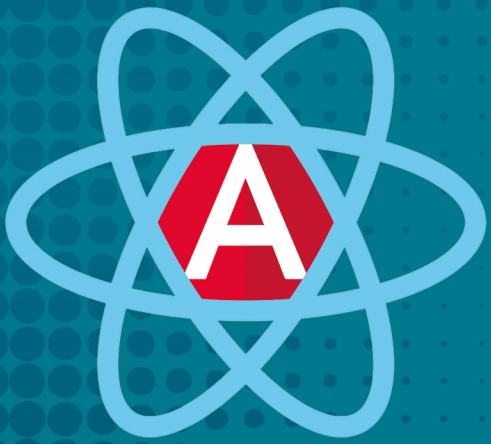
Building Web Applications Using Angular & React



STEP computer ACADEMY

Lesson 9

React: Advanced Techniques

Contents

Routing and Query String **3**
Redirect Routes **9**
Component Lifecycle **20**
Homework **31**

Routing and Query String

We continue our acquaintance with routes and are about to add a work with a query string to the mechanisms already studied. What is a query string? You have seen it many times. The most common example is the search box:

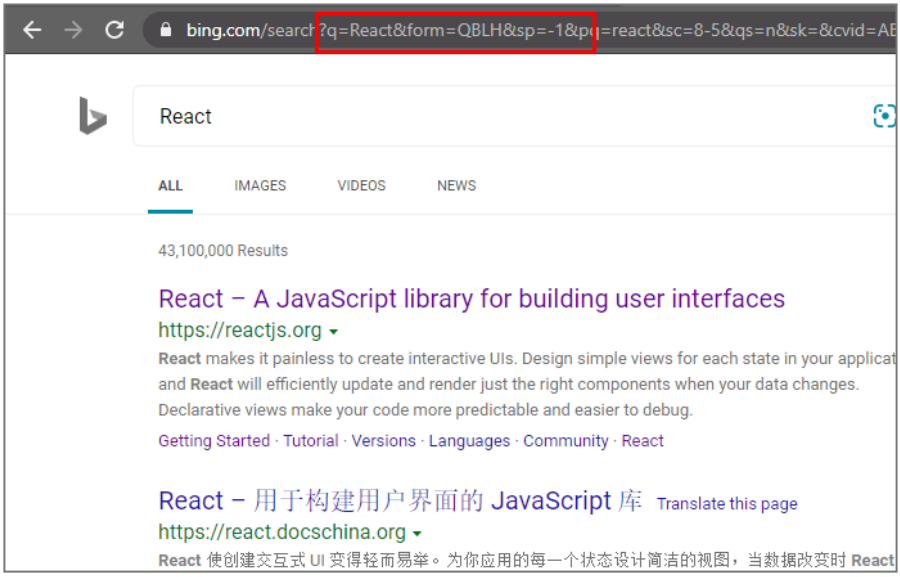


Figure 1

You can see the query string on the search results page in Bing. We are dealing with the GET method, this is why we can see the query string. The query string contains values. For example, q=React, where q is a name, and React is a value. The query string helps to pass a set of values. These values can be used for various purposes. In our case, Bing uses the query string to get the information we are looking for.

How can we use a query string in our apps? For instance, we can pass a set of values that will be used to filter information.

Let's consider an example. Our app will display data about cars.

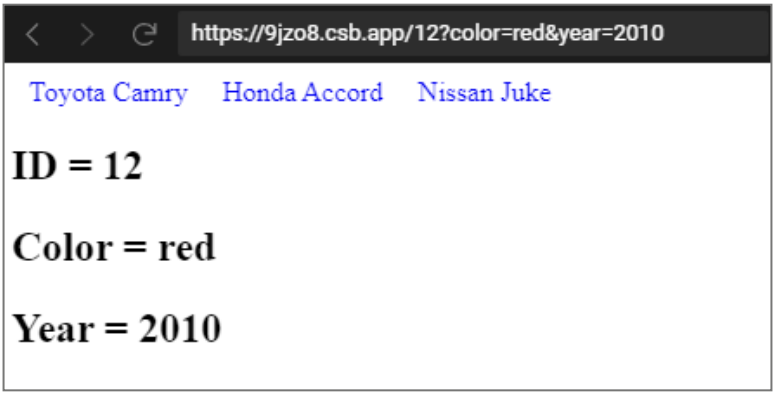


Figure 2

When clicking on a specific car, its data are displayed. The address bar shows query params.

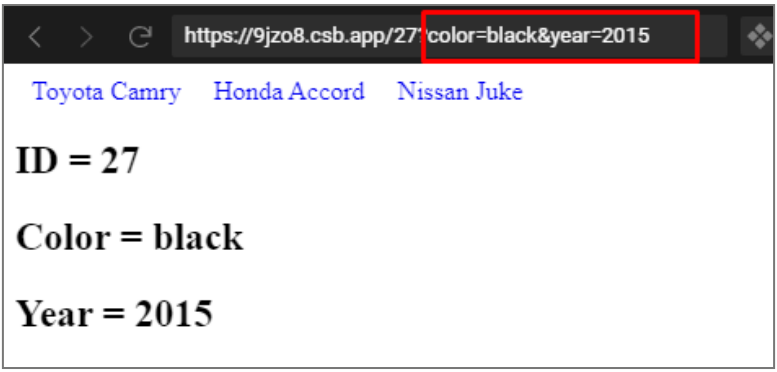


Figure 3

They come after the question mark. Query parameter names in our example are color and year.

App.js code:

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

function Main(props) {
  console.log(props.match);
  console.log(props.location);
  return (
    <div>
      <h2>ID = {props.match.params.id}</h2>
      <h2>Color = {
         new URLSearchParams(props.location.search).
                    get("color")}</h2>
      <h2>Year = {
         new URLSearchParams(props.location.search).
                    get("year")}</h2>
    </div>
  );
}

export default function App() {
  return (
    <>
      <Router>
        <nav>
           <Link to="/12?color=red&year=2010">
               Toyota Camry
           </Link>
           <Link to="/27?color=black&year=2015">
              Honda Accord
           </Link>
           <Link to="/27?color=yellow&year=2018">
              Nissan Juke
           </Link>
        </nav>
```

```
        <Switch>
           <Route path="/:id?" component={Main} />
        </Switch>
      </Router>
    </>
  );
}
```

The code of our app is relatively simple.

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

Since we use routing, make sure to add react-router-dom and its important parts.

```
<Router>
    <nav>
        <Link to="/12?color=red&year=2010">
            Toyota Camry
        </Link>
        <Link to="/27?color=black&year=2015">
            Honda Accord
        </Link>
        <Link to="/27?color=yellow&year=2018">
            Nissan Juke
        </Link>
    </nav>

    <Switch>
       <Route path="/:id?" component={Main} />
    </Switch>
</Router>
```

The navigation block contains a new path format. We form a query string using this format.

```
<Link to="/12?color=red&year=2010">Toyota Camry</Link>
```

12 is a car id, color and year are query params. They must be separated by &.

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

The route has id and ? params.
This allows our route to handle calls that contain ?.

```
function Main(props) {
   console.log(props.match);

   console.log(props.location);

   return (
      <div>
         <h2>ID = {props.match.params.id}</h2>
         <h2>Color = {
           new URLSearchParams(props.location.search).
                                get("color")}</h2>
         <h2>Year = {
           new URLSearchParams(props.location.search).
                                 get("year")}</h2>
      </div>
   );

}
```

In the component code, we display the content of match and location to the console.
Query params are inside props.location.search.

To get a parameter value, we use URLSearchParams.

```
new URLSearchParams(props.location.search).get("color")
```

The number of values and names in the query string depend only on your imagination and tasks within a specific project.

- Link to the project code.

Redirect Routes

How often have you tried to open a page and have been redirected to another one while surfing the Internet? There are many reasons for this. The information is no longer relevant on this page, or the site moved to a new address, to name a few.

A React app redirects using the Redirect from react-router-dom.

The syntax is as follows:

```
<Redirect from="where_from" to="where_to" />
```

Specify the initial address in the from attribute, and the redirection address in the to attribute. Let's create an app with redirection.

The appearance of our app:

Figure 4

When clicking on the Archive link, the app will be redirected to News.

App.js code:

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

import "./styles.css";

function Main() {
  return <h1>Main page</h1>;
}
/*
  All attempts to access Archive will be redirected
  to News
*/
function News() {
  return <h1>News page</h1>;
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news">News</Link>
            <Link to="/archive">Archive</Link>
          </nav>
        </div>
        <Switch>
          <Route exact path="/" component={Main} />
```

```
            <Route path="/news" component={News} />
            <Redirect from="/archive" to="/news" />
          </Switch>
        </Router>
      </div>
    );
}
```

We imported Redirect for use. The router code section:

```
<Router>
    <div>
        <nav>
            <Link to="/">Main</Link>
            <Link to="/news">News</Link>
            <Link to="/archive">Archive</Link>
        </nav>
    </div>
    <Switch>
        <Route exact path="/" component={Main} />
        <Route path="/news" component={News} />
        <Redirect from="/archive" to="/news" />
    </Switch>
</Router>
```

You have two main components: Main and News. We have not created a component for the Archive link because we use redirection. We inserted Redirect in the route description for it.

```
<Redirect from="/archive" to="/news" />
```

When attempting to access /archive, the user will be redirected to /news.

- Link to the project code.

A problem of data transfer may arise during redirection. It is solved with the mechanisms you already know. Let's add the passing of parameters to our example. The appearance of the app:

Figure 5

When clicking News, we pass id 56; when redirecting from Archive to News, we pass 44.

Figure 6

App.js code:

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

import "./styles.css";

function Main() {
   return <h1>Main page</h1>;
}

/*
  All attempts to access Archive will be redirected
  to News
*/
function News(props) {
   return (
      <>
         <h1>News page</h1>
         <h2>ID = {props.match.params.id}</h2>
      </>
   );
}

/*
  Redirect to News and pass the received parameter
*/
function Archive(props) {
    return <Redirect to={`/news/${props.match.
                                  params.id}`} />;
}
```

```
export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news/56">News</Link>
            <Link to="/archive/44">Archive</Link>
          </nav>
        </div>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/news/:id" component={News} />
          <Route path="/archive/:id"
                 component={Archive} />
          <Route children={() => <h1>404 page</h1>} />
        </Switch>
      </Router>
    </div>
  );
}
```

The code block for the routing differs from the previous example.

```
<Router>
    <div>
        <nav>
          <Link to="/">Main</Link>
          <Link to="/news/56">News</Link>
          <Link to="/archive/44">Archive</Link>
        </nav>
    </div>
    <Switch>
        <Route exact path="/" component={Main} />
```

```
        <Route path="/news/:id" component={News} />
        <Route path="/archive/:id"
               component={Archive} />
        <Route children={() => <h1>404 page</h1>} />
    </Switch>
</Router>
```

When describing the links News and Archive, we specify a value for id (56 for News, 44 for Archive). We do not use Redirect in the router block anymore.

```
<Switch>
    <Route exact path="/" component={Main} />
    <Route path="/news/:id" component={News} />
    <Route path="/archive/:id"
           component={Archive} />
    <Route children={() => <h1>404 page</h1>} />
</Switch>
```

Instead, we specify the Archive component. Redirection will take place in this component. This is the code of the Archive component:

```
function Archive(props) {
   return <Redirect to={'/news/${props.match.params.
                                  id}'} />;
}
```

When redirecting inside the path, we indicate the value of the received parameter.

- Link to the project code.

Let's add verification of the received identifier to our redirection mechanism.

If the parameter is 44, we will redirect the user to News. If we get a different value, the user will be redirected to the main page.

Figure 7

ID 44 received.

Figure 8

We specified the path with id 99 in the address bar, so after Enter is pressed, the user will be redirected to the main page.

Figure 9

App.js code:

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

import "./styles.css";

function Main() {
   return <h1>Main page</h1>;
}
/*
  All attempts to access Archive will be redirected
  to News
*/
function News(props) {
   return (
```

```
      <>
        <h1>News page</h1>
        <h2>ID = {props.match.params.id}</h2>
      </>
  );
}

/*
  Redirect to News and pass the received parameter
*/
function Archive(props) {
  if (Number.parseInt(props.match.params.id, 10) ===
        44)
    return <Redirect to={`/news/${props.match.
                                  params.id}`} />;
    else return <Redirect to="/" />;
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news/56">News</Link>
            <Link to="/archive/44">Archive</Link>
          </nav>
        </div>

        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/news/:id" component={News} />
          <Route path="/archive/:id"
                 component={Archive} />
          <Route children={() => <h1>404 page</h1>} />
        </Switch>
```

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

The code of the Archive component in our example is arranged in a new way:

```
function Archive(props) {
  if (Number.parseInt(props.match.params.id, 10) === 44)
    return <Redirect to={'/news/${props.match.
                          params.id}'} />;
  else return <Redirect to="/" />;
}
```

Check the identifier. If it is 44, the user is redirected to News; otherwise, to the main page.

- Link to the project code.

We hope that you fully understand the router mechanism now. True understanding, however, comes with practice only. Experiment with the new tool in your projects.

Component Lifecycle

We already know a lot about components. We can create them, pass parameters to them, work with a state, and much more. Now we will talk about another important aspect of components — lifecycle. The concept of a life cycle is probably familiar to you from the biology course you had at school.

A **component life cycle** is a set of stages a component goes through during its life. We can create functions for processing a certain stage. Lifecycle event handling is different for class and functional components. Let's begin with class components.

The list of the most popular functions for lifecycle event handling in class components is as follows:

- constructor(props) — a constructor for a class component. It is called before the component is added inside DOM. It can initialize a state, assign event handlers, etc.
- render() — is used to render a component. It is also called when changing props and state
- componentDidMount() — is called after the component is attached. DOM already exists. Inside the function, we can interact with DOM, timers, make http requests, change a state, begin to work with frameworks, etc.
- componentDidUpdate() — is called after the DOM update. It is not called during the initializtion.
- componentWillUnmount() — is called before the component is deleted from DOM. Here you can free up the received and allocated resources.

These are not all functions available for state handling. An introduction will, however, be enough for us to understand

the concept of lifecycle. You can read about other functions in the official React documentation.

Let's consider an example where we will see the sequence of launching a particular function. The appearance of the app:

Figure 10

We increase the counter value when clicking on a button.

Figure 11

Figure 12

In the code, we will display information messages in the developer's console:

App.js code:

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

export default class App extends React.Component {
  constructor(props) {
    super(props);
    console.log("Constructor");
    this.state = {
      counter: 0
    };
  }

  componentDidMount() {
    console.log("Component did mount");
  }

  componentDidUpdate() {
    console.log("Component did update");
  }

  componentWillUnmount() {
    console.log("Component will unmount");
  }

  render() {
    console.log("render");
    const clickHandler = () => {
      this.setState({ counter: this.state.counter + 1 });
    };

    return (
      <div className="App">
        <h1>Lifecycle sample</h1>
```

```
          <button onClick={clickHandler}>
                          {this.state.counter}
          </button>
        </div>
      );
    }
}
```

To create and update a counter, we use a state. The most interesting thing in this example is the sequence of calling lifecycle functions.

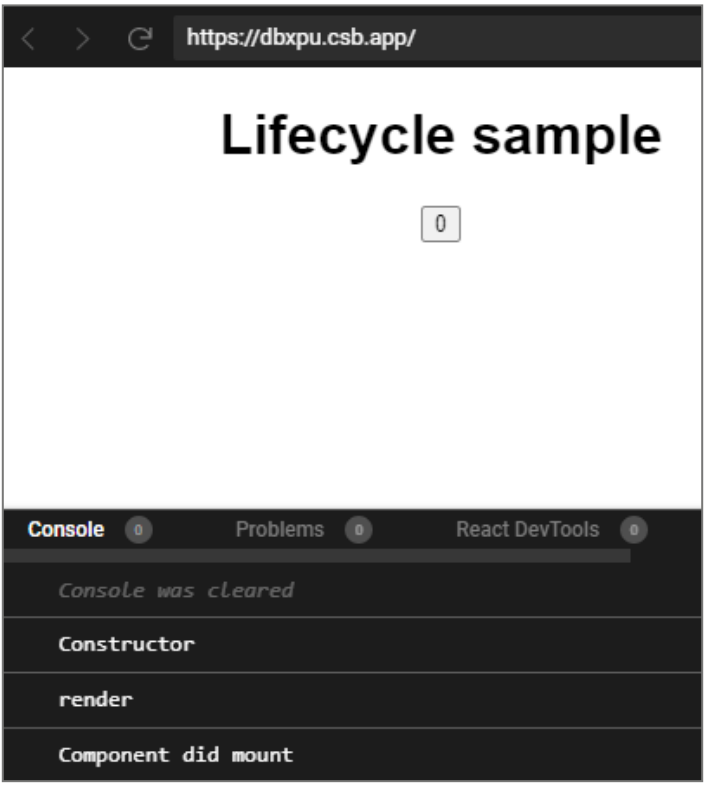


Figure 13

When the application starts, three functions are called: constructor, render, componentDidMount. Click on the button and see the result in the console:

Figure 14

Two new items were added to the received call list: render, componentDidUpdate. The render function was called because we updated the state. The function componentDidUpdate was called because DOM was updated. If we click the button again, the same two methods will be called.

- Link to the project code.

Let's create a new project to work with the lifecycle. We will use timer options in it. Our code displays the current time. It is updated every second.

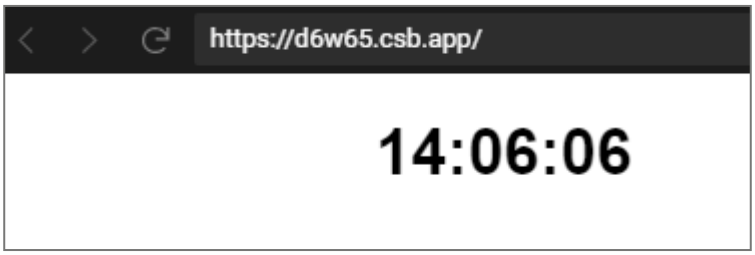


Figure 15

Let's ask ourselves, how can we implement it?

Time should be stored inside the state so that its change leads to redrawing DOM. What functions do we need?

1. A constructor to initialize the state. If you want to, you can initialize the state using alternative syntax without a constructor. We showed you this method through one of the examples in the previous lessons.
2. The render function for redrawing a component. It will be called automatically when the state changes.
3. The componentDidMount function. It creates a timer.
4. The componentWillUnmount function. We will free up all the resources allocated for the timer.

App.js code:

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

export default class App extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {
      currDate: new Date()
    };
  }

  /*
    Set the timer. It will be triggered every second
    The timerAction will be called every second
  */
  componentDidMount() {
    this.handlerOfTimer = setInterval(() =>
         this.timerAction(), 1000);
  }

  /*
    Timer handler. Change the state
  */
  timerAction() {
    this.setState({ currDate: new Date() });
  }

  componentWillUnmount() {
    clearInterval(this.handlerOfTimer);
  }

  render() {
    return (
      <div className="App">
        <h1>
            {this.state.currDate.toLocaleTimeString()}
        </h1>
      </div>
    );
  }
}
```

We initialize the state inside the constructor:

```
constructor(props) {
    super(props);
    this.state = {
       currDate: new Date()
    };
}
```

We can set up the timer when DOM is created and ready to work. This is why we have to use componentDidMount:

```
componentDidMount() {
    this.handlerOfTimer = setInterval(() =>
         this.timerAction(), 1000);
  }
```

We set a timer to every second in the code. The timerAction will be called when ticking. Besides, we saved the timer descriptor because we need to free up timer resources when our app stops.

```
timerAction() {
    this.setState({ currDate: new Date() });
}
```

The timer handler changes the state. After changing the state React automatically calls the render function.

```
componentWillUnmount() {
    clearInterval(this.handlerOfTimer);
}
```

We have to free up the timer resources. For this we use componentWillUnmount.

- Link to the project code.

Let's move on to functional components. The useEffect hook is used for embedding in a component lifecycle. It allows responding to componentDidMount and componentDidUpdate for the functional component.

The syntax is as follows:

```
useEffect(handler)
```

Let's consider the use of hook through an example. Create an app that displays the counter value in a document and browser window title.

The appearance of the app:

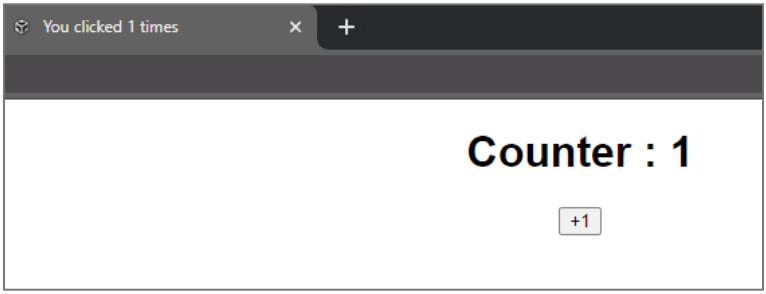


Figure 5

App.js code:

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

export default function App() {
  const [counter, setCounter] = useState(0);
```

```
  useEffect(() => {
    document.title = `You clicked ${counter} times`;
  });

  return (
    <div className="App">
      <h1>Counter : {counter}</h1>
      <button onClick={() => setCounter(counter + 1)}>
        +1
      </button>
    </div>
  );
}
```

The main code for us is calling the useEffect.

```
useEffect(() => {
    document.title = `You clicked ${counter} times`;

});
```

We created a handler for componentDidMount and componentDidUpdate inside the useEffect call. Our code will be called whenever these functions are triggered. Inside the code, we display the current value of the state variable in the window title. And, as always, the code of the functional component is much simpler than that for the class component.

- Link to the project code.

But there is a problem in our code. Update of any state variable will call the handler code specified in useEffect. In order to avoid this, we should change the call of useEffect.

```
useEffect(() => {
    document.title = `You clicked ${counter} times`;
  },[counter]);
```

The second parameter of useEffect has an array of variable states specified by us, and changes made to it should call the code in the first parameter. In our case, the array contains only one counter variable.

Homework

1. Create an app that displays the current time. Be sure to use functional components.
2. Create an app Timer. The user enters the number of seconds and clicks the Start button. The countdown begins immediately after the button is clicked. The user can also use Stop and Pause buttons.
3. Create a text transliteration app. The user enters text in the text field. Another text field displays transliterated text.

Lesson 9

React: Advanced Techniques