Software Studio
軟體設計與實驗

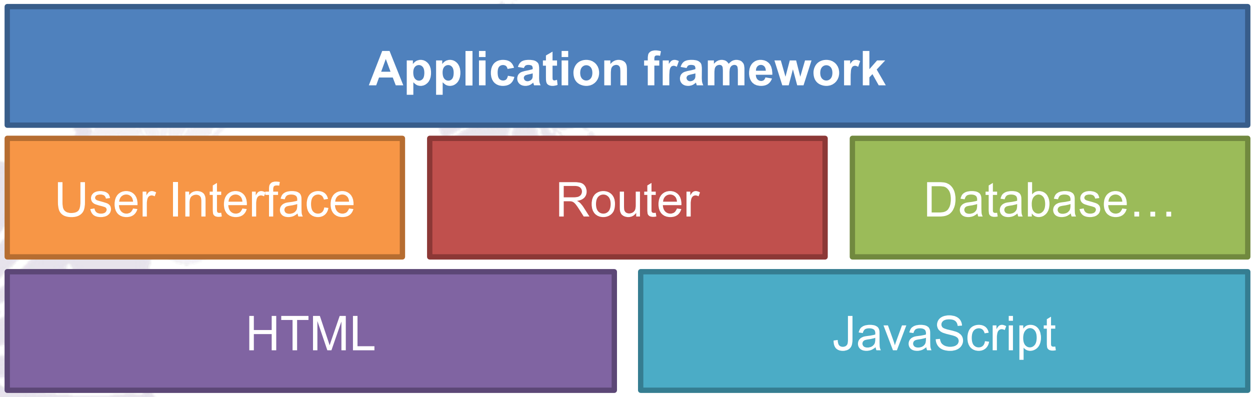
# Web Application Framework (Part I)

**Hung-Kuo Chu**
Department of Computer Science
National Tsing Hua University

CS2410

## What is Web Application Framework?

- We have learned how to write a website with HTML and JavaScript.
- Many applications nowadays use **web application framework** to build up.
- A set of APIs that facilitates the development of web applications.



## User Interface Framework

| React | Angular | VueJS |
|---|---|---|
| • Only deal with view.<br>• React Native.<br>• Big Community. | • Use Typescript to implement. | • Only deal with view.<br>• Vue Native.<br>• Laravel Cooperation |

JavaScript 框架大比拼！

# Outline

- Introduction to React
- Introduction to Webpack
- Environment setting
- Tutorial
- Advanced techniques

# React

## Introduction

- React is a JavaScript library for building user interfaces.
- React can be used as a base in the development of single-page or mobile applications.

# Why we use React?

- Component-Based, easy to develop
- Use JavaScript to generate HTML
- Use **Virtual DOM**, more efficient

## Component-Based

- Pre-defined HTML tags such as <div>, <image> or <input> are sometimes not flexible.
- React supports **customized components** by packing the HTML structures into JS codes.
- Using JS to generate dynamic HTML structures without editing HTML codes.

## Example

```
class Example extends React.Component {
  render() {
    return (
      <ul>
        <li>Coffee</li>
        <li>Tea</li>
        <li>Milk</li>
      </ul>
    );
  }
}
```

**list.js**

## Example (Cont'd)

```
class Home extends React.Component {
  render() { // define HTML structure
    return (
      <Example /> // customized react component
    );
  }
}

ReactDOM.render(<Home />,
document.getElementById("div-home"));
```

**home.js**

## Example (Cont’d)

```
<body>
 //…
 <div id="div-home"></div>
</body>
```

**Index.html**

```
<body>
 //…
 <ul>
   <li>Coffee</li>
   <li>Tea</li>
   <li>Milk</li>
 </ul>
</body>
```

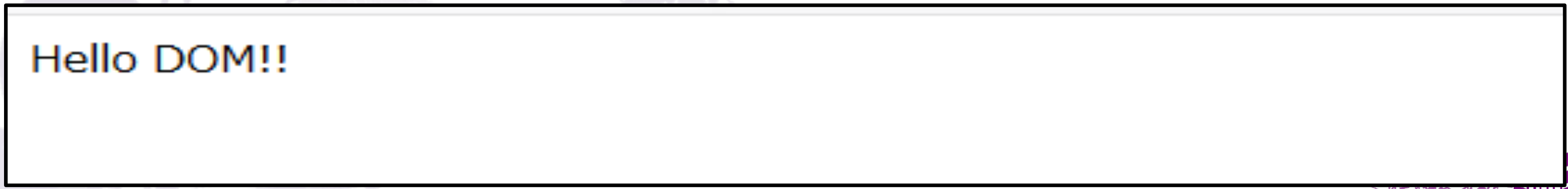
**Result**

## What is the Virtual DOM?

- A programming concept where an ideal, or “**virtual**”, representation of a UI is kept in memory and synced with the “**real**” DOM by a library such as ReactDOM.
- When any changes of UI occurred, a new virtual DOM tree is created and is “differed” with the previous virtual DOM tree.
- More efficient than conventional DOM.

# Recap: DOM

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>DOM example #1</title>
    <script type="text/javascript">
        function init() {
            var text = document.getElementById("dom1");
            text.innerHTML = "Hello DOM!!";
        }
    </script>
</head>
<body onload="init();">
    <p id="dom1"></p>
</body>
</html>
```



```
Hello DOM!!
```

# DOM Example (explained)

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>DOM example #1</title>
    <script type="text/javascript">
     function init() {
          var text =
          document.getElementById("dom1");
          text.innerHTML = "Hello DOM!!";
    }
    </script>
</head>
<body onload="init();">
    <p id="dom1"></p>
</body>
</html>
```

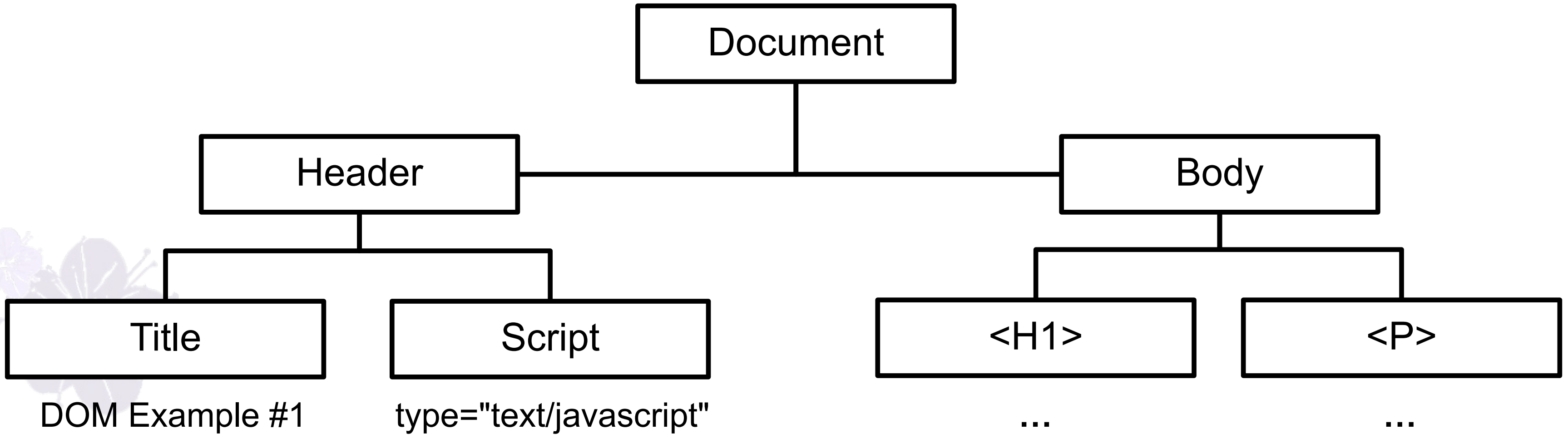
First, we use **getElementById** to get the object with specific id. ("dom1“ in this example)

And then we use **innerHTML** to modify **content** of this object to display our string.

The <p> is an object in JavaScript with "**dom1**“ as its id.

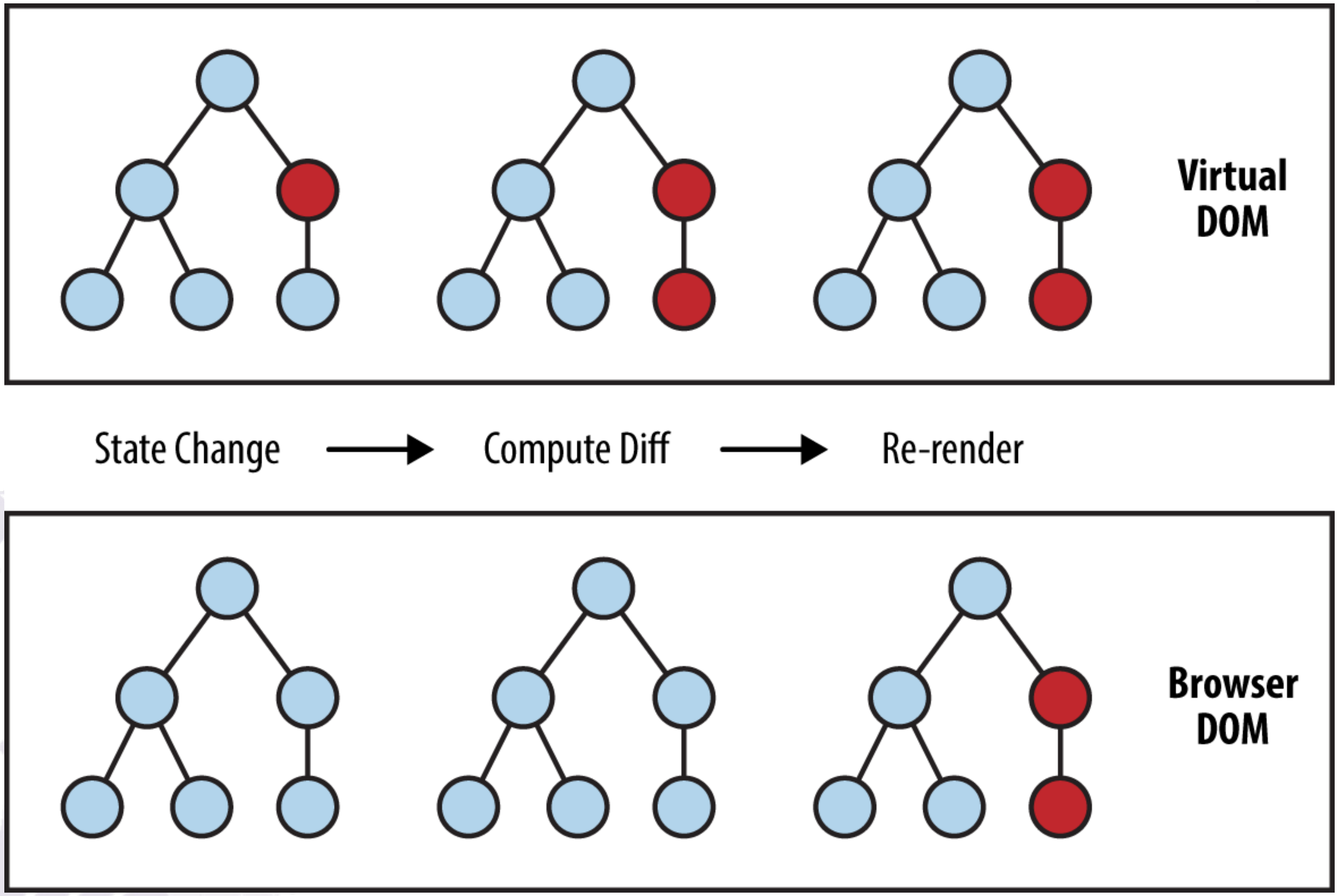
We can use getElementById to modify its content.

# More about DOM



HTML page is a tree structure with many nodes.
Each node has its own data and attribute.

# Virtual DOM: Concept



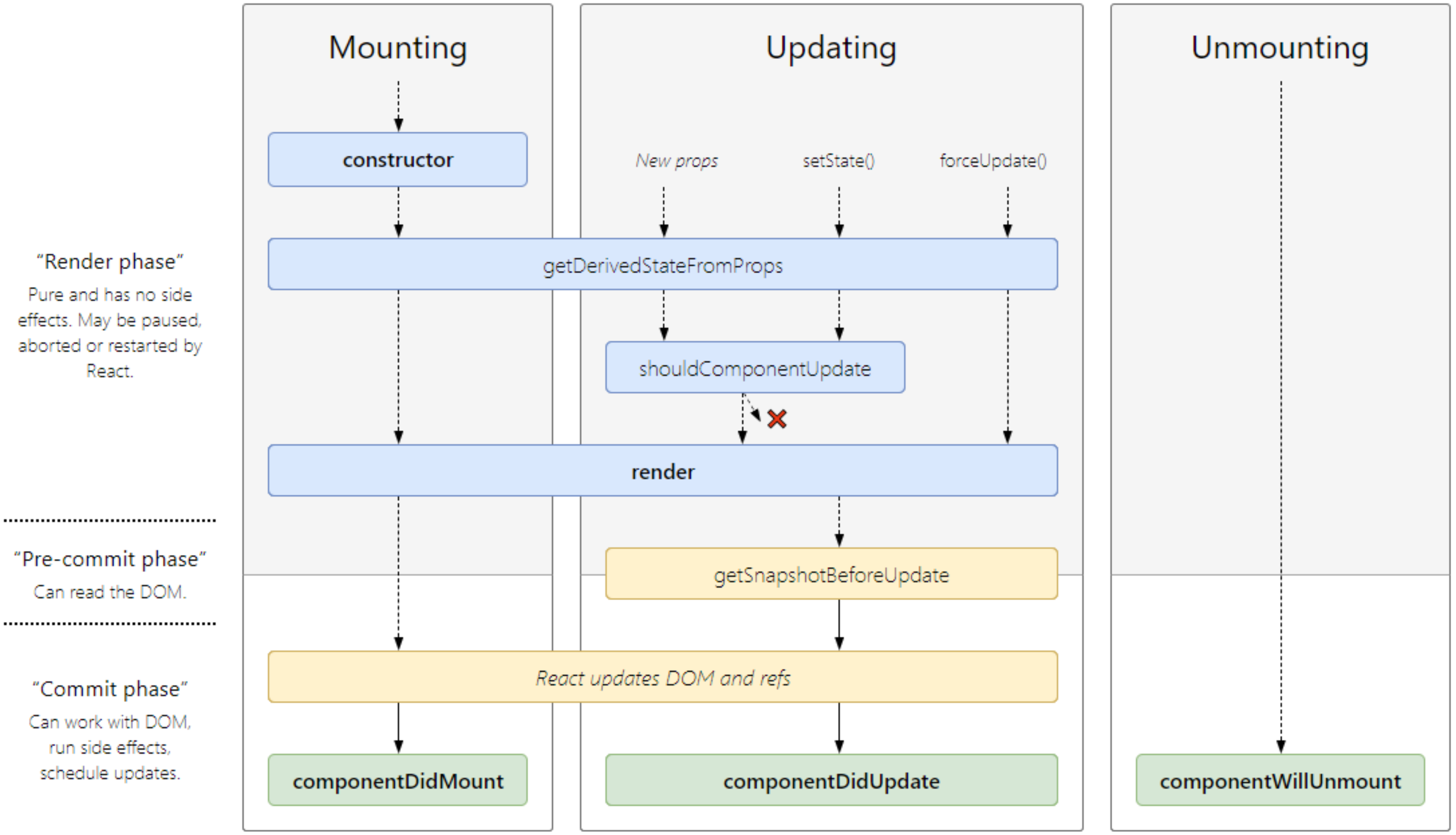
source: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html

React Virtual DOM Explained

## React Component Lifecycle

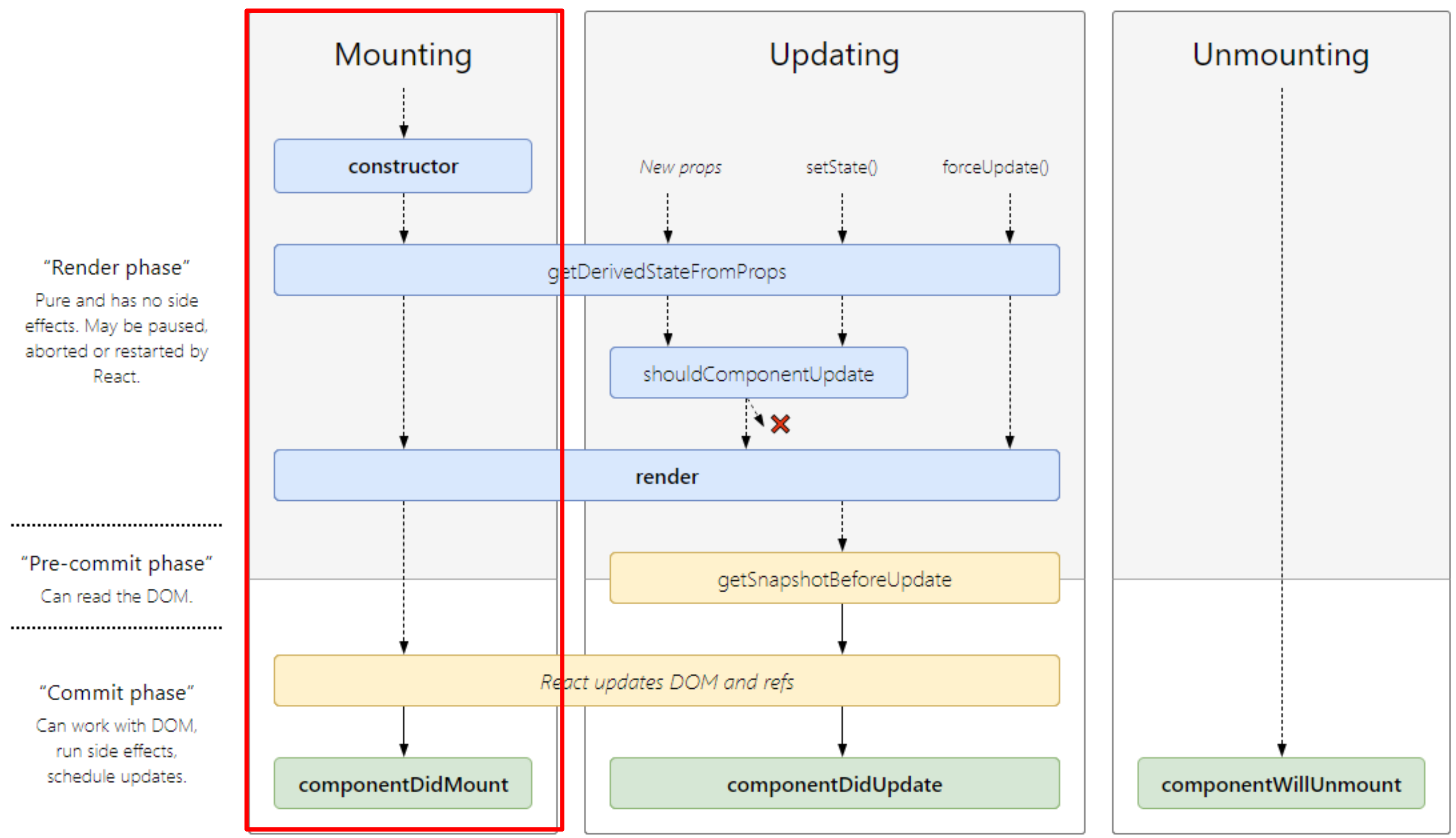
- React component has 3 types of lifecycle:
  1. **Mounting** : Component initialization
  2. **Updating** : Component update
  3. **Unmounting** : Component uninstall
- In each state, React provides **lifecycle method** that you can override to run customized codes during the process.

# React Component Lifecycle



[React.Component]

# Mounting State



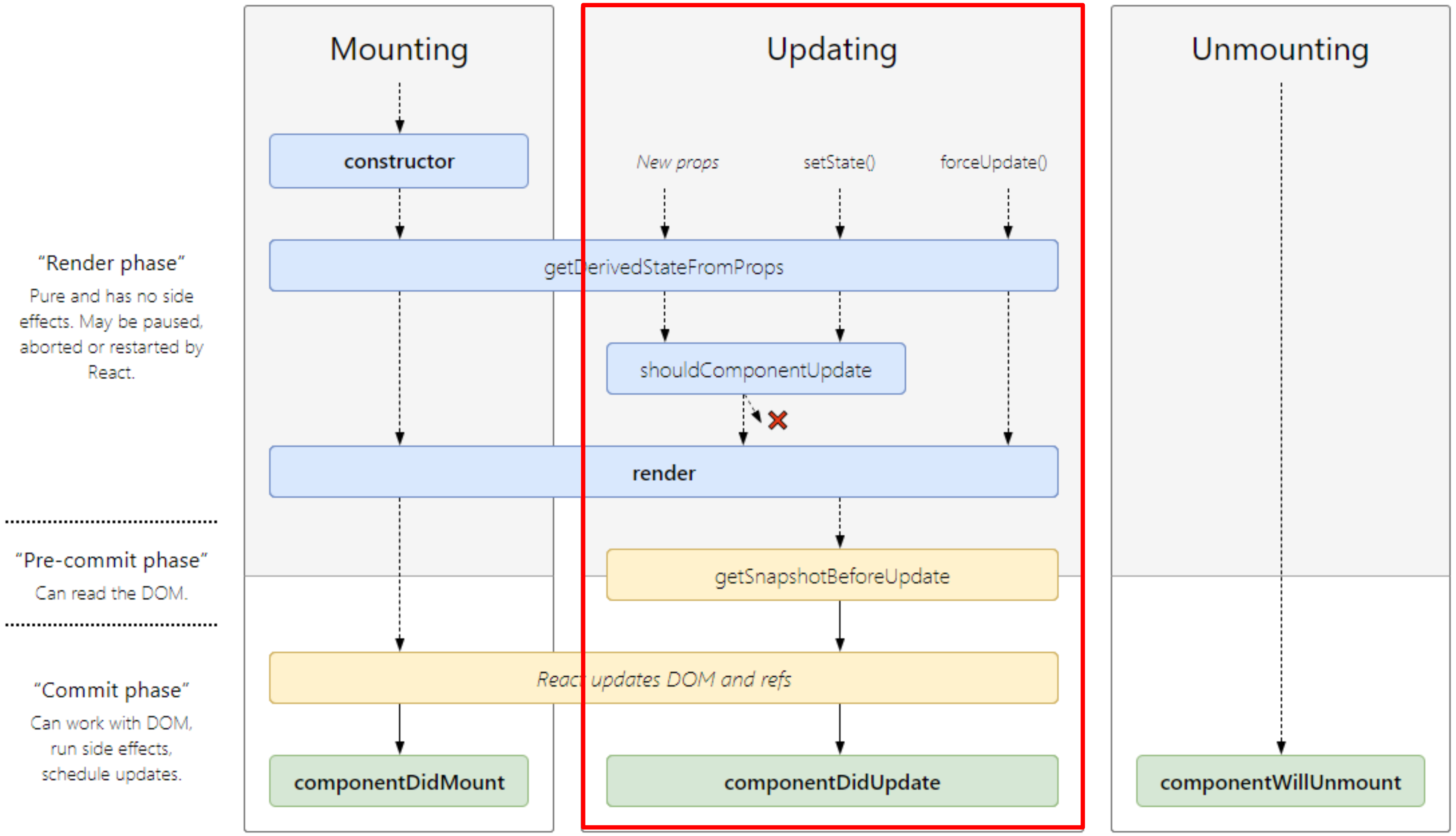
## Mounting State Methods

- The following methods are called in the specified order when an instance of a component is **created and inserted into the DOM.**
  1. constructor()
  2. static getDerivedStateFromProps()
  3. render()
  4. componentDidMount()

## Mounting State: Rule of Thumb

- Interacting with the browser (if needed) in **componentDidMount()** or the other lifecycle methods instead.
- Keeping **render()** pure makes components easier to think about.
- Avoid introducing any **side-effects** (e.g., data fetching or an animation) or **subscriptions** in the **constructor**, using **componentDidMount()** instead.

# Updating State



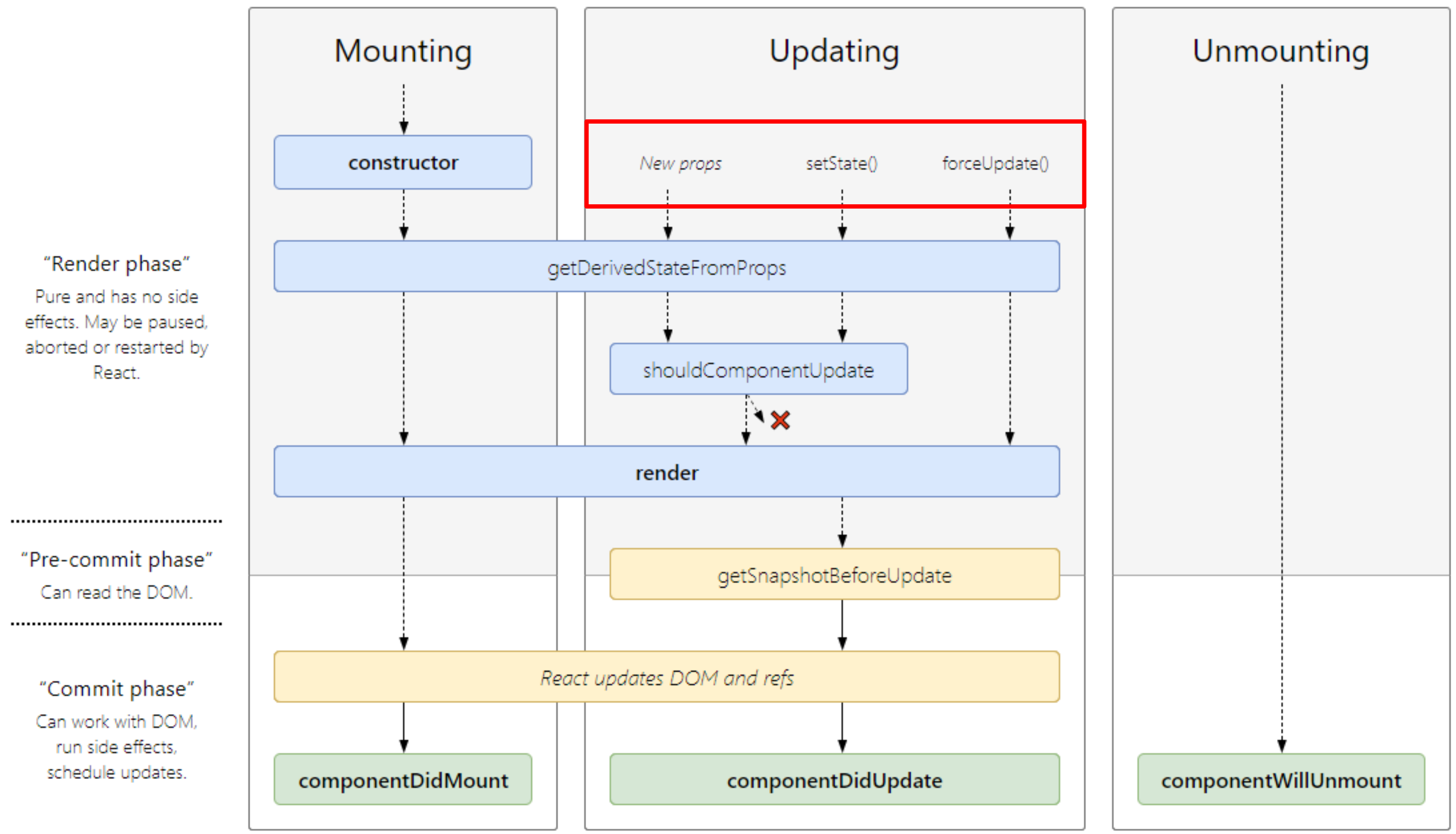
## Updating State Methods

- The following methods are called in the specified order when a component is being re-rendered.
  1. static getDerivedStateFromProps()
  2. shouldComponentUpdate()
  3. render()
  4. getSnapshotBeforeUpdate()
  5. componentDidUpdate()

## Updating State: Rule of Thumb

- Using **componentDidUpdate()** to performing any **side-effects** (e.g., data fetching or an animation) in response to a change in props.

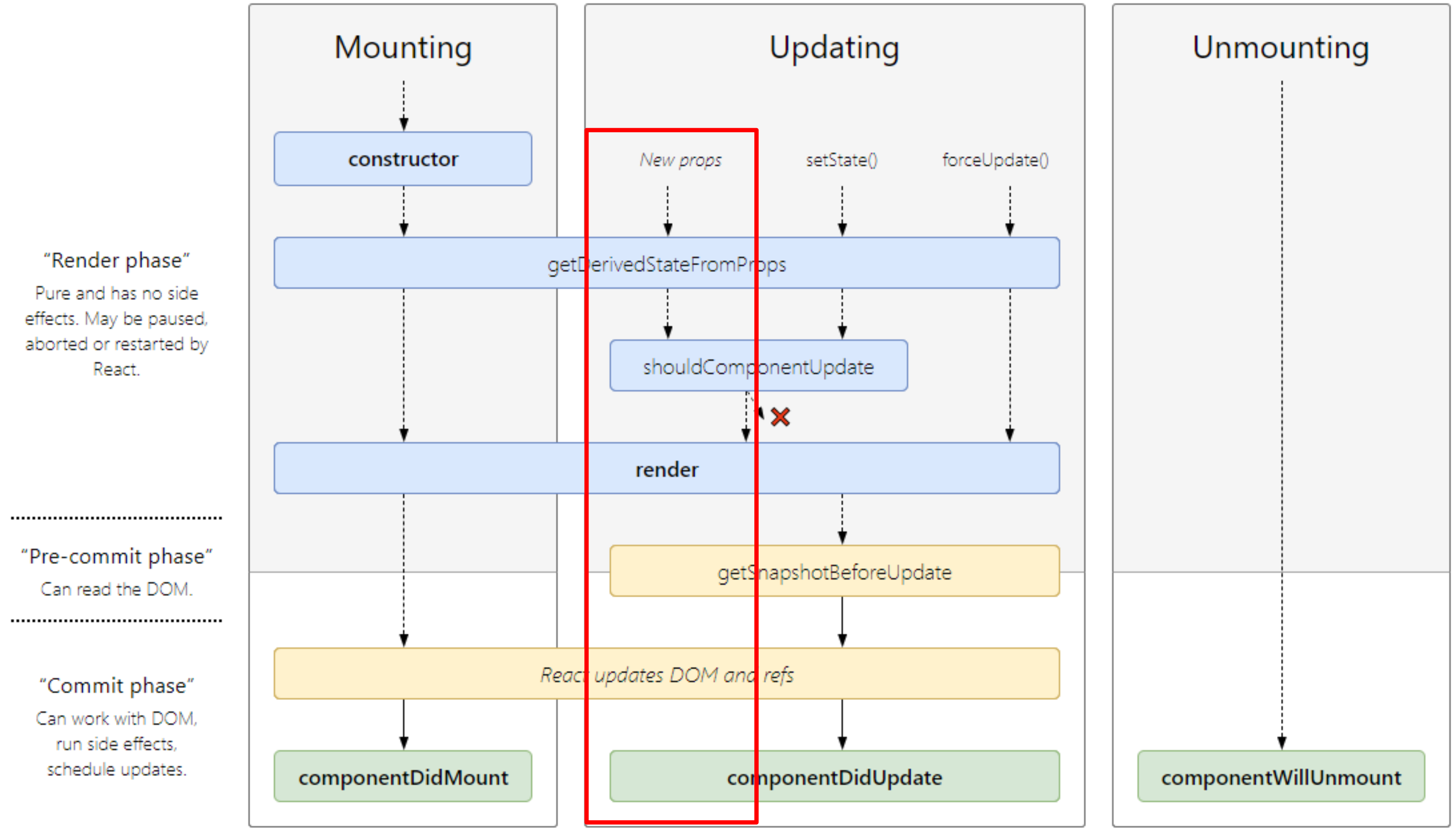
# Updating State



## Updating State

- React won't continuously update component.
- React only updates components in the following situations:
  1. **props changed**
  2. **state changed**
  3. **forceUpdate()**
- Different trigger mechanism corresponds to different calling sequence of state methods.

# Updating State



## React Component: props

- **props** is a **read-only** parameter used for communication between a component and its child component(s).
- When the component builds up a child component, it can assign parameter(s) to it.
- We can then use **this.props** in the child component to get parameters from its parent.

# props: Example

```
class Home extends React.Component {
  render() { // define HTML structure
    return (
      <div>
        <Example id={‘0’}/>   // id represents a prop
        <Example id={‘1’}/>
      </div>
    );
  }
}
```

**home.js**

```
class Example extends React.Component {
  constructor(props){
    super(props);
    console.log("this component id : " + this.props.id);
  }
}
```

**list.js**

## props: Example (Cont'd)

```
var first_id = "0";
var second_id = "1";

class Home extends React.Component {
    render() { // define HTML structure
    return (
      <div>
        <Example id={first_id}/>   // id represents a prop
        <Example id={second_id}/>
      </div>
    );
  }
}
```

**home.js**

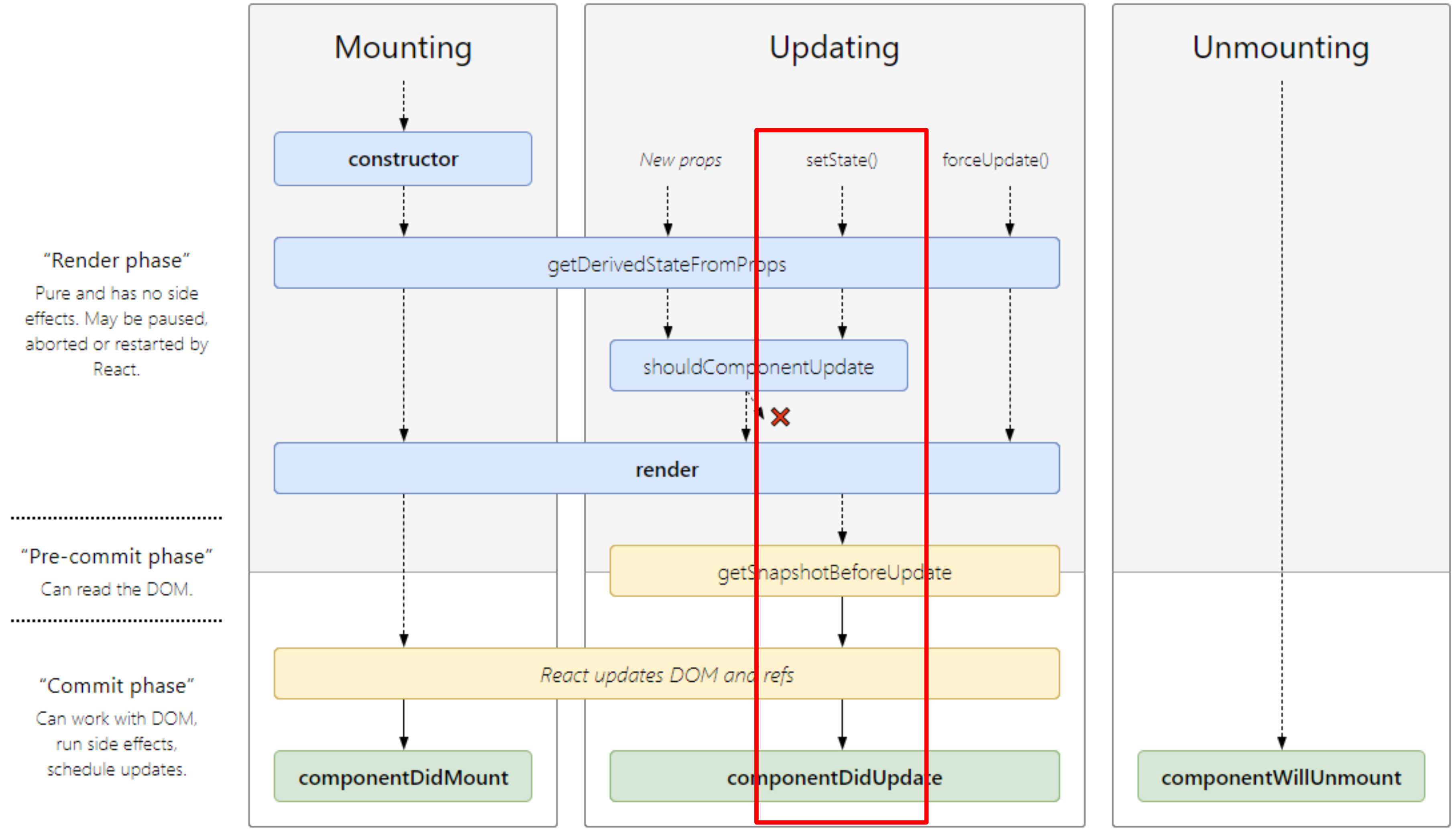
## props: Example (Cont'd)

```
var first_id = "0", second_id = "1";

class Home extends React.Component {
  constructor(props){
    super(props);
    this.user_name_1 =  "cgvlab";
    this.user_name_2 =  "James";
  }
  render() { // define HTML structure
    return ( // here we pass two props to Example component!
      <div>
        <Example id={first_id} name={this.user_name_1} />
        <Example id={second_id} name={this.user_name_2} />
      </div>
    );
  }
}
```

**home.js**

# Updating State



## React Component: state

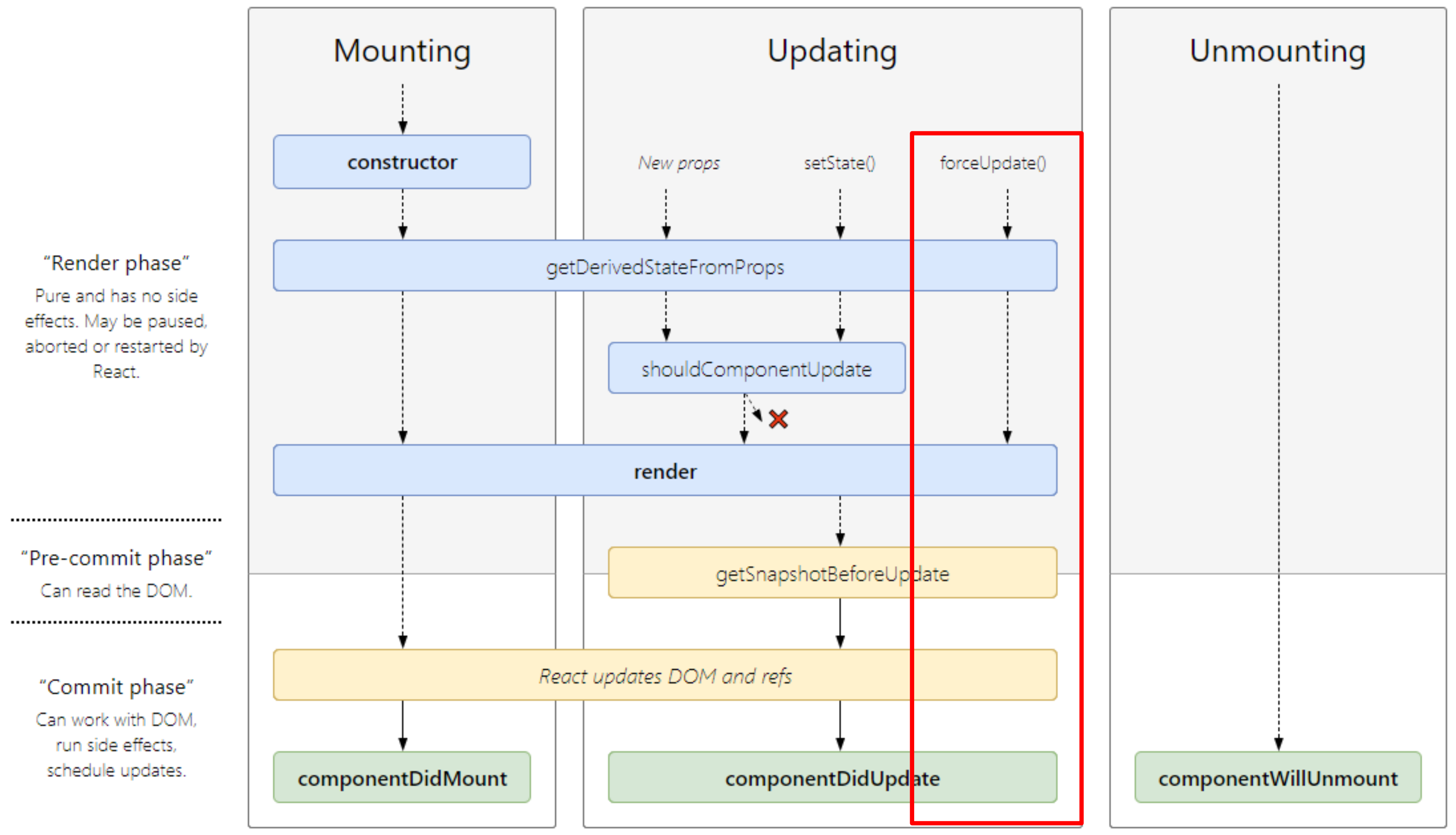
- state is an **inner variable** of component.
- It **can't** be changed directly.
- Its value can only be edited through **this.setState()**.
- Each call of **this.setState()** will lead to component re-rendering. Use it wisely to avoid infinite loop!

# state: Example

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      user_id: -1,
      user_name: ""
    }
  }
  changeUserProfile(new_id, new_name){
    this.setState({
       user_id : new_id,
       user_name : new_name
    })
  }
}
```

**list.js**

# Updating State



## React Component: forceUpdate()

- In some case, we use mechanisms other than **props** or **state** to control the appearance of component.
- React provides forceUpdate() to forcibly trigger the updating (a.k.a re-render the component).

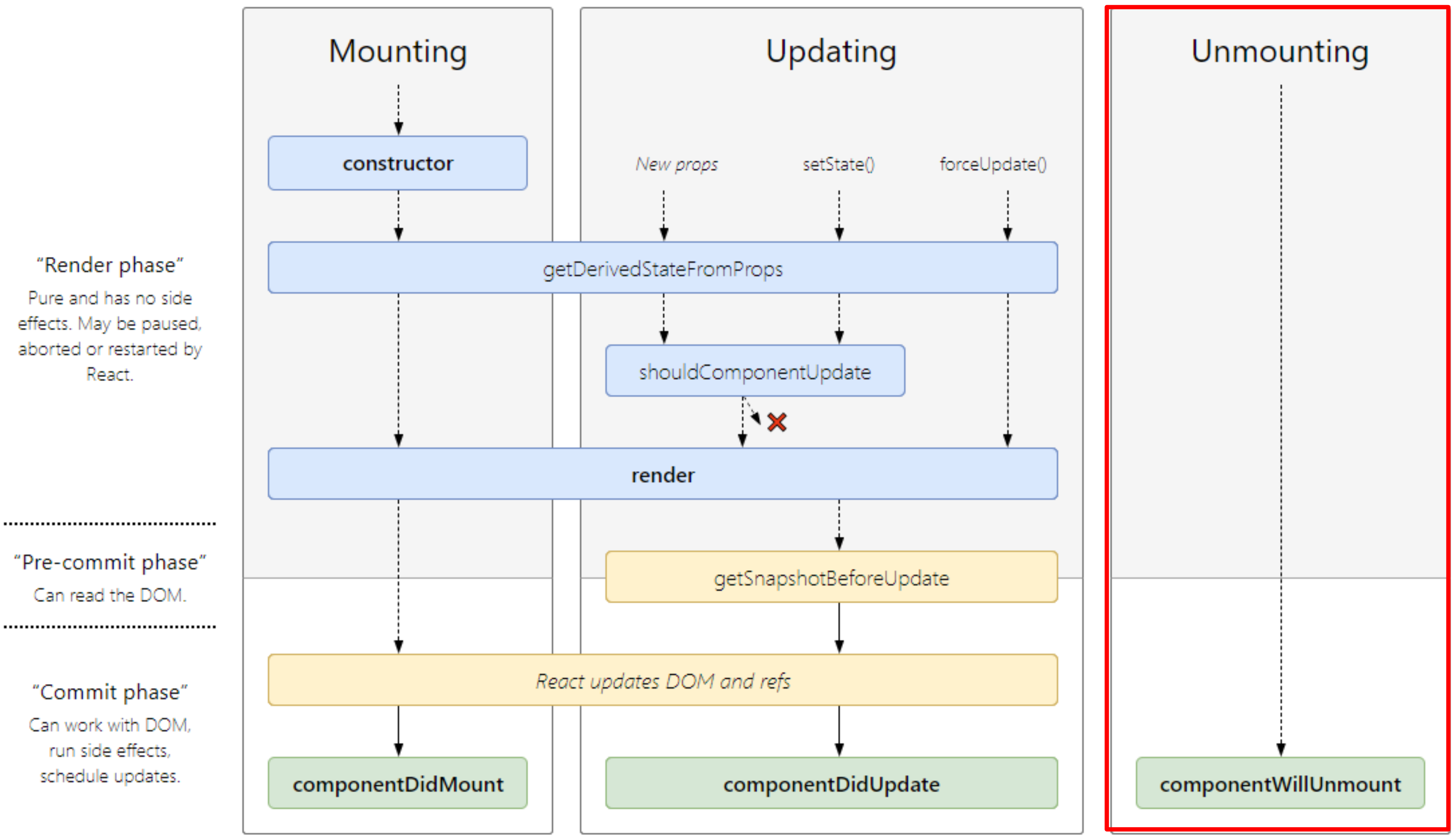
## forceUpdate(): Example

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.user_id = -1;
    this.user_name = "";
  }

  changeUserProfile(new_id, new_name){
    this.user_id = new_id;
    this.user_name = new_name;
    this.forceUpdate();
  }
}
```

**list.js**

# Unmounting State



## Unmounting State: Method

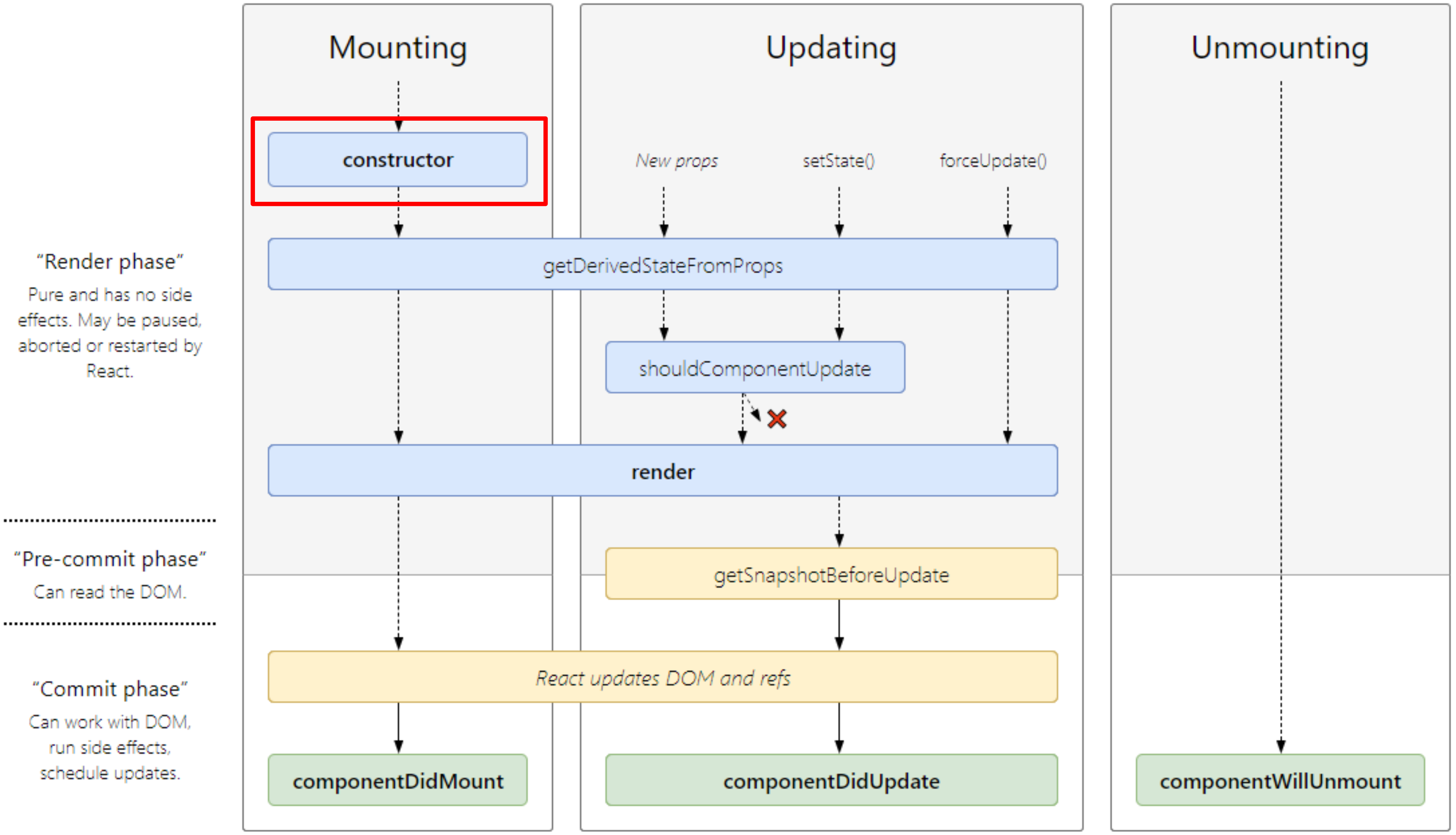
- There is only one method in the unmounting state.
- This method is called when a component is being removed from the DOM
    1. componentWillUnmount()

# Unmounting State: Example

```
var isShow = false;
class Home extends React.Component {
  render() {
    return (
      <div>
        <Example />
        {
              isShow == true ?<Example /> : null
             // Enter unmounting state when isShow changes from true to false
             // Enter mounting state when isShow changes from false to true
        }
      </div>
    );
  }
  triggerExample(){
    isShow = !isShow;
    this.forceUpdate();
  }
}
```

**home.js**

# State Method: constructor()



## State Method: constructor()

- Optional method.
- Initialize variables and state, binding functions.
- Important!!!: When implementing a customized constructor, you **MUST** call super(props) before using “**this.props**”. Otherwise, this.props will be undefined.

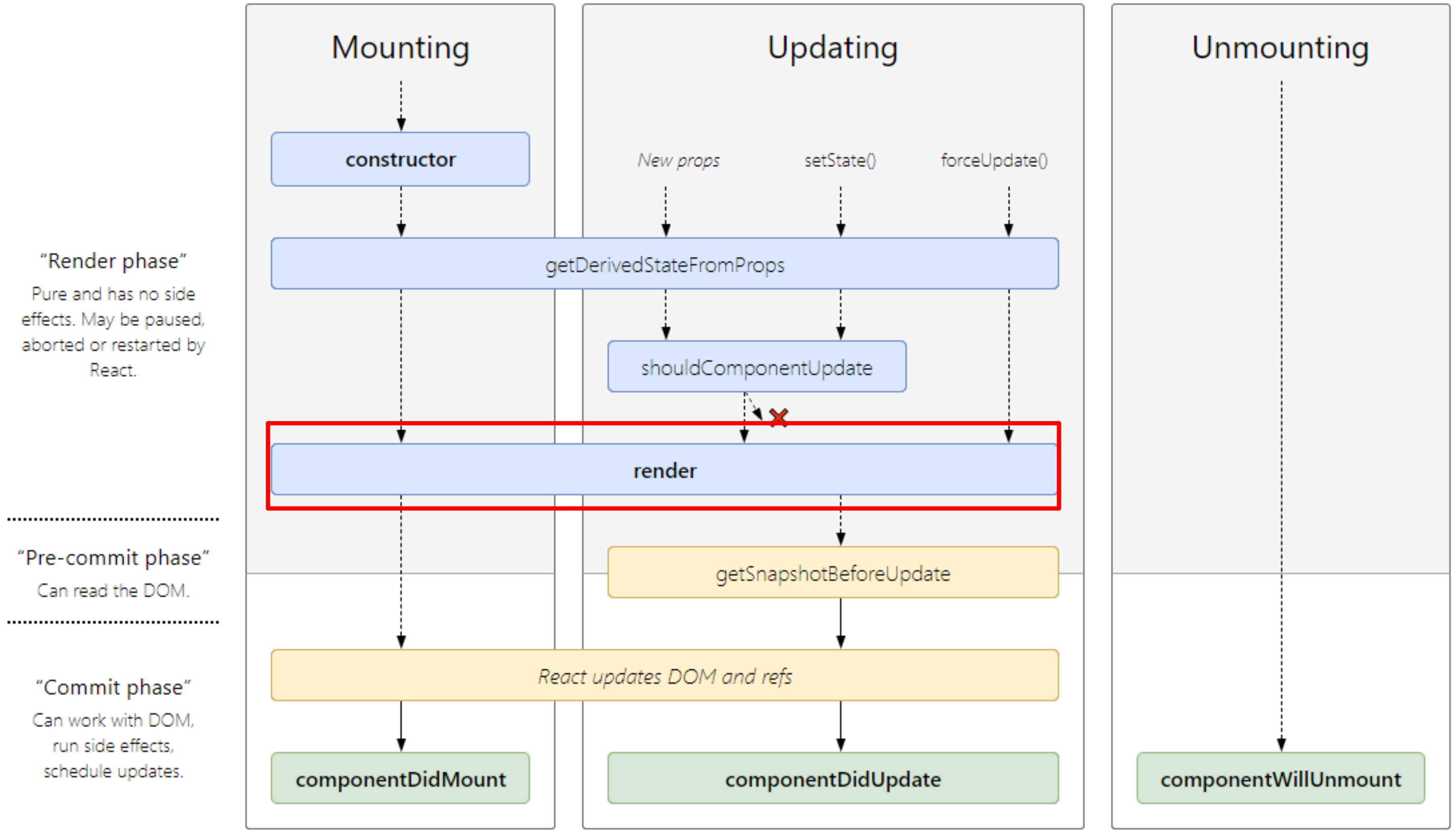
## constructor(): Example

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.user_name = "";   // initialize a variable by default value
    this.user_id = this.props.user_id;  // initialize a variable by props
    this.state = { counter: 0 };        // initialize a state variable
    this.handleClick = this.handleClick.bind(this);  // binding a function
  }

  handleClick(){
    this.user_name = "James";
  }
}
```

**list.js**

# State Method: render()



## State Method: render()

- The only method you **MUST** define in a React.Component subclass.
- This method examines **this.props** and **this.state** and return one of the following types:
  1. **React elements**
  2. Arrays and fragments
  3. Portals
  4. String and numbers
  5. Booleans or null

# render(): React elements

```
class Home extends React.Component {
  render() {
    return (
      <div>
        <Example />    // customized React component
        <Example />
      </div>
    );
  }
}
```

**home.js**

## render(): HTML structure

- You can use **{}** to combine JS codes with HTML.
- This helps us manage the html structure to connect with JS variables or functions.

```
class Example extends React.Component {
  render() {
    return (
        <ul>
            <li>User Profile</li>
            <li>{this.user_id}</li>    // use {} to insert variable into HTML
            <li>{this.user_name}</li>
        </ul>
    );
  }
}
```

**list.js**

## render(): Single Node

- The following codes is **forbidden** in render().
- **You can only return one DOM node in a React component.**

```
class Example extends React.Component {
  render() {
    return (
      <li>User Profile</li>          // first node
      <li>{this.user_id}</li>        // second node
    );
  }
}
```

**forbidden**

**list.js**

## render(): Single Node

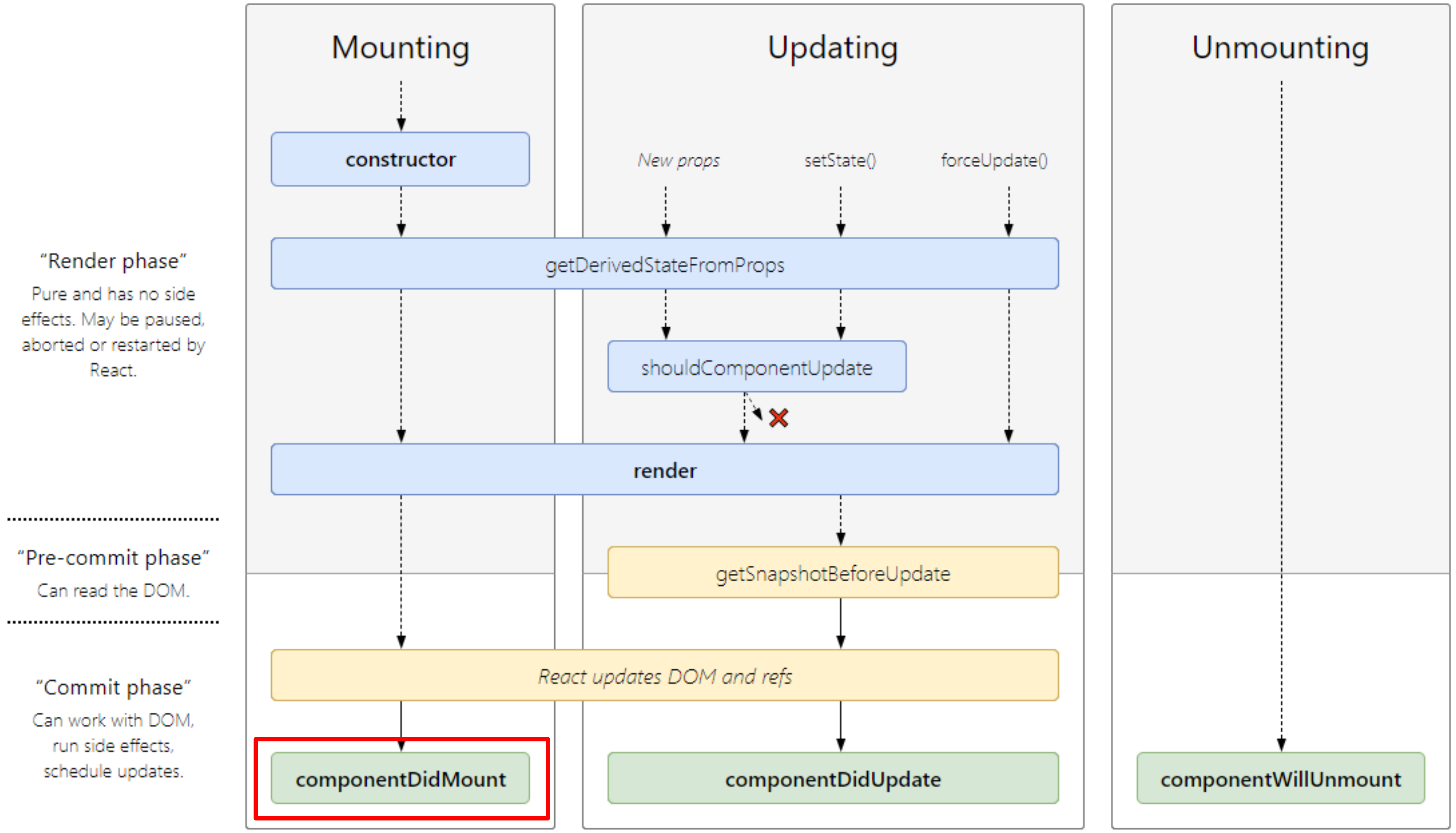
- Same as we return components in {}.

```
class Example extends React.Component {
  render() {
    return (
      <ul>
      {
            return(
              <li>User Profile</li>   // first node
              <li>{this.user_id}</li> // second node
            );
      }
      </ul>
    );
  }
}
```

forbidden

list.js

# componentDidMount()



## State Method: componentDidMount()

- It is invoked immediately after a component is mounted (i.e., inserted into the DOM tree).
- Initialization of **DOM nodes** that are created during the render method.
- A good place to perform side-effects such as **instantiate the network request** (e.g., loading an image).

## Link JS to HTML elements

- Sometimes we want to distinguish or get the React components and the HTML elements we created.

```
class Home extends React.Component {
  render() {
    return (
      <div>
        <Example /> // how can we do different things in two Example components?
        <Example />
      </div>
    );
  }
}
```

**home.js**

## Link JS to HTML elements: ref

- Using **ref** can help us get the components we created.

```
class Home extends React.Component {
  constructor(props){
    this.example_1 = null;
    this.example_2 = null;
  }
  render() {
    return (
      <div>
        <Example ref={(myRef) =>{this.example_1 = myRef}} />
        <Example ref={(myRef) =>{this.example_2 = myRef}} />
      </div>
    );
  }
  componentDidMount(){
     // now you can use this.example_1 and this.example_2 to control Example separately
  }
}
```

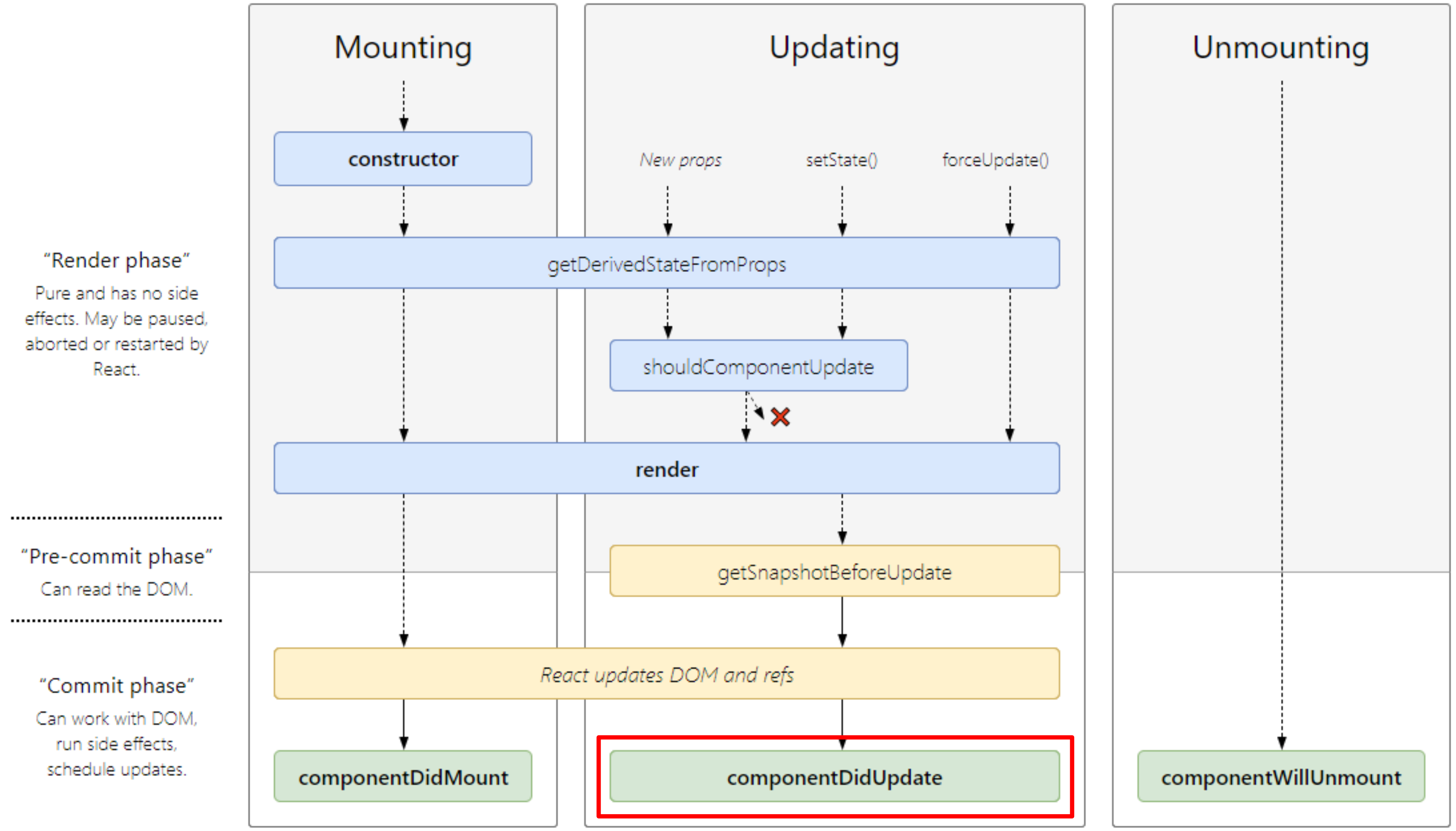
**home.js**

# componentDidMount(): Example

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.block = null;            // variable to record React component element
  }
  render() {
    return (
      <div ref={(curRef) => {this.block = curRef;}}>No data</div>
      // link element to our variable
    );
  }
  componentDidMount() {
    this.GetDataFromDatabase().then((data)=>{
       this.block.innerHTML = data;   // use variable to access the HTML element
    })
  }
}
```

**list.js**

# componentDidUpdate()



# State Method: componentDidUpdate()

- Operating on **DOM nodes** that are updated during the render method.
- The method can take **none** parameter or two parameters “**prevProps**” and “**prevState**”, which are used to compare the current props and state with those before updating.
- A good place to perform side-effects such as **instantiate the network request** (e.g., loading an image).

# componentDidUpdate(): Example

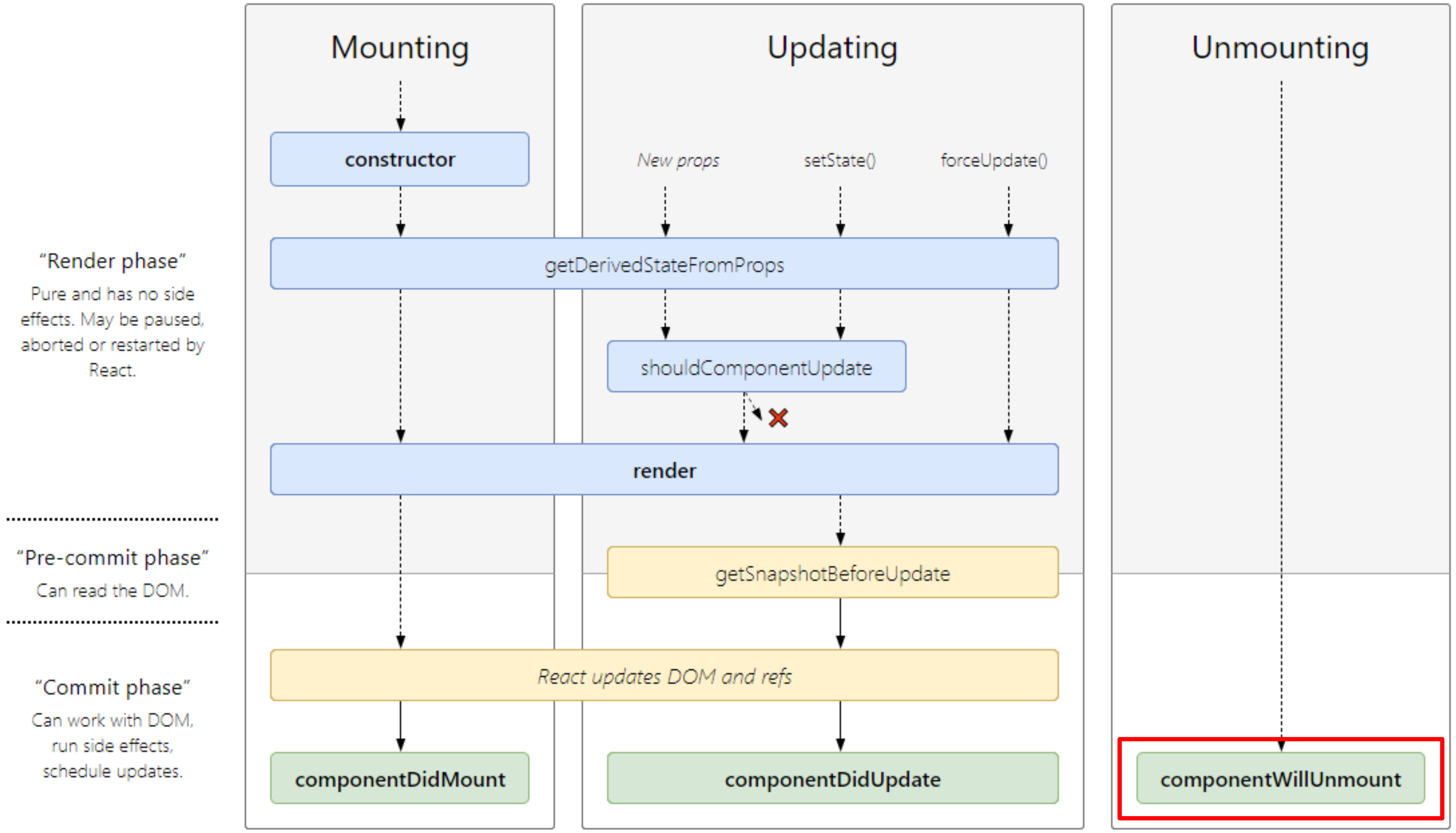
```
class Example extends React.Component {
  componentDidUpdate() {
      this.fetchData(this.props.userID);
  }
}
```

**list.js**

```
class Example extends React.Component {
  componentDidUpdate(prevProps, prevState) {
    // prevProps is the props variable before Update
    // prevState is the state variable before Update
    if (this.props.userID !== prevProps.userID) {
      this.fetchData(this.props.userID);
    }
  }
}
```

**list.js**

# componentWillUnmount()

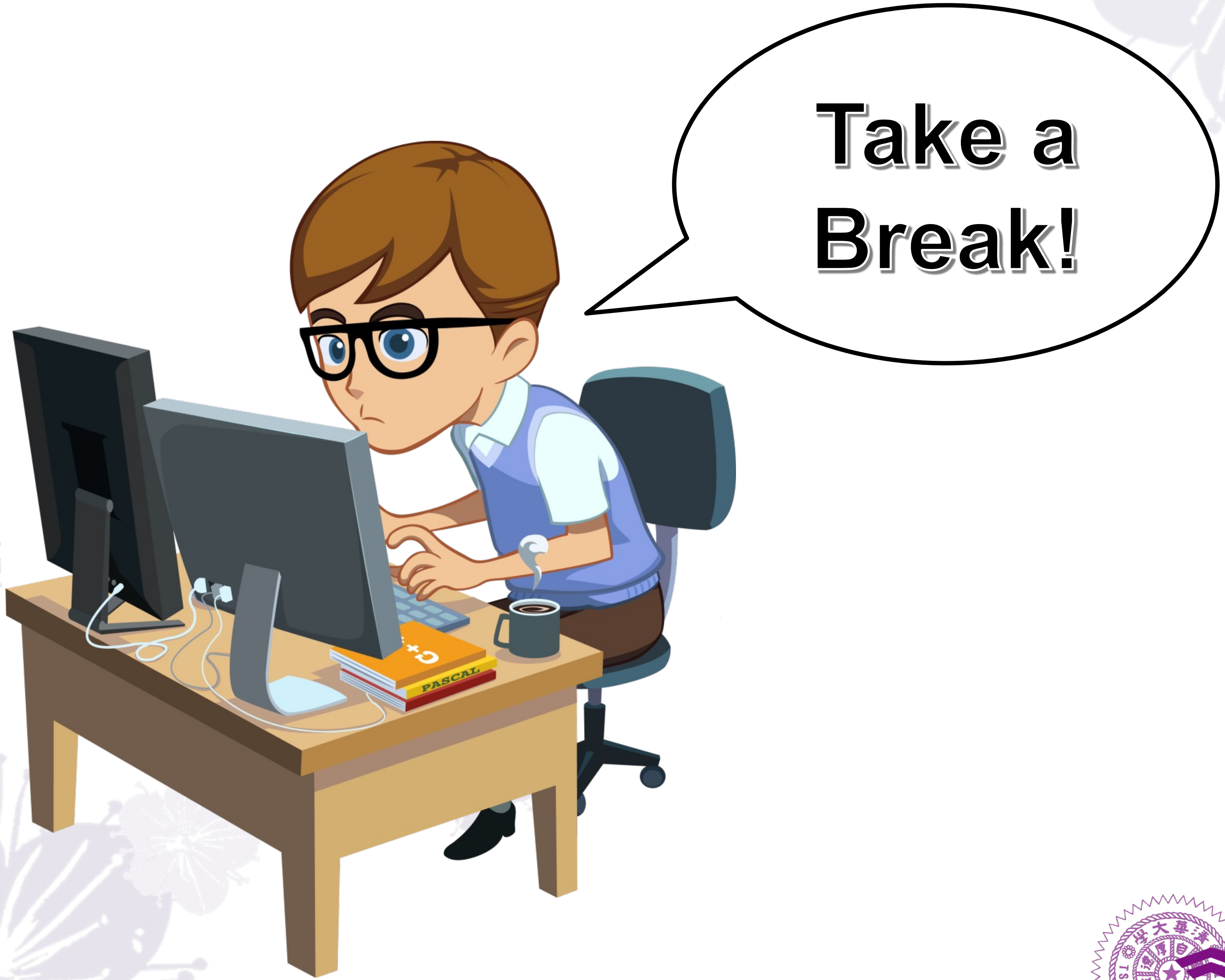


## State Method: componentWillUnmount()

- It is invoked immediately **before** a component is unmounted and destroyed.
- Perform any necessary **cleanup** in this method, such as invalidating timers, canceling network requests, or cleaning up

```
class Example extends React.Component {
  componentWillUnmount() {
    this.saveData(this.user_id, this.user_name);  // save data to database
    this.user_id = null; // clean inner variable
  }
}
```

**list.js**

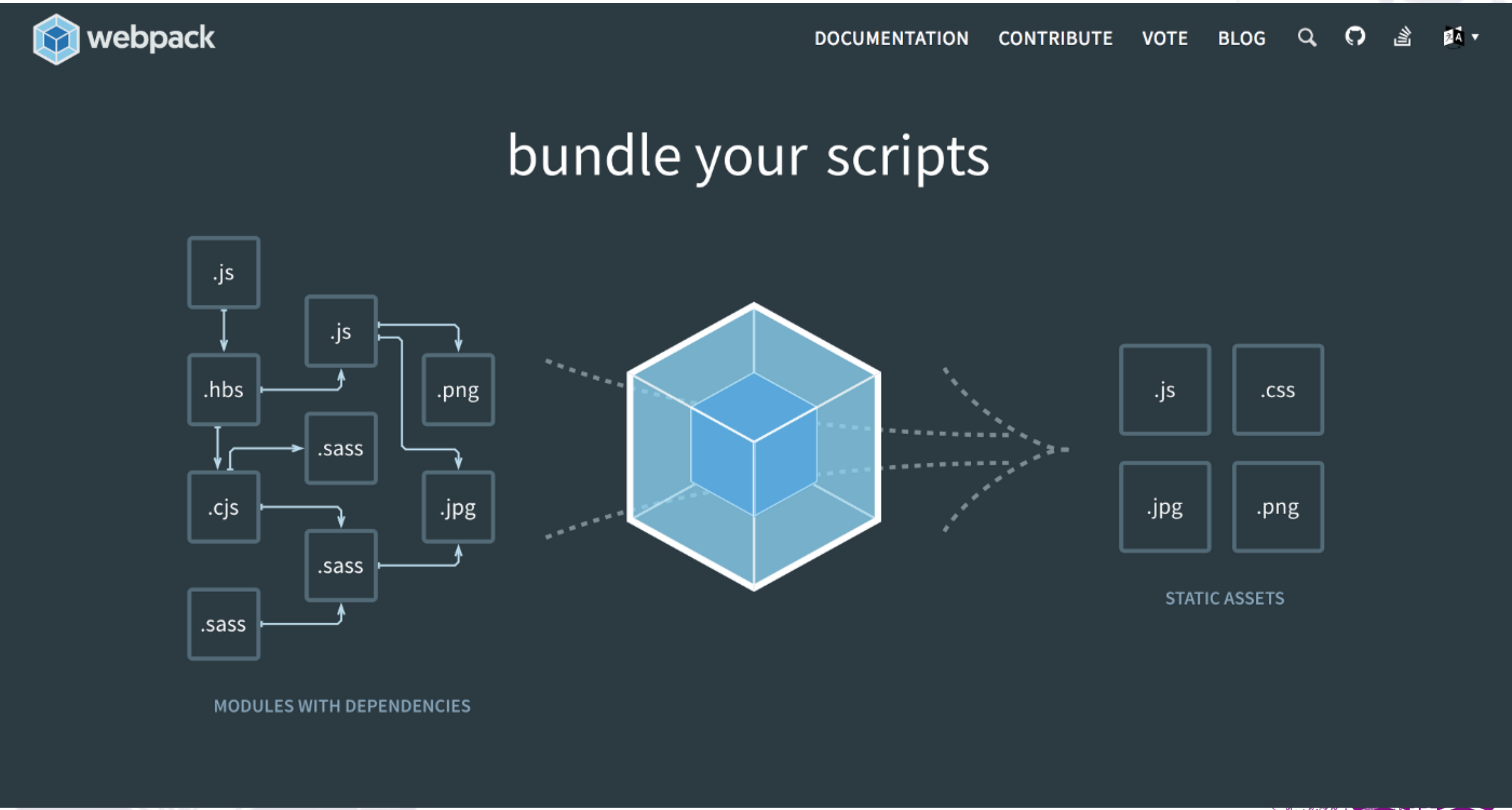

Take a Break!
C++
PASCAL

webpack

## Introduction

- Webpack is a static module bundler for modern JavaScript applications.
- When webpack processes your application, it internally builds a dependency graph which maps every module used in your project and generates one or more bundles.

# Webpack Concept



## Why we use Webpack?

- There are many advantages of Webpack.
  1. Bundle js files into one file
  2. Use npm (Node Package Manager) packages in front-end website (e.g., React, Firebase)
  3. Support HMR (Hot Module Replacement)

## Hot Module Replacement

- Hot Module Replacement can exchange, add, or remove modules on the fly without reloading the application.
- Hence, we can simultaneously edit the codes and run testing server at the same time.

## ENVIRONMENT SETTING

## Project Initialization

- Install npm via Node.js
- Create an empty folder
- Open the terminal, go to the project folder and run the following command

```
npm init
```

- Follow the instructions to create a **package.json** file in the folder.

## Packages Installation

- **ReactDOM** is used in the top-level component of your application.
- It is responsible for linking React model and html, allowing the DOM generated by React to be rendered in html.
- Run the following command in terminal.

```
npm install --save-dev react react-dom
```

弄懂 npm install 的 –save 與 –save-dev

## Packages Installation (Cont'd)

- Install Webpack and plugins "**webpack-cli**" and "**webpack-dev-server**" as follows:

```
npm install --save-dev webpack webpack-cli webpack-dev-server
```

- These plugins are used to build up a local testing server.

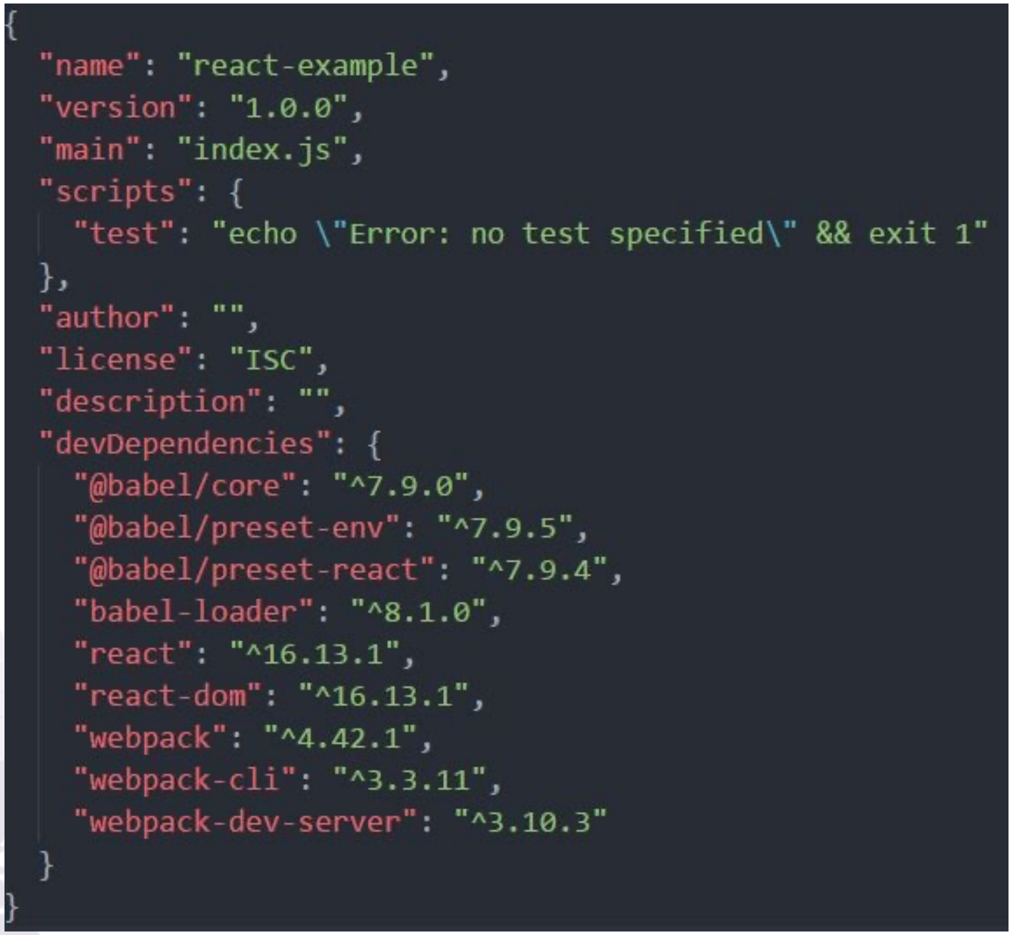
## Packages Installation (Cont'd)

- Since we will use .js file to write our code, we need to install packages for Webpack to do transpiling (i.e., compilers to JS and React) as follows:

```
npm install --save-dev @babel/core babel-loader
@babel/preset-env @babel/preset-react
```

# You Are Ready to GO!

- The generated package.json file looks like



```
{
  "name": "react-example",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "@babel/core": "^7.9.0",
    "@babel/preset-env": "^7.9.5",
    "@babel/preset-react": "^7.9.4",
    "babel-loader": "^8.1.0",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "webpack": "^4.42.1",
    "webpack-cli": "^3.3.11",
    "webpack-dev-server": "^3.10.3"
  }
}
```

## TUTORIAL #1
## HELLO WORLD!

## Hello World!

- Let’s start from an easy example.
- We are going to build up a Webpack local testing server that can show text “Hello world!” in browser.

## Create a React Component

- First, create a **index.js** file in the root folder.
- Then, create a class as follows.

```
export class Example extends React.Component {
  render() {
    return (<div><h1>Hello world! </h1></div>);
  }
}
```

**index.js**

## Create a HTML File

- Now, create a **index.html** file, the default entry point used in local testing server.
- Add a div label into the html body and set its id to “**example**”.

```
<!DOCTYPE html>

<html>
<head>…</head>

<body>
   <div id="example"></div>
</body>
</html>
```

**index.html**

## Setup Webpack Config File

- Create a **webpack.config.js** file with the content shown in the block below.
- All of setting of Webpack will be set in module.exports block. The details of field will be introduced later.

```
var webpack = require('webpack');

module.exports = {…}
```

**webpack.config.js**

Webpack Concepts

## Block Fields in module.exports

```
var webpack = require('webpack');

module.exports = {
        entry: […],
        output: {…},
        resolve: {…},
        module: {…},
        plugins:[…],
        mode: '…',
        devServer: {…}
};
```

**webpack.config.js**

Webpack Concepts

## Webpack Config: entry

- Webpack will compile and pack every file we enter in entry block. Other files used in the application, such as js, css or image, will be packed too.
- Here is an example of entry field. The value corresponds to the file path to the entry point, **index.js** file, in our case.

**webpack.config.js**

```
entry: ['./index.js'],
```

[Webpack Concepts]

# Webpack Config: output

- Output block defines the output of Webpack.
  - **filename** represents the output file name.
  - **publicPath** specifies the public URL of the output directory when referenced in a browser. The URL is resolved relative to the entry html page (index.html).

```
output: {
        filename: 'compiled.js',
        path: path.resolve(__dirname, 'dist'),
        publicPath: '/'
}
```

**webpack.config.js**

[Webpack Concepts]

## Webpack Config: resolve

- This block changes how modules are resolved. Webpack provides reasonable defaults, but it is possible to change the resolving in detail.
- Now we use .js files and simply add ‘.js’ in extensions field.

```
resolve: {
        extensions: ['.js']
}
```

**webpack.config.js**

[Webpack Concepts]

## Webpack Config: module

- This block determines how different types of modules within a project will be treated.
- An array of **rules** which match to requests when modules are created.
- These rules can modify how the module is created. They can apply loaders to the module or modify the parser.

Webpack Concepts

## Webpack Config: module

- The example rules below deal with only the js file. If you need to handle other type of files, you need to add more rules.

```
module: {
    rules: [{
        test: /\.(js)$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        options: {
            presets: ['@babel/preset-react', '@babel/preset-env']
        },
    }]
}
```

**webpack.config.js**

Webpack Concepts

# Webpack Config: plugins

- Webpack has a rich plugin interface. Most of the features within Webpack itself use this plugin interface.
- This makes webpack flexible.
- Here we use two plugins to help us develop the application.

Webpack Concepts

# Webpack Config: plugins

- **HotModuleReplacementPlugin** will enable Hot Module Replacement.
- **ProvidePlugin** will automatically load modules and link to specified keywords.

```
plugins: [
        new webpack.HotModuleReplacementPlugin(),
        new webpack.ProvidePlugin({
            React: 'react',
            ReactDOM: 'react-dom'
        })]
```

**webpack.config.js**

Webpack Concepts

## Webpack Config: mode

- Three modes: ‘development’, ‘production’, ‘none’
- Tell Webpack to use different mode configuration in its built-in optimizations.
- Default value is ‘production’.

```
mode: 'development' | 'production’ | 'none’
```

**webpack.config.js**

Webpack Concepts

## Webpack Config: devServer

- This block field defines the parameters required by the **webpack-dev-server.**

```
devServer: {
        static: './',
        hot: true,
        compress: true,
        host: 'localhost',
        port: 8080
}
```

**webpack.config.js**

Webpack Concepts

## Webpack Config: result

- The final webpack.config.js

# Link HTML and Component

- Call **ReactDOM.render()** in index.js to render a React component (root) on the specific HTML element.

```
ReactDOM.render(<Example />,
document.getElementById("example"));
```

**index.js**

## Link HTML and Component

- In the index.html, we add script element to link the compiled JS generated by the Webpack.
- The filename and path MUST consist to the parameters resolved in the module.exports.output block field

```
<body>
  <div id="example"></div>
  <script src="./compiled.js"></script>
</body>
```

**index.html**

## Running Local Testing Server

- Now, we have finished setting the project. We can use the plugin to run a local testing server.
- Add script in package.json as follows.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "serve": "webpack serve"
},
```

**package.json**

# Running Local Testing Server

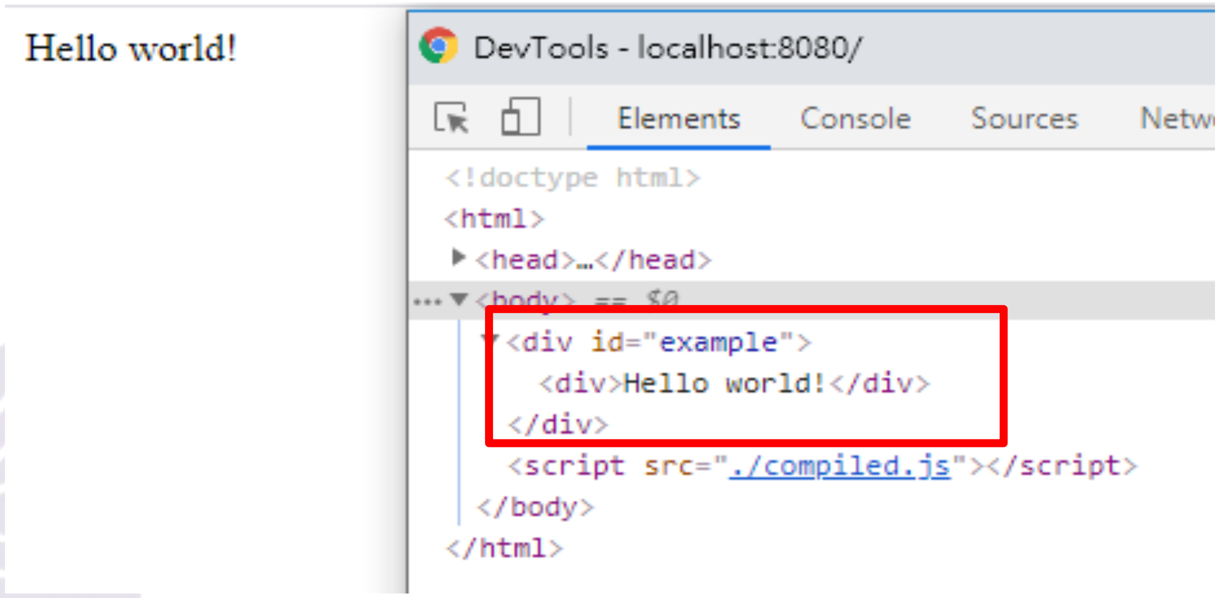
- Run the following command to start the local testing server

```
npm run serve
```

- Thanks to the HMR, every time we edit and save the script in text edtior, webpack-dev-server will automatically reopen the server and refresh the applicaiton.

## Hello world!

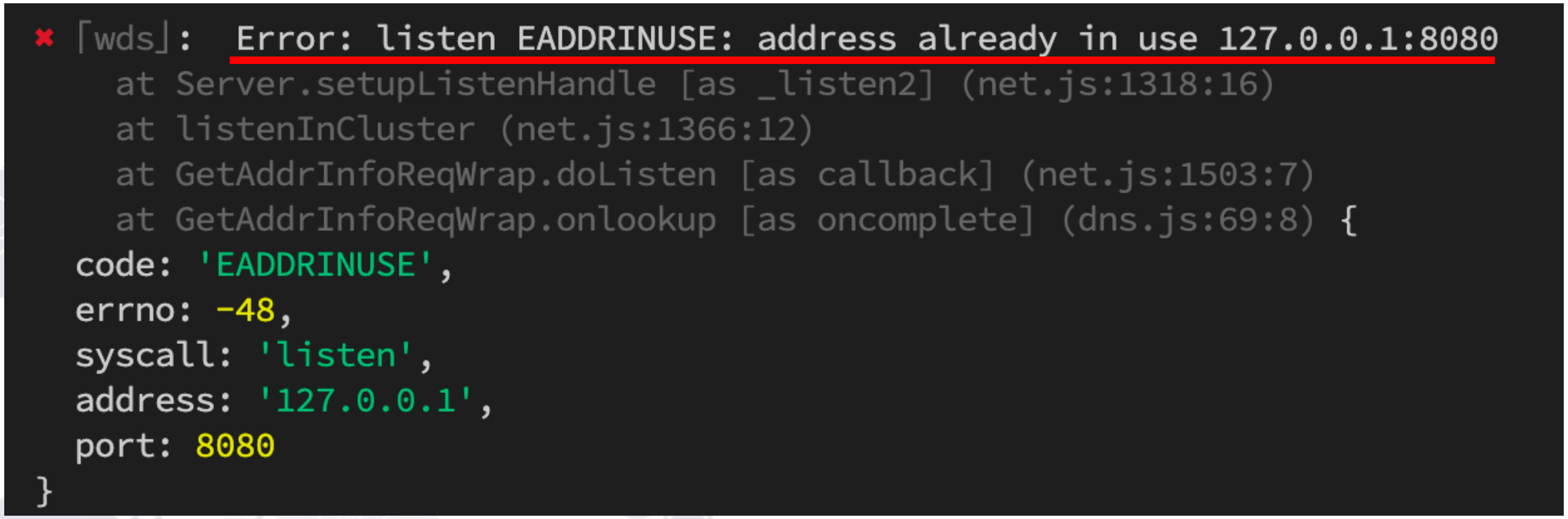
- To see the result, visit the following link http://localhost:8080/
- Open DevTools to see how ReactDOM renders the customized component.



```
<!doctype html>
<html>
  ▶<head>…</head>
…▼<body> == $0
    ▼<div id="example">
       <div>Hello world!</div>
     </div>
     <script src="./compiled.js"></script>
   </body>
</html>
```

# Problem

- If you want to run ‘npm run serve’ again but…



```
✖ ｢wds｣:  Error: listen EADDRINUSE: address already in use 127.0.0.1:8080
    at Server.setupListenHandle [as _listen2] (net.js:1318:16)
    at listenInCluster (net.js:1366:12)
    at GetAddrInfoReqWrap.doListen [as callback] (net.js:1503:7)
    at GetAddrInfoReqWrap.onlookup [as oncomplete] (dns.js:69:8) {
  code: 'EADDRINUSE',
  errno: -48,
  syscall: 'listen',
  address: '127.0.0.1',
  port: 8080
}
```

# Solution

- Windows:

```
netstat -aon|findstr "8080"
taskkill /t /f /pid [PID]
```

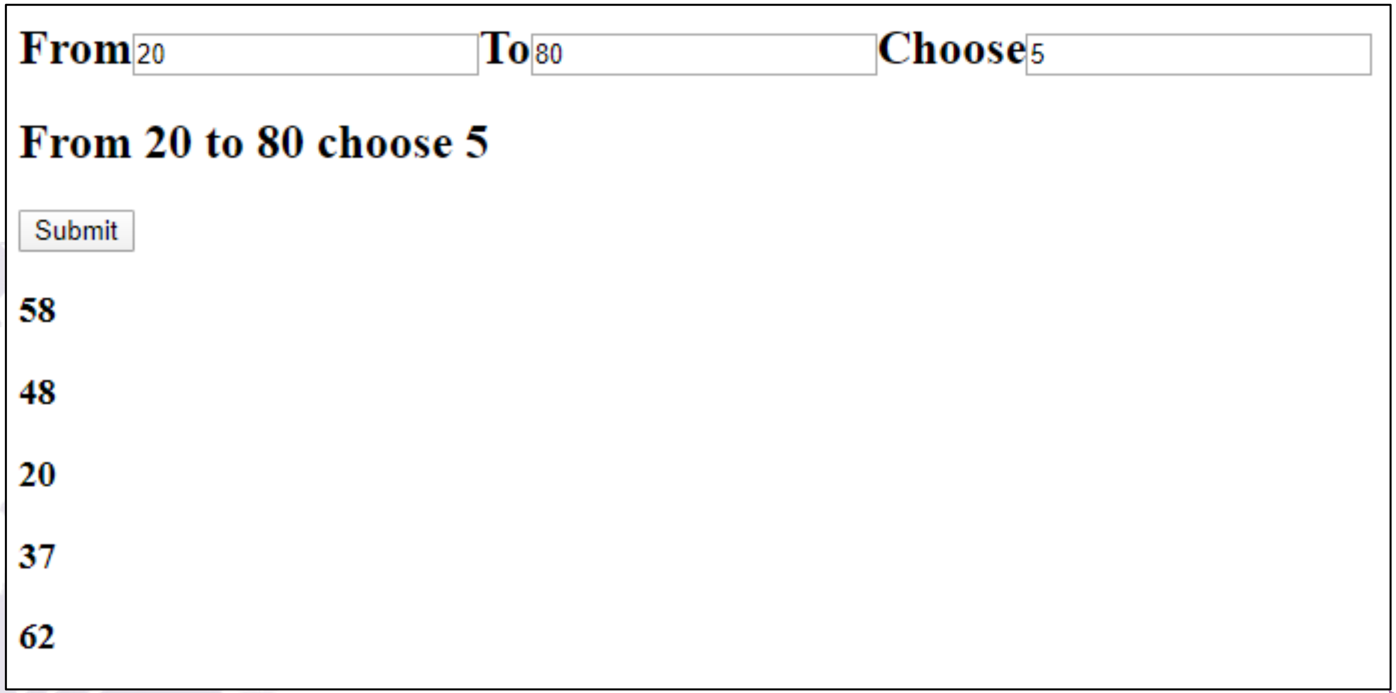
- MacOS:

```
sudo lsof -i: 8080
sudo kill -9 [PID]
```

## TUTORIAL #2
## RANDOM NUMBER GENERATOR

## Random Number Generator

- Next, we are going to extend our project to a random number generator.



From 20 To 80 Choose 5

**From 20 to 80 choose 5**

**58**

**48**

**20**

**37**

**62**

## Input Control: Variables

- Add a constructor() in **Example** class.
- Use ‘this.state’ to initialize state variable.
- Add a variable ‘this.submit’ to check whether the Submit button is clicked or not.

```
constructor(props) {
    super(props);
    this.state = {           // state variable
      min : 0,               // the range of value
      max : 0,
      number : 1             // how many numbers to generate
    }
    this.submit = false; // check status of submit button
}
```

**index.js**

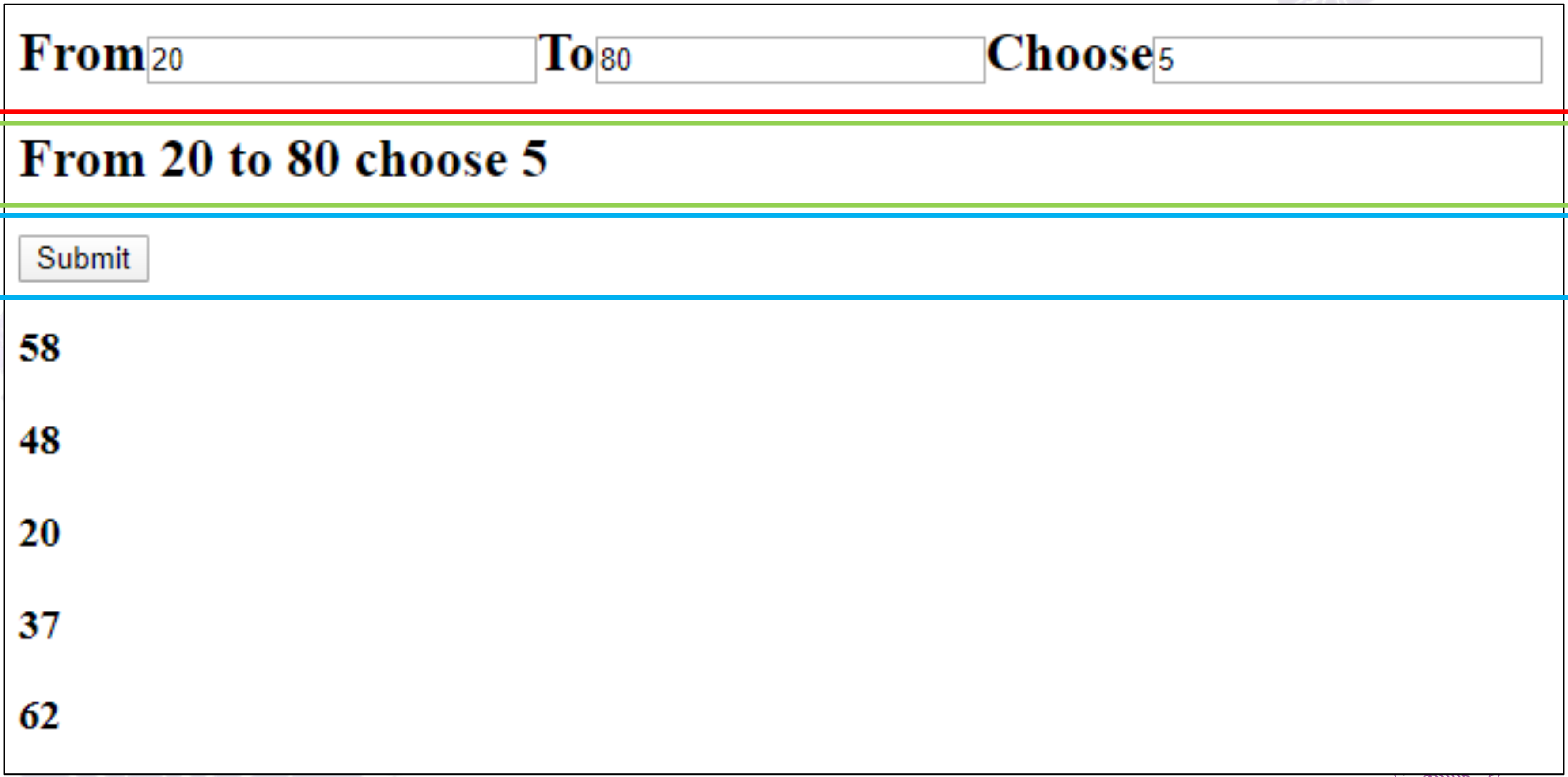
## Input Control: Link with HTML

- Use **{}** to link callback function, **onChange** and **onClick,** with React component

```
render() {
return (
<div>
  <h2>
    From<input type="text" onChange={e => this.stateChange(0, e)} />
    To<input type="text" onChange={e => this.stateChange(1, e)} />
    Choose<input type="text" onChange={e => this.stateChange(2, e)} />
  </h2>
  <h2>
    {"From " + this.state.min + " to " + this.state.max + " choose " +
       this.state.number}
  </h2>
  <button onClick={() => this.submitValue()}>Submit</button>
</div>
); }
```

**index.js**

# Input Control: Link with HTML


From 20 To 80 Choose 5

From 20 to 80 choose 5

Submit

58

48

20

37

62

# Input Control: Callback Function

- Add a **stateChange** function in **Example** class.
- In this function, we update different state variables by different input index.

```
stateChange(index, e) {            // for text input blocks
  this.submit = false;
  if (e.target.value != undefined) {
    if (index == 0) {
      this.setState({ min: parseInt(e.target.value) });
    } else if (index == 1) {
      this.setState({ max: parseInt(e.target.value) });
    } else {
      this.setState({ number: parseInt(e.target.value) });
    }
  }
}
```

**index.js**

## Input Control: Callback Function

- Add a **submitValue** function in **Example** class.
- In this function, we update this.submit, and use this.forceUpdate() to trigger re-rendering so that we can show our list of random number.

```
submitValue() {        // for submit button
  if (this.state.number > 0) {
    this.submit = true;
    this.forceUpdate();
  }
}
```

**index.js**

## Add a Component: List

- Next, we want to create a child component to
  - generate random numbers; and
  - show the numbers in a list.
- Create a new new **list.js** file in the ‘script’ folder.
- Define a new **List** React component.
- Import component **List** to index.js, so that it can be used in the component **Example**.

**index.js**

```
import { List } from "./script/list";
```

## Component Example: render

- Use this.submit in **Example** to determine whether List component should render or not.

```
render() {
return (
<div>
  ….
  <button onClick={() => this.submitValue()}>Submit</button>
  { this.submit && <List var={this.state} /> }
  // equivalent to {this.submit ? <List var={this.state} /> : null }
</div>
); }
```

**index.js**

## Component List

```
export class List extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    var arr = [];
    for (var i = 1; i <= this.props.var.number; i++) {  // generate random numbers
      arr.push(this.getRandom(this.props.var.min, this.props.var.max));
    }
    return (
      <div>
      {arr.map((value, index)=>{ return (<h3 key={index}>{value}</h3>) })}
      // key is a keyword used for VirtualDOM to tell which DOM node is which
      </div>
    );
  }
  getRandom(min, max) {
            return Math.floor(Math.random() * (max - min + 1)) + min; } }
}
```

```
<List var={this.state} />
```

**list.js**

## Component List: render()

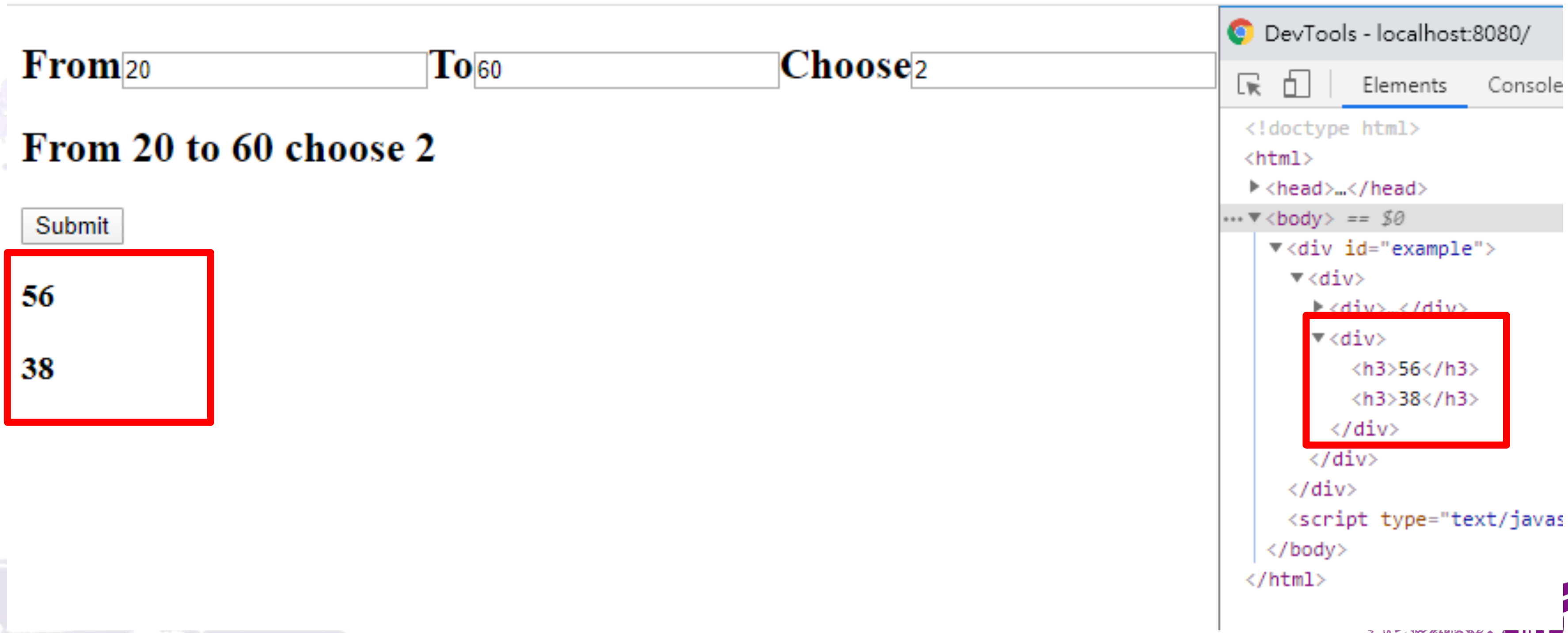
- map() can let us list all the element in array. We use **{}** to create component list.
- **key** is a keyword used for VirtualDOM to tell which DOM node is which

```
render(){
  …
  return (
    <div>
    {arr.map((value, index)=>{ return (<h3 key={index}> {value} </h3>) })}
    </div>
  );
}
```

**list.js**

## Result

- You can go DevTools to check how React deal with the html element we return.



From 20 To 60 Choose 2

**From 20 to 60 choose 2**

Submit

**56**

**38**

DevTools - localhost:8080/

```
<!doctype html>
<html>
 ▶<head>…</head>
…▼<body> == $0
  ▼<div id="example">
   ▼<div>
    ▶<div>…</div>
    ▼<div>
       <h3>56</h3>
       <h3>38</h3>
     </div>
    </div>
   </div>
   <script type="text/javas
  </body>
</html>
```

## ADVANCED TECHNIQUES

1. Use CSS in application
2. Bundle files & Deploy

## Use CSS in Application

- Like .js files, we must import compiler to webpack to let it pack .css files. Here we using two packages as follows.
- **style-loader** is used to add CSS to the DOM by injecting a **<style>** tag
- **css-loader** interprets **@import** and **url()** like **import/require()** and resolves them.

```
npm install --save-dev css-loader style-loader
```

## Use CSS in Application

- To interpret css codes, modify the Webpack configuration as follows:

```
module: {
rules: [ {…}, // module for js file
    {
      test: /\.css$/,
      use: [
        'style-loader',  // execute second (order is important)
        'css-loader '    // execute first
      ]
    }
}
```

**webpack.config.js**

## Use CSS in application

- Create a .css file and import it into your component.
- Note that once you import the css file into a component, all its child components can use it too.

```css
h2 {
    color: green;
}
```

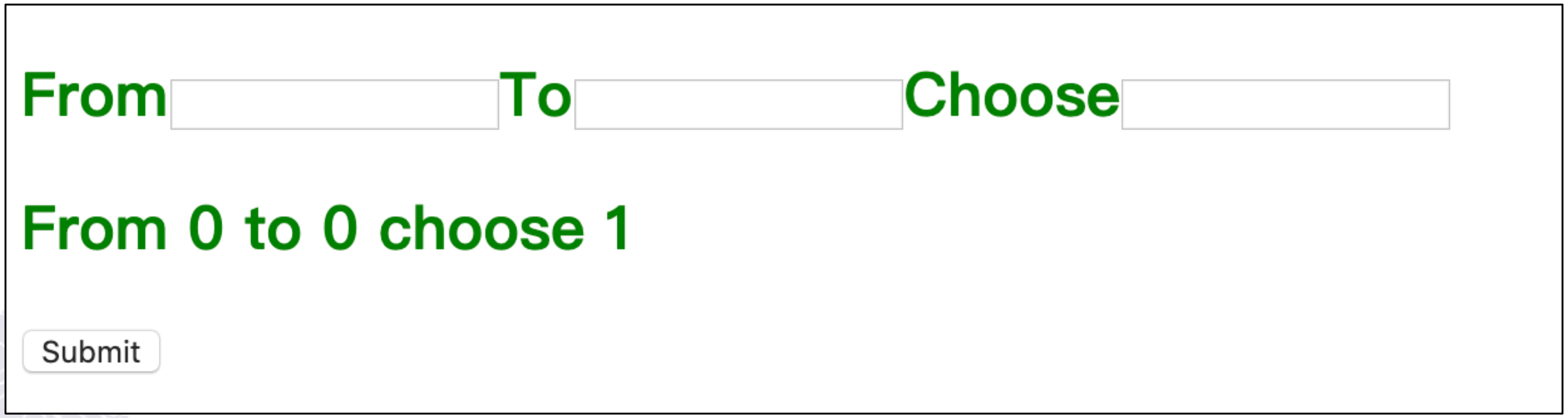
**example.css**

```js
import "./css/example.css"
```

**index.js**

## Use CSS in Application

- There we go!



From To Choose

From 0 to 0 choose 1

## Bundle Files

- Here we show how to bundle project files for deploying to other services such as GitLab or Firebase.
- First, add a new command in package.json.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "serve": "webpack serve”,
  "build": "webpack"
},
```

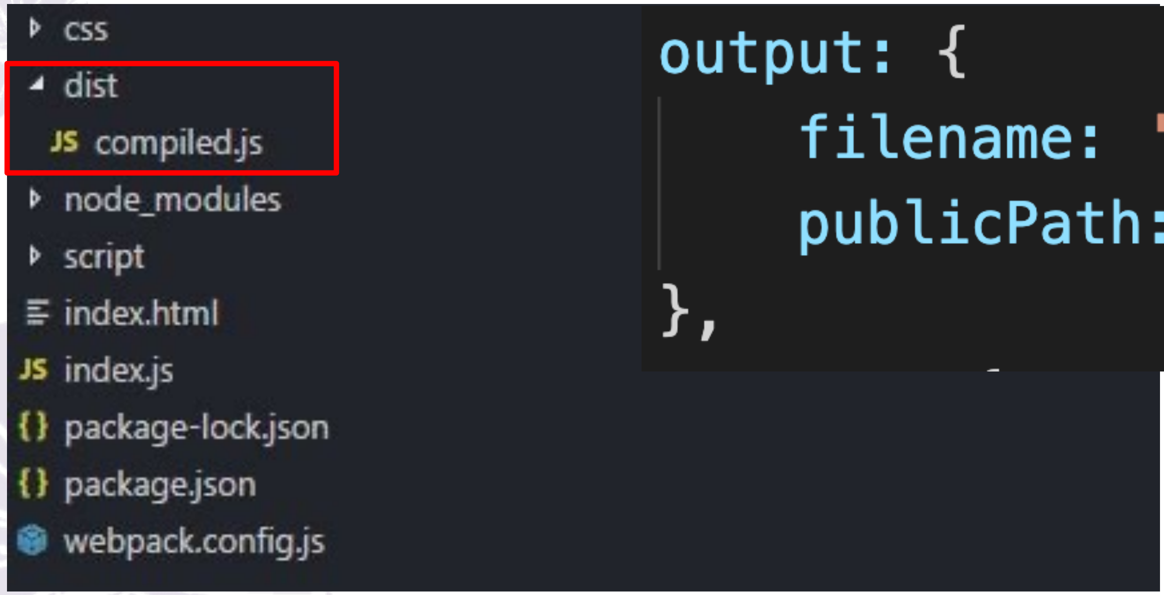
**package.json**

## Bundle Files

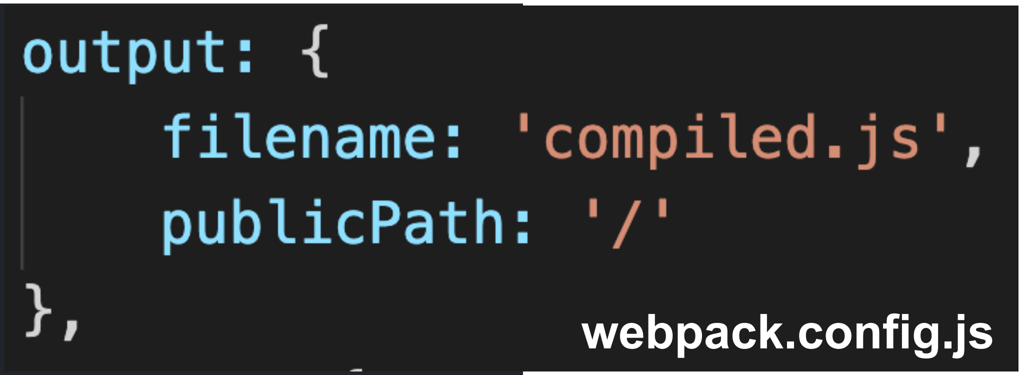
- Run the following command in the terminal.

```
npm run build
```

- The bundle file (**compiled.js**) can be found in the “**dist**” folder.



```
▸ css
▾ dist
    compiled.js
▸ node_modules
▸ script
  index.html
  index.js
  package-lock.json
  package.json
  webpack.config.js
```

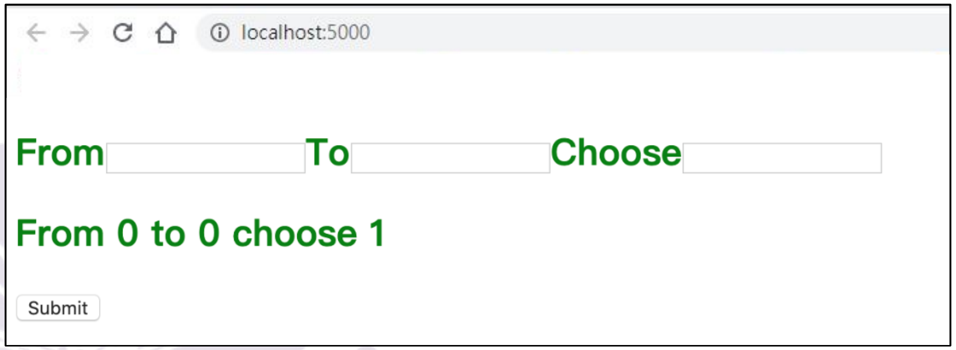


```
output: {
    filename: 'compiled.js',
    publicPath: '/'
},
```

webpack.config.js

## Deploy to Firebase

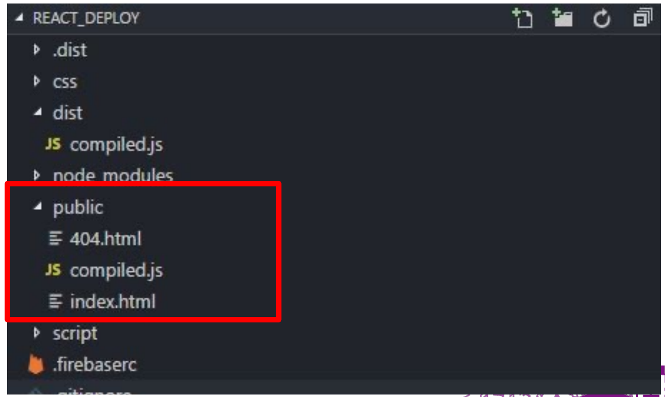
- Run **firebase init** first, copy **index.html** and **compiled.js** to the **public** folder.
- Follow the steps we taught in the **Firebase Hosting** lecture.
- Run your React application on the localhost server or firebase server.



From To Choose

From 0 to 0 choose 1

Submit



## Reference

- React official documentation
- Webpack official documentation
- 猴子也能看懂的 React 教學
- react入門篇

thank you!

Question?