REACT Bausteine

written by Hong Le Nguyen 08.2024

React Component

component-tree

In React the user interface (UI) is mapped in a tree structure.
Each node is *a component that represents a UI element* e.g. ToDoForm, ToDoList.
The root note e.g. App represents actually UI application.
We have the concept of parent and child components, where each parent component may itself be a child of another component.

rendering the UI App on a web page

Note that the root component App is the React UI Application
The App component is referred to as <App/> element

```
//  index.html file
<html lang="en">
  ...
  <div id="root"></div>

//  index.js file
import ReactDOM from 'react-dom/client';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App/>);
```

component definition

React component is a JavaScript function that accepts props object as an argument, returns a JSX code.

```
function Hello(props) {
  let greeting = `Hello ${props.name}`
  return (
    <h1>{greeting}</h1>  /*JSX code*/
  )
}
```

or as Arrow function

```
const Hello = (props) => { ...;  return ( /* JSX code */ )}
```

You can use object destructuring to pick the properties required by a React component.
The property name needed by the Hello component

```
function Hello({ name }) {
  let greeting = `Hello ${name}`
  ...
}

const Hello = ({ name }) => { ... return ( /* JSX code */ )}
```

using component in parent component

You can embed and display Hello component in a other component e.g. App
The Hello component is referred to as <Hello /> element in JSX code of App component.
Argument "Hong" is passed to the Hello component via "name" attribute of <Hello/> element.

```
function App() {
 return (
  <div>
    <Hello name="Hong" />
  </div>
 );
}
```

props

Props can be null or an object. They are passed into component via attributes of element in JSX code.
We can also send *functions as props*, so *a child component can call a function in the parent component*.

An example, ToDoForm component is referred to as < ToDoForm /> element in JSX code of parent component App, we pass argument dispatch (a function) via attribute "dispatch" of < ToDoForm /> element

```
function App() {
   const [state, dispatch] = useReducer(todosReducer, todosInitialState)

   return (
      <ToDoForm dispatch={dispatch} />
      …
   );
}
```

A special prop is called ***children***. That contains the value of anything that is passed between the opening and closing tags of the component.

JSX code

We call *JSX code* everything inside return(/* JSX code */).
JSX (JavaScript XML) allows us to write HTML elements in JavaScript and place them in the DOM as UI element.
Under the hood, React will process the JSX and it will transform it into JavaScript that the browser will be able to interpret.

Embedding JavaScript in JSX code
Inside the curly brackets { } we can add any JavaScript statement

```
 let greeting = `Hello ${props.name}`
 return (
  <h1>{greeting}</h1>
 )
```

Using JSX to compose UI
In particular, in a React component, you can import other React components, and you can embed and display them.

Conditional rendering

using conditional statements to decide what content should be rendered.
Reasons to use conditional rendering:
create dynamic user interfaces that adapt to changes in data and user interactions
avoid rendering unnecessary components

using if statement and render variables ---

```
function Greeting({isLoggedIn}) {
  let display = <h1>Please sign up.</h1>;
  if (isLoggedIn){
     display = <h1>Welcome back!</h1>;
  }
 return (
  <div>
    {display}
  </div>
 );
}
```

using the ternary operator ---

When you have a simple “if … else” situation with a single condition, a ternary operator can be a good fit.

```
function Greeting({isLoggedIn}) {
 return (
  <div>
    {isLoggedIn ? (
      <h1>Welcome back!</h1>
    ) : (
      <h1>Please sign up.</h1>
    )}
  </div>
 );
}
```

using Early return and Return null ---

to hide/display the component in parent-component

```
function Shop() {  // parent
   const [books, setBooks] = useState([]);

   return (
      <div className="row">
           <BookList books={books} />
      </div>
   )
}
```

```
const BookList = ({books}) => {
  …
  if (!books.length) {  // Early return null
    return null;
  }

  return (
     <div className="list-group">
       {books.map((book) => ...}
     </div>
  )
}
```

Alternativ ---

```
function Shop() {  // parent
  const [books, setBooks] = useState([]);
  …
  let book_list = <div></div>;

  if (books.length > 0) {
    book_list = <BookList books={books} />;
  }

  return (
    <div className="row">
          …
          {book_list}
    </div>
  )
}

const BookList = ({books}) => {
  …
  return (
    <div className="list-group">
        {books.map((book) => ...}
    </div>
  )
}
```

event

Embedding Events in JSX code and adding event-handlers in component

```
function Hello() {
  const eventHandler = () => { console.log("Hello") }
  return (
    <button onClick={eventHandler}>Say Hello!</button>
  );
```

React events are written in camelCase syntax: onClick instead of onclick
React event handlers are written inside curly braces: onClick={eventHandler} instead of onclick="
eventHandler()"

To pass an argument to an event handler, use an arrow function.

```
const eventHandler = (a) => { console.log(a) }
return (
  <button onClick={ () => eventHandler("Hi!") }>Say Hi!</button>
);
```

Access event object
Event handlers have access to the React event that triggered the function.

```
const eventHandler = (event) => {
  console.log(`Hello on Event ${event.type}`)
}
return (
  <button onClick={eventHandler}>Hello on Event click</button>
);
```

or

```
return (
  <button onClick={ (event) => { console.log(`Hello on Event ${event.type}`) } }>
    Hello on Event click
  </button>
);
```

with another parameter

```
const eventHandler = (a, event) => {
  console.log(`${a} on Event ${event.type}`);
}
return (
  <button onClick={ (event) => eventHandler("Hi!", event) }>Say Hi on click!</button>
);
```

events handling in form

```
const SearchForm = ({ setBooks }) => {

  const [filterText, setFilterText] = useState('');

  const submitHandle = (event) => {
    event.preventDefault();
    ...
  }

  return (
    <form onSubmit={submitHandle} >
        <input
          type="text"
          value={filterText}
          onChange={ event => setFilterText(event.target.value) }
        />
        <button type="submit" >Search</button>
    </form>
  )
};
```

React hooks

are functions to "hook"/add React features such as state, lifecycle management into component.
There are 3 rules for hooks:

- Hooks can only be called *inside React function components*.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

component state

React component can have its own state. The state is the set of data that can be managed using React hooks useState or useReducer. This is important to know:

We can't alter the value of a state variable directly. We must call its modifier function. This is the way we can tell React that the component state has changed. Otherwise, the React component will not update its UI to reflect the changes of the data.

The useState hook
is ideal for components with simple state management.

```
function ToDoForm () {
  const [userInput, setUserInput] = useState('');
```

The state variable is userInput, its modifier function setUserInput.
The Function call looks like: setUserInput("hans")

The useReducer hook
is best for components involving complex state management.

```
const initialState = { ... }
function todosReducer(state, action){
  switch(action.type){
    case 'add':
      ...
      return (new-)state

function ToDoApp() {
  const [state, dispatch] = useReducer(todosReducer, initialState)
```

The state variable is state, its modifier function dispatch. The dispatch function triggers the reducer function to carry out a particular task. The function call looks like:

```
dispatch({type: 'add', payload: data})
```

The object {type: 'add', payload: data} is called “action” object.

component's lifecycle and component rendering

Mounting Phase (initial-render)

a component is being created and inserted into the DOM

Updating Phase (re-render)
when the component needs to be updated in the DOM. Usually, this happens as a result of a user interacting with the app or some external data coming through via an asynchronous request or some subscription model.

Component **re-rendering** happens when
component state changes
a property passed to it changes,
the parent-component re-renders,
the context changes (*)

(*) When the value in Context Provider changes, all components that use this Context will re-render, even if they don't use the changed portion of the data directly.

Unmounting Phase
when the component is being removed from the DOM (Document Object Model).

useEffect hook

we need to do something ***after*** our component renders, how ?

useEffect hook is used to handle side effects in functional components, such as fetching data, updating the DOM, and setting up subscriptions or timers.
In contrast to lifecycle methods, effects don't block the UI because they run asynchronously.

How does it work?

You call useEffect with *a callback function* that contains the side effect logic.
By default, this function runs after every render of the component.

You can optionally provide *a dependency array* as the second argument.
The effect will only run again if any of the values in the dependency array change.

```
import { useEffect } from 'react';

function MyComponent() {
  ...
  useEffect( ()->{ // callback function
    //  side effect logic code
  },  [props, state] )

  return (
    <div>
        ...
    </div>
  );

}
```

To run useEffect only once on the first render pass any empty array in the dependecy

```
useEffect(()->{
  …
}, [] )
```

When to use useEffect() ?
Use the hook when you want to do some side effects (data fetching, subscription, DOM manipulation, etc.) on render of a component or when certain values have changed from the last render of the component.

when not to use useEffect() ?
You don't need the useEffect() hook when you, for example, have a function to fetch some data on some event (onClick, onSubmit, etc.), as these won't be run automatically on component render (they're run only on user events).

An example
We use the useState hook to store the data that we get from the server.
We use the useEffect hook to make the GET request when the component mounts.
The empty array [] passed as the second argument to useEffect ensures that the effect only runs once, when the component mounts.

```
function Posts() {
  const [data, setData] = useState([]);

  const getData = async () => {
    try {
      const res = await axios.get('https://jsonplaceholder.typicode.com/posts');
      setData(res.data) ;
    } catch (error) {
      console.log(error);
    }
  }

  useEffect(() => {
    getData();
  }, []);

  return (
    <div>
      {data.map(item => (
        <p key={item.id}>{item.title}</p>
      ))}
    </div>
  );
}
```

How to communicate between two React components

React applications are typically composed of numerous components, each responsible for a specific piece of the user interface. To build dynamic and interactive applications, these components often need to communicate with each other.

In React, the communication between components comes under three categories.
Parent to child component communication.
Child to parent component communication.
Communication between to components that do not have a parent child relationship between them.

Prop drilling

Parent can pass data to child component using props.
A child component is inside parent. Hence parent can easily send data to the child component through props.

```
function ChildComponent({title}){ … }

function App() {
   return (
      <div>
         <ChildComponent title="This comes from parent" />
      </div>
    );
}
```

Lifting state up

Child passing data to parent through custom events

Child does not know which parent it is embedded into. The only way child can notify its parent is if parent gives child an "event handler" method, which child calls whenever it wants to inform its parent about something.

```
function ChildComponent({setChildValue}){
   return (
      <div>
         <p>I am child</p>
         <input type='text' onChange={ e => setChildValue(e.target.value) } />
      </div>
   )
}

function App() {
  const [childValue, setChildValue] = useState('');

  return (
     <div className='thickborder'>
        <p> I am parent</p>
        <p> Value sent by child: {childValue} </p>
       <ChildComponent setChildValue={setChildValue}/>
     </div>
  );
}
```

Input data is sent to parent via call function set Child Value passed from parent to child component to handle event onChange.

Sibling communication

What if a component wants to pass data to another component which is neither its parent not child ?
In React this can be done only if they share a common parent at some level. Since there is one root component in react app, it is always possible to find a common parent for any two components. They can communicate with each other is via the common parent.

Sibling communication is merely the combination of methods 1 and 2 we discussed earlier.
The child1 that wants to pass data to another sibling, sends it to its parent first using callback mechanism
The parent receives the data from one child1 and then passes it to the child2 using props of that child.

```
const ChildOne = ({ sendToParent }) => {
 return (
  <div>
   Child 1<br/>
   <input type="text" onChange={(e) => sendToParent(e.target.value)} />
  </div>
 );
};

const ChildTwo = ({ titleFromParent }) => {
 return (
  <div>
   Child 2<br/>
   Data from Child 1: <b>{titleFromParent}</b>
  </div>
 );
};

function App() {
 const [myValue, setMyValue] = useState("");

 return (
  <div className="App">
   <h1> I am parent </h1>
   <ChildOne sendToParent={setMyValue} /><br/>
   <ChildTwo titleFromParent={myValue} />
  </div>
 );
}
```

Passing Data Deeply with Context

problem
Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information.

solution
Context lets the parent component (Provider) make some information available to any component (Consumer) in the tree below it —no matter how deep— without passing it explicitly through props. You can think of *context as an object that can be shared across different components no matter how they are related*.

An Example
The data shared between components ToDoForm and ToDoList is stored in the common parent component App.

```
function App() {
  const [state, dispatch] = useReducer(todosReducer, initialState)
```

We store state accessible for child components (e.g. ToDoForm and ToDoList) in the React Context using a context provider.

```
import { createContext, useReducer, useContext, useState } from 'react';
const TodoContext = createContext();

function App() {
  const [state, dispatch] = useReducer(todosReducer, initialState)

  return (
    <div className="App">
      <TodoContext.Provider value={{state, dispatch}}>
        <ToDoForm />
        <ToDoList />
      </TodoContext.Provider>
    </div>
  );
}
```

we can define the Provider as a separate component

```
function TodoProvider({children}) {  // container component
 const [state, dispatch] = useReducer(todosReducer, initialState)
 return (
   <TodoContext.Provider value={{state, dispatch}}>
    {children}
   </TodoContext.Provider>
 );
}
```

then we wrap child components under the provider

```
function App() {
  return (
    <div className="App">
      <TodoProvider>
        <ToDoForm />
        <ToDoList />
      </TodoProvider>
    </div>
  );
}
```

In (Consumer) components we can get the context

```
const ToDoList = () => {
  const { state, dispatch } = useContext(TodoContext);

const ToDoForm = () => {
  const { dispatch } = useContext(TodoContext);
```

The context can have state and its modifier function and allow children to modify the state.
Any changes to context then automatically become available to whoever is using it.

Client side routing using React Router

Client side routing

By changing the URL via a link click, the app can render a new UI *without* making another request for a different document from the server.

How React Router works ?

React Router listens for a change to the browser's location (when a user clicks a Link to a specific URL), then looks at the routes defined in the Routes component - as mapping rules between URLs and React components - and renders the corresponding component.

Installing React Router

npm i react-router-dom for the web or
npm i react-router-native for the React Native

Three things you need to do in order to use React Router

Setup router, Define routes, Handle navigation

Setting up the router

The router you need (BrowserRouter for the web) is imported in the index.js file, wraps root component App. The router works just like a context in React and provides all the necessary information to your application so you can do routing and use all the custom hooks from React Router.

```
// index.js
import ReactDOM from 'react-dom/client';
import { BrowserRouter as Router } from 'react-router-dom';
import App from './App.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Router>
    <App />
  </Router>
);
```

Defining Routes

The next step is to define how to map the app's location (URL) to certain React components using Routes and Route. This is generally done at the top level of your application, such as in the App component, but can be done anywhere you want.

Routes

Inside <Routes>, we'll use a <Route> for each component we want to render.

Route

<Route> component has path and element properties. When the browser's location (URL) matches the path, the router will render the element.

For example the route

```
<Route path="/books" element={<BookList />} />
```

When the location is "/books", the router will render the BookList component.

```
// App.js
import { Route, Routes } from "react-router-dom"
export function App() {
 return (
  <div>
    <Routes>
     <Route path="/" element={<Home />} />
     <Route path="/books" element={<BookList />} />
     <Route path="*" element={<PageNotFound />} />
    </Routes>
 )
}
```

Handling Navigation

The final step is the ability to navigate between components. This is the purpose of the Link component.

Link
The <Link> helps the user navigate from one component to another in an Single-page application *without* reloading the entire web page.
To tell Link what path to take the user to when clicked, you pass it a "to" prop.

```
// App.js
export function App() {
 return (
   <div>
     <ul>
        <li> <Link to="/">Home</Link> </li>
        <li> <Link to="/books">Books</Link> </li>
     </ul>

     <Routes>
```

Advanced Route Definitions ---

Dynamic Routing

Assume that we want to render out Book component for individual books in application. We could hardcode each of those routes, but if we have hundreds of books then it is impossible to hardcode all these routes. Instead we need a dynamic route.

```
// App.js
export function App() {
 return (
   <div>
     <Routes>
        ...
        <Route path="/books" element={<BookList />} />
        <Route path="/books/:id" element={<Book />} />
     </Routes>
```

The final route in the above example is a dynamic route that has *a dynamic parameter of :id*. In our case our dynamic route will match any URL that starts with '/books' and ends with some value. For example, /books/1, /books/bookName will all match our dynamic route.

Pretty much always when you have a dynamic route like this you want to access *the dynamic value* in your Book component which is where the *useParams hook* comes in.

```
// Book.js
import { useParams } from "react-router-dom"

export function Book() {
  const { id } = useParams()
  return <h1>Book {id}</h1>
}
```

Nested Routes

You might have a BookList component, and within that BookList, you have different sections with their own routes.
*To create this **nested structure**, you define child routes within the parent route component*.

```
<Routes>
  ...
  <Route path="/books" element={<BookList />} >
    <Route index element={<p>Please select a book or add a new book.</p>} />
    <Route path=":id" element={<Book />} />
    <Route path="new" element={<NewBook />} />
  </Route>
</Routes>
```

Note
The path of a nested route is automatically composed by concatenating the paths of its ancestors with its own path. For example, URL for NewBook will be '/books' + 'new' == '/books/new'.
Using the "*index*" prop instead path prop we can specify a "default" component to load.

<Outlet>
Components defined in child routes should be rendered within a specific area of the parent route's component. This is where *Outlet component* comes into play.

In our example the <Outlet> element takes place for Book or NewBook component in BookList component.

```
// BookList.js
const BookList = () => {
  …
  return (
    <div id="books">
      <h3>Books</h3>
      …

      <Outlet />
    </div>
```

React Router 6 has a native prop of Outlet called context, which behind the scenes is a React context provider. The context accepts an array of states.

```
<Outlet context={[bookData]} />
```

To accept the context in child component, the child must use React Router's provided useOutletContext hook.

```
// Book.js
import { useOutletContext } from 'react-router-dom';
…
const Book = () => {
 const [bookData] = useOutletContext()
```

Links in nested Components

The URL in *a Link (to a child component) defined in the parent component* is automatically expanded with the parent's URL as a prefix. For example,

URL in Link to BookList component is '/books', defined in App

```
// App.js
export default function App() {
 return (
  <div>
    <Link to='/books'>Books</Link>
```

URL in Link to NewBook component is 'new', defined in BookList component (parent of NewBook component)

```
// BookList.js
const BookList = () => {
  return (
   <div>
     <Link to='new'>Add new book</Link>
```

The result URL in Link to NewBook component at run time is '***/books/new***'

useNavigate Hook ---

used for programmatic navigation /redirection after Actions or in Event Handlers

```
// SearchForm.js
import { useNavigate } from 'react-router-dom';
const searchUrl = 'http://localhost:8000/api/books/search?title=';

const SearchForm = ({ setBooks }) => {

   const [filterText, setFilterText] = useState('');
   const navigate = useNavigate();
   …
   const submitHandle = async (e) => {
      …
      try {
         const res = await axios.get(`${searchUrl}${filterText}`)
         setBooks(res.data.data);
         navigate('/books')
      } catch (error) {
         throw(error);
      }
    }
```

```
    return (
        <form onSubmit={submitHandle} className="form-inline">
```

Add a Layout component to Application ---

```
// App.js
…
return (
  <div>
  …
  <Routes>
     <Route path='/' element={<Layout />}>
          <Route index element={<Home/>} />
          <Route path='books' element={<BookList/>} >
             <Route index element={<p>Please select a book or add new book.</p>} />
             <Route path=':id' element={<Book />} />
             <Route path="new" element={<NewBook />} />
           </Route>
      </Route>
   </Routes>

// Layout.js
const Layout = () => {
 …
 return (
  <div id="layout">
    <h2>Layout</h2>
    <div>
        <Link to={'/'}>Home</Link> | <Link to={'books'}>Books</Link>
    </div>

    <Outlet />
  </div>
 )
}
```