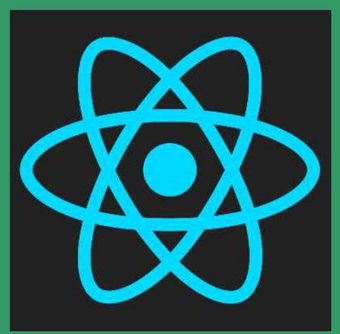
4

React JS VDOM Architecture

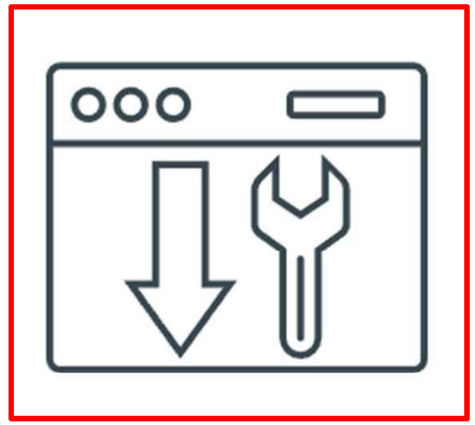


Deep Dive

Why ?

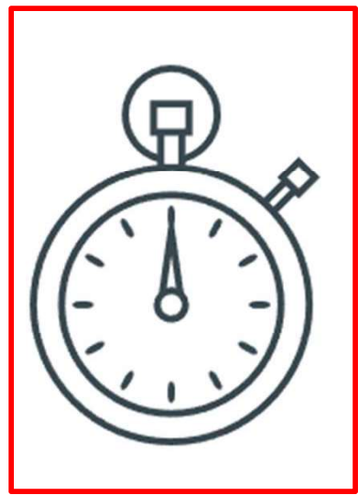
Reeact JS A different kind of library

- Closer to the DOM
  - React provides the thinnest possible Virtual DOM abstraction on top of the DOM. It builds on stable platform features, registers real event handlers and plays nicely with other libraries.
  - React can be used directly in the browser without any transpilation steps.



- Small Size
  - Most UI frameworks are large enough to be the majority of an app's JavaScript size. Preact is different: it's small enough that *your code* is the largest part of your application.
  - That means less JavaScript to download, parse and execute - leaving more time for your code, so you can build an experience you define without fighting to keep a framework under control.

- Big Performance
  - React is fast, and not just because of its size. It's one of the fastest Virtual DOM libraries out there, thanks to a simple and predictable diff implementation.
  - We automatically batch updates and tune React to the extreme when it comes to performance. We work closely with browser engineers to get the maximum performance possible out of React.



- Portable & Embeddable
  - React's tiny footprint means you can take the powerful Virtual DOM Component paradigm to new places it couldn't otherwise go.
  - Use React to build parts of an app without complex integration. Embed React into a widget and apply the same tools and techniques that you would to build a full app.

- Instantly Productive
  - Lightweight is a lot more fun when you don't have to sacrifice productivity to get there. React gets you productive right away. It even has a few bonus features:
    - props, state and context are passed to render()
    - Use standard HTML attributes like class and for

- Ecosystem Compatible
  - Virtual DOM Components make it easy to share reusable things - everything from buttons to data providers. React's design means you can seamlessly use thousands of Components available in the React ecosystem.
  - Adding a simple React/compat alias to your bundler provides a compatibility layer that enables even the most complex React components to be used in your application.

Todo List

Run in REPL

```
export default class TodoList extends Component {
    state = { todos: [], text: '' };
    setText = e => {
        this.setState({ text: e.target.value });
    };
    addTodo = () => {
        let { todos, text } = this.state;
        todos = todos.concat({ text });
        this.setState({ todos, text: '' });
    };
    render({ }, { todos, text }) {
        return (
            <form onSubmit={this.addTodo} action="javascript:">
                <label>
                  <span>Add Todo</span>
                  <input value={text} onInput={this.setText} />
                </label>
                <button type="submit">Add</button>
                <ul>
                    { todos.map( todo => (
                        <li>{todo.text}</li>
                    )) }
                </ul>
            </form>
        );
    }
}
```

Running Example

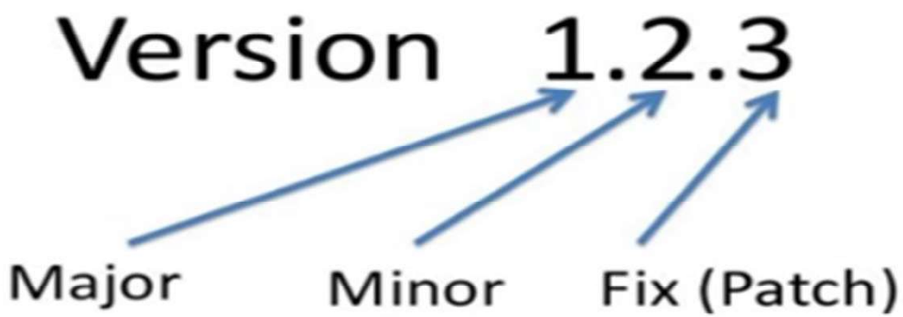
```
import TodoList from './todo-list';
```

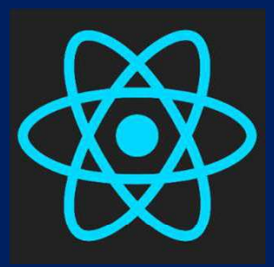
Why React JS?

Why React?

- ReactJS should not be seen as a replacement to any other JavaScript framework. Rather it is a unique in a sense that it is based on VDOM and provides data binding capabilities that can be used to create complex user interfaces.
- ReactJS library is very light weight and puts no extra load on the webpage.
- ReactJS is absolutely free and open source. It has large community support as well.
- ReactJS comes with detailed documentation and comes with lots of online help.

Semantic Versioning



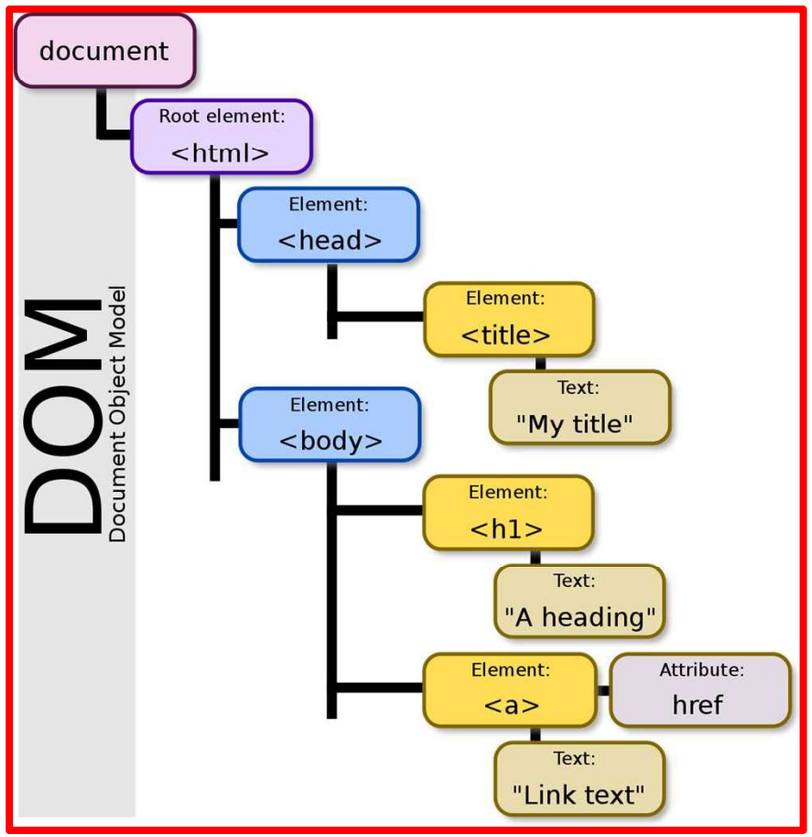


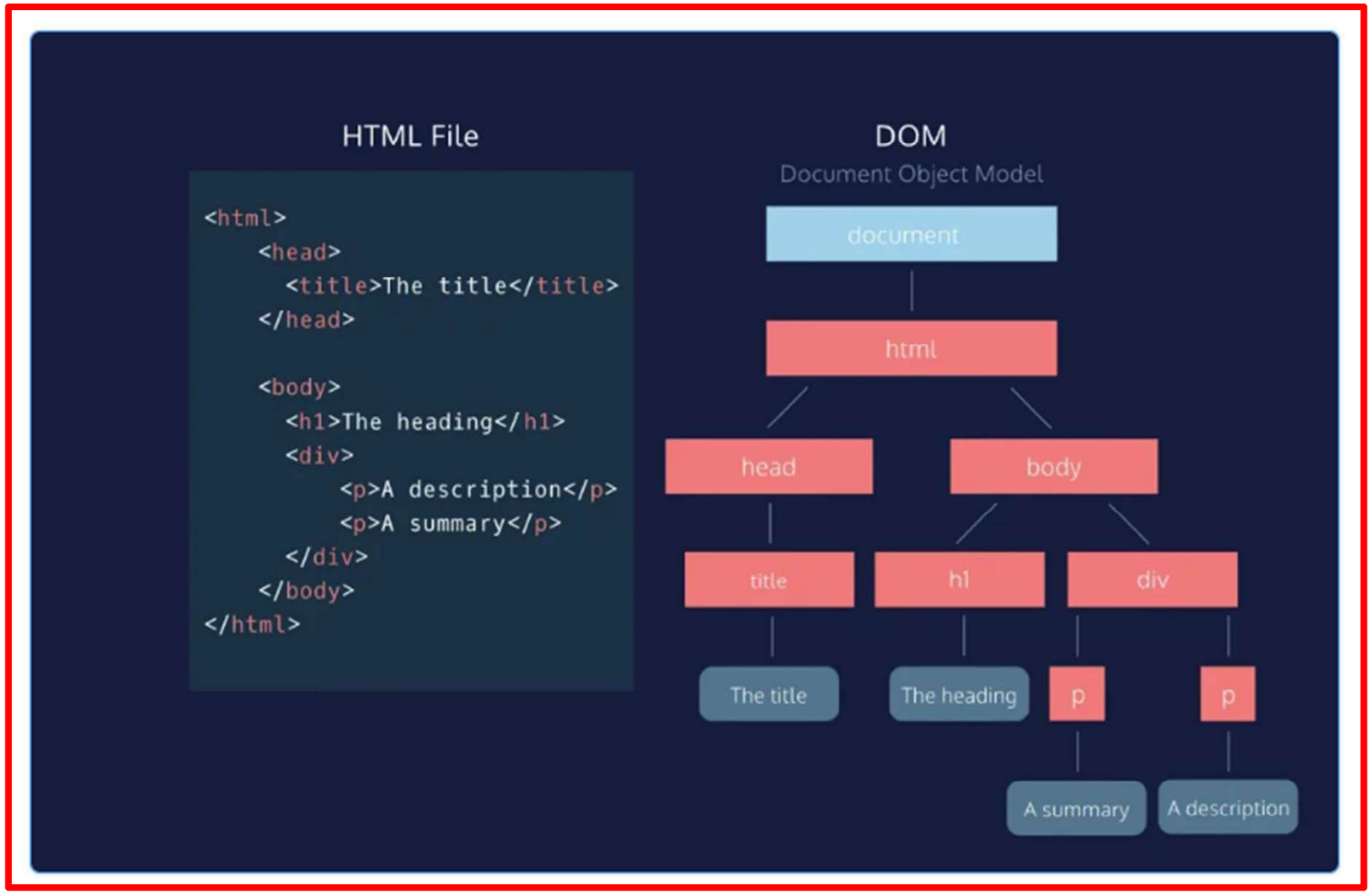
DOM

- React JS Virtual DOM is an in-memory representation of the DOM. DOM refers to the **Document Object Model** that represents the content of XML or HTML documents as a tree structure so that the programs can be read, accessed and changed in the document structure, style, and content.

What is DOM ?

- DOM stands for **‘Document Object Model’**.
- In simple terms, it is a structured representation of the HTML elements that are present in a webpage or web app.
- DOM represents the entire UI of your application.
- The DOM is represented as a tree data structure. It contains a node for each UI element present in the web document.
- It is very useful as it allows web developers to modify content through JavaScript, also it being in structured format helps a lot as we can choose specific targets and all the code becomes much easier to work with.


document
Root element:
<html>
Element:
<head>
Element:
<title>
Text:
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribute:
href
Text:
"Link text"
DOM
Document Object Model


HTML File
<html>
<head>
<title>The title</title>
</head>
<body>
<h1>The heading</h1>

A description
A summary

</body>
</html>
DOM
Document Object Model
document
html
head
body
title
h1
div
The title
The heading
p
p
A summary
A description

Disadvantages of real DOM :

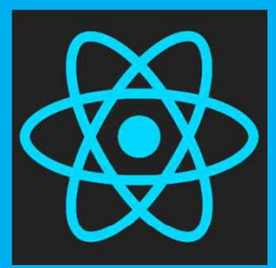
- Every time the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. For this, each time there is a component update, the DOM needs to be updated and the UI components have to be re-rendered.

```
// Simple getElementById() method
document.getElementById('some-id').innerValue = 'updated value';
```

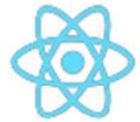
When writing the above code in the console or in the JavaScript file, these things happen:

- The browser parses the HTML to find the node with this id.
- It removes the child element of this specific element.
- Updates the element(DOM) with the ‘updated value’.
- Recalculates the CSS for the parent and child nodes.
- Update the layout.
- Finally, traverse the tree and paint it on the screen(browser) display.

Recalculating the CSS and changing the layouts involves complex algorithms, and they do affect the performance.



Virtual DOM [VDOM]



React – The Virtual DOM

- DOM manipulation is the heart of most modern and interactive web applications.
- DOM manipulation is slow, moreover many JavaScript libraries and frameworks update the DOM more than needed.
- In React for every DOM object there is a corresponding Virtual DOM object.
- A Virtual DOM object is an in-memory representation of the real DOM object.
  - It has the same properties as the real DOM object.
- It lacks the power to directly change what is on the screen.

Virtual DOM

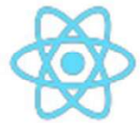
- React uses Virtual DOM exists which is like a lightweight copy of the actual DOM(a virtual representation of the DOM).
- So for every object that exists in the original DOM, there is an object for that in Preact Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document.
- Manipulating DOM is slow, but manipulating Virtual DOM is fast
- As nothing gets drawn on the screen. So each time there is a change in the state of our application, the virtual DOM gets updated first instead of the real DOM.

How does Virtual DOM actually make things faster?

- When anything new is added to the application, a virtual DOM is created and it is represented as a tree.
- Each element in the application is a node in this tree. So, whenever there is a change in the state of any element, a new Virtual DOM tree is created.
- This new Virtual DOM tree is then compared with the previous Virtual DOM tree and make a note of the changes.
- After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again.

How virtual DOM Helps React?

- In React, everything is treated as a component be it a functional component or class component. A component can contain a state. Whenever the state of any component is changed react updates its Virtual DOM tree. Though it may sound like it is ineffective the cost is not much significant as updating the virtual DOM doesn't take much time.
- React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM.

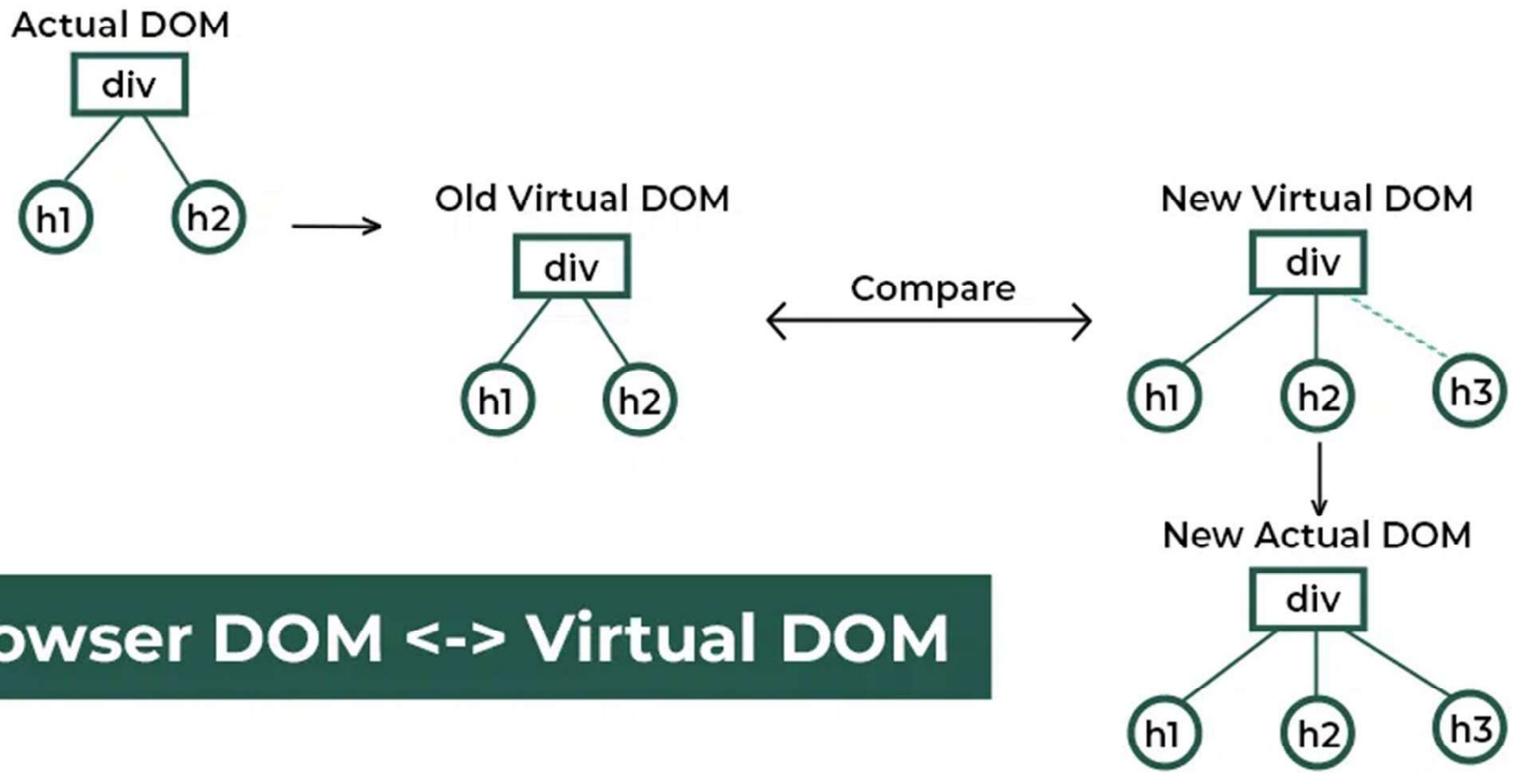


Virtual DOM...

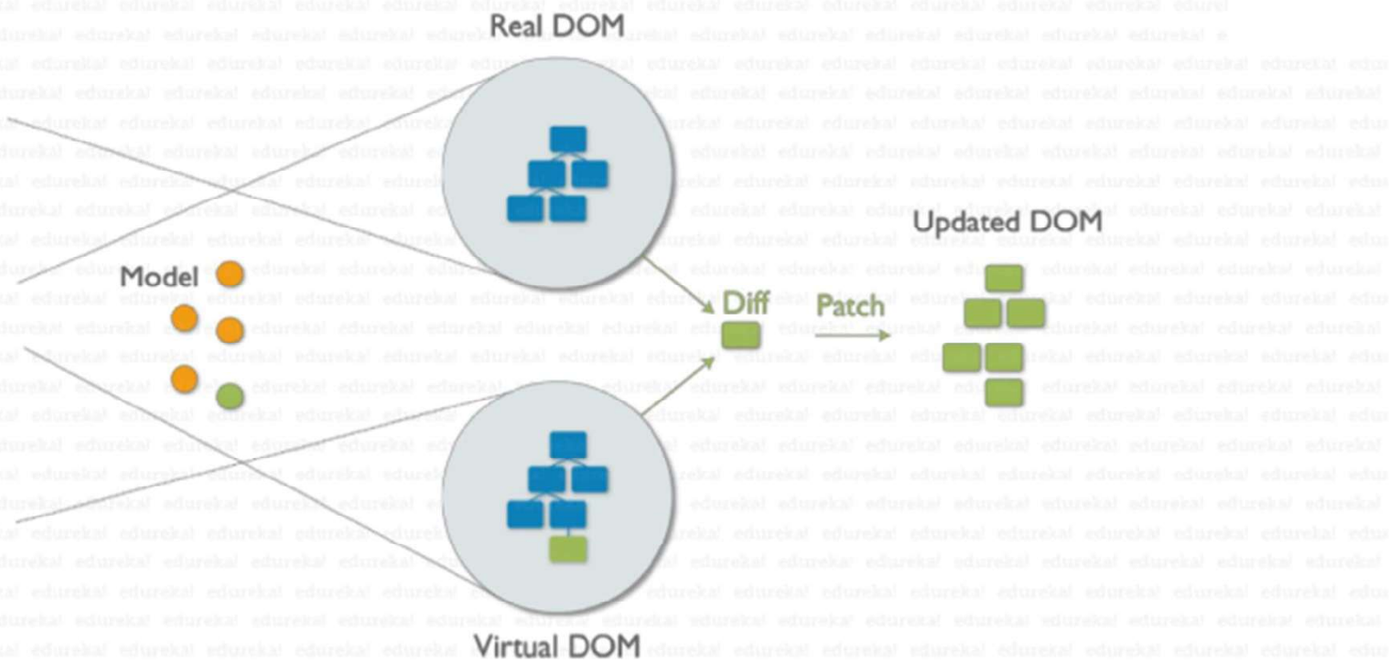
- Manipulating the DOM is slow whereas manipulating the virtual DOM is fast as nothing gets drawn on the screen.
- Once the virtual DOM is updated React compares it with its previous state.
- "**Diffing**" is the process by which React figures the virtual DOM objects that have changed.
- React updates only the changed objects in the real DOM.
- Changes on the real DOM cause the screen to change.

Cont …

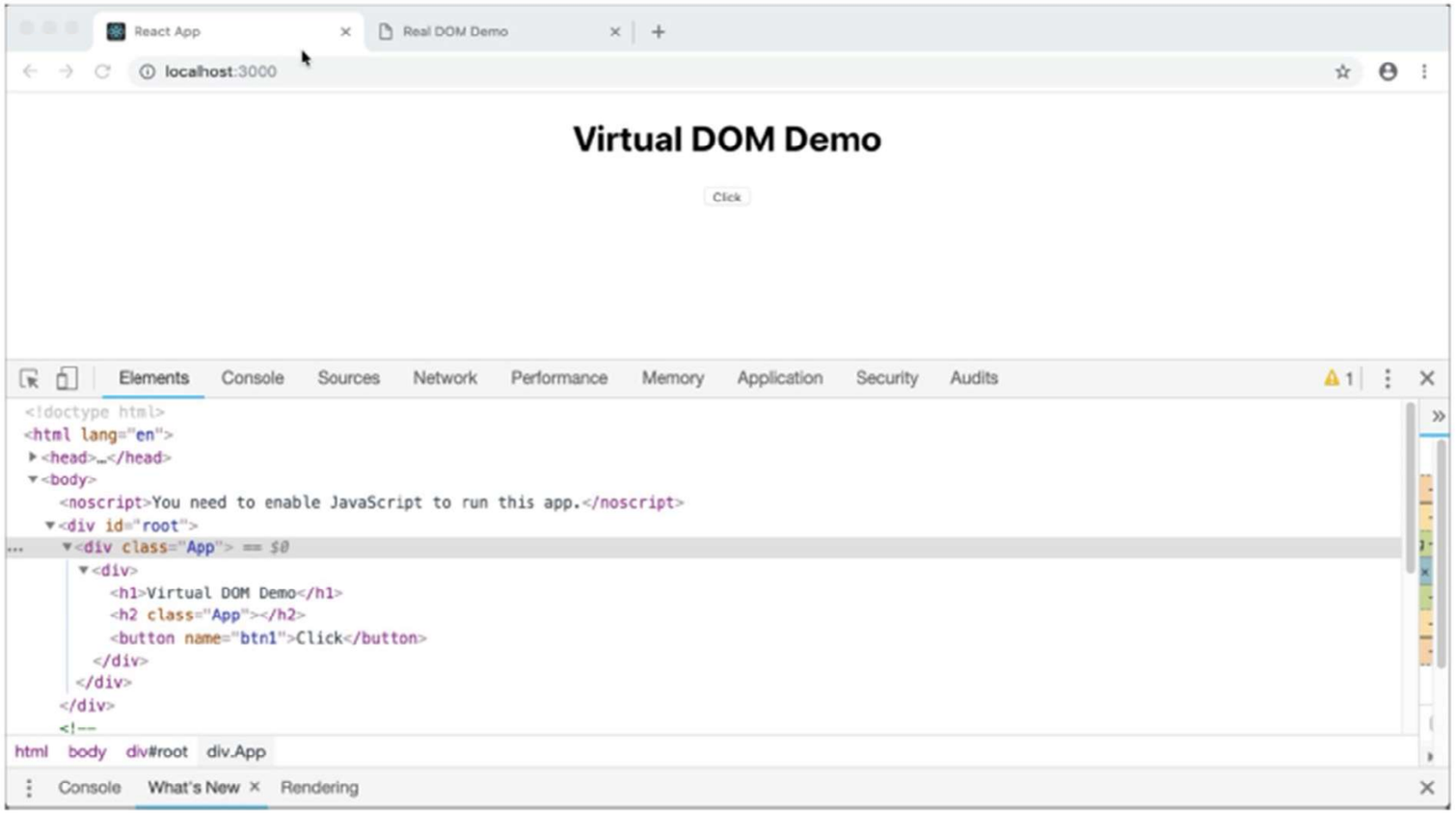
- Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as **‘diffing’**. Once React finds out what exactly has changed then it updates those objects only, on real DOM.
- React uses something called batch updates to update the real DOM. It just means that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component.


Actual DOM
div
h1
h2
Old Virtual DOM
div
h1
h2
Compare
New Virtual DOM
div
h1
h2
h3
New Actual DOM
div
h1
h2
h3
Browser DOM <-> Virtual DOM

The Virtual DOM



Virtual DOM / Real DOM



Virtual DOM Key Concepts :

- Virtual DOM is the virtual representation of Real DOM
- React update the state changes in Virtual DOM first and then it syncs with Real DOM
- Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
- Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with “Real DOM ” by a library such as ReactDOM and this process is called reconciliation
- Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions

A Virtual DOM is a simple description of a tree structure using objects:

```
let vdom = {
  type: 'p',          // a <p> element
  props: {
    class: 'big',     // with class="big"
    children: [
      'Hello World!' // and the text "Hello World!"
    ]
  }
}
```

Creating Virtual DOM trees

There are a few ways to create Virtual DOM trees:

1. createElement(): a function provided by Preact
2. JSX: HTML-like syntax that can be compiled to JavaScript
3. HTM: HTML-like syntax you can write directly in JavaScript

- Simplest approach, which would be to call Preact's **createElement()** function directly:

```
import { createElement, render } from 'preact';

let vdom = createElement(
  'p',              // a <p> element
  { class: 'big' }, // with class="big"
  'Hello World!'    // and the text "Hello World!"
);

render(vdom, document.body);
```

- The code creates a Virtual DOM "description" of a paragraph element. The first argument to createElement is the HTML element name. The second argument is the element's "props" - an object containing attributes (or properties) to set on the element. Any additional arguments are children for the element, which can be strings (like 'Hello World!') or Virtual DOM elements from additional createElement() calls.

- The last line tells Preact to build a real DOM tree that matches our Virtual DOM "description", and to insert that DOM tree into the <body> of a web page.

Now with more JSX!

- We can rewrite the previous example using JSX without changing its functionality.

```
import { createElement, render } from 'preact';

let vdom = <p class="big">Hello World!</p>;

render(vdom, document.body);
```

```
let maybeBig = Math.random() > .5 ? 'big' : 'small';

let vdom = <p class={maybeBig}>Hello {40 + 2}!</p>;
                // ^---JS---^       ^--JS--^
```

- We can rewrite the previous example using JSX without changing its functionality. JSX lets us describe our paragraph element using HTML-like syntax, which can help keep things readable as we describe more complex trees. The drawback of JSX is that our code is no longer written in JavaScript, and must be compiled by a tool like Babel. Compilers do the work of converting the JSX example below into the exact createElement() code we saw in the previous example.

- It looks a lot more like HTML now!

- There's one final thing to keep in mind about JSX: code inside of a JSX element (within the angle brackets) is special syntax and not JavaScript. To use JavaScript syntax like numbers or variables, you first need to "jump" back out from JSX using an {expression} - similar to fields in a template. The example below shows two expressions: one to set class to a randomized string, and another to calculate a number.

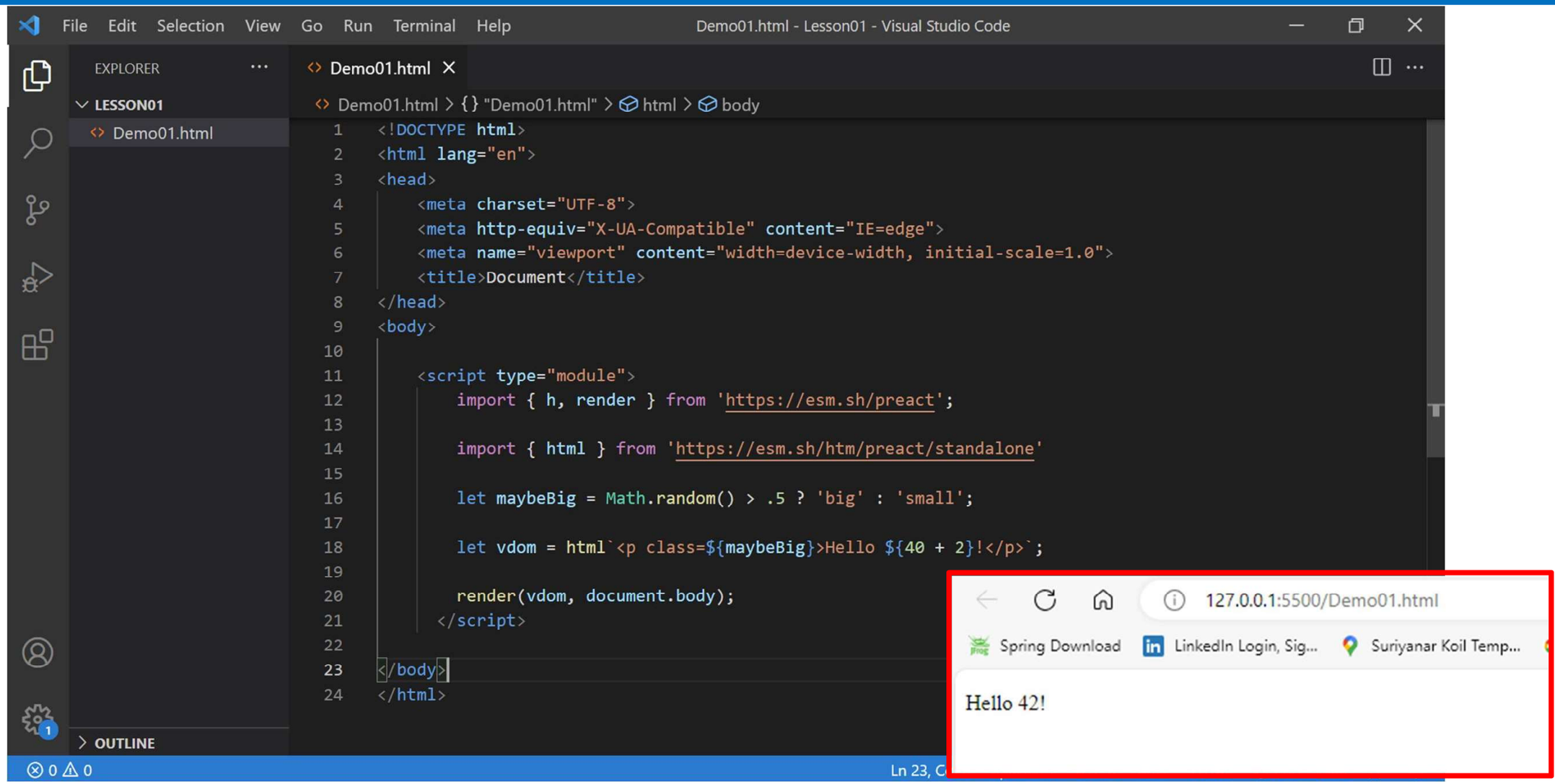
Once more with HTM

- HTM is an alternative to JSX that uses standard JavaScript tagged templates, removing the need for a compiler.
- If you haven't encountered tagged templates, they're a special type of String literal that can contain ${expression} fields:

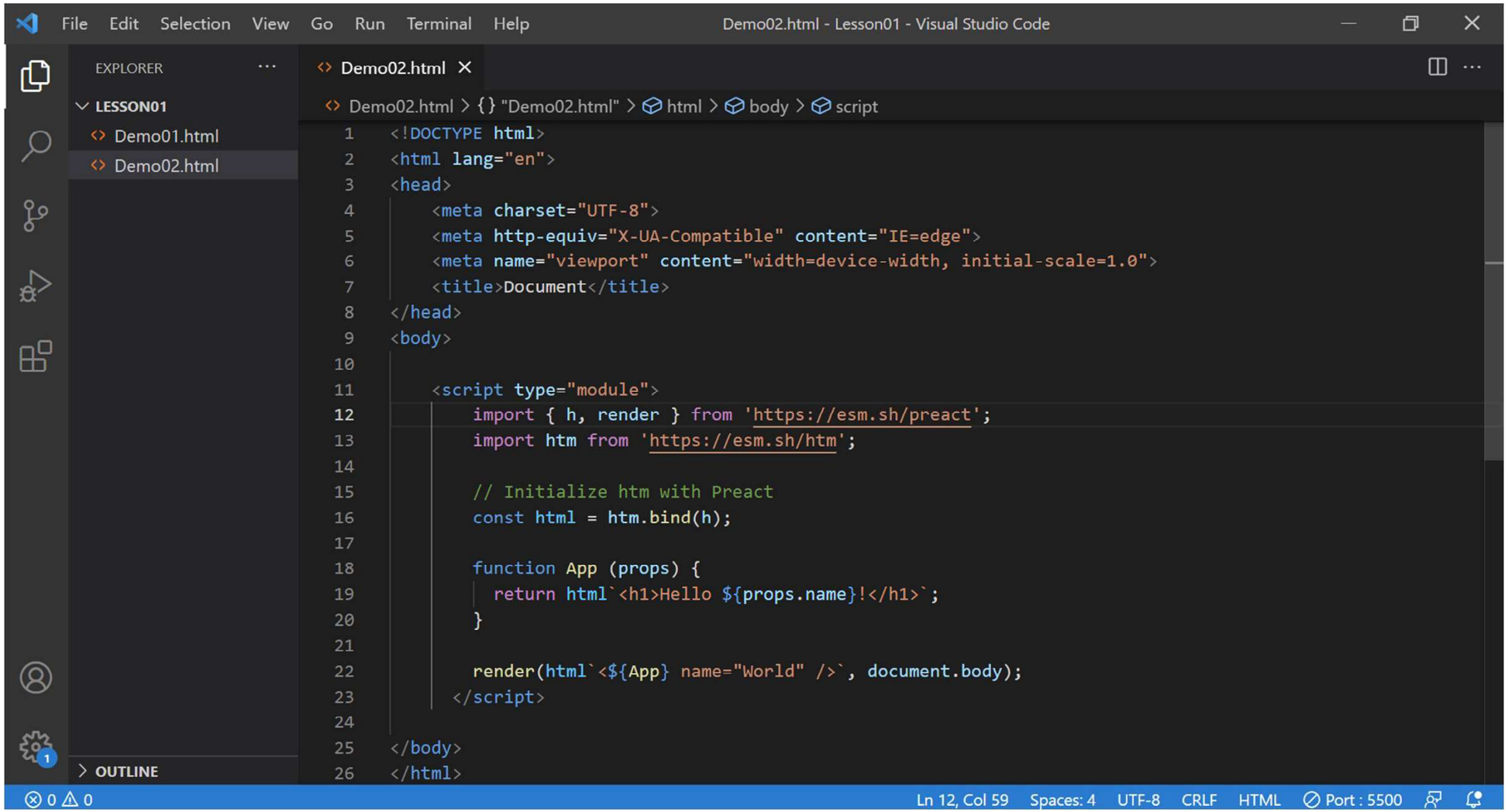
```
let str = `Quantity: ${40 + 2} units`;  // "Quantity: 42 units"
```

- HTM uses ${expression} instead of the {expression} syntax from JSX, which can make it clearer what parts of your code are HTM/JSX elements, and what parts are plain JavaScript:

```
import { html } from 'htm/preact';

let maybeBig = Math.random() > .5 ? 'big' : 'small';

let vdom = html`<p class=${maybeBig}>Hello ${40 + 2}!</p>`;
                        // ^--JS--^          ^-JS-^
```

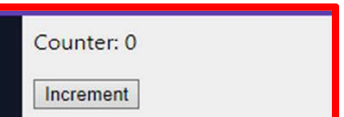

File Edit Selection View Go Run Terminal Help
Demo01.html - Lesson01 - Visual Studio Code
EXPLORER
LESSON01
Demo01.html
Demo01.html
Demo01.html > {} "Demo01.html" > html > body
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <meta http-equiv="X-UA-Compatible" content="IE=edge">
6 <meta name="viewport" content="width=device-width, initial-scale=1.0">
7 <title>Document</title>
8 </head>
9 <body>
10
11 <script type="module">
12 import { h, render } from 'https://esm.sh/preact';
13
14 import { html } from 'https://esm.sh/htm/preact/standalone'
15
16 let maybeBig = Math.random() > .5 ? 'big' : 'small';
17
18 let vdom = html`<p class=${maybeBig}>Hello ${40 + 2}!</p>`;
19
20 render(vdom, document.body);
21 </script>
22
23 </body>
24 </html>
OUTLINE
0 0
Ln 23,
127.0.0.1:5500/Demo01.html
Spring Download
LinkedIn Login, Sig...
Suriyanar Koil Temp...
Hello 42!

As a Component



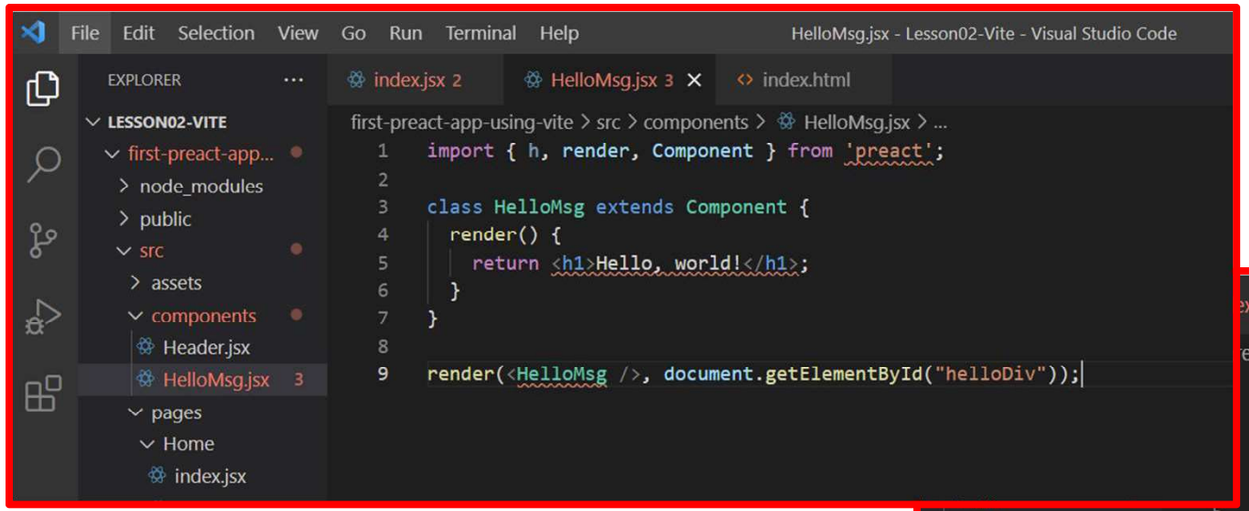
Class Based Component

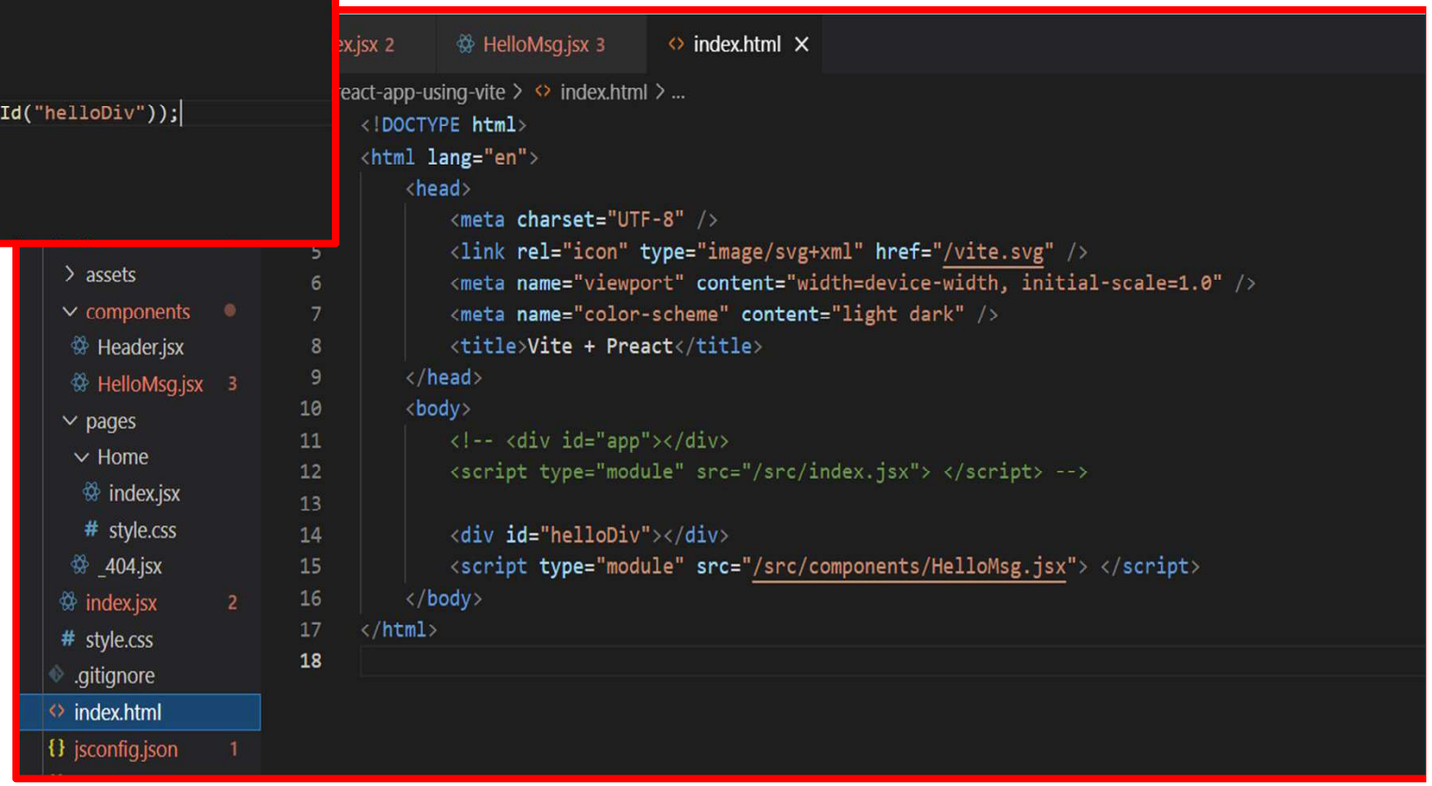
```
import { render, Component } from "preact";

class Counter extends Component {
  state = {
    value: 0
  };

  increment = () => {
    this.setState(prev => ({ value: prev.value +1 }));
  };

  render(props, state) {
    return (
      <div>
        <p>Counter: {state.value}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

render(<Counter />, document.getElementById("app"));

```



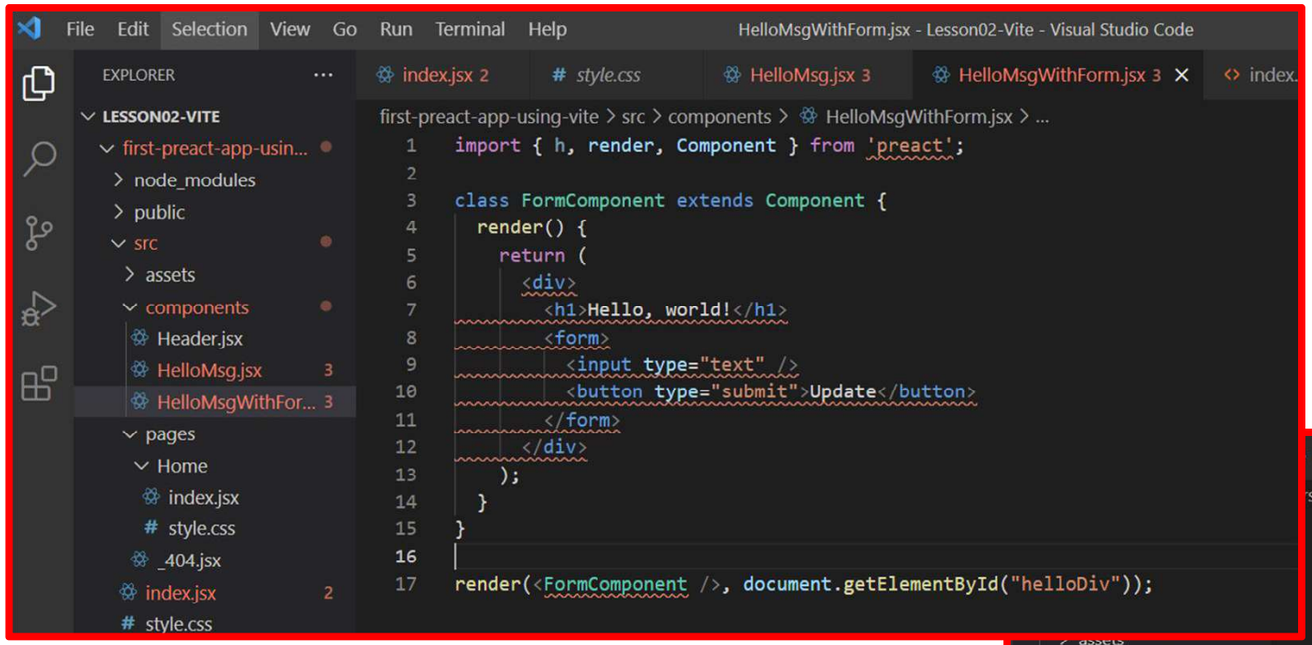
Interactive Welcome Msg

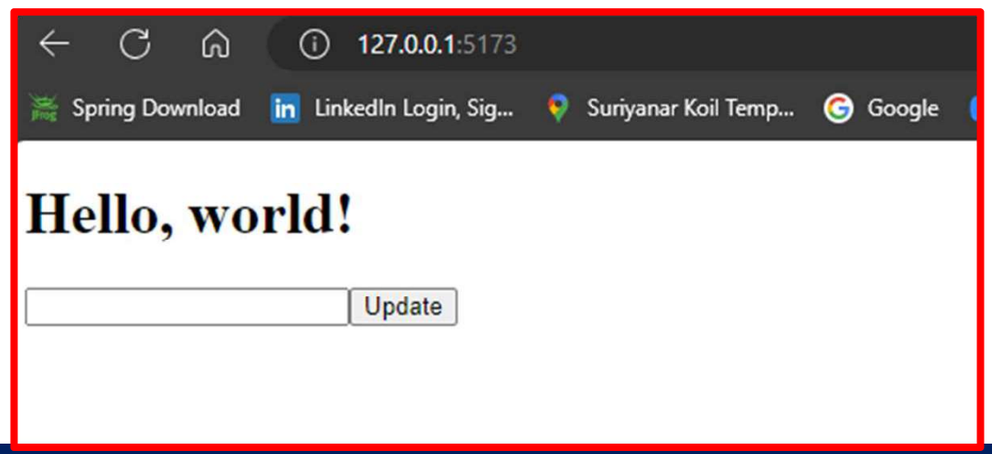
- Rendering text is a start, but we want to make our app a little more interactive. We want to update it when data changes.
- Our end goal is that we have an app where the user can enter a name and display it, when the form is submitted. For this we need to have something where we can store what we submitted.
- This is where **Components** come into play.

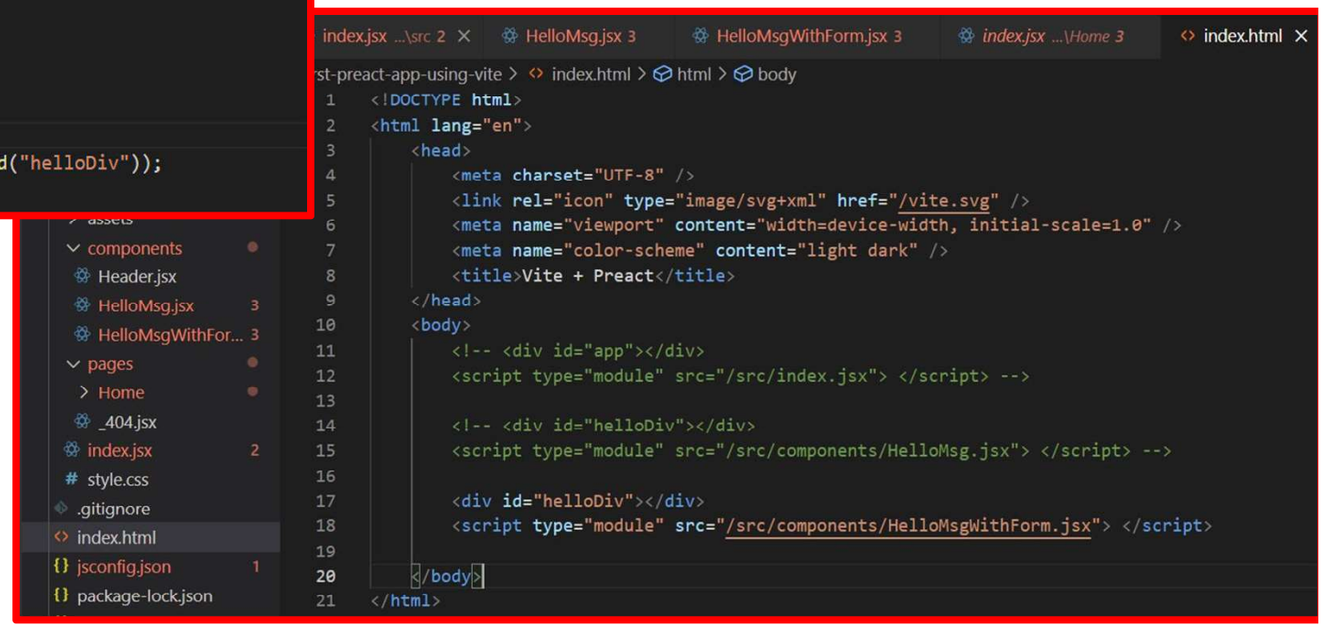

File Edit Selection View Go Run Terminal Help
HelloMsg.jsx - Lesson02-Vite - Visual Studio Code
EXPLORER
index.jsx 2
HelloMsg.jsx 3
index.html
LESSON02-VITE
first-preact-app...
node_modules
public
src
assets
components
Header.jsx
HelloMsg.jsx 3
pages
Home
index.jsx
first-preact-app-using-vite > src > components > HelloMsg.jsx > ...
1 import { h, render, Component } from 'preact';
2
3 class HelloMsg extends Component {
4 render() {
5 return <h1>Hello, world!</h1>;
6 }
7 }
8
9 render(<HelloMsg />, document.getElementById("helloDiv"));


HelloMsg.jsx 3
index.html
react-app-using-vite > index.html > ...
assets
components
Header.jsx
HelloMsg.jsx 3
pages
Home
index.jsx
style.css
_404.jsx
index.jsx 2
style.css
.gitignore
index.html
jsconfig.json 1
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<link rel="icon" type="image/svg+xml" href="/vite.svg" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<meta name="color-scheme" content="light dark" />
<title>Vite + Preact</title>
</head>
<body>
<!--
<script type="module" src="/src/index.jsx"> </script> -->

<script type="module" src="/src/components/HelloMsg.jsx"> </script>
</body>
</html>


File Edit Selection View Go Run Terminal Help
HelloMsgWithForm.jsx - Lesson02-Vite - Visual Studio Code
EXPLORER
LESSON02-VITE
first-preact-app-usin...
node_modules
public
src
assets
components
Header.jsx
HelloMsg.jsx 3
HelloMsgWithFor... 3
pages
Home
index.jsx
style.css
_404.jsx
index.jsx 2
style.css
index.jsx 2
style.css
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
first-preact-app-using-vite > src > components > HelloMsgWithForm.jsx > ...
1 import { h, render, Component } from 'preact';
2
3 class FormComponent extends Component {
4 render() {
5 return (
6
7 <h1>Hello, world!</h1>
8 <form>
9 <input type="text" />
10 <button type="submit">Update</button>
11 </form>
12
13);
14 }
15 }
16
17 render(<FormComponent />, document.getElementById("helloDiv"));

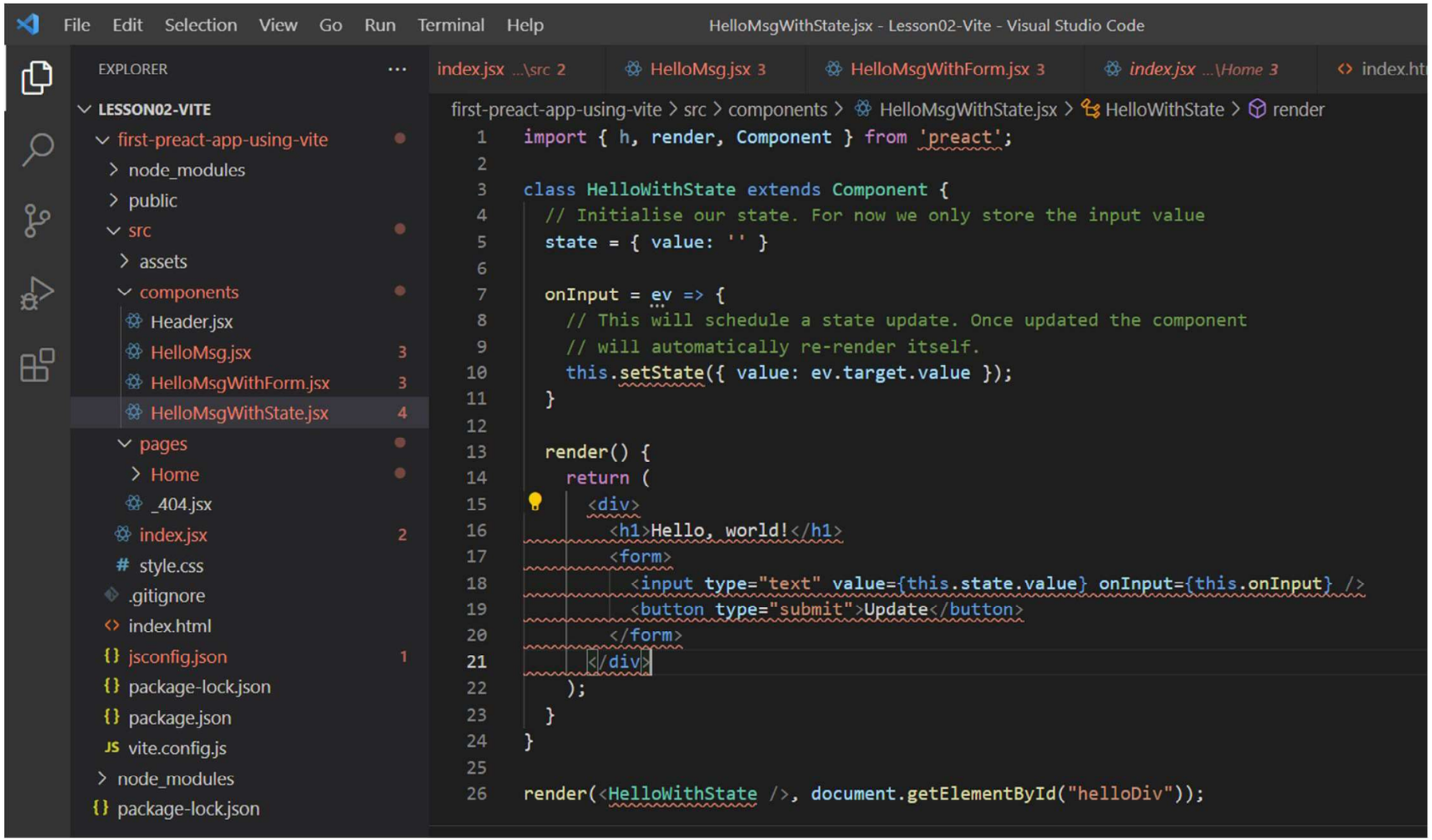

127.0.0.1:5173
Spring Download
LinkedIn Login, Sig...
Suriyanar Koil Temp...
Google
Hello, world!
Update


index.jsx ...\src 2
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
index.jsx ...\Home 3
index.html
first-preact-app-using-vite > index.html > html > body
components
Header.jsx
HelloMsg.jsx 3
HelloMsgWithFor... 3
pages
Home
_404.jsx
index.jsx 2
style.css
.gitignore
index.html
jsconfig.json 1
package-lock.json
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8" />
5 <link rel="icon" type="image/svg+xml" href="/vite.svg" />
6 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7 <meta name="color-scheme" content="light dark" />
8 <title>Vite + Preact</title>
9 </head>
10 <body>
11 <!--
12 <script type="module" src="/src/index.jsx"> </script> -->
13
14 <!--
15 <script type="module" src="/src/components/HelloMsg.jsx"> </script> -->
16
17
18 <script type="module" src="/src/components/HelloMsgWithForm.jsx"> </script>
19
20 </body>
21 </html>

- We'll change "Hello world!" to "Hello, [userinput]!", so we need a way to know the current input value.

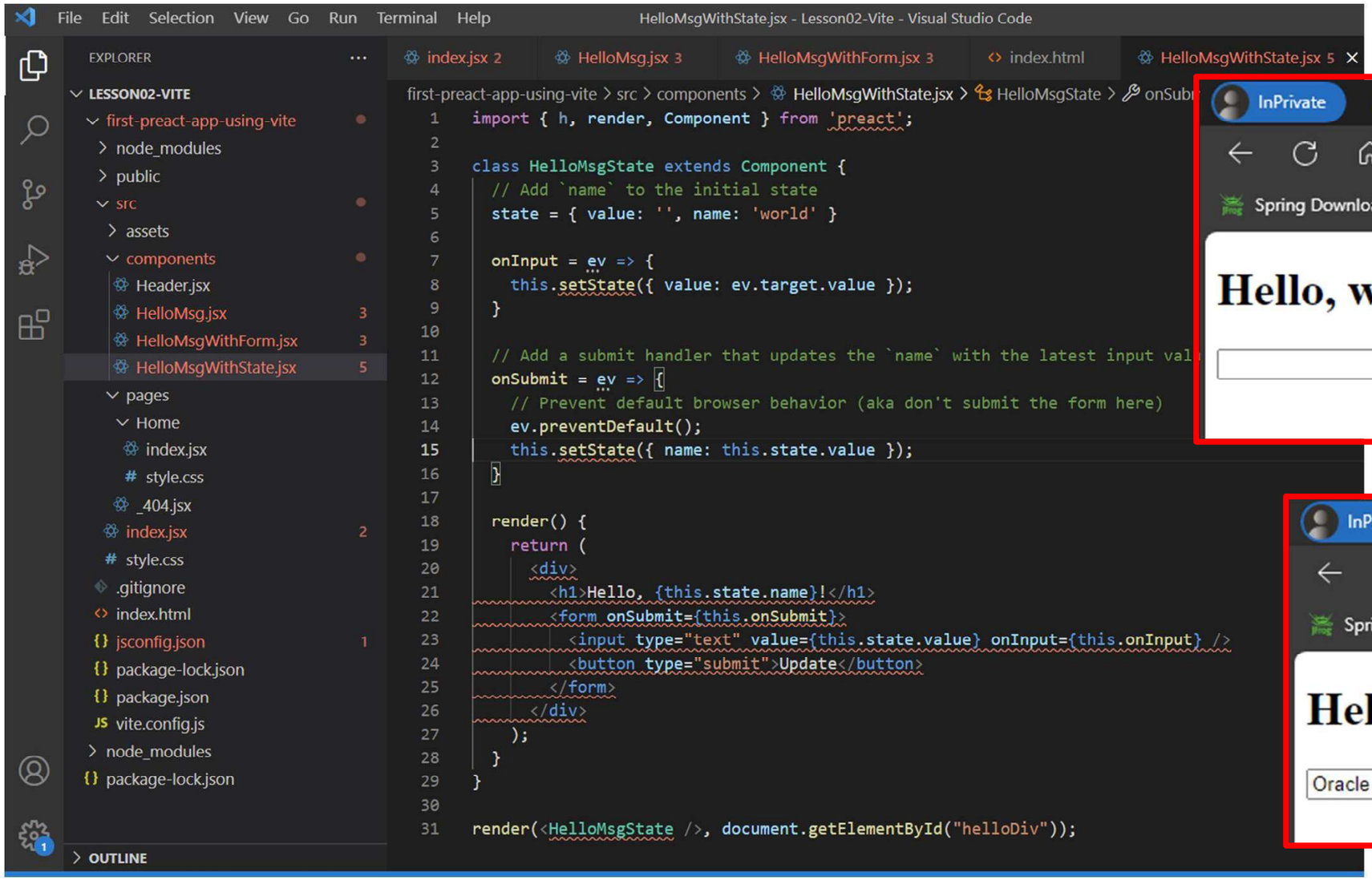
- We'll store it in a special property called state of our Component. It's special, because when it's updated via the setState method, Preact will not just update the state, but also schedule a render request for this component. Once the request is handled, our component will be re-rendered with the updated state.

- Lastly we need to attach the new state to our input by setting value and attaching an event handler to the input event.

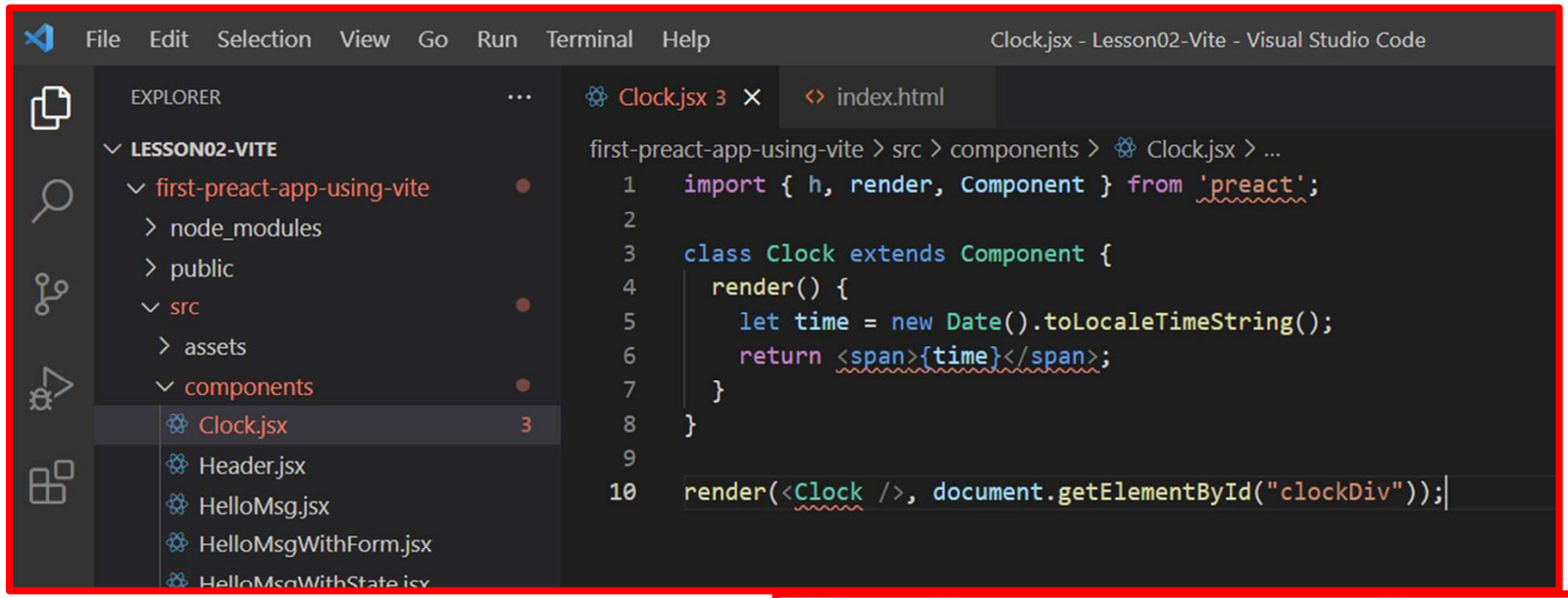

File Edit Selection View Go Run Terminal Help
HelloMsgWithState.jsx - Lesson02-Vite - Visual Studio Code
EXPLORER
LESSON02-VITE
first-preact-app-using-vite
node_modules
public
src
assets
components
Header.jsx
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
HelloMsgWithState.jsx 4
pages
Home
_404.jsx
index.jsx 2
style.css
.gitignore
index.html
jsconfig.json 1
package-lock.json
package.json
vite.config.js
node_modules
package-lock.json
index.jsx ...\src 2
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
index.jsx ...\Home 3
index.ht
first-preact-app-using-vite > src > components > HelloMsgWithState.jsx > HelloWithState > render
1 import { h, render, Component } from 'preact';
2
3 class HelloWithState extends Component {
4 // Initialise our state. For now we only store the input value
5 state = { value: '' }
6
7 onInput = ev => {
8 // This will schedule a state update. Once updated the component
9 // will automatically re-render itself.
10 this.setState({ value: ev.target.value });
11 }
12
13 render() {
14 return (
15
16 <h1>Hello, world!</h1>
17 <form>
18 <input type="text" value={this.state.value} onInput={this.onInput} />
19 <button type="submit">Update</button>
20 </form>
21
22);
23 }
24 }
25
26 render(<HelloWithState />, document.getElementById("helloDiv"));

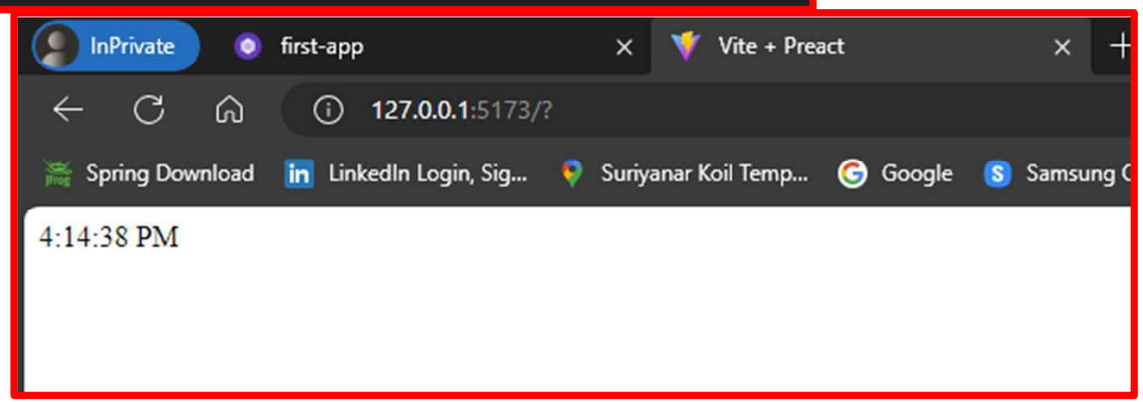
- At this point the app shouldn't have changed much from a users point of view, but we'll bring all the pieces together in our next step.

- We'll add a handler to the submit event of our <form> in similar fashion like we just did for the input. The difference is that it writes into a different property of our state called name. Then we swap out our heading and insert our state.name value there.


File Edit Selection View Go Run Terminal Help
HelloMsgWithState.jsx - Lesson02-Vite - Visual Studio Code
EXPLORER
LESSON02-VITE
first-preact-app-using-vite
node_modules
public
src
assets
components
Header.jsx
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
HelloMsgWithState.jsx 5
pages
Home
index.jsx
style.css
_404.jsx
index.jsx 2
style.css
.gitignore
index.html
jsconfig.json 1
package-lock.json
package.json
vite.config.js
node_modules
package-lock.json
OUTLINE
index.jsx 2
HelloMsg.jsx 3
HelloMsgWithForm.jsx 3
index.html
HelloMsgWithState.jsx 5
first-preact-app-using-vite > src > components > HelloMsgWithState.jsx > HelloMsgState > onSubi
1 import { h, render, Component } from 'preact';
2
3 class HelloMsgState extends Component {
4 // Add `name` to the initial state
5 state = { value: '', name: 'world' }
6
7 onInput = ev => {
8 this.setState({ value: ev.target.value });
9 }
10
11 // Add a submit handler that updates the `name` with the latest input val
12 onSubmit = ev => {
13 // Prevent default browser behavior (aka don't submit the form here)
14 ev.preventDefault();
15 this.setState({ name: this.state.value });
16 }
17
18 render() {
19 return (
20
21 <h1>Hello, {this.state.name}!</h1>
22 <form onSubmit={this.onSubmit}>
23 <input type="text" value={this.state.value} onInput={this.onInput} />
24 <button type="submit">Update</button>
25 </form>
26
27);
28 }
29 }
30
31 render(<HelloMsgState />, document.getElementById("helloDiv"));
InPrivate
first-app
127.0.0.1:5173/?
Spring Download
LinkedIn Login, Sig...
Suriyanar
Hello, world!
Update
InPrivate
first-app
127.0.0.1:5173/?
Spring Download
LinkedIn Login, Sig...
Hello, Oracle!
Oracle
Update

Clock Component





- Preact is not intended to be a reimplementation of React. There are differences.
- The reason Preact does not attempt to include every single feature of React is in order to remain **small** and **focused -** otherwise it would make more sense to simply submit optimizations to the React project

API Reference

- **API Reference is the BEST Friend for Developer**
  - https://preactjs.com/guide/v10/api-reference
- **Some of the Important References**
  1. Component
     A. Component.render(props, state)
  2. render()
  3. hydrate()
  4. h() / createElement()
  5. toChildArray
  6. cloneElement
  7. createContext
  8. createRef
  9. Fragment

Loads less script

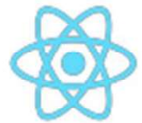
Preact's small size is valuable when you have a tight loading performance budget. On average mobile hardware, loading large bundles of JS leads to longer load, parse and eval times. This can leave users waiting a long time before they can interact with your app. By trimming down the library code in your bundles, you load quicker by shipping less code to your users.

Faster time to interactivity

If you're aiming to be interactive in under 5s, every KB matters. Switching React for Preact in your projects can shave multiple KBs off and enable you to get interactive in one RTT. This makes it a great fit for Progressive Web Apps trying to trim down as much code as possible for each route.

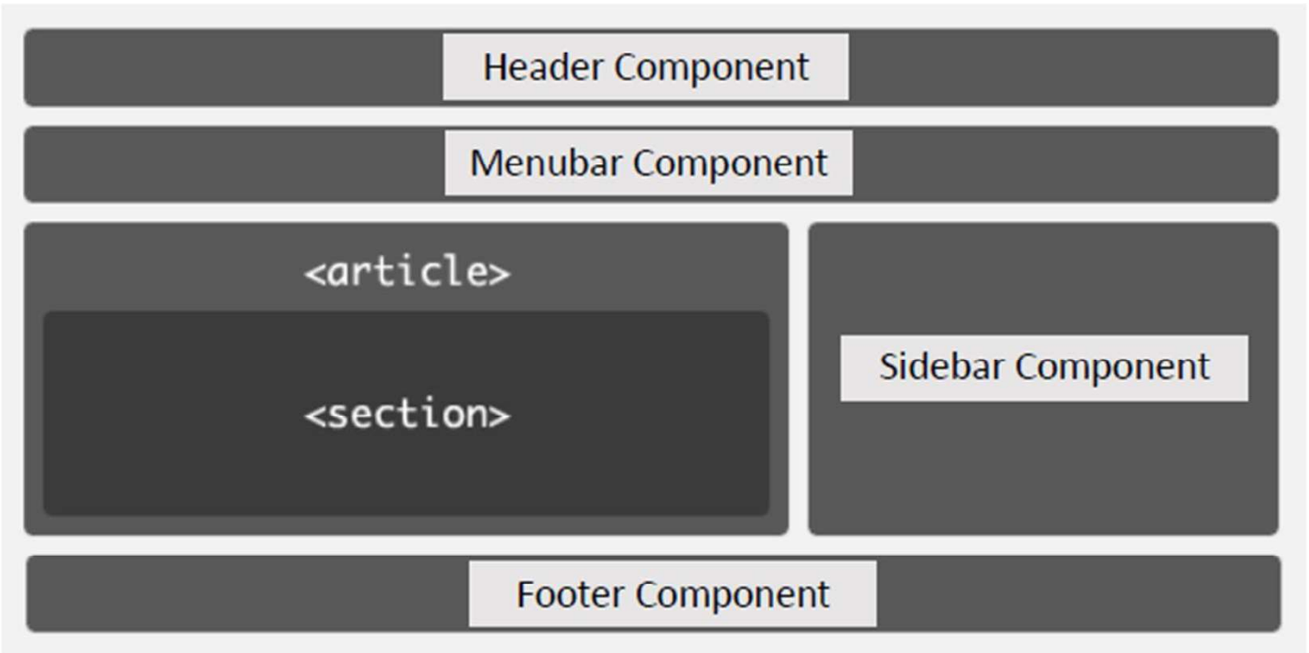
A building block that works great with the React ecosystem

Whether you need to use React's server-side rendering to get pixels on the screen quickly or use React Router for navigation, Preact works well with many libraries in the ecosystem.

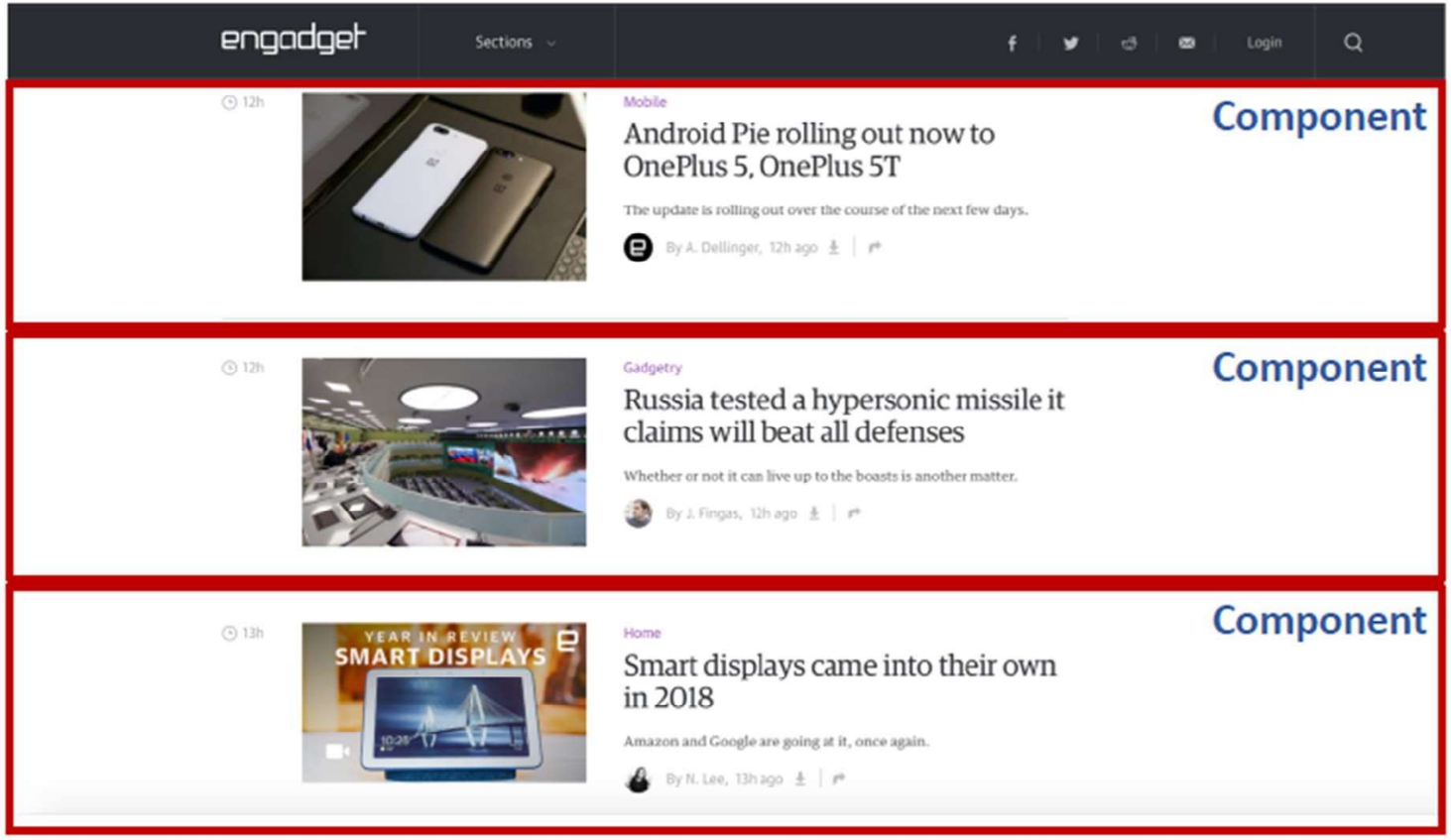


Components?

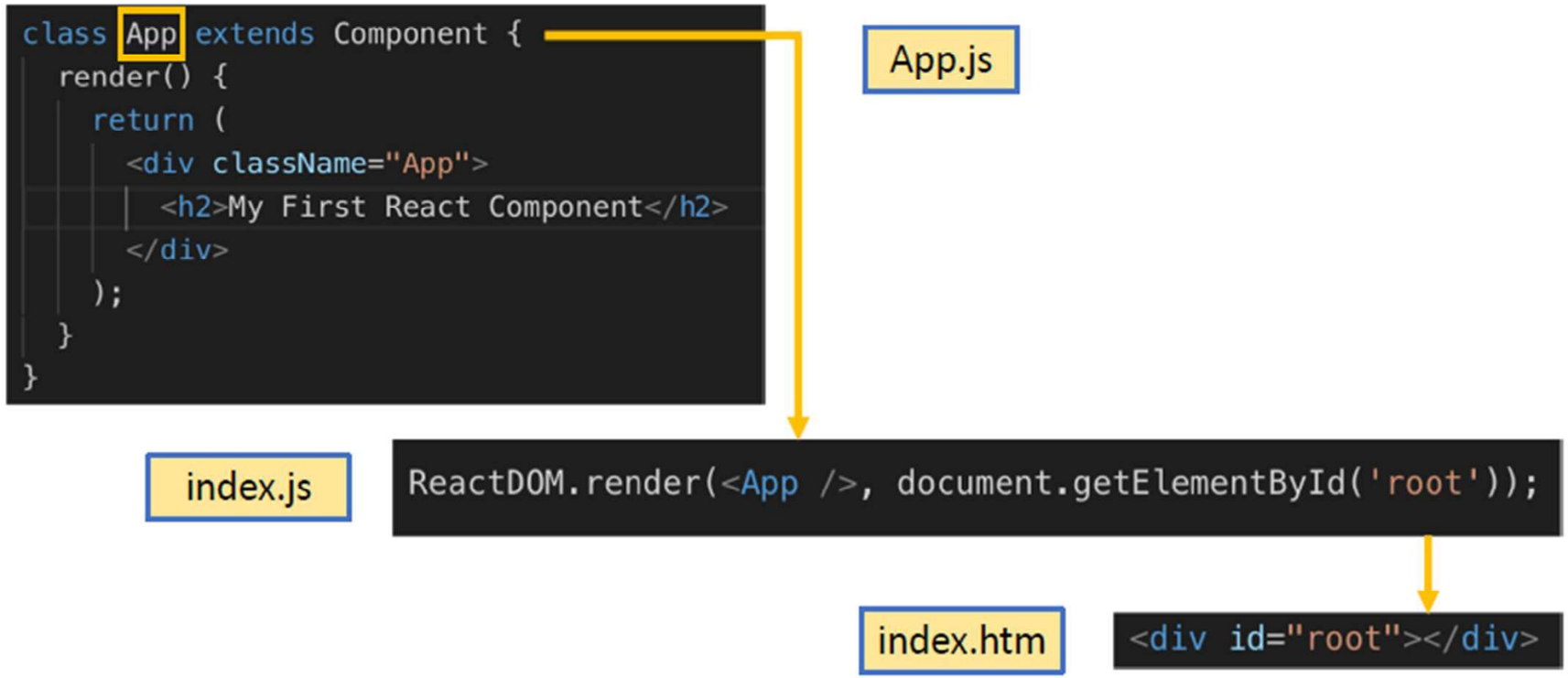
- Every web page can be split into components.
- Components can be reused and hence help simplify the creation of web pages

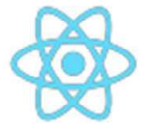


Components…


engadget
Sections
Login
12h
Mobile
Android Pie rolling out now to OnePlus 5, OnePlus 5T
The update is rolling out over the course of the next few days.
By A. Dellinger, 12h ago
Component
12h
Gadgetry
Russia tested a hypersonic missile it claims will beat all defenses
Whether or not it can live up to the boasts is another matter.
By J. Fingas, 12h ago
Component
13h
YEAR IN REVIEW
SMART DISPLAYS
Home
Smart displays came into their own in 2018
Amazon and Google are going at it, once again.
By N. Lee, 13h ago
Component

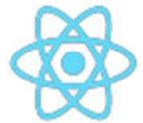
React Components





React Components

- In a react application generally, one root component is rendered.
- All other components needed by the application are nested in there.
- A react component is extended from the **`Component`** class that is imported from the **`react`** library.
- The component has a method called **`render()`**
  - The **`render()`** method returns "HTML like" content.
  - This method is called by ***react*** to emit content into the DOM and thereby render content to the view.

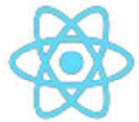


JSX?

- JavaScript eXtension (JSX) is used in React to describe the UI.
- JSX is a preprocessor step that adds XML syntax to JavaScript.
  - Like in XML, JSX tags have a tag name, attributes and children.
  - If a value is enclosed in quotes, it is regarded as a string.

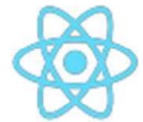
```
const appElement = <h2 className='App'>React Component</h2>;
```

- JSX makes React a lot more elegant though React can be used without JSX.
- JSX allows HTML to be put into JavaScript.



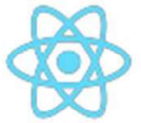
JSX

- React embraces "*separation of concerns*" with loosely coupled units called components that contain both markup and logic.
- React uses JSX for templating instead of regular JavaScript.
- JSX syntax is intended to be used by preprocessors (like Babel) to transform HTML-like text into standard JavaScript objects that can be parsed by a JavaScript engine.
- Familiarity of HTML helps create templates quicker.
- JSX is faster since optimization is performed when compiling code into JavaScript.
- JSX produces React "Nodes"



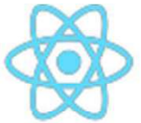
React Nodes

- A React Node is defined as a light, stateless, immutable and virtual representation node.
- React nodes are not real DOM nodes themselves, but a representation of a potential DOM node.
- The representation is considered the virtual DOM.
- React is used to define a virtual DOM using React nodes.
- React nodes can be created using JSX or JavaScript.



Creating React Nodes

- Creating React nodes using JavaScript is accomplished by **`React.createElement()`**
- This method is used to create a virtual DOM representation of an element node.
- To create the virtual DOM the React element node should be rendered to a real DOM.
- This is achieved using the **`ReactDOM.render()`** method.



JSX...

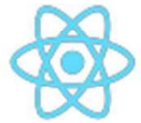
- JSX is simply converting XML-like markup into JavaScript.

```
<h2>React Component</h2>
```

- gets compiled to

```
React.createElement('h2',null,'React Component')
```

  - The first argument is the element that is to be rendered
  - The second argument is a JavaScript object and is optional
  - The third argument represents the children (the content nested in the element mentioned as the first argument)



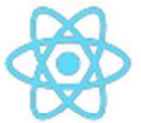
JSX...

```
<div className="App"><p>React Component</p></div>
```

- transforms to

```
React.createElement('div',{className:'App'},
                    React.createElement('p', null, 'React
                                                   Component'))
```

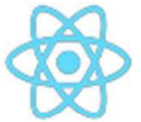
- React uses **`className`** instead of the traditional DOM **`class`**.
  - **`class`** is a keyword in ES6



Expressions

- The **{ }** brackets in JSX indicate that the content is JavaScript.
- It is eventually parsed by the JavaScript engine.
- **{ }** can be used anywhere among the JSX expressions as long as the result is a valid JavaScript.

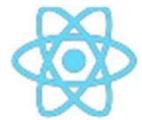
```
const contentNode = 'React Component';

<h2>{ contentNode }</h2>
```



Conditional statements

- `if-else` statements do not work in JSX.
- `if` statements can be used outside JSX to determine the content.
- Ternary operator can be employed in JSX

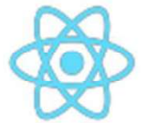
```
<select name='selCity'>
  <option>Delhi</option>
  <option>Mumbai</option>
  { currCity==='Bangalore' ? null :
                    <option>Bangalore</option>
  }
</select>
```



IIFE

- Immediately invoked function expression can be used in JSX for logic.

```
{(() => {
      if(localStorage.getItem('user'))
        this.userName = localStorage.getItem('user')
      else
        this.userName = 'Guest';
     return <p>{this.userName}</p>
    }
)()}
```



Creating Components

- A React application can be depicted as a component tree - having one root component ("App") and a number of nested child components.
- Components can be created in two possible ways:
  - **Functional components**
    - Also known as *presentational* or *stateless* components
  - **Class-based components**
    - Also known as *containers* or *stateful* components

Component Props

- Component ***Props*** (short for properties) function like HTML attributes.
- Props provide configuration values for the component.
  - Props is **readonly**

```
<User userName=' Rahul ' />
<User userName='Sam' />
```

App.js

```
const user = (props) => {
    return <p>{props.userName}</p>
}
```

User.js

Props – Children property

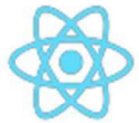
- React provides access to a special prop called children.
- Children is a reserved term referring to the all the content between the component tag.

```
<User userName = 'Lakshman'>Lakshman M N</User>
<User userName = 'Sam' />
```

App.js

User.js

```
<p>User Name : {props.userName}</p>
<p>Name : {props.children}</p>
```



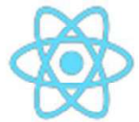
Props Validation

- With the developmental growth of an application a number of bugs can be screened using type checking.
- The `prop-types` library can be used to run type checking on the props for a component.

```
npm install --save prop-types
```

- `prop-types` can be used to document the intended types of properties passed to components.
- React will check props passed to components against those definitions and warn if they do not match.

Props Validation

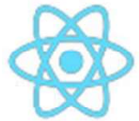


```
import PropTypes from 'prop-types';
```

- `PropTypes` exports a range of validators that can be used to ensure valid data.

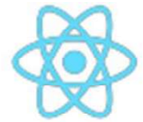
```
course.propTypes = {
    name : PropTypes.string,
    duration : PropTypes.number
    }
```

- If default props are set for the React component, the values are first resolved before type-checking against `propTypes`.
  - Default values are also subject to prop type definitions.
- `propTypes` type-checking only happens in development mode.



Props Validation

- Some of the prominent validators available include:
  - `PropTypes.bool`
  - `PropTypes.number`
  - `PropTypes.string`
  - `PropTypes.func`
  - `PropTypes.array`
  - `PropTypes.object`
  - `PropTypes.string.isRequired` – `isRequired` can be chained to any prop validator



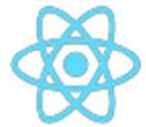
Props Validation

- Multiple Types
  - `PropTypes.oneOf` – the prop is limited to a specific set of values

```
prodCondition : PropType.oneOf(['New', 'Used', 'NA'])
```

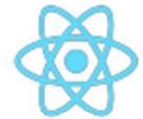
  - `PropTypes.oneOfType` – the prop should be one of a specified set of types

```
courseDuration : PropType.oneOfType([
                    PropType.string,
                    PropType.number
                    ])
```



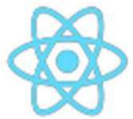
Props Validation

- Custom validators
  - `prop-types` allow custom validation functions for type-checking
  - The validation function accepts three arguments
    - **props** – an object containing all the props passed to the component
    - **propName** – the name of the prop to be validated
    - **componentName** – the name of the component
  - It should return an `Error` object if the validation fails.



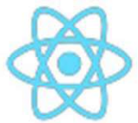
Component **State**

- Most components should take in *props* and render.
- Components also offer ***state*** and this is used to store information about the component that can change over time.
- A ***state*** change implicitly re-renders the component.
- The ***state*** object should only contain minimal amount of data needed for the UI.
- ***state*** is available only in components that extend the **`Component`** class.



Props and State

- **Props** and **State** are core concepts in React.
- Both are plain JavaScript objects.
- Both can have default values.
- Both should be accessed using **`this`** (`this.props` or `this.state`)
  - Both are **readonly** when using `this`
- Changes to **Props** and/or **State** trigger React to re-render components and potentially update the DOM in the browser.



Props and State

- **Props**
  - Props are passed into the component from above (generally parent component)
  - Props are intended as configuration values passed into the component (like arguments passed to a function)
  - Props are immutable to the component receiving them.
- **State**
  - State is a serializable representation of data (JS object) generally associated with the UI.
  - Only class based components can define and use state.
  - State should always start with a default value.
  - State can only be mutated by the component that contains it.
  - *State should be avoided if possible*