

Babel :-

Today we have modern JS and so we can access its modern features (like const, let keywords came up in ES6, which is the newer version of JS). Now, if the end-user's browser is too old, it will not understand these features. So, there is a replacement code for these functionalities that is compatible with the older versions of a browser. When we use browserslist in our package.json, our code is converted into older code too.

This conversion is done by **Babel** (and it takes help of browserslist to know which versions of browsers should our app support). Babel is JS library/package, i.e. it is a piece of JS code that takes some js code as input and converts it into another piece of js code. So, the task of Babel is actually to convert code written ECMAScript2015+ to backwards compatible JS. That's why it's called a **JS transpiler**. See more about it from the link below :-

<https://medium.com/swlh/the-role-of-babel-in-react-dbcf78c69125>

Changes in scripts in package.json

Earlier we were using "npx parcel index.html" to run our app on a server, instead, we can modify our package.json by creating a new script named "start" (known as start script) in package.json and keeping its value as "parcel index.html".

So, earlier we used to do :-

```
C:\Users\Arpan Kesh\Desktop\Development\Namaste_React_Akshay_Saini\Namaste_React_Codes\Day 3\Part_1>npx parcel index.html
Server running at http://localhost:1234
✓ Built in 1.17s
```

Now, in the package.json, do :-

```
1  "scripts": {
2    "test": "jest",
3    "start": "parcel index.html"
4  },
```

And, in the command line, write :- **"npm run start"**

```
C:\Users\Arpan Kesh\Desktop\Development\Namaste_React_Akshay_Saini\Namaste_React_Codes\Day 3\Part_1>npm run start
> part_1@1.0.0 start
> parcel index.html
Server running at http://localhost:1234
✓ Built in 1.23s
```

However we can also write

- ⇒ **"start" : "npx parcel index.html"** in the package.json, and
- ⇒ **npm start** in the cmd to start the server

Similarly, to build the script, we can modify the code by writing

- ⇒ **"build" : "npx parcel build index.html"** OR **"parcel build index.html"** in the package.json, and
- ⇒ **npm run build** in the cmd to start the server

```
"scripts": {
  "test": "jest",
  "start": "npx parcel index.html",
  "build": "npx parcel build index.html"
```

At last, our package.json should look like :-

babel-Plugin-transform-remove-console :-

In the last class, we got to know that Parcel removes the console logs from the js file. But it was wrong. (You can go to one of the js files in the dist [not the map js files], and search for console.log and it will be there). It can be removed with the help of another package named **babel-plugin-transform-remove-console**. You can read about it from either the

- Babel docs :- [Link](#)
- npmjs docs of that plug-in :- [Link](#)

Install it using “ **npm install babel-plugin-transform-remove-console --save-dev**”:-

Installation

```
Shell Copy  
npm install babel-plugin-transform-remove-console --save-dev
```

Now, it will not automatically remove the console logs. We need to configure it first. Read about its usage in the npmjs docs too. In that doc, it is mentioned to write a js object in a .babelrc file, which is a babel configuration file.

Usage

Via .babelrc (Recommended)

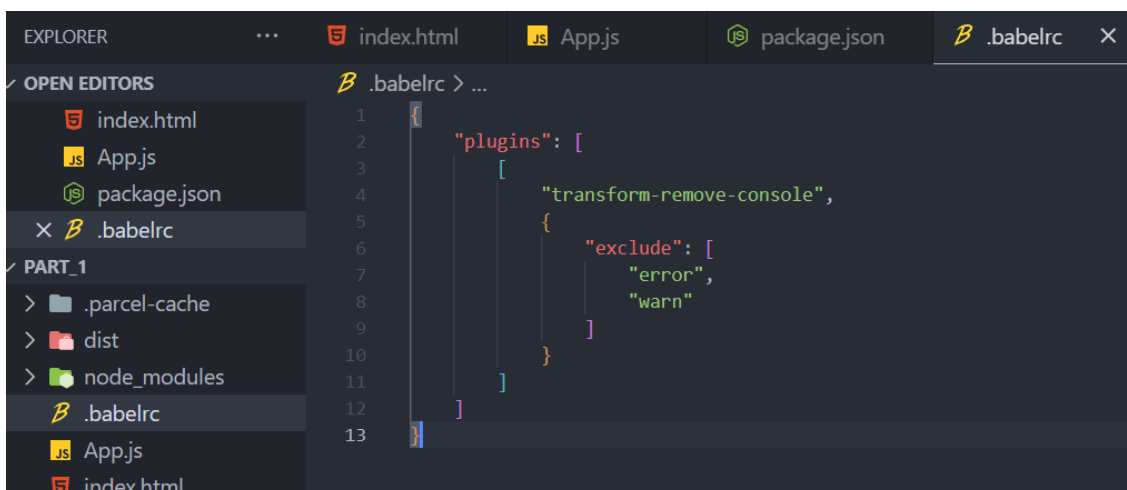
.babelrc

```
// without options  
{  
  "plugins": ["transform-remove-console"]  
}
```

```
// with options  
{  
  "plugins": [ [ "transform-remove-console", { "exclude": [ "error", "warn" ] } ] ]  
}
```

The first one means that it will remove all types of consoles from our code in the build. The second ones means that it will do the same thing as earlier but will not remove the console.error()'s and console.warning()'s. We can try out including other options too. So,

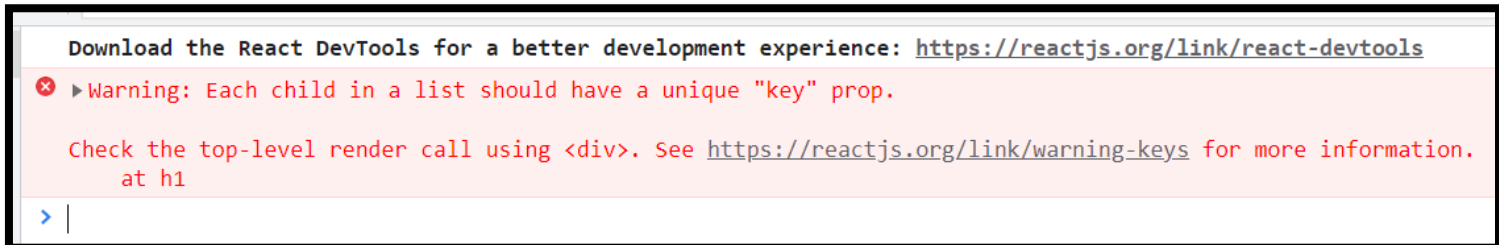
- create a .babelrc file in the root directory and write the second plugin from the above image.



- Then, delete the dist folder and run the build command “**npm run build**” (Deleting the dist folder is not necessary, but it is the best practise because dist contains many temporary build files from earlier builds, so before creating a new build, it is better to remove the earlier ones).
- Now, again 5 files will be created in a newly formed dist folder. Check the .js file (not the map one) and search for console.log() in it using Ctrl+F. There will be none. However, there will still be console.error() and console.warn().

Keys :-

Now, we are still getting this warning :-



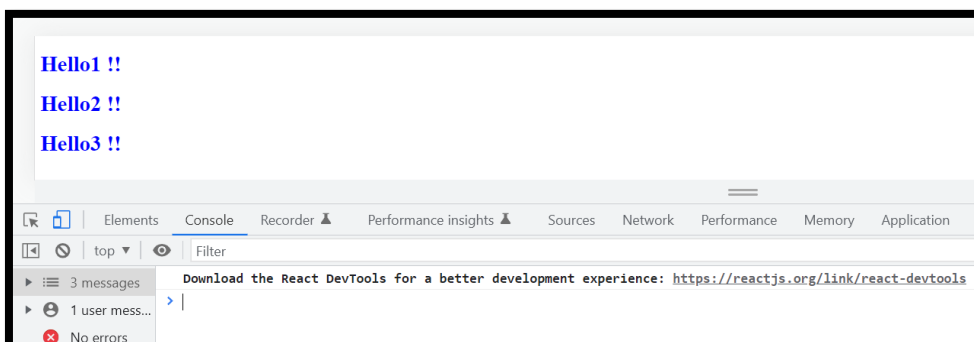
This happens because in the div named container, we have multiple children in the array of the children section of the React.createElement() :- heading1, heading2, heading3. But for all those headings we have not given any key in their respective props.

```
1 const heading1 = React.createElement("h1", { id: "title" }, "Hello1 !!");
2 const heading2 = React.createElement("h1", { id: "title" }, "Hello2 !!");
3 const heading3 = React.createElement("h1", { id: "title" }, "Hello3 !!");
4
5 const container = React.createElement("div", { id: "container" }, [heading1, heading2, heading3]);
```

To remove this warning, give a key to each child in their props and the key should be unique to each.

```
1 const heading1 = React.createElement("h1", { id: "title", key: "h1" }, "Hello1 !!");
2 const heading2 = React.createElement("h1", { id: "title", key: "h2" }, "Hello2 !!");
3 const heading3 = React.createElement("h1", { id: "title", key: "h3" }, "Hello3 !!");
4
5 const container = React.createElement("div", { id: "container" }, [heading1, heading2, heading3]);
```

Now, that warning will be no more.



So why Keys are needed? :-

When we are using React's render() functions, it creates a tree of React elements. On the next update of our code, the render() will return a different tree of React elements, but it does it efficiently by updating only those things that are different from the most recent tree. This concept is a part of another broad concept of React known as Reconciliation. Now, from the below link read the motivation, Recursing on Children, Keys (you can also go through the other concepts to learn new things) :- [Link](#)

(If link does not work search in google for "React reconciliation Keys")

#NOTE :-

To see the console.logs for our own project, I have removed the remove-console plugin from the .babelrc.

What does React.createElement() do?

It creates a React element and returns us an object which is then converted into HTML and inserted into DOM with the help of ReactDOM.

Why use JSX? (from here codes are in Part-2 folder)

See the video from 54:30 to 59:57. It is used to update the HTML using JS in a better way. We can minimise the existing code by importing only the createElement function of the React library and writing createElement("h1", {}, "namaste") like this :-

```
1 import { createElement } from "react";
2 import ReactDOM from "react-dom/client";
3
4 const heading1 = createElement("h1", { id: "title", key: "h1" }, "Hello1 !!");
5 const heading2 = createElement("h1", { id: "title", key: "h2" }, "Hello2 !!");
6 const heading3 = createElement("h1", { id: "title", key: "h3" }, "Hello3 !!");
7 console.log(heading1);
8
9 const container = createElement("div", { id: "container" }, [heading1, heading2, heading3]);
```

But this is still way too long. We can also use a shortform for createElement function while importing it like :-

```
1 import { createElement as ce } from "react";
2 import ReactDOM from "react-dom/client";
3
4 const heading1 = ce("h1", { id: "title", key: "h1" }, "Hello1 !!");
5 const heading2 = ce("h1", { id: "title", key: "h2" }, "Hello2 !!");
6 const heading3 = ce("h1", { id: "title", key: "h3" }, "Hello3 !!");
7 console.log(heading1);
8
9 const container = ce("div", { id: "container" }, [heading1, heading2, heading3]);
```

But these will still be very complex to use for large apps. So JSX was developed by facebook developers. **It is Javascript XML language. JSX is not HTML inside JS, it is a HTML-like syntax used in JS.** For example :- in jsx, React keeps a track of our key, but not our id prop, even though they uniquely identify that child element. There are many other differences.

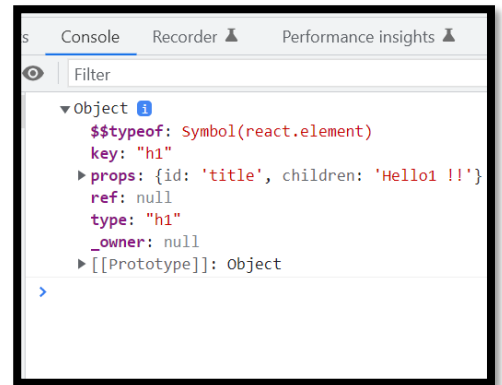
We can use JSX like :-

```

1 import React from "react";
2 import ReactDOM from "react-dom/client";
3
4 const heading1 = (
5   <h1 id="title" key="h1" className="headings">
6     Hello !!
7   </h1>
8 );
9 console.log(heading1);
10
11 const root = ReactDOM.createRoot(document.getElementById("root"));
12 root.render(heading1);

```

Now, you will see that the line 9 will give an object as an output in the console of the browser



JSX and Babel

JSX cannot be understood by our browser, that's why if we write the heading1 in the browser and console log it, it will throw an error because the browser's JS engine cannot read JSX. Here comes Babel, which is JS compiler which transforms these JSX into ECMAScript that the browser can read.

Now how is JSX read ? - When Babel encounters a JSX syntax, it converts that into a React element using `React.createElement()`, which is then converted into an object and that is the reason you will see the line9 gives an object. These things happen behind the scenes.

You can also see how Babel converts the JSX into browser-compatible JS code in their own official site by typing in any JSX on the left and the browser code will be on the right

[Docs](#)
[Setup](#)
[Try it out](#)
[Videos](#)
[Blog](#)

[Donate](#)
[Team](#)
[GitHub](#)

Use next generation JavaScript, today.

Babel 7.20 is released! Please read our [blog post](#) for highlights and [changelog](#) for more details!

Put in next-gen JavaScript	Get browser-compatible JavaScript out
<pre>const heading1 = <h1 id="h1">Namaste</h1></pre>	<pre>const heading1 = /*#__PURE__*/React.createElement(id: "h1", "Namaste");</pre>

#NOTE :-

- JSX is not a part of React. It is a completely different thing
- Babel is already installed as a dependency for Parcel, we don't need to install it separately

React Components :- [\(codes from here are in Part-3\)](#)

There are 2 types

- Functional Components => New way
- Class Based Components => Old way

Functional Components :-

- It is a JS function that returns a piece of JSX, a React element (i.e. if we make a react element using createElement() function), a composition of React elements or a component itself.
- The name of a Functional Component **always starts with a Capital letter**. It is not mandatory, but it's a good practise.
- We can return JSX from that function using the return keyword.
- If the JSX is only of 1 line, then end it with a semicolon, if it has multiple lines then wrap it in brackets and also put semicolon at the end

```
const HeaderComponent = () => {  
  return <div><h1>Namaste React functional component</h1><h2>This is a h2 tage</h2></div>;  
};
```

OR

```
return {  
  <div>  
    <h1>Namaste React functional component</h1>  
    <h2>This is a h2 tage</h2>  
  </div>  
};
```

OR

```
const HeaderComponent = function () {  
  return (  
    <div>  
      <h1>Namaste React functional component</h1>  
      <h2>This is a h2 tage</h2>  
    </div>  
  );  
};
```

- We can also return without using the return keyword like :-

```
const HeaderComponent = () => (  
  <div>  
    <h1>Namaste React functional component</h1>  
    <h2>This is a h2 tage</h2>  
  </div>  
);
```

- Now, to render a normal React element you used to write pass the name of that element in the render() function as an argument. However, to render a functional component, we need to pass "<name_of_component />" as the argument of the render() [This is called using as a Tag].

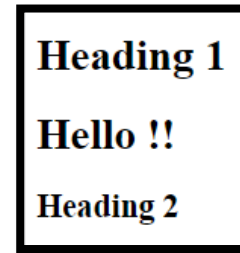
```
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(<HeadingComponent />);
```

- Now, if you do console.log(HeadingComponent), it will give a weird output which isn't an object

```
(()=>/*#__PURE__*/(0, _jsxDevRuntime.jsxDEV)("div", {  
  children: [  
    /*#__PURE__*/(0, _jsxDevRuntime.jsxDEV)("h1", {  
      children: "Heading 1"  
    }, void 0, false,
```

- **To use a React element inside a functional component** :- write { name_of_react_element } inside the JSX which we want to return like :-

The Output :-



```

1  const heading1 = (
2    <h1 id="title" key="h1" className="headings">
3      Hello !!
4    </h1>
5  );
6  console.log(heading1);
7
8  const HeadingComponent = () => (
9    <div>
10     <h1>Heading 1</h1>
11     {heading1}
12     <h2>Heading 2</h2>
13   </div>
14 )

```

- **To use a component inside another functional component** :- We can include the former component as a tag inside the latter one or just calling the former component within “{}” (since it’s a normal JS function) like :-

```

1  const Heading1 = () => (
2    <h1 id="title" key="h1" className="headings">
3      Hello !!
4    </h1>
5  );
6
7  const HeadingComponent = () => (
8    <div>
9      <h1>Heading 1</h1>
10     <Heading1 />
11     <h2>Heading 2</h2>
12   </div>
13 )

```

```

1  const HeadingComponent = () => (
2    <div>
3      <h1>Heading 1</h1>
4      {Heading1()}
5      <h2>Heading 2</h2>
6    </div>
7  )

```

- We can write any piece of JS code inside the JSX by writing it within the curly braces “{}”, like :-

```

1  var x = 10;
2  const HeadingComponent = () => (
3    <div>
4      <h1>Heading 1</h1>
5      {2 + 2}
6      <h2>Heading 2</h2>
7      {x}
8    </div>
9  )

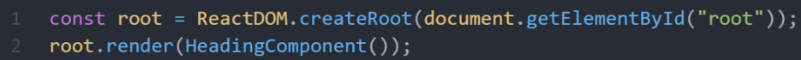
```

- Also see from 2:20:19 to 2:23:10, to see how JSX also provides security to our React app. This is called Sanitization by JSX

Component Composition :- If I have to use a component inside another component, then it's called that

Polyfill :- It is the old piece of code that Babel generates so that our app can be compatible with the older versions of browsers

We can also pass our functional component using a function call, just like what we did when we wanted a functional component inside another one. Like :-



```
1 const root = ReactDOM.createRoot(document.getElementById("root"));
2 root.render(HeadingComponent());
```