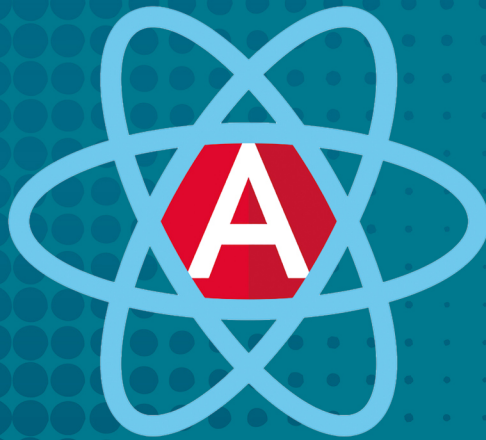


# Building Web Applications Using **Angular** & **React**



# Lesson 7

## React: Advanced Techniques

### Contents

<b>Return to Javascript.....</b>	<b>3</b>
<b>Using arrays in React Apps .....</b>	<b>9</b>
<b>Key .....</b>	<b>17</b>
<b>Forms .....</b>	<b>25</b>
<b>Refs .....</b>	<b>29</b>
<b>State and Forms .....</b>	<b>35</b>
<b>Homework .....</b>	<b>42</b>

# Return to Javascript

In order to move on and learn new things about React, we should remember JavaScript. We need the `map` method which you are possibly familiar with. It is used to transform arrays. Simplified syntax of this method:

```
array_name.map(function)
```

Method principle:

1. A function specified as a parameter is called for each array element called by `map`. It means that if there are three elements in an array, the function will be called three times.
2. `map` returns a new array. Elements of this array will be values received after the parameter function operates on each value from the original array.
3. The `map` method does not change the original array.
4. The parameter function is called only for elements with values in the original array.

Let's consider a usage example. A function that calculates the square root is called for each element in the array.

```
/*  
    Original array  
*/  
var arr = [9, 4, 16];  
  
/*  
    9 4 6  
*/
```

```

console.log(arr);

/*
  Math.sqrt function is called for each element
  in the arr array map returns a new array, whose
  elements are arr elements, upon the Math.sqrt call
  for each of them. The original array does not change
*/
var mArr = arr.map(Math.sqrt);

/*
  3 2 4
*/
console.log(mArr);

```

Every time `map` is called, the square root function is called for each `arr` element. `map` returns a new array with the values from `arr` upon calculating the square root. This is why `mArr` contains elements with values `3, 2, 4`.

In this example, we used a standard `sqrt` function, but what if we need to call a custom (user-defined) function? How can we do it? To solve this problem, we could use the following `map` syntax:

```

array_name.map(custom_function(current element){
  body
})

```

We pass the custom function to the `map` call. It will be called for each element in the array. Our function will take the current array element being analyzed as a parameter. This parameter will be populated automatically.

In our next example, a custom function will square each element in the array.

```
/*
    Original array
*/
var arr = [3, 4, 7];

/*
    3 4 7
*/
console.log(arr);

/*
    Math.pow function is called for each element
    in the arr array map returns a new array, whose
    elements are arr elements, upon the Math.pow call
    for each of them. The original array does not change
*/
var mArr = arr.map(function(item) {
    return Math.pow(item, 2);
});

/*
    9 16 49
*/
console.log(mArr);
```

Our custom function does not have a name. Although we could have easily created a function and passed its name as a parameter. Function code is called for each element of the `arr` array. In our case, it is `Math.pow(item,2)`.

Notice that the custom function should return a value.

Let's modify this example. We are going to use the newer syntax to create functions:

```

/*
    Original array
*/
var arr = [3, 4, 7];
/*
    3 4 7
*/
console.log(arr);

/*
    Math.pow function is called
    for each element in the arr array
    map returns a new array,
    whose elements are arr elements, upon
    the Math.Math.pow call for each of them
    The original array does not change

*/
var mArr = arr.map((item) => {
    return Math.pow(item, 2);
});
/*
    9 16 49
*/
console.log(mArr);

```

Why write **function** if we can omit it 😊. The function body does not change.

We may not write **return** because there is only one line in the function body. New code:

```

/*
    Original array
*/
var arr = [3, 4, 7];

```

```

/*
  3 4 7
*/
console.log(arr);

var mArr = arr.map(item => Math.pow(item, 2));
/*
  9 16 49
*/
console.log(mArr);

```

We do not write `()`, `{}`, `return` in the code. We do not write `()` because our function has one parameter, which means that `()` can be omitted. We do not write `{}`, because our function has one line, which means that `{}` can be omitted. The `return` keyword will be plugged automatically with this type of syntax.

Let's consider another example. We will operate with an array of objects in it. Our object will contain last name and age. The custom function will increase the age by 10 years.

```

var users = [{ lastName: "Brown", age: 30 },
  { lastName: "Davids", age: 40 }];

/*
  In the zeroth element the age is 40
  In the first element 50
*/
var mUsers = users.map(item => {
  item.age += 10;
  return item;
});

```

The `users` array contains user information. Each element in the array is an object. The custom function takes an array

element (specific object) as a parameter. It will be called for each object from the `users` array. We increase the age of the current element by 10 years inside the function. The function returns the modified element. The result of the `map` is returning a new array with objects inside. The age of each object will be increased by 10 years compared to the corresponding element inside the `users`. The original `users` array does not change.



# Using arrays in React Apps

We have not yet worked with data arrays in our React apps. Now is the time to use a data array and display its values on the screen. When can this come in handy? For instance, we can get a data array upon a request to a web service.

Before diving into the new topic, let's recall the destructuring of objects. We worked with it in the previous lesson. Let's analyze the code:

```
let user = {name:"Dan",lastName:"Brown"};

let {...concreteU} = user;

console.log(concreteU);
```

We created a `user` object in this code. It has two fields: `name` and `lastName`. We create a new `concreteU` object using the destructuring syntax `{...concreteU}`. The content and arrangement of the `user` object will be copied to it. It means that there will be two fields inside the `concreteU`: `name` and `lastName`. The value of `name` is `Dan`. The value of `lastName` is `Brown`. You will run against this syntax later in this lesson.

Let's start with the basic example of using an array. We will display array elements in an app. In the code below, we have an object array with information about writers (first and last name). We will display it on the screen. This is the appearance of our app (Fig. 1).

Dan Brown
Joanne Rowling
Stephen King

Figure 1

To solve this problem, let's create some more functional components. The first component will contain information about one writer. Let's call it `Writer`. The second component will display a list of all writers and information about them. Let's call it `WritersList`.

*App.js* code:

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

/*
  Object array
*/
let writers = [
  { name: "Dan", lastName: "Brown" },
  { name: "Joanne", lastName: "Rowling" },
  { name: "Stephen", lastName: "King" }
];

/*
  Component with information about a writer
*/
function Writer(props) {
  return (
    <>
```

```

        <div>
          {props.name} {props.lastName}
        </div>
      <hr />
    </>
  );
}

/*
  A list of writers
*/
function WritersList() {
  return (
    <div>
      <Writer {...writers[0]} />
      <Writer {...writers[1]} />
      <Writer {...writers[2]} />
    </div>
  );
}

export default function App() {
  return (
    <>
      <WritersList />
    </>
  );
}

```

There are three components and one array in our code. Let's begin to analyze the example with the array

```

/*
  Object array
*/
let writers = [

```

```

{ name: "Dan", lastName: "Brown" },
{ name: "Joanne", lastName: "Rowling" },
{ name: "Stephen", lastName: "King" }
];

```

The **writer** object array has information about writers. Each element is an individual writer. The **writers** array is declared globally in our code. This is done to simplify the code.

The **Writer** component is responsible for displaying writer's data.

```

/*
  Component with information about a writer
*/

function Writer(props) {
  return (
    <>
      <div>
        {props.name} {props.lastName}
      </div>
      <hr/>
    </>
  );
}

```

Information about the writer will be passed inside the component using **props**.

The **WritersList** component will display the writer array. It means that we will use an object of a specific writer inside its code (an object of the **Writer** component).

```

/*
  A list of writers
*/
function WritersList() {
  return (
    <div>
      <Writer {...writers[0]} />
      <Writer {...writers[1]} />
      <Writer {...writers[2]} />
    </div>
  );
}

```

The major new thing in the code of `WritersList` is the following construct:

```
<Writer {...writers[0]} />
```

We create a `Writer` object and pass the `writers[0]` as its `props`. The destructuring syntax does the job here. The content of `writers[0]` will be copied to `props`. It means that the properties `name` and `lastName` will be copied inside the `props`. Their values will be copied from the relevant fields of `writers[0]`.

Our array has three elements, so we repeat the creation of the `Writer` component three times in the code with different indices (0,1,2).

The code of the `App` component is familiar to us:

```

export default function App() {
  return (
    <>
      <WritersList />
    </>
  );
}

```

We created an object of a writers list and nothing more.

- [Link](#) to the project code.

Our array code has a small flaw. We explicitly indicate each array element in `WritersList`. We did it three times in the code above. It is clumsy and non-scalable. Let's fix this in the new version of the code. The interface remains the same. All changes occur in the display logic.

*App.js* code:

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

var writers = [
  { name: "Dan", lastName: "Brown" },
  { name: "Joanne", lastName: "Rowling" },
  { name: "Stephen", lastName: "King" }
];

function Writer(props) {
  return (
    <>
      <div>
        {props.name} {props.lastName}
      </div>
      <hr />
    </>
  );
}

function WritersList(props) {
  return (
    <div>
      {props.data.map(item => <Writer {...item} />)}
    </div>
  );
}
```

```
export default function App() {  
  return (  
    <>  
      <WritersList data={writers} />  
    </>  
  );  
}
```

Our code has two new things. The first one is inside the `App` code:

```
export default function App() {  
  return (  
    <>  
      <WritersList data={writers} />  
    </>  
  );  
}
```

We pass the `writers` array through the `data` property inside the `WritersList` component.

There are also changes inside the `WritersList`.

```
function WritersList(props) {  
  return (  
    <div>  
      {props.data.map(item => <Writer {...item} />)}  
    </div>  
  );  
}
```

We no longer have code with each element of the `writers` array. Instead, we use an already familiar `map` method. It returns the array of the `Writer` components.

```
{props.data.map(item => <Writer {...item} />)}
```

This syntax may look frightening, but we have already encountered each of them before. Let's analyze it piece by piece:

```
props.data.map(
```

Call the `map` method for the `data` property (it refers to the `writers` array).

```
item => <Writer {...item} />
```

This code is called for each element of the `writers` array. An element for a new array is returned in this code. In fact, we create a new `Writer` element and fill its `props` with the values of the current element being analyzed.

The `map` call will be replaced with a newly created `Writer` array.

Thanks to `map`, we got rid of the necessity to indicate each element manually.

- [Link](#) to the project code.



# Key

You have probably noticed that there is a problem in our data array example 😊.

If you have not noticed it, we will point it out at once. The online code editor [CodeSandbox](#) has an interface block that displays the console. If you open our array example there, you will see something like this in the lower right corner:

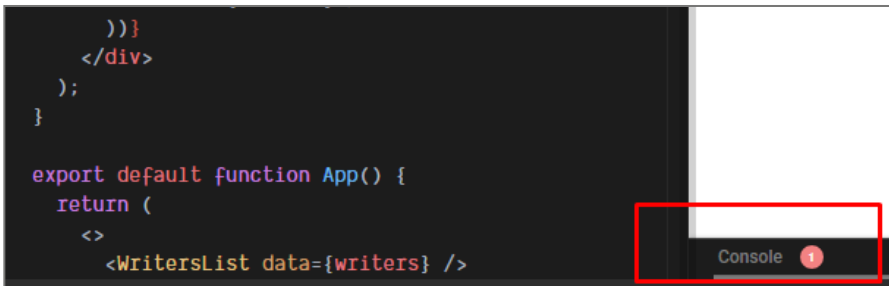


Figure 2

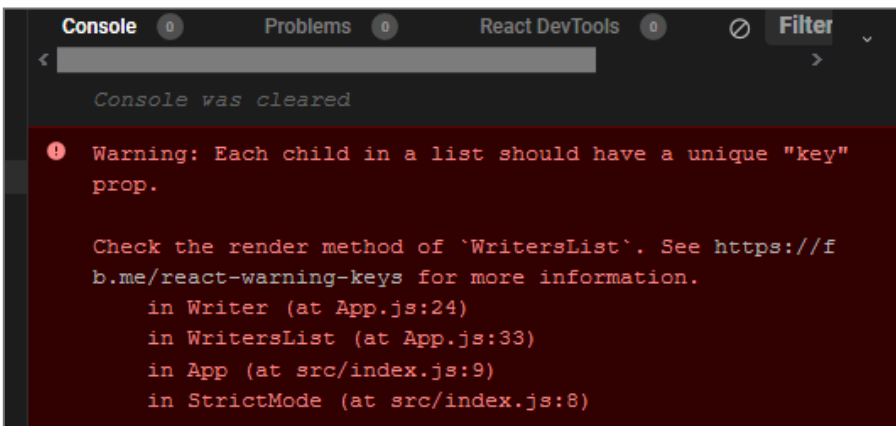


Figure 3

The red 1 indicates that we have a problem. Open the console and read this message (Fig. 7).

What does it mean? React warns us that each element in the list should have a unique key. We set a value for the element. But we did not set values for keys.

What is a key? It is a unique value that will never change. React uses keys in order to understand which elements were added, deleted, modified. If you want to be sure that the interface updates correctly, you must specify keys.

Key uniqueness must be maintained within one list. It means that keys in different lists can be not unique.

Keys are set with a `key` attribute.

Let's consider an example of working with keys. There will be an array of cities to be displayed in our example. Each item of the city list will have a unique key set:



Figure 4

*App.js* code:

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

const cities = ["London", "Paris", "Tokio"];

function City(props) {
  return <div>{props.value}</div>;
}
```

```
function CityList(props) {
  const citiesData = props.data.map(
    city => <City key={city} value={city} />
  );
  return <>{citiesData}</>;
}

export default function App() {
  return (
    <>
      <CityList data={cities} />
    </>
  );
}
```

This code works similarly to the previous one. We have components for the city and a list of cities. Let's begin the analysis with the `cities` array and the `City` component.

```
const cities = ["London", "Paris", "Tokio"];
function City(props) {
  return <div>{props.value}</div>;
}
```

The array with city names is declared. The `City` component displays a name of a specific city. The code of the `CityList` component:

```
function CityList(props) {
  const citiesData = props.data.map(
    city => <City key={city} value={city} />
  );
  return <>{citiesData}</>;
}
```

In this example, we took the `map` call away from `JSX`. We did it to show you another possible approach. The parameter function for `map` looks differently:

```
city => <City key={city} value={city} />
```

We say that an element of the list of cities has two properties. The first property is `value` — the city name. The second property is `key` — the key for the current line being created. The key is not displayed in the interface. The user does not see it. This key is important to `React`. We used a city name as a key in this example. If the city name occurs in the array more than once, then our approach stops working.

In order to avoid potential problems, developers use identifiers as a key. We will show you this technique in another example.

We do not use destructuring syntax because we have a string array, not object array.

The code of the `App` component:

```
export default function App() {  
  return (  
    <>  
      <CityList data={cities} />  
    </>  
  );  
}
```

We passed the array through the `data` property.

► [Link](#) to the project.

It is essential to remember that the key should be passed wherever the list of cities is created, not inside the City component. The code below is wrong:

```
function City(props) {  
  return <div key={props.value}>{props.value}</div>;  
}  
  
function CityList(props) {  
  const citiesData = props.data.map(city =>  
    <City value={city} />);  
  return <>{citiesData}</>;  
}
```

We set the `key` attribute inside the City component in the `div` tag. This approach does not let React identify the list strings. The console warning will tell you about this:

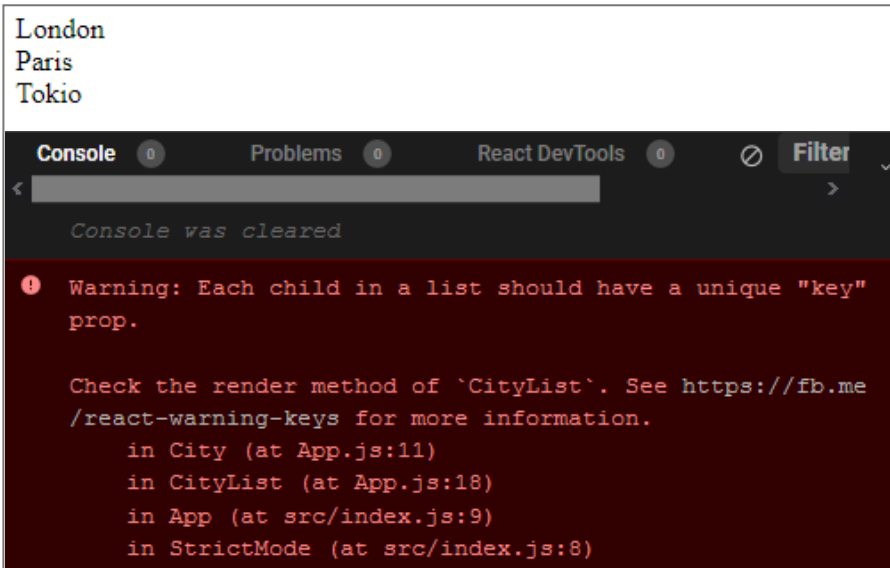


Figure 5

Let's change the writers code by implementing key support. We will use unique identifiers as a key. For this, we will add the `id` property inside the writer object.

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

var writers = [
  { id: 1, name: "Dan", lastName: "Brown" },
  { id: 2, name: "Joanne", lastName: "Rowling" },
  { id: 3, name: "Stephen", lastName: "King" }
];

function Writer(props) {
  return (
    <>
      <div>
        {props.name} {props.lastName}
      </div>
      <hr />
    </>
  );
}

function WritersList(props) {
  return (
    <div>
      {props.data.map(item => (
        <Writer key={item.id} {...item} />
      ))}
    </div>
  );
}

export default function App() {
  return (
    <>
```

```

    <WritersList data={writers} />
  </>
);
}

```

What has changed in the code? First, an identifier has appeared inside the writer object: the **id property**.

```

var writers = [
  { id: 1, name: "Dan", lastName: "Brown" },
  { id: 2, name: "Joanne", lastName: "Rowling" },
  { id: 3, name: "Stephen", lastName: "King" }
];

```

Be reminded that values for **id** should be unique.

Second, the code of the **WritersList** component has changed.

```

function WritersList(props) {
  return (
    <div>
      {props.data.map(item => (
        <Writer key={item.id} {...item} />
      ))}
    </div>
  );
}

```

When describing the **Writer** component, we explicitly specify the **key** attribute and set its value to **item.id**.

There are no more changes in our code. The use of **key** made the warning disappear, but most importantly, now React will be able to tell one string of the list from another.

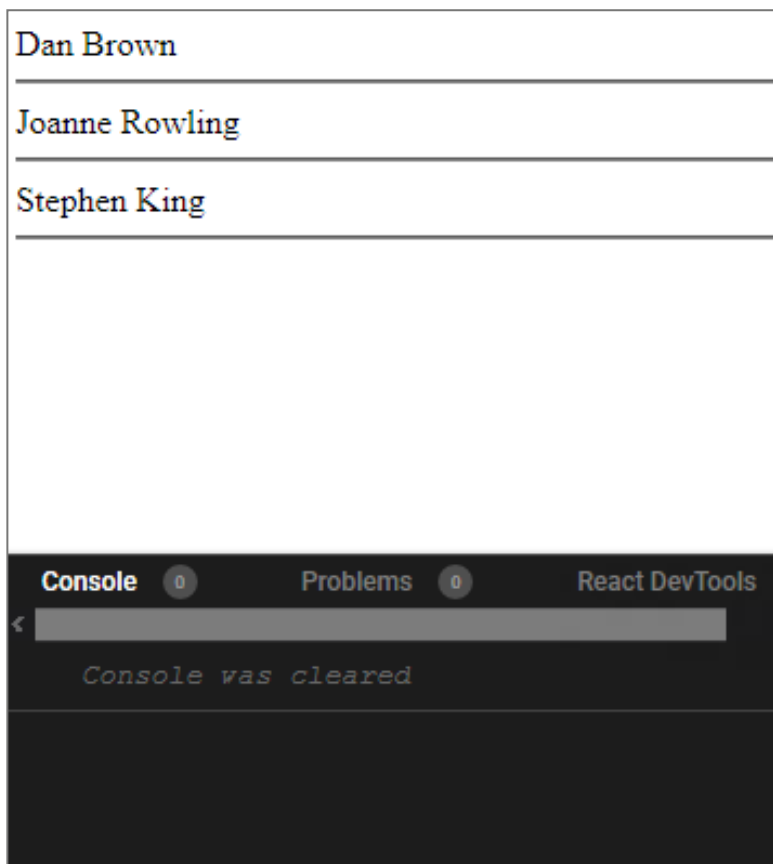


Figure 6

- [Link](#) to the project code.



# Forms

You are already familiar with the mechanism of forms in HTML. It is time to discuss how React works with forms using basic mechanisms. In order to dive, let's begin with a basic example of form display.

A screenshot of a web form. It features a single-line text input field on the left, which is empty. To the right of the input field is a button with the text "Click me". The entire form is enclosed in a thin rectangular border.

Figure 7

Our UI has a text box and a button. Nothing unusual. The code of our form is as follows:

```
function Form() {  
  return (  
    <form>  
      <input type="text" />  
      <input type="submit" value="Click me" />  
    </form>  
  );  
}
```

We use typical tags for working with forms in order to create one. The `form` tag declares a form. Two `input` tags create a text box and a `submit` button (used to send information to the server). In the first `input`, the `type` attribute is equal to `text`, which means that we create a text box. In order to create the `submit` button, we indicate `submit` as we describe the `type` attribute. If the user presses the `submit` button, form data will be sent and the page will reload.

We did not use any attributes for the `form` tag. It means that we did not specify the path to the server-side script that will analyze the form data. In real life, you will need to do this using the `action` attribute. So, where do data go if `action` is not specified? Data from the form on a page will be sent to the page itself. Let's say our form is in the `myform.php` file on the server; when the submit button is clicked, data will be sent to `myform.php`.

► [Link](#) to the project code.

Can we slow down the sending of data upon pressing the `submit` button? Of course, we can! And we will use the `preventDefault` for this. Let's show this with an example:

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

function Form() {
  const handleSubmit = event => {
    /*
     * Cancel the response of the default handler
     */
    event.preventDefault();
    /*
     * Get access to the text box
     */
    let uName = document.getElementById("userName");
    alert(uName.value);
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" id="userName" />
      <input type="submit" value="Click me" />
    </form>
  );
}
```

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

The appearance of our app has not changed. The difference is only in the code.

```
<form onSubmit={handleSubmit}>
```

When describing the form, we specified the processing of data sending. An event occurs when the user presses the [submit](#) button inside the form.

```
const handleSubmit = event => {

  /*
    Cancel the response of the default handler
  */
  event.preventDefault();

  /*
    Get access to the text box
  */
  let uName = document.getElementById("userName");
  alert(uName.value);
};
```

In the code of our handler, we cancel the default event handling by calling the [preventDefault](#). After this we get access to the text box and display its value.

In order to get a link to the text box, we use the already familiar `getElementById`.

- ▶ [Link](#) to the project code.

But is it possible to get access to the value of a form element without using `getElementById`? Of course! The answer is in the next section.

# Refs

**Refs** are tools for binding a form element to some variable. Due to this binding you can exchange values with a control. The one responsible for this interaction is [React](#).

A ref variable is created with the [React.createRef\(\)](#) method. Steps to create a ref variable:

1. Create a ref variable by calling [createRef](#);
2. Attach it to a specific control through the [ref](#) attribute.

After we complete these steps, our variable is ready to go! Let's create an app with this interface:



Figure 8

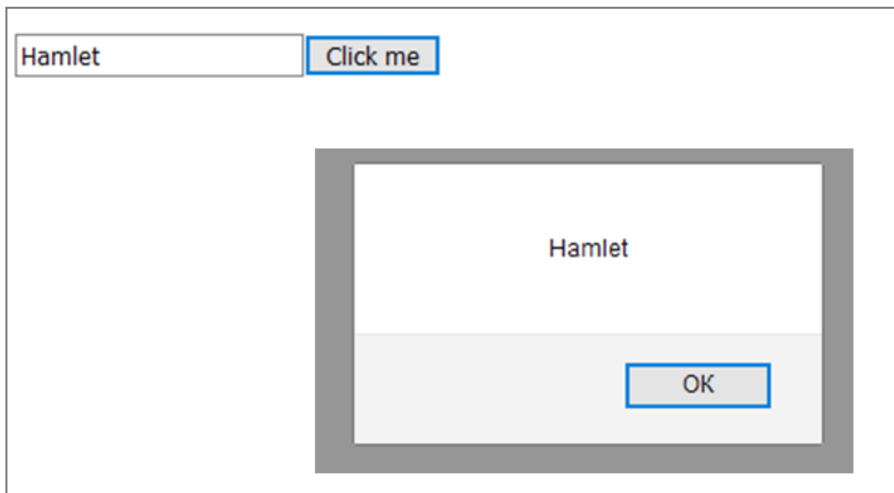


Figure 9

By clicking on the button we display a message box with the value entered in the text box. Data are not sent to the server because we will call `preventDefault` in the `onSubmit` handler. The appearance of the app upon button click (Fig. 4).

In our example, we are going to create a functional component and a ref variable.

The *App.js* code:

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

function Form() {
  let uRef = React.createRef();
  const handleSubmit = event => {
    event.preventDefault();
    alert(uRef.current.value);
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" id="userName" ref={uRef} />
      <input type="submit" value="Click me" />
    </form>
  );
}

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

Important steps in the code.

```
let uRef = React.createRef();
```

Creation of a ref variable. It is not bound to any control yet.

```
const handleSubmit = event => {  
  event.preventDefault();  
  alert(uRef.current.value);  
};
```

In the `onSubmit` handler, we access the value of a control to which the ref variable will be bound. The construct `uRef.current.value` is used for this.

```
<input type="text" id="userName" ref={uRef} />
```

Here we link the text box and the ref variable. The link occurs through the `ref` attribute. All the issues on sending data are solved by React.

► [Link](#) to the project code.

The second example of working with ref variables will be implemented through class components.

The appearance of the app:

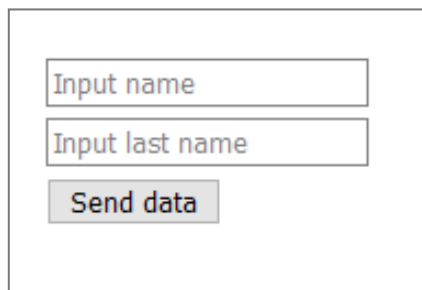


Figure 10

Data will not be sent upon button click. We will display a window with data received from the user.

*App.js* code:

```

import React, { Component } from "react";
import "./styles.css";

class Form extends Component {
  nameRef = React.createRef();
  lastNameRef = React.createRef();
  render() {
    const handlerSubmit = event => {
      event.preventDefault();
      let str = this.nameRef.current.value+" ";
      str += this.lastNameRef.current.value;
      alert(str);
    };
    return (
      <form onSubmit={handlerSubmit}>
        <div className="formElement">
          <input
            type="text"
            placeholder="Input name"
            ref={this.nameRef}
            required
          />
        </div>
        <div className="formElement">
          <input
            type="text"
            placeholder="Input last name"
            ref={this.lastNameRef}
            required
          />
        </div>
        <div className="formElement">
          <input type="submit" value="Send data" />
        </div>
      </form>
    );
  }
}

```



```

    );
  }
}

export default class App extends Component {
  render() {
    return (
      <>
        <Form />
      </>
    );
  }
}

```

We would like to draw attention to important points in the code.

```

import React, { Component } from "react";

import "./styles.css";

class Form extends Component {

```

We imported `Component` to cut the notation of `React.Component` to `Component` upon inheritance.

```

class Form extends Component {
  nameRef = React.createRef();
  lastNameRef = React.createRef();

```

The creation and usage of ref variables works just like in functional components. We created two ref variables that we are going to use to access controls.

```
const handlerSubmit = event => {  
  event.preventDefault();  
  let str = this.nameRef.current.value+" ";  
  str += this.lastNameRef.current.value;  
  alert(str);  
};
```

The code of the onSubmit handler differs from the previous example only in the presence of `this` keyword.

```
<input  
  type="text"  
  placeholder="Input name"  
  ref={this.nameRef}  
  required  
>
```

A control can also be bound through the `ref` attribute.

- [Link](#) to the project code.

# State and Forms

We are already used to the fact that React is responsible for the state and interface update in our app, however controls in form update their appearance on their own. For instance, if a user enters a letter into a text box, it automatically appears there.

May we need React for this process? Of course. There may be many reasons for this.

For example, in order to maintain a uniform approach to the entire UI interface of the app. Or maybe we want to task React with displaying text in a box if the user did not enter a forbidden value.

The well-known mechanism of state comes to our aid.

Let's start with an example where we will assign the text box update to React.

The appearance of our app:


A screenshot of a web form. It consists of a rectangular container. Inside the container, on the left, is a text input field with the placeholder text "Input name". To the right of the input field is a button with the text "Click me".

Figure 11

In this example, if you try to input text in the text box, nothing will happen.

```
import React, { useState } from "react";
import "./styles.css";

function Form() {
  const [nameState, setNameState] = useState("");
  return (
```

```

    <form>
      <input type="text" placeholder="Input name"
        value={nameState} required />
      <input type="submit" value="Click me" />
    </form>
  );
}

export default function App() {
  return (
    <>
      <Form />
    </>
  );
}

```

We create a state variable in the code.

```
const [nameState, setNameState] = useState("");
```

In order to assign the state variable to the text box, we use the **value** property.

```

<input type="text" placeholder="Input name"
  value={nameState}

```

To assign the ref variable, we used the **ref** property.

Why is the text not displayed? This is due to the fact that we assigned the element update to React but did not specify the logic of this process.

- [Link](#) to the project code.

The update logic is implemented through the **onChange** handler for the text box.

Let's now modify the example so that the typed text appears in the text box.

```
import React, { useState } from "react";
import "./styles.css";

function Form() {
  const [nameState, setNameState] = useState("");
  const handlerSubmit = event => {
    event.preventDefault();
    alert(nameState);
  };

  const handlerChange = event => {
    setNameState(event.target.value);
  };

  return (
    <form onSubmit={handlerSubmit}>
      <input
        type="text"
        placeholder="Input name"
        value={nameState}
        onChange={handlerChange}
        required
      />
      <input type="submit" value="Click me" />
    </form>
  );
}

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

When creating the text box, we specified that we have a change handler there.

```
<input
  type="text"
  placeholder="Input name"
  value={nameState}
  onChange={handlerChange}
  required
/>
```

This is a `handlerChange` function.

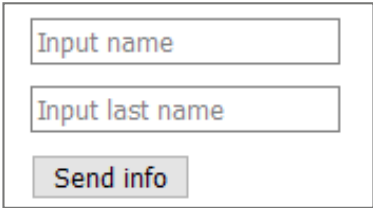
```
const handlerChange = event => {
  setNameState(event.target.value);
};
```

We take the current text being typed by using `event.target.value` and update the state through `setNameState` in the function body. After the state variable is updated, React automatically updates the control it is assigned to.

► [Link](#) to the project code:

The next example will analyze the text typed by the user and cancel input if the current value does not suit us.

The appearance of the app:



The image shows a simple web form layout. It consists of two text input fields stacked vertically. The first field has the placeholder text 'Input name' and the second has 'Input last name'. Below these two fields is a button with the text 'Send info'. The entire form is enclosed in a thin black border.

Figure 12

We will forbid to enter `test` as a name. This project was created using class components.

*App.js* code:

```
import React, { Component } from "react";
import "./styles.css";

class UserForm extends Component {
  state = {
    nameState: "",
    lastNameState: ""
  };

  render() {
    const handlerSubmit = event => {
      event.preventDefault();
      alert(this.state.nameState + " " +
        this.state.lastNameState);
    };

    const handlerNameChanged = event => {
      let name = event.target.value;
      if (name.trim().toUpperCase() === "TEST") {
        alert("Wrong name!");
        this.setState({ nameState: "" });
      } else {
        this.setState({ nameState:
          event.target.value });
      }
    };

    const handlerLastNameChanged = event => {
      this.setState({ lastNameState:
        event.target.value });
    };

    return (
```

```

    <form onSubmit={handlerSubmit}>
      <input
        type="text"
        className="formElement"
        placeholder="Input name"
        value={this.state.nameState}
        onChange={handlerNameChanged}
        required
      />

      <input
        type="text"
        className="formElement"
        placeholder="Input last name"
        value={this.state.lastNameState}
        onChange={handlerLastNameChanged}
        required
      />

      <input type="submit"
        className="formElement"
        value="Send info" />
    </form>
  );
}
}

export default class App extends Component {
  render() {
    return (
      <>
        <UserForm />
      </>
    );
  }
}

```



We create two state variables, one for each text box.

```
class UserForm extends Component {  
  state = {  
    nameState: "",  
    lastNameState: ""  
  };  
};
```

We are the ones who use the set properties, not the constructor, as we have done many times before.

To check the value in the text box, we use the `onChange` event handler:

```
const handlerNameChanged = event => {  
  let name = event.target.value;  
  if (name.trim().toUpperCase() === "TEST") {  
    alert("Wrong name!");  
    this.setState({ nameState: "" });  
  } else {  
    this.setState({ nameState: event.target.value });  
  }  
};
```

If the value is `test`, we change the state variable by writing an empty value to it.

If the value is not `test`, we update the state variable by writing the current value to it.

Analyze the code very carefully and check how it works.

- [Link](#) to the project code.

# Homework

---

1. Create an app Personal Page. Use text boxes to display pieces of information. For example, name, phone number, email, city of residence, etc. The presence of text boxes allows the user to modify the source data. Also, add a button that returns the initial values. Use `state` and class components.
2. Create a registration form. The user should enter his nickname, email, gender, age. Use React to work with forms. Use React for validation of the entered values.
3. Create a photo upload form. The user should enter his nickname, password, email, photo, description of the photo, tags. Use React to work with forms. Use React for validation of the entered values.
4. Create an app to display information about the cities in your country. Be sure to use object arrays and the `map` function.
5. Create an app to display information about your favorite bands. Be sure to use object arrays, the `map` function, and class components.





## Lesson 7

# React: Advanced Techniques

© STEP IT Academy, [www.itstep.org](http://www.itstep.org).

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.