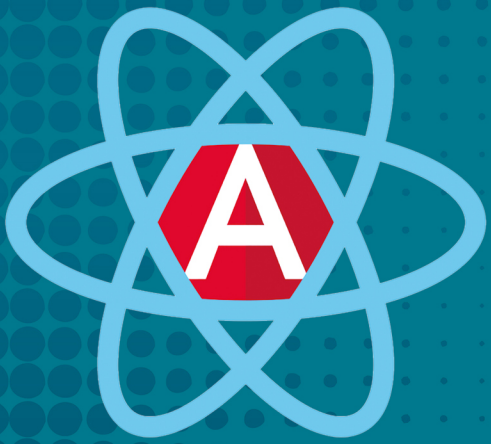
Building Web Applications Using Angular & React



STEP computer ACADEMY

Lesson 5

React: Basics

Contents

React: Basics

What Is React?

React is a JavaScript library by Facebook for building user interfaces. You can obviously create user interfaces using pure JavaScript, however, it takes a lot of time and effort. One of the most popular web development tools — React — would be a much wiser choice.

The first publicly available library appeared on May 29, 2013. But its history has began much earlier. The creator of React is Jordan Walke (a Facebook developer). React was first mentioned in the Facebook news feed in 2011. The library is currently supported by Facebook and a large community of developers. React is an open-source project.

You can read more about the history of React on Wikipedia. We recommend that you do this after reading this lesson. For you will find a lot of essential and helpful information here!

Two libraries have the name of React:

- **React** for building web interfaces;
- **React Native** for building cross-platform apps for iOS, Android, etc.

The objective of this course is React. However, many of the concepts learned will come in handy when getting acquainted with React Native.

The React website is located at https://reactjs.org/. We are sure that you will visit it often.

Goals and Objectives of React

React was created for building user interfaces. You may object that even in 2013, there was a huge number of tools for that. And you are absolutely right. React was created to simplify the building of interfaces to the extent possible because the tools of that time were quite complicated. Simplicity is the React's motto. The whole philosophy of the library is saturated with it. You will have an opportunity to confirm this.

A React app is based on the concept of a component. **A component** is an entity for solving a specific problem in your application. For example, a component can display a registration form or an image and information about it. Your React app will consist of a set of such components. Speaking more formally, a **React app** is a composition of components.

Why Use React?

There are many reasons for using React. Let's analyze some of them:

- **Popularity** — high demand for React specialists around the world.
- **Speed** — React apps update the interface at a very high speed. And it is not lost even in large and complex systems. This advantage is achieved through *Virtual DOM*. We will talk about this later.
- **Easy to embed** — React is relatively easy to add to existing projects.
- **Isomorphism** — React can render on the client and server.

- **Battle-tested** — React was created by Facebook. And Facebook is actively using it. It means that all bugs are fixed as quickly as possible. As long as Facebook uses React, it is not in danger of oblivion.
- **Simplicity** — we have already mentioned the simplicity of the app structure. In addition, React is very easy to learn. It has a small number of concepts that need to be used to start a project.

Who Uses React?

Many companies use it all over the world. We will give you some names: Microsoft, Amazon, Apple, Twitter, Adobe, Salesforce, Netflix, Dropbox, Airbnb, PayPal, and many others. Better ask: "Who doesn't use React?" ☺.

React Features

What makes React so special? Why is it so popular? There are many reasons for this. We have already discussed some of them above. Let's touch on its technical advantages.

- **JSX** is a mechanism for embedding markup in JavaScript code.
- **Virtual DOM** allows making effective changes only to the updated parts of DOM.

You will learn about other important concepts a bit later.

Where Are React Apps Developed?

We can develop React apps online and locally. To work on a local computer, you need to follow a sequence of steps to build a basic application. It may be rather difficult provided

that you only begin to dive into new technology, so we will begin with an online editor. It will allow us to learn the basics without delving into a large number of details. Once you have the basics, we will by all means set up the React environment on your computer.

Online React Editors

A large number of online React editors are available on the Internet. Each of them has pros and cons, like any software product. Let's have a quick review of the options:

- **repl.it** is a monster in the world of online editors. It has a huge number of templates for various programming languages and technologies. Lots of useful tools are available for developers.

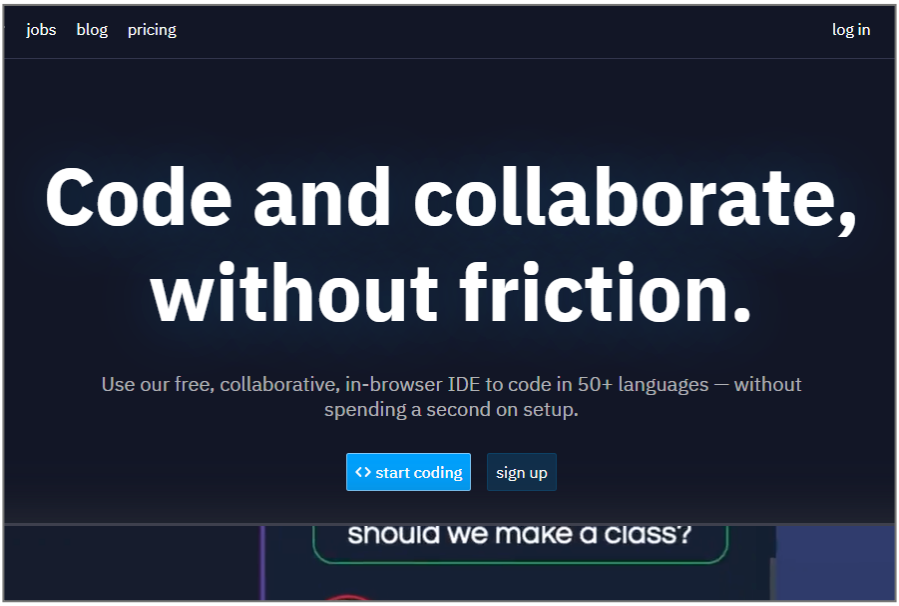


Figure 1

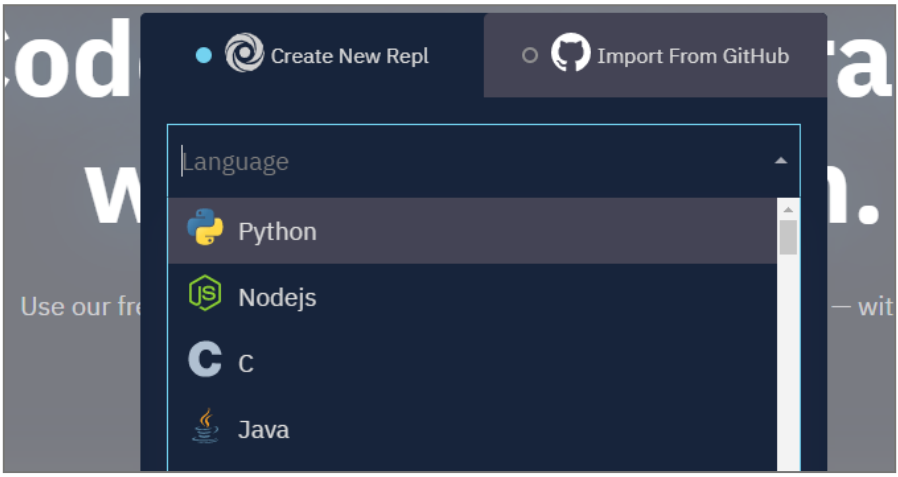


Figure 2

- [Link] to the website.

- **jsfiddle.net** is one of the oldest and most stable players in the market of online editors. Good React support and user-friendly design.

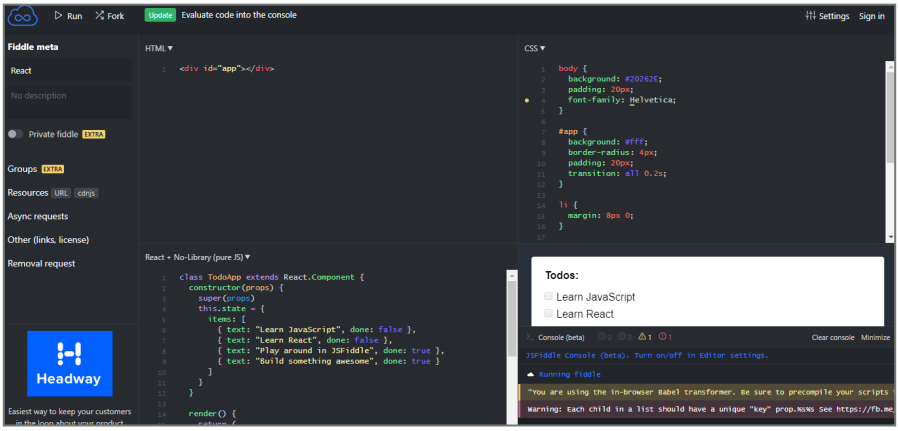


Figure 3

- [Link] to the website.

- **codepen.io** is one of the most popular and famous online editors. A large number of users, lots of project templates, React support.

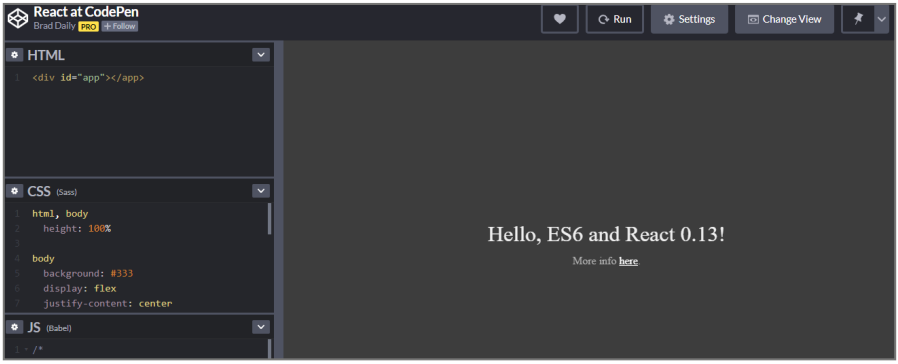


Figure 4

- Link to the website.

- **codesandbox.io** is an online editor that is rapidly gaining popularity. It provides great functionality and performance, even in a free mode. And this is what we are going to use in our lessons.

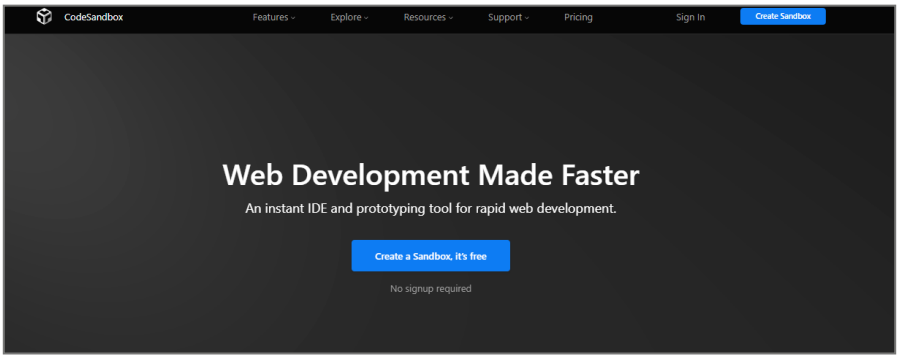


Figure 5

- Link to the website.

Setting Up an Online Environment

In order to start learning React, first we need to set up our online environment. CodeSandbox uses a GitHub account for logging in. Go ahead and click Sign in to make sure of this. So our first step would be to create a GitHub account if you do not have one.

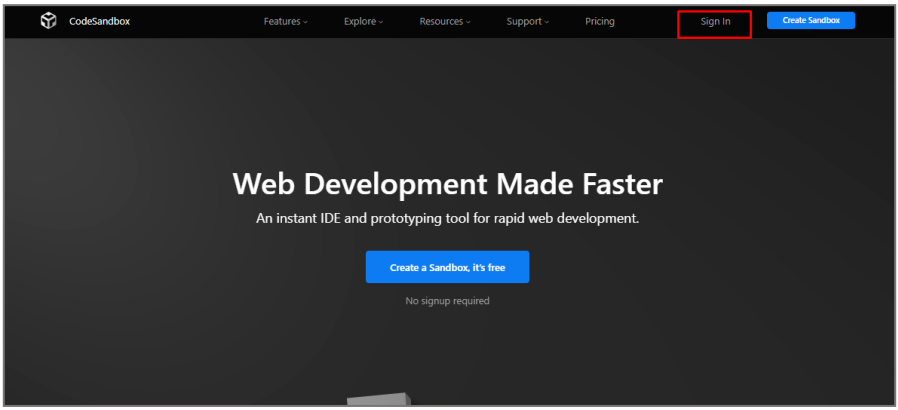


Figure 6

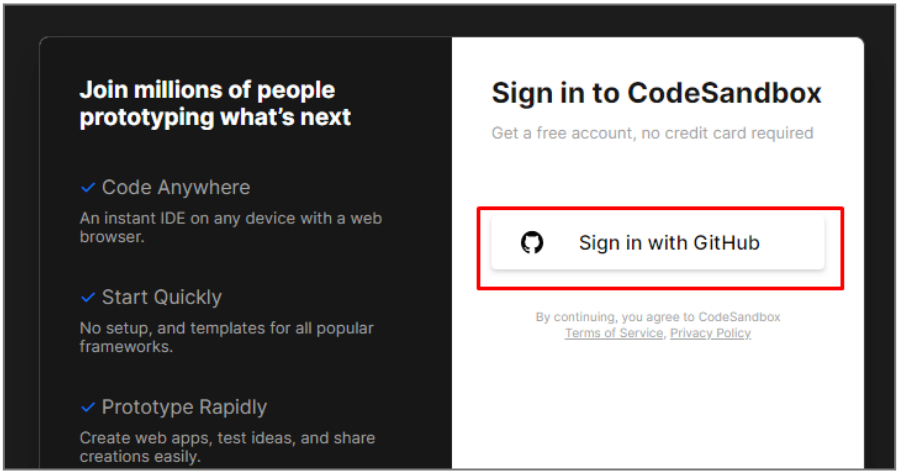


Figure 7

To create a GitHub account, go to https://github.com/ and fill out the registration form.

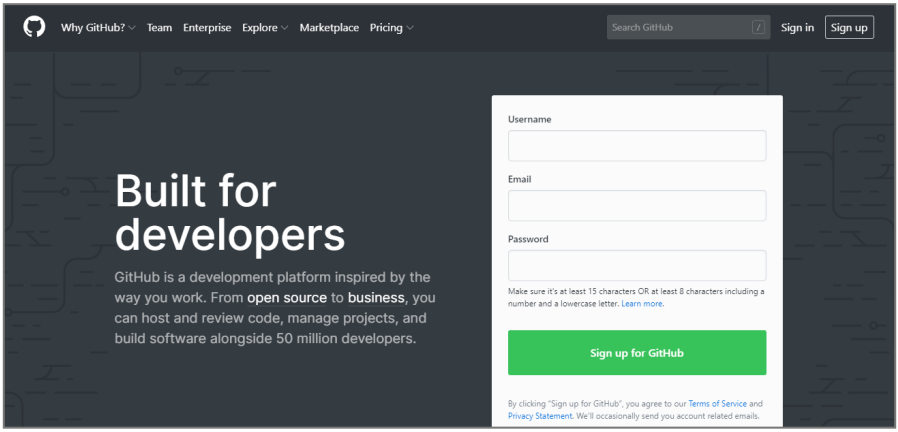


Figure 8

Once you are done, you will receive an automatic confirmation letter to your email. Remember to click on the link to confirm. After that, your GitHub account becomes fully operational.

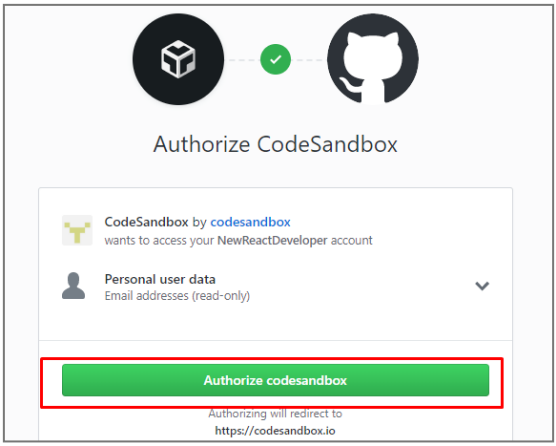


Figure 9

Now click Sign in with GitHub in CodeSandbox. You will see the following window (Fig. 9).

Click the Authorize codesandbox button to complete your CodeSandbox registration. If everything goes well, you will see the editor window:

Figure 10

This online editor is based on the concept of a sandbox (it is, in fact, the project we are working on). Now we need to create our first React sandbox.

Click the Create Sandbox button.

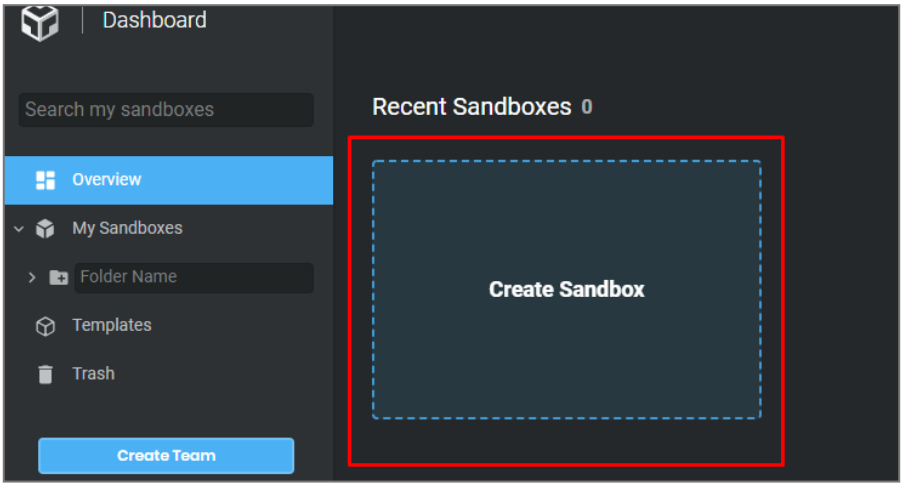


Figure 11

Select React in a new window. This is a React app template made by the CodeSandbox team.

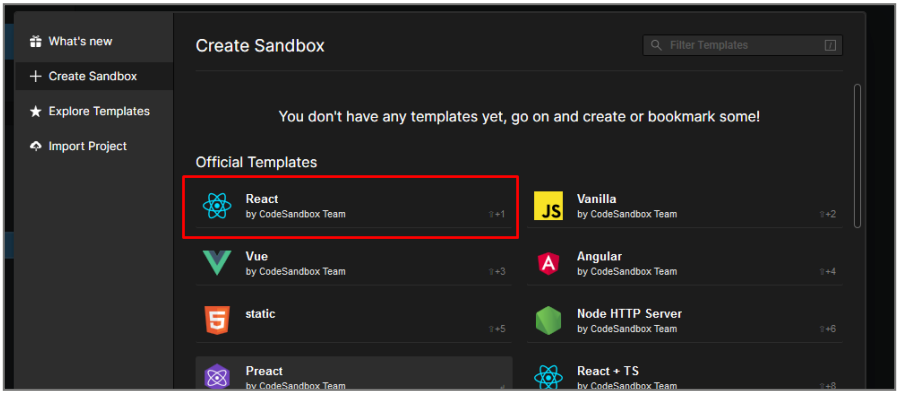


Figure 12

After you click React, an app skeleton will be created in a new window.

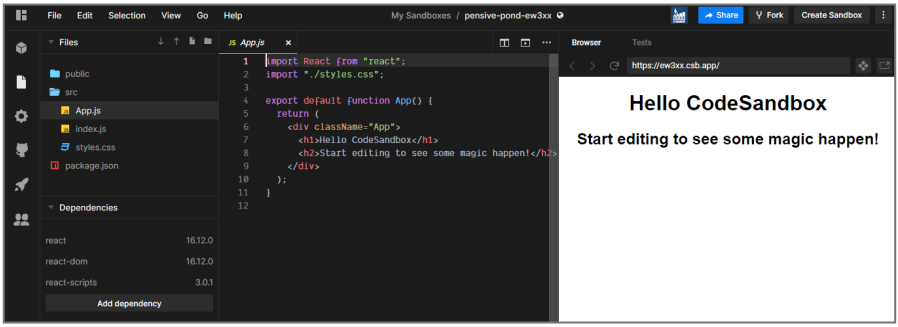


Figure 13

Hooray! We have created a skeleton of the first app! Beginning is the hardest part, but it will get easier ☺.

It is time to look around inside our editor. The workspace is divided into three parts by default. There are files of our project on the left.

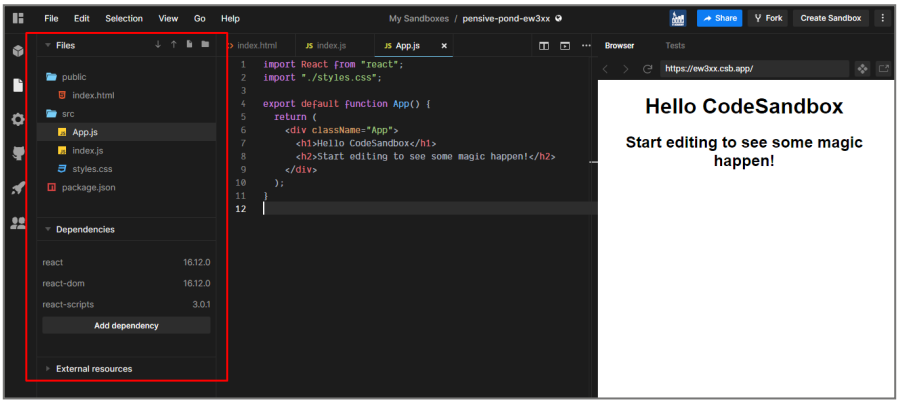


Figure 14

Our project has two folders: *public* (should contain files accessible from outside) and *src* (source code of our app). There is also a *package.json* file in the root folder of our project (it has project settings).

The source code editor is in the center of the window. This is where we will edit the text of our files (Fig. 15).

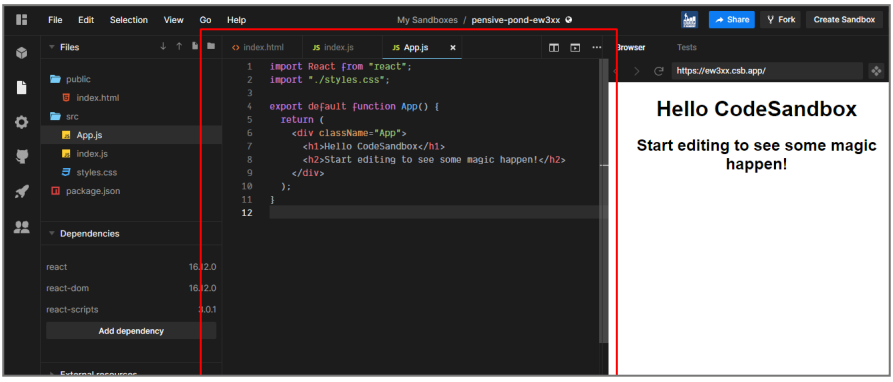


Figure 15

The source code of the *App.js* file is currently open. We see the code of the simplest component.

The embedded browser window is on the right. We will use it for output. When you change the source code, the output in the browser changes.

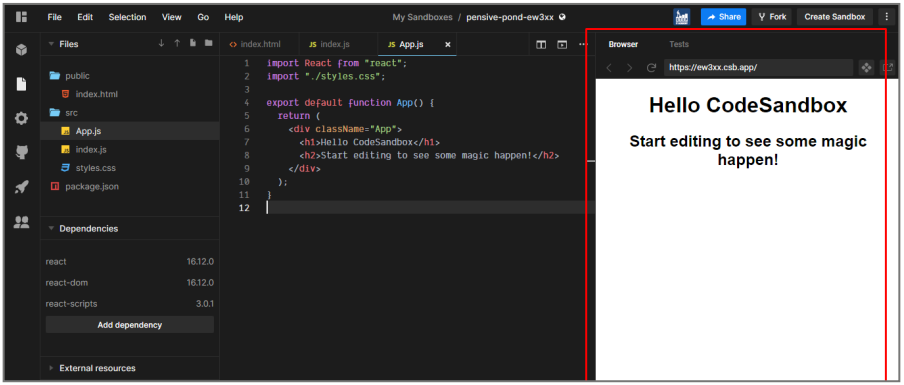


Figure 16

Notice the address bar inside the browser. It contains an address that can be copied and pasted in an address bar of a new browser window, so that you could go to your first React app.

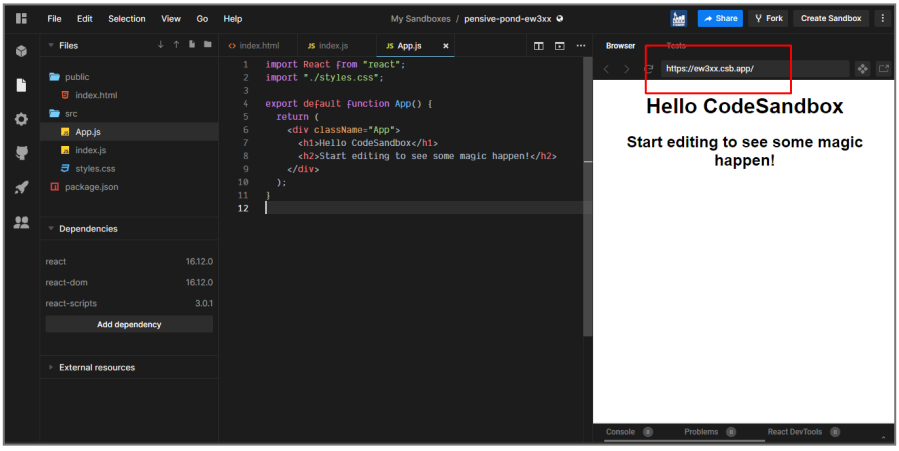


Figure 17

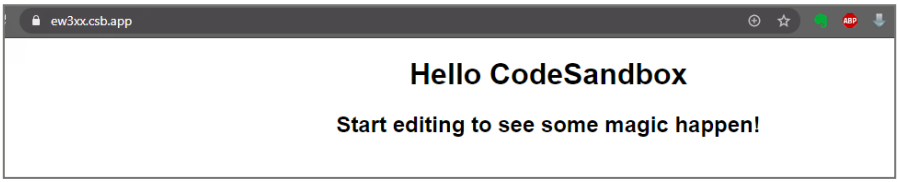


Figure 18

What is the name of our new app, and how do we change it? Move the mouse to the caption My Sandboxes/name at the top and click on it.

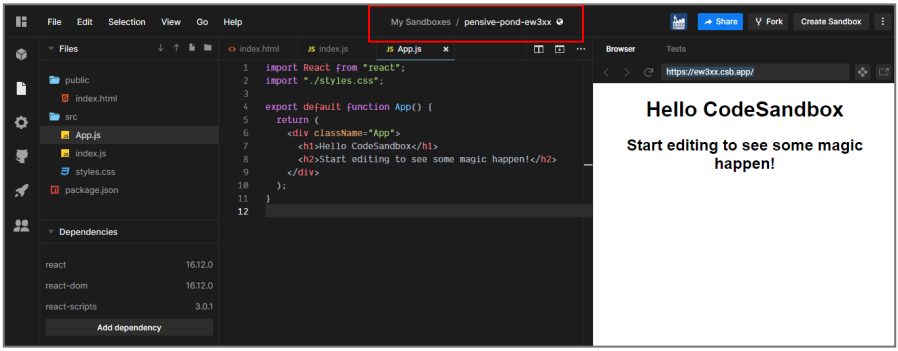


Figure 19

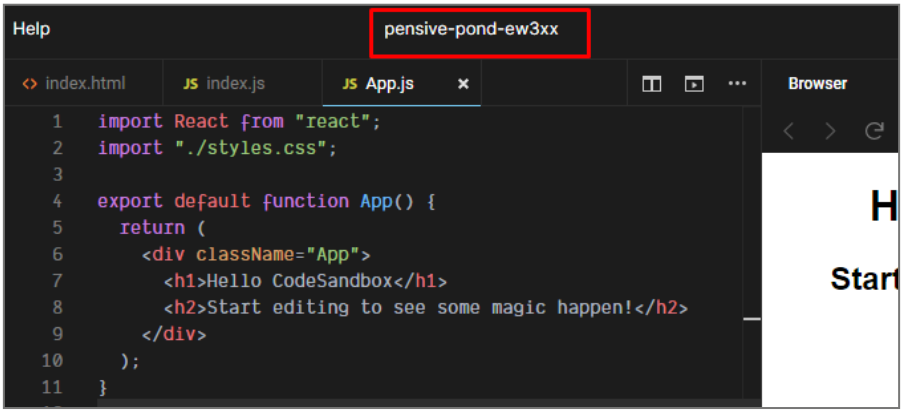


Figure 20

The default project name is ill-defined: pensive-pond-ew3xx. Let's change it to helloworld. We recommend you change the project name immediately after creation.

In order to return to the start window, click on the square in the upper left corner of the screen. If you did not save your changes, the browser would warn you about this. To save changes, click on the File and select the relevant item from the Save set.

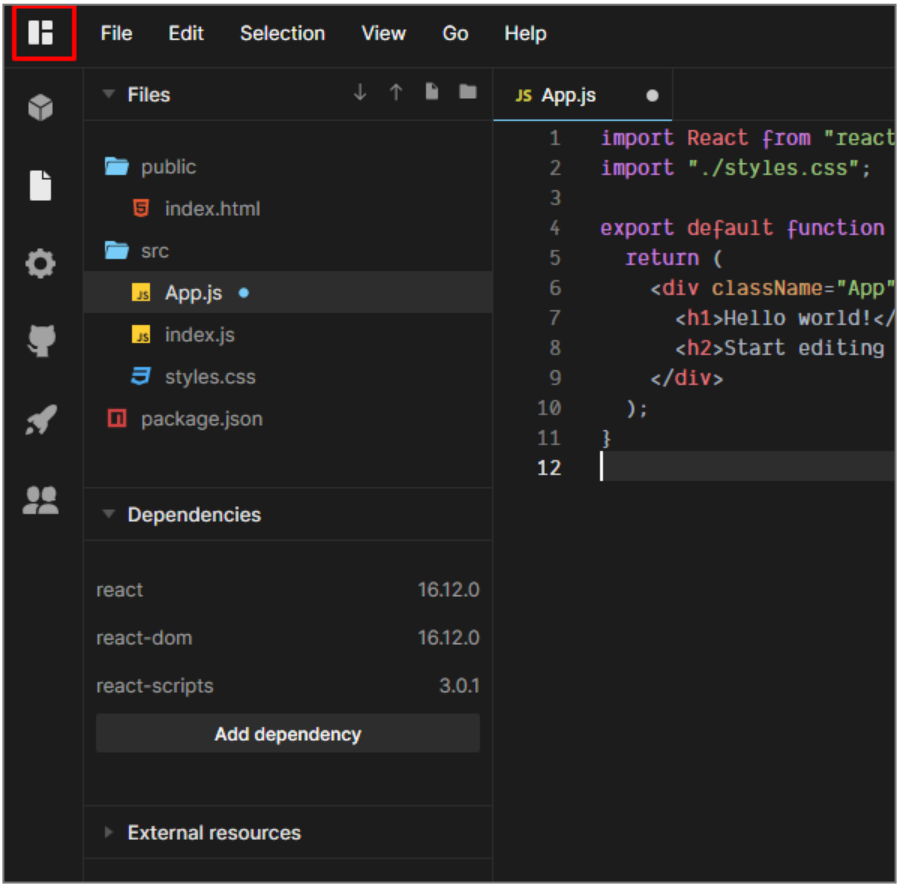


Figure 21

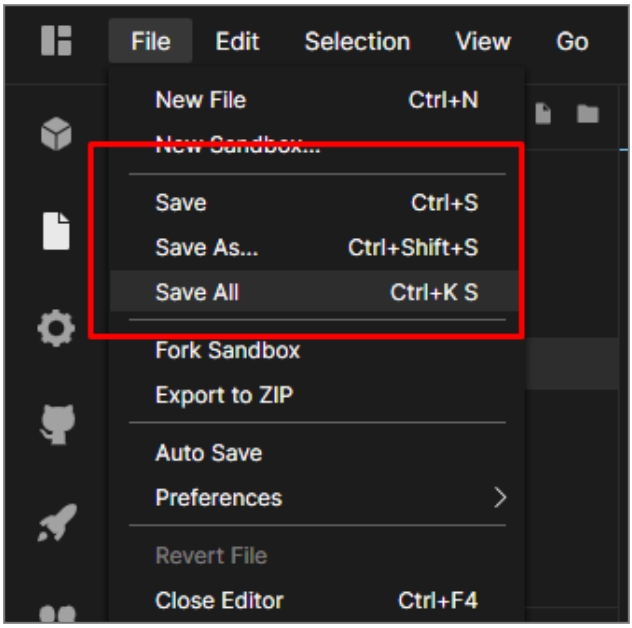


Figure 22

So, we have returned to the start window. Now it displays our first created project.

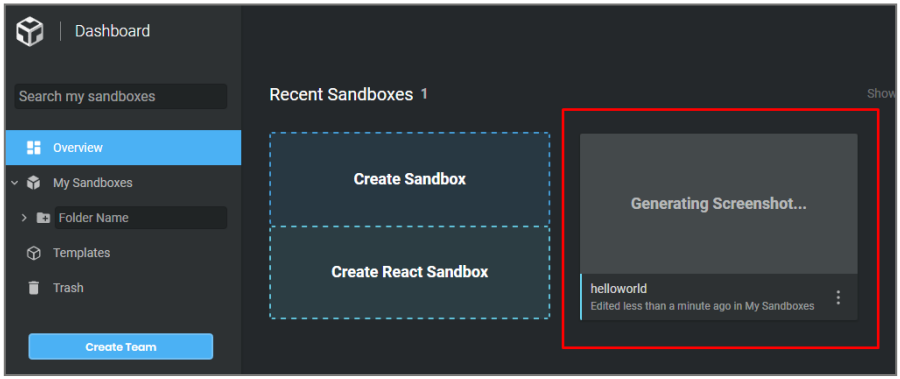


Figure 23

If you click helloworld, our project will open.

Project Structure

Let's proceed with the analysis of the project structure. We already know that there are several folders and a set of files. What is inside each of them?

We will begin with the public folder.

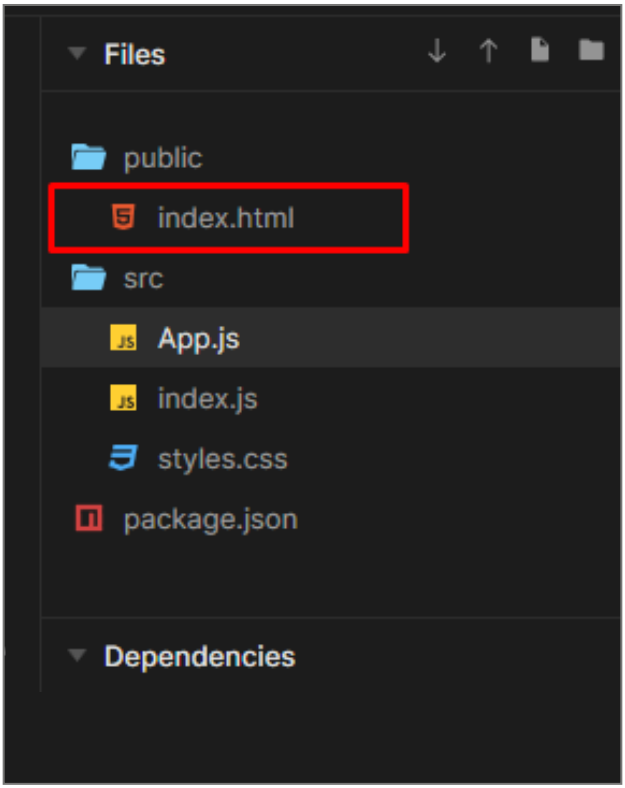


Figure 24

The folder name indicates that this folder should contain publicly available files. Our folder currently has one file — *index.html*. It has a web page layout. We will access it from the outside (Fig. 25).

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <meta name="theme-color" content="#000000">
  <!--
      manifest.json provides metadata used when your web app is added to the
      homescreen on Android. See https://developers.google.com/web/fundamentals/engage-a
    -->
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
  <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
  <title>React App</title>
</head>

<body>
  <noscript>
    You need to enable JavaScript to run this app.
```

Figure 25

The Figure provides only a part of this document. We are interested in the body section.

```
<body>
    <noscript>
        You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
</body>
</html>
```

It has no special layout since we have a React app, and the main code will be in other folders. Notice div with the root identifier. We will use it to implement our React code.

The next very important folder is *src*. As the name implies, this folder stores the source code. Three files are inside.

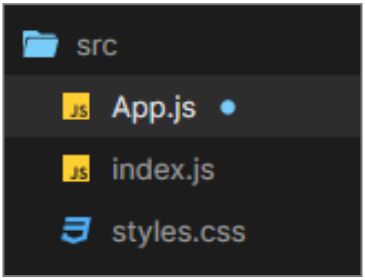


Figure 26

Let's begin with *index.js*.

Index.js is a default entry point in a React app. This file starts the execution of the app.

This code provides access to div with the root identifier, and we can load our React component there.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    rootElement
);
```

Let's analyze the code line by line.

```
import React from "react";
import ReactDOM from "react-dom";
```

The first line imports the React package to our script. It has the functionality that is basic to any React app. Our code will not work without this import.

The second line imports the ReactDOM package. It is used for Virtual DOM to work. You already know what DOM is (*Document Object Model*). **DOM** is a logical, tree-like structure of a web page. It has all the elements of your web page. You use JavaScript DOM to modify a page. However, any DOM modification is resource-intensive, as it forces the browser to redraw the entire page. Virtual DOM is a copy of DOM. When you make changes to DOM from your React app, you actually change Virtual DOM. If you need to edit a real DOM, React compares the differences between Virtual DOM and DOM. Based on the result, only the edited parts of the page are updated without redrawing the entire page. This gives a big performance boost to your application.

```
import App from "./App";
```

We import our user component named App from the *App.js* file.

A component is a building block of a React app. You will create a large number of components in this course. The name App is not a keyword. You can change it. **App** is short for Application.

```
const rootElement = document.getElementById("root");
```

If everything goes well, the rootElement will refer to div with the root identifier. You can replace const with var and let. Using const is a good programming style.

```
ReactDOM.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    rootElement
);
```

This line calls the render function from the ReactDOM package. It is used to render a React component. **The first function argument** is a React component we want to display. React.StrictMode enables strict mode checks inside this tag. Be sure to have an opening tag and a closing tag. React.StrictMode does not have a visual display and will work only at the development stage, but not in production. App is our displayed component. The <App/> form is the JSX we talked about. It allows you to mix JavaScript code and tags.

The second argument is a link to the element where we want to display our React component.

The *App.js* file contains the code of our component.

```
import React from "react";
import "./styles.css";

export default function App() {
    return (
        <div className="App">
            <h1>Hello CodeSandbox!</h1>
            <h2>
               Start editing to see some magic happen!
            </h2>
        </div>
  );
}
```

Let's analyze it line by line. Add React and styles from the *style.css* file.

```
import React from "react";
import "./styles.css";
```

And look at the code of our component.

```
export default function App() {
    return (
        <div className="App">
           <h1>Hello CodeSandbox!</h1>
           <h2>
               Start editing to see some magic happen!
           </h2>
    </div>
  );
}
```

The component is the function named App. The code specified in return is the visual part of our component. We use JSX for the visual part of the component. Our component consists of div and the tags h1, h2 inside of it. We style it using styles from the *style.css* file. The className attribute was specified for this. In a regular HTML layout, we use the class attribute; in the JSX code, it is replaced with className. If the JSX code consists of multiple lines, it should be enclosed in brackets. export default indicates that the App component is exported and can be used by other modules. You cannot use components outside their files without export, because they will be private, or closed for external use. A file can contain only one export default. If you need to export other entities, apply export without indicating default. We will definitely talk about this mechanism in future lessons.

The *style.css* file has styles for our component.

```
.App {
    font-family: sans-serif;
    text-align: center;
}
```

The *package.json* file has app senttings. Specifically, an entry point in our app.

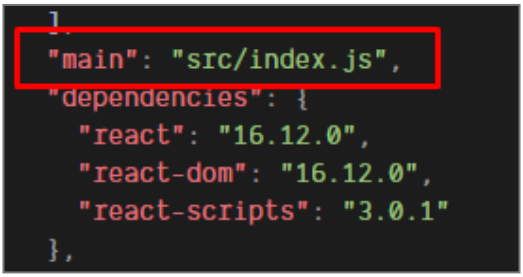


Figure 27

We will talk in more detail about other settings of this file later.

So we have completed the analysis of the application skeleton. There are many starter React templates. If you use another online editor, your starter code may differ. Nonetheless, the base will be very similar. It is time to create and configure components.

Components

What Is a Component?

As you already know, a React app consists of a set of components. **A component** is a building block for our apps. Components allow us to split our UI (User Interface) into

independent parts that can be used separately. Let's say you created a component to display random numbers for a project, well you can easily transfer it to another project. In terms of programming, a component is a code fragment solving a specific problem and displaying in our interface. You have already created your first component, but that was only the tip of an iceberg.

Functional and Class Components

Today React allows you to create two types of components: functional (functional components) and class (class components). In the beginning, React provided only one type of components: class components. The name implies that the component's code is inside a class. A large amount of code was written with this type of components.

Functional components (the component's code is inside a function) appeared much later. But now they are a recommended mechanism for creating components. This is due to the convenience and ease of writing code. If you create a new app, you should definitely use functional components.

Does this mean that you do not need to know how to work with class components? Of course not. If you come across a project with class components, your knowledge will be beneficial.

Our course will focus on functional components, but mention class components as well. Let's immediately agree on one important rule: a component's name should always begin with a capital letter. React believes that names of html elements start with a lowercase letter, and names of all components are capitalized.

Create Your First Component

Let's create our first intended React app. We are going to use a functional component. Our first app will print a well-known Shakespeare's line: "To be or not to be."

Create a project in CodeSandbox, as described above. Rename it to Shakespeare.

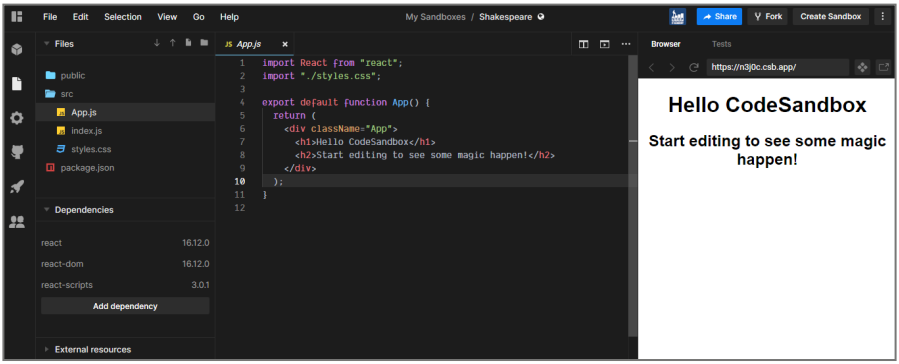


Figure 28

Upon creation, we see the same project template. First, change its title. For this, go to the *index.html* and edit the content of the title tag.

```
<title>Shakespeare</title>
```

In order to print the line "To be or not to be," go to the *App.js* file and edit the code that returns return.

```
import React from "react";

export default function App() {
  return (
    <div>
      <h1>To be or not to be,</h1>
```

```
      <h2>that is a question</h2>
    </div>
  );
}
```

We removed styles, so we do not have a style import at the beginning of the file. The code of our first functional component is up and running.

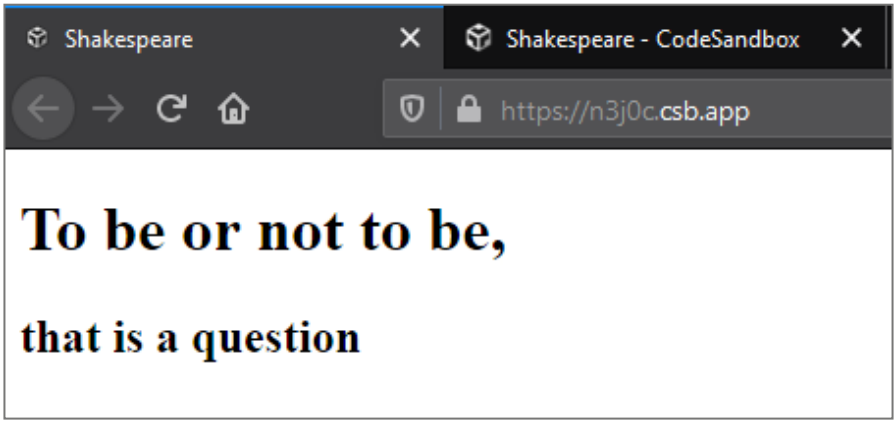


Figure 29

- You can access the code here.

We have seen import multiple times in the code. Recall that we imported the react package with it. We provide access to the received features through the React object.

```
import React from "react";
```

This type of import is called default import. It imports whatever was exported with export default. But what if we need to import something that was exported with export

without indicating default? Then we use a so-called named import. In our code, we imported the react-dom package to access render. Instead, we could have used a named import for render. Then, instead of exporting all the features, we could have accessed render only.

```
import React from "react";
import { render } from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");
render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    rootElement
);
```

This code has two differences. **The first difference** — it is a named import.

```
import { render } from "react-dom";
```

This import has {} that allow us to specify what exactly we will import. **The second difference** — we access render without specifying an object.

```
render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    rootElement
);
```

You will use both types of import in your code.

- Link to the project code.

We used JSX in the code of our examples, so we could embed tags directly in the JavaScript code. Upon processing, the JSX code turns into a regular JavaScript. If you want to, you can use JavaScript instead of JSX in your code.

Let's consider this possibility. Create a project and name it Shakespeare2. The app will print a quote from King Lear: "Nothing will come of nothing."

First, use JSX.

```
import React from "react";
import "./styles.css";

export default function App() {
  return <p>Nothing will come of nothing</p>;
}
```

Turn it into a pure JavaScript.

```
import React from "react";

export default function App() {
  return React.createElement("p", null,
                "Nothing will come of nothing");
}
```

We use the createElement method to create an element. The first argument of the function is a tag name, the second is properties (attributes) of the tag, the third is element content.

We create a paragraph, do not specify any properties, and put the line "Nothing will come from nothing" inside the paragraph.

Let's move on to the *index.js* file. The code below utilizes JSX.

```
import React from "react";
import ReactDOM from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Turn it into a pure JavaScript.

```
import React from "react";
import ReactDOM from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(React.createElement(App, null),
                rootElement);
```

We have changed <App/> to the call of createElement with the transfer of parameters. The first parameter is the name of the created component (App), the second is properties (null since we did not use them).

Even this small example shows you that JavaScript is less convenient than JSX. We will use only the latter in the next examples.

- You can access the code here.

Let's go back to the JSX syntax and create an example that prints the current time. We want to display a new time

every time the app starts. The Date object will be used to get time. The received value will be inserted inside the JSX. Every time you need to insert calculated values inside JSX, use {}. Create a project and change the content of *App.js*.

```
export default function App() {
    /*
      Get the current time.
      We will use it as a value inside h2.
      To insert a value that must be dynamically
      populated, use {}.
    */
    let currentTime = new Date().toLocaleTimeString();

    return (

        <div className="App">
            <h1>Current time is</h1>
            <h2>{currentTime}</h2>
        </div>
    );
}
```

Pay special attention to the code:

```
<h2>{currentTime}</h2>
```

currentTime is our variable with the current time. We use {} to insert it inside the JSX. It is a standard React mechanism that you will constantly use in your projects.

- Link to the project code.

We have already created several functional components. Let's now try to create a class component. Class components

are contained in classes. One class — one component. A component class can be inherited from React.Component. We must implement the render method in a class component. This method returns the UI of your component. Let's create a project with a class component. The component will print a famous Walt Disney quote. Create a project again and go to *App.js*.

```
import React from "react";
import "./styles.css";
/*
    Class components are classes.
    They must be inherited from React.Componet.
    The render method must be implemented inside
    the class.
    It must return the UI of our component.
*/

export default class App extends React.Component {
    /*
      The render method is a must
      It must be implemented in any class component.
    */
  render() {
    return (
      <div className="App">
        <h1>
            The way to get started is to quit
            talking and begin doing
        </h1>
        <h2>Walt Disney</h2>
      </div>
    );
  }
}
```

The function is replaced with the class with the required render method. There are no other major changes in our code.

- Link to the project code.

Let's repeat the example with printing the current time, but this time use a class component. Create a project again and change the *App.js* content.

```
import React from "react";

import "./styles.css";

export default class App extends React.Component {
  render() {
    /*
     This time we did not create a variable.
     We inserted the time update right into the JSX code.
     Be sure to use {}.
    */
    return (
      <div className="App">
        <h1>Current time is:</h1>
        <h2>{new Date().toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

We did without an additional variable. The code for getting current time was put directly in JSX, inside {}.

- Link to the project code.

Even a short introduction to class components shows us some bulkiness of their design.

At the moment we are operating with one component, but there are always more components in real life. Let's try to create several components in a new app. We will have two components. One component will display the current date, and the second will display the current time. Each component is located in its function. Let's consider the *App.js* file:

```
import React from "react";
import "./styles.css";
/*
   Component for displaying the current date
*/
function CurrentDate() {
    return <h2>{new Date().toLocaleDateString()}</h2>;
}
/*
    Component for displaying the current time
*/
function CurrentTime() {
  return <h2>{new Date().toLocaleTimeString()}</h2>;
}
/*
    App component that puts date and time together
*/
export default function App() {
  return (
    <div className="App">
      <CurrentDate />
      <CurrentTime />
    </div>
  );
}
```

Our project has three components.

- App is an app component we are already familiar with.
- CurrentDate is a date component.
- CurrentTime is a time component.

Both new components are functional components. We include them in the App component code with the already familiar <component name/>.

- Link to the project code.

You may have a question: “Can we remove this div? Can we write like this?”

```
export default function App() {
  return (
      <CurrentDate />
      <CurrentTime />
  );
}
```

Of course you can ☺. However, the code will not run because React sees it as an error. The error description will be displayed in the browser on the right:

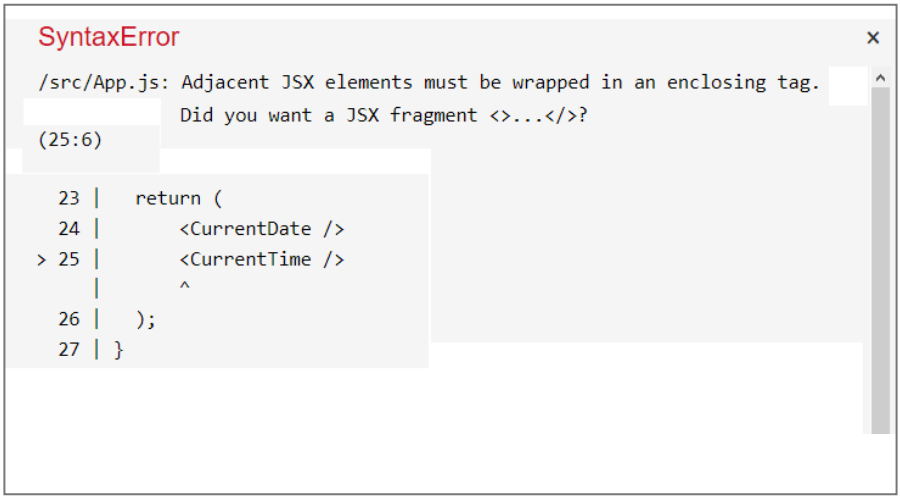


Figure 30

When creating JSX code, you should always include elements in the main, parent, aggregating element. This element has always been div in our examples.

If you do not want to use a parent element that spawns new DOM nodes, as div does, then use React.Fragment.

<React.Fragment> allows you to group many elements within itself. And it will not create new nodes in DOM. Let's rewrite our code using Fragment:

```
export default function App() {
  return (
    <React.Fragment>
      <CurrentDate />
      <CurrentTime />
    </React.Fragment>
  );

}
```

React.Fragment has a shorthand notation: <>. You can use it instead of classic, long construct:

```
export default function App() {
  return (
    <>
      <CurrentDate />
      <CurrentTime />
    </>
  );

}
```

The result of this code will be the same as if you used React.Fragment.

Let's solve the same problem using class components.

```
import React from "react";
import "./styles.css";


class CurrentDate extends React.Component {
  render() {
    return <h2>{new Date().toLocaleDateString()}</h2>;
  }
}

class CurrentTime extends React.Component {
  render() {
    return <h2>{new Date().toLocaleTimeString()}</h2>;
  }
}

export default class App extends React.Component {
  render() {
    return (
      <div className="App">
        <CurrentDate />
        <CurrentTime />
      </div>
    );
  }
}
```

Our code has three class components. Each of these components has the render function implemented.

- Link to the project code.

We do not put our components in different files for now because they are quite simple, and it is easier to get started with React this way. We will do this in the following lessons.

We have not used event handlers in our components' codes yet. Let's try to create a component with an event handler. Our component will respond to a button click. If the event occurs, we will display a message window.

```
import React from "react";
function handlerClick() {
  alert("You clicked the button");
}
export default function App() {
  return <button onClick={handlerClick}>Click me!
         </button>;
}
```

handlerClick is our event handler function. We will display the information window in it. We bind the click event to the button using the onClick attribute. Recall that you used the onclick attribute in a regular HTML. In the onClick we used the already familiar {}. We indicate the handler name in them. You can use an arrow function instead of a regular function in this code.

```
export default function App() {

  // Arrow function as a handler
  const handlerClick = () =>
        alert("Hello from arrow function!");
  return <button onClick={handlerClick}>Click me!
        </button>;
}
```

Arrow functions are quite typical for modern JavaScript code.

- Link to the project code.

Props

In our examples, we did not pass parameters for components from the outside. May we need such a mechanism? Of course. Parameters passed from the outside allow the user of the component to configure it before use. Let's say you create a component that prints information about a book. You can pass a book title, data about the author, genre, etc. through its properties.

So how do we pass parameters to a component? There is a mechanism named Props (short for properties) in React. When creating a component object, you specify a set of attributes with values in its description. For example,

```
<SomeComponent text = value color = value … />
```

We create an object of SomeComponent. We specify values for its attributes (properties) text and color upon creation. Property names can be anything. The main condition is that they should make sense. The values can be fixed or set with {}.

If you have a functional component, you should specify at least one argument for it. This argument is usually called props (it is common practice, but you can change the name if you want to). Properties with names specified upon creation of the component will be created inside this argument.

In our case, it will look as follows:

```
function SomeComponent(props){
    let t = props.text;
    let c = props.color
      ……….
}
```

If you have a class component, the access will be through this. It will look like this:

```
this.props.text
this.props.color
```

Let's consider props through examples. Begin with the creation of a functional component that will print a quote and author. The quote and author will be passed through props.

```
/*
  We specified the parameter for the functional
  component in order
  to be able to get attribute values.
  This parameter is usually called props,
  but the name can be anything.
  The parameter is populated automatically.
  To access the attribute value, use
  props.name_attribute
*/
function Quote(props) {
  return (
    <>
      <h2>{props.text}</h2>
      <h2>{props.author}</h2>
    </>
  );
}
export default function App() {
  /*
    You can get this symbol ` by pressing Alt 96,
    96 on the numeric keypad.
  */
  let qText = 'Tell me and I forget.
               Teach me and I remember.
               Involve me and I learn.';
```

```
  let qAuthor = "Benjamin Franklin";
  /*
    Use the props mechanism to pass data to
    the component.
    Pass data to the component through attributes.
    Attribute names were invented by us.
  */

  return (
    <div className="App">
      <Quote text={qText} author={qAuthor} />
    </div>
  );
}
```

The component is called Quote in our code. We pass the quote and information about the author to it through props. We specified values for the properties text and author and used {} for this.

```
<Quote text={qText} author={qAuthor} />
```

We use props to access values inside the component's code.

```
function Quote(props) {
  return (
    <>
      <h2>{props.text}</h2>
      <h2>{props.author}</h2>
    </>
  );
}
```

The output is as follows:

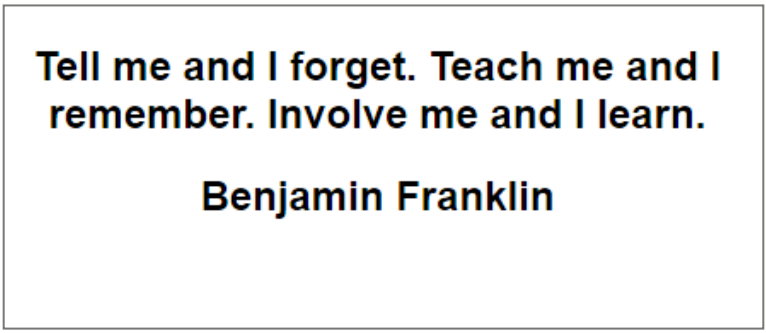


Figure 31

- Link to the project code.

To consolidate the material, let's use one more functional component. It will output a random number in the range we specify. We are going to use props to set the start and end of the range.

```
import React from "react";
function RandomVal(props) {
  /*
     Generate a random number in the range
     from min to max
  */
let currentVal =
    Math.floor(Math.random() *
              (props.max - props.min + 1)) + props.min;
  return <h2>{currentVal}</h2>;
}

export default function App() {
  let start = 1;
  let end = 7;
```

```
  /*
    Pass the start values through props
  */
  return (
    <>
      <RandomVal min={start} max={end} />
    </>
  );
}
```

And again, we specify values for the min and max properties upon the component creation. We access them using the props parameter in the component's code.

- Link to the project code.

Let's look at another example of working with props. This time we will access props inside the class component. Our class component will display the quote and author.

```
import React from "react";
import "./styles.css";

class Quote extends React.Component {
  render() {
    /*
      To access props inside the class, we should use
      this
    */
    return (
      <>
        <h2>{this.props.text}</h2>
        <h2>{this.props.author}</h2>
      </>
    );
```

```
  }
}

export default function App() {
  const qText = "Stay hungry, stay foolish";
  const qAuthor = "Steve Jobs";
  return (
    <div className="App">
      <Quote text={qText} author={qAuthor} />
    </div>
  );
}
```

Although we use a class component, properties are passed in the same way as in the functional component. The only difference is in how we access the properties. For this we use this construct: this.props.property_name.

```
<>
    <h2>{this.props.text}</h2>
    <h2>{this.props.author}</h2>
</>
```

- Link to the project code.

To consolidate the material, try to create your own component that will use props.

Homework

1. Build and run a React app that displays brief info about you in a browser. For example, your name, contact phone number, email.
   Use functional components and JSX syntax.
2. Build and run a React app that displays brief info about your city in a browser. For example, city, country, city year, some photos of the sights.
   Use functional components and JSX syntax.
3. Build and run a React app that displays a recipe in a browser. For example, name, ingredients (and quantity), cooking steps, a photo of the finished dish.
   Use functional components and JSX syntax.
4. Create an app that displays Shakespeare's bio using element rendering. Create several components to implement different parts of the app. For example, a component to display general information about Shakespeare, a component to display information about his specific work.
5. Create an app Favorite Movie. It will contain information about your favorite movie: title, director, year of release, film studio, poster, etc. Be sure to use functional components and props.
6. Create an app Personal Page. It will contain information about you (name, phone number, email, city of residence, work experience, skills, photo, etc.) Be sure to use functional components and props.

Lesson 5

React: Basics