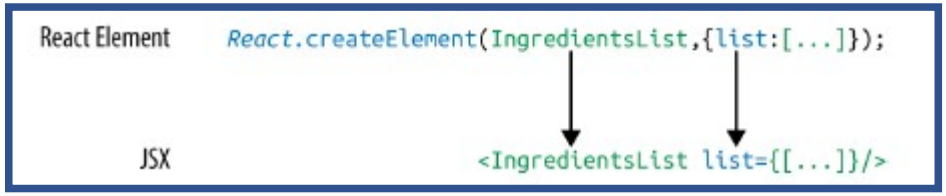
Javascript Syntax Extension (JSX)

- Alternative to *React.createElement(...)*;
- Javascript extension that allows to define React elements using syntax that looks similar to XML/HTML;
- Idea: to provide a concise (and more readable) syntax for creating complex DOM trees with attributes;
- In JSX, an element's type is specified with a tag. The tag's attributes represent the properties. The element's children (or nested components) can be added between the opening and closing tags:

```
class Recipe extends React.Component {
  render() {
    return (<section id={this.props.id} >
        <h1 id="recipe-0" data-type="title">{this.props.name}</h1>
        <IngredientsList items= {this.props.items} ></IngredientsList>
        <Instructions instructions= {this.props.instructions}></Instructions>
      </section>);
  }
}
```

JSX

- JSX also works with components. Simply define the component using the class name.
- You can also pass parameters as props using the element attributes.



- Props will take two types: either a **string** or a **Javascript expression** which can include arrays, objects or functions.
- *Javascript expression* - piece of code that are to be interpreted as JS – are surounded by curly braces **{ }**.

JSX

- JSX is actually closer to Javascript, not HTML, so there are a few key differences to note when writing it:
  - class is a reserved keyword in Javascript:
    use **className instead of class** class attribute;
  - **Properties and methods** in JSX **are camelCase**:
    onclick will become onClick;
  - Self-closing tags must **end in a slash**:
    e.g. <img />
  - **Javascript expressions** - such as variables, functions or properties - can also be **embedded** inside JSX **using curly braces**:
    const **name** = 'Tania'
    const heading = <h1>Hello, **{name}**</h1>

JSX

- All Javascript expressions will get evaluated: operations - such as concatenation or addition - will occur and functions will be invoked:

```
<ul>
    {this.props.ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient}</li>
    )}
</ul>
```

Tutorial React13

- Step 1: Copy react_example_01-HelloReact to **react_example_13-HelloJSX** and open in VSCode.
- Step 2: Inside the render() **return a React Element**. Use the JSX to build the React Element.

```
<script type="text/babel">
  class Hello extends React.Component {
    render() {
      return <h1>Hello React!</h1>
    }
  }

</script>
```

The element content

Create and return a new React element of the given type.

Tutorial React13

- Step 3: Finally, use the **React DOM render()** method to render the Hello Element into the root div in the HTML.

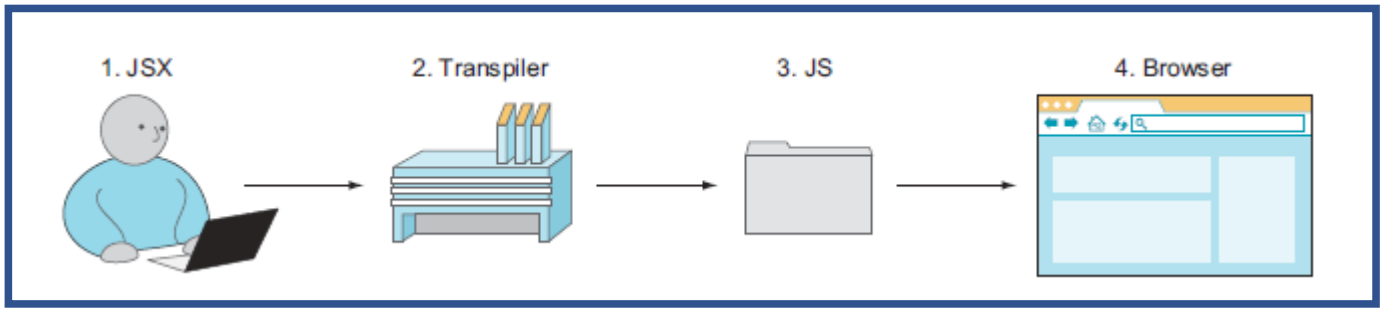
```
<script type="text/babel">
  class Hello extends React.Component {
    render() {
      return <h1>Hello React!</h1>
    }
  }
  ReactDOM.createRoot(<Hello />, document.getElementById('root'));
</script>
```

container

The React Element

JSX

- JSX isn't regular Javascript, and can't be interpreted by the browser.
- You might have noticed that in the previous example we ran JSX through Babel.
- To execute JSX, **we need to convert it** to regular (vanilla) Javascript code. This process is called **transpilation** (from compilation and transformation), and various tools are available to do the job.



Babel

- JavaScript is an interpreted language: the browser interprets the code as text. So… no need to compile!
- However not all browsers support the latest syntax of ecmascript, and **no browser supports JSX**.
- We need a way to convert the source code (latest features and jsx) into something that the browser can interpret. This process is called **transpilation**.
- It is what Babel is designed to do.

Babel

- Some recommended ways to use Babel:
  - Node.js or browser JavaScript script (API approach)

    A script can import the babelcore package and transpile JSX programmatically (babel.transform). This allows for low-level control and removes abstractions and dependencies on the build tools and their plug-ins.
  - Babel command-line interface (CLI) tool

    The babel-cli package provides a command for transpilation. This approach requires less setup and is the easiest to start.
  - Build tool

    A tool such as Grunt, Gulp, or Webpack can use the Babel plug-in. This is the most popular approach.

ESLint

- Lint, linter or linting is the **process** of running a program that will **analyse code for potential programming errors, suspicious constructions and known bugs**.
- The name comes from an 1978 debugging application (Lint) created by Stephen C. Johnson from Bell Labs.
- ESLint is a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code, with the goal of making code more consistent and avoiding bugs.
- Babel-ESLint is a parser that allows ESLint to run on source code that is transformed by Babel (thus supports all recent ECMAScript features).

Tutorial React14

- Step 1: Checkout Github project

```
> git clone https://github.com/exemploTrabalho/react-tutorial.git
```

...and open **react_example_14-BabelCli** in VSCode.

- This is a project that was created with *npm init* and *npx gitignore node* with mostly default options set.
- The html file is already built on *public/index.html*

Tutorial React14

- Step 2: Add to file *src/like_button.js:*

```
class LikeButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { liked: false };
  }

  render() {
    if (this.state.liked) {
      return "You liked comment number " + this.props.commentID;
    }
    return <button onClick={() => this.setState({ liked: true })}>Like</button>;
  }
}
```

We are not using Proptypes yet… this will raise an warning with ESLint.

```
static propTypes = {
  commentID: PropTypes.number.isRequired,
};
```

Tutorial React14

- Step 3: Add to file *src/script.js*:

```
// Find all DOM containers, and render Like buttons into them.
document.querySelectorAll(".like_button_container").forEach((domContainer) => {
  // Read the comment ID from a data-* attribute.
  const commentID = parseInt(domContainer.dataset.commentid);
  ReactDOM.createRoot(<LikeButton commentID={commentID} />, domContainer);
});
```

Tutorial React14

- Step 4: Add dev dependencies to @babel/cli, @babel/core, @babel/preset-env and @babel/preset-react.

```
> npm i -D @babel/core @babel/cli @babel/preset-react @babel/preset-env
```

- Step 5: create the file babel.config.js (or *.babelrc*):

```
module.exports = {
    presets:[
        ["@babel/preset-react", {"runtime": "automatic"}]
    ]
}
```

This structure is to see the output in lib folder.
Please change it afterwards to:

```
module.exports = {
    presets:[
        "@babel/preset-env",
        "@babel/preset-react"
    ]
}
```

Tutorial React14

- Step 6: Add dev dependencies for eslint

```
> npm i -D eslint @babel/eslint-parser
```

- Step 7: Set the configuration file for eslint (*.eslintrc.json*)

```
> npx eslint --init
```

```
√ How would you like to use ESLint? · problems
√ What type of modules does your project use? · esm
√ Which framework does your project use? · react
√ Does your project use TypeScript? · No / Yes
√ Where does your code run? · browser
√ What format do you want your config file to be in? · JSON
The config that you've selected requires the following dependencies:

eslint-plugin-react@latest
√ Would you like to install them now with npm? · No / Yes
Installing eslint-plugin-react@latest
```

If you don't do this step you'll need to run:
> npm i -D eslint-plugin-react@latest

  - You may need ESLint Vscode extension

- See aditional ESLint configurations from ReactJS.

Tutorial React14

- Step 8: Read the warnings raised by ESLint and edit *.eslintrc.json* file to avoid those warnings:

```
…

    "parserOptions":{
…
        "requireConfigFile": true
    }
    "parser": "@babel/eslint-parser",
    "rules": {
        "no-undef": "off",
        "no-unused-vars": "off",
        "react/react-in-jsx-scope": "off",
        "react/jsx-no-undef": "off"
    }
…
```

Recent version will require:
"plugins": ["react", "jsx"],

And to add eslint-plugin-jsx as dev dependency

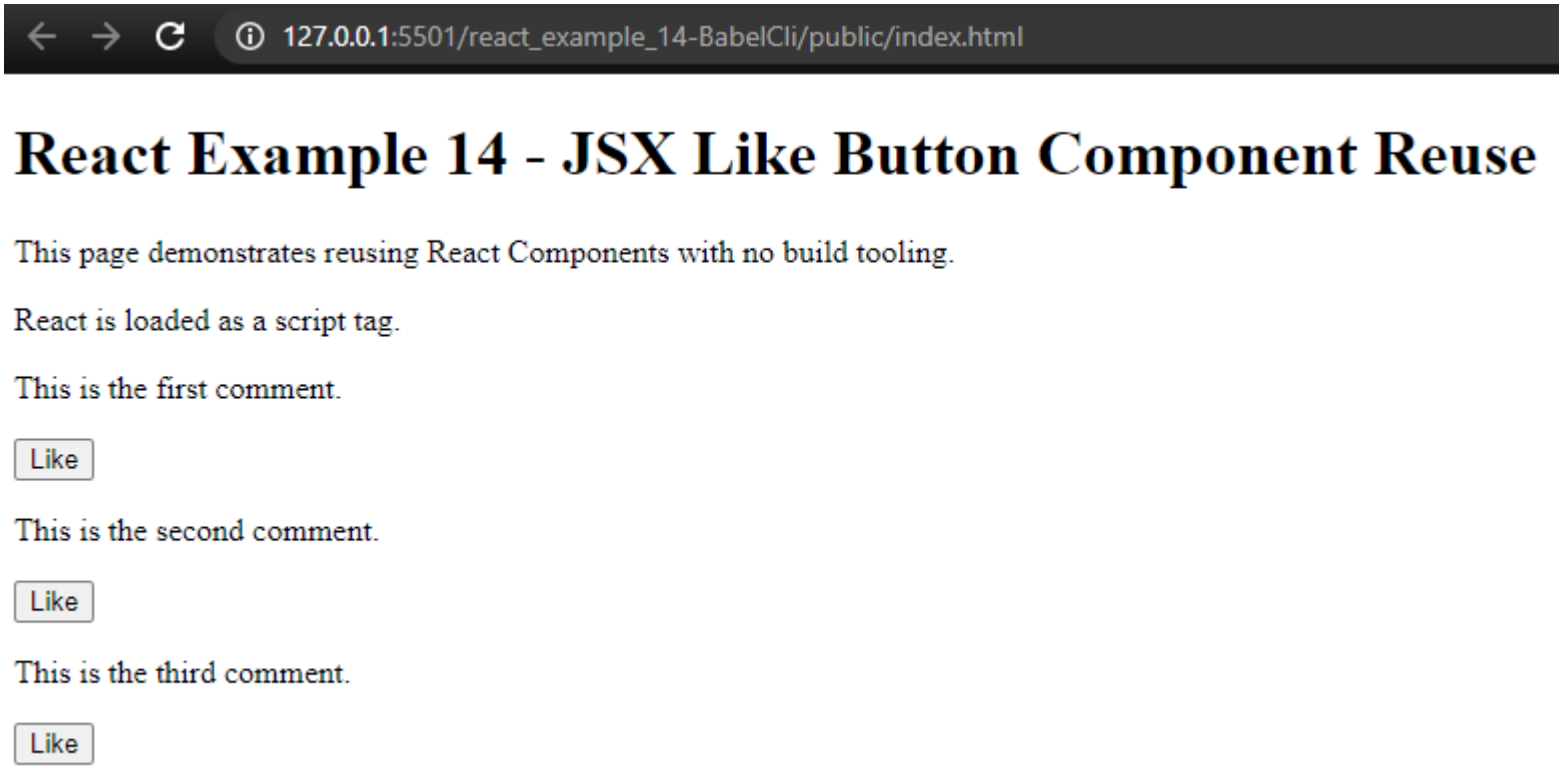
- Step 9: Run the transpiler

```
> npx babel src --out-dir lib
```

- Step 10: See folder *lib/* contents

Tutorial React14

- Step 11: Open the HTML file in a browser.



Tutorial React14

- Step 12: Alternative to *babel.config.js*, run babel:

```
$ npx babel src --presets @babel/preset-react --out-dir lib
```

- Step 13: Alternative to running transpiler from command line, add a script to *package.json*:

```
  "scripts": {
    "build": "npx babel src --out-dir lib",
```

- and run:

```
$ npm run build
```

- ESLint may also be added to script in *package.json*:

```
  "scripts": {
    "eslint": "eslint **/*.js",
```

Lab React15

- Step 1: Copy react_example_11-PassingState to **react_example_15-JSXPassingState** and open in VSCode.
- Step 2: Convert the project to JSX.

Lab React16

- Step 1: Copy react_example_12-PassingUpState to **react_example_16-JSXPassingUpState** and open in VSCode.
- Step 2: Convert the project to JSX, creating a new file for each component.

Component Composition

- React has a powerful composition model, and its recommend using composition instead of inheritance to reuse code between components.
- See the Composition vs Inheritance page on ReactJs website for examples on using composition using props.children or more complex composition strategies where props.children is not suficient.

Webpack

- At its core, webpack is a static module bundler for modern JavaScript applications. When webpack processes an application, it internally builds a dependency graph which maps every module the project needs and generates one or more bundles.
- Webpack also has modules, and is not the only module bundler out there.
- There are some core concepts to understand to extract more than the default operation, such as Entry, Output, Loaders, Plugins, Mode, Browser Compatibility and Environment.

Tutorial React17

- Step 1: Copy react_example_14-BabelCli to **react_example_17-**WebpackCli and open in VSCode.
- Step 2: Instal *Webpack* and *Webpack-cli* as a Dev dependency

```
> npm i -D webpack webpack-cli
```

- Step 3: We will need *babel-loader* to Webpack to transpile Javascript through Babel:

```
> npm i -D babel-loader
```

- Step 4: We will need React, ReactDOM and PropTypes as a dependency:

```
> npm i react react-dom prop-types
```

Tutorial React17

- Step 5: Webpack can work with default options, but we need to change some in a configuration file for our specific build. create the file *webpack.config.js*:

```
const path = require("path");
module.exports = {

  entry: "./src/script.js",
  Output: { filename: "main.js", path: path.resolve(__dirname, "public") },
  module: {
    rules: [
      {
        test: /\.m?js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
          options: {
            presets: ["@babel/preset-env", "@babel/preset-react"],
          }
        }
      }
    ]
  }
};
```

The module of entry to build the bundle.

The name of the output module and the where it will be created.

Copy the presets in .babelrc.js into the options for babel-loader.

There is a npm package to create an intital webpack config file

Tutorial React17

- Step 6: Since we are creating a bundle, we don't need to load React through a <script> element anymore.

Edit the *public/index.html* file. **Delete the lines**:

```
<script src="https://unpkg.com/react@18.3.1/umd/react.development.js"
crossorigin></script>
<script src="https://unpkg.com/react-dom@18.3.1/umd/react-dom.development.js"
crossorigin></script>
<script src="https://unpkg.com/prop-types@15.6/prop-types.js" crossorigin></script>
...
<script src="../lib/like_button.js"></script>
<script src="../lib/script.js"></script>
```

And **move the file** to *src/index.html*

Tutorial React17

- Step 7: Create *src/components* folder and move the file *like_button.js* into it.
- Step 8: **React**, **ReactDOM** and **PropTypes** will no longer be available on the browser.
  Edit *like_button.js* to import React and PropTypes:

```
import React from 'react';
import PropTypes from 'prop-types';
```

  Edit *like_button.js* to export LikeButton:

```
// …
export {LikeButton};
```

  Import React, ReactDOM and LikeButton into *src/script.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { LikeButton } from './components/like_button';
```

Tutorial React17

- Step 9: Now we can **reconfigure the linting**.

  Delete from *.eslintrc.json*:

```
"no-undef": "off",
"no-unused-vars": "off",
"react/jsx-no-undef": "off"
```

  The file *webpack.config.js* might be throwing errors. This is a configuration file and we don't need it to be checked by eslint.
  Create the file *.eslintignore* and add:

```
webpack.config.js
```

Tutorial React17

- Step 10: So… no errors on vscode. Lets use webpack:

```
> npx webpack --mode development
```

No errors on the console!
The file ***public/main.js*** **(our bundle) exists**!
But where is the index.html file?

Tutorial React17

- Step 11: We need a webpack plugin module to connect React Component to the DOM:

```
> npm i -D html-webpack-plugin
```

- Step 12: Edit *webpack.config.js*:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
...
  module: {
...
  },
  plugins: [
    new HtmlWebpackPlugin({
      filename: "index.html",
      template: path.join(__dirname, "src", "index.html")
    })
  ]
};
```

Tutorial React17

- Step 13: Run webpack again :

```
> npx webpack --mode development
```

- Step 14: Open the HTML file in a browser.

127.0.0.1:5500/public/index.html

React Example 16 - Webpack JSX Like Button

This is the first comment.

Like

This is the second comment.

You liked comment number 2

This is the third comment.

Like

Tutorial React17

- Step 15: Add the scripts into *package.json*

```
"scripts": {
  "eslint": "eslint src/**/*.js",
  "dev": "webpack --mode development",
  "build": "webpack --mode production",
```

- And test with

```
> npm run dev
```

- Or

```
> npm run build
```

Lab React18

- Step 1: Copy react_example_16-JSXPassingUpState to **react_example_18-WebpackJSXPassingUpState** and open in VSCode.
- Step 2: Convert the project to webpack module.

The Component Lifecycle

- In applications with many components, it's very important to free up resources taken by the components when they are destroyed.
- Each component has several "lifecycle methods" that can be override to run code at particular times in the process.
- **Lifecycle methods** are custom functionality that gets executed during the different phases of a component. There are methods available when the component gets created and inserted into the DOM (mounting), when the component updates, and when the component gets unmounted or removed from the DOM.

Mounting Lifecycle

- These methods are called in the following order when an instance of a component is being created and inserted into the DOM:
  - constructor()
  - static getDerivedStateFromProps()
  - render()
  - componentDidMount()

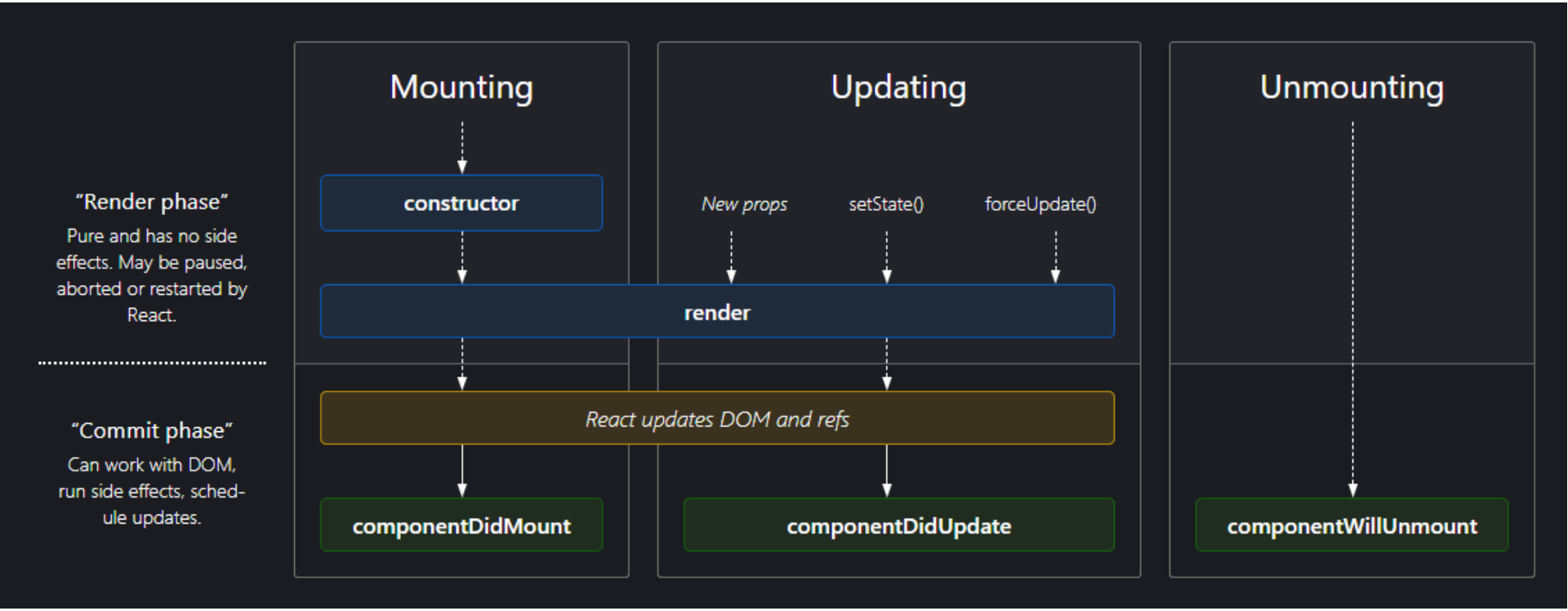
Updating Lifecycle

- An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:
  - static getDerivedStateFromProps()
  - shouldComponentUpdate()
  - render()
  - getSnapshotBeforeUpdate()
  - componentDidUpdate()

Unmounting Lifecycle

- This method is called when a component is being removed from the DOM:
  - componentWillUnmount()

Commonly Used Lifecycle Methods



See React Lifecycle methods diagram

Tutorial React19

- Step 1: Copy react_example_15-JSXPassingState to **react_example_19-Lifecicle** and open in VSCode.
- Step 2: Create file src/clockPresenter.jsx:

```
class ClockPresenter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showClock: true};
  }

  onClick = () => {
    let value = !this.state.showClock;
    this.setState({ showClock: value });
  };

  render() {
    return (
    <div>
      <button onClick={this.onClick}>{this.state.showClock ? "Hide" : "Show"}</button>
      {this.state.showClock && <Clock />}
    </div>
    );
  }
}
```

Show "Hide" if this.state.showClock is true or "Show" otherwise.

Show clock if this.state.showClock is true.

Tutorial React19

- Step 3: Edit public/index.html:

```
<!DOCTYPE html>
<html>
  <!-- ... -->
    <div id="content"></div>
    <script src="../lib/clockPresenter.js"></script>
    <!-- ... -->
    <script src="../lib/script.js"></script>
  </body>
</html>
```

- Step 4: Edit src/script.jsx

```
ReactDOM.createRoot(
  <ClockPresenter />,
  document.getElementById('content')
)
```

Tutorial React19

- Step 5: Transpile the code with:

```
> npm run build
```

- Step 6: Open index.html in browser, and notice the error in the console when clock is hidden:

```
Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a   react_devtools_backend.js:2560
memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in the componentWillUnmount method.
    at Clock (http://127.0.0.1:5500/lib/clock.js:33:5)
```

Tutorial React19

- Step 7: Edit clock.jsx:

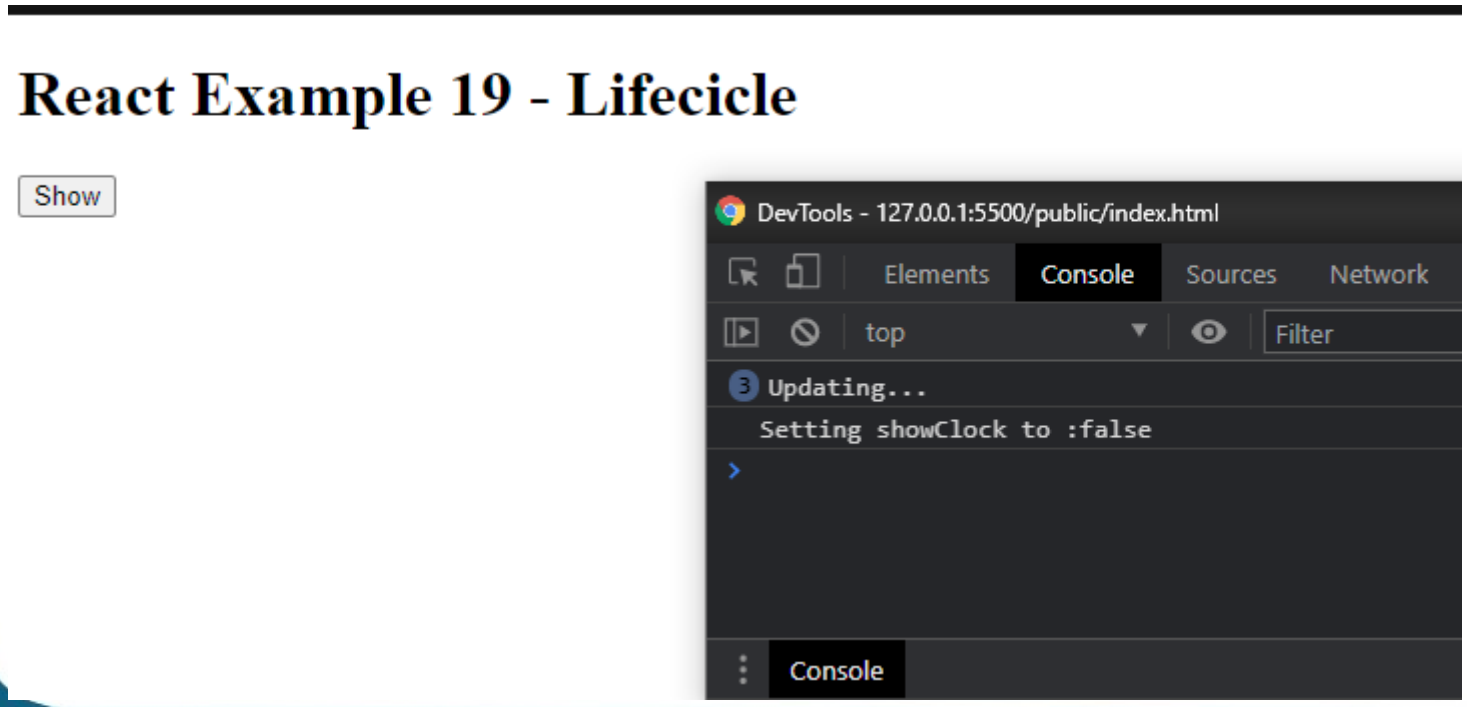
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {currentTime: new Date().toLocaleString()};
  }

  componentDidMount() {
    this.timerID = setInterval(()=> {
      console.log('Updating...')
      this.setState({currentTime: (new Date()).toLocaleString()})
    }, 1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

// …

}
```

Tutorial React19

- Step 8: Transpile the code with:

```
> npm run build
```

- Step 9: Open index.html in browser



Environment Structure