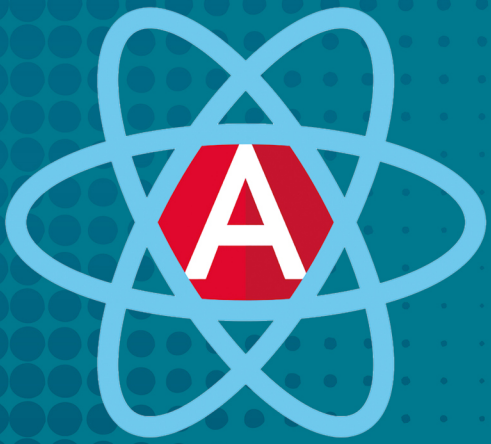
Building Web Applications Using Angular & React



STEP computer ACADEMY

Lesson 8

React: Advanced Techniques

Contents

And a Bit More About Forms

There are two very difficult things in everything you do. It is always hard to start. And it is always very hard to continue. Continuing is much harder than starting. If you read this, it means you continue to study React. Good job! You are doing great!

We will once again dig into specific features of forms and talk about some aspects of their use that we have not studied earlier. Let's begin with the example that has the following UI.

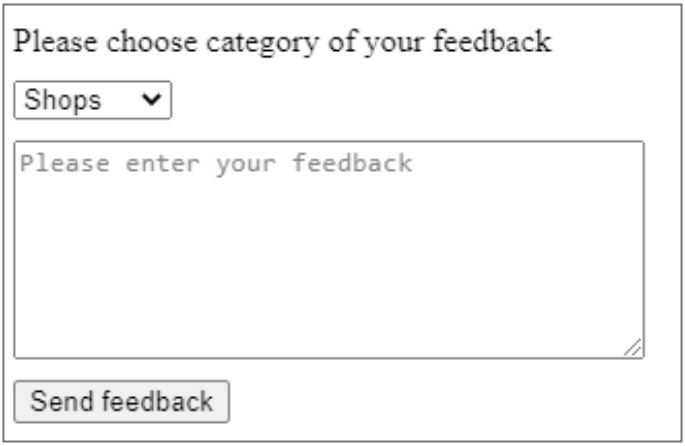


Figure 1

The user selects a feedback category from the list. Then she types text and sends it by clicking on the Send feedback button. We will display an information message with the data in the code without sending data to the server.

We will use a list and a large text box in this example. The code of *UserForm.js* (we have renamed *App.js*):

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

export default function UserForm() {
  const [content, setContent] = useState("");
  const [selectedItem, setSelectedItem] =
                                  useState("Shops");

  const handlerTextAreaChanged = event => {
    setContent(event.target.value);
  };
  const handlerSelectChanged = event => {
    setSelectedItem(event.target.value);
  };
  const handlerSubmit = event => {
    event.preventDefault();
    const msg = 'Your feedback about ${selectedItem}:
                 \n${content}';
    alert(msg);
  };

  return (
    <div>
      <form className="userForm"
            onSubmit={handlerSubmit}>
        <label>
          Please choose category of your feedback
          <select value={selectedItem}
                  onChange={handlerSelectChanged}>
            <option>Service</option>
            <option>Products</option>
            <option>Shops</option>
          </select>
        </label>
        <textarea
          value={content}
```

```
            onChange={handlerTextAreaChanged}
            placeholder="Please enter your feedback"
            required
          />

          <input type="submit" value="Send feedback" />
        </form>
      </div>
    );
}
```

In the example code, we renamed the App component to UserForm. Besides, we added the new name wherever necessary. The tags textarea and select are used to create a text box and a list. Let's begin with the code that creates textarea.

```
<textarea
    value={content}
    onChange={handlerTextAreaChanged}
    placeholder="Please enter your feedback"
    required
/>
```

To bind a state variable and a control, we use the value attribute. To display changes in the text box, we created the onChange handler.

The code that creates a list:

```
<select value={selectedItem}
        onChange={handlerSelectChanged}>
    <option>Service</option>
    <option>Products</option>
    <option>Shops</option>
</select>
```

We used the select tag to create a list. And we use option to fill the list with lines. And again, we use value and onChange. The list item, whose value was specified in value, will be highlighted. It is a line named Shops at the start. Notice that a regular HTML uses the selected attribute to highlight a line.

```
const [content, setContent] = useState("");
const [selectedItem, setSelectedItem] =
       useState("Shops");
```

As for the code for working with the state, you already know it. We use the state hook twice. The content variable will be responsible for the text box. And the selectedItem for the list.

```
  const handlerTextAreaChanged = event => {
    setContent(event.target.value);
  };
  const handlerSelectChanged = event => {
    setSelectedItem(event.target.value);
  };
  const handlerSubmit = event => {
    event.preventDefault();
    const msg = 'Your feedback about ${selectedItem}:
                 \n${content}';
    alert(msg);
  };
```

The code of the onChanged handler for controls is pretty similar. The only difference is how the function is called. It is setContent for a text box and setSelectedItem for a list.

The code of the onSubmit handler should not be hard for you.

- Link to the project code.

In the last example, the code of the onChange handler for various controls was quite similar. In the new example, we will create one handler for two controls. And we will use the checkbox control. The appearance of the app is as follows:

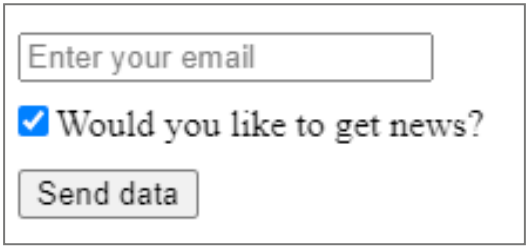


Figure 2

The user enters her email address. Specifies if she needs a newsletter subscription. After this, she clicks the Send data button. We will display an information message with her data without sending data to the server. The code of *UserForm.js*:

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

export default function UserForm() {
  const [news, setNews] = useState(true);
  const [email, setEmail] = useState("");
  const handlerSubmit = event => {
    event.preventDefault();
    let msg = "";
    if (news === true) {
      msg = "Thank you for subscription!\n";
    }
    msg += "Your email:" + email;
    alert(msg);
  };
```

```
  /*
    One event handler for two inputs
  */
  const handlerChanged = event => {
    const target = event.target;
    /*
      Check which element triggered the event
    */
    target.name === "aboutNews"
      ? setNews(event.target.checked)
      : setEmail(target.value);
  };

  return (
    <div>
      <form className="userForm"
            onSubmit={handlerSubmit}>
        <input
          name="userEmail"
          type="email"
          required
          placeholder="Enter your email"
          value={email}
          onChange={handlerChanged}
        />
        <input
          type="checkbox"
          name="aboutNews"
          checked={news}
          onChange={handlerChanged}
        />
        <label>Would you like to get news?</label>
        <input type="submit" value="Send data" />
      </form>
    </div>
  );
}
```

We use the input tag to display checkbox.

```
<input
    type="checkbox"
    name="aboutNews"
    checked={news}
    onChange={handlerChanged}
/>
```

The checked property determines whether the checkbox is checked. We bind our state variable to this property. We add the handlerChanged handler function to onChange. We specified that the checkbox attribute name is set to aboutNews. We will need this value when we check which control triggers the onChange handler.

We also use the input tag to create an input field for an email address.

```
<input
    name="userEmail"
    type="email"
    required
    placeholder="Enter your email"
    value={email}
    onChange={handlerChanged}

/>
```

We specified that the attribute name of the email input field is set to userEmail. We will need this value when we check which control triggers the onChange handler. We bind the same handlerChanged handler to the onChange. Here is its code:

```
const handlerChanged = event => {
  const target = event.target;
  /*
     Check which element triggered the event
  */
  target.name === "aboutNews"
    ? setNews(event.target.checked)
    : setEmail(target.value);
};
```

We should check which control called the handler. For this we use the target.name property. If it is set to aboutNews, then the handler was triggered by the checkbox. And we need to update its state; otherwise, the handler was called for a text box. And in this case it is it that needs a state update. Analyze the entire code of the example again and be attentive in order to better understand it.

- Link to the project code.

Routes

You know the concept of a route from real life. We are sure that you have favorite routes for walking around the city — Routes are vital to us. We cannot get from point A to point B without a clear route.

And we also need the mechanism of routes to develop apps using React. It will allow us to integrate navigation options into our projects.

For now we use one address in all our projects, but any web app uses different addresses. For instance, one to display information about company, the other to display information about employees, etc.

In order to add the routing mechanism, we should use the react-router-dom module in the project.

It is not added to the project template at CodeSandbox by default. In order to add it, you should click on the button "Add dependency" in the lower left corner.

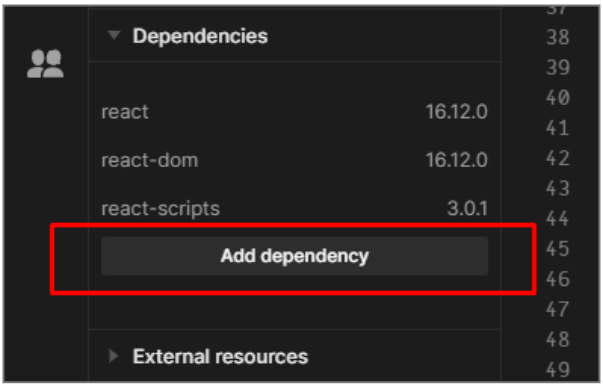


Figure 3

In the window for adding modules, type react-router-dom.

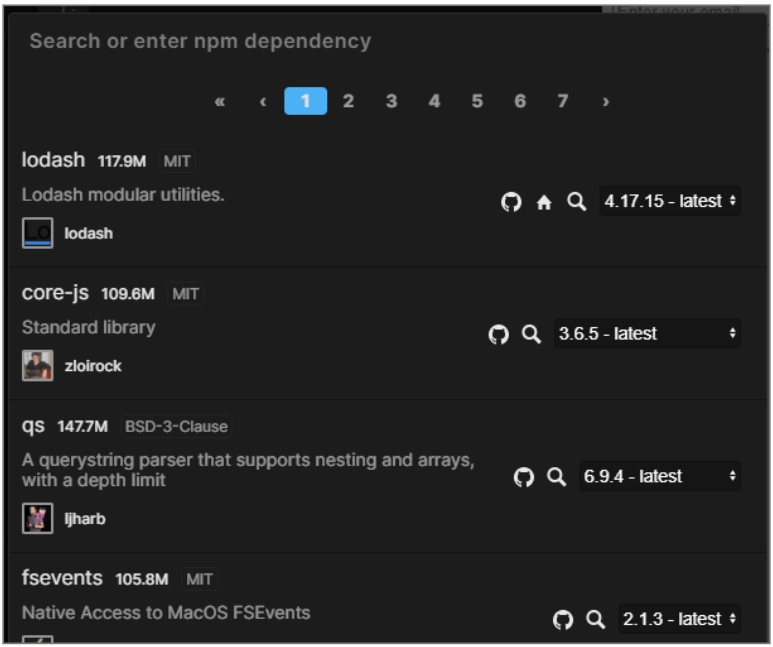


Figure 4

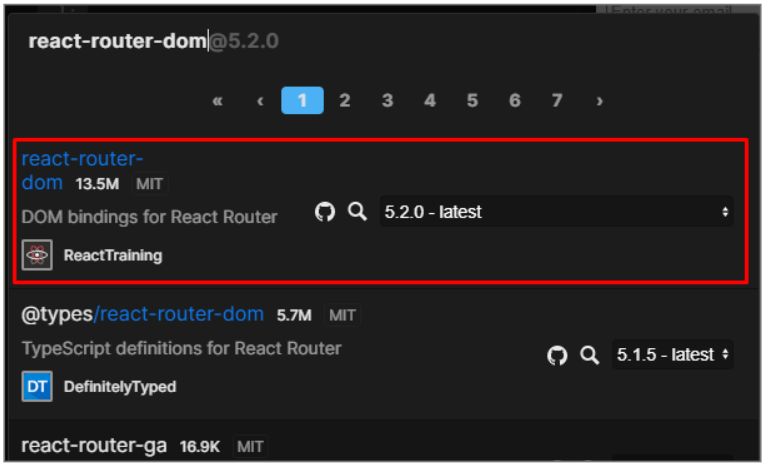


Figure 5

Click on the first link to add this module in our project. If everything goes well, react-router-dom appears in the Dependencies window of our project.

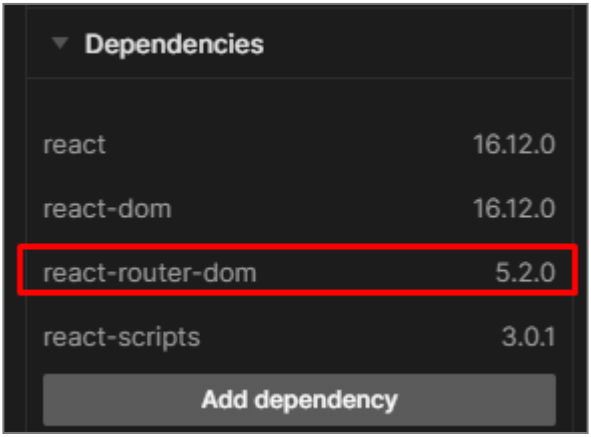


Figure 6

The first steps of the prearrangement are made. Now we will begin to analyze how to integrate routing into our projects.

In our first example, we will configure several routes for users to visit web apps. We will have three routes. The first will lead to the main page, the second to the About Company page, the third to the News page. If the user tries to follow an unknown route, we will display a message that this page has not been found.

Visiting the main page:

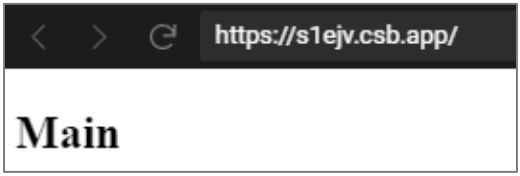


Figure 7

Notice that the main page opens if the root address of the website is specified. We did not indicate any additional path.

Visiting the About company page:

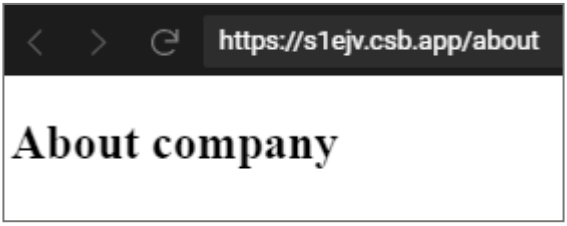


Figure 8

In order to get to this page, we added /about to the path.

Visiting the news page:

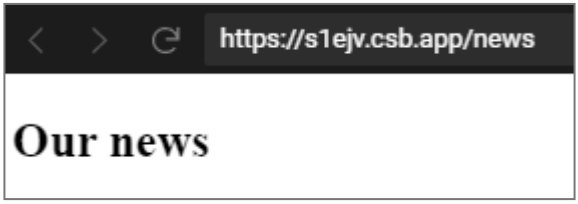


Figure 9

In order to get to this page, we added /news to the path.

We will begin to analyze the code with the overview of the *App.js* code:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
from "react-router-dom";
import "./styles.css";

function Main() {
  return <h2>Main</h2>;
}
```

```
function AboutCompany() {
  return <h2>About company</h2>;
}
function News() {
  return <h2>Our news</h2>;
}
function NotFound() {
  return <h2>Not found</h2>;
}
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
                 component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}
```

In order to use routing options in the code, we not only need to add a dependency but import a number of objects from react-router-dom.

```
import { BrowserRouter as Router, Route, Switch }
       from "react-router-dom";
```

We specified that we import BrowserRouter and named it Router. Renaming was optional. We could also use the default name.

BrowserRouter will contain all routes. It can be called a route aggregator. Switch helps to choose just one suitable route. Route is a specific route. Let's consider the code for creating routes.

```
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
                 component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}
```

All routes are enclosed inside Router. Besides, all routes are enclosed in Switch.

```
<Router>
    <Switch>
        Routes
    </Switch>
</Router>
```

You may wonder if it is possible to remove Switch. It is. We will show you the consequences of this later. For now just believe us that this is what the example needs.

A specific route is described as follows:

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

We mentioned that when accessing the route address, we should upload the Main component. The route is specified with the path attribute. To specify a component's name, you should use the component attribute. As for exact, we will talk about it later.

The code of the Main component:

```
function Main() {
  return <h2>Main</h2>;
}
```

There is nothing unusual in the code. Let's consider the creation of another route:

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

When accessing the address root_address/about, you need to upload the AboutCompany component.

The NotFound route is described in a similar way. Let's try to activate it.

Figure 10

We tried to access an unknown address. The NotFound component was activated in response.

Let's ask ourselves what happens if we add one more address to the routes we defined. For example, we could add a city name to the path to about: route_address/about/london

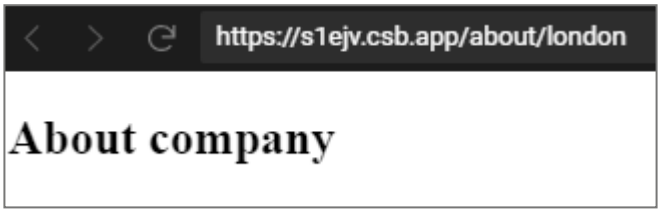


Figure 11

Despite fuzzy matching, the About component is loaded. This is due to the fact that we did not specify exact when describing this route. The exact attribute is used when we need an exact match. If we add exact to the description of the About route, attempting to access a fuzzy route will cause the NotFound component to load (path not found, we asked for an exact match):

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

Figure 12

Let's try to remove exact from the description of the root address.

```
<Route path="/" component={Main} />
```

What do you think will happen? Think for a minute. Correct answer: when attempting to access any address, the Main component will be loaded. This happens because we do not

require an exact address, and the path to the root address is in any path in our app.

The rest of the routes will never be called. Do not forget to specify exact for your root address to avoid these curious consequences.

Now try to remove Switch from our code.

```
<Router>
    <Route exact path="/" component={Main} />
    <Route path="/about" component={AboutCompany} />
    <Route path="/news" component={News} />
    <Route component={NotFound} />
</Router>
```

Let's access the root address:

Figure 13

And About:

Figure 14

Why does the NotFound component loads in addition to the right component?

Switch is responsible for the choice of the first suitable route; after it is found, the rest of the routes are not analyzed. Switch works like the switch construct in any programming language.

There is no Switch in our code yet. It means that the route that will be displayed will be not the suitable one but any that meets the condition. The NotFound component does not have the path attribute. This leads to the conclusion that it fits any path. This is why we see About first, and then NotFound loads.

In order to avoid this effect, use Switch.

- Link to the project code.

The children Attribute

We can describe a component for a specific route at its definition. A children attribute is used for this. Let's modify our code a little bit to illustrate it.

App.js code:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
         from "react-router-dom";

function Main() {
  return <h2>Main</h2>;
}

function AboutCompany() {
  return <h2>About company</h2>;
}
```

```
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route exact path="/about"
                        component={AboutCompany} />
          <Route strict path="/news/"
                 children={() =>
                 <h2>Our news</h2>} />
          <Route children={() => <h2>Not found</h2>}/>
        </Switch>
      </Router>
    </div>
  );
}
```

The code describes several routes. However, we specified the component body for the routes news and NotFound in the children attribute.

```
<Route children={() => <h2>Not found</h2>} />
```

Also, there is a new attribute strict in the code. It requires an even stricter match. It means that the news route should have / because we specified / in the path.

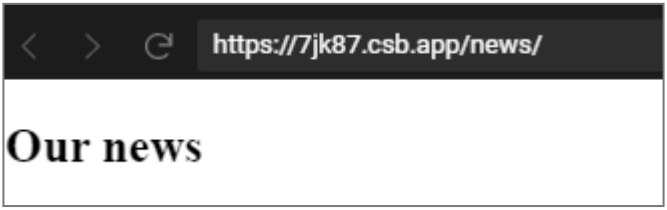


Figure 15

This works. We have a strict match of news/

Figure 16

And this does not work. There is no /.

If there were exact instead of strict, both addresses would cause the load of the News component.

- Link to the project code.

Child Paths

Let's modify our example. Leave the main component, AboutCompany, News, NotFound. Configure child routes to branch news in a specific city inside the News component.

The appearance of the app when accessing the news route:

Figure 17

The appearance of the news route for a specific city:

Figure 18

Let's consider the *App.js* code:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
         from "react-router-dom";
import "./styles.css";

function Main() {
  return <h2>Main</h2>;
}

function AboutCompany() {
  return <h2>About company</h2>;
}

function News() {
  return (
    <div>
      <h2>Our news</h2>
      <Switch>
        <Route path="/news/london"
               component={London}/>;
        <Route path="/news/berlin"
               component={Berlin}/>;
        <Route path="/news/paris"
```

```
                component={Paris}/>;
      </Switch>
    </div>
  );
}

function NotFound() {
  return <h2>Not found</h2>;
}

function London() {
  return <h3>London</h3>;
}

function Paris() {
  return <h3>Paris</h3>;
}

function Berlin() {
  return <h3>Berlin</h3>;
}

export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
                 component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}
```

The news route is set up in a manner familiar to us.

```
<Route path="/news" component={News} />
```

And the body of the News component is implemented in a slightly different way:

```
function News() {
  return (
    <div>
      <h2>Our news</h2>
      <Switch>
        <Route path="/news/london"
               component={London} />;
        <Route path="/news/berlin"
               component={Berlin} />;
        <Route path="/news/paris"
               component={Paris} />;
      </Switch>
    </div>
  );
```

We specify routes to news of a specific city inside the News component. Each route has its own component. For example, London has the London component.

We hope that the mechanism of child routes will not cause you any difficulties.

- Link to the project code.

Creating a Navigation Menu

So far, we created routes only in the code of our projects. In the new example, we will add a navigation menu. With its help the user will be able to activate the route he needs.

The appearance of our app will be as follows:

Figure 19

When clicking on About, the AboutCompany component will open.

Figure 20

Enough of images. Let's dive in the code:

```
import React from "react";
import {BrowserRouter as Router, Route, Switch, Link}
        from "react-router-dom";
import "./styles.css";

function Main() {
  return <h2>Main</h2>;
}

function AboutCompany() {
  return <h2>About company</h2>;
}
```

```
function News() {
  return <h2>Our news</h2>;
}

function NotFound() {
  return <h2>Not found</h2>;
}

function NavMenu() {
  return (
    <>
      <Link to="/" className="links">
        Main
      </Link>
      <Link to="/about" className="links">
        About
      </Link>
      <Link to="/news" className="links">
        News
      </Link>
    </>
  );
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <NavMenu />
          <Switch>
            <Route exact path="/" component={Main} />
            <Route path="/about"
                   component={AboutCompany} />
            <Route path="/news" component={News} />
            <Route component={NotFound} />
          </Switch>
```

```
        </div>
      </Router>
    </div>
  );

}
```

What did we do to add a navigation menu?

```
import {BrowserRouter as Router, Route, Switch, Link}
        from "react-router-dom";
```

We imported Link. It is responsible for creating links. For Link we should specify the to attribute. It is used to set a path for the link. To display a link menu, we created a NavMenu component.

```
function NavMenu() {
  return (
    <>
      <Link to="/" className="links">
        Main
      </Link>
      <Link to="/about" className="links">
        About
      </Link>
      <Link to="/news" className="links">
        News
      </Link>
    </>
  );

}
```

We described a set of links in its body.

```
<Link to="/" className="links">
    Main
</Link>
```

The description of a specific link. In this case, the link leads to the root page. We specified the name of our custom CSS class for design. Its description is in *style.css* of the project.

```
.links {
    margin: 10px;
    text-decoration: none;
}
.links:hover {
    color: red;
}
```

When we click on a specific link, it activates the route by the address in the link. To display our functional component NavMenu, we use the following code:

```
export default function App() {
  return (
    <div>
      <Router>
        <div>
          <NavMenu />
          <Switch>
            <Route exact path="/" component={Main} />
            <Route path="/about"
                   component={AboutCompany} />
            <Route path="/news" component={News} />
            <Route component={NotFound} />
          </Switch>
```

```
        </div>
      </Router>
    </div>
  );
}
```

We added the creation of NavMenu in the Router block before Switch.

- Link to the project code.

Passing Route Parameters

We have not used parameters in our examples yet. When might you need parameters in a path?

For example, if we create a web store, parameters in the address bar will help us display information about a specific product. Let's consider how parameters are applied through an example.

In the app, we will display a page with information about branches, a page of a specific branch, news for a specific branch.

The appearance of the page with all branches:

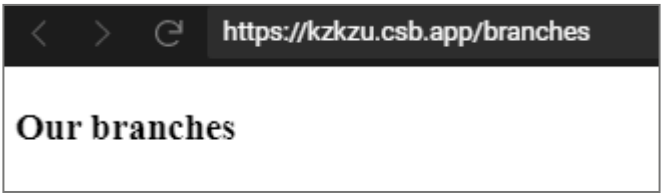


Figure 21

We did not work through the business logic of all this. This is why our page only has the caption Our branches. The page of a specific branch:

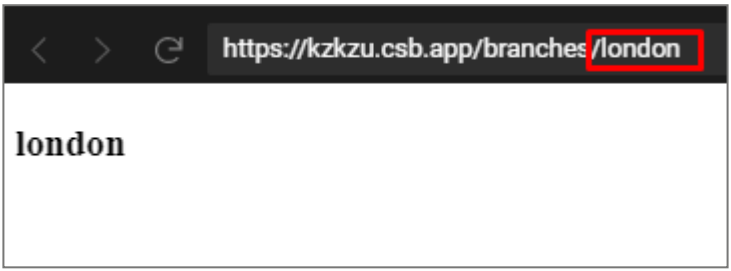


Figure 22

London in our path is a parameter that we substituted. There can be any city name instead of London.

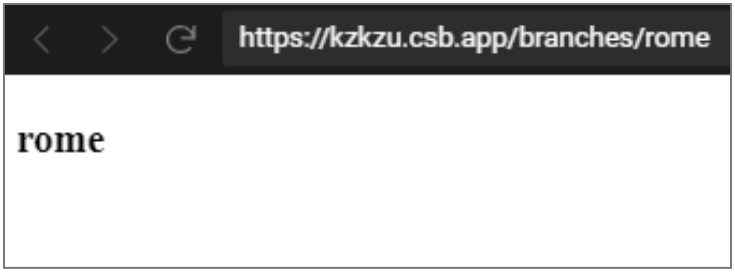


Figure 23

The page of a specific piece of news in a specific branch:

Figure 24

If we created the logic for processing this query, we would need to show news number one in the London branch. The branch name and news number are parameters of our route.

Figure 25

In this case, the branch parameter is Berlin, and the news number is 99.

Implementation in *App.js*:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
         from "react-router-dom";

function Main() {
  return <h2>Main page</h2>;
}

function BranchList() {
  return <h3>Our branches</h3>;
}

/*
  Display the branch name
*/
function Branch(props) {
  return <h3>{props.match.params.name}</h3>;
}
```

```
/*
  Check two routes
  If the route leads to branches, display a list of
  branches
  If the route leads to branches and has a parameter,
  display a specific branch
*/
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
                    component={BranchList} />
      <Route path="/branches/:name" component={Branch} />
    </Switch>
  );
}
/*
  Display a branch name and news id
*/
function News(props) {
  const branch = props.match.params.branch;
  const id = props.match.params.id;

  return (
    <div>
      <h3>Branch:{branch}</h3>
      <h3>Id: {id}</h3>
    </div>
  );
}

export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main}/>
```

```
            <Route path="/branches" component={Branches}/>
            <Route path="/news/:branch/:id"
                   component={News} />
            <Route children={() => <h2>Not Found</h2>}/>
          </Switch>
        </Router>
      </>
    );
}
```

The code has lots of constructs already familiar to you. Let's run over new aspects.

```
<Route path="/news/:branch/:id" component={News} />
```

In order to specify a parameter for the path, we use this syntax: path/:parameter_name.

The news route has two required parameters: branch and id

When we say that a parameter is required, it means that we need to pass it in order to activate the route; otherwise, the path will not be found, and the NotFound component will display.

Figure 26

We did not pass the news number, and the route was not found.

To get access to route parameters, the component body uses props.match.params.

```
function News(props) {
  const branch = props.match.params.branch;
  const id = props.match.params.id;
  return (
    <div>
      <h3>Branch:{branch}</h3>
      <h3>Id: {id}</h3>
    </div>
  );

}
```

To access a specific parameter, we should specify its name. For instance, props.match.params.branch

The code for branch routes:

```
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
                  component={BranchList} />
      <Route path="/branches/:name"
             component={Branch} />
    </Switch>
  );

}
```

If we do not specify a parameter in a path, the BranchList component loads. If the parameter is specified, then the Branch component loads.

Carefully study this example to get a better understanding.

- Link to the project code.

Optional Parameters

As you might have guessed, there are optional parameters. These are parameters that you can not pass.

Let's add a Management route to our example. It will be responsible for displaying branch management. In this route we will use optional parameters: branch name and worker's surname.

UI of our route:

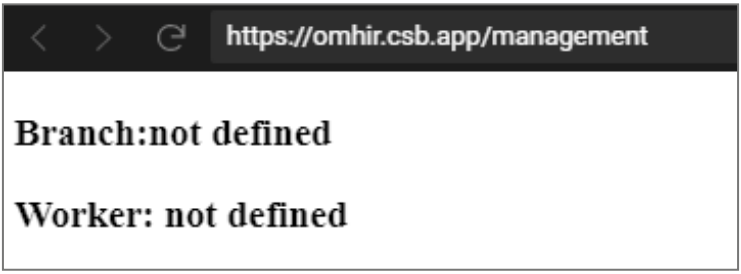


Figure 27

No parameter was specified.

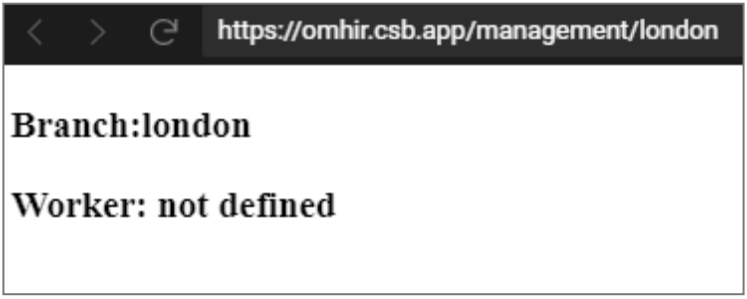


Figure 28

We specified the branch name but did not specify the worker's surname.

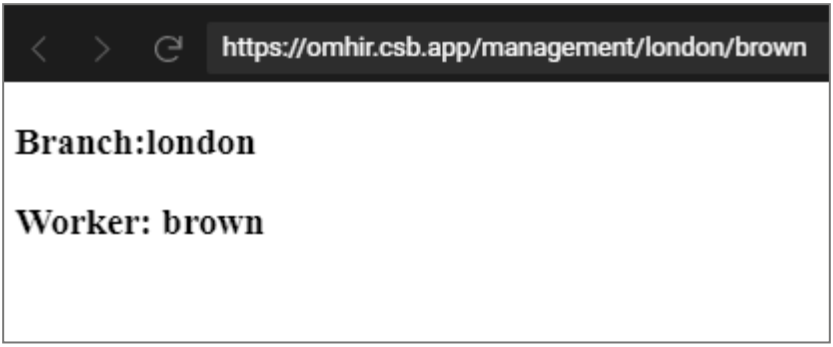


Figure 29

Both parameters were specified. We will not give the full code of *App.js*. Instead, let's analyze only some pieces:

```
export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main}/>
          <Route path="/branches" component={Branches}/>
          <Route path="/news/:branch-:id(\d+)"
                 component={News} />
          <Route path="/management/:branch?/:worker?"
                 component={Management} />
          <Route children={() => <h2>Not Found</h2>}/>
        </Switch>
      </Router>
    </>

  );
```

We create a route with optional parameters inside the Router, like any other route.

```
<Route path="/management/:branch?/:worker?"
       component={Management} />
```

We use ? after the parameter name to specify that the parameter is optional. For instance, :branch?

The code of the Management component:

```
function Management(props) {
  let branch = "not defined";
  let worker = "not defined";
  if (typeof props.match.params.branch !== "undefined")
    branch = props.match.params.branch;
  if (typeof props.match.params.worker !== "undefined")
    worker = props.match.params.worker;
  return (
    <div>
      <h3>Branch:{branch}</h3>
      <h3>Worker: {worker}</h3>
    </div>
  );
}
```

Inside we check if the parameter is defined. And we use typeof for this. If it is not defined, typeof will return undefined.

Notice some new tricks in the code:

```
<Route path="/news/:branch-:id(\d+)" component={News} />
```

When defining parameters, you do not need to specify /. We used – as a separator in the code above.

Figure 30

You can see here that when passing parameters we specified — because it was specified in the path attribute.

When describing a parameter, we can impose a restriction on its content. Regular expressions familiar to you from the JavaScript course are used for this. In the code above we specified :id(\d+). It means that id can only contain numbers. If you try to enter at least one letter, the route will not be found:

Figure 31

- Link to the project code.

Routes, Links, Arrays

Let's create one more parameter where we combine all studied concepts. The app will display a list of branches, and if we click on a branch name, we will be taken to its page. There will be an array of branch objects in our app.

The appearance of the app will be as follows:

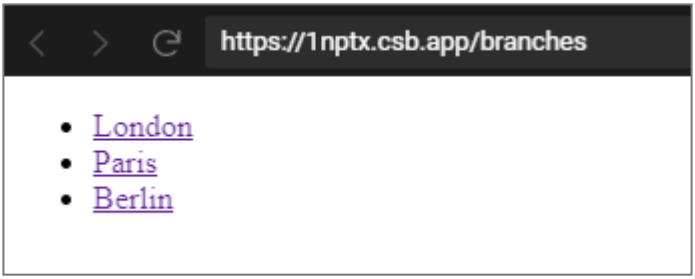


Figure 32

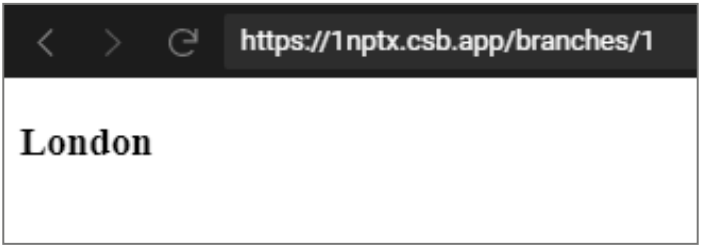


Figure 33

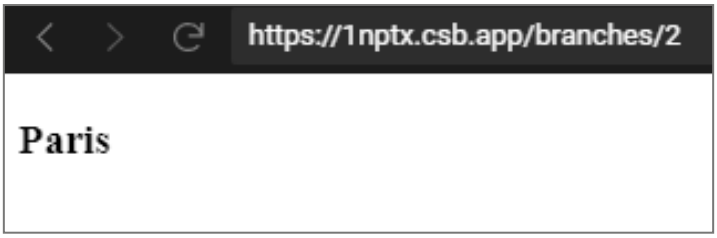


Figure 34

App.js code:

```
import React from "react";
import {BrowserRouter as Router, Route, Switch, Link}
        from "react-router-dom";
```

```
/*
  Data array for testing
*/
const branches = [
  { id: 1, name: "London" },
  { id: 2, name: "Paris" },
  { id: 3, name: "Berlin" }
];

function Main() {
  return <h2>Main page</h2>;
}

/*
  Display a list of branches. If you click on a link,
  a page of a specific branch will open
*/
function BranchList() {
  return (
    <ul>
      {branches.map(item => {
        return (
          <li key={item.id}>
            <Link to={'/branches/${item.id}'}>
                      {item.name}</Link>
          </li>
        );
      })}
    </ul>
  );
}
/*
  Display the branch name. But first, we check
  if our object array has the branch id
*/
function Branch(props) {
  let branchId;
```

```
  let branch;
  branchId = parseInt(props.match.params.id, 10);

  for (let i = 0; i < branches.length; i++) {
    if (branches[i].id === branchId) {
      branch = branches[i];
      break;
    }
  }

  if (branch !== undefined) {
    return <h3>{branch.name}</h3>;
  } else {
    return <h3>Branch is not found!</h3>;
  }
}
/*
  Check two routes. If the route leads to branches,
  display a list of branches.
  If the route leads to branches and has a parameter,
  display a specific branch
*/
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
                  component={BranchList}/>
      <Route path="/branches/:id"
             component={Branch} />
    </Switch>
  );
}
export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main}/>
```

```
          <Route path="/branches" component={Branches}/>
          <Route children={() => <h2>Not Found</h2>}/>
        </Switch>
      </Router>
    </>
  );
}
```

The array of branch objects is defined in our code:

```
const branches = [
  { id: 1, name: "London" },
  { id: 2, name: "Paris" },
  { id: 3, name: "Berlin" }
];
```

We need id for the key attribute when displaying a specific list item. We have talked about the key attribute earlier.

The code of the Branches component:

```
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
             component={BranchList} />
      <Route path="/branches/:id" component={Branch} />
    </Switch>
  );
}
```

If the path is /branches, we display a list of branches. If the path is /branches/:id, we display information about a specific branch. Let's begin with the code of BranchList.

```
function BranchList() {
  return (
    <ul>
      {branches.map(item => {
        return (
          <li key={item.id}>
            <Link to={'/branches/${item.id}'}>
                        {item.name}</Link>
          </li>
        );
      })}
    </ul>
  );
}
```

We create a list using the map method already familiar to you. A list item is a link to a specific city. We use Link to create a link.

Let's now consider the code of Branch.

```
function Branch(props) {
  let branchId;
  let branch;

  branchId = parseInt(props.match.params.id, 10);

  for (let i = 0; i < branches.length; i++) {
    if (branches[i].id === branchId) {
      branch = branches[i];
      break;
    }
  }

  if (branch !== undefined) {
    return <h3>{branch.name}</h3>;
```

```
  } else {
    return <h3>Branch is not found!</h3>;
  }
}
```

We check if there is the identifier that we got in the array of branch objects. If it is there, display information about the branch; otherwise, report an error.

Figure 35

Below is the example of passing a wrong id.

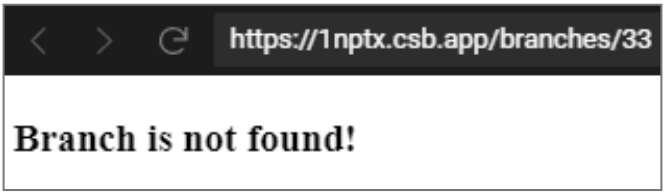


Figure 36

- Link to the project code.

Homework

1. Use routes to create an app dedicated to a famous artist. One route may lead to the artist's biography, the other to his most famous painting, the third to a collection of his paintings.
2. Supplement Task 1 with a link mechanism that allows going from the main page to routes by links.
3. Supplement Task 1 with the passing of parameters when routing.
4. Use routes to create an app dedicated to your city. One route should lead to the info about the city, the other to its most popular point of interest, the third to other places worth seeing, the fourth to photos of the city.
5. Supplement Task 4 with a link mechanism allowing you to go from the main page to routes by links.
6. Supplement Task 4 with the passing of parameters when routing.

Lesson 8

React: Advanced Techniques