

# Web Technologies

CAP 756

Presenter: Dr. Mirza Shuja

# Babel

- Many software languages require you to compile your source code.
- JavaScript is an interpreted language: the browser interprets the code as text, so there's no need to compile JavaScript.
- However, not all browsers support the latest JavaScript syntax, and no browser supports JSX syntax.
- Since we want to use the latest features of JavaScript along with JSX, we're going to need a way to convert our fancy source code into something that the browser can interpret.
- This process is called compiling, and it's what Babel is designed to do.
- **BabelJS** is a JavaScript transpiler which transpiles new features into old standard.
- With this, the features can be run on both old and new browsers, hassle-free.

# Why BabelJS?

- JavaScript is the language that the browser understands.
- We use different browsers to run our applications – Chrome, Firefox, Internet Explorer, Microsoft Edge, Opera, UC browser etc.
- ECMA Script is the JavaScript language specification; the ECMA Script 2015 ES6 is the stable version which works fine in all new and old browsers.
- After ES5, we have had ES6, ES7, and ES8.
- ES6 released with a lot of new features which are not fully supported by all browsers.
- The same applies to ES7, ES8 and ESNext (next version of ECMA Script).
- It is now uncertain when it will be possible for all browsers to be compatible with all the ES versions that released.

- In case we plan to use ES6 or ES7 or ES8 features to write our code it will tend to break in some old browsers because of lack of support of the new changes.
- Therefore, if we want to use new features of ECMA Script in our code and want to run it on all possible browsers available, we need a tool that will compile our final code in ES5.
- **Babel** does the same and it is called a **transpiler** that **transpiles** the code in the ECMA Script version that we want.
- It has features like **presets** and **plugins**, which configure the ECMA version we need to transpile our code.
- With Babel, developers can write their code using the new features in JavaScript.
- The users can get the codes **transpiled** using **Babel**; the codes can later be used in any browsers without any issues.

- There are many ways of working with Babel. The easiest way to get started is to include a link to the **Babel CDN** directly in your HTML, which will compile any code in script blocks that have a type of “text/babel.”
- Babel will compile the source code on the client before running it. Although this may not be the best solution for production, it’s a great way to get started with JSX:
- To work with BabelJS we need following setup:
  1. NodeJS
  2. Npm
  3. Babel-CLI
  4. Babel-Preset
  5. IDE for writing code

- Babel comes with a built-in command line interface, which can be used to compile the code.
- We have to use *npm init* to create the project.
- **babel-cli**
- Execute the following command to install babel-cli
- **`npm install --save-dev babel-cli`**
- **babel-preset**
- Execute the following command to install babel-preset
- **`npm install --save-dev babel-preset-env`**
- **babel-core**
- Execute the following command to install babel-core
- **`npm install --save-dev babel-core`**

# WebPack

- Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset.
- **Install**-Install with npm:
- **`npm install --save-dev webpack`**
- **Plugins**
- Webpack has a **rich plugin interface**. Most of the features within webpack itself use this plugin interface. This makes webpack very **flexible**.
- **Loaders**
- Webpack enables the use of loaders to preprocess files. This allows you to bundle any static resource way beyond JavaScript. You can easily write your own loaders using Node.js.
- Loaders are activated by using **loadername!** prefixes in `require()` statements, or are automatically applied via regex from your webpack configuration.

- A perfect way to say why webpack exists is the quote “**necessity** is the mother of **invention**”.
- But to understand it better we need to go back, way back, to when JavaScript was not the new thing, in those old timey ages when a website was just a small bundle of good old .html, CSS, and probably one or a few JavaScript files in some cases. But very soon all of this was going to change.
- **What was the problem?**
- The entire dev community was involved in a constant quest of improving the overall user and developer experience around using and building javascript/web applications. Therefore, we saw a lot of new **libraries and frameworks** introduced.
- A few **design patterns** also evolved over time to give developers a better, more powerful yet very simple way of writing complex JavaScript applications



- Websites before were no more just a small package with an odd number of files in them.
- They started getting bulky, with the introduction of **JavaScript modules**, as writing encapsulated small chunks of code was the new trend.
- Eventually all of this led to a situation where we had 4x or 5x of files in the overall application package.
- **Not only was the overall size of the application a challenge**, but also there was a huge gap in the kind of code developers were writing and the kind of code browsers could understand.
- Developers had to use a lot of helper code called **polyfills** to make sure that the browsers were able to interpret the code in their packages.
- To answer these issues, webpack was created. **Webpack is a static module bundler.**

# So how was Webpack the answer?

- In brief, Webpack goes through your package and creates what it calls a **dependency graph** which consists of various **modules** which your webapp would require to function as expected.
- Then, depending on this graph, it creates a new package which consists of the very bare minimum number of files required, often just a single **bundle.js** file which can be plugged in to the html file easily and used for the application.

- **Installation Phase**
- **npm int**
- **npm init**
- This will create a starter package and add a package.json file for us. This is where all the dependencies required to build this application will be mentioned.
- Now for creating a simple React application, we need two main libraries: React and ReactDOM. So let's get them added as dependencies into our application using npm.
- **npm i react react-dom --save**
- Next up we need to add webpack, so we can bundle our app together. Not only bundle, but we will also require hot reloading which is possible using the webpack dev server.
- **npm i webpack webpack-dev-server webpack-cli --save--dev**

- The **--save--dev** is to specify that these modules are just dev dependencies.
- Now since we are working with React, we must keep in mind that React uses ES6 classes and import statements, something that all the browsers may not be able to understand.
- To make sure that the code is readable by all browsers, we need a tool like babel to transpile our code to normal readable code for browsers.
- *\$ npm i babel-core babel-loader @babel/preset-react @babel/preset-env html-webpack-plugin --save-dev*
- Moving on, let's add some code and let's get the webpack configuration started.

- **The Code:** Let's start by adding a webpack.config.js file in the root of our application structure. Add the following code in your webpack.config file.

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  //This property defines where the application starts
  entry: './src/index.js',

  //This property defines the file path and the file name
  //which will be used for deploying the bundled file
  output: {
    path: path.join(__dirname, '/dist'),
    filename: 'bundle.js'
  },
  //Setup loaders
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  },
  // Setup plugin to use a HTML file for serving bundled js files
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ]
}
```

- **React Code**

- Since the starting point for the application is the index.js file in src folder, let's start with that. We will start by requiring both React and ReactDOM for our use in this case. Add the below code in your **index.js** file.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './Components/App';
ReactDOM.render(<App />, document.getElementById('app'));
```

- So we simply import another file from our components folder, which you will create, and add another file in the folder called App.js. So let's see what's inside the **App.js** file:

```
import React, { Component } from 'react'
class App extends Component {
  render() {
    return (
      <div>
        <h1>Webpack + React setup</h1>
      </div>
    )
  }
}
export default App;
```

- We are almost done. The only thing left now is to enable hot reloading. This means that every time a change is detected, the browser auto reloads the page and has the ability to build and bundle the entire application when the time comes.
- We can do this by adding script values in the **package.json** file. Remove the test property in the scripts object of your **package.json** file and add these two scripts:

**"start": "webpack-dev-server --mode development --open --hot",**

**"build": "webpack --mode production"**

# React Router

- Routing is a process in which a user is directed to different pages based on their action or request.
- ReactJS Router is mainly used for developing Single Page Web Applications.
- React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.
- React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package.
- It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.



# Need of React Router

- React Router plays an important role to display multiple views in a single page application.
- Without React Router, it is not possible to display multiple views in React applications.
- Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

## React Router Installation

- React contains three different packages for routing.
- These are:

**1.react-router:** It provides the core routing components and functions for the React Router applications.

**2.react-router-native:** It is used for mobile applications.

**3.react-router-dom:** It is used for web applications design.

- It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application.
- The below command is used to install react router dom.
- **npm install react-router-dom**

### ❖ Components in React Router

- There are two types of router components:
  1. **<BrowserRouter>**: It is used for handling the dynamic URL.
  2. **<HashRouter>**: It is used for handling the static request.

- Example

- **Step-1:** In our example, we will create two more components along with **App.js**, which is already present. **About.js & Contact.js**

#### About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

#### Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact</h1>
  }
}
export default Contact
```

#### App.js

```
import React from 'react'
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Home</h1>
      </div>
    )
  }
}
export default App
```

- **Step-2:** For Routing, open the index.js file and import all the three component files in it.
- Here, you need to import line:
- **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'**
- which helps us to implement the Routing.
- **What is Route?**
- It is used to define and render component based on the specified path.
- It will accept components and render to define what should be rendered.

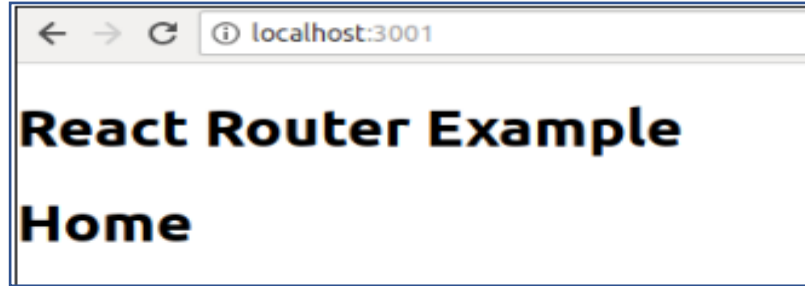
- Now, our **index.js** file looks like below.

### Index.js

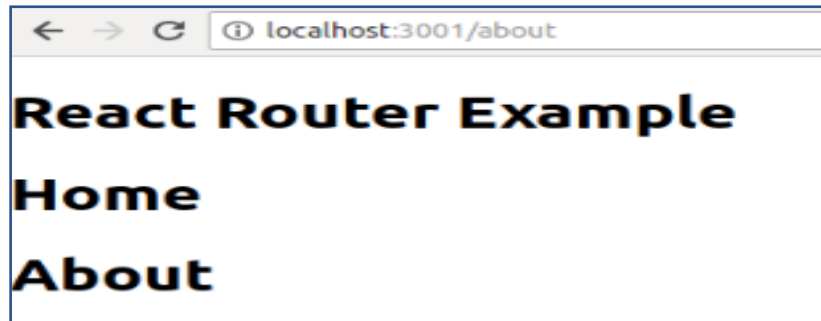
```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

- **Step-3:** Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



- Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



- **Step-4:** In the above screen, you can see that **Home** component is still rendered. It is because the home path is '/' and about path is '/**about**', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

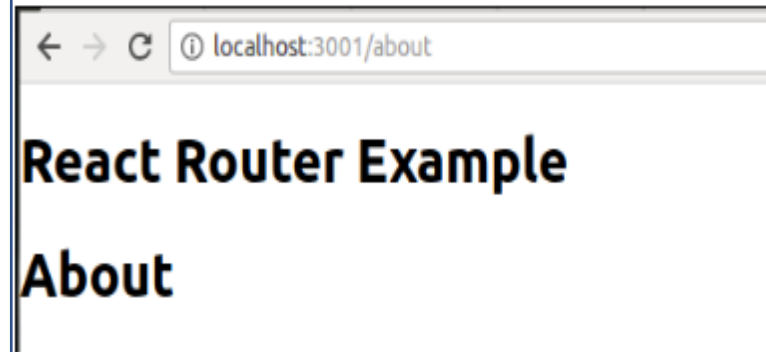
- Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom';
import './index.css';
import App from './App';
import About from './about';
import Contact from './contact';

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)

ReactDOM.render(routing, document.getElementById('root'));
```

## Output





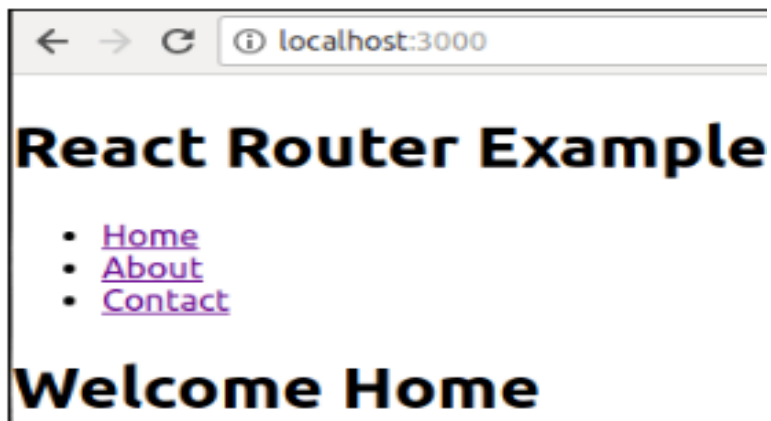
## Adding Navigation using Link component

- Sometimes, we want to need **multiple** links on a single page.
- When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page.
- To do this, we need to import **<Link>** component in the **index.js** file.
- What is **< Link>** component?
- This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

## Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
```

## Output



```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

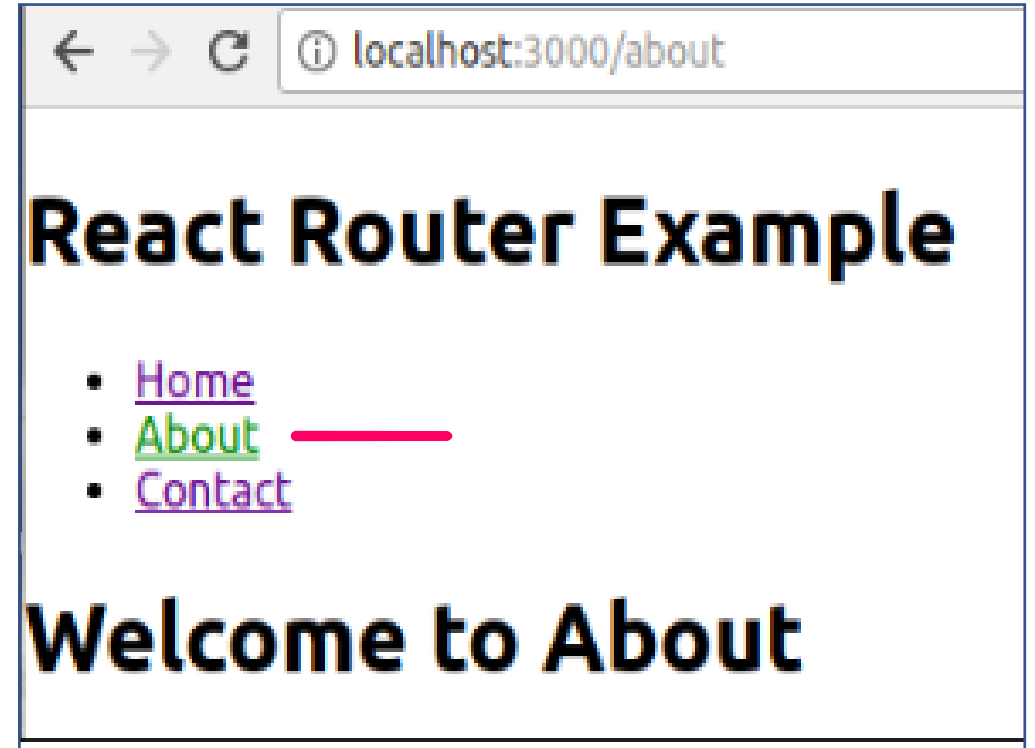
- Now, we need to add some **styles** to the Link.
- So that when we click on any particular link, it can be easily **identified** which Link is **active**.
- To do this react router provides a new trick **NavLink** instead of **Link**.
- Now, in the **index.js** file, replace Link from NavLink and add properties **activeStyle**.
- The activeStyle properties mean when we click on the Link, it should have a specific style so that we can differentiate which one is currently active.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
```

```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
            {color:'green'}
          }>About</NavLink>
```

```
        </li>
      </ul>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

- Output



# React Router Switch

- The **<Switch>** component is used to render components only when the path will be **matched**. Otherwise, it returns to the **not found** component.
- To understand this, first, we need to create a **notfound** component.

notfound.js

```
import React from 'react'  
  
const Notfound = () => <h1>Not found</h1>  
  
export default Notfound
```

- Now, import component in the index.js file as shown in following code

## Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import NotFound from './notfound'
```

```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}
          }>Home</NavLink>
        </li>
```

```
      <NavLink to="/about" exact activeStyle={
        {color:'green'}
      }>About</NavLink>
    </li>
    <li>
      <NavLink to="/contact" exact activeStyle={
        {color:'magenta'}
      }>Contact</NavLink>
    </li>
  </ul>
  <Switch>
    <Route exact path="/" component={App} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
    <Route component={NotFound} />
  </Switch>
</div>
</Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

# React Router <Redirect>

- A **<Redirect>** component is used to redirect to another route in our application to maintain the old URLs. It can be placed anywhere in the route hierarchy.
- **Nested Routing in React**
- Nested routing allows you to render **sub-routes** in your application.
- The following example explains it.



index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import {BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import Notfound from './notfound'
```

```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
            {color:'green'}
          }>About</NavLink>
        </li>
```

<li>

```
      <NavLink to="/contact" exact activeStyle={
        {color:'magenta'}
      }>Contact</NavLink>
    </li>
  </ul>
  <Switch>
```

</li>

</ul>

<Switch>

```
    <Route exact path="/" component={App} />
```

```
    <Route path="/about" component={About} />
```

```
    <Route path="/contact" component={Contact} />
```

```
    <Route component={Notfound} />
```

</Switch>

</div>

</Router>

)

```
ReactDOM.render(routing, document.getElementById('root'));
```

- In the contact.js file, we need to import the React Router component to implement the subroutes.


contact.js

```
import React from 'react'
import { Route, Link } from 'react-router-dom'

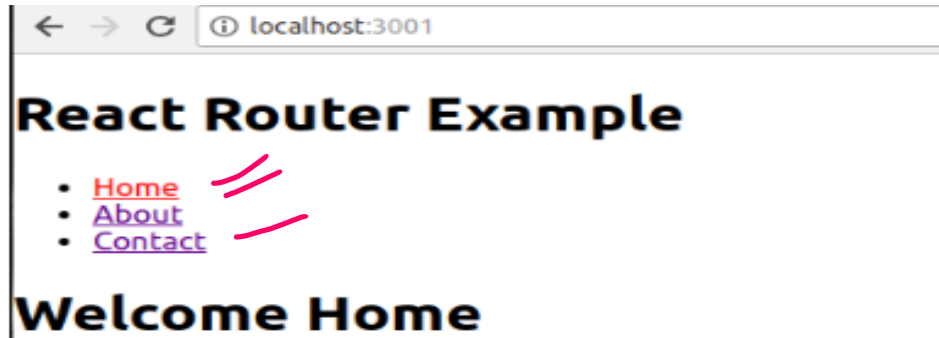
const Contacts = ({ match }) => <p>{match.params.id}</p>

class Contact extends React.Component {
  render() {
    const { url } = this.props.match
    return (
      <div>
        <h1>Welcome to Contact Page</h1>
        <strong>Select contact Id</strong>
        <ul>
          <li>
            <Link to="/contact/1">Contacts 1 </Link>
          </li>
          <li>
            <Link to="/contact/2">Contacts 2 </Link>
          </li>
          <li>
            <Link to="/contact/3">Contacts 3 </Link>
          </li>
          <li>
            <Link to="/contact/4">Contacts 4 </Link>
          </li>
        </ul>
        <Route path="/contact/:id" component={Contacts} />
      </div>
    )
  }
}

export default Contact
```



- **Output**
- When we execute the above program, we will get the following output.



- After clicking the **Contact** link, we will get the contact list. Now, selecting any contact, we will get the corresponding output. As shown below.

