SWPP Practice Session #3
Introduction to React

2022 Sep 21

Recap on the last practice session

- Useful git commands
  - add, commit, push, status
- Cooperation workflow using Github
  - create issue
  - pull request
  - resolve merge conflict

Today’s Objective

- JavaScript and TypeScript Syntax Basics
- Frontend Basics
- React Basics
- React Router

Javascript Syntax Basics

Javascript template string

- Useful way to interpolate strings into a **template string**.

```
> const name = “Changjin”, age = 20;

> console.log(“Hello, My name is ” + name + “and I’m ” + age)

> console.log(`Hello, My name is ${name} and I’m ${age}`) // note that using `(backtick) instead of quotation marks.
```

Javascript `var, let, const`

- `const`: signal that the identifier will not be reassigned.
- `let`: signal that the variable may be reassigned (block scoped)
- `var`: weakest signal available (function scoped)
- TL;DR
  - Use **const** and **let** whenever possible.
  - Only use **var** when you come across compatibility issue

```
> function hello() {
    var a = “hello”
    for (var i = 0; i < 10; i++) {
      var b = “bye”
      let c = “let bye”
    }
    console.log(a);
    console.log(b);
    console.log(c);
  }
> hello()
hello
bye
Thrown:
ReferenceError: c is not defined at
hello
```

Javascript Object Pack/Unpack Syntax

```javascript
const record = {
  year: 2018,
  name: "John",
  gender: "M",
  rank: 49,
  rankChange: -2
};
// pack
const extendedRecord = {lastName: "Doe",
...record};
// pack syntax can override previous value
when put before new values
const extendedRecord2 = {...record,
lastName: "Doe", rank: 1};
// But not the case when it is put after
values
const extendedRecord3 = {lastName: "Doe",
rank: 1, ...record};
```

```javascript
const record = {
  year: 2018,
  name: "John",
  gender: "M",
  rank: 49,
  rankChange: -2
};
// unpack a local variable
// const year = record.year, name = record.name
const { year, name } = record;
console.log(`Winner of year ${year} is ${name}!!`);
// unpack a function argument
function logger({ year, name }) {
  console.log(`Winner of year ${year} is ${name}!!`);
}
logger(record);
```

Javascript Other shorthands

```javascript
const year = 2018;
// same as {year: year, name: "Olivia"}
const record = {year, name: "Olivia"};

// Set default value of function argument
function beautifyName(name = "Einstein") {
  return "Dr. " + name;
}

// Set default value of unpack syntax
const {name = "defaultName"} = {};
function test({name = "defaultName", age}) {
  return `You, ${name}, are ${age > 18? "an adult" : "not adult"}`
}

// You can also pack and unpack an array
const primes = [2, 3, 5, 7, 11, 13, 17, 19];
const [head1, head2] = primes;
console.log(`1rd prime: ${head1}, 2nd prime: ${head2}`);

const morePrimes = [...primes, 23, 29];
```

Javascript Arrow Function

- `# General function declarations`
- `function add(a, b) { return a + b; };`
- `function negate(a) { return -a; };`
- `# Arrow function declarations`
- `const add = (a, b) => { return a + b; } # Or,`
- `const add = (a, b) => a + b; # Useful if the function returns only a single expression`
- `const negate = a => { return -a; }; # if there’s only one argument, you can omit parentheses`
- `const negate = a => -a # The simplest one-liner`

Javascript Arrow Function (Cont.)

- There is a subtle difference between general and arrow function declaration.
  - General function does not carry instance context inside function body.
- TL;DR: **Use arrow functions** for non top-level functions and lambda functions. Otherwise, use whichever you want for top-level functions.

```javascript
class Dog {
    constructor(name = "Doggo") {
        this.name = name;
    }
    bark() {
        const barker = () => {
            console.log(`${this.name}: Woof!`);
        };
        barker();
    }
    errornousBark() {
        const barker = function () {
            // Error! `this` is undefined because
            // function() { } does not carry it
            // into the body of function.
            console.log(`${this.name}: Woof!`);
        };
        barker();
    }
}
const dog = new Dog();
dog.bark(); // "Doggo: Woof!"
dog.errornousBark(); // Error!
```

Javascript `map, filter, reduce`

- Runs callback function for each item of iterable element
  - These functions are not in-place operations. That is, they construct and return a fresh array.

```
> let a = [1, 2, 3, 4, 5]

> a.map(x => x + 1)  // returns [2, 3, 4, 5, 6]

> a.filter(x => x <= 3)  // returns [1, 2, 3]

> a // [1, 2, 3, 4, 5]

> a.reduce((accum, x) => accum + x) // returns 15
```

Javascript `forEach, some, every, find`

```
> let a = [1, 2, 3, 4, 5]

> a.forEach(x => { console.log(`Hello ${x}!`) })  // iterate over an array (return value of forEach is not relevant)

> a.some(x => x == 3) // true

> a.some(x => x <= 0) // false

> a.every(x => x > 0) // true

> a.every(x => x > 1) // false

> a.find(x => x % 2 == 0) // 2

> a.find(x => x > 5) // undefined
```

Javascript is great, but its history is not

1. Javascript has rapidly evolved from tiny script to full-featured language. As a result, you could easily shoot yourself in foot if you use it carelessly. *Use modern JS language features*, not classic pre-ES6 features.
2. There are web browser compatibilities issues between JS versions.
   - Let's think about *notorious MS Internet Explorer* 😱. Check out caniuse.com for API's browser compatibility.
1. There are so many outdated practices of old era of JS on the internet out there.
   - Callback pattern has been superceded by Promise pattern. For now, Promise pattern is competing with coroutine and "await" statement.
1. JS community focuses on advancing their features and tends to not care about stability, long-term support and broken APIs.

Javascript `class`

- Basically class declaration syntax resembles the one in Java.
- Unlike Java, type declarations of arguments and return value are not required.
- Unlike python, only positional arguments exist syntactically. Rather, a dictionary-like object parameter takes the role of keyword arguments.
- Some syntax could be confusing if you’re used to python class.
  - class property declaration in Python vs. default instance attribute declaration in JS

Javascript `class` (Cont.)

```javascript
class Animal {
  constructor(maxHP) {
    this.hp = maxHP;
  }
  speak() {
    console.log("Someone is saying now..");
  }
}
```

```javascript
class Human extends Animal {
  constructor(name) {
    super(100);
    this.name = name;
  }
  speak() {
    super.speak();
    console.log(`Hi, I'm ${this.name}`);
  }
}

class Dog extends Animal {
  power = 90; // This is same as doing "this.power = 90"
in constructor
  constructor() {
    super(50);
  }

  speak() { super.speak(); console.log("Woof Woof"); }
  bite(other) { other.hp -= this.power; }
}
```

TypeScript: Type tool for JavaScript

- TypeScript is not a new language, is superset of JavaScript.
- Can make safer code.
- Has a Static Type Checker (check in compile time)
- Need to know JavaScript. TypeScript share syntax and runtime behavior with JavaScript

- Use types and interface for simple type-checking first, until familiar to ES(JavaScript) syntax
- Then recommended to read https://www.typescriptlang.org/docs/

Simple type Example

```
// printCoord.js
// $node printCoord.js
function printCoord(pt) {
 console.log("The coordinate's x value is " + pt.x);
 console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
// printCoord.ts
// $npx ts-node printCoord.ts
type Point = {
 x: number;
 y: number;
};


function printCoord(pt: Point) {
 console.log("The coordinate's x value is " + pt.x);
 console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
printCoord({ x: “100”, y: “100” });
```

Type 'string' is not assignable to type 'number'.ts(2322)
printCoord.ts(4, 3): The expected type comes from property 'x' which is declared here on type 'Point'

Simple interface Example

```javascript
// printCoord.js
// $node printCoord.js
function printCoord(pt) {
 console.log("The coordinate's x value is " + pt.x);
 console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```typescript
// printCoord.ts
// $npx ts-node printCoord.ts
interface Point {
 x: number;
 y: number;
};

interface BoundingBox extends Point {
  width: number;
  height: number;
};

function printCoord(pt: Point) {
 console.log("The coordinate's x value is " + pt.x);
 console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

interface can be extended

Simple Generic Example

```javascript
// generic.js

class GenericNumber {
 zeroValue;
 add;
}

let myGenericNumber = new GenericNumber();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
 return x + y;
};
console.log(myGenericNumber.add(5, 10));
```

```typescript
// generic.ts
// with "strictPropertyInitialization": false
class GenericNumber<NumType> {
 zeroValue: NumType;
 add: (x: NumType, y: NumType) => NumType;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
 return x + y;
};
console.log(myGenericNumber.add(5, 10));
```

Ref:

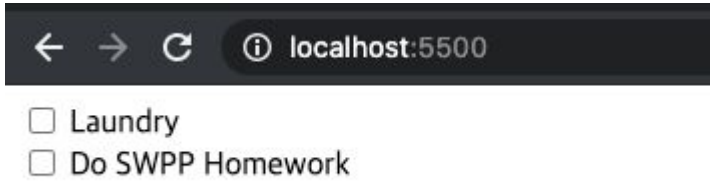
- https://www.typescriptlang.org/docs/handbook/2/generics.html#handbook-content

Frontend Basic

HTML

- Content and Structure of Web Page
  - `<div>`, `<span>`, `<p>`, `<input>`, `<form>`, etc.
- Composed of nested tags with annotations such as ID, className, name, etc.
  - Nestable: A checkbox inside “todo-item”
  - Attributes: className, name, id
- Meta information on the web page
  - title
  - which scripts to load
  - which stylesheets(CSS) to load

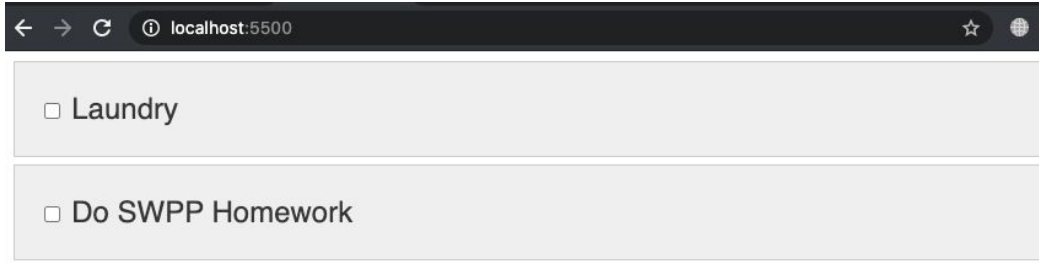
```html
<!doctype html>
<html>
    <head>
        <title>Metaa</title>
        <link rel="stylesheet" href="/test.css">
        <script src="/test.js"></script>
    </head>
    <body>
        <div class="todo-item">
            <input name="done" type="checkbox">
            <span>Laundry</span>
        </div>
        <div class="todo-item">
            <input name="done" type="checkbox">
            <span>Do SWPP Homework</span>
        </div>
    </body>
</html>
```



CSS

- CSS decorates html elements
- You can designate which elements to decorate by className.
- CSS is out of the scope of this practice session. Check out materials on CSS.
- Handling CSS itself could be cumbersome and boring work. There are useful CSS frameworks:
  - Materialized(https://materializecss.com/)
  - Semantic UI(https://semantic-ui.com/)

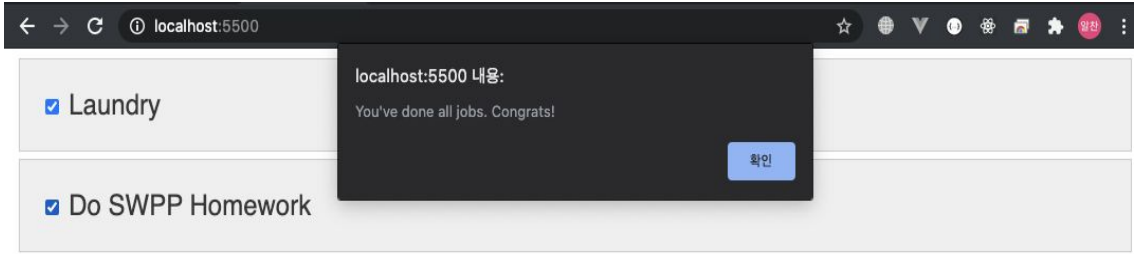
```
.todo-item {
    font-family: Helvetica;
    font-size: 1.5rem;
    margin: 0.4rem;
    padding-left: 0.8em;
    padding-top: 1em;
    padding-bottom: 1em;
    background-color: #efefef;
    border: 1px solid #ccc;
    color: #373737;
}
```



Java(Type)script as web scripting language

- handles user events (e.g. page load, click, mouse scroll, form submit, input value, etc.)
  - Able to attach event listeners on html elements.
- Dynamically add, remove or modify html elements using DOM APIs.

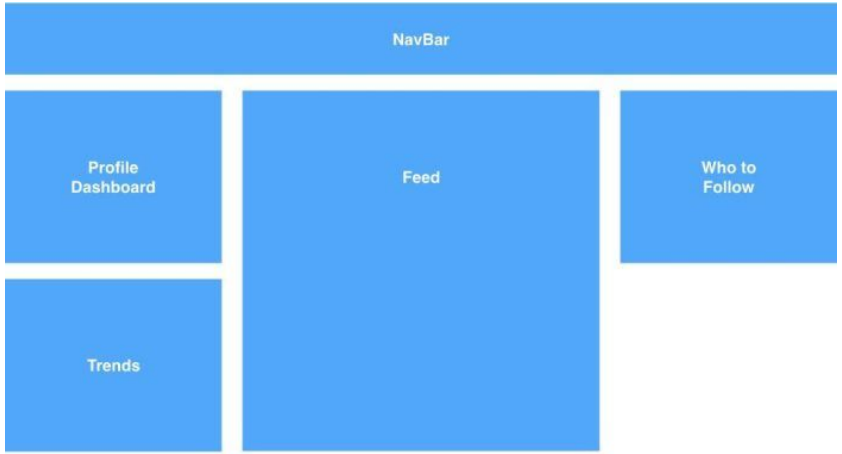
```javascript
document.addEventListener("DOMContentLoaded", () => {
    const elements = Array.from(document.getElementsByTagName("input"));
    elements.forEach(element => {
        element.addEventListener("change", () => {
            if (elements.every(element => element.checked) ) {
                setTimeout(() => alert("You've done all jobs. Congrats!"), 100);
            }
        });
    })
});
```

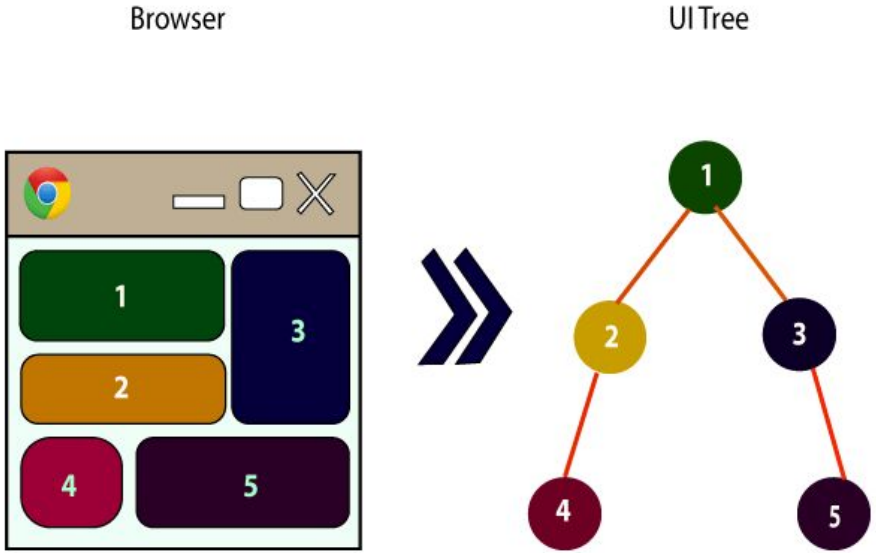


Component-based Web Framework

- Building a web site structure with plain HTML and JS for large website is a repetitive and tedious work.
- Also, the result of it is often hard to maintain and reuse.
- Plus, Plain JS API is not programmer-friendly for generating html elements.
- Here, component frameworks come to save us!
- By using component framework, you can think about web page in piece-by-piece manner and assemble components. You can also reuse those components.

Component-based Web Framework

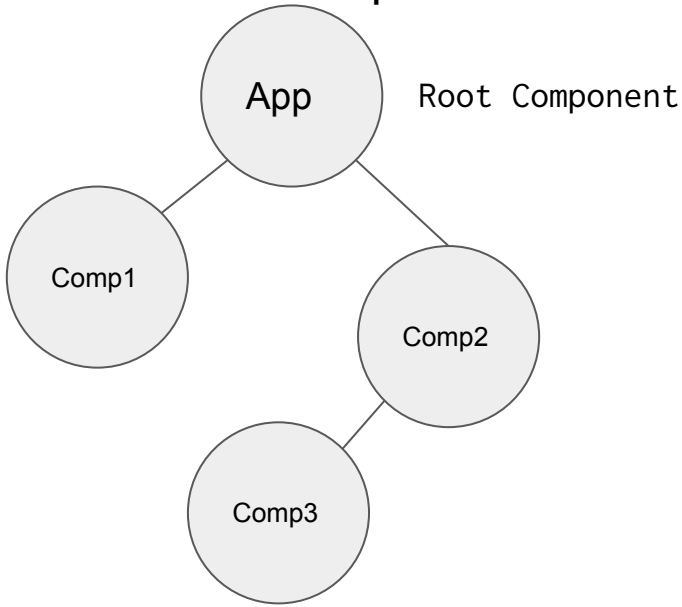




React Basics

React

- Declarative, efficient, and flexible JavaScript library for building UI developed by Facebook
- Compose complex UIs from “**components**”, small and isolated pieces of code.
- Facebook, Instagram, Netflix… using React!
- https://github.com/facebook/react



Notes about React practice session

- Learning React using these dense materials might be overwhelming to follow up right now. It’s totally fine!
- Because of the complexity of React, we’re not going to demand you to hand out exercise today. You may hand it out until tomorrow 9PM.
- If you have a question after practice session, please leave it to our issue board.

Fork and Clone Project

- Fork repository
  https://github.com/swpp22fall-practice-sessions/swpp-p3-react-tutorial.git
- And clone your repository.
- Type `yarn` in command line to install frontend dependencies

```
* [master] initial stage
 ! [newtodo] new todo finished
  ! [routing] finish routing
   ! [tododetail] todo detail finished
    ! [todolist] done until basic todolist
---
  +   [routing] finish routing
 ++   [newtodo] new todo finished
 +++  [tododetail] todo detail finished
 ++++ [todolist] done until basic todolist
*++++ [master] initial stage
```

The `root` folder (reference)

File	Purpose
`node_modules/`	`Node.js` creates this folder and puts all **third party modules** listed in `package.json` inside of it.
`.gitignore`	Git configuration to make sure auto-generated files are not committed to source control.
`package.json`	`npm/yarn` configuration listing the third party packages your project uses. You can also add your own [custom scripts]() here.
`yarn.lock`	“Locked” Version information of dependencies in package.json
`public`	has `index.html`, favicon, logo etc...
`tsconfig.json`	TypeScript settings for type checking and compile [doc]()

Keep the App Running & First Step

- # under project root folder, type following command
- $ yarn start
- # you can check out your app on http://localhost:3000
- # modify App.tsx file as

```
import "./App.css";

function App() {
 return <div className="App"></div>;
}

export default App;
```

Let’s Create the First Component!

- Structuring workspace is important when you build large project
- Make directory at `src/containers/TodoList`
- `Edit src/containers/TodoList/TodoList.tsx`
- The simplest form of component is

```
interface IProps {
 title: string;
}

export default function TodoList(props: IProps) {
   const {title} = props
   return  <div className="TodoList">{title}</div>;
}
```

Code inside function will execute whenever

What’s wrapped in {} is interpreted as a javascript expression.
Without this, it would show the text “title” as is.

Basic Component Structure

- Function components are a simpler way to write components.
- It returns a **single React element**, written in JSX syntax.
- We can write a function that takes **props** as input and returns what should be rendered.
- Function components are less tedious to write than classes, and many components can be expressed this way.

```
import TodoList from "./containers/TodoList/TodoList"; // can omit.js

function App() {
 return <div className="App"><TodoList title={"My TODOs!"} /></div>;
}
```

props passed

JSX: Why are those HTML tags inside code?

- You might have noticed some kind of HTML tag is included in JS/TS code.
- React handles web structure in code, not in *.html files. For this, React uses **JSX**, a syntactic extension of Javascript. It helps a programmer to mix up HTML with JS/TS.
- React transpilers convert this tags into equivalent code stub. By this manner, React includes HTML facilities into code without hurting readability.
- TypeScript must have “.tsx” extention (not “.ts”) for JSX transpiling.

```
<div className='TodoList'></div>
/* actually, it uses className, not class to avoid collision btw JS class
* this syntax is compiled to React.createElement(‘div’, {className: ‘TodoList’})
* by React Transpiler. */
```

Add Some TODOs

- Functional Component holds its state using `useState<S>(initialState)`
- Set a components initial state as

```
import { useState } from "react";
...
type TodoType = { id: number; title: string; content: string; done: boolean;};

export default function TodoList(props: IProps) {
 const { title } = props;
 const [todos, setTodos] = useState<TodoType[]>([
   { id: 1, title: "SWPP", content: "take swpp class", done: true },
   { id: 2, title: "Movie", content: "watch movie", done: false },
   { id: 3, title: "Dinner", content: "eat dinner", done: false },
 ]);
...}
```

Add Todo Component

```
// src/components/Todo/Todo.tsx
interface IProps {
 title: string;
 clicked?: React.MouseEventHandler<HTMLDivElement>;  // Defined by React
 done: boolean;
}

const Todo = (props: IProps) => {
 return (
   <div className="Todo">
     <div className={`text ${props.done && "done"}`} onClick={props.clicked}>
       {props.title}
     </div>
     {props.done && <div className="done-mark">&#x2713;</div>}
   </div>
 );
};
export default Todo;
```

Render Todos in App

- at `TodoList.tsx`, import `Todo`

```
import Todo from '../../components/Todo/Todo';
```

- Modify `render()` as

```
return (
  <div className="TodoList">
    <div className="title">{title}</div>
    <div className="todos">
      {todos.map((td) => {
        return <Todo key={td.id} title={td.title} done={td.done} />
      })}
    </div>
  </div>
);
```

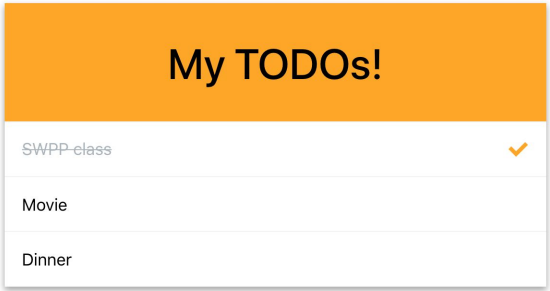
Each child in a list should have a unique "key" prop.
React use key for manage nodes efficiently.

Looks like...

My TODOs
SWPP
✓
Movie
Dinner

Add some CSS!

- Download and add css files
- Add `import "./TodoList.css"` and `import "./Todo.css"` at the beginning of `TodoList.tsx` and `Todo.tsx` respectively.
- https://github.com/chang-jin/swpp-react-tutorial/blob/todolist/src/containers/TodoList/TodoList.css
- https://github.com/chang-jin/swpp-react-tutorial/blob/todolist/src/components/Todo/Todo.css



Hooks

- Hooks are a new addition in React 16.8 (2018)
- Hooks let you use more of React’s features without classes.
- Basic hooks: useState, useEffect, useContext
- Additional hooks: useReducer, useCallback, useMemo...
- See more: https://reactjs.org/docs/hooks-intro.html

Hooks

- Only Call Hooks at the Top Level
  - Don’t call Hooks inside loops, conditions, or nested functions.
- Only Call Hooks from React Functions
  - In function component or custom hooks

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

function Example() {
 const [count, setCount] = useState(0);

 return (
   <div>
     <p>You clicked {count} times</p>
     <button onClick={() => setCount(count + 1)}>Click me</button>
   </div>
 );
}
```

Custom Hooks

- Lets you extract component logic into reusable functions.

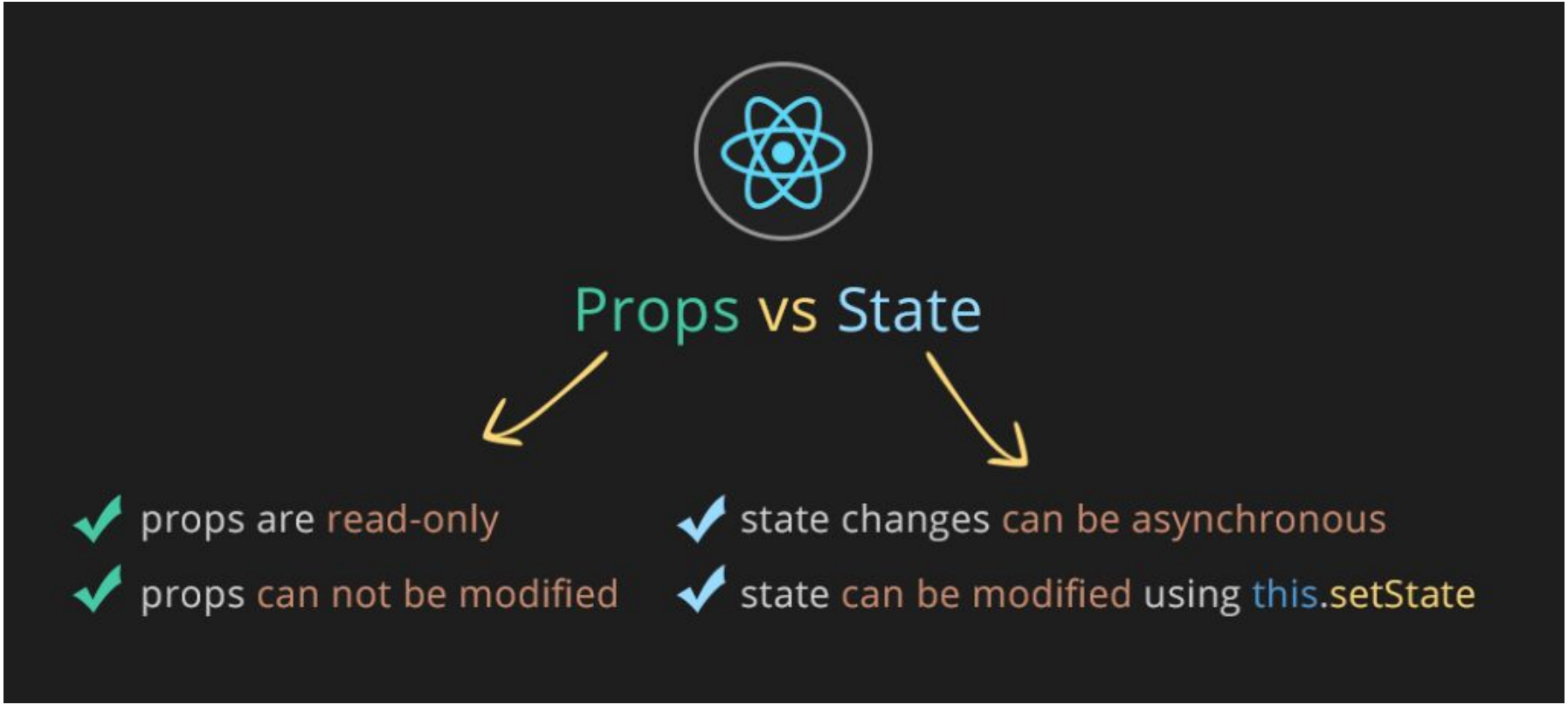
```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
 const [isOnline, setIsOnline] = useState(null);
 useEffect(() => {
   function handleStatusChange(status) { setIsOnline(status.isOnline);}

   ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
   return () => {
     ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
   };
 });
 return isOnline;
}
```

Class-based component

- Some (old) codes would be using class-based component.
- You need to understand component lifecycle, State, Props for using Class-based component.
- https://reactjs.org/docs/react-component.html

```
class ScrollingList extends React.Component {
   constructor(props) {
     super(props);
     this.listRef = React.createRef();
   }
    getSnapshotBeforeUpdate(prevProps, prevState) {
     if (prevProps.list.length < this.props.list.length) {
       const list = this.listRef.current;
       return list.scrollHeight - list.scrollTop;
     }
     return null;
   }
    componentDidUpdate(prevProps, prevState, snapshot) {
     if (snapshot !== null) {
       const list = this.listRef.current;
       list.scrollTop = list.scrollHeight - snapshot;
     }
   }
    render() {
     return (
       <div ref={this.listRef}>{/* ...contents... */}</div>
     );
   }
 }
```

Two informative objects in component: Props and State



Show Detailed Information

- add `selectedTodo` to state in `TodoList`

```
export default function TodoList(props: IProps) {
...
  const [selectedTodo, setSelectedTodo] = useState<TodoType | null>(null);
...
}
```

- set `onClick` event for each `todos` in `TodoList`

```
{todos.map((td) => {
  return <Todo title={td.title} done={td.done} clicked={() => clickTodoHandler(td)} />;
})}
```

Show Detailed Information

- In `TodoList`, implement `click` handler

```
const clickTodoHandler = (td: Todo) => {
  if (selectedTodo === td) {
    setSelectedTodo(null);
  } else {
    setSelectedTodo(td);
  }
};
```

- **Never** mutate `` `state` `` directly. Use `` `setState` ``.

Show Detailed Information

- At src/components/TodoDetail, write `TodoDetail` component and put [css](#) file

```
import "./TodoDetail.css";

type Props = {
 title: string;
 content: string;
};

const TodoDetail = (props: Props) => {
 return (
   <div className="TodoDetail">
     <div className="row">
       <div className="left">Name:</div>
       <div
className="right">{props.title}</div>
     </div>
```

```
     <div className="row">
       <div className="left">Content:</div>
       <div className="right">
         {props.content}
       </div>
     </div>
   </div>
 );
};
export default TodoDetail;
```

Pass SelectedTodo to TodoDetail

- inside `TodoList()`, (and do not forget to import `TodoDetail` and `useMemo`!)

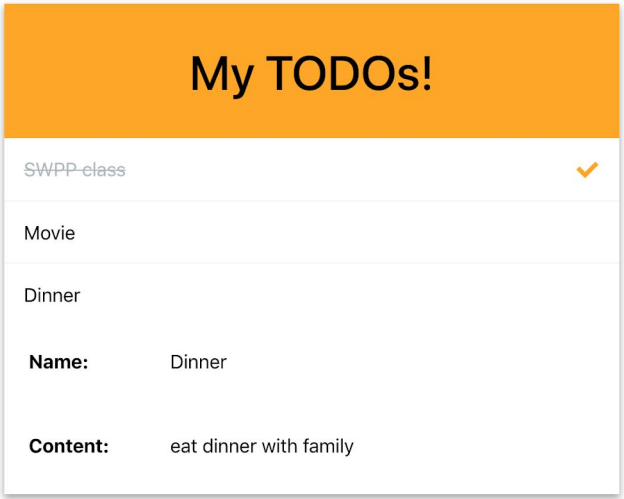
```
const todoDetail = useMemo(() => {
  return selectedTodo ? (
    <TodoDetail title={selectedTodo.title} content={selectedTodo.content} />
  ) : null;
}, [selectedTodo]);
```

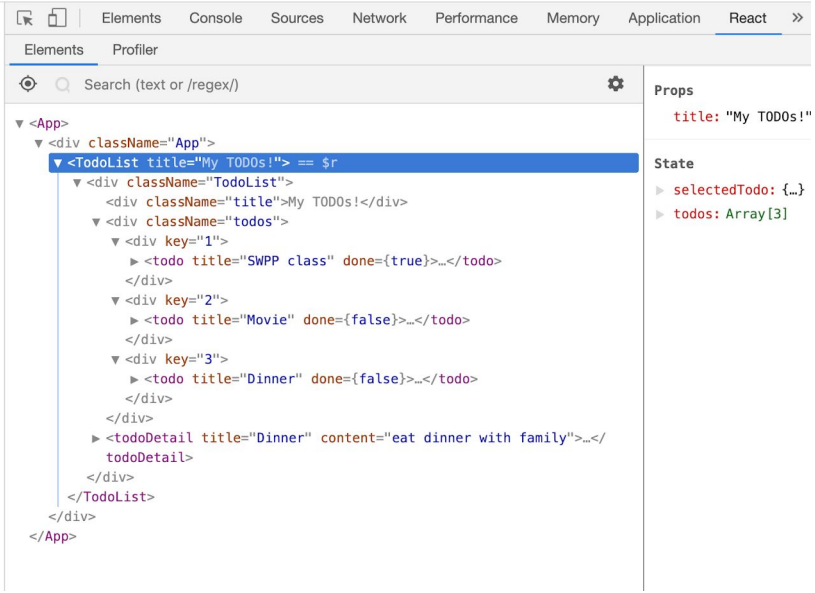
- Finally, modify `return()` statement of `TodoList()` as

```
...
  <div className="title">{title}</div>
  <div className="todos">
    ...
    {todoDetail}
  </div>
...
```

Useful Tip

- You can install React chrome extension in here
- Explore components structure in developer tool





Add New Todo

- We want to add new todo item to our TodoList
- Let’s add a new component in `containers/TodoList/NewTodo/NewTodo.tsx`
- Get css from here

```react
import { useState } from "react";
import "./NewTodo.css";

export default function NewTodo() {
 const [title, setTitle] = useState<string>("");
 const [content, setContent] = useState<string>("");
 const [submitted, setSubmitted] =
useState<boolean>(false);

 return (
   <div className="NewTodo">
     <h1>Add a Todo</h1>
     <label>Title</label>
     <input
       type="text"
       value={title}
       onChange={
         (event) => setTitle(event.target.value)
       }
     />
     <label>Content</label>
     <textarea
       rows={4}
       value={content}
       onChange={
         (event) => setContent(event.target.value)
       }
     />
     <button onClick={
       () => alert("Submitted")
     }>Submit</button>
   </div>
 );
}
```

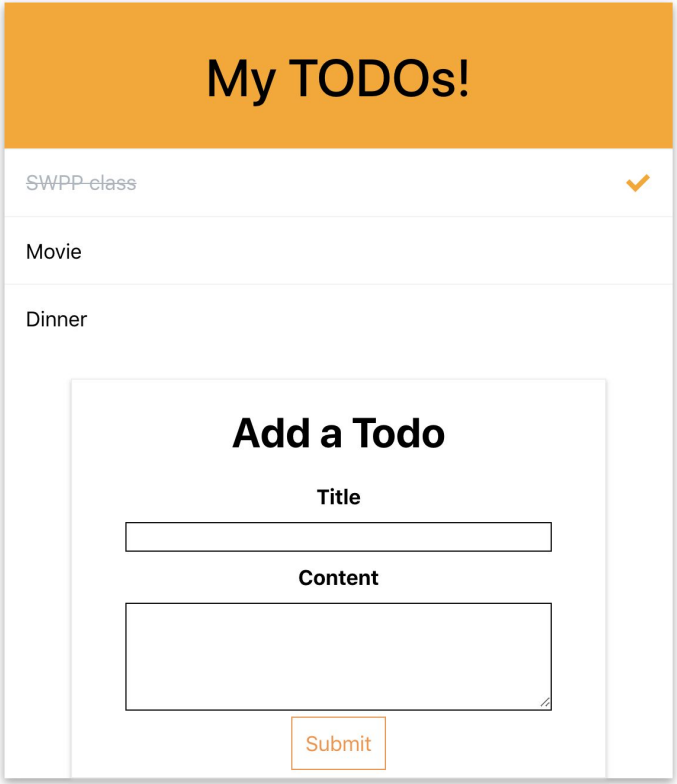
2way binding

Add NewTodo to TodoList

- inside `NewTodo()`, (and do not forget to import `NewTodo`!)

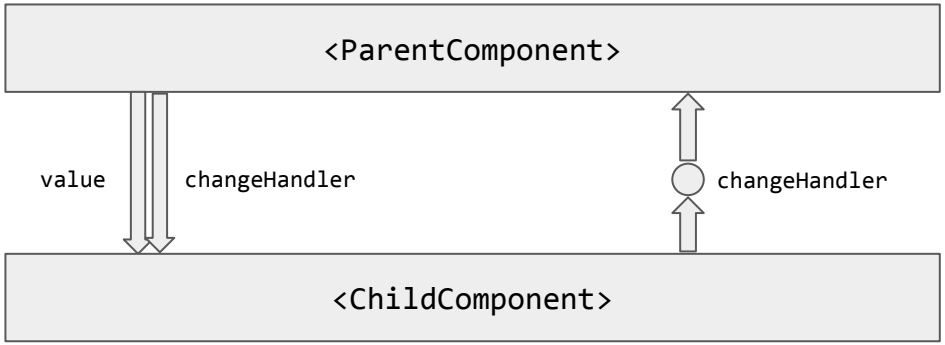
```
...
  <div className="title">{title}</div>
  <div className="todos">
    ...
    {todoDetail}
  <NewTodo />
  </div>
...
```

Now it looks like...



Pass down to and Receive from a Child Component

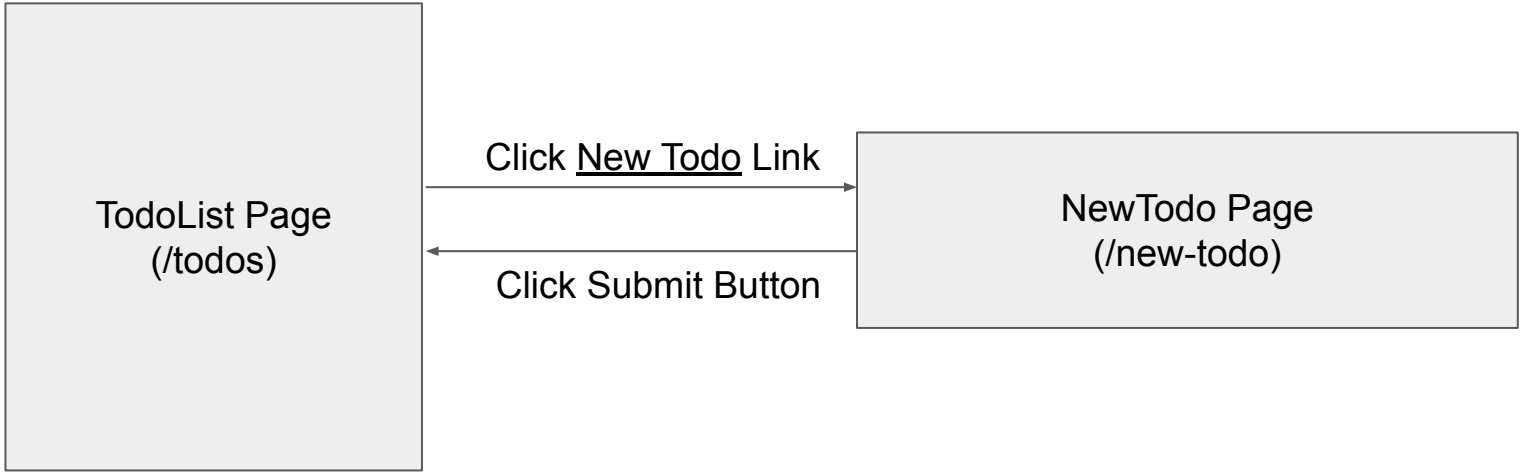
- A parent component passes a new value to its child via **props**
- A child component tosses information back to its parent with **callback that was passed in props**.
- This is a common communication scheme called ***two-way binding***.
- An obvious example is `<input>` component where value changes flows back to its parent by `onChange` handler. This mechanism can be also utilized in child-parent communication between user-defined components.



React Router

Route Pages

- Single react project == single web page
- Want to make app looks like multi-page app



Create NavLink to NewTodo

- Inside `TodoList` component

```
import { Link } from "react-router-dom";
```

- Create `NavLink` after `{todoDetail}`, then check `localhost:3000/new-todo`

```
    ...
    {todoDetail}
    <NavLink to="/new-todo" >New Todo</NavLink>
  </div>
  ...
```

Install Required Dependency

- `# Run following command`
- `$ yarn add react-router react-router-dom`

- `# now start app again!`
- `$ yarn start`

Wrap App.tsx Component

- Import components

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
```

- Modify `return()` as
  - `BrowserRouter` allows child components to use `Route`

```
<div className="App">
  <BrowserRouter>
    <Routes>
      <Route path="todos" element={<TodoList title={"My TODOs!"} />} />
      <Route path="new-todo" element={<NewTodo />} />
    </Routes>
  </BrowserRouter>
</div>
```

Add Some Route Behavior

- Replace `return` of `NewTodo` component. Use If-else state using `submitted`
- Don't forget to add `import { Navigate } from "react-router-dom";`
- Then check in browser!

```
...
import { Navigate } from "react-router-dom";
...

  if (submitted) {
    return <Navigate to="/todos" />;
  } else {
    return (
      <div className="NewTodo">
      ...
```

Add Some Route Behavior

- We want to go home right after when we submit new todo
- First of all, at `NewTodo.tsx`, let's implement `postTodoHandler()`

```
const postTodoHandler = () => {
 const data = { title: title, content: content };
 alert("Submitted\n" + data.title + "\n" + data.content);
 setSubmitted(true);
};
```

- Now modify `onClick()` as

```
<button onClick={() => postTodoHandler()}>Submit</button>
```

Add Some Route Behavior

- Alternative way to redirect is using `useNavigate`

```
import { useNavigate } from "react-router-dom";

...
 const navigate = useNavigate()
 const postTodoHandler = () => {
   const data = { title: title, content: content };
   alert("Submitted\n" + data.title + "\n" + data.content);
   setSubmitted(true);
   navigate('/todos')
 };
...
```

Add Some Route Behavior

- We want to alias `localhost:3000` as `localhost:3000/todos` (actually, it's a redirect)
- Go back to `App.tsx` and add follows

```
import { BrowserRouter, Routes, Route, Navigate } from "react-router-dom";
...
   <Routes>
     <Route path="/todos" element={<TodoList title={"My TODOs!"} />} />
     <Route path="/new-todo" element={<NewTodo />} />
     <Route path="/" element={<Navigate replace to={"/todos"} />} />
   </Routes>
...
```

Use `replace` option to prevent stacking history

Add Some Route Behavior

- Finally we want to notice users that they are in invalid page
- Modify and check some invalid page (e.g. /swpp)

```
<div className="App">
  <BrowserRouter>
    <Routes>
      <Route path="/todos" element={<TodoList title={"My TODOs!"} />} />
      <Route path="/new-todo" element={<NewTodo />} />
      <Route path="/" element={<Navigate replace to={"/todos"} />} />
      <Route path="*" element={<h1>Not Found</h1>} />
    </Routes>
  </BrowserRouter>
</div>
```

Add Some Route Behavior

- Let's assume we want to make independent, separated `TodoDetail` page
- Go back to `App.tsx` and add follows
- We will modify `TodoDetail` next week with Redux + Axios

```
<div className="App">
  <BrowserRouter>
    <Routes>
      <Route path="/todos" element={<TodoList title={"My TODOs!"} />} />
      <Route path="/todos/:id" element={<TodoDetail />} />
      <Route path="/new-todo" element={<N
      <Route path="/" element={<Navigate
      <Route path="*" element={<h1>Not Fo
    </Routes>
  </BrowserRouter>
</div>
```

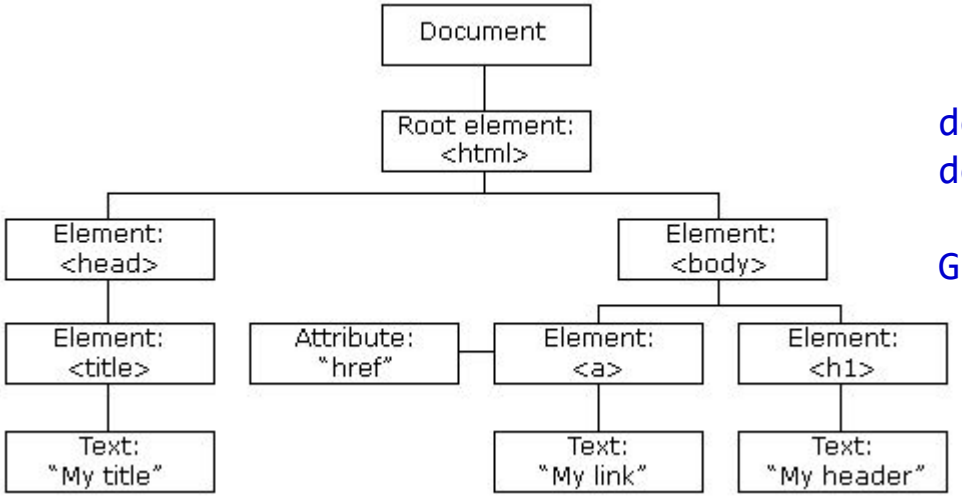
```
Type '{}' is missing the following
properties from type 'Props': title,
content: ts(2739)
```

Wrap up

- Functional Component
- Hooks
- Routing
- There’s no additional exercise today
  - Please pull-request your works to the original repository.
  - You need to finish practice until tomorrow 21:00 for your attendance check.

Additional React Features

DOM (Document Object Model)

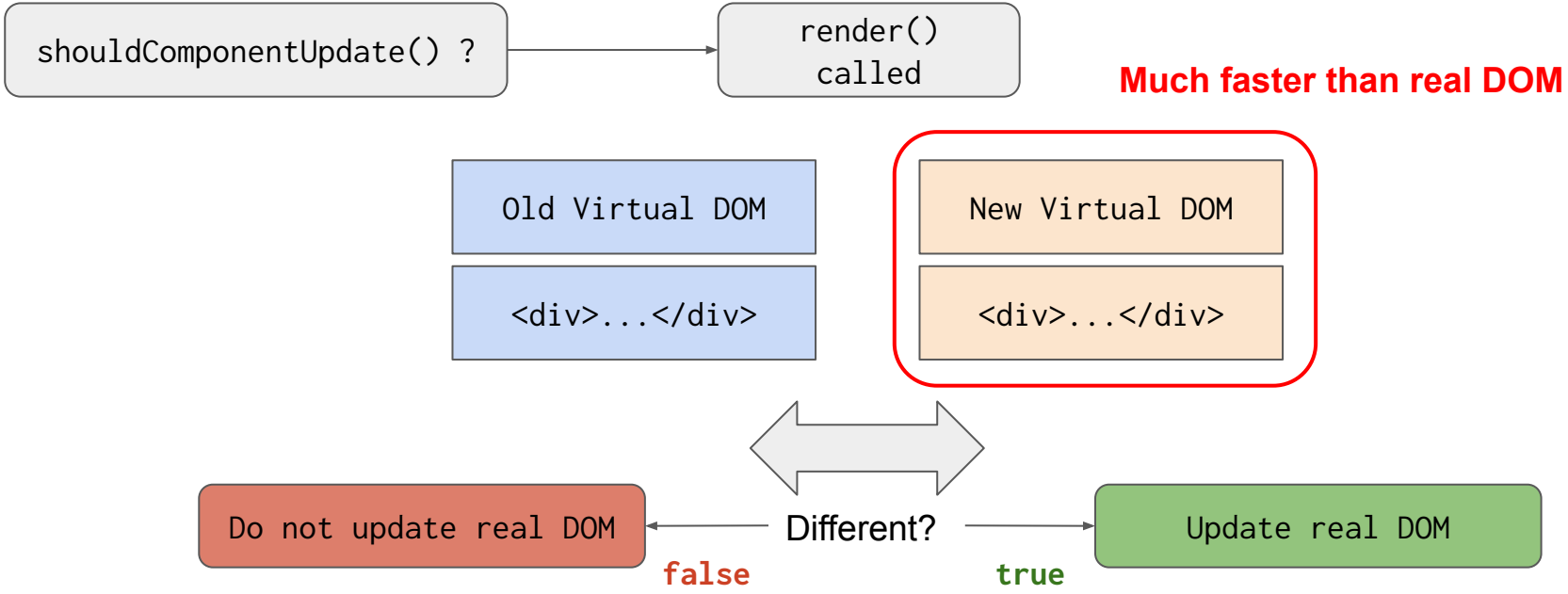



```
document.getElementById('uniqueID')
document.body.getElementById('uniqueID')
```

Great Interfaces provided by the Browser

How React Updates DOM

- React uses **Virtual DOM**



Next Week...

- State management with redux
- Flux/Promise design pattern
- HTTP request using axios

Useful References

1. https://reactjs.org/tutorial/tutorial.html
2. https://velopert.com/
3. https://code.tutsplus.com/tutorials/stateful-vs-stateless-functional-components-in-react--cms-29541
4. https://reactjs.org/docs/hooks-intro.html
5. https://reactjs.org/docs/jsx-in-depth.html
6. https://www.typescriptlang.org/docs/