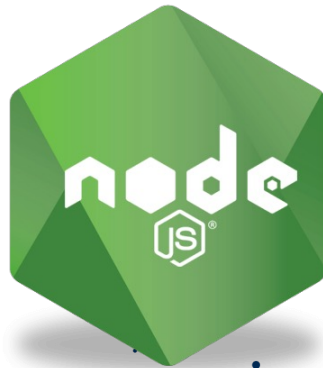




**ĐẠI HỌC  
BÁCH KHOA HÀ NỘI**  
HANOI UNIVERSITY  
OF SCIENCE AND TECHNOLOGY

# FROM DOM TO REACT



ONE LOVE. ONE FUTURE.

## 1. Document Object Model

- Document Object Model (DOM)
- Browser Object Model (BOM)

## 2. React

- Single-Page Application (SPA)
- React



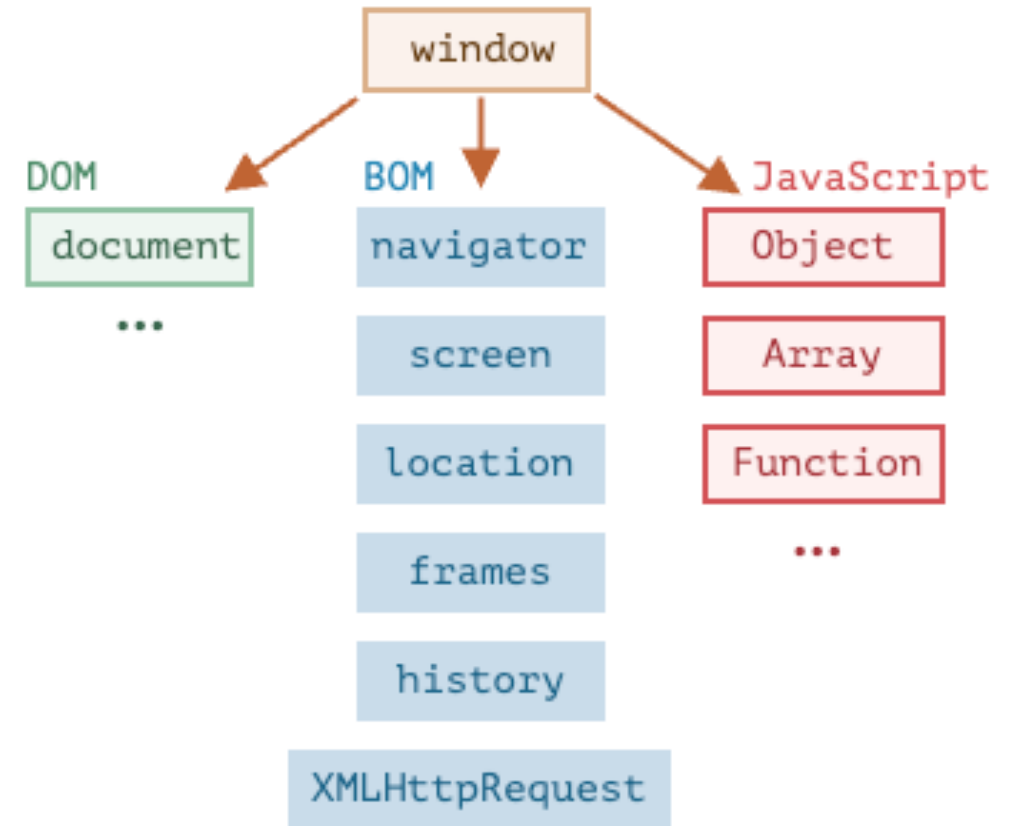
**HUST**

# Document Object Model (DOM)



# Browser environment

- JavaScript language was initially created for web browsers
  - it has evolved into a language with many uses and platforms
- A platform provides its own objects and functions in addition to the language core.
  - Web browsers give a means to control web pages
  - Node.js provides server-side features



# What is DOM?

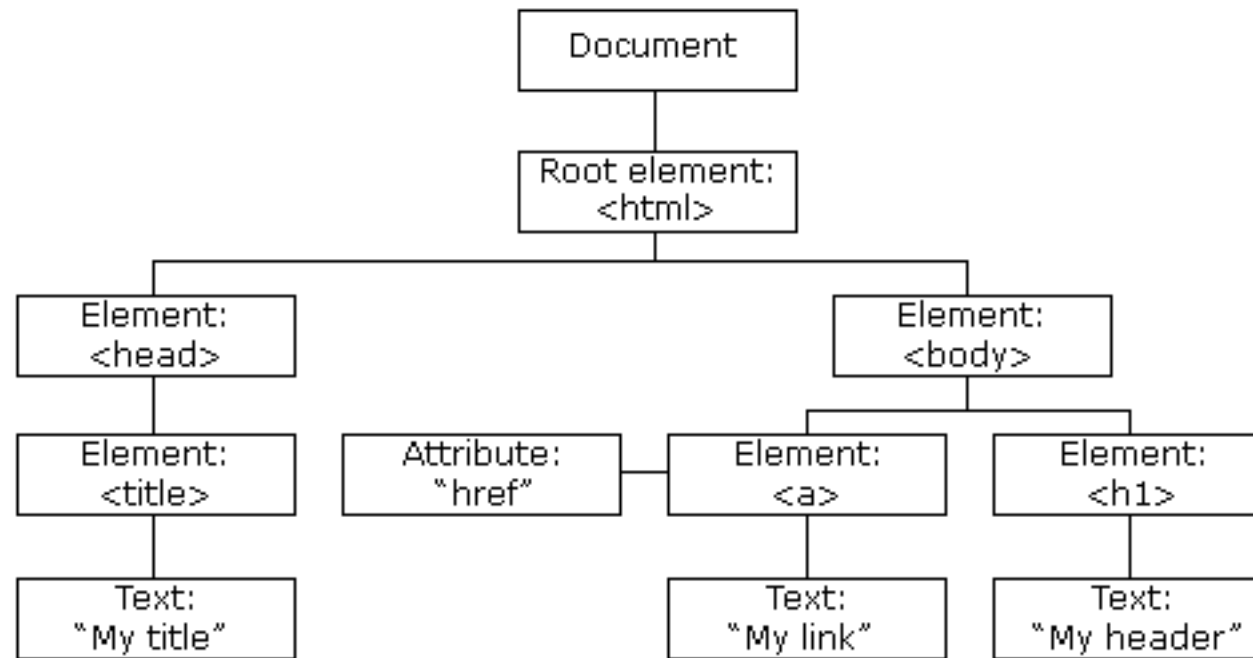
- A W3C (World Wide Web Consortium) standard.
- A programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

- The W3C DOM standard is separated into 2 parts:
  - DOM Core - standard model for all document types (HTML, XML)
  - DOM HTML - standard model for HTML documents
- The DOM HTML defines:
  - The HTML elements as objects
  - The properties of all HTML elements
  - The methods to access all HTML elements
  - The events for all HTML elements
- In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

# DOM Tree

- When a web page is loaded, the browser creates a Document Object Model of the page.

## The HTML DOM Tree of Objects



- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
  - JavaScript can add, remove or change all the HTML **elements** in the page
  - JavaScript can add, remove change all the HTML **attributes** in the page
  - JavaScript can change all the **CSS styles** in the page
  - JavaScript can create new HTML **events** or react to all existing HTML events in the page



# Searching: document.getElementById(id)

- If an element has the **id** attribute, we can get the element using the method **document.getElementById(id)**, no matter where it is.

```
1 <div id="elem">
2   <div id="elem-content">Element</div>
3 </div>
4
5 <script>
6   // get the element
7   let elem = document.getElementById('elem');
8
9   // make its background red
10  elem.style.background = 'red';
11 </script>
```

# Searching: document.getElementById(id)

- The id must be unique:
  - There can be only one element in the document with the given id.
  - If there are multiple elements with the same id, then the behavior of methods that use it is unpredictable
- Only **document.getElementById**, not anyElem.getElementById
  - The method getElementById can be called only on **document** object. It looks for the given id in the whole document.

# Searching: `getElementsByTagName*`

- There are also other methods to look for nodes
- Today, they are mostly history, as `querySelector` is more powerful and shorter to write.
  - `elem.getElementsByTagName(tag)`  
e.g., `table.getElementsByTagName("td");`
  - `elem.getElementsByClassName(className)`  
e.g., `document.getElementsByClassName("example");`
  - `document.getElementsByName(name)`  
e.g.,  
`<input name="animal" type="checkbox" value="Cats">`  
`document.getElementsByName("animal");`

# Searching: `getElementsBy*`

```
1 <table id="table">
2   <tr>
3     <td>Your age:</td>
4
5     <td>
6       <label>
7         <input type="radio" name="age" value="young" checked> less than 18
8       </label>
9       <label>
10        <input type="radio" name="age" value="mature"> from 18 to 50
11      </label>
12      <label>
13        <input type="radio" name="age" value="senior"> more than 60
14      </label>
15    </td>
16  </tr>
17 </table>
18
19 <script>
20   let inputs = table.getElementsByTagName('input');
21
22   for (let input of inputs) {
23     alert( input.value + ': ' + input.checked );
24   }
25 </script>
```

# Searching: querySelectorAll

- The most versatile method:  
`elem.querySelectorAll(css)` returns  
**all** elements inside **elem** matching  
the given CSS selector
- Any CSS selector can be used
- Can use pseudo-classes like `:hover`  
and `:active`

| Selector                    | Example         |
|-----------------------------|-----------------|
| <u>.class</u>               | .intro          |
| .class1.class2              | .name1.name2    |
| .class1 .class2             | .name1 .name2   |
| <u>#id</u>                  | #firstname      |
| <u>*</u>                    | *               |
| <u>element</u>              | p               |
| <u>element.class</u>        | p.intro         |
| <u>element,element</u>      | div, p          |
| <u>element element</u>      | div p           |
| <u>element&gt;element</u>   | div > p         |
| <u>element+element</u>      | div + p         |
| <u>element1~element2</u>    | p ~ ul          |
| [ <u>attribute</u> ]        | [target]        |
| [ <u>attribute=value</u> ]  | [target=_blank] |
| [ <u>attribute~=value</u> ] | [title~=flower] |

# Searching: querySelectorAll

```
1 <ul>
2   <li>The</li>
3   <li>test</li>
4 </ul>
5 <ul>
6   <li>has</li>
7   <li>passed</li>
8 </ul>
9 <script>
10   let elements = document.querySelectorAll('ul > li:last-child');
11
12   for (let elem of elements) {
13     alert(elem.innerHTML); // "test", "passed"
14   }
15 </script>
```

- The call to `elem.querySelector(css)` returns the **first** element for the given CSS selector
- The result is the same as `elem.querySelectorAll(css)[0]`
  - The latter is looking for all elements and picking one
  - `elem.querySelector` just looks for one. So it's faster and also shorter to write.

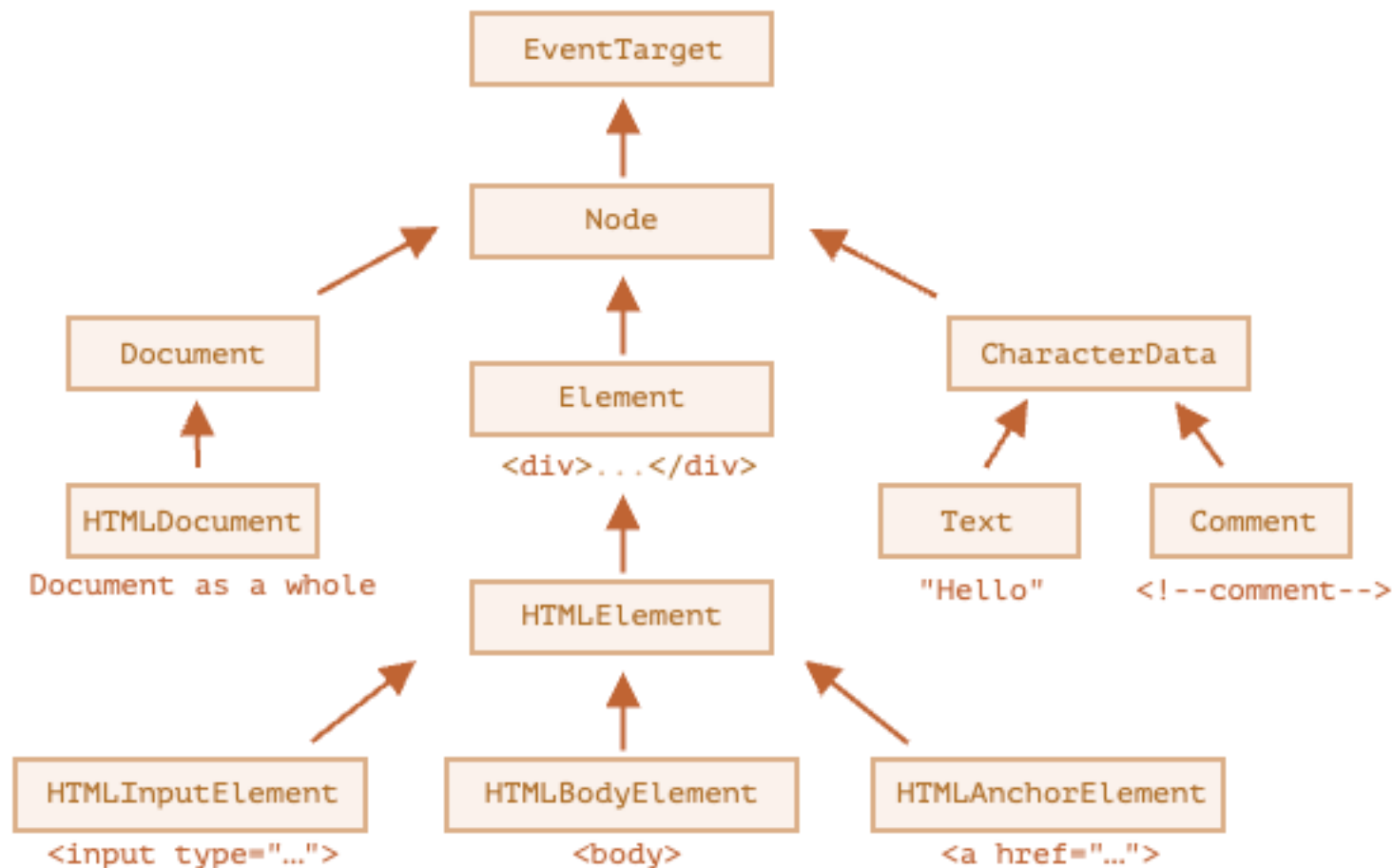
# Searching: Summary

| Method                              | Searches by... | Can call on an element? |
|-------------------------------------|----------------|-------------------------|
| <code>querySelector</code>          | CSS-selector   | ✓                       |
| <code>querySelectorAll</code>       | CSS-selector   | ✓                       |
| <code>getElementById</code>         | id             | -                       |
| <code>getElementsByName</code>      | name           | -                       |
| <code>getElementsByTagName</code>   | tag or '*'     | ✓                       |
| <code>getElementsByClassName</code> | class          | ✓                       |



- Different DOM nodes may have different properties.
  - An element node corresponding to tag <a> has link-related properties
  - The one corresponding to <input> has input-related properties and so on.
  - Text nodes are not the same as element nodes.
- But there are also common properties and methods between all of them

# Node properties: hierarchy



# The “nodeType” property

- The **nodeType** property provides one more, “old-fashioned” way to get the “type” of a DOM node.
- It has a numeric value:
  - `elem.nodeType == 1` for element nodes,
  - `elem.nodeType == 3` for text nodes,
  - `elem.nodeType == 8` for comment nodes,
  - `elem.nodeType == 9` for the document object

# “nodeType” property example

```
1 <body>
2   <script>
3     let elem = document.body;
4
5     // let's examine: what type of node is in elem?
6     alert(elem.nodeType); // 1 => element
7
8     // and its first child is...
9     alert(elem.firstChild.nodeType); // 3 => text
10
11    // for the document object, the type is 9
12    alert( document.nodeType ); // 9
13  </script>
14 </body>
```

# Tag: nodeName and tagName

- Given a DOM node, we can read its tag name from nodeName or tagName properties:

```
1 alert( document.body.nodeName ); // BODY
2 alert( document.body.tagName ); // BODY
```

- Is there any difference between tagName and nodeName?
  - The tagName property exists only for Element nodes.
  - The nodeName is defined for any Node:
    - for elements it means the same as tagName.
    - for other node types (text, comment, etc.) it has a string with the node type.

# Tag: nodeName and tagName

```
1 <body><!-- comment -->
2
3 <script>
4     // for comment
5     alert( document.body.firstChild.tagName ); // undefined (not an element)
6     alert( document.body.firstChild.nodeName ); // #comment
7
8     // for document
9     alert( document.tagName ); // undefined (not an element)
10    alert( document.nodeName ); // #document
11 </script>
12 </body>
```

# innerHTML: the contents

- The innerHTML property allows to get the HTML inside the element as a string.
- We can also modify it. So it's one of the most powerful ways to change the page.
- The example shows the contents of document.body and then replaces it completely:

```
1 <body>
2   <p>A paragraph</p>
3   <div>A div</div>
4
5   <script>
6     alert( document.body.innerHTML ); // read the current contents
7     document.body.innerHTML = 'The new BODY!'; // replace it
8   </script>
9
10 </body>
```

# outerHTML: full HTML of the element

- The outerHTML property contains the full HTML of the element. That's like innerHTML plus the element itself.

```
1 <div id="elem">Hello <b>World</b></div>
2
3 <script>
4   alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
5 </script>
```



# nodeValue/data: text node content

- The innerHTML property is only valid for element nodes.
- Other node types, such as text nodes, have their counterpart: nodeValue and data properties.
- These two are almost the same for practical use.

```
1 <body>
2   Hello
3   <!-- Comment -->
4   <script>
5     let text = document.body.firstChild;
6     alert(text.data); // Hello
7
8     let comment = text.nextSibling;
9     alert(comment.data); // Comment
10  </script>
11 </body>
```

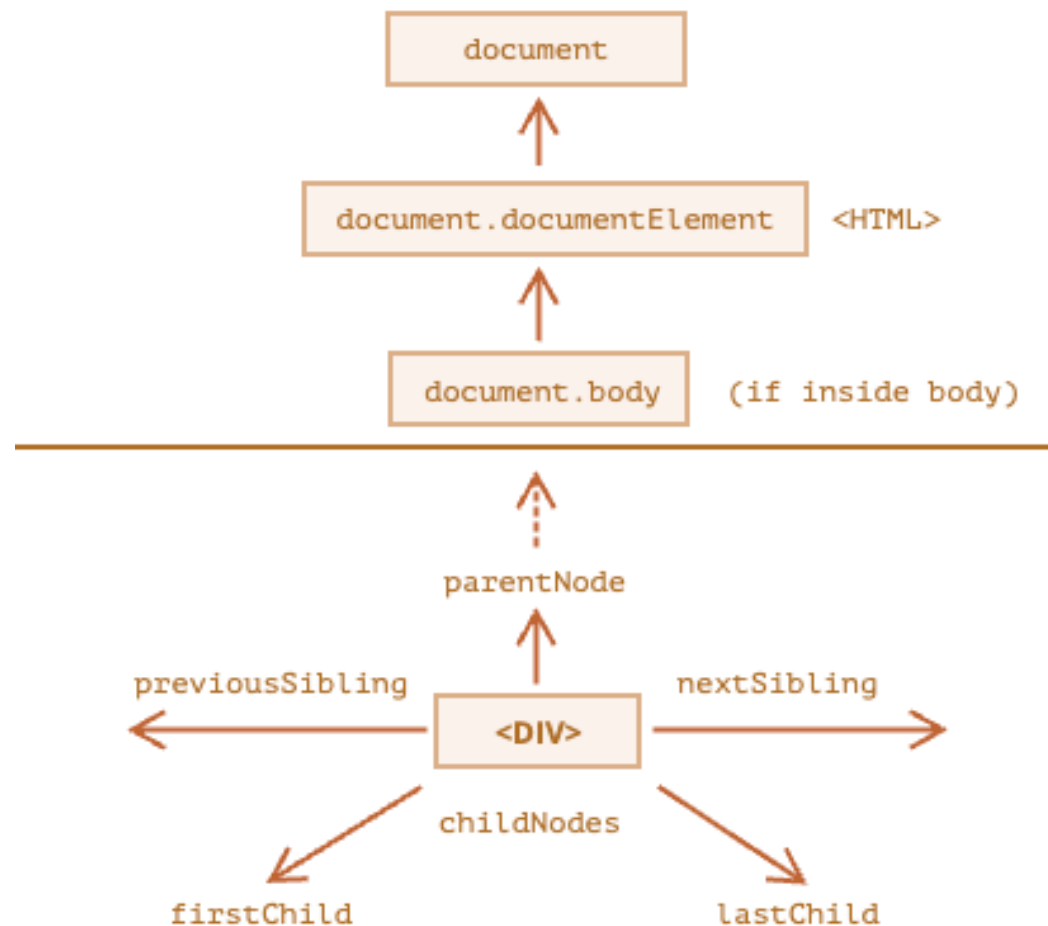
# textContent: pure text

- The textContent provides access to the text inside the element: only text, minus all <tags>.

```
1 <div id="news">
2   <h1>Headline!</h1>
3   <p>Martians attack people!</p>
4 </div>
5
6 <script>
7   // Headline! Martians attack people!
8   alert(news.textContent);
9 </script>
```

# Walking the DOM

- **Child nodes** (or children): elements that are direct children
- **Descendants** – all elements that are nested in the given one, including children, their children and so on.



# Children: childNodes

- The childNodes collection lists all child nodes, including text nodes.

```
1 <html>
2 <body>
3   <div>Begin</div>
4
5   <ul>
6     <li>Information</li>
7   </ul>
8
9   <div>End</div>
10
11  <script>
12    for (let i = 0; i < document.body.childNodes.length; i++) {
13      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL,
14    }
15  </script>
16  ...more stuff...
17 </body>
18 </html>
```

# Children: firstChild, lastChild

- Properties firstChild and lastChild give fast access to the first and last children
- If there exist child nodes, then the following is always true:

```
1 elem.childNodes[0] === elem.firstChild  
2 elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

# Modifying the document

- DOM modification is the key to creating “live” pages.
- Here we’ll see how to create new elements “on the fly” and modify the existing page content.
- To create DOM nodes, there are two methods:

```
document.createElement(tag)
```

Creates a new *element node* with the given tag:

```
1 let div = document.createElement('div');
```

```
document.createTextNode(text)
```

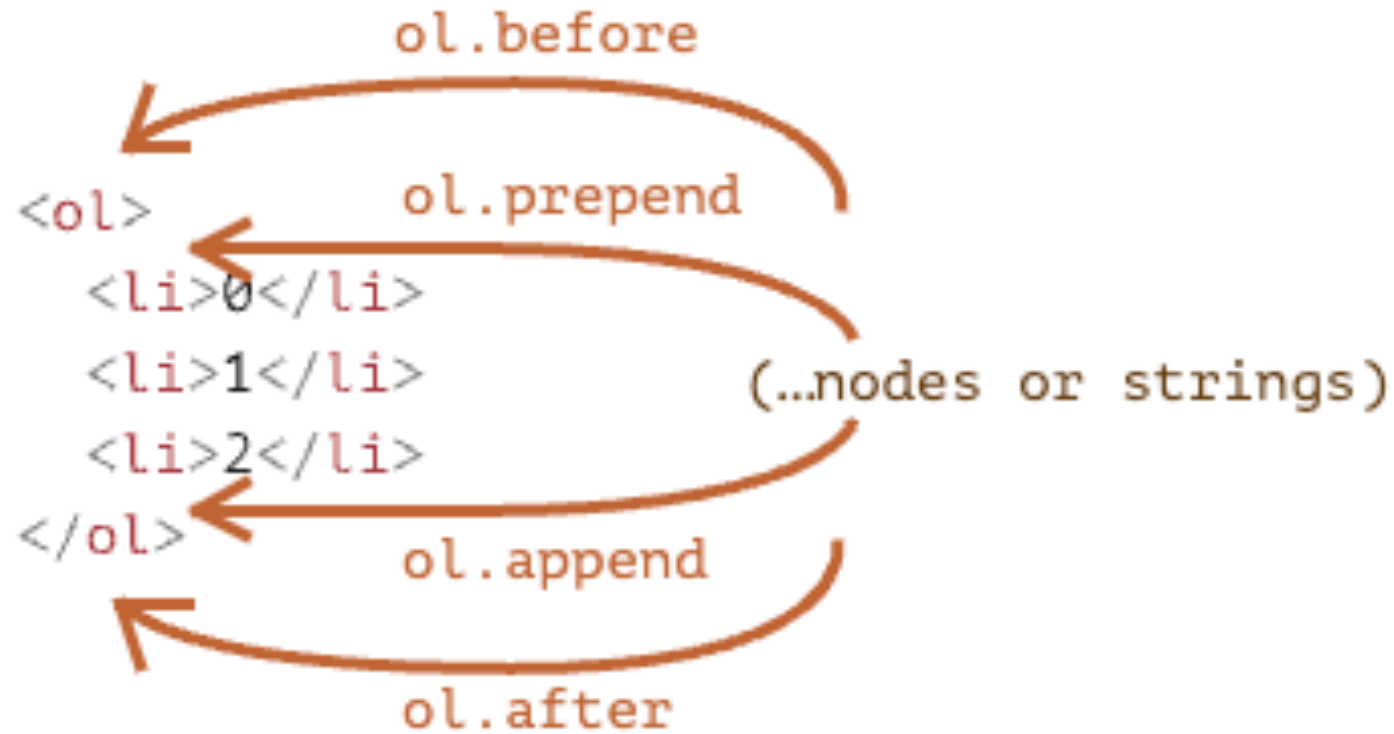
Creates a new *text node* with the given text:

```
1 let textNode = document.createTextNode('Here I am');
```

- Here are more insertion methods, they specify different places where to insert:
  - **node.append(...nodes or strings)** – append nodes or strings *at the end* of node,
  - **node.prepend(...nodes or strings)** – insert nodes or strings *at the beginning* of node,
  - **node.before(...nodes or strings)** – insert nodes or strings *before* node,
  - **node.after(...nodes or strings)** – insert nodes or strings *after* node,
  - **node.replaceWith(...nodes or strings)** – replace node with the given nodes or strings.

# Insertion methods

- Here's a visual picture of what the methods do:





# Modifying the document: Example

```
1 <style>
2 .alert {
3     padding: 15px;
4     border: 1px solid #d6e9c6;
5     border-radius: 4px;
6     color: #3c763d;
7     background-color: #dff0d8;
8 }
9 </style>
10
11 <script>
12     let div = document.createElement('div');
13     div.className = "alert";
14     div.innerHTML = "<strong>Hi there!</strong> You've read an important messag
15
16     document.body.append(div);
17 </script>
```



**HUST**

# Browser Object Model (BOM)

# Browser Object Model (BOM)

- There are no official standards for the Browser Object Model (BOM).
- Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

# Window Object

- The **window** object is supported by all browsers. It represents the browser's window.
- All global JavaScript objects, functions, and variables automatically become members of the window object.
  - Global variables are properties of the window object.
  - Global functions are methods of the window object.
  - Even the document object (of the HTML DOM) is a property of the window object

- Two properties can be used to determine the size of the browser window, both properties return the sizes in pixels:
  - `window.innerHeight` - the inner height of the browser window (in pixels)
  - `window.innerWidth` - the inner width of the browser window (in pixels)
- Other Window Methods:
  - `window.open()` - open a new window
  - `window.close()` - close the current window
  - `window.moveTo()` - move the current window
  - `window.resizeTo()` - resize the current window

- The window.screen object contains information about the user's screen.
  - The window.screen object can be written without the window prefix.
- Properties:
  - screen.width
  - screen.height
  - screen.availWidth
  - screen.availHeight
  - screen.colorDepth
  - screen.pixelDepth

# Window Location

- The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.
- [https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_loc\\_href](https://www.w3schools.com/js/tryit.asp?filename=tryjs_loc_href)
  - window.location.href returns the href (URL) of the current page
    - `https://www.w3schools.com/js/tryit.asp?filename=tryjs_loc_href`
  - window.location.hostname returns the domain name of the web host
    - `www.w3schools.com`
  - window.location.pathname returns the path of the current page:
    - `/js/tryit.asp`
  - window.location.protocol returns the web protocol used
    - `https`
  - window.location.assign() loads a new document

# Window History, Window Navigator

- Window Navigator:

- navigator.appVersion
- navigator.userAgent
- navigator.platform
- language
- onLine
- javaEnabled()
- ...

## Window History:

- history.back() - same as clicking back in the browser
- history.forward() - same as clicking forward in the browser





**HUST**

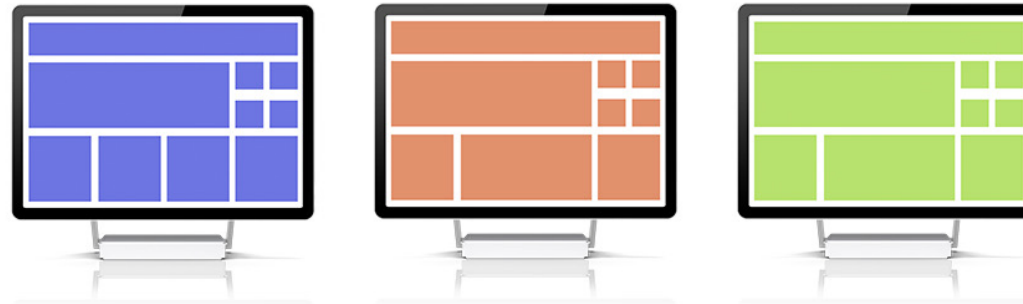
# Single-Page Application (SPA)

# Multi-Page Application vs Single-Page Application

- There are two general approaches to building web applications today:
  - Traditional web applications that perform most of the application logic on the server
  - Single-page applications (SPAs) that perform most of the user interface logic in a web browser, communicating with the web server primarily using web APIs

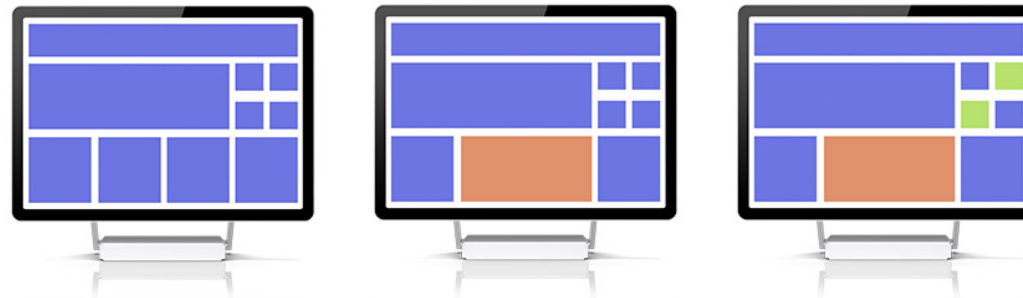
## Traditional

Every request for new information gives you a new version of the whole page.



## Single Page Application

You request just the pieces you need.



# What is a Single-Page Application?

- A multiple-page application
  - An app has multiple pages with static information (text, image, etc.)
  - Every change requests rendering a new page from the server => browser reloads the content of a page completely and downloads the resources again
- A single-page application
  - Loads only a single page, and then updates the body content of that single document via JavaScript APIs
  - Examples: Gmail, Google Maps, Tiki, etc.

- Faster performance: all the resources are loaded during one session, only the necessary data is changed
- Data caching: provides ability to work offline
- Improved user experience: the use of AJAX and JavaScript frameworks (like React, Angular) allows building a more flexible and responsive interface

# SPA Disadvantages



- Faster initial page load.
- MPAs are easy and good for SEO management.
- MPAs provide lots of analytics and data about how a website works.

- Slow performance: application reloads every time a user clicks on a new tab
- Hard to maintain: developers need to maintain each page separately and regularly
- More development time: application has a higher number of features compared to a SPA, so it requires more effort and resources

A graphic on the left side of the slide featuring a dark blue background with a large, stylized circular pattern of red dots. The dots are arranged in a way that creates a sense of depth and movement, resembling a spiral or a series of concentric circles that are slightly offset from each other.

# HUST

# REACT



# Create React app

- Create React App (CRA) is a comfortable environment for learning React, and is the best way to start building a new single-page application in React.
- It offers a setup with no configuration, let you focus on code, not build tools

```
npx clear-npx-cache  
npx create-react-app my-app  
cd my-app  
npm start
```

# Built-in commands

- `npm start`
  - Runs the app in development mode.
  - Open `http://localhost:3000` to view it in the browser.
  - The page will automatically reload if you make changes to the code.
  - You will see the build
- `npm run build`
  - Builds the app for production to the build folder.
  - It correctly bundles React in production mode and optimizes the build for the best performance.
  - The build is minified and the filenames include the hashes.
  - Your app is ready to be deployed.

# Adding Stylesheets

- CRA setup uses [webpack](#) for handling all assets.
- webpack offers a custom way of “extending” the concept of import beyond JavaScript

```
import React, { Component } from 'react';
import './Button.css'; // Tell webpack that Button.js uses these styles

class Button extends Component {
  render() {
    // You can use them as regular CSS styles
    return <div className="Button" />;
  }
}
```

# CRA Problems:

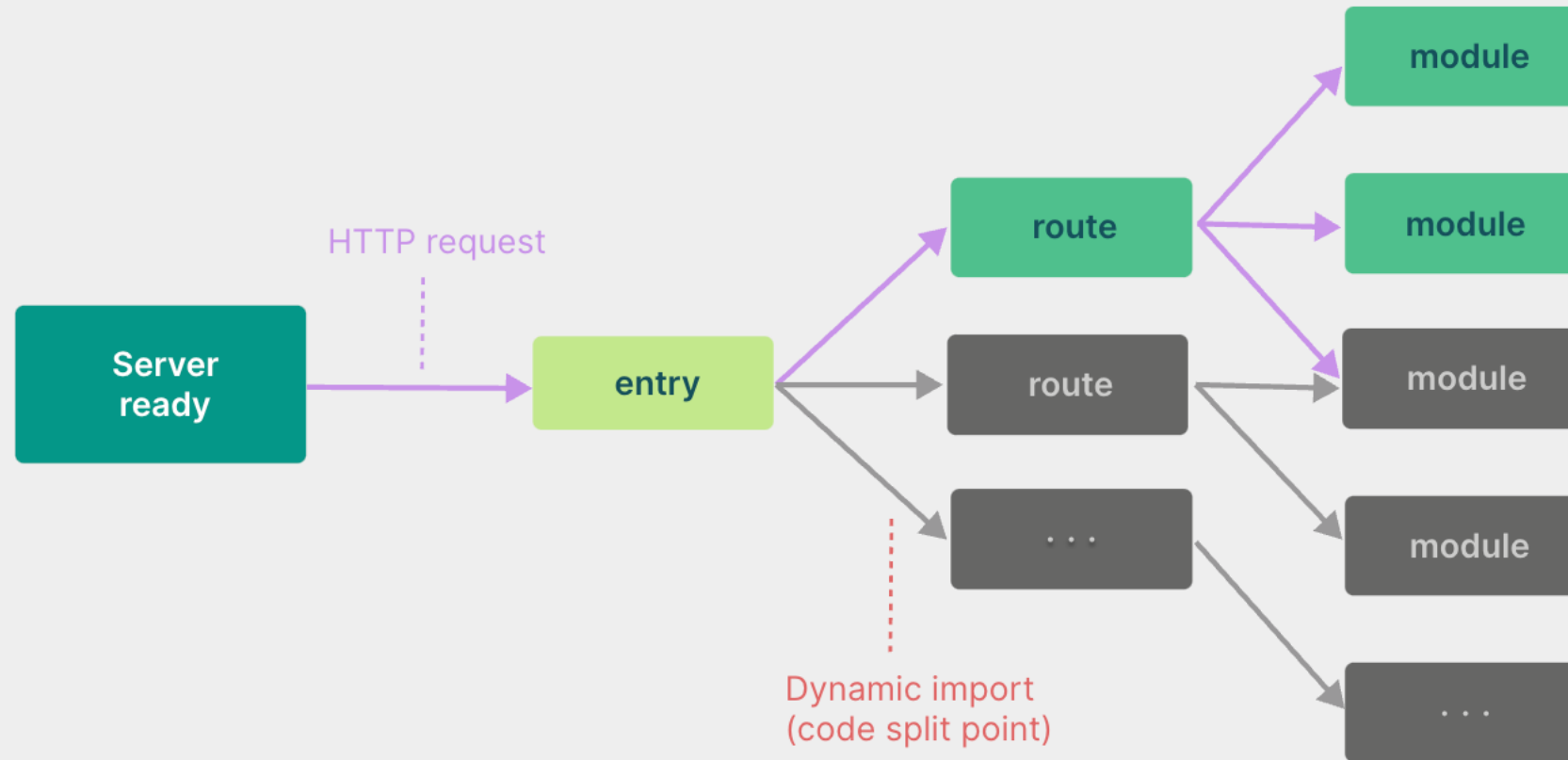
- as we build more and more ambitious applications, the amount of JavaScript we are dealing with is also increasing dramatically.
    - It is not uncommon for large scale projects to contain thousands of modules
  - We are starting to hit a performance bottleneck for JavaScript based tooling:
    - it can often take an unreasonably long wait (sometimes up to minutes!) to spin up a dev server
    - and even with Hot Module Replacement (HMR), file edits can take a couple of seconds to be reflected in the browser
- The slow feedback loop can greatly affect developers' productivity and happiness.

- aims to address those issues by leveraging new advancements in the ecosystem:
  - the availability of native ES modules in the browser,
  - and the rise of JavaScript tools written in compile-to-native languages.
- divides the modules in an application into two categories:
  - **Dependencies**
  - **Source code**

- **Dependencies:** are mostly plain JavaScript that do not change often during development. Some large dependencies (e.g. component libraries with hundreds of modules) are also quite expensive to process.
- → Vite pre-bundles dependencies using esbuild. esbuild is written in Go and pre-bundles dependencies 10-100x faster than JavaScript-based bundlers.

- **Source code:** often contains non-plain JavaScript that needs transforming (e.g. JSX, CSS or Vue/Svelte components), and will be edited very often. Also, not all source code needs to be loaded at the same time (e.g. with route-based code-splitting).
- Vite serves source code over native ESM.
  - This is essentially letting the browser take over part of the job of a bundler
  - Vite only needs to transform and serve source code on demand, as the browser requests it.
  - Code behind conditional dynamic imports is only processed if actually used on the current screen.

## Native ESM based dev server





- React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components.
- It is maintained by Meta (formerly Facebook) and a community of individual developers and companies.
- React can be used as a base in the development of single-page, mobile, or server-rendered applications with frameworks like Next.js.
- However, React is only concerned with state management and rendering that state to the DOM, so creating React applications usually requires the use of additional libraries for routing, as well as certain client-side functionality.

- JSX is a syntax extension to JavaScript.
- Similar in appearance to HTML, JSX provides a way to structure component rendering.
- React components are typically written using JSX, although they do not have to be (components may also be written in pure JavaScript).

```
const element = <h1>Hello, world!</h1>;
```

# JSX Represents Objects

- Babel compiles JSX down to `React.createElement()` calls.
  - These two examples are identical:

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
);
```

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hello, world!'  
);
```

# JSX Represents Objects

- `React.createElement()` essentially creates an object like this:
- These objects are called “React elements”.
- You can think of them as descriptions of what you want to see on the screen.
- React reads these objects and uses them to construct the DOM and keep it up to date.

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

# Components and Props

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation
- Conceptually, components are like JavaScript functions.
  - They accept arbitrary inputs (called “props”)
  - and return React elements describing what should appear on the screen.
  - note that props are Read-Only
- You can also use an ES6 class to define a component:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

# State and Lifecycle

- Clock (class component) with this.state.date

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>  
      </div>  
    );  
  }  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Clock />);
```

# State and Lifecycle

- Adding Lifecycle Methods to a Class
  - We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called “mounting” in React.
  - We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called “unmounting” in React.
- We can declare special methods on the component class to run some code when a component mounts and unmounts:

```
componentDidMount() {  
  this.timerID = setInterval(  
    () => this.tick(),  
    1000  
  );  
}
```

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

# State and Lifecycle

1. When `<Clock />` is passed to `root.render()`, React calls the constructor of the Clock component
  - it initializes `this.state` with an object including the current time
2. React then calls Clock component's `render()` method
  - This is how React learns what should be displayed on the screen
3. When the Clock output is inserted in the DOM, React calls the `componentDidMount()` lifecycle method.
4. Every second the browser calls the `tick()` method.
5. If the Clock component is ever removed from the DOM, React calls the `componentWillUnmount()` lifecycle method so the timer is stopped.

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

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

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

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```



# Composing Components

- Components can refer to other components in their output.
  - This lets us use the same component abstraction for any level of detail

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
function App() {  
  return (  
    <div>  
      <Welcome name="Sara" />  
      <Welcome name="Cahal" />  
      <Welcome name="Edite" />  
    </div>  
  );  
}
```

# Handling Events

- Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:
  - React events are named using camelCase, rather than lowercase.
  - With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>  
  Activate Lasers  
</button>
```



# Summary

A usual React component comprises three parts:

- structures: html
  - represented by JSX
- styles: css
  - how it is expressed depends on the styling solution.
- behaviours: js
  - state variables
  - event handlers

```
import * as React from 'react';
import Box from '@mui/material/Box';
import TextField from '@mui/material/TextField';

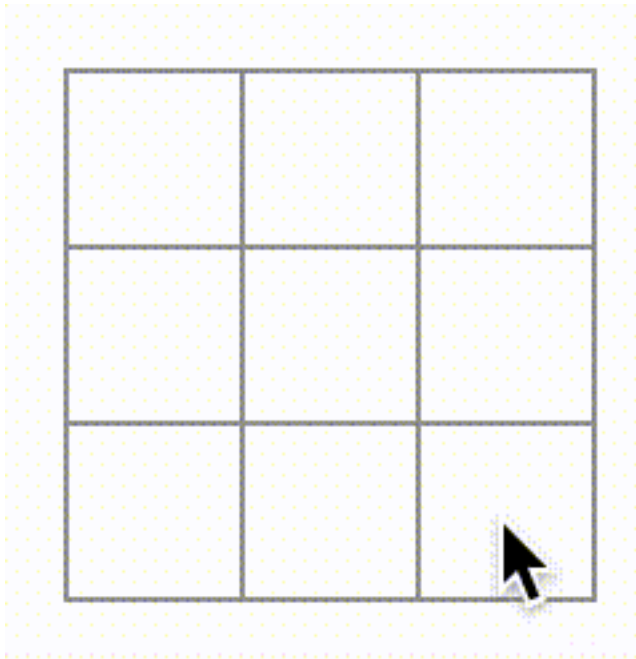
export default function StateTextFields() {
  const [name, setName] = React.useState('Cat in the Hat');
  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setName(event.target.value);
  };

  return (
    <Box
      component="form"
      sx={{
        '& > :not(style)': { m: 1, width: '25ch' },
      }}
      noValidate
      autoComplete="off"
    >
      <TextField
        id="outlined-name"
        label="Name"
        value={name}
        onChange={handleChange}
      />
      <TextField
        id="outlined-uncontrolled"
        label="Uncontrolled"
        defaultValue="foo"
      />
    </Box>
  );
}
```

# Ex1. Tic Tac Toe

Link: <https://react.dev/learn/tutorial-tic-tac-toe>

```
npx clear-npx-cache  
npx create-react-app tic-tac-toe  
cd tic-tac-toe  
npm start
```



Winner: X

|   |   |   |
|---|---|---|
| X |   | O |
|   | X | O |
|   |   | X |

1. Go to game start
2. Go to move #1
3. Go to move #2
4. Go to move #3
5. Go to move #4
6. Go to move #5

## Ex2: React-101

A front end that fetches & displays user info from:  
<https://jsonplaceholder.typicode.com/users>

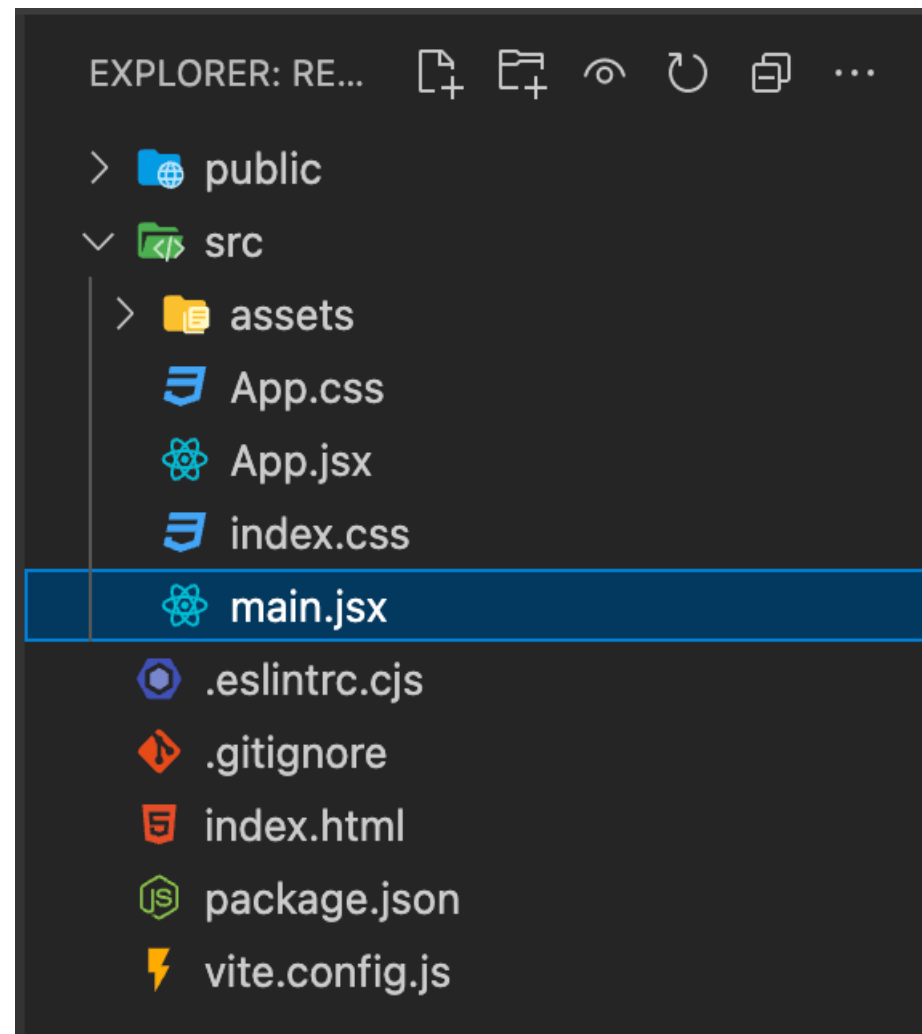
Welcome to React 101!

|          |                     |
|----------|---------------------|
| UserId   | 2                   |
| Username | Antonette           |
| Name     | Ervin Howell        |
| Email    | Shanna@melissa.tv   |
| Phone    | 010-692-6593 x09125 |
| Website  | anastasia.net       |

## Ex2: React-101 (2)

### Init project using Vite

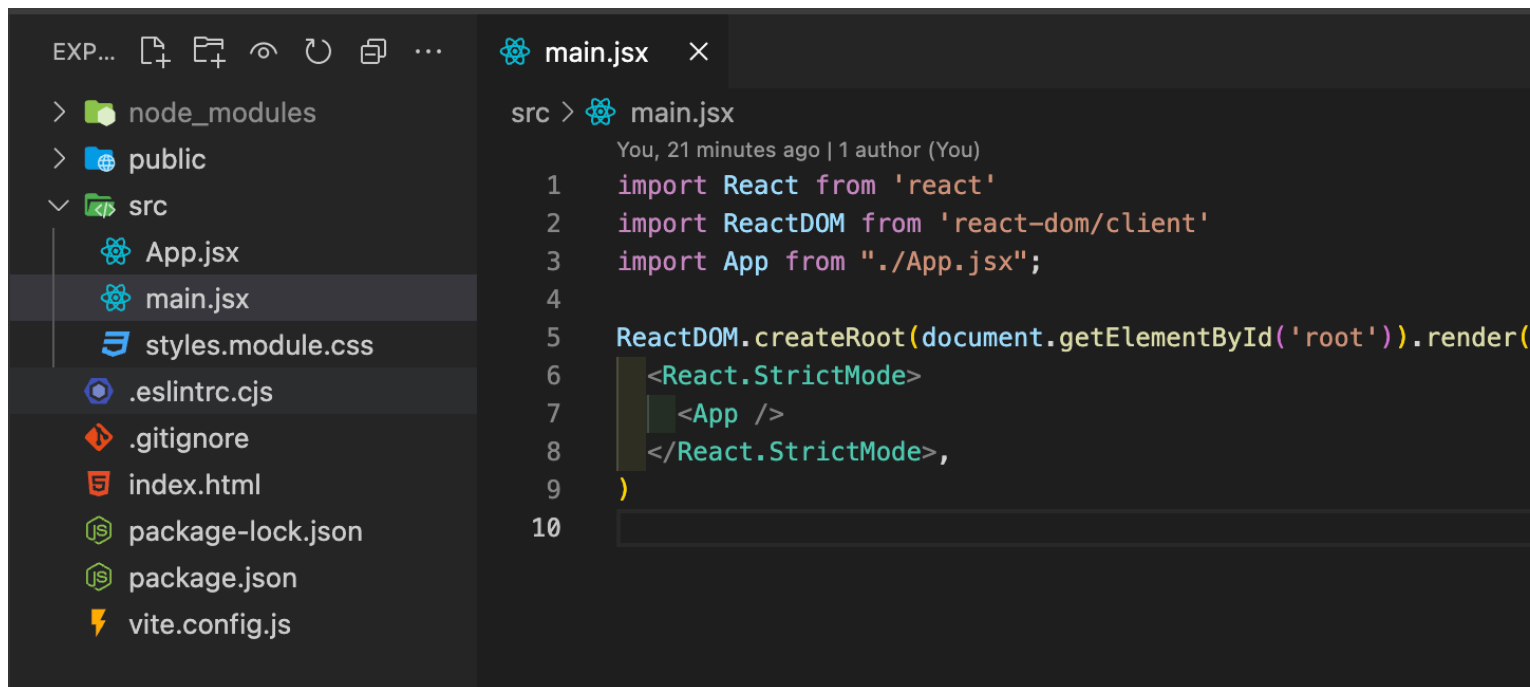
```
npm create vite@latest react-101 -- --template react
cd react-101
npm install
npm run dev
```



# Ex2: React-101 (3)

## Install MUI:

- `npm install @mui/material @emotion/react @emotion/styled @mui/icons-material`



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a `src` directory containing `App.jsx`, `main.jsx`, and `styles.module.css`. The code editor shows the content of `main.jsx`, which imports `React` and `ReactDOM` from `react` and `react-dom/client` respectively, and imports `App` from `./App.jsx`. The code then uses `ReactDOM.createRoot` to render the `App` component into the `root` element.

```
src > main.jsx
You, 21 minutes ago | 1 author (You)
1  import React from 'react'
2  import ReactDOM from 'react-dom/client'
3  import App from './App.jsx';
4
5  ReactDOM.createRoot(document.getElementById('root')).render(
6    <React.StrictMode>
7      <App />
8    </React.StrictMode>,
9  )
10
```

## Ex2: React-101 (4)

## Update App.jsx:

1. Define state variables
2. Event handlers
3. Fetch data function
4. Fetch data when userId has changed

```
function App() {  
  const [userId, setUserId] = useState(1);  
  const [userInfo, setUserInfo] = useState(null);  
  
  function nextUser() {  
    setUserId(userId - 1);  
  }  
  
  function previousUser() {  
    setUserId(userId + 1);  
  }  
  
  async function fetchUserInfo() {  
    try {  
      const baseUrl = "https://jsonplaceholder.typicode.com/users";  
      const response = await axios.get(`${baseUrl}/${userId}`);  
      console.log(response.data);  
      setUserInfo(response.data);  
    } catch (error) {  
      console.error(error);  
      alert("Failed to load user info!");  
    }  
  }  
  
  useEffect(() => {  
    fetchUserInfo();  
  }, [userId]);  
  
  return (  
    <>  
    <CssBaseline />  
    <Container className={styles.container}>
```



## Ex2: React-101 (5)

Build UI using MUI:

1. Previous user button
2. Table show user info
3. Next user button

```
<Container className={styles.container}>
  <Grid container className={styles.gridContainer}>
    <Grid item xs={2} className={styles.navigator}>
      <IconButton onClick={nextUser} disabled={userId === MIN_USER_ID}>
        <NavigateBeforeIcon />
      </IconButton>
    </Grid>
    <Grid item xs={8}>
      <Typography variant="h4" align="center" gutterBottom>
        Welcome to React 101!
      </Typography>
      <Paper className={styles.paper}>
        <TableContainer>
          <Table>
            <TableRow>
              <TableCell style={{ width: "20%" }}>UserId</TableCell>
              <TableCell>{userId}</TableCell>
            </TableRow>
            <TableRow>
              <TableCell>Username</TableCell>
              <TableCell>{userInfo.username}</TableCell>
            </TableRow>
            <TableRow> ...
          </Table>
        </TableContainer>
      </Paper>
    </Grid>
    <Grid item xs={2} className={styles.navigator}>
      <IconButton onClick={previousUser} disabled={userId === MAX_USER_ID}>
        <NavigateNextIcon />
      </IconButton>
    </Grid>
  </Grid>
</Container>
```

- [https://www.w3schools.com/js/js\\_htmlDOM.asp](https://www.w3schools.com/js/js_htmlDOM.asp)
- <https://javascript.info/>
- [https://www.w3schools.com/js/js\\_window.asp](https://www.w3schools.com/js/js_window.asp)
- <https://hygger.io/blog/mpa-vs-spa-traditional-web-apps-or-single-page-applications/>
- <https://create-react-app.dev/docs/getting-started/>
- <https://reactjs.org/docs/getting-started.html>

