


# React Practice

## App Component

```
export default function App() {
  async function getAdvice() {
    const response = await fetch("https://api.adviceslip.com/advice");
    const data = await response.json();
    console.log(data);
  }

  return (
    <div>
      <h1>Hello World!</h1>
      <button onClick={getAdvice}>Get Advice!</button>
    </div>
  );
}
```



---

```
import { useEffect, useState } from "react";
```

```
export default function App() {
  /* useState is a function that returns an array in react
  and here we are destructuring that array 1st position of the
  array is value of the state, the 2nd value is a setter function that
  we can use to update the piece of state */
  const [advice, setAdvice] = useState("");
  const [count, setCount] = useState(0);

  async function getAdvice() {
    const res = await fetch("https://api.adviceslip.com/advice");
    const data = await res.json();
    setAdvice(data.slip.advice);
    /* Now in this getAdvice function we can use setAdvice function to
    update the state, whenever the piece of state is updated
    user interface will also be updated */
    // take current count add 1 and that will become the new count
    setCount((c) => c + 1);
  }

  // Generate very first piece of advice when loaded
  /* useEffect takes two arguments 1st a function that
  we want to execute at the beginning when the component loads and 2nd
  argument is dependency array */
}
```

```

useEffect(function () {
  getAdvice();
}, []);
return (
  <div>
    <h1>{advice}</h1>
    <button onClick={getAdvice}>Get Advice!</button>
    {/* include this component like it is another html element */}
    {/* Pass count as a prop to Message. Props are like parameters
    to function. We call the prop count and pass in the prop
    value */}
    <Message count={count} />
  </div>
);
}

// In react we divide user interfaces into components
// components are reusable pieces of code
// components are used to render UI
// Name of all components should start with capital letter(convention)
// Now accept the props object as a parameter. In this props object count is
now a property
function Message(props) {
  // update count dynamically
  return (
    <p>
      You have read <strong>{props.count}</strong> pieces of advice
    </p>
  );
}

```

---

JSX Template: Babel transpiles JSX code by transforming it into regular JavaScript code. JSX (JavaScript XML) is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript. Babel is a toolchain that is primarily used to convert ECMAScript 2015+ (ES6+) code into a backwards-compatible version of JavaScript that can be run in older browsers or environments.

When Babel encounters JSX code, it transforms it into function calls that create React elements. Here's a simplified explanation of how Babel transpiles JSX:

1. **\*\*Parsing\*\***: Babel first parses the JSX syntax into an Abstract Syntax Tree (AST), which is a structured representation of the code.

2. **Transformation**: Babel then transforms the JSX AST into equivalent JavaScript code. This transformation involves replacing JSX elements with `React.createElement()` function calls.

3. **Generation**: Finally, Babel generates the transpiled JavaScript code from the transformed AST.

For example, given the following JSX code:

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

Babel transpiles it into the following JavaScript code:

```
```javascript
const element = React.createElement("h1", null, "Hello, world!");
```
```

In this transpiled code, `React.createElement()` is used to create a React element representing the `<h1>` element with the text content "Hello, world!".

It's important to note that Babel itself doesn't understand JSX syntax natively; instead, it relies on plugins like `@babel/preset-react` to handle JSX transformation. This preset includes the necessary plugins to transform JSX into JavaScript that can be understood by browsers or other JavaScript environments.

**App.js:**

```
-----
import "./App.css";
function App() {
  const title = "Welcome to the new blog!";
  const likes = 50;
  // can output: number, string, and array
  // can't output: boolean and objects
  const link = "https://www.google.com/";

  const ninja = { name: "Yoshi", age: 30 };
  // returning dynamic values
  return (
    <div className="App">
      <div className="content">
        <h1>{title}</h1>
        <p>Liked {likes} times</p>
        { /* <p>{ninja}</p> */ }
      </div>
    </div>
  );
}
```

```

    <p>`${ninja.name} is a ${ninja.age} years old ninja!`</p>
    { /* bunch different element in arrays together and bunch them as string
*/}

    <p>{[1, 25, 686]}</p>
    { /* use dynamic values as attribute values in element tags */}
    <p>{Math.random() * 20}</p>
    <a rel="noreferrer" target="_blank" href={link}>
      Link to google!
    </a>
  </div>
</div>
);
}

```

```
export default App;
```

Components contain template which makes up the html along with js logic. Components return jsx and In background a transpiler named babel converts all this jsx template into regular JavaScript code when we save the file. class is a reserved keyword in js so we can't use it in jsx.

Components return jsx and exported to use elsewhere.

React converts out (whatever data type we give it to it) into strings before it renders it in the browser.

can output: number, string, and array

can't output: boolean and objects

-----