

SUPSI

Master of Science in Engineering

Full-Stack Web Development with React Specialization (Hong Kong University)

Student

Olmo Barberis

Module

HK_Seminar

Data

September 9, 2021

STUDENTSUPSI

The purpose of this document is to provide an overview of all the topics covered during the *Full-Stack Web Development with React Specialization on Coursera*¹. The original course is composed of four parts, but the *HK_Seminar* will cover only two parts: an introduction to React, a front-end framework for developing single-page web applications, and server-side development with NodeJS, Express, and MongoDB.

Front-End Web Development with React

This course introduces the React framework, its components, and essential elements like routing, forms, and animations. Shared state management, with the corresponding tools like Redux, is another critical topic covered alongside fetching data.

Single Page Application

A traditional website is a collection of pages and resources fetched to the server for each page that is navigated. Even if many pages have parts in common, like footer or header, they are requested entirely every time a new page is navigated. In a Single Page Application (SPA), the entire website is downloaded at first, then only the data that changes are re-downloaded from the server. A SPA enables delivering a user experience closer to a desktop application, does not need to be entirely reloaded, and allows the browser to re-render only the changed parts. One downside of a SPA is that Search Engine Optimization (SEO) is more difficult to achieve, and the initial page load can be slower than a traditional website.

React Overview

React is a JavaScript library for building component-based user interfaces using a declarative approach. React focuses only on user interface and is designed for speed, simplicity, and scalability.

A React component is a JavaScript class or function that is imported from the React Module. It is then rendered using the React's rendering function `ReactDOM.render(...)`.

Components

A component is an independent and reusable set of React elements that should appear on the screen. Every component can accept any input and is composed of React tags that always start with a capital letter and native tags, which start with lowercase letters and are treated as DOM tags. Every component has a local state which can hold multiple information that can be passed to children components using `props`. The state is an immutable object that can be updated using the `setState(...)` directive. This function accepts the property

¹<https://www.coursera.org/specializations/full-stack-react>

to update and merge it with the actual state. Only class components can have a local state. The state must never be manipulated directly.

Handling event is possible in a similar way as on DOM elements:



```
function onDishSelect() { ... };  
<Card onClick={() => this.onDishSelect(dish)}></Card>  
<Card onClick={this.onDishSelect}></Card>
```

In the above figure is possible to see how map a function to a click event on the React element `Card`.

Lifecycle

React provides life cycle hooks/methods that can be invoked to perform certain operations. A component is created and then mounted in the application, and when it's not required anymore is unmounted. There are three stages of the lifecycle: mounting, updating, and unmounting. Each stage provide, when component is declared as class, several methods like a constructor (mounting), `componentDidMount()` (called after mounting is finished), `render()` (call when rendering UI) and many others.

Document Object Model

In the browser, there is an object called `Browser DOM` (Document Object Model) and is the representation of the structure and data of a webpage. React uses a lightweight version of DOM called `Virtual DOM`, which uses an in-memory tree data structure of plain JavaScript objects, is extremely fast to manipulate compared to browser DOM and is fully recreated on every `setState`.

A diffing algorithm detects which nodes are changed and updates only the minimum number of components in the sub-tree that is updated.

React 16.8 introduced React Hooks to use lifecycle hooks also in a functional component. This topic is not covered in the course but will be discussed in the end of this chapter.

Functional vs Class

Until the release of `React 16.8` in 2018, there were two ways of declaring a component: class component or functional component. Functional components are s JavaScript function that returns a React element, can receive props but cannot provide local state or lifecycle hooks.



```
const Menu = ({ dishes, onClick }) => {
  return (
    <div className="container">
      <div className="row">{menu}</div>
    </div>
  )
}

// or

function Menu({ dishes, onClick }) {
  return (
    <div className="container">
      <div className="row">{menu}</div>
    </div>
  )
}
```

In the above figure is possible to see how a functional component is defined.

React Router

React Router gives the ability to navigate between views using links. It's a module that needs to be additionally installed into the React application called `react-router-dom`. It is a collection of navigational components that enable navigation among views and support browser-based bookmarkable URLs to navigate in the web app. It is also possible to pass optional parameters. This dependency allows you to use the `<BrowserRouter>` tag, which enables navigation between multiple pages. There is also the possibility to use `<Route>` to specify the path to a page or component, or `<Switch>` to group several routers and route the page according to state (like any switch). Navigation is possible using `<Link>` or `<NavLink>`.

Parameters

It is possible to pass parameters through the URL and pass it to the view. For example `/menu/42` can be rendered in the view mapped to `/menu` with an input parameter `42`.

Forms

In React, there are two types of forms: controlled or uncontrolled. Controlled forms are directly and bidirectionally tied to the state, and every state mutation is associated with a

handler.

An uncontrolled form is not tied to the state but holds the values internally in the DOM. It is possible to retrieve data, for example, when sending information to a REST API and does not need to have a handler for every state update.

MVC Framework

A Model-View-Controller is an architectural pattern commonly used to allow reusable components, isolate logic from user interface and permit independent development, testing, and maintenance. The *Model* part usually is used to manage behavior and data, respond to requests about state or instruction to change state, and notify the View when there is a change in the state. The *View* is used to present the information in the user interface. A *Controller* receives the input from the users, instructs the Model on which action to perform, and initiates a response.

React is not a complete MVC framework (or its descendent Model-View View-Model). It only provides the presentation logic. The Model and Controller can be developed independently from the View using React.

Flux and Redux

The Flux architecture is an alternative to the MVC approach. It is a unidirectional data flow, from an action to the view through a dispatcher and a store. All updates have one unidirectional flow, and the central unit is the store. If a view wants to update the store, it must use an action. It cannot modify the store directly. New action is propagated through the systems in response to user interaction, and the dispatcher controls all the changes made to the store.

Redux

Redux is the place to store the application's state and allow to have a consistent way to access it. Redux is widely adapted by the React community but is not strictly connected to React itself. Redux makes state mutations predictable. The state is a single object and is the single source of truth, it is read-only, and the only way to change the state is through actions. Every change must be made with pure functions that take the previous state and return a newly mutated one. Using Redux is possible to implement logging utilities, API handling, undo/redo of a state, and many other features. The store holds the current state value as a plain JavaScript object. There are some utilities provided from Redux to manage the store:

- *createStore* - method that allow to create a store with an initial state
- *dispatch* - update the state with the provided action object

- *subscribe* - accepts a callback function that will be run every time an action is dispatched

The bindings between React and Redux are possible using the package *react-redux*. It will provide a function *connect()* that generates a wrapper container that subscribes to the store. It takes two additional arguments:

- *mapStateToProps()* - called every time the store state changes. Return an object full of data with each field being a prop for the wrapped component
- *mapDispatchToProps()* - receives the *dispatch()* method and should return an object full of functions that use *dispatch()*

Surrounding the React App with a tag `<Provider>` allows to provide the store as an attribute and make it accessible as a singleton to all the connected components.

React-Redux-Form

React-redux-form is a versatile, fast, and intuitive library for creating complex and performant forms in React using Redux, providing a collection of reducer and action creators. Form data are stored in a Redux store in a model. It is suitable to use when there is a need to persist form data across the component lifecycle.

Redux Actions

Redux Actions are payloads of information that are sent, through *store.dispatch()* from your application to the store to effect any change in the store's ticket. Every action should have a property *type*, usually a constant string, and a payload containing all the data necessary for updating the state.

Reducers

A reducer is a function that takes the previous and the data that should be updated. The last state is copied and merged with the data to create the next state. A reducer should never mutate directly the state.

Redux Middleware and Thunk

The Redux Middleware can run a code after an action is dispatched and before it reaches the reducer. It is a place that enables third-party extensions to be injected into your Redux application. For example, suppose the developer wants to log all the actions that have been dispatched. In that case, a middleware is a good point to capture the action, log what the action is, then let it move on to the reducer, and then log the application's state after the action is effected.

Middleware forms a pipeline that wraps around the `dispatch()`. Then it can make any modification and pass the action forward (pass-through). It can restart the dispatch pipeline and also access the store state. Is it helpful for: inspecting the actions and the state, modify actions, dispatch other actions or stop actions from reaching the reducers. It is used with `applyMiddleware()` that sets up the middleware pipeline and returns a "store enhancer" that is passed to `createStore()`.

Thunk

A subroutine is used to inject an additional calculation in another subroutine: delay a calculation until its results are needed or insert operations at the beginning or end of the other subroutine. Middleware allows writing action creators that return functions instead of an action. For example, it can be used to delay the dispatch of an action or dispatch only if a particular condition is met.

The inner function receives the `dispatch()` and `getState()` store methods. The state can be examined and decide if it allows the action to proceed. Is it useful for complex synchronous logic like multiple dispatches, conditional dispatches, or simple async logic.

Fetching Data

Since React is focused only on the view, it is possible to handle data fetching in multiple ways. The `fetch` API, a replacement for the old `XMLHttpRequest`, is the most commonly used fetching resource that provides a promise-based interface.

Animations

There were two libraries when the course was delivered, which were widely used for animations:

- *react-transition-group*
- *react-animnation-components*

The last update in *react-animnation-component* is dated 04 April 2018 and, because of that, was skipped during this course. On the other hand, *react-transition-group* is still widely used and maintained. It provides a set of components for managing component states (including mounting and unmounting) over time, specifically designed with animation in mind. The most common components are `<Transition>`, `<CSSTransition>`, and `<TransitionGroup>`. A `Transition` is used to describe the transition from one component to another with some states: `entered`, `exiting`, or `exited`. It is used to animate the mounting and unmounting of a component. A set of `<Transition>` components can be grouped in a list using the `<TransitionGroup>` tag. `CSSTransition` is applied together with `<Transition>`. It applies a pair of class names

during the transition's appear, enter, and exit stages. It also uses props to decide when to apply the transition classes

React Hooks

React Hooks were not covered in the course, but since they are a significant change made by React 16.8, I decided to go over them anyway. Hooks have radically changed how development is done in React, making class-defined components obsolete in favor of functional components.

Hooks provide some functions that allow for manipulating the internal state of functional components. For example, the `useState()` function permits the insertion and manipulation of the state within a component.

```
import React, { useState, useEffect } from 'react'

function Example() {
  const [count, setCount] = useState(0)

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`
  })

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

`useEffect` is another important hook that allows manipulating the state during the lifecycle of the component. It replaces the `componentDidMount`, `componentDidUpdate` and `componentWillUnmount` functions with a single function. It is possible to cause the component to render when one or more parts of the state are changed and define its custom hooks. Hooks can only be used by a component at the top level, not within loops or conditions.

Server-side Development with NodeJS, Express and MongoDB

The second course taken introduced server-side application development with Node.js and its integration with the MongoDB database.

Node.js

Node Modules

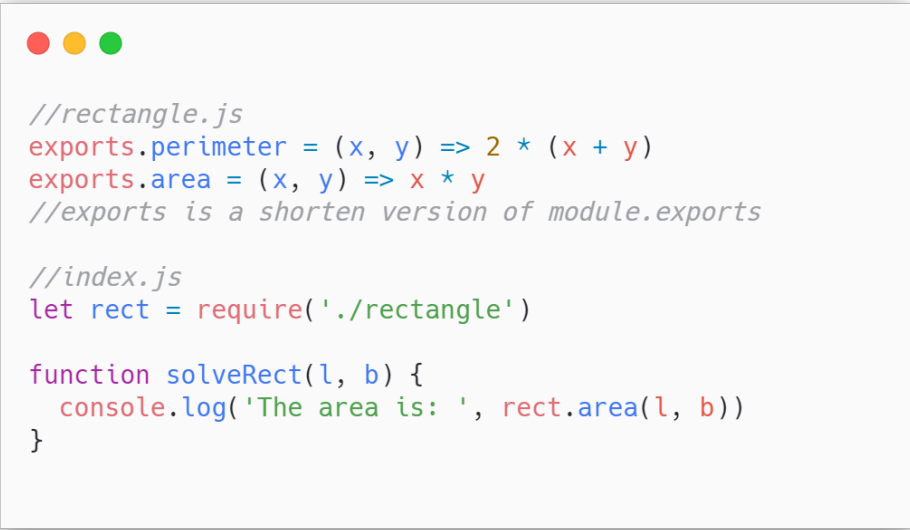
A Node.js application is developed in modules, where each file corresponds to a module. The CommonJS² project defines this principle.

The module variable gives access to the current module definition in a file that can be exported using *module.exports* or his shorthand *exports*. The *require* function is used to import a module. There are several categories of module:

- file-based modules
- core modules - part of Node's core like *path*, *fs*, or *util*
- external modules - third-party modules installed using a package manager.

A module can be included using the *require* function. A file-based module can be imported with *require(module)* specifying the relative path to the file. It is enough for the core and external module to specify the module name, and Node will automatically look for external modules starting from *node_modules* folder up the hierarchy until the module is found.

²<https://en.wikipedia.org/wiki/CommonJS>



```
//rectangle.js
exports.perimeter = (x, y) => 2 * (x + y)
exports.area = (x, y) => x * y
//exports is a shorten version of module.exports

//index.js
let rect = require('./rectangle')

function solveRect(l, b) {
  console.log('The area is: ', rect.area(l, b))
}
```

Figure 1: Example of a module and how to import

First-class functions and closures

Node.js relies heavily on first-class functions and closures. A first-class function is a function that can be treated the same way as any other variable. For example, that can be passed as an argument to another function. A closure is a function defined inside another function that has access to all the variables declared in the outer function (outer scope). The inner function will continue to have access to the other variables from the outer scope even after the external function has returned.

Callback and Error handling

Node.js is single-threaded, but at the same time, it can achieve a fast rate of completion of work. It is possible because of the judicious use of callbacks, the asynchronous execution of I/O requests, like file accesses or long-running processing that can be done independently behind the scenes.

Node.js continuously executes an event loop consisting of several phases:

- timers - executes callbacks scheduled by `setTimeout()` and `setInterval`
- I/O callbacks - executes almost all callbacks with the exception of close callbacks and timers
- idle, prepare - internally used

- poll - retrieve I/O events, incoming connections, data, and others.
- check - *setImmediate()* callbacks
- close callbacks - close callbacks like *socket.on('close', ...)*

Each of these phases maintains its own separate queue, and the Node.js event loop picks up requests from each of these queues, and handles them.

Networking

Since network operations can cause unexpected delays and data is not instantaneously available, there is a need to write applications recognizing the asynchronous nature of communication. Network communications happens using the HTTP Client-Server communication protocol using the `http` module. Using this module is possible to create a server with *http.createServer((res, req) +> ...)* and make it listen with *server.listen(port,...)*. The incoming messages are available through the *req* parameter.


Express

Express is the most popular third-party framework for building web servers. It is a fast, unopinionated, minimalist web framework for Node.js. It provides a robust set of features, and there are many third-party middlewares to extend functionality. An express application can be easily created using *express-generator* through NPM.

A middleware provide a lot of plug-in functionality that can be used within your Express application. For example, *morgan* is a plugin that can be used for logging

Router

Express support routing URI through *app.all*, *app.get*, *app.post*, *app.put* and *app.delete* methods. This methods can be used to construct REST service:



```
app.all('/dishes', function (req, res, next) {...})
app.get('/dishes', function (req, res, next) {...})
app.post('/dishes', function (req, res, next) {...})
app.put('/dishes', function (req, res, next) {...})
app.delete('/dishes', function (req, res, next) {...})
```

It is possible to specify URI with parameters and to parse the body of a request using the *body-parser* module.

There is also *Express Router* for creating mini-express application with routing abilities using *myRouter = express.Router()* and then *myRouter.route('/')*;

Authentication

In Express, a request/response can pass through several middlewares that elaborate and propagate the modified request/response to the following middleware. For example, middleware could control if a request contains the necessary information to authenticate the user. If the user can be authenticated, the request can be propagated using the *next()* function; otherwise, it returns an error.

Passport

A node module that is an authentication middleware for Node.js. It is modular and flexible, it supports various strategies for authentication (local, openID, OAuth singgle sign-on, ...) and sessions. Passport can be combined with Mongoose to simplify building login with username and password. It makes available Mongoose schema support for managing users, and it adds username, hash, and salt fields to store the username, the hashed password, and the salt value. It also provides additional methods to the User schema to support local authentication

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    passportLocalMongoose = require('passport-local-mongoose')
var User = new Schema({})
User.plugin(passportLocalMongoose)
module.exports = mongoose.model('User', User)

// requires the model with Passport-Local Mongoose plugged in
var User = require('./models/user')
// use static authenticate method of model in LocalStrategy
passport.use(new LocalStrategy(User.authenticate()))
// use static serialize and deserialize of model for passport session support
passport.serializeUser(User.serializeUser())
passport.deserializeUser(User.deserializeUser())
```

Is it also possible to implement token-based authentication, especially if the server must be stateless and scalable or if the authentication should be shared between multiple servers. Token-based authentication helps to handle Cross-Origin resource sharing (CORS) problems and Cross-site request forgery. The workflow is:

1. User requests access with username and password
2. Server validates credentials

3. Server create a signed token and send it to the client (nothing is stored in the server)
4. All subsequent requests from the client should include the token
5. Server verifies the token and responds with data if validated

To achieve token-based authentication in Express is possible to use `jsonwebtoken` and `Passport-JWT` modules. `Passport-JWT` creates and configures a new `Passport` strategy based on JWT authentication. It is possible to extract the JWT from an incoming request.

Sessions

`Sessions` uses a combination of cookies and server-side storage to track users. A session is stored by default in a permanent memory server-side and it can be wiped out when the server restarts. It can be implemented using a middleware.

```
var session = require('express-session')
var FileStore = require('session-file-store')(session)
app.use(
  session({
    name: 'session-id',
    secret: '12345-67890-09876-54321',
    saveUninitialized: false,
    resave: false,
    store: new FileStore(),
  })
)
// Session information are available as req.session
```

MongoDB

MongoDB is a document-based NoSQL DB. The document is a self-contained piece of information; it is possible to have a collection of documents, while a DB is a set of collections. A collection is a set of documents; a document is effectively a JSON file with some additional features. A NoSQL server can support multiple DBs. NoSQL's best features are scalability, availability, consistency, partition tolerance, and ease of deployment.

In MongoDB, documents are stored in a BSON (Binary JSON) format. It supports length prefix on each value, information about the field value type, and additional primitives type not supported by raw JSON like UTC date-time, raw binary, and `ObjectID`.

An ObjectId is a mandatory field `_id` created by MongoDB when a document is inserted. MongoDB driver provides a high-level API for a Node application to interact with the MongoDB server for performing operations like connection, insertions, deletions, updates, and documents querying.

Mongoose ODM

Mongoose ODM is a node module that gives a specific structure to documents and enforces the structure among all the applications. MongoDB stores data in the form of documents, but no structure is imposed on the document. Any document can be stored in any collection and relies on the developer's discipline to maintain the structure of the documents. MongoDB stores data in the form of documents, but no structure is imposed on the document. Any document can be stored in any collection and relies on the developer's discipline to maintain the structure of the documents.

Mongoose Schema

It defines all the fields of the document and their types. It can do validation. Various schema types are String, number, date, buffer, boolean, mixed, ObjectId, and array. The schema is used to create a model function. Mongoose's schema types include String, Number, Date, Buffer, Boolean, Mixed, ObjectId, and Array. The Array schema type allows the creation of an array of sub-documents inside the document. Once a schema is defined, it is used in Mongoose to create a model function, which allows determining the structure for the documents in the database. Schemas can be nested to enable supporting embedded or sub-documents.

Mongoose Population

NoSQL databases like MongoDB usually do not explicitly support relations like the SQL databases. All documents are generally expected to be self-contained. However, it is possible to store references to other documents within a document by using ObjectIds.



```
Dishes.find({})  
  .populate('comments.author')  
  .then((err, dish) => {})
```

Population automatically replaces specified paths within a document with documents from another collection using cross-reference with ObjectIds helps. For example, an object 'comment' can have a rating, a comment, and an author. It is possible to populate a Dish with a list of comments using Mongoose Population.

Conclusion

This document summarizes the major and most important topics covered during these two courses. Within each course, there were exercises and assignments that were performed and completed and are freely available at this address: https://github.com/gunghio-school/hk_react_course.git

In addition, a personal project covering most of the topics was carried out in parallel. It is a web app for booking articles for a butcher shop. The web app was developed using Next.js, a framework that combines frontend in React and backend in Node in a single application, and relies on a database in MongoDB.

The webapp can be accessed at carne.gunghio.ch after authentication using username *neggio* and password *matur*, while its source code is available at the following address: <https://github.com/OlmoBarberis/food-gunghio.git>.

In addition, all notes taken during the two courses are freely available at this address: https://gunghio.notion.site/HK_Seminar-fcf71778e4bf4c2081368321089c6f1e