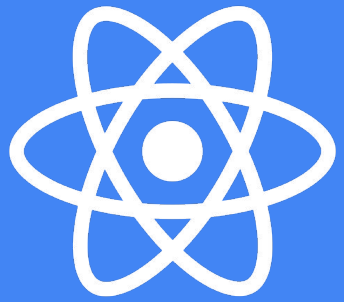


NEXT.JS

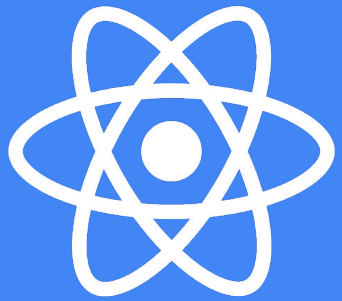
React Performance

Google

Better Web

Table of contents

Better Web

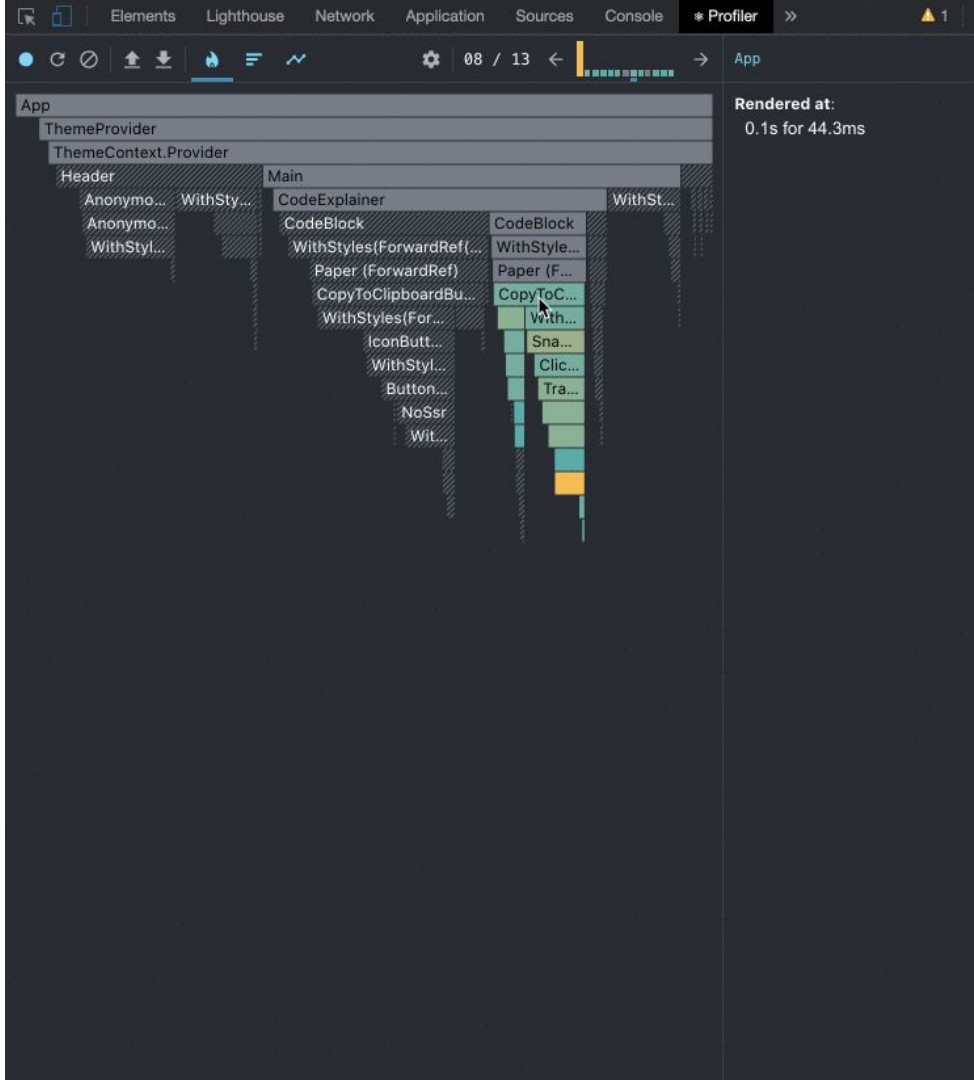


Google

Better Web

Identify re-renders with React Dev Tools

- Extension for Chrome Dev Tools
- Use **Profiler** to check for any re-rendering of components that shouldn't happen



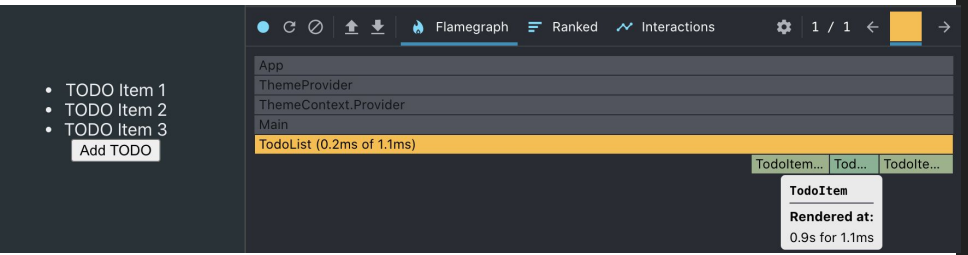
Google

Source: chrome.google.com/webstore/detail/react-developer-tools/
Learn more about auditing React: blog.logrocket.com/

Prevent unnecessary re-renders

PureComponent

- Adding/removing/editing one list item re-renders the list
- Subsequently all the list items are re-rendered
- Editing one list item shouldn't re-render all other items




```
import React from 'react'

class TodoItem extends React.Component {
  render() {
    return <li>{this.props.children}</li>
  }
}

class TodoList extends React.Component {
  constructor() {
    super()
    this.state = {
      todos: [
        {id: 123, text: "TODO Item 1"},
        {id: 456, text: "TODO Item 2"}
      ]
    }
  }
  render() {
    const todoItems = this.state.todos.map(todo => (
      <TodoItem key={todo.id}>{todo.text}</TodoItem>
    ))

    return <ul>{todoItems}</ul>
  }
}
```

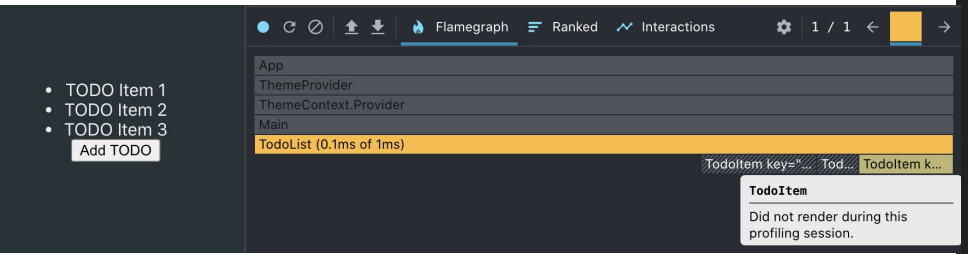

Learn more: reactjs.org/docs/react-api.html#reactpurecomponent

Prevent unnecessary re-renders

PureComponent

- By changing the list items to `PureComponent` we prevent re-renders when the parent component changes
- Re-renders *only* when its `props` have changed

```
import React from 'react'

class TodoItem extends React.PureComponent {
  render() {
    return <li>{this.props.children}</li>
  }
}

class TodoList extends React.Component {
  constructor() {
    super()
    this.state = {
      todos: [
        {id: 123, text: "TODO Item 1"},
        {id: 456, text: "TODO Item 2"}
      ]
    }
  }
  render() {
    const todoItems = this.state.todos.map(todo => (
      <TodoItem key={todo.id}>{todo.text}</TodoItem>
    ))

    return <ul>{todoItems}</ul>
  }
}
```



Learn more: reactjs.org/docs/react-api.html#reactpurecomponent

Prevent unnecessary re-renders

PureComponent

- Pure components will re-render if the ***reference*** to their props changes
- Inline functions will be re-instantiated on every render, thus creating another reference in memory
- This new function will cause an unnecessary re-render of the component

```
class Video extends React.Component {
  render() {
    return (
      <VideoPlayer>
        <VolumeControls
          onIncrease={vol => this.setState({vol})}
        />
      </VideoPlayer>
    )
  }
}
```


Learn more: reactjs.org/docs/react-api.html#reactpurecomponent

Prevent unnecessary re-renders

PureComponent

- Create a named method and pass it to the child component props
- Avoid unnamed/inline functions
- Remember to pass in props by reference

```
class Video extends React.Component {
  handleVolumeControl(volume) {
    this.setState({ volume });
  }
  render() {
    return (
      <VideoPlayer>
        <VolumeControls
          onIncrease={this.handleVolumeControl}
        />
      </VideoPlayer>
    )
  }
}
```


Learn more: reactjs.org/docs/react-api.html#reactpurecomponent

Prevent unnecessary re-renders

React.memo

- Wrap Function components in `React.memo`
- Pass, as a second argument, a function for checking if props are still the same
  - return `true` if are the same, thus prevent re-rendering
  - return `false` if should re-render

```
const ItemsCounter = React.memo(function ItemsCounter(props) {
  return (
    <span>{props.items.length} items in your basket</span>
  )
}, (oldProps, newProps) => {
  const oldItemsLength = oldProps.items.length;
  const newItemsLength = newProps.items.length;
  return oldItemsLength === newItemsLength;
});
```


Learn more: reactjs.org/docs/react-api.html#reactmemo

Prevent unnecessary re-renders

shouldComponentUpdate

- For more complex props use the `shouldComponentUpdate` lifecycle event
- Check differences in `props` and re-render the component logically
  - returning true if *should* re-render
  - opposite from `React.memo` logic

```
shouldComponentUpdate(nextProps) {
  const oldItemsLength = this.props.items.length;
  const newItemsLength = newProps.items.length;

  return oldItemsLength !== newItemsLength;
}
```

⚠️ Make sure you don't mutate data before setting the state.
You should be able to compare old and new state for this to work.

Learn more: reactjs.org/docs/react-component.html#shouldcomponentupdate

Cache expensive calculations

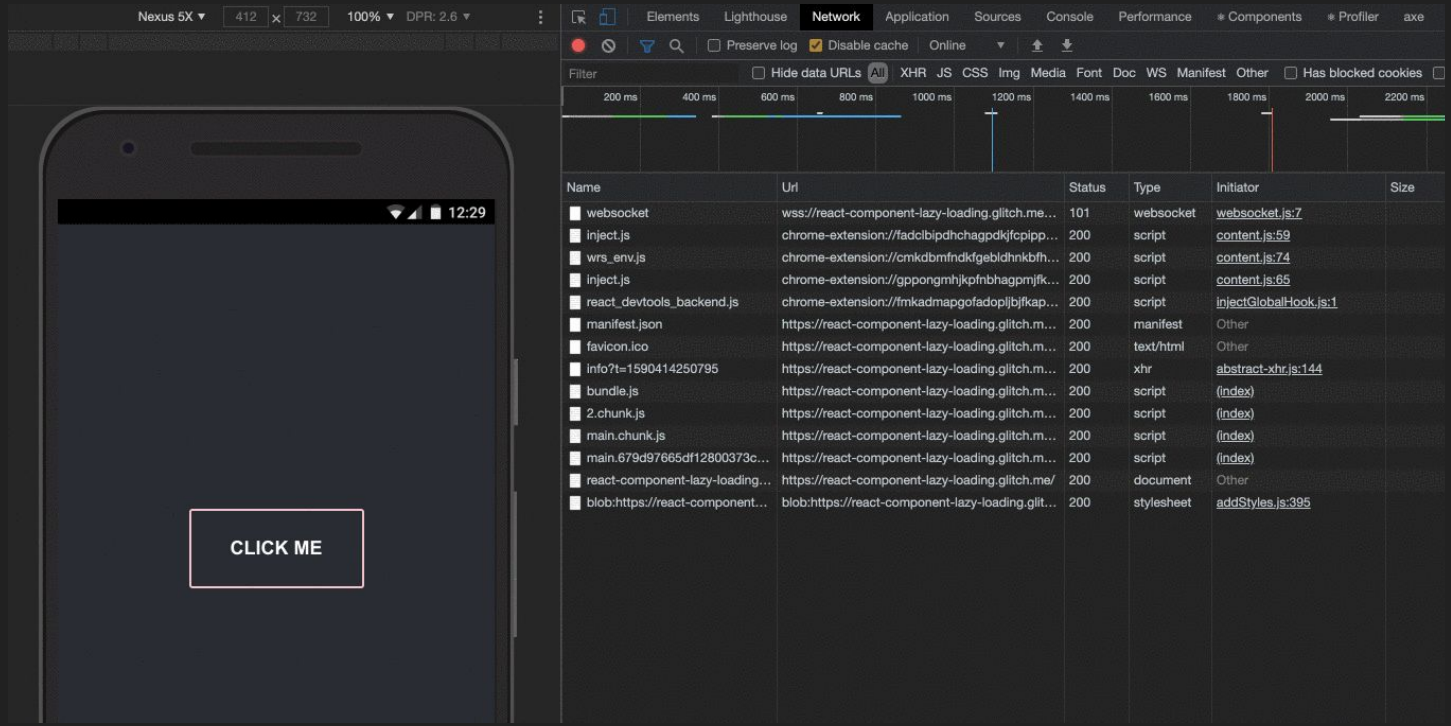
- Similar to `React.memo`, the `useMemo` hook can cache expensive calculations inside of a component
- The cache will be invalidated whenever the second argument in the `useMemo` function has changed, i.e. `item` or `value`

```
const memoizedValue = useMemo(() => {
  return computeExpensiveValue(item, value)
}, [item, value]);
```


Source: reactjs.org/docs/hooks-reference.html#usememo

Code-splitting & lazy loading

React components



Source: react-component-lazy-loading.glitch.me
Video source: recordit.co/IW8Wx8IOWa

Code-splitting & lazy loading

React components

- Divide large components into their own files to include component-specific code only
  - i.e. side menus, modals, dialogs, dropdown lists and items...
- Lazy load components after a user interaction using `React.lazy` and `React.Suspense`
- Provide a fallback during loading of the lazy component

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

function App() {
  return (
    <div className="App">
      {showModal && (
        <Suspense fallback={
          <div className="modal-fallback">
            Loading…
          </div>
        }>
          <Modal />
        </Suspense>
      )}
    </div>
  );
}
```


Try it: glitch.com/~react-component-lazy-loading
Learn more: reactjs.org/docs/code-splitting.html#reactlazy
Case study: web.dev/how-mercadolibre-optimized-web-vitals

Code-splitting & lazy loading

React Router

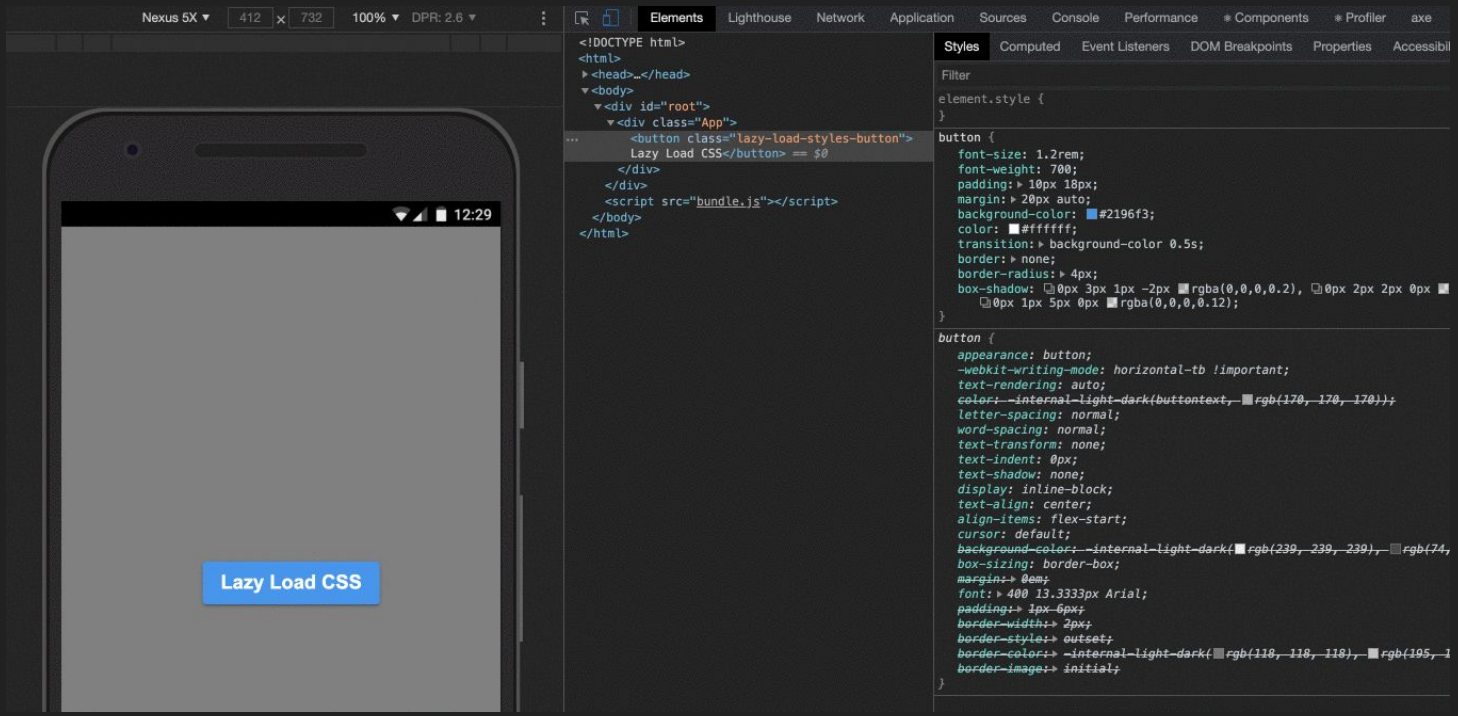
- Split files based on routes
- Route-based code-splitting is a common optimisation for SPA architectures
- React Router (`react-router-dom`) supports `React.lazy` and `React.Suspense`

```
import React, { Suspense, lazy } from 'react';
import {
  BrowserRouter as Router,
  Route,
  Switch
} from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
 <Router>
   <Suspense fallback={<div>Loading...</div>}>
     <Switch>
       <Route exact path="/" component={Home}/>
       <Route path="/about" component={About}/>
     </Switch>
   </Suspense>
 </Router>
);
```


Learn more: reactjs.org/docs/code-splitting.html#route-based-code-splitting

Code-splitting & lazy loading

CSS encapsulation



Source: react-css-lazy-loading.glitch.me
Video source: recordit.co/jsCljhLAwG

Code-splitting & lazy loading

CSS encapsulation

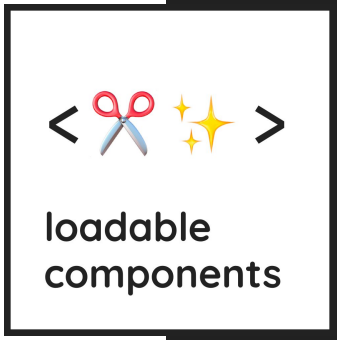
- Encapsulating component-specific CSS will enable lazy loading of CSS styles
- Bundler will create a different chunk for CSS
- Make sure to include specific styles only for each component
- Lazy load any other needed CSS upon user interaction

```
async loadStylesOnClick() {
   await import('./lazy-styles.css');
   this.setState({ lazyLoadedStyles: true });
}

render() {
   const { lazyLoadedStyles } = this.state;
   return (
     <button
       className="lazy-load-styles-button"
       onClick={this.loadStylesOnClick}
     >
       { lazyLoadedStyles ?
         "Lazy loaded 😎" :
         "Lazy Load CSS" }
     </button>
   )
}
```


Try it: glitch.com/~react-css-lazy-loading

Code-splitting & lazy loading

Server side rendering



- `React.lazy` and `Suspense` are not yet available for server-side rendering
- Use **Loadable Components** for code-splitting and component lazy loading in a server side rendered app
- Lazy load server side rendered components

```
import loadable from "loadable-components";

const Modal = loadable(
   () => import("../Components/Modal")
);
```

Source: github.com/gregberge/loadable-components
Guides: loadable-components.com/docs/server-side-rendering, marmelab.com/blog/2017/10/17/code-splitting.html#setting-up-server-side-rendering

Code-splitting & lazy loading
react-lazyload

- Get **more control** over how and when components are lazy loaded
- Support for both **one-time** lazy load and **continuous** lazy load mode
- **Throttle** or **debounce** `resize` and `scroll` events
- Server Side Rendering friendly

```
import LazyLoad from 'react-lazyload';
import MyComponent from './MyComponent';

const App = () => {
  return (
    <div className="list">
      <LazyLoad height={200} once >
        <MyComponent />
      </LazyLoad>

      <LazyLoad height={200} offset={100}>
        <MyComponent />
      </LazyLoad>
    </div>
  );
};
```

This component will be loaded when it's top edge is 100px from viewport

Github repo: github.com/twobin/react-lazyload

Code-splitting & lazy loading

react-lazyload

- Use a decorator to lazy load a component by default wherever it is being used
- The component will only be **mounted** when it's **visible in viewport**, before that a placeholder will be rendered
- Use the `once` option to lazy load the component once without detecting `scroll`/`resize` events after it has been loaded. Useful for images or simple components

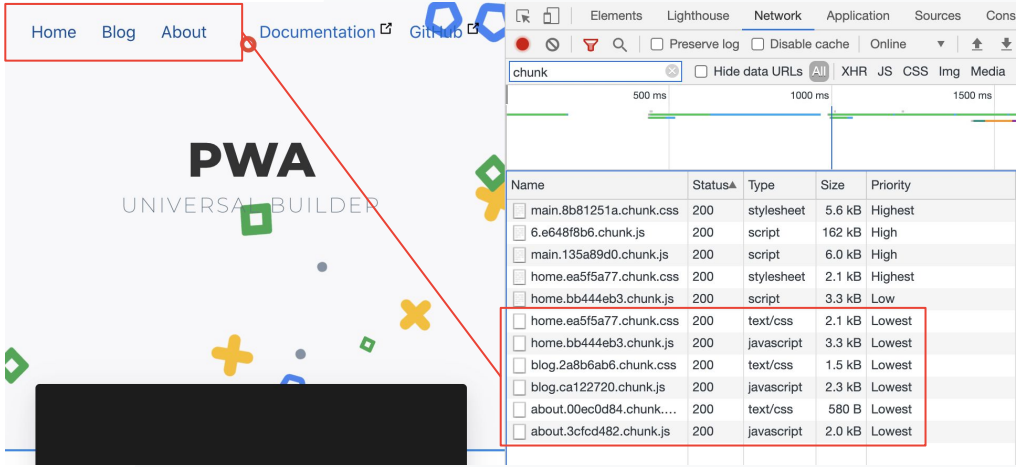
```
import { lazyload } from 'react-lazyload';

@lazyload({
   height: 200,
   once: true,
   offset: 100
})
class MyComponent extends React.Component {
  render() {
    return <div>I'm lazy-loaded by default!</div>
  }
}
```


Github repo: github.com/twobin/react-lazyload

quicklink

Prefetching with quicklink in SPAs

- Quicklink is a library to **automatically prefetch in-viewport links** to speed up navigations
- Combine the best of both techniques: Route-based code splitting and Prefetching
- **Prefetching** tells the browser to load the chunks for in-viewport links at the **lowest priority**, **during browser's idle time**



Name	Sta…	Type	Initiator	Size	Time	Priority
blog.2a8b6ab6.chunk.css	200	stylesheet	(index):1	(prefetch cache)	2 ms	Highest
blog.e7f6b4c0.chunk.js	200	script	(index):1	(prefetch cache)	2 ms	Low

Google

Learn more: web.dev/quicklink

Better Web

Prefetching with quicklink in SPAs

1. Install dependencies
2. Configure `webpack-route-manifest` to generate a manifest file associating routes with their corresponding chunks
3. Configure quicklink by wrapping each route with the `withQuicklink` higher order component

1

```
npm install webpack-route-manifest --save-dev
npm install --save quicklink
```

webpack.config.js

2

```
const RouteManifest = require('webpack-route-manifest')

const plugins = [
  new RouteManifest({
    minify: true,
    filename: 'rmanifest.json'
  }),
  ...
]
```

App.js

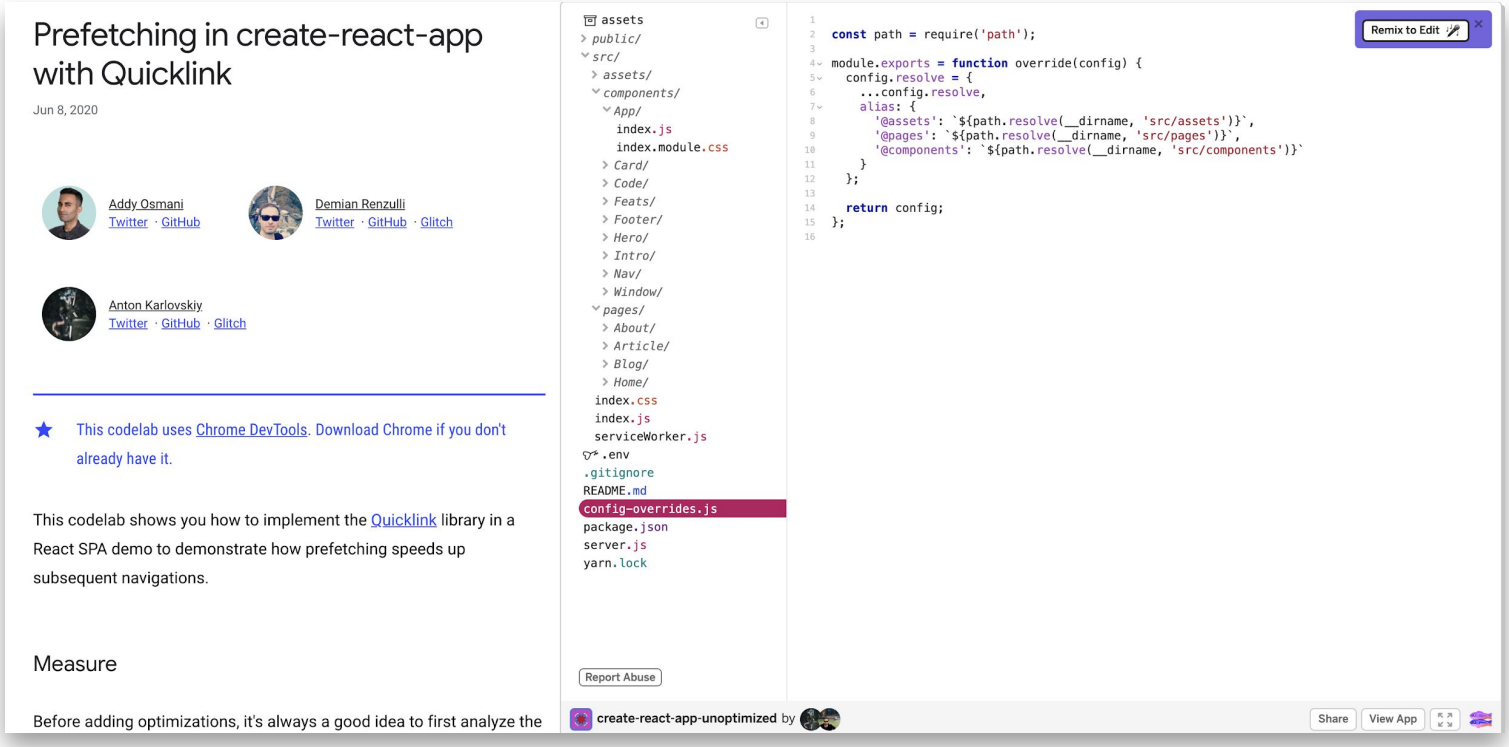
3

```
import {withQuicklink} from 'quicklink/dist/react/hoc.js'

<Route path="/" exact
  component={ withQuicklink(Home, options) } />
<Route path="/blog" exact
  component={ withQuicklink(Blog, options) } />
```

Google

Learn more: github.com/GoogleChromeLabs/quicklink#single-page-apps-react

Quicklink codelab: Try it out!



Prefetching in create-react-app with Quicklink

Jun 8, 2020

Addy Osmani
Twitter · GitHub

Demian Renzulli
Twitter · GitHub · Glitch

Anton Karlovskiy
Twitter · GitHub · Glitch

★ This codelab uses Chrome DevTools. Download Chrome if you don't already have it.

This codelab shows you how to implement the Quicklink library in a React SPA demo to demonstrate how prefetching speeds up subsequent navigations.

Measure

Before adding optimizations, it's always a good idea to first analyze the

web.dev/codelab-quicklink/

Remove `propTypes` from the production build

- React `propTypes` are only used in development
- You can save bandwidth by removing them
- Use the babel plugin

```
npm install -D babel-plugin-transform-react-remove-prop-types
```

In

```
const Component = (props) => (
  <div {...props} />
);

Component.propTypes = {
  className: PropTypes.string
};
```

Out

```
const Component = (props) => (
  <div {...props} />
);
```

Google

Source: github.com/oliviertassinari/babel-plugin-transform-react-remove-prop-types
Case study: web.dev/how-mercadolibre-optimized-web-vitals

Do Not Use Inline Function Definition

- If we are using the inline functions, every time the `render` function is called, a new instance of the function is created.
- During virtual DOM diffing, React finds a new function instance each time, so during the rendering phase, it binds the new function and leaves the old instance for garbage collection.

```
export default class Example extends React.Component {
  render() {
    return (
      <button
        onClick={() => {this.setState({clicked: true})}}
      >CLICK ME</button>
    )
  }
}
```


Learn more: cdb.reacttraining.com/react-inline-functions-and-performance-bdff784f5578

Use named functions

- Use named functions instead of inline and pass them to the component props
- Make sure you bind them to the Component, inside its constructor

```
export default class Example extends React.Component {
  constructor() {
    this.state = {
      clicked: false
    }
    this.onClickListener = this.onClickListener.bind(this)
  }

  onClickListener(e) {
    this.setState({clicked: true})
  }

  render() {
    return (
      <button
        onClick={this.onClickListener}
      >CLICK ME</button>
    )
  }
}
```

Use named functions

Avoid arrow functions

- Arrow functions seem to be a great advantage but with the benefit comes a downside
- Arrow functions are added as the object instance and not the prototype property of the class
- **Re-usability is reduced**
- When reusing components, there will be **multiple instances** for these (arrow) functions in each object created out of the component

```
export default class Example extends React.Component {
  constructor() {
    this.state = {
      clicked: false
    }
    this.onClickListener = this.onClickListener.bind(this)
  }

  onClickListener = (e) => {
    this.setState({clicked: true})
  }

  render() {
    return (
      <button
        onClick={this.onClickListener}
      >CLICK ME</button>
    )
  }
}
```

Avoid the inline style attribute

- The inline style added is a JavaScript object and not a style tag
- The process of applying the styles involves scripting and performing JavaScript execution (bidirectional object diffing)
- Scripting cost increases for each style object found in jsx

```
export default class Example extends React.Component {
  render() {
    return (
      <button
        style={{"backgroundColor": "blue"}}
      >CLICK ME</button>
    )
  }
}
```


Learn more: reactjs.org/docs/reconciliation.html

Avoid the inline style attribute

Import a CSS file

- Add a class to any element you would like to style
- And then import a CSS file into the component instead

```
import "Example.css"

export default class Example extends React.Component {
  render() {
    return (
      <button
        className={"btn btn-blue"}
      >CLICK ME</button>
    )
  }
}
```

Use a unique key for iteration

- Avoid using `index` as a key when iterating components
- When adding a new item in a list we are running the risk for React interpreting the fact that all the components have changed
- So updates are done to all the components in the list, thus reducing performance

```
export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.state.todos.map((todo, index) => (
          <li key={index}>{todo.text}</li>
        ))}
      </ul>
    )
  }
}
```


Learn more: reactjs.org/docs/reconciliation.html

Use a unique key for iteration

- Use a unique generated ID as a key instead of `index`
- Add a new ID property to your model or hash some parts of the content to generate a key
- The key only has to be unique among its siblings, not globally unique

```
export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.state.todos.map((todo) => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
    )
  }
}
```

Exception from the rule

- The list and items are static; they are not computed and they do not change
- The list is **never** reordered or filtered

Learn more: reactjs.org/docs/reconciliation.html#recursing-on-children

Virtualize long lists
with react-window

- If your application renders long lists of data you may need to use the “windowing” technique
- This technique only renders a small subset of your rows at any given time
- Dramatically reduces the time it takes to re-render the components as well as the number of DOM nodes created.

Alternative (advanced)
react-virtualized

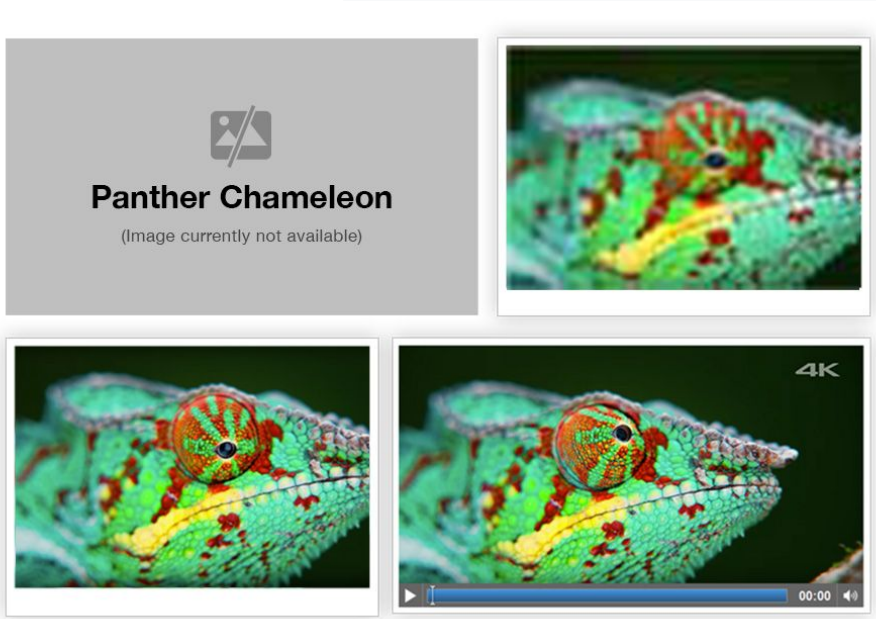
```
import { FixedSizeList as List } from 'react-window';

const Row = ({ index, style }) => (
  <div style={style}>Row {index}</div>
);

const Example = () => (
  <List
    itemCount={1000}
    itemSize={35}
  >
    {Row}
  </List>
);
```

Google

Source: react-window.now.sh/#/examples/list/fixed-size
Learn more: reactjs.org/docs/optimizing-performance.html#virtualize-long-lists

Adaptive serving

- Adapt React components based on users' network connection
- Example of component adaptations:
  - **offline**: placeholder with alt text
  - **2g** / **save-data mode**: low-res image
  - **3g**: high-res retina image
  - **4g**: HD video



Learn more: mxb.dev/blog/connection-aware-components/, dev.to/vorillaz/serving-adaptive-components-using-the-network-information-api-lbo
Effective connection types thresholds: wicg.github.io/netinfo/

Adaptive serving
react-adaptive-hooks

- Use the `react-adaptive-hooks` library to target low-end devices while progressively adding high-end-only features
- Provide the best experience best suited to user's device and network constraints

```
npm i react-adaptive-hooks
```

```
import {
  useNetworkStatus
} from 'react-adaptive-hooks/network';

import {
  useSaveData
} from 'react-adaptive-hooks/save-data';

import {
  useHardwareConcurrency
} from 'react-adaptive-hooks/hardware-concurrency';

import {
  useMemoryStatus
} from 'react-adaptive-hooks/memory';
```


Source: github.com/GoogleChromeLabs/react-adaptive-hooks

Adaptive serving

adaptive loading & code-splitting

- Code-split and lazy load components based on network information
- Use the `react-adaptive-hooks` library to determine *which* component to lazy-load based on network information
- Use `React.lazy` and `React.Suspense` to *load* the components

```
import React, { Suspense, lazy } from 'react';

import { useNetworkStatus } from 'react-adaptive-hooks/network';

const Full = lazy(() => import(/* webpackChunkName: "full" */
'./Full.js'));
const Light = lazy(() => import(/* webpackChunkName: "light" */
'./Light.js'));

const MyComponent = () => {
 const { effectiveConnectionType } = useNetworkStatus();
 return (
   <div>
     <Suspense fallback={<div>Loading...</div>}>
       { effectiveConnectionType === '4g' ? <Full /> : <Light /> }
     </Suspense>
   </div>
 );
};

export default MyComponent;
```

Google

Source: github.com/GoogleChromeLabs/react-adaptive-hooks

Better Web

Google

Code splitting

Route-based vs Component-based

- Route-based 🚀
  - Page specific code-splitting
  - On by default
- Component-based 🚀 🚀
  - Render on user interaction
  - Custom optimisation

Learn more: web.dev/code-splitting-with-dynamic-imports-in-nextjs/#route-based-and-component-based-code-splitting

Dynamic `import()`

- Dynamically import JavaScript modules
- Loading each import as a separate chunk

```
<script type="module">
  (async () => {
    const module = await import('./utils.mjs');
    module.default();
    // → logs 'Hi from the default export!'
    module.doStuff();
    // → logs 'Doing stuff…'
  })();
</script>
```


Learn more: v8.dev/features/dynamic-import

Dynamic `import()`

component-based code-splitting

- Dynamically import React components
- Loading each import as a separate chunk
- Dynamically imported components are server-side rendered by default

```
import dynamic from "next/dynamic";
const Modal = dynamic(() => import("../Modal"));

export default class Example extends React.Component {
  render() {
    const { isModalOpen } = this.state;
    return (
      <App>
        <button onClick={() => this.setState({
          isModalOpen: !isModalOpen
        })}>SHOW MODAL</button>

        { isModalOpen && <Modal /> }
      </App>
    );
  }
}
```


Learn more: web.dev/code-splitting-with-dynamic-imports-in-nextjs/#dynamic-imports-in-action

Dynamic `import()`

with custom loading indicator

- Provide a loading indicator in case there are any delays
- Supply a component as a second argument during the import

```
import dynamic from "next/dynamic";
const Modal = dynamic(() => import("../Modal"), {
  loading: () => (
    <div className="modal-loader">Loading...</div>
  )
});

export default class Example extends React.Component {
  render() {
    const { isModalOpen } = this.state;
    return (
      <App>
        <button onClick={() => this.setState({
          isModalOpen: !isModalOpen
        })}>SHOW MODAL</button>

        { isModalOpen && <Modal /> }
      </App>
    );
  }
}
```


Learn more: web.dev/code-splitting-with-dynamic-imports-in-nextjs/#dynamic-imports-with-custom-loading-indicator

Dynamic `import()`

without server-side rendering

- Lazy loaded components are SSR'd by default
- Easily turn SSR off while importing

```
import dynamic from "next/dynamic";
const Modal = dynamic(() => import("../Modal"), {
  ssr: false
});

export default class Example extends React.Component {
  render() {
    const { isModalOpen } = this.state;
    return (
      <App>
        <button onClick={() => this.setState({
          isModalOpen: !isModalOpen
        })}>SHOW MODAL</button>

        { isModalOpen && <Modal /> }
      </App>
    );
  }
}
```


Learn more: web.dev/code-splitting-with-dynamic-imports-in-nextjs/#dynamic-imports-without-ssr

Remove unused CSS
PurgeCSS

- Good fit if you are using a CSS framework like Bootstrap, Materializecss, Foundation, etc
- **PurgeCSS** can **remove unused selectors** from your CSS, resulting in smaller CSS files
- `next-purgecss` requires one of the following CSS Next plugins:

  `next-css`
  `next-less`
  `next-sass`

```
npm install @zeit/next-css next-purgecss --save-dev
```

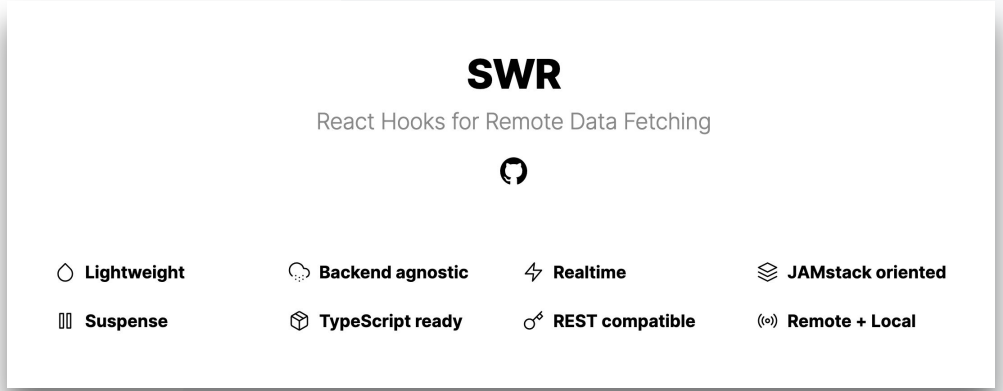
next.config.js

```
const withCss = require("@zeit/next-css");
const withPurgeCss = require("next-purgecss");

module.exports = withCss(withPurgeCss({
  purgeCss: {
    whitelist: () => ["my-custom-class"],
  },
  purgeCssPaths: [
    "pages/**/*",
    "components/**/*",
    "other-components/**/*",
  ],
}))
```


Learn more: purgecss.com/guides/next

Cache remote data using React Hooks

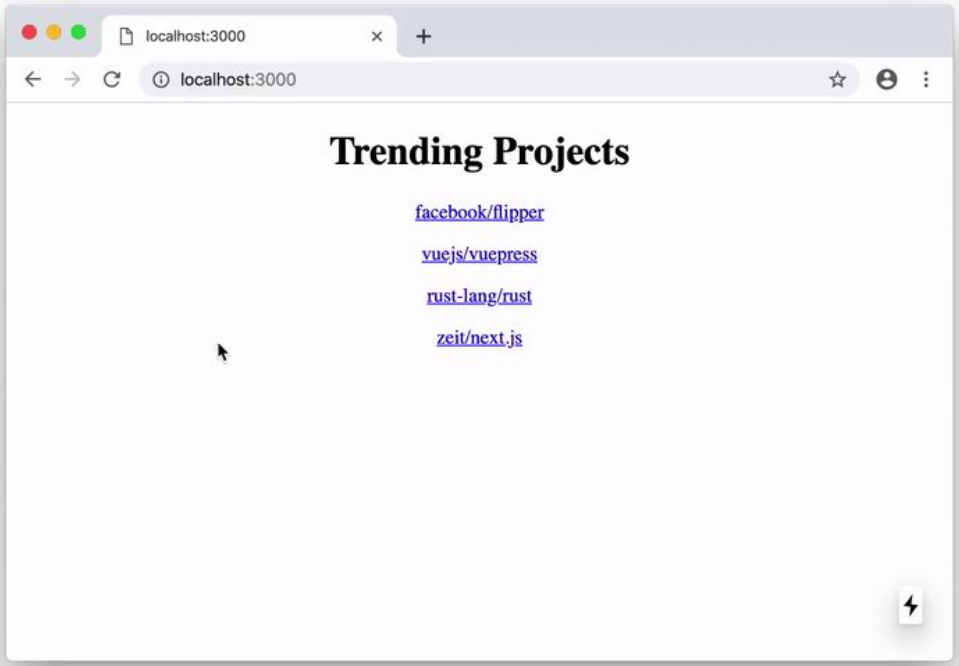
- Vercel team (formly Now and creators of Next.js) have created the **SWR** react hooks library
- **SWR** is a React Hooks library for **handling remote data fetching**
- **SWR** stands for "**stale-while-revalidate**", a HTTP cache invalidation strategy popularized by RFC 5861



Further reading
Use case article

Google

Sources: swr.now.sh, tools.ietf.org/html/rfc5861



Video source: assets.vercel.com/video/upload/q_auto,e_accelerate:70/v1572278352/swr/example-videos/fast-navigation.mp4

Cache remote data
useSWR

- SWR will make cached pages render much faster, then update the cache with the latest data in the background
- Navigations or interactions based on fetching remote data, will feel much faster in subsequent requests

```
import fetch from 'isomorphic-unfetch'
import useSWR from 'swr'
import Profile from './Components/Profile'

const fetcher = async function (...args) {
  const res = await fetch(...args)
  return res.json()
}

function App() {
  const { data } = useSWR('/api/user', fetcher);
  return (
    <div>
      {data ?
        <Profile user={data} /> :
        <div>loading...</div>
      }
    </div>
  )
}
```


Source: swr.now.sh/#fast-navigation
Github repo: github.com/vercel/swr

Cache remote data

useSWR + Suspense

- Use SWR Hooks with React Suspense
- Just enable `suspense: true` in the SWR config

```
import { Suspense } from 'react'
import useSWR from 'swr'

function Profile() {
  const { data } = useSWR('/api/user', fetcher, {
    suspense: true }
  )
  return <div>hello, {data.name}</div>
}

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


Source: swr.now.sh/#suspense

Cache remote data

useSWR + Server render

- Combine Next.js `getServerSideProps` with the SWR `initialData` option to support Server-Side Rendering
- The application will fetch the data server-side and then receive it as props
- That data will be passed as `initialData` to SWR
- Once the application starts client-side, SWR will revalidate it against the API and update the DOM, if it's required, with the new data

```
import useSWR from 'swr'

function Profile({ initialData }) {
  const { data } = useSWR('/api/user', fetcher, {
    initialData
  })
  return <div>hello, {data.name}</div>
}

export async function getServerSideProps() {
  const data = await fetcher(URL)
  return { props: { initialData: data } }
}
```


Example: github.com/vercel/swr/tree/master/examples/server-render

Cache remote data

useSWR + prefetch/preload

- If in a browser, run the fetch and mutate outside the component
- Use a `<link preload>` to get the browser load the data while rendering the HTML
- When the user moves the mouse over a link trigger a fetch and mutate for the next page

```
if (typeof window !== 'undefined') prefetchWithProjects()

export default () => {
  const { data } = useSWR('/api/data', fetch)
  function handleMouseEnter(event) {
    prefetchItem(event.target.getAttribute("href").slice(1))
  }

  return (<>
    <Head>
      <link preload="/api/data" as="fetch" />
    </Head>
    <div>{data ? data.map(project =>
      <Link href='/[user]/[project]' as={`/${project}`}>
        <a onMouseEnter={handleMouseEnter}>{project}</a>
      </Link>
      ) : 'loading...'}
    </div>
  </>)
}
```


Full example: github.com/vercel/swr/tree/master/examples/prefetch-preload

Automatic Prefetching
by default

- Next.js prefetches **only links that appear in the viewport**. Uses Intersection Observer API to detect them
- Dynamically injects `<link rel="preload">` tags to download components for subsequent navigations
- **Disabled** when **network connection is slow** or users have **data-saving** option turned on

```
<Link href="/pineapple-pizza">
   <a>Pineapple pizza</a>
</Link>
```

⚠ Next.js only fetches the JavaScript; **it doesn't execute it**

⚠ You can see this in action only in **production mode**.

Learn more: web.dev/route-prefetching-in-nextjs/#how-automatic-prefetching-works

Prevent automatic prefetching

- Disable prefetching for rarely visited pages
- Save user's bandwidth and money (network data)

```
<Link href="/pineapple-pizza" prefetch={false}>
   <a>Pineapple pizza</a>
</Link>
```


Learn more: web.dev/code-splitting-with-dynamic-imports-in-nextjs/#dynamic-imports-without-ssr

Thanks!

Google