

Online Multimedia

Winter Semester 2019/20

Tutorial 06 – Front-end Tooling



Today's Agenda

- Testing
- Debugging & Benchmarking
- Roundup Quiz



Understanding JSX/TSX

- JSX/TSX is just a syntax sugar to eliminate cumbersome JS calls
- React transpiles this syntax to JS calls

```
export default class OmmCounter extends Component {  
  ...  
  render() {  
    return  
    React.createElement('div', null,  
      React.createElement('span', null, this.state.count),  
      React.createElement('div', null,  
        React.createElement('button', {onClick: this.inc}, '+'),  
        React.createElement('button', {onClick: this.dec}, '-'))  
  }  
}
```

omm-counter.tsx

```
export default class OmmCounter extends Component {  
  ...  
  render() {  
    return (  
      <div>  
        <span>{ this.state.count }</span>  
        <div>  
          <button onClick={this.inc}>+</button>  
          <button onClick={this.dec}>-</button>  
        </div>  
      </div>  
    )  
  }  
}
```

omm-counter.tsx

Functional Components

- Class components ("stateful") was the only way to create an component
- However, building UI is very rare in designing a class system
- Functional components ("stateless") simply use JS functions to create an component:

```
const c: React.FC = () => {  
  return (<div>TSX</div>)  
}
```

```
import React from 'react';  
import './App.css';  
import OmmCounter from './components/omm-counter/omm-counter';  
const App: React.FC = () => {  
  return (  
    <div className="app">  
      <div className="header"><h1>Counter - React  
Hooks</h1></div>  
      <OmmCounter />  
    </div>  
  );  
}  
export default App;  
App.tsx
```

State Hook

- In functional components, there are no state, to manage states, we can involve useState hook:

```
const [currentState, setCurrentState] = useState('any data')  
const F = () => { setCurrentState('new state') }
```

- useState() returns an array with exactly two elements:
 - **Getter**: Your current state value
 - **Setter**: A method to update your state value

Props

- props works like function parameters, which allows you pass data from a parent (wrapping) component to a child (embedded) component.
- Changes in props trigger React to re-render the components and potentially update the DOM in the browser.

```
import React, { Component } from 'react';
import Child from './child'
export default class Parent extends Component {
  render() {
    return (
      <div><Child name={"Max"} age={18}/></div>
    )
  }
}
```

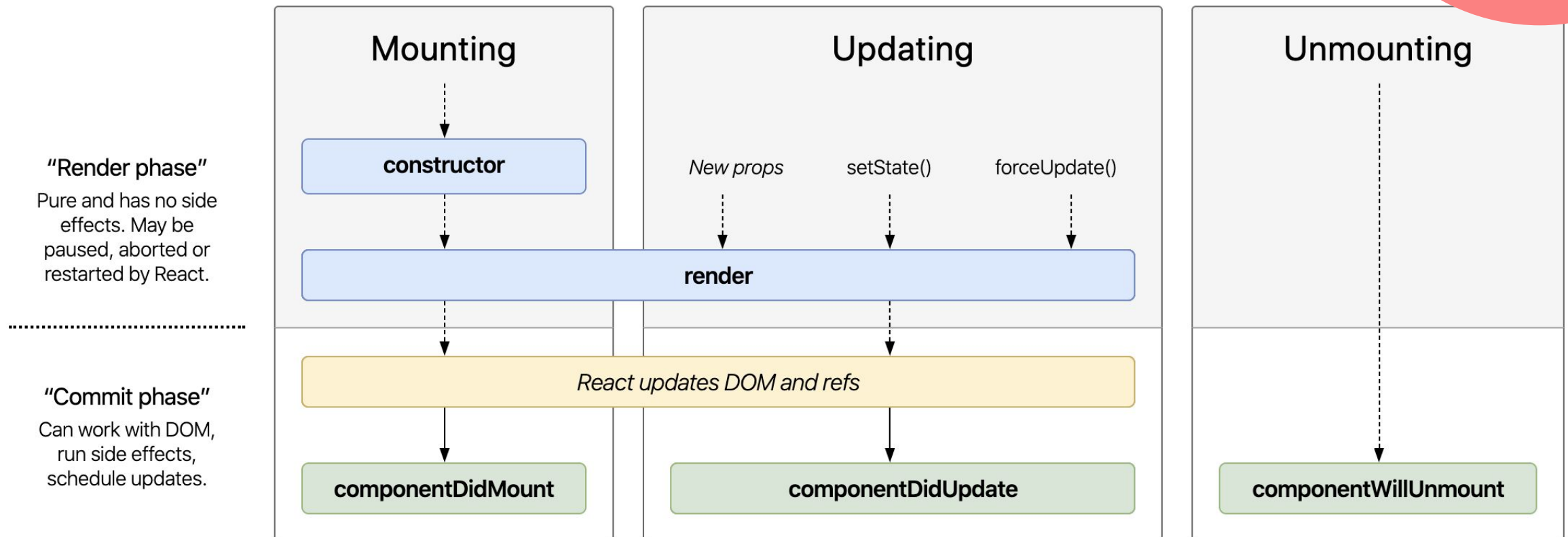
parent.tsx

```
import React, { Component } from 'react';
interface ChildProps { name: string, age: number }
export default class Child extends Component<ChildProps> {
  render() {
    return (<div>
      <div>name: {this.props.name}</div>
      <div>age: {this.props.age}</div>
      /* <div>role: {this.props.role} </div> */
    </div>)
  }
}
```

child.tsx

React Lifecycle Method Diagram

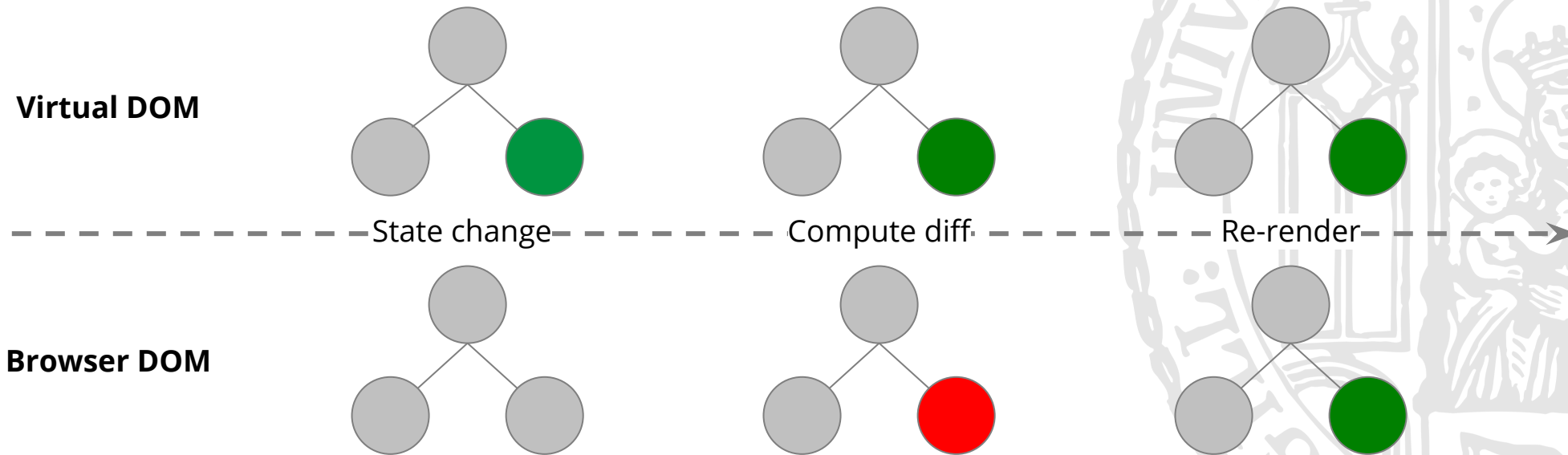
(only in class-based components)



<http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

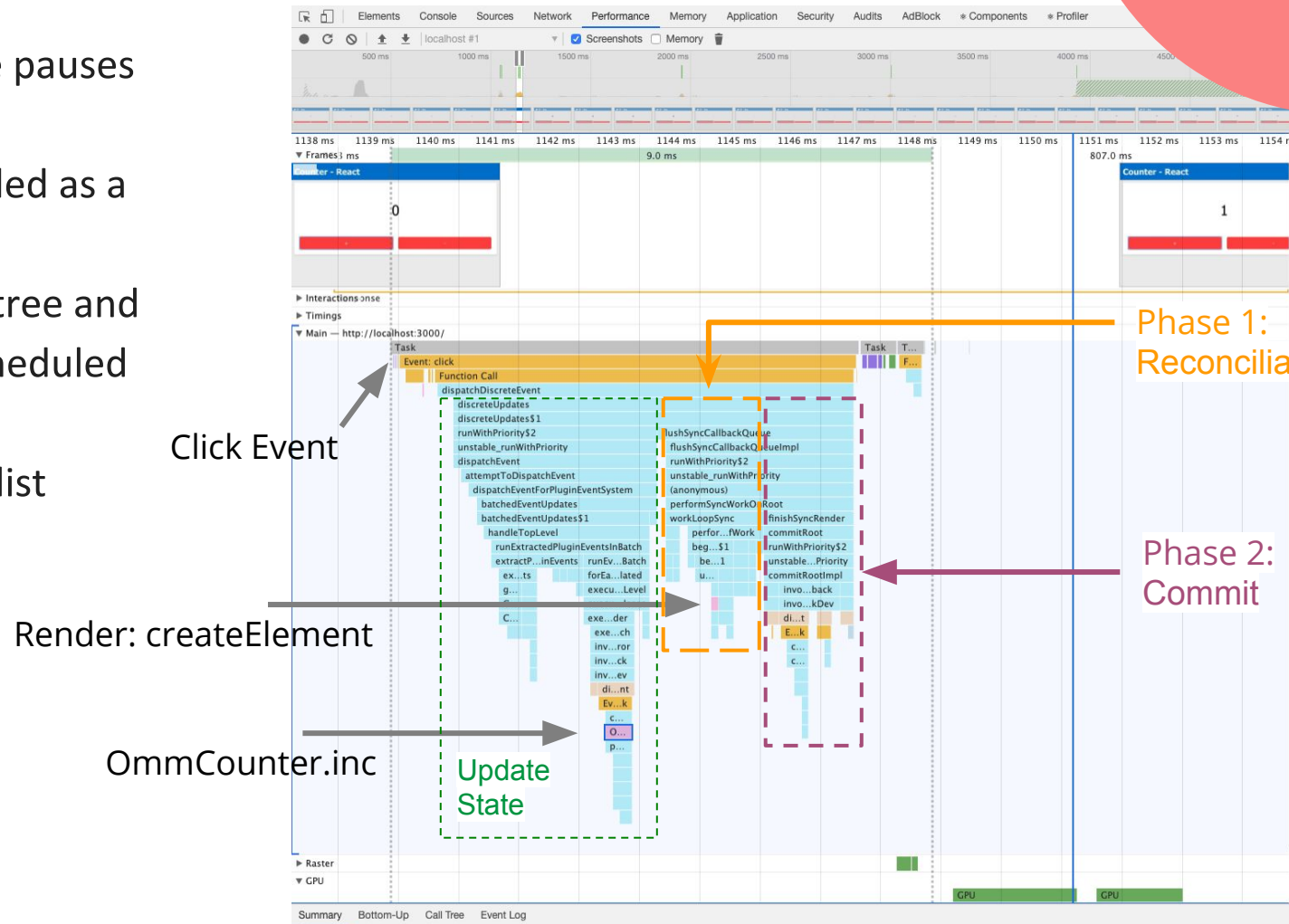
Reconciliation

- React keeps a "virtual" representation (*Virtual DOM*) of a UI in memory and synced with the "real" DOM by ReactDOM. The process of synchronization is called reconciliation.
- When the component state is updated, the ReactDOM.render will be recalled and compares the "real" DOM to patch their diff.



Fiber Engine: Two-Phase Update Processes

- Traverse update a large DOM tree pauses main thread (recall event loop)
- Hence, an update patch is scheduled as a fiber node
- **Reconciliation phase** builds fiber tree and ensures higher priority tasks is scheduled after a timeslice (interruptible)
- **Commit phase** commits all effect list (uninterruptible)



Conditional and Repetitive Rendering

- Conditional:

```
if (condition) {  
  optionalRender = (<div>render under a condition</div>)  
}  
return (<div>{optionalRender}</div>)
```

- Repetitive:

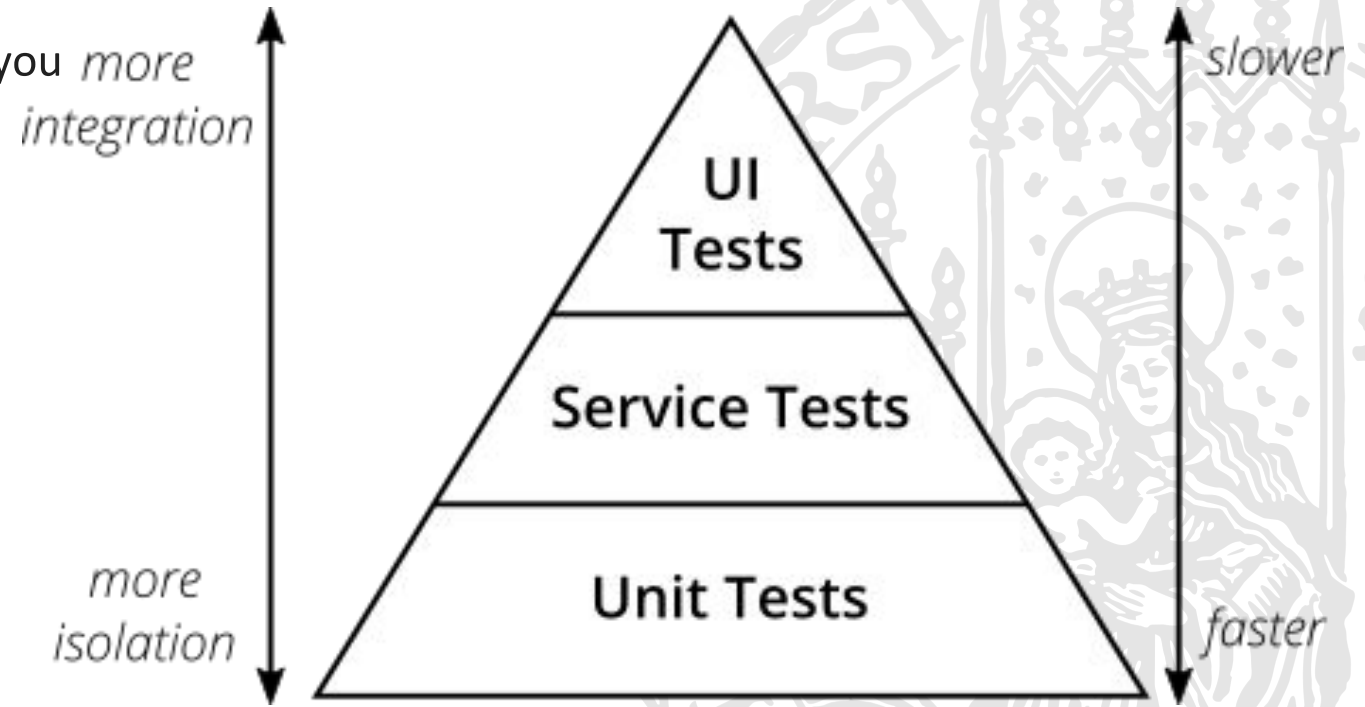
```
return (<div>{  
  this.state.stateArray.map(item => {  
    return <div key={item.id}>{item.property}</div>  
  })  
}</div>)
```

Testing in React



Test Pyramid

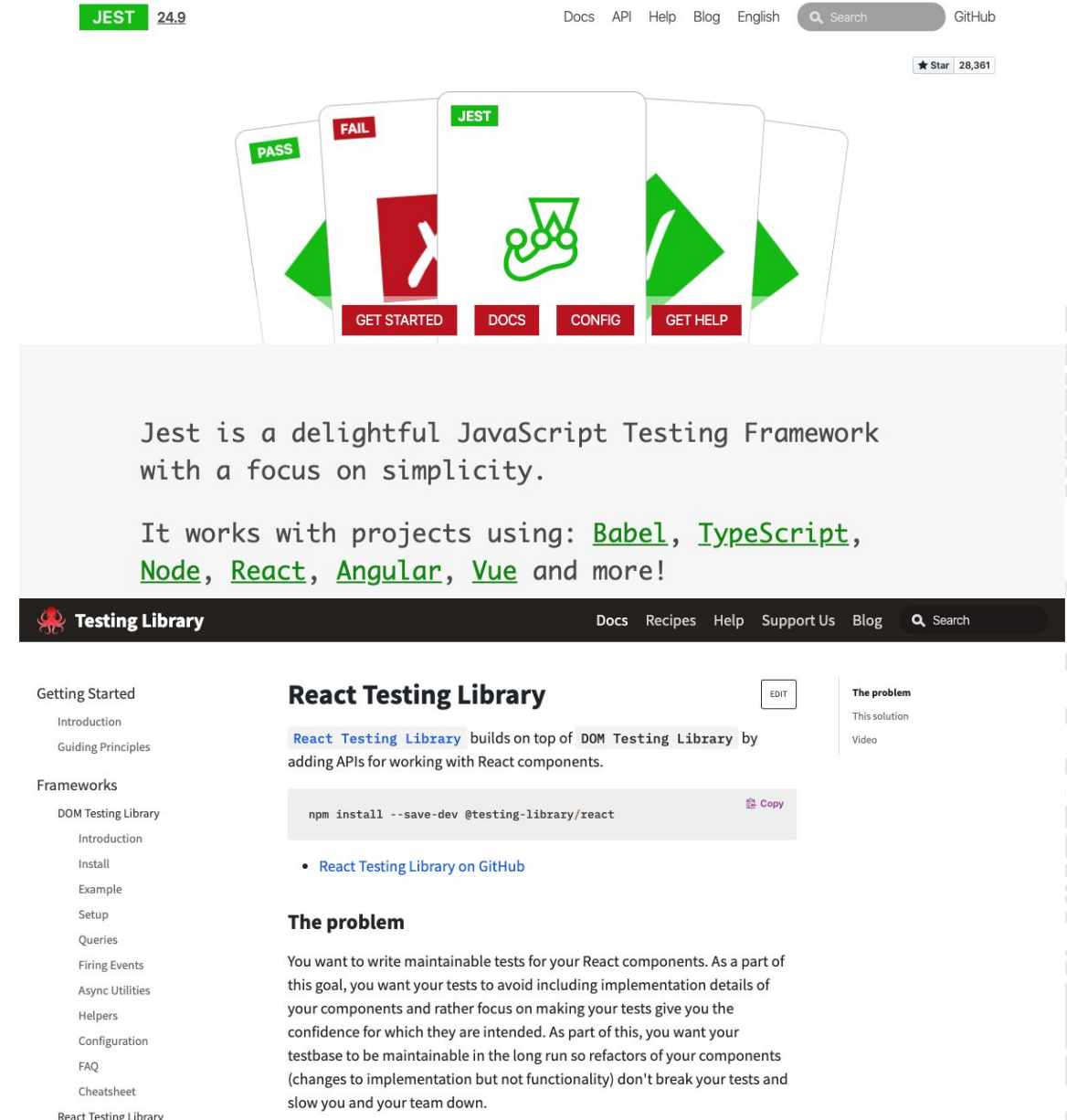
- Tests save yourself time maintaining application, but hard to
 - find a good amount of tests
 - avoid testing implementation details
- You should rarely have to change tests when you *more refactor code*.
- UI Test
 - Usually performs End-to-End test
 - Test all parts of application
- Service Test (Integration Test)
 - Tests connectivity
 - E.g. web services - APIs
- Unit Test
 - Tests are local
 - Check if code works



<https://martinfowler.com/articles/practical-test-pyramid.html>

Test Utilities in React

- **Jest** and **React Testing Library** are the two official recommended testing library for React
- **Jest** is a general purpose JavaScript testing library, e.g. `expect(val1).toBe(val2)`
- **React Testing Library** builds on top of DOM Testing Library, offers various rendering utilities for React components.
- APIs
 - <https://jestjs.io/docs/en/api>
 - <https://testing-library.com/docs/react-testing-library/api>



The screenshot shows the Jest website header with navigation links for Docs, API, Help, Blog, English, and a search bar. A GitHub star count of 28,361 is visible. Below the header is a graphic with cards for PASS, FAIL, and JEST, and buttons for GET STARTED, DOCS, CONFIG, and GET HELP. The main content area describes Jest as a delightful JavaScript Testing Framework with a focus on simplicity, and lists supported projects: Babel, TypeScript, Node, React, Angular, and Vue. The footer includes the React Testing Library logo and navigation links for Docs, Recipes, Help, Support Us, and Blog.

Jest is a delightful JavaScript Testing Framework with a focus on simplicity.

It works with projects using: [Babel](#), [TypeScript](#), [Node](#), [React](#), [Angular](#), [Vue](#) and more!

React Testing Library

[React Testing Library](#) builds on top of [DOM Testing Library](#) by adding APIs for working with React components.

```
npm install --save-dev @testing-library/react
```

• [React Testing Library on GitHub](#)

The problem

You want to write maintainable tests for your React components. As a part of this goal, you want your tests to avoid including implementation details of your components and rather focus on making your tests give you the confidence for which they are intended. As part of this, you want your testbase to be maintainable in the long run so refactors of your components (changes to implementation but not functionality) don't break your tests and slow you and your team down.

Test at Glance

- Create React App automatically gives a sample test in `src/App.test.tsx`
- Create React App provides the option to let you run your test: `npm test`



```
ou-lmu □ dev ● 1 zsh (tmux)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.126s, estimated 2s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

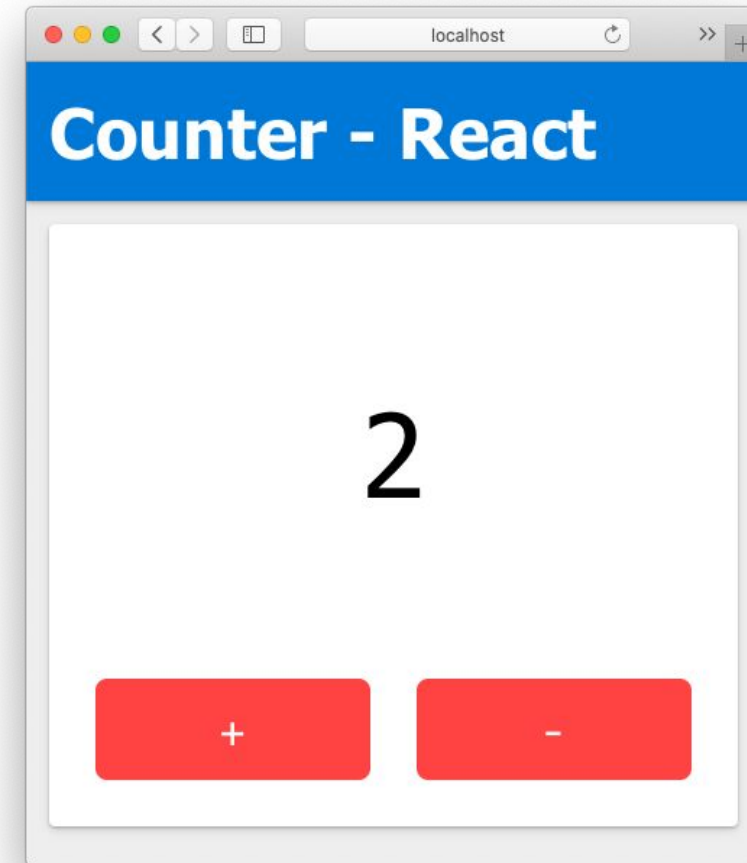
App.test.tsx

```
{
  ...
  "scripts": {
    ...
    "test": "react-scripts test",
    ...
  },
  ...
}
```

package.json

What to Test?

- Take the OmmCounter as an example
- Questions:
 - what are the most important features in OmmCounter?
 - what is a testable user flow?
- **Component renders without crash**
- **"+" button clicked → label = old label + 1**
- **"-" button clicked → label = old label - 1**



What to Test?

- Component renders without crash
- "+" button clicked → label = old label + 1
- "-" button clicked → label = old label - 1



```
import React from 'react';
import OmmCounter from './omm-counter';
import { render, fireEvent } from '@testing-library/react';

it('renders without crashing', async () => {
  const { getByText } = render(<OmmCounter />);
  const text = document.querySelector('.counter-state');
  expect(text!.innerHTML).toBe('0');
});

it('plus button works', () => {
  const { getByText } = render(<OmmCounter />);
  fireEvent.click(getByText('+'));
  const text = document.querySelector('.counter-state');
  expect(text).toBeDefined();
  expect(text!.innerHTML).toBe('1');
});

it('minus button works', () => {
  ... // similar to above
});
```

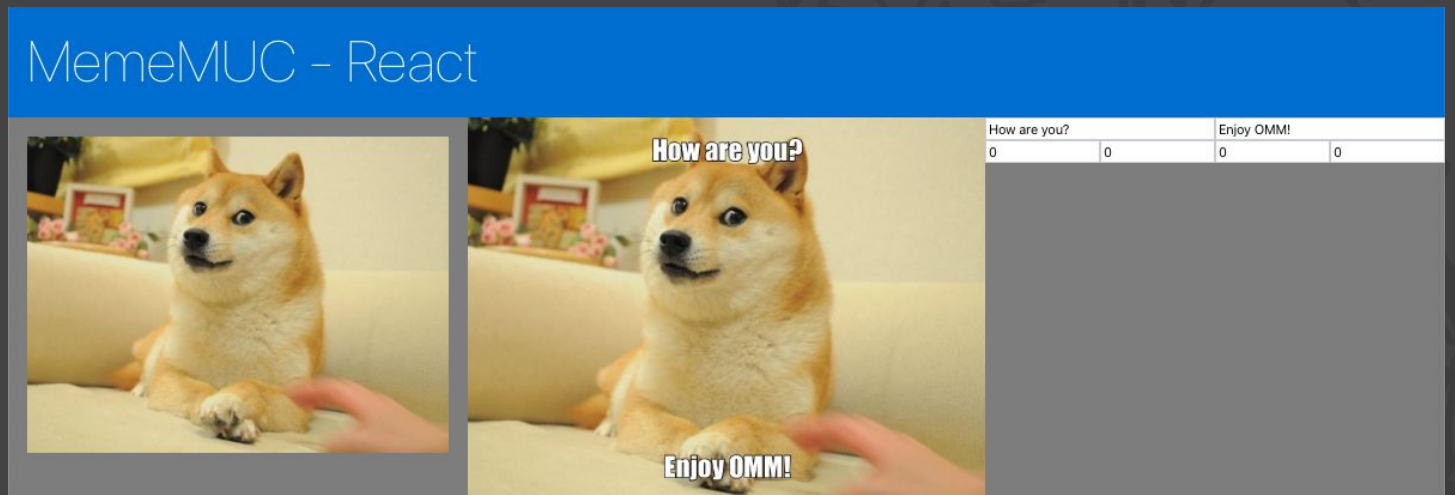
omm-counter.test.tsx

Breakout #1: Testing MemeMUC Component

Write tests to test the following aspects:

1. MemeMUC component renders without errors
2. The middle image shows correctly if user clicks images from the left
3. The middle image is rendered correctly if user inputs upper text or lower text

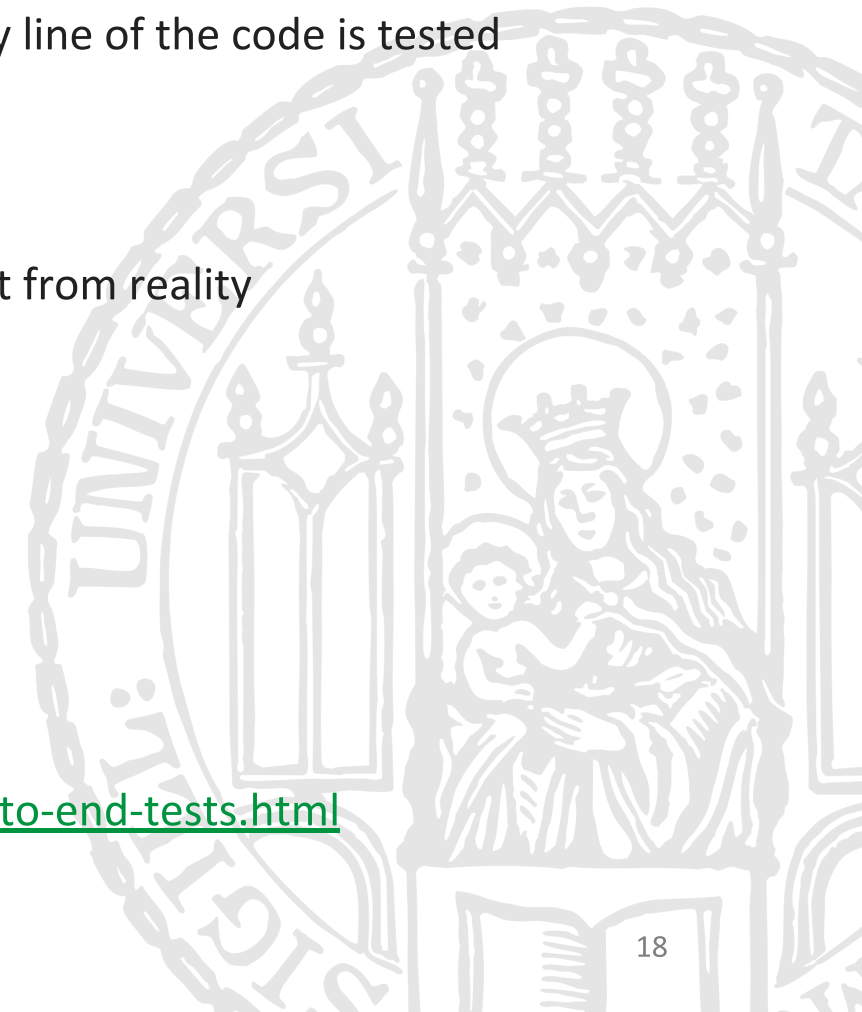
Timeframe: **20 Minutes**



Testing is Hard

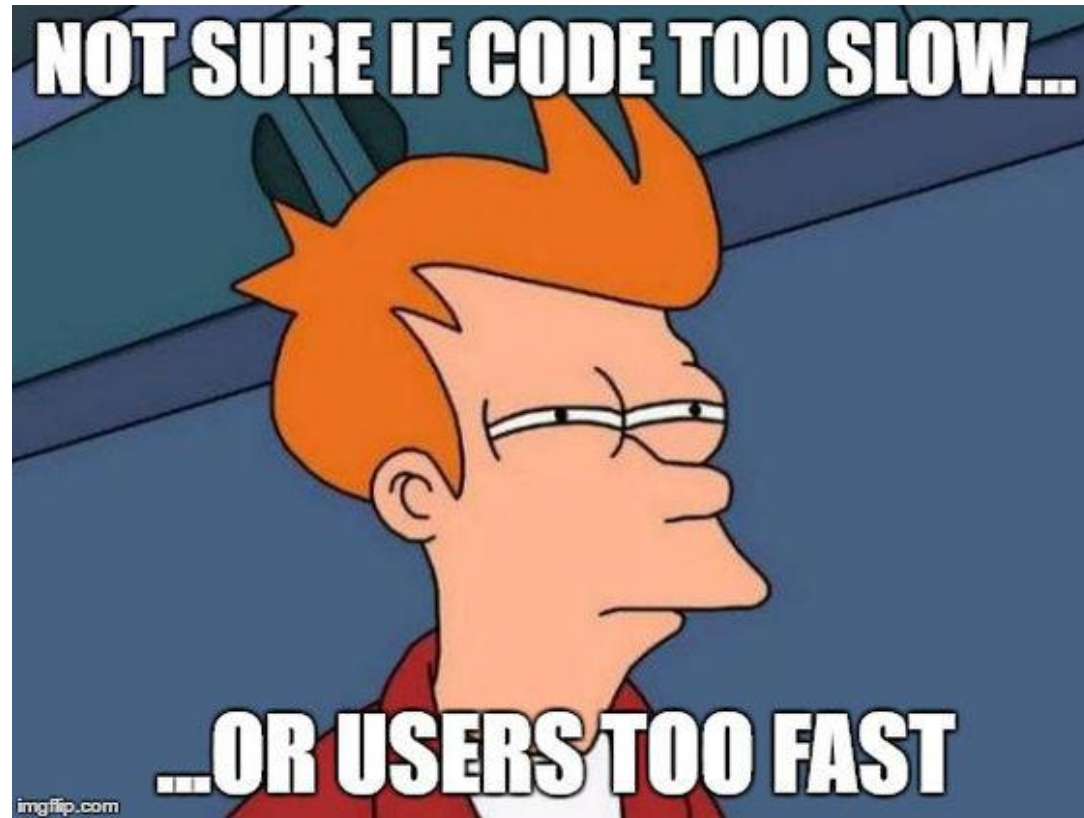
- Testing is hard and difficult because we need to test the code for both valid and **invalid inputs**
- Testing always need to give the inputs in such a way that each and every line of the code is tested efficiently, i.e. 100% coverage takes huge effort
- Non-deterministic code behavior is not deterministically triggered
- Some tests can only be tested in a simulated environment which is apart from reality

- Didn't cover:
 - (Expensive) End-to-end Testing
 - Just Say No to More End-to-End Tests
<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>



Debugging & Benchmarking

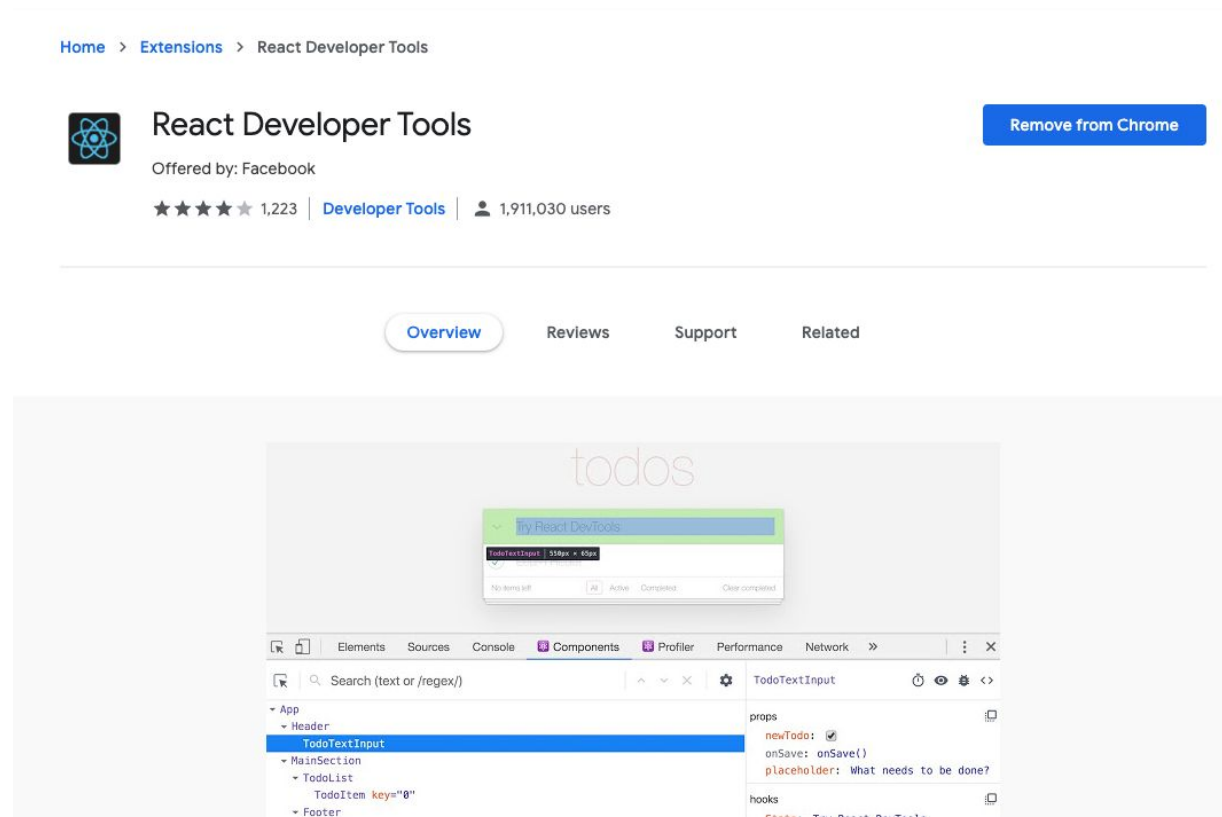




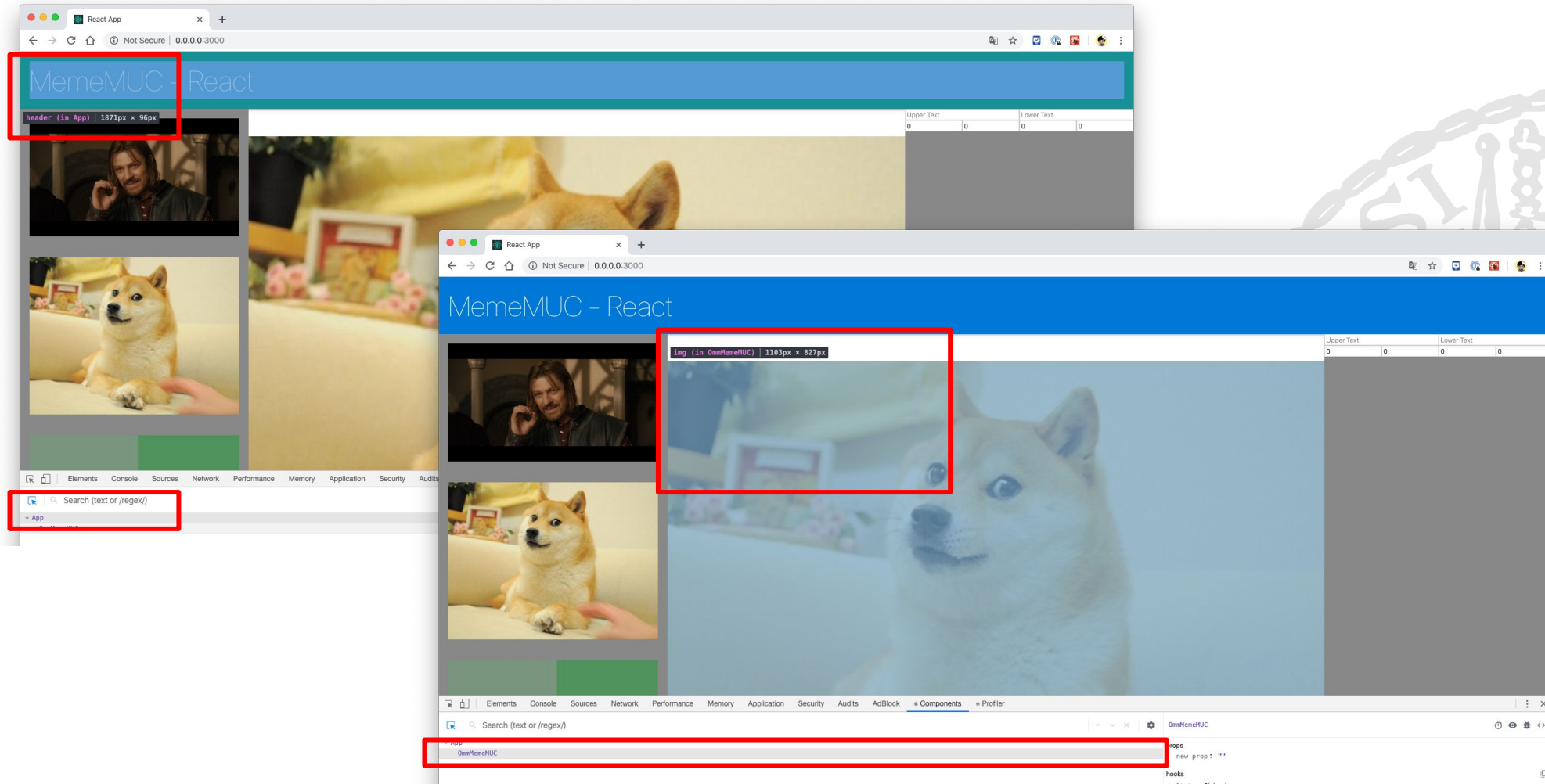


React DevTools

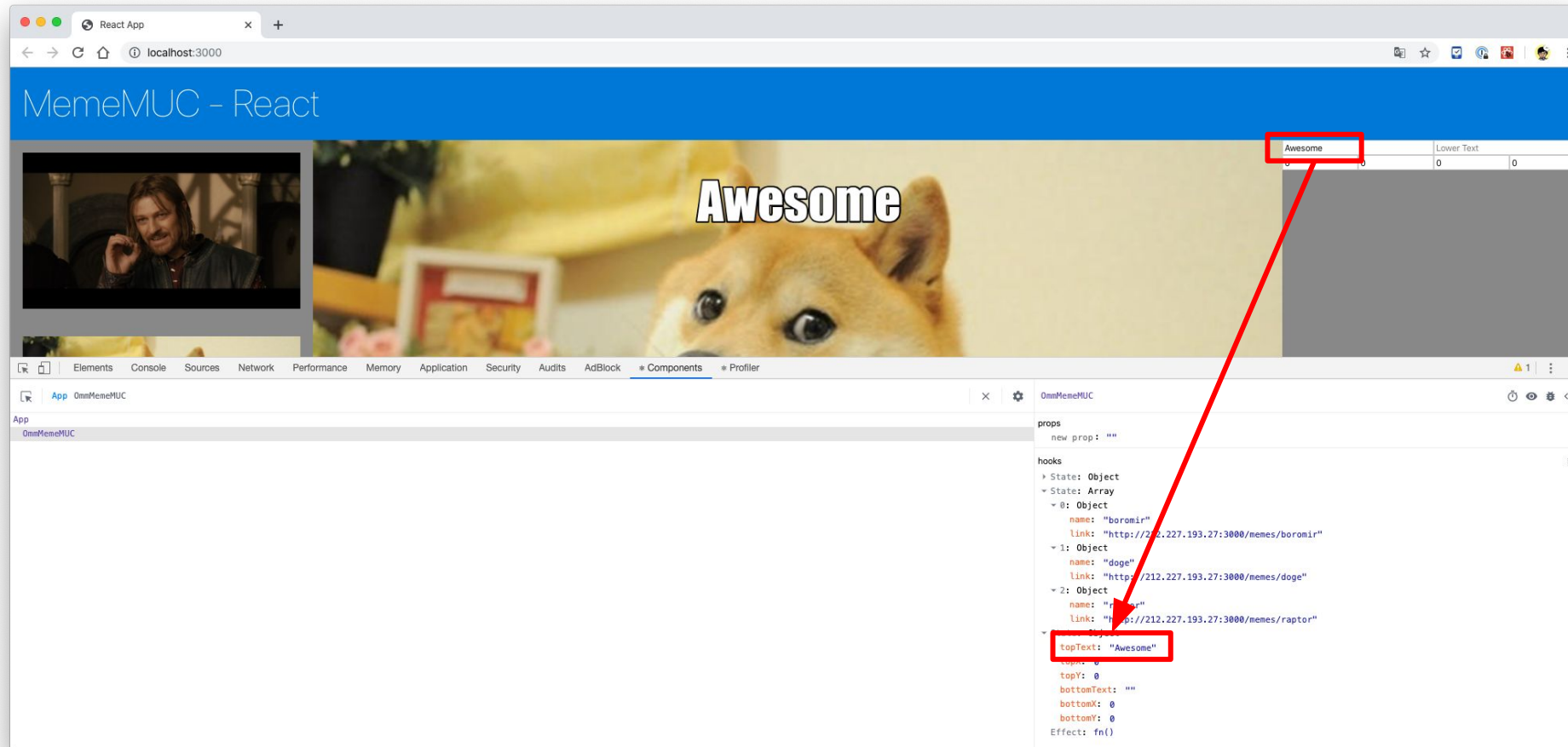
- Allow you debugging and benchmarking React apps per-component
- Interactive Tutorial: <https://react-devtools-tutorial.now.sh/>



React DevTools: Components

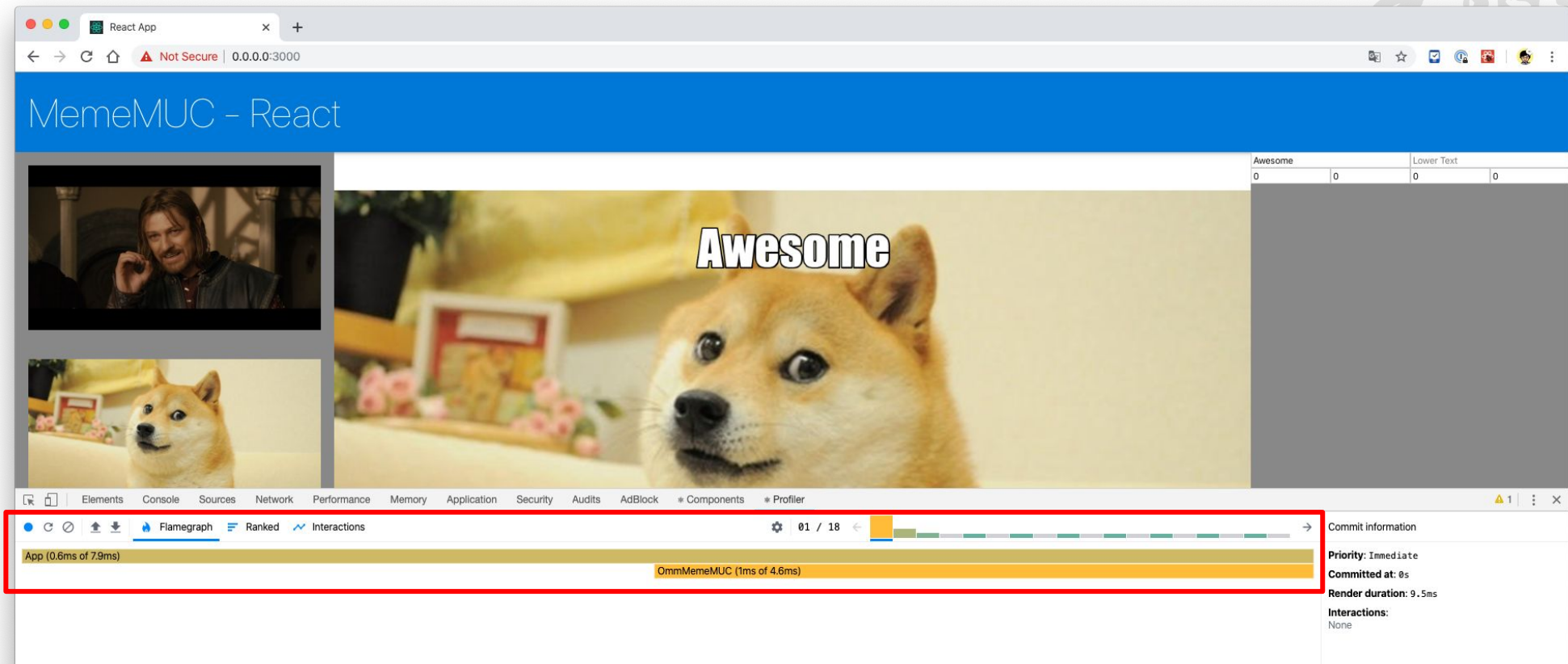


React DevTools: Components



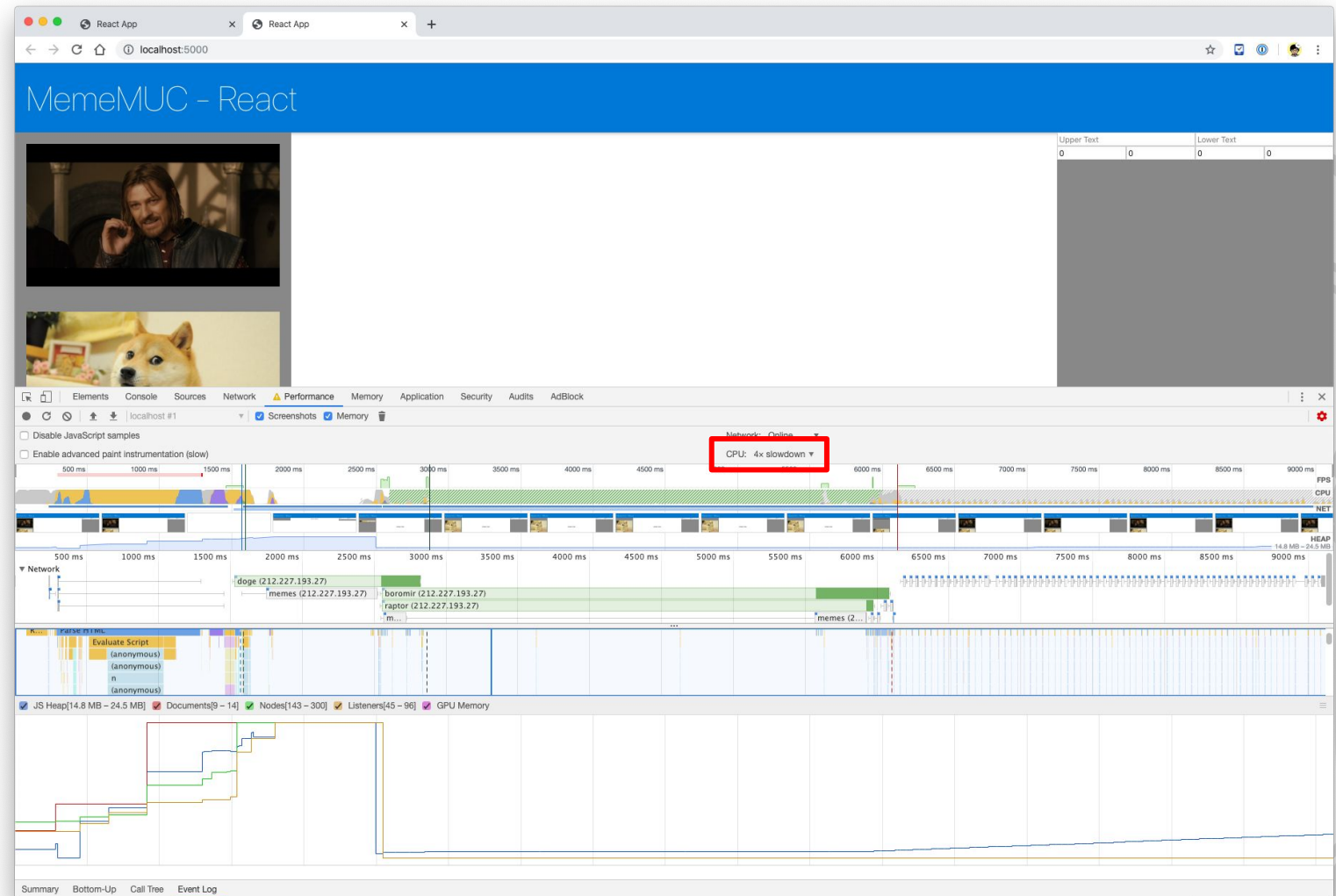
React DevTools: Profiler

- The App component is rendered in 7.9ms
- OmmMemeMUC consumes 4.6ms (58% of total rendering time)



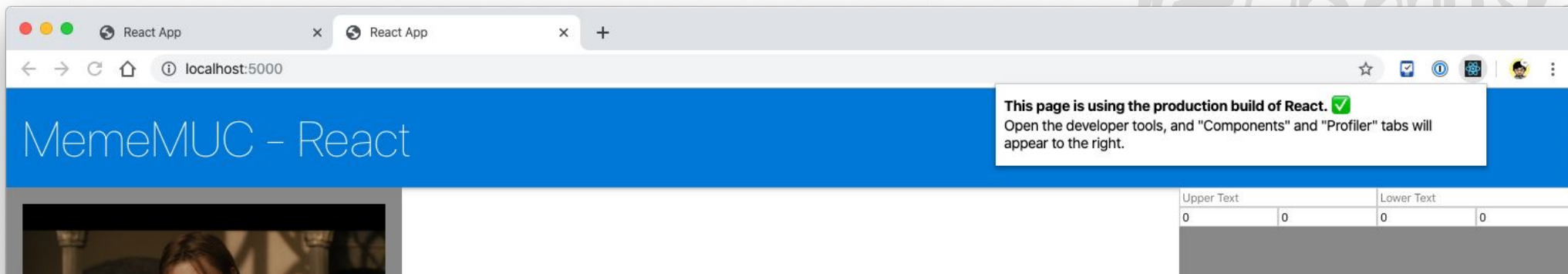
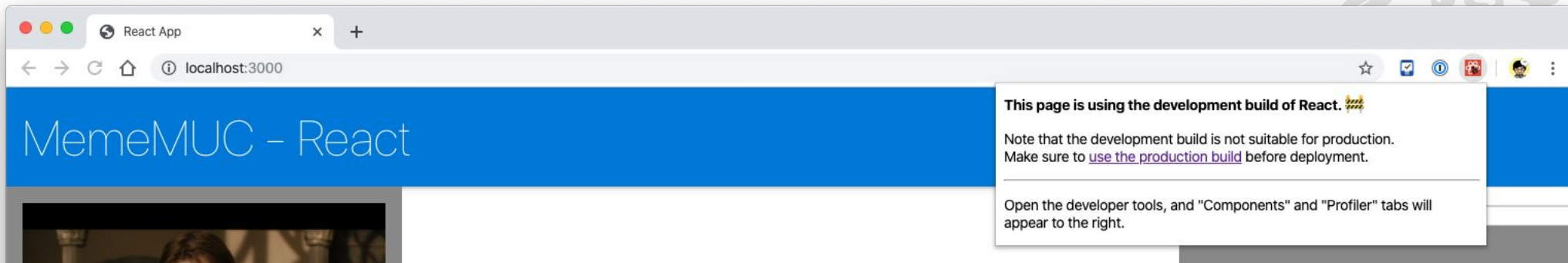
Performance Profiling

- Use Chrome Development Tool - Performance Tab
- Turn off React Developer Tools
- Make CPU 4x slowdown



Use the Production Build

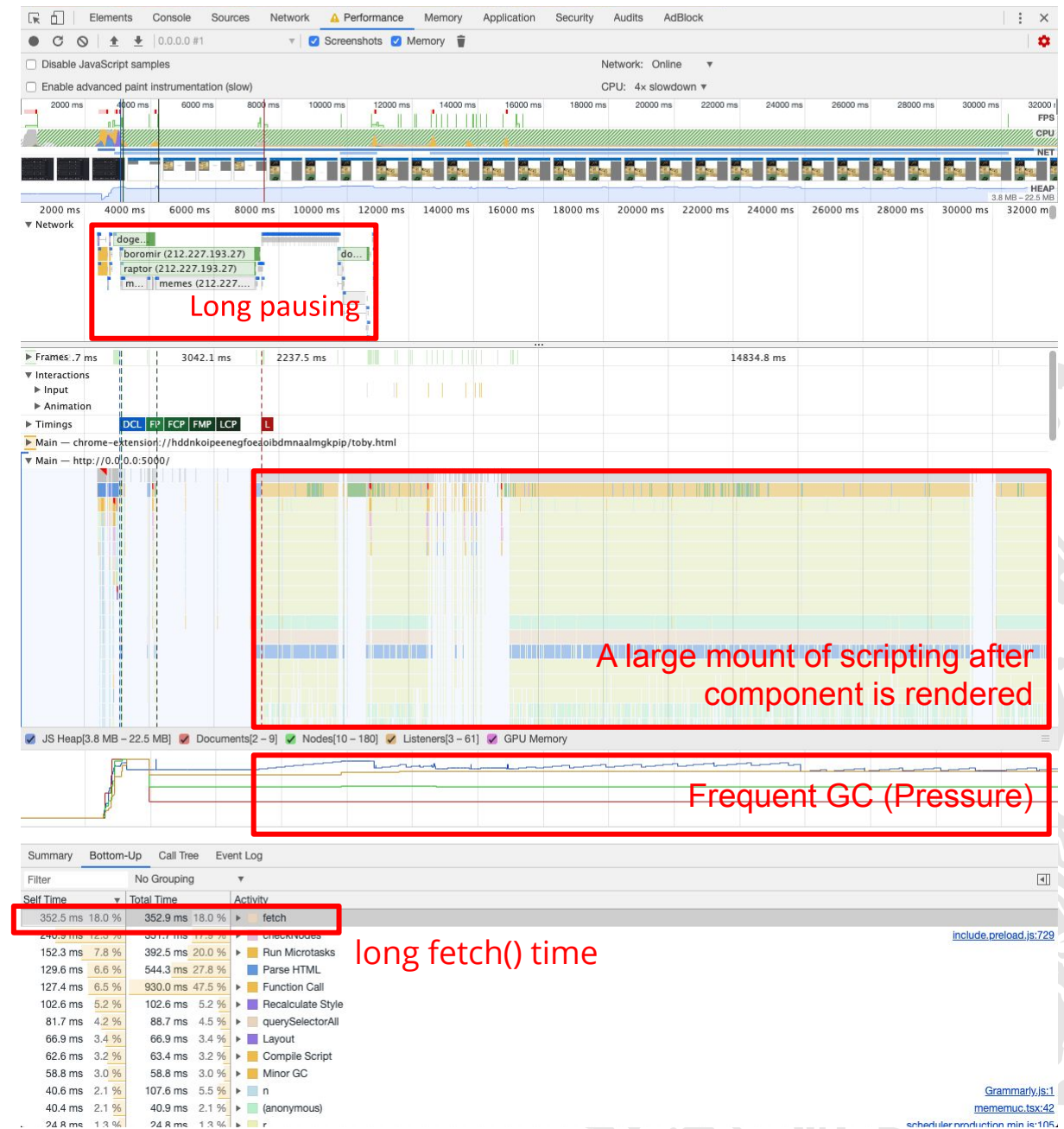
- Development builds include many debugging code which can influence profiling metrics
- Don't do any performance critical profiling under development mode.



Identifying Issues

- Why MemeMUC "feels" so slow?
- Try to recording performance from the page loading

- Loading does not providing any feedbacks
- There is a infinity calling in useEffect ()



Breakout #2: Fixing Identified Issues

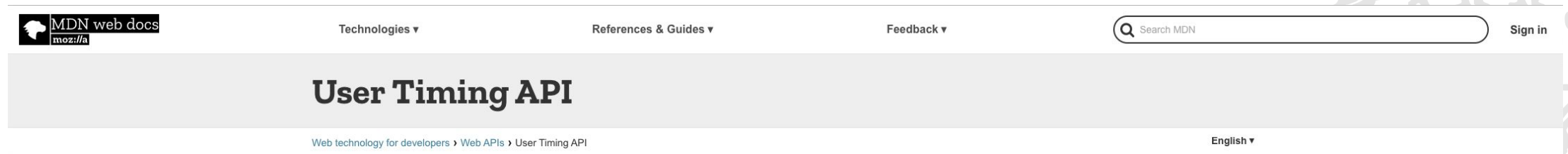
Please fixing the identified issues.

- Adding a load indicator to the MemeMUC,
 - you may want check this: <https://material-ui.com/components/progress/>
- Fix the infinity calling of `getMeme()`
 - Hint: update states only if states has changed

Timeframe: **15 Minutes**

Native Web APIs for Benchmarking

- React uses standard Timing API for benchmarking



MDN web docs
Technologies ▾ References & Guides ▾ Feedback ▾ Search MDN Sign in

User Timing API

Web technology for developers > Web APIs > User Timing API English ▾

On this Page

- Performance [marks](#)
- Performance [measures](#)
- Interoperability
- See Also

Related Topics

- ▾ Interfaces
 - Performance
 - PerformanceEntry
 - PerformanceMark
 - PerformanceMeasure

The **User Timing** interface allows the developer to create application specific [timestamps](#) that are part of the browser's *performance timeline*. There are two types of *user* defined timing event types: the "mark" [event type](#) and the "measure" [event type](#).

mark events are *named* by the application and can be set at any location in an application. **measure** events are also *named* by the application but they are placed between two marks thus they are effectively a *midpoint* between two marks.

This document provides an overview of the [mark](#) and [measure performance event types](#) including the four [User Timing](#) methods that extend the [Performance](#) interface. For more details and example code regarding these two performance event types and the methods, see [Using the User Timing API](#).

Performance marks

A performance **mark** is a *named performance entry* that is created by the application. The mark is a [timestamp](#) in the browser's *performance timeline*.

Creating a performance mark

The `performance.mark()` method is used to create a performance mark. The method takes one argument, the *name* of the mark (for example `performance.mark("mark-1")`).

The mark's [performance entry](#) will have the following property values:

Reconciliation in Repetitive Rendering

1. Check out our ToDoList example
2. Prepare 20 ToDo
3. Click performance recording
4. Click "Add New to Start"
5. Stop performance recording

key=inde
Add New to Start Add New to End


ID	created at
1	14:09:13 GMT+0100 (Central European Standard Time)
2	14:09:14 GMT+0100 (Central European Standard Time)
3	14:09:14 GMT+0100 (Central European Standard Time)
4	14:09:14 GMT+0100 (Central European Standard Time)
5	14:09:14 GMT+0100 (Central European Standard Time)
6	14:09:14 GMT+0100 (Central European Standard Time)
7	14:09:14 GMT+0100 (Central European Standard Time)
8	14:09:14 GMT+0100 (Central European Standard Time)
9	14:09:15 GMT+0100 (Central European Standard Time)
10	14:09:15 GMT+0100 (Central European Standard Time)
11	14:09:15 GMT+0100 (Central European Standard Time)
12	14:09:15 GMT+0100 (Central European Standard Time)
13	14:09:15 GMT+0100 (Central European Standard Time)
14	14:09:15 GMT+0100 (Central European Standard Time)
15	14:09:16 GMT+0100 (Central European Standard Time)
16	14:09:16 GMT+0100 (Central European Standard Time)
17	14:09:16 GMT+0100 (Central European Standard Time)
18	14:09:16 GMT+0100 (Central European Standard Time)
19	14:09:16 GMT+0100 (Central European Standard Time)
20	14:09:17 GMT+0100 (Central European Standard Time)


Elements Console Sources Network Performance Memory Application Security Audits AdBlock * Components * Profiler

(no recordings) Screenshots Memory

Disable JavaScript samples Network: Online

Enable advanced paint instrumentation (slow) CPU: No throttling

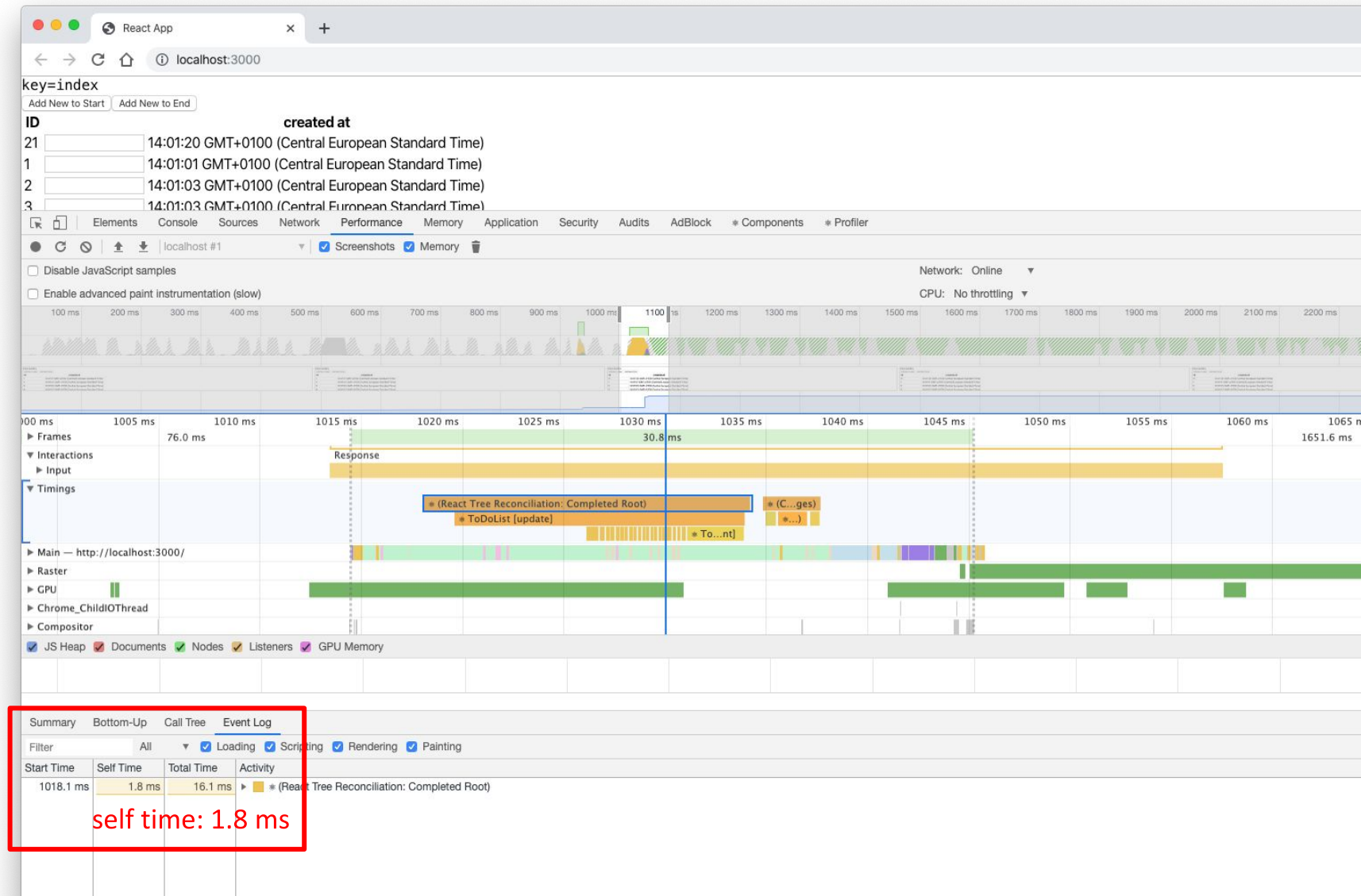
Click the record button  or hit **⌘ E** to start a new recording.

Click the reload button  or hit **⌘ ⌘ E** to record the page load.

After recording, select an area of interest in the overview by dragging. Then, zoom and pan the timeline with the mousewheel or **WASD** keys. [Learn more](#)

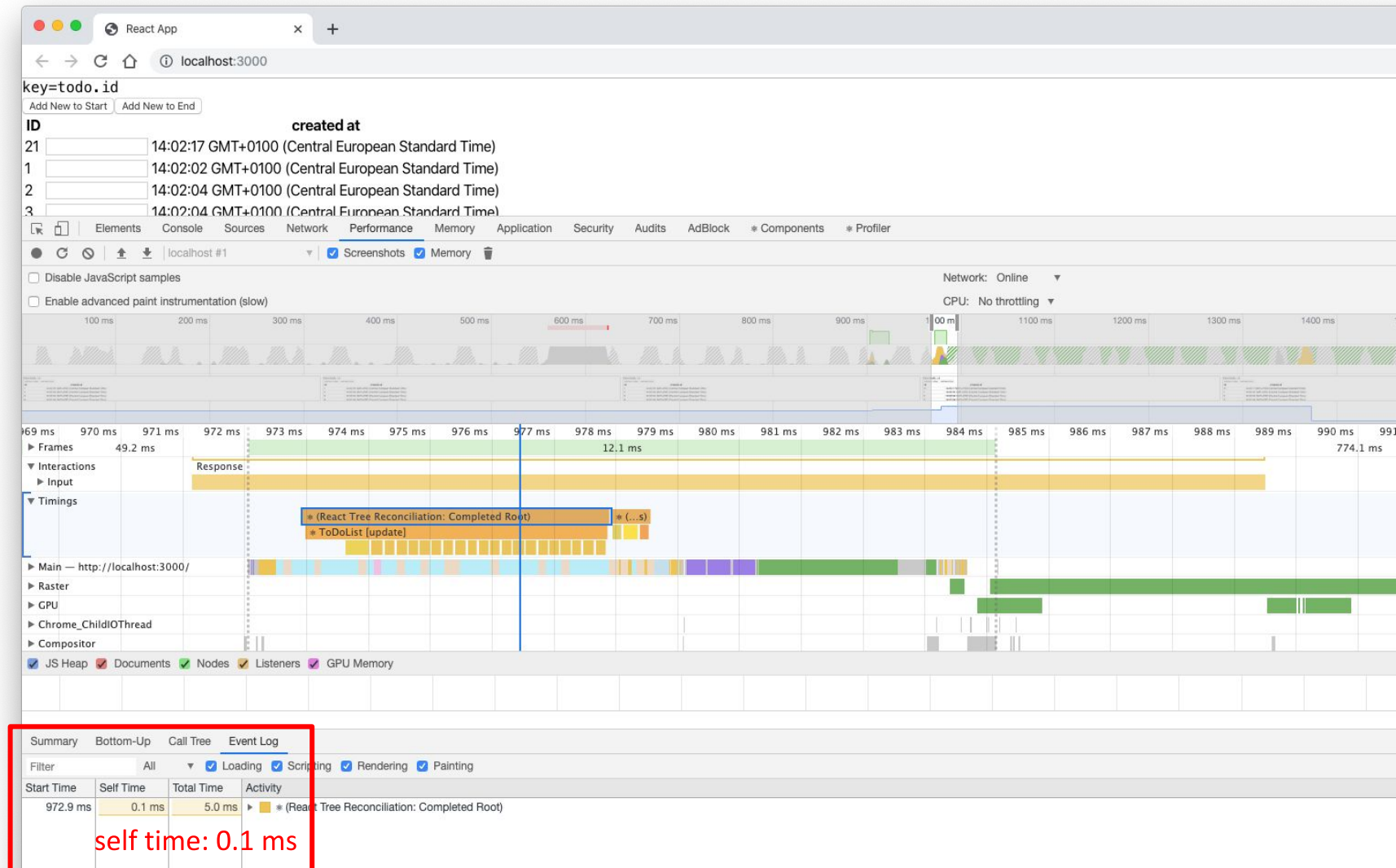
Reconciliation: Using unstable array index

- Checkout **Timings**
- Select "React Tree Reconciliation: Completed Root"
- The update time using array index is **1.8ms**



Reconciliation: Using self-defined unique ID

- Change ToDo component's key from `index` to `todo.id`
- Redo the measurement
- The update time using `todo.id` is **0.1ms**



Reconciliation Performance

- When state or props update, ReactDOM.render returns a different DOM tree, reconciliation finds out the difference and update the browser DOM **efficiently**
- The generic state-of-the-art solution is $O(n^3)$
 - 1000 elements \Rightarrow 1,000,000,000 comparisons
- React reconciliation algorithm is $O(n)$ under two assumption (perf critical)
 - **Two elements of different types will produce different trees**
 - React will not attempt to diff them, but rather replace the old tree completely
 - **Keys should be "stable, predictable and unique."**
 - Diffing of lists is performed using key prop

Round-up Quiz

1. Name three types of test method.
2. How to test a function returns an expected value in Jest?
3. What is the name convention for the testing file name?
4. Name a difference between `render` from `@testing-library/react` and `ReactDOM.render`
5. Name a reason to use production mode in performance profiling
6. Under what assumption(s) that the React reconciliation is in $O(n)$ instead of $O(n^3)$ complexity?
7. Explain why key prop should be stable, predictable and unique.

Thanks!
What are your questions?



More Links about Tooling

- Jest <https://jestjs.io/>
- React Testing Library <https://testing-library.com/docs/react-testing-library/intro>
- Timing API https://developer.mozilla.org/en-US/docs/Web/API/User_Timing_API
- Profiling React performance with React 16 and Chrome Devtools
<https://building.calibreapp.com/debugging-react-performance-with-react-16-and-chrome-devtools-c90698a522ad>
- Reconciliation and Diffing Algorithms <https://reactjs.org/docs/reconciliation.html>
- A survey on Tree Edit Distance and Related Problems
https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf

E2E Test with **cypress**

- Cypress is a Javascript End to End framework
- Install `npm install cypress --save-dev`
- Write your first e2e test:

<https://docs.cypress.io/guides/getting-started/writing-your-first-test.html#Add-a-test-file>

