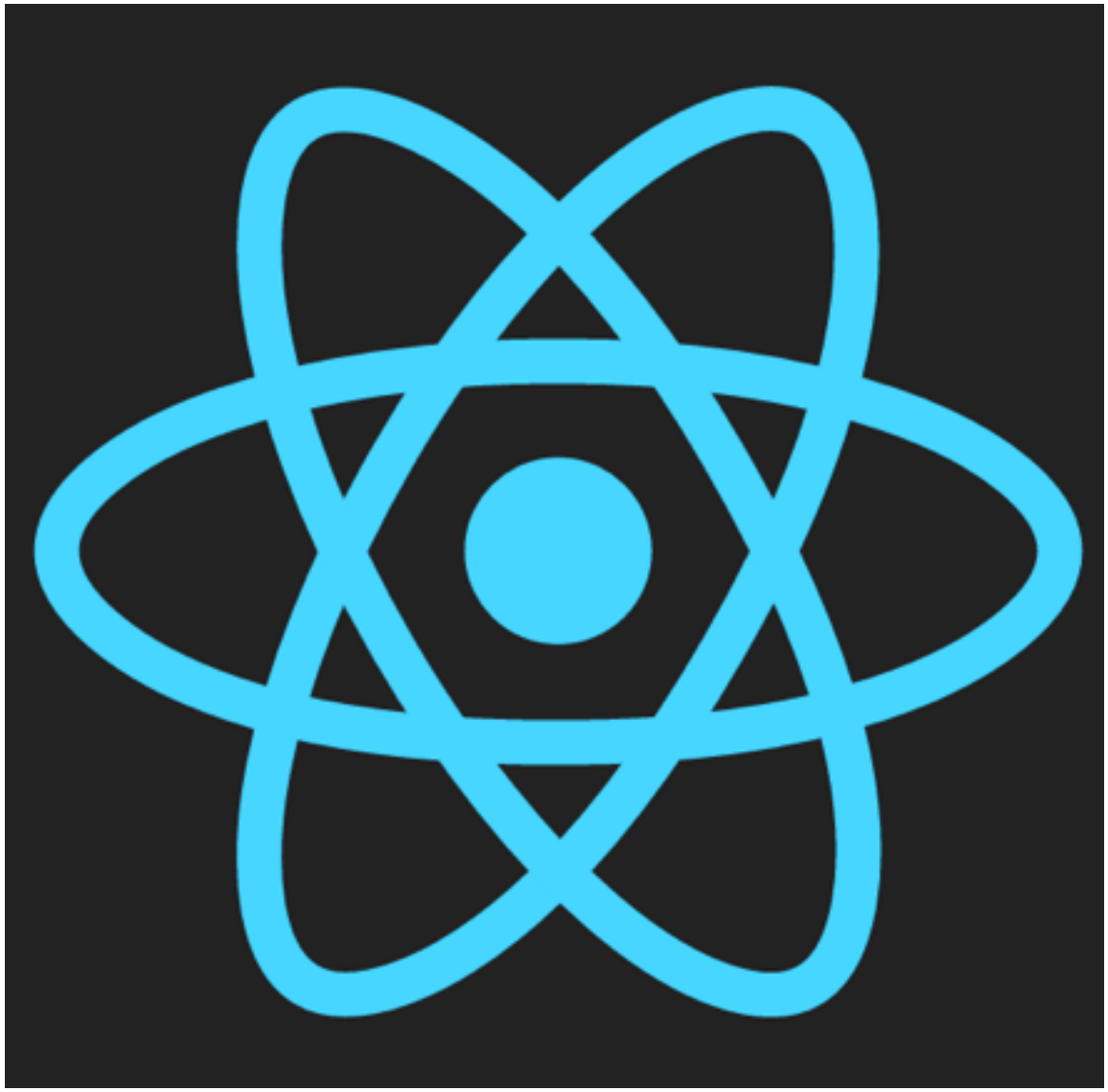
Advanced React Development



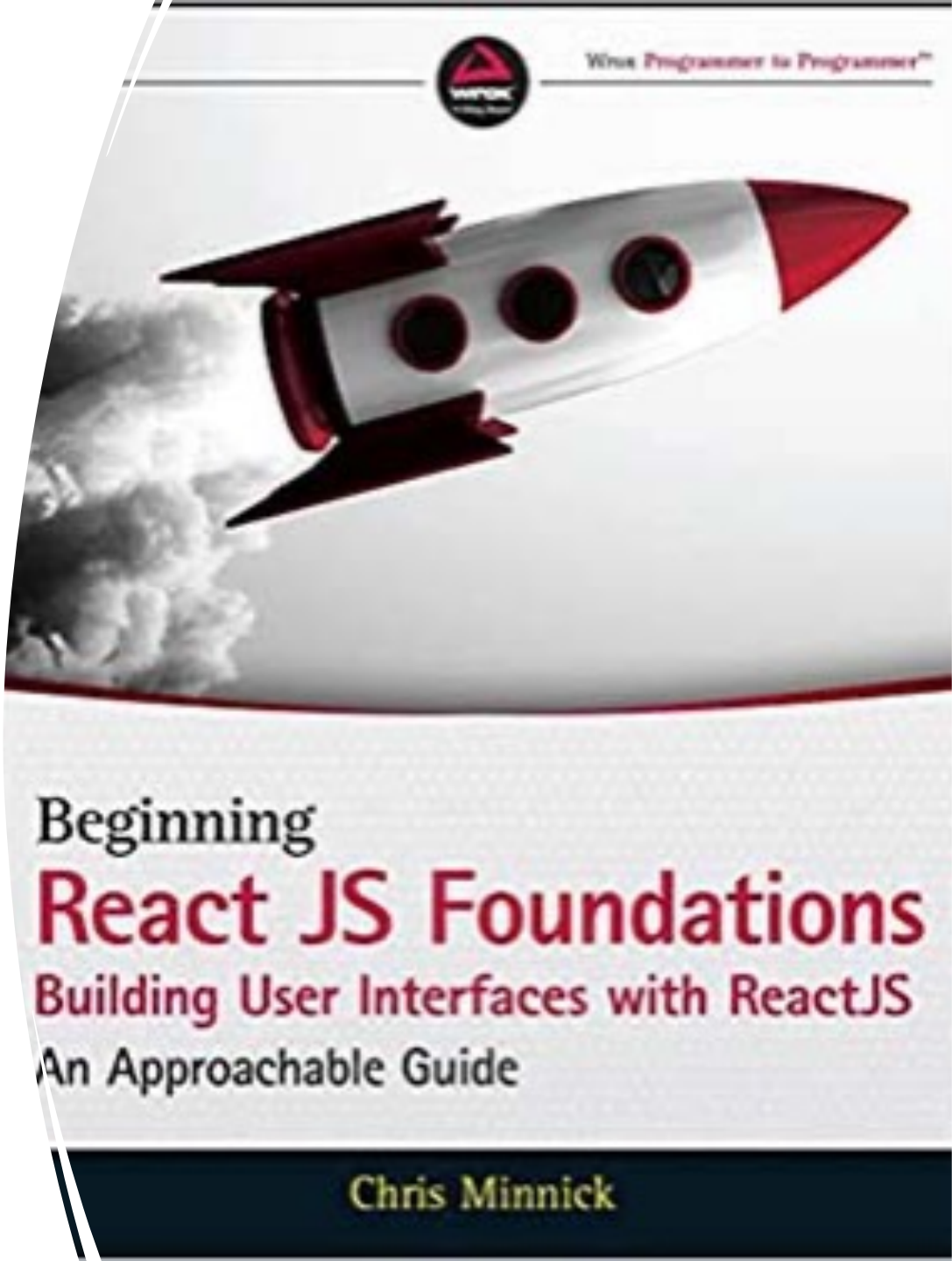
version 1.0.0, September 2023

Introduction

- Objectives
  - Who am I?
  - Who are you?
  - Daily Schedule
  - Course Schedule and Syllabus

About Me

- Chris Minnick
  - Author of 12+ books, including **React JS Foundations** (Wiley, March 2022) and JavaScript All-In-One For Dummies (Wiley, May 2023)
  - 20+ years experience in full-stack web development
  - Training & coding with React since 2015

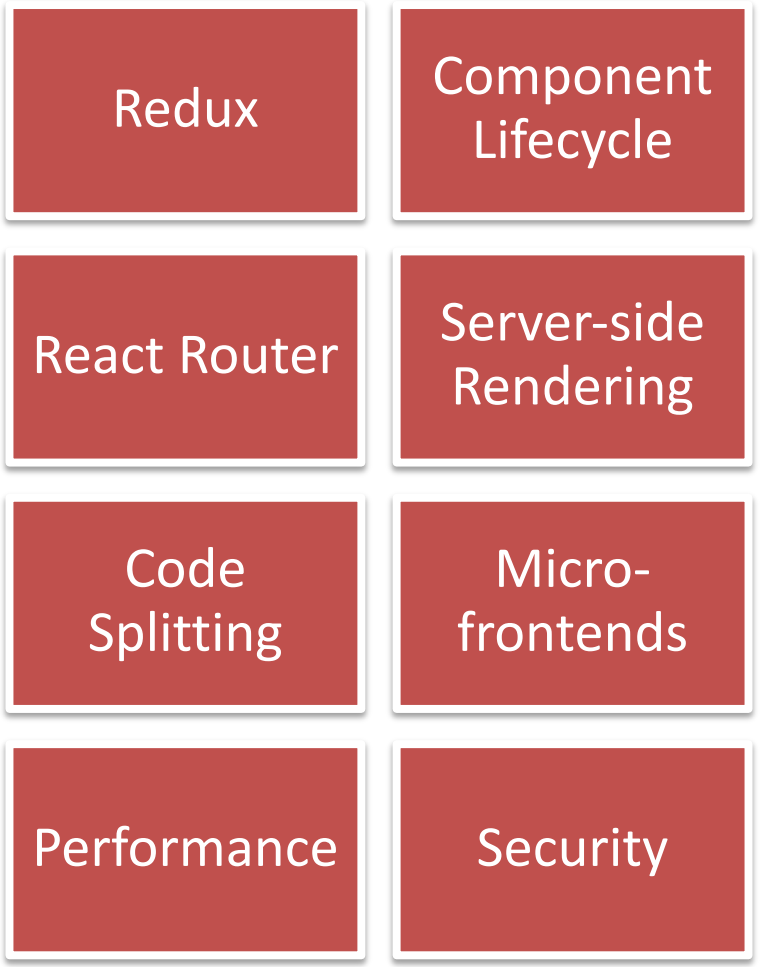


Introductions

- What's your name?
- What do you do?
- Where are you?
- JavaScript level (beginner, intermediate, advanced)?
- What do you want to know at the end of this course?
- What do you do for fun?

The Big Picture

Some of the Topics
Covered in this Course



Course Repo and More Examples

- https://github.com/chrisminnick/advanced-react
  - Here you'll find all the files you need to complete the labs, as well as the completed solutions for each lab.
- https://reactjsfoundations.com
  - This is the website for *ReactJS Foundations*, by Chris Minnick (Wiley, March, 2022). It contains all the example code from the book, and more.

Daily Schedule

- 9:00 - 10:30
- 15 minute break
- 11:45 - 12:00
- 1 hour lunch break
- 1:00 - 2:00
- 15 minute break
- 2:15 - 3:15
- 15 minute break
- 3:30 - 5:00 : labs and independent study

The Labs

- Instructions: advanced-react-labs.pdf
- Time to complete labs vary between 15 mins and 1 hour+.
- Let me know if something doesn't work.
- If you finish a lab early:
  - try lab challenges,
  - help other students,
  - do additional reading,
  - take a break.

Agenda

- 1. Day 1: React Fundamentals
- 2. Day 2: Building Robust Applications
- 3. Day 3: Scaling with Best Practices

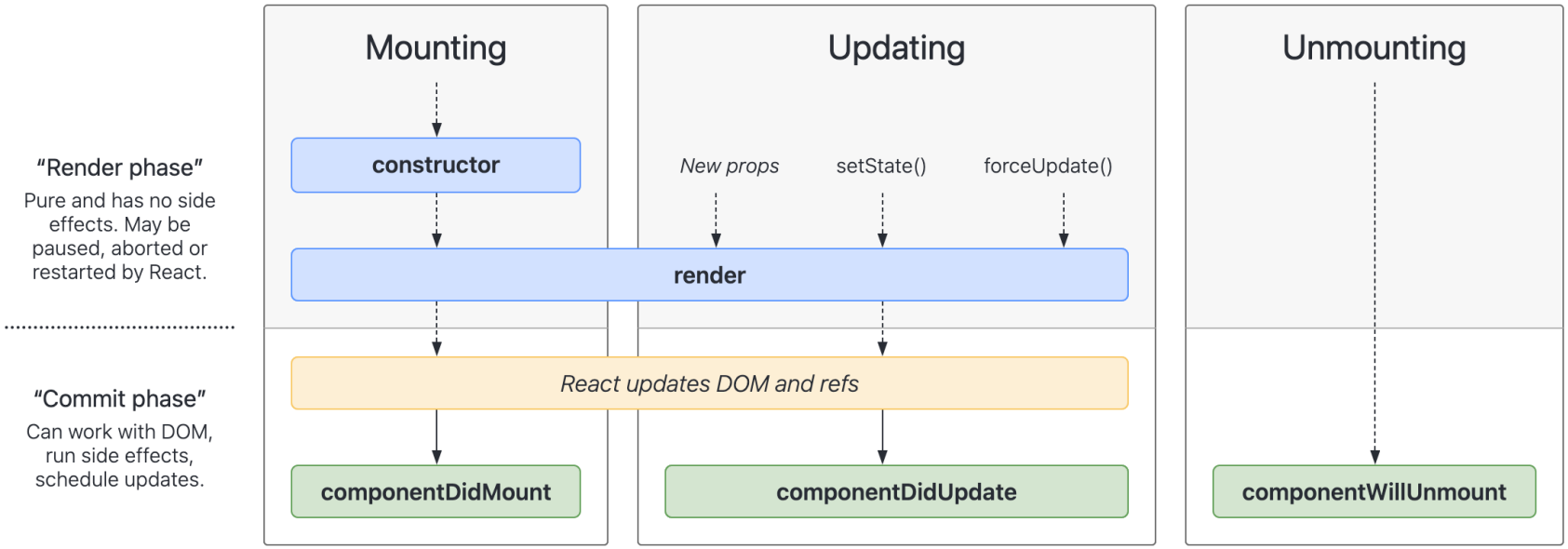
Day 1 Overview

- React Component Lifecycle
- Hooks vs. Class Components
- React Router

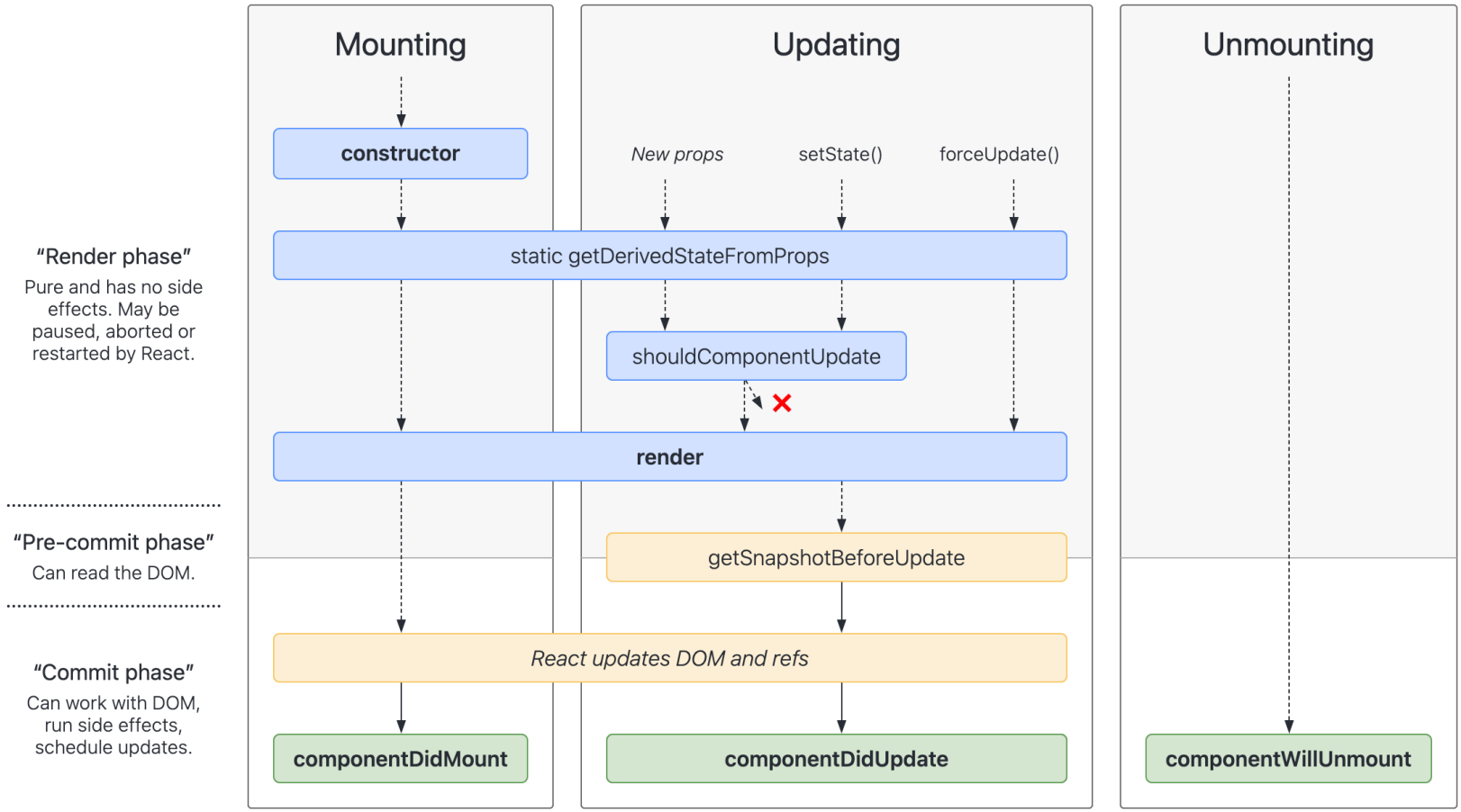
React Component Lifecycle

- Mounting Phase
- Updating Phase
- Unmounting Phase

React Lifecycle



React Lifecycle



Introduction to Component Lifecycle

- What is Component Lifecycle?
- Importance in React Development

Mounting Phase

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

Updating Phase

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Unmounting Phase

- componentWillUnmount()
- Called just before the component is removed from the DOM
- Best practices
  - Use for cleaning up side-effects initiated in other lifecycle methods (timers or event listeners).

Error Handling Phase

- static getDerivedStateFromError()
- componentDidCatch()

Lifecycle Methods in Hooks

- useEffect()
- useState()

Lifecycle Best Practices

1. **Avoid Heavy Computations in render Method**: Offload heavy calculations to lifecycle methods to avoid unnecessary re-renders.
2. **Use componentDidMount for API Calls**: Fetching data in this method ensures that the component has mounted before attempting network requests.
3. **Optimize Child Components**: Utilize React.memo or PureComponent for functional and class components, respectively.
4. **Cleanup in componentWillUnmount**: Always remove event listeners, cancel network requests, and invalidate timers before a component unmounts.

Class Components

- JavaScript classes that extend React.Component
- Can hold and manage state.
- Have access to lifecycle methods.
- Must have a render method.
- Useful for more complex components or components that require direct access to lifecycle methods.

State Management in Class Components

- setState()
  - Takes an object or a function that returns an object as its argument.
  - Merges the object with this.state
  - Is asynchronous
  - If you pass a function, the function always receives the latest state.
  - If the new state depends on the previous state, always use a function

Function Components

- Basically, the render() method from a class component.
- Cannot access lifecycle methods.
- Cannot maintain its own state.
- Access to lifecycle and state from functions is done using Hooks.

Hooks

- Functions provided by React that give functions access to internal workings of React.
- Useful for simulating component lifecycle and associating state with function components.

How to think about Hooks

- Hooks seem magical at first.
- Using them correctly requires a deeper understanding.
- Important Points:
  - Hooks store data outside of a function.
  - A Hook is associated with an “instance” of a function component

How to think about Hooks (cont.)

- Function components don’t technically have instances, and they don’t technically render.
- A function component is just a function that returns a piece of the UI when it’s invoked.
- Unless you store data outside of the function and access it when the function runs, functions have no way of persisting data between invocations.

Understanding useState

- The useState hook sets a persistent value in React and associates it with the “instance” of the function component.
- Every time the function runs, each call to useState() gets its latest value from the React library (except the first time, when the value passed to useState() initializes the value).
- useState() returns an array with 2 elements
  - the latest value
  - a function for updating the value
    - this is called the “state dispatch function”

Understanding useEffect

- The useEffect takes a function as its first argument, and an array of dependencies as its second argument.
- Each time the function component renders, useEffect compares the previous values of the dependencies with the new values.
- If the new values of any dependencies have changed, the function passed to useEffect runs.

Understanding useEffect (cont.)

- If no dependency array is passed, the function passed to useEffect will run every time the function component “instance” renders.
- If an empty array is passed, the effect will only run on the first invocation of the “instance” of the function component.
- If the effect function returns a function, that function will run when the component is “unmounted.”

Lab: React Fundamentals

- Add new features to an existing real-time chat application

React Router

- Route Configuration
- Navigation
- Protected Routes

Introduction to React Router

- What is React Router?
  - dynamic routing library built on top of the React framework. React Router uses a component-based structure to define routing rules.
- Why is it important?
  - SPA Navigation
  - Declarative routing
  - Code splitting

Router Component

- Wraps around the Routes component
  - data APIs
    - version 6.4+
    - createBrowserRouter
      - recommended for browser-based apps
    - createHashRouter
    - createMemoryRouter
    - createStaticRouter
  - pre-6.4 (still supported)
    - <BrowserRouter>
    - <MemoryRouter>
    - <HashRouter>
    - <NativeRouter>
    - <StaticRouter>

Routes Component

- Wraps around a group of Route components.

Route Component

- Matches paths to elements
- Must be a direct child of a Routes component

Link and NavLink Components

- Explanation and differences

redirect

- Redirects to a new location.

```
import { redirect } from "react-router-dom";
const loader = async () => {
  const user = await getUser();
  if (!user) {
    return redirect("/login");
  }
  return null;
};
```

Dynamic Routing

- Route parameters
  - define routes with variable parts
  - /users/:userId matches /users/123, /users/abc, etc
  - Query Strings and URL parsing
    - /search?query=react allows the component to access query

Hooks in React Router

- useHistory
  - provides access to the history object
  - push(path)
  - replace(path)
- useParams
  - const {userId} = useParams();
- useLocation
  - Provides access to the location object
  - const location = useLocation();
  - return Current path: {location.pathname} ;

Outlet Components

- Used in a parent route element to render its child route elements.

Nested Routes

- Create a component that renders an Outlet
- Nest Route elements inside the Outlet route.

```
function Dashboard() {
  return ( <Outlet /> );
}
function App() {
  return (
    <Routes>
      <Route path="/" element={<Dashboard />}>
        <Route path="inbox" element={<Inbox />} />
        <Route path="tasks" element={<Tasks />} />
      </Route>
    </Routes>
    );
}
```

Implementing Protected Routes

```
import { Navigate, Outlet } from 'react-router-dom’
const PrivateRoutes = () => {
  let auth = {'token’:false}
  return (
    auth.token ? <Outlet/> : <Navigate to='/login’/>
  )
}
function App() {
  return (
    <Routes>
      <Route element={<PrivateRoutes />}>
        <Route path=“inbox” element={<Inbox />} />
        <Route path=“tasks” element={<Tasks />} />
      </Route>
      <Route path=”/login” element={<Login />} />
    </Routes>
    );
}
```

Alternatives to React Router

- wouter
- Next JS
- React Location

Lab: Implementing Routes

- Create a signup component and a login component
- Use react router to create the following routes:
  - /login
  - /signup
  - /logout

Day 2 Overview

- Redux: A State Management Library
- Redux Toolkit

Redux: Introduction to Redux

- What is Redux?
- Why Use Redux?

Redux: Redux Architecture

- Basic Architecture
  - Store
  - Actions
  - Reducers
- Flow of Data
  - Action -> Store -> Reducer -> New State -> View Update

Redux: Actions

- Information payloads that send data from the application to the store.
- Plain JavaScript objects containing a type property to indicate the type of action being performed.
- Optionally contain additional data.

Redux: Dispatching Actions

- How to Dispatch

```
const increment = () => {
   return {
     type: 'INCREMENT',
   };
};

// Dispatching Action
dispatch(increment());
```

Dispatching Async Actions (Thunk)

```
const fetchData = () => {
  return (dispatch) => {
    fetch('https://api.example.com/data')
      .then((response) => response.json())
      .then((data) => dispatch({ type:
'FETCH_DATA_SUCCESS', payload: data }))
      .catch((error) => dispatch({ type:
'FETCH_DATA_ERROR', payload: error }));
  };
};

// Dispatching Async Action
dispatch(fetchData());
```

Redux: Reducers

- Functions that determine how the state will change in response to an action.
- Receive the current state and an action as arguments, and return a new state.
- Must remain pure functions – no side effects or mutations.

Redux: Store

- Single source of truth.
- Holds the state of the entire application.
- Created using the createStore method.

Redux: Redux Middleware

- What is Middleware?
- Common Middleware Examples

Redux: Redux Thunk

- Async Actions
- Thunk Middleware

Redux: Redux Toolkit

- What is Redux Toolkit?
  - Official toolset for efficient Redux development
- Benefits
  - Simplifies common use cases
  - Provides good defaults for store setup
  - Write “mutative” immutable update logic
  - Create “slices” automatically

Redux: createAction

- Generates action creators based on the given action type.

Basic Example:

```
const increment = createAction('INCREMENT');
```

With Payload:

```
const add = createAction('ADD');
```

Dispatch:

```
dispatch(increment());
```

or `dispatch(add(5));`

Redux: createReducer

- Simplifying Reducer Creation

```
const myReducer =
createReducer(initialState, {
   [actionType]: (state, action) => {
     // reducer logic
   }
});
```

createReducer with Payload

```
const add = createAction('ADD');


const counterReducer =
createReducer(0, {
  [add]: (state, action) => state +
action.payload,
});
```

createSlice

```
import { createSlice } from '@reduxjs/toolkit';

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

const { increment, decrement } = counterSlice.actions;

dispatch(increment());
```

Redux: createSelector

- What is createSelector?
  - creates memoized selectors
- Benefits of Using
  - helps to optimize re-rendering and re-computation by caching the result of complex calculations based on the Redux state.

```
const selector = createSelector(
  [inputSelector1, inputSelector2],
  (output1, output2) => {
    // perform computation
  }
);
```

Redux: Use with React

- react-redux
  - Official React bindings for Redux
  - <Provider>
  - useSelector
  - useDispatch

Lab: Implementing Redux

- Convert the social media app to use Redux

Day 3 Overview

- Code Splitting
- Micro-frontends
- Performance Optimization

Micro Frontends

- Micro Frontends decompose frontend applications into independent units in the same way that microservices decompose the back end.
- Gives teams the ability to independently develop, test, and deploy each micro frontend.

Introduction to Micro-frontends

- Why consider using them?
  - Deploy microfrontends independently
  - Use multiple frameworks on the same page
  - Lazy load code for improved initial load

React Micro Frontends

- Challenges:
  - Overhead: Each micro frontend needs to access the same libraries, CSS, routes
    - Can create massive problem of duplication of code and poor performance
  - Cross-cutting concerns: Each frontend needs to share authentication, for example.
  - Communication between micro frontends is more complex than between components in an app.

Common strategies for React Micro frontends

- Use a container application to handle rendering and routing between the micro frontends.
- Use a CDN to host shared frontend libraries and resources.

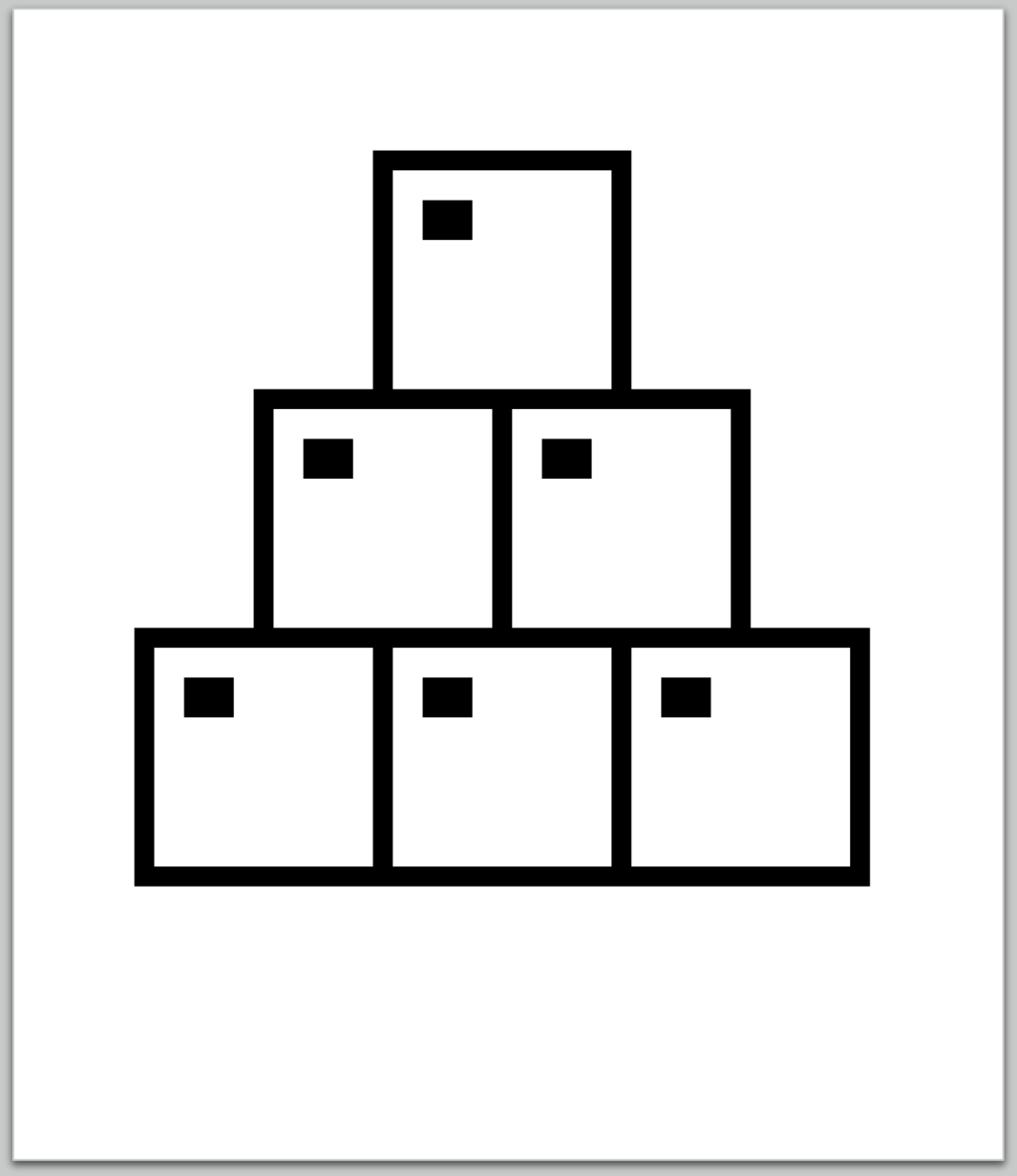
Shared State in React Micro Frontends

- Four strategies
  - Web workers
  - Props and callbacks
  - Custom events
  - Pub Sub library

Tools for Implementing Micro-frontends

- iFrames
- Web Components
- Server-Side Includes
- Single SPA (single-spa.js.org)
- Module Federation (Webpack 5)
- Tailor.js
- Piral

Lab: Creating Micro Frontends



Performance Optimization

Measuring Performance

- Lighthouse
- React DevTools Profiler
- reportWebVitals();

How Code Splitting Works

- Code Splitting is a feature supported by bundlers like Webpack and Browserify which can create multiple bundles that can be dynamically loaded at runtime. It helps in reducing the initial load time by splitting the application into smaller chunks which are loaded as they are needed.

Dynamic Imports

- The import() syntax is a function-like expression that allows loading an ECMAScript module asynchronously and dynamically.
- `import ('./SomeComponent')`
- Import declarations (import SomeComponent from './SomeComponent) is static. The imported module is evaluated at load time.

React.lazy() and Suspense

- React.lazy() lets you render a dynamic import as a regular component.
- `const SomeComponent = lazy(() => import('./SomeComponent'));`
- The component won't be loaded until it's rendered the first time.
- Wrap a lazy-loaded component with <Suspense> to specify what should display while the component is loading.

Lazy & Suspense Example

```
const SomeComponent = React.lazy(() =>
import('./SomeComponent'));

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

Route-based Code Splitting

- Delay loading components for Routes until those Routes are rendered.

```
function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route exact path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </Suspense>
    </Router>
  );
}
```

Performance Optimization in Class Components

- **shouldComponentUpdate().** Returning false from this lifecycle method will skip render().
- **PureComponent**. Extend (instead of React.Component) to indicate that a component's output is wholly dependent on its props (it's "pure"). Render will be skipped if props are the same.

Performance Optimization in Functions

- useMemo memorizes values. Use it to memorize the output of expensive calculations.
- useCallback memorizes functions. Use it to avoid unnecessary renders.
- React.memo: pass function components to React.memo() to cause them to skip re-rendering if the props passed are unchanged.

Batching Updates

- Avoid extra renders by taking advantage of state's asynchronous nature.

```
const handleUpdate = () => {
  setState(prevState => ({
      …prevState, count1: prevState.count1 + 1
  }));
  setState(prevState => ({
      …prevState, count2: prevState.count2 + 1
  }));
}
```

- Starting with React 18, React automatically batches updates.

Lazy Initialization

- Avoids recalculating or refetching initial state on every render.
- Instead of:
  - const initialState = expensiveCalculation(props)
  - const [state, setState] = useState(initialState);
- Do this:
  - const getInitialState = () => window.localStorage.getItem('state');
  - const [state, setState] = useState(getInitialState);

Data Fetching Libraries / Hooks

- SWR
- React Query

Web Workers

- Off-the-main-thread calculations

Use Throttling to Limit Event Handlers

- Throttling is used to create a throttled function that can only call the func parameter once per every wait milliseconds.
- import React, { useEffect } from 'react';
- import { throttle } from 'lodash';

- https://codesandbox.io/s/throttling-dr695r?file=/src/ThrottlingExample.js

Debouncing

- The Lodash debounce function returns a debounced function that will execute a function after X milliseconds pass since its last execution.
- Use for limiting API requests.
- https://codesandbox.io/s/throttling-dr695r?file=/src/DebouncingExample.js

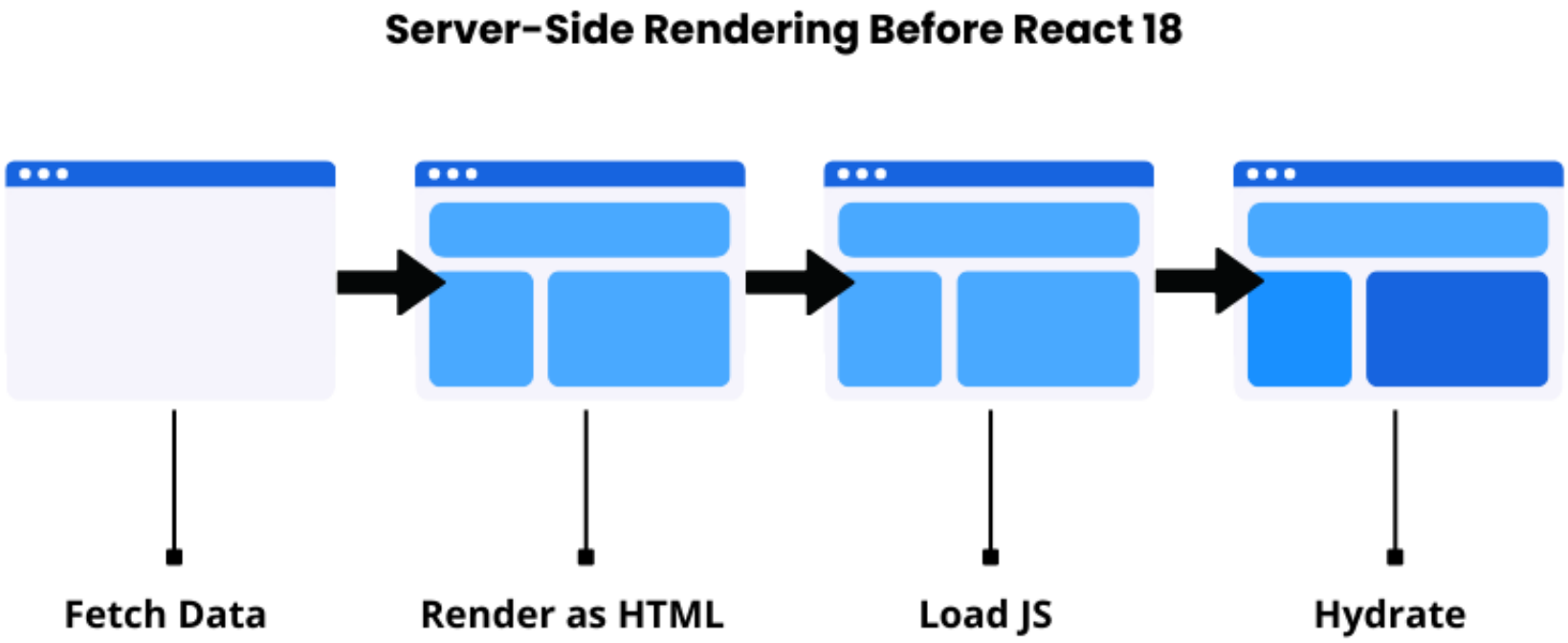
Introduction to Server-Side Rendering

- What is SSR?
- Why is it important?

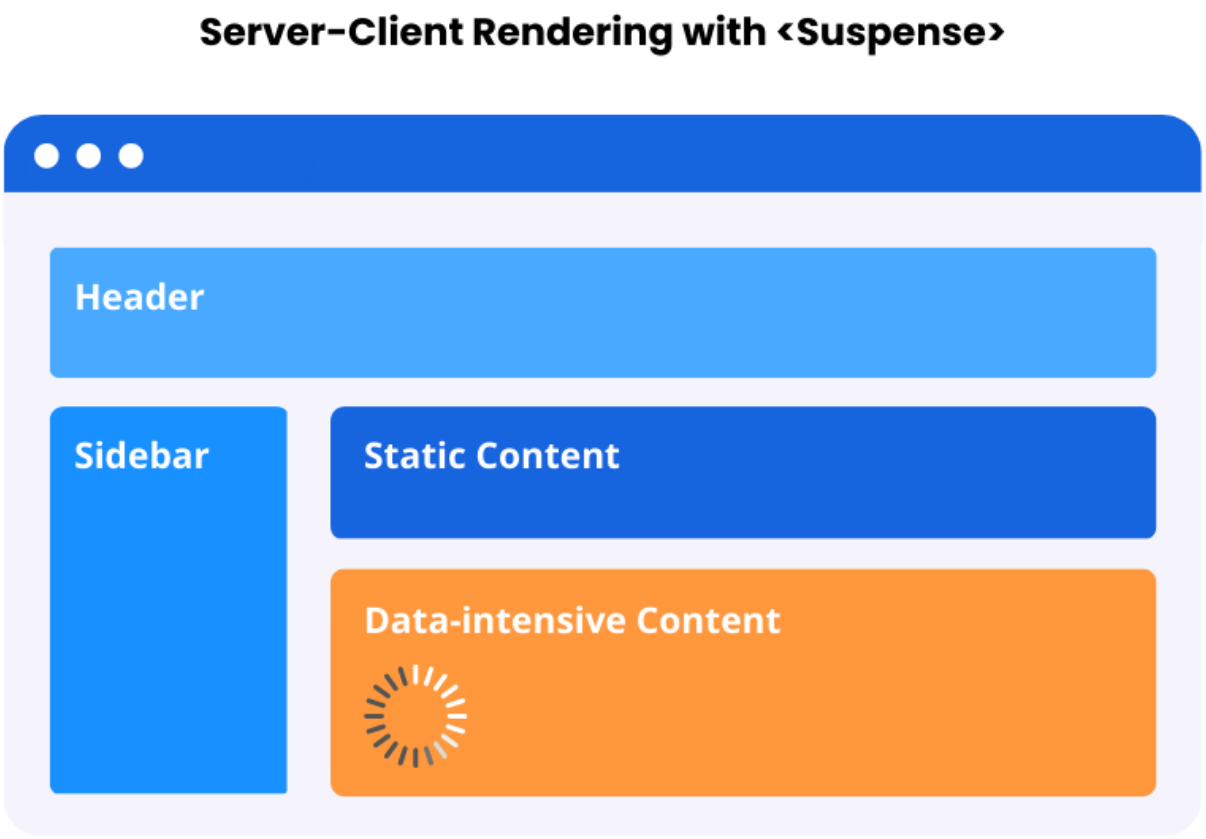
Client-Side Rendering vs Server-Side Rendering

- Key differences
- Pros and Cons

How SSR Works (< React 18)



How SSR works in React 18



Tools for SSR in React

- Next.js
- Gatsby

Advantages of SSR

- SEO benefits
- Faster initial page loads

Drawbacks of SSR

- Server load
- Complexity

Hydration

- What is Hydration?
  - Using client-side JavaScript to add application state and interactivity to server-rendered HTML
- How it works

```
import { hydrateRoot } from 'react-dom/client';

hydrateRoot(document.getElementById('root'), <App />);
```

- https://codesandbox.io/s/hydrate-qyy2mr?file=/public/index.html

Lab: Performance Optimizations

SECURITY BEST PRACTICES

Use the Latest Version of React

- Always make sure you're using the latest version of React and its associated libraries. This ensures that you benefit from the latest security patches and updates.

Escape User Input

- React automatically escapes content rendered with JSX to prevent Cross-Site Scripting (XSS) attacks. However, when you use methods like dangerouslySetInnerHTML, you're bypassing this protection. Use this method sparingly, and always sanitize any user input or content that might be rendered.

Avoid Inline Event Handlers with User Data

- Do not inject user data directly into inline event handlers in JSX. It could create potential security risks if not properly sanitized.

Beware of Third-party Components

- Not all npm packages or third-party React components follow best security practices. Always audit and review the code of third-party components before including them in your project.

Use HTTPS

- Serve your React application over HTTPS to ensure that data exchanged between the client and the server remains encrypted and secure.

Manage Secrets Properly

- Never hard-code sensitive information or secrets in your React code. Use environment variables or server-side configurations for this.

Validate and Sanitize Prop Data

- If you're passing data to components via props, ensure the data is validated and sanitized, especially if the data originates from user input or external sources.

Handle Redirections Carefully

- If your application includes redirections based on user input, validate the input to ensure that it doesn't lead to unintended or malicious URLs.

Use Tokens for Authentication

- When implementing authentication, consider using JWT (JSON Web Tokens) or similar token-based methods, and store them securely. Avoid storing them in local storage due to potential XSS attacks; instead, use HttpOnly cookies.

Keep Dependencies Updated

- Regularly update all your project's dependencies. This helps patch vulnerabilities that might exist in older versions of libraries you're using.

Regularly Audit Dependencies

- Use tools like npm audit or yarn audit to identify and fix vulnerabilities in your project's dependencies.

Implement Rate Limiting and Throttling

- If your React app communicates with an API or server, implement rate limiting to prevent abuse.

Use Strong CORS Policies

- If your React app communicates with an API, ensure that Cross-Origin Resource Sharing (CORS) policies are correctly set up to limit which domains can access your resources.

Q&A

Thank You

- chris@minnick.com