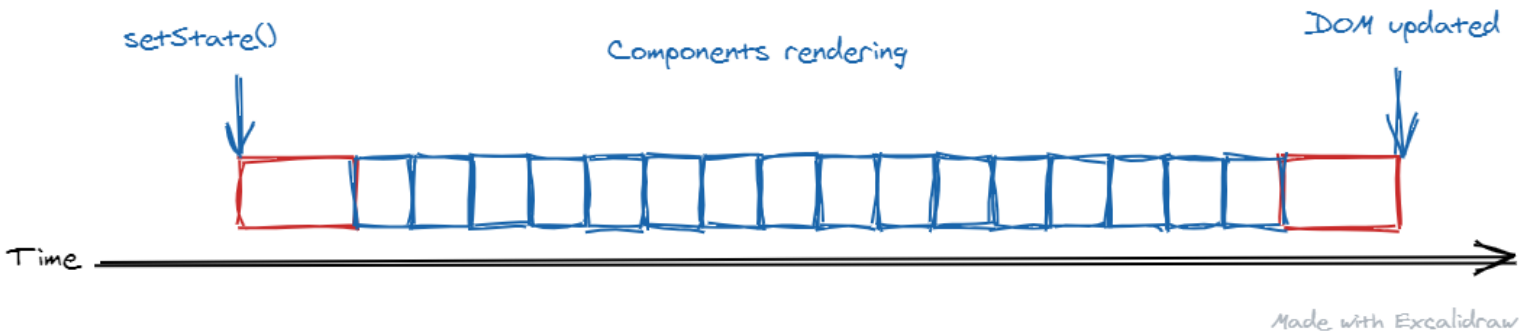
UserCard.tsx

src > components > UserCard.tsx > ...

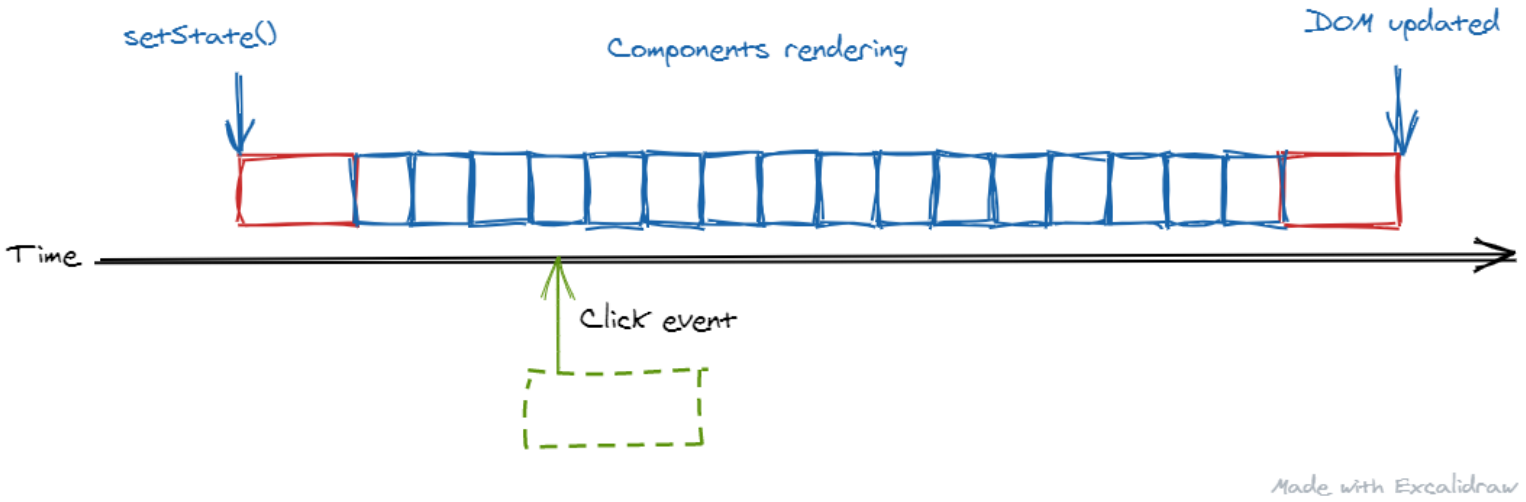
```tsx
const UserCard: React.FC<Props> = ({ user }) => (
  <div className="col mt-3">
    <div className="card h-100">
      <img src={user.picture} className="card-img-top" alt={user.firstname} />
      <div className="card-body">
        <h5 className="card-title text-truncate">{`${user.firstname} ${user.surname}`}</h5>
        <p className="card-text">{user.email}</p>
        <p className="card-text">{user.phone}</p>
      </div>
      <div className="card-footer">
        <resource.Link
          to={`/user/${user.id}/movie/${user.favorite_movie}`}
          className="btn btn-primary"
          cacheKeys={[
            `https://the-problem-solver-sample-data.azurewebsites.net/accounts/${user.id}?sleep=2000`,
            `https://the-problem-solver-sample-data.azurewebsites.net/top-rated-movies/${user.favorite_movie}?sleep=1000`,
          ]}
        >
          Details
        </resource.Link>
      </div>
    </div>
  </div>
);
```

Concurrent Mode

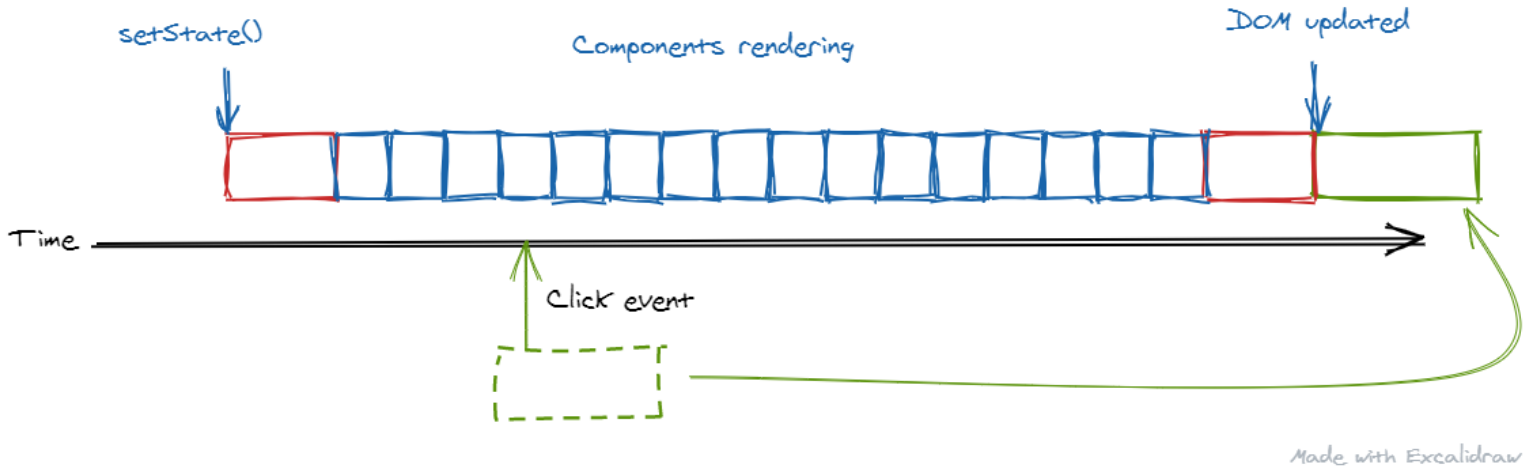
React rendering component



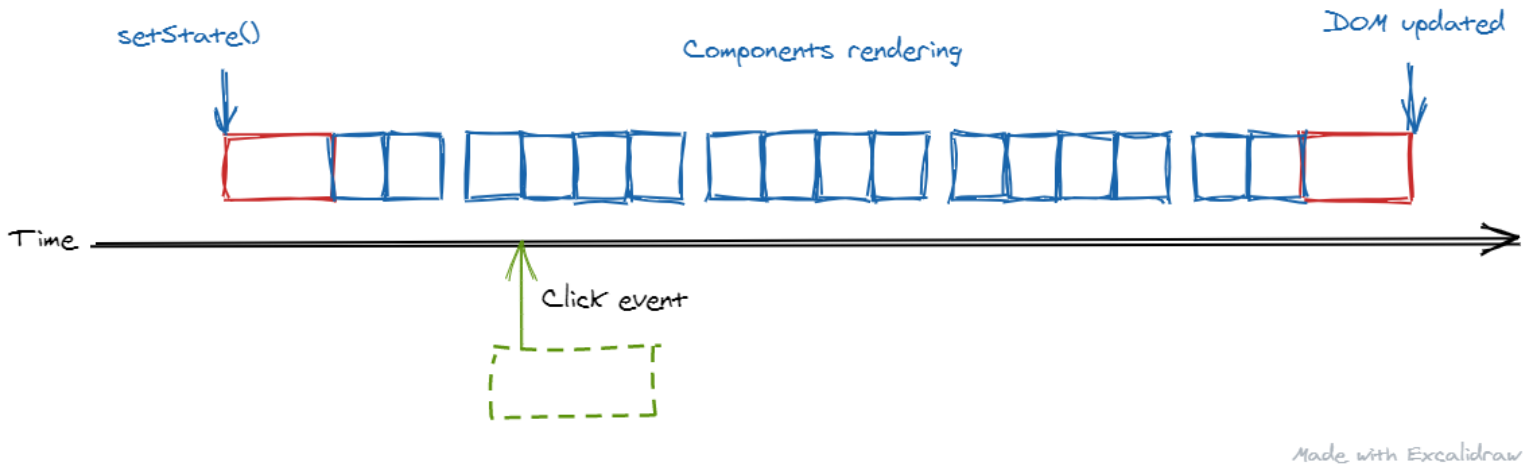
Use click event



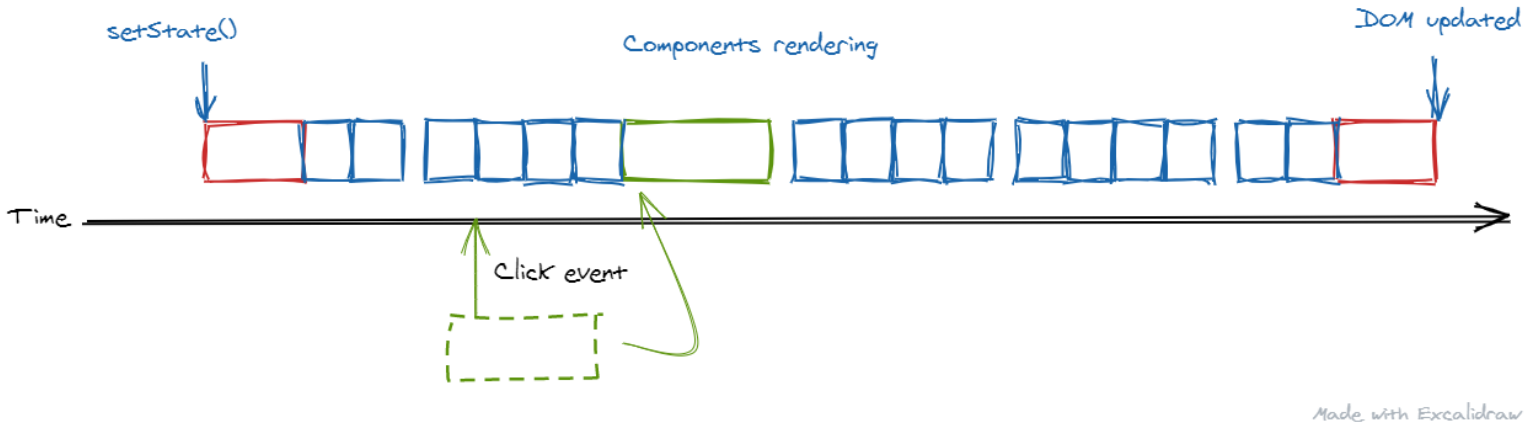
Event running



Concurrent mode



Event running with concurrent mode



Concurrent Mode

Feature Comparison

Feature Comparison

	Legacy Mode	Blocking Mode	Concurrent Mode
String Refs	✅	🚫**	🚫**
Legacy Context	✅	🚫**	🚫**
findDOMNode	✅	🚫**	🚫**
Suspense	✅	✅	✅
SuspenseList	🚫	✅	✅
Suspense SSR + Hydration	🚫	✅	✅
Progressive Hydration	🚫	✅	✅
Selective Hydration	🚫	🚫	✅
Cooperative Multitasking	🚫	🚫	✅
Automatic batching of multiple setStates	🚫*	✅	✅
Priority-based Rendering	🚫	🚫	✅
Interruptible Prerendering	🚫	🚫	✅
useTransition	🚫	🚫	✅
useDeferredValue	🚫	🚫	✅
Suspense Reveal "Train"	🚫	🚫	✅

Concurrent Mode

- **Legacy is the normal way** of rendering a React application
  - The only option that is fully supported
- **Concurrent mode** is the goal
  - Only available in experimental builds of React
- **Blocking mode** is an intermediate mode
  - Only available in experimental builds of React
- Use the **experimental type definitions** with TypeScript

- *npm install react@experimental react-dom@experimental*

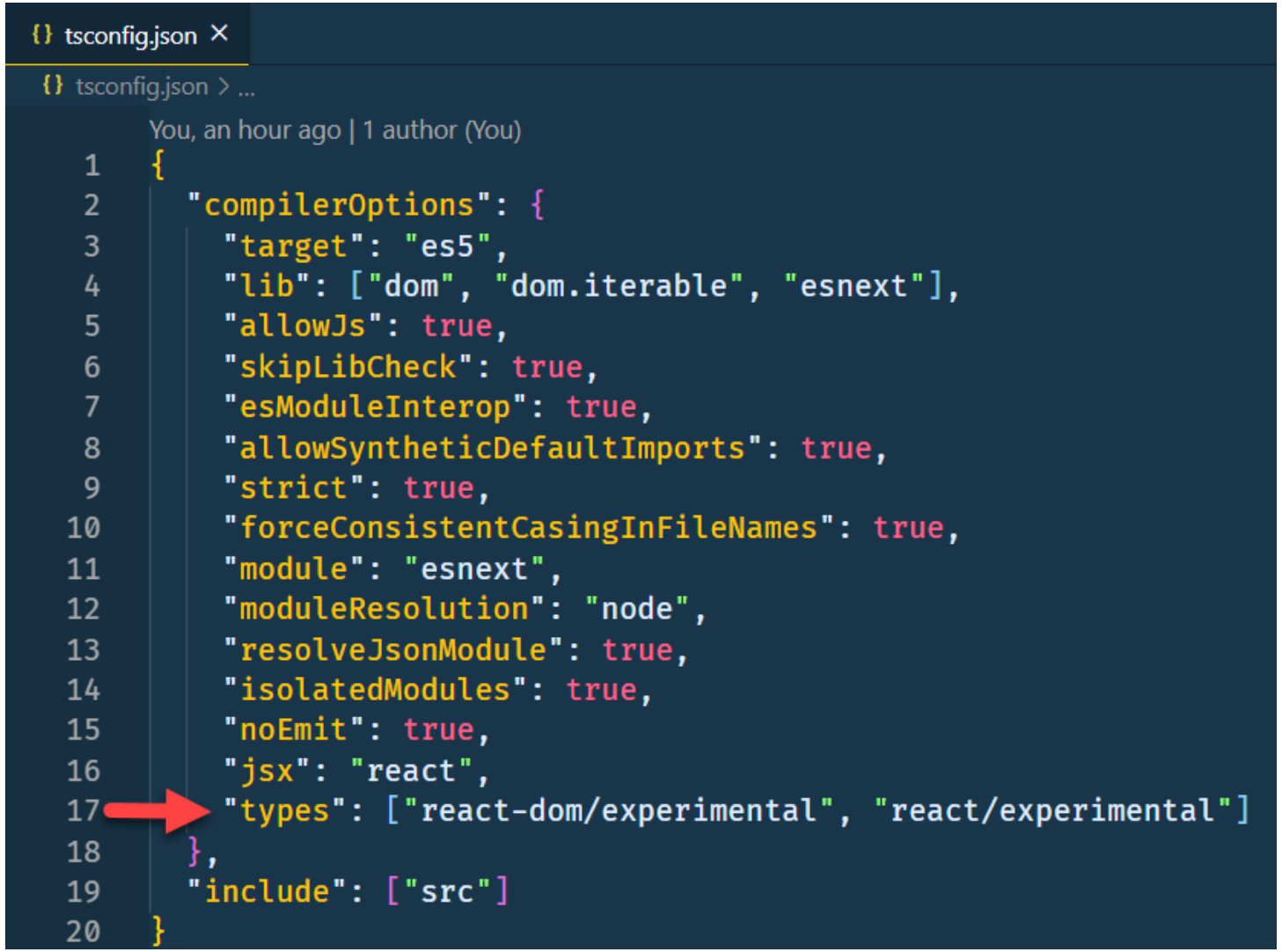
Use the Experimental React

{} package.json ×

{} package.json > {} scripts

```
4     "dependencies": {
5       "bootstrap": "4.5.2",
6       "react": "0.0.0-experimental-94c0244ba",
7       "react-dom": "0.0.0-experimental-94c0244ba",
8       "react-error-boundary": "2.3.1",
9       "react-router-dom": "5.2.0",
10      "use-abortable-fetch": "2.6.14"
11    },
```

TypeScript Definitions



Concurrent Mode

- Use **ReactDOM.createRoot()** to create a root object
  - And use it to render the React application

- Note: Still has the **"unstable_"** prefix

Using createRoot()

index.tsx

src > index.tsx

```tsx

ReactDOM.unstable_createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <Suspense fallback={<Loading />}>
      <ErrorBoundary
        fallbackRender={({ error, componentStack }) => (
          <ErrorDisplay error={error} componentStack={componentStack} />
        )}
      >
        <App />
      </ErrorBoundary>
    </Suspense>
  </React.StrictMode>
);
```

Using <SuspenseList />

Orchestrating <Suspense /> boundaries

Using <SuspenseList />

- **<SuspenseList />** will let you control how multiple **<Suspense />** components **render their fallback**
  - The order in which child components show when ready
  - If multiple child fallbacks components are displayed

Using <SuspenseList />

TS UserDetails.tsx ×

src > components > TS UserDetails.tsx > ...

```tsx
5
6  const UserDetails: React.FC = () => (
7    <React.unstable_SuspenseList revealOrder="forwards" tail="collapsed">
8      <Suspense fallback={<Loading />}>
9        <UserUserDetails />
10     </Suspense>
11     <Suspense fallback={<Loading />}>
12       <UserMovieDetails />
13     </Suspense>
14   </React.unstable_SuspenseList>
15 );
```

Using useTransition()

To prioritize work

Using useTransition()

- The **useTransition() hook** can be used to control how React renders when components suspend
  - Prevent the fallback component being rendered immediately
- The **new components will be rendered** when:
  - Their resources are ready
  - The timeout is expired
- The "old" UI can use the **isPending** state when rendering

Using useTransition()

UserUserDetails.tsx

src > components > UserUserDetails.tsx > ...

```tsx
const UserUserDetails: React.FC = () => {
  const history = useHistory();
  const { userId } = useParams<RouteParams>();
  const data = resource.read<User>(
    `https://the-problem-solver-sample-data.azurewebsites.net/accounts/${userId}?sleep=2000`
  );

  const [startTransition, isPending] = React.unstable_useTransition({
    timeoutMs: 5000,
  });

  return (
    <div>
      <div>
        <button
          className="btn btn-primary float-right"
          onClick={() => {
            startTransition(() => {
              history.push("/users");
            });
          }}
        >
          Back to users
          {isPending && (
            <span
              className="spinner-border spinner-border-sm ml-3"
              role="status"
            />
          )}
        </button>
      </div>
```

Conclusion

- You can use **<Suspense /> today**
  - Suspend when lazily load components
  - Suspend when fetching data
- Better UI **approaches** can make your application more responsive
  - Start fetching data before you render the component
  - It's a trade off
- **Concurrent mode**
  - Not recommended for production applications today
  - Can make large applications more responsive

Maurice de Beijer

@mauricedb

maurice.de.beijer
@gmail.com

© ABL - The Problem Solver