基本設置

1. 建立資料夾proshop，進入後執行npx create-react-app fronted安裝react
2. 將fronted裡的.git刪除(使用指令)，並將.gitignore移到最上層的資料夾
3. .gitignore內，將/node\_modules改成node\_module並增加node\_modules/ 及 .env
4. 到最上層資料夾啟用git init安裝版本控制，之後git add . 及git commit –m

製作資料

製作component資料都要記得導出

export default products

安裝樣式

在fronted內npm install react-bootstrap

按需引入樣式

import { Container } from 'react-bootstrap'

建立路由

執行npm install react-router-dom@5 react-router-bootstrap

在App.js引入路由器及路由

import { BrowserRouter as Router, Route } from 'react-router-dom'

<Router>

      <Header />

      <main className="py-3">

        <Container>

          <Route path="/" component={HomeScreen} exact />

          <Route path="/product/:id" component={ProductScreen} />

        </Container>

      </main>

      <Footer />

    </Router>

(版本六需要引入Routes)

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom'

路由上改以element標示導入頁面並需要插入完整標籤

    <Router>...

          <Routes>

            <Route path="/" element={<HomeScreen />} exact />

          </Routes>

    </Router>

到Header引入react-bootstrap專用的路由

import { LinkContainer } from 'react-router-bootstrap'

將組件庫的標籤用LinkContainer包裹

          <LinkContainer to="/">

            <Navbar.Brand>Proshop</Navbar.Brand>

          </LinkContainer>

              <LinkContainer to="/cart">

                <Nav.Link>

                  <i className="fas fa-shopping-cart"></i>Cart

                </Nav.Link>

              </LinkContainer>

後端設置

在底層執行npm init並安裝npm install express

建立服務器位置

const express = require('express')

const app = express()

app.listen(5000, () => {

  console.log('Server is running on port 5000.')

})

運行服務器 到運行腳本把增加start

這樣運行只需輸入npm start

  "scripts": {

    "start": "node backend/server"

  },

新增backend資料夾並建立server.js及新增data資料夾存放資料

並將導出改成module.exports

module.exports = products

設定api讓服務器找到指定的json資料

app.get('/api/products', (req, res) => {

  res.json(products)

})

app.get('/api/products/:id', (req, res) => {

  const product = products.find((p) => p.\_id === req.params.id)

  res.json(product)

})

到客戶端安裝axios

以後端設置的api獲取資料

const ProductScreen = ({ match }) => {

  const [product, setProduct] = useState({})

  useEffect(() => {

    const fetchData = () => {

      axios.get(`/api/products/${match.params.id}`).then((res) => {

        let product = res.data

        setProduct(product)

      })

    }

    fetchData()

  }, [])

解決跨域問題

到package.json加入代理服務器

  "proxy": "http://localhost:5000/",

配置便捷的執行環境

執行npm install -D nodemon concurrently

增加運行配置

    "server": "nodemon backend/server",

    "client": "npm start --prefix frontend", //可以到前端資料夾執行npm run start

    "dev": "concurrently \"npm run server \" \"npm run client\"" //可以同時運行

建立隱藏環境

執行npm install dotenv並獲取

const dotenv = require('dotenv')

新增檔案.env到最上層，並加入隱藏內容

NODE\_ENV = development

PORT = 5000

const PORT = process.env.PORT || 5000

app.listen(PORT, () => {

  console.log(

    `Server is running in ${process.env.NODE\_ENV} mode on port ${PORT}.`,

  )

})

統一引入技術

在package.json加入

  "type":"module",

並將引入方式改成import from

引入檔案需要加上後綴檔名

import express from 'express'

import dotenv from 'dotenv'

import products from './data/products.js'

資料也要以export default方式導出

export default products

連接資料庫

得到資料庫URI

在後端資料夾新增config並在內建立db.js做為資料庫的連接檔

import mongoose from 'mongoose'

const connectDB = async () => {

  try {

    const connect = await mongoose.connect(process.env.MONGO\_URI, {

      useUnifiedTopology: true,

      useNewUrlParser: true,

    })

    console.log(`MongoDB connected: ${connect.connection.host}`)

  } catch (error) {

    console.error(`Error: ${error.message}`)

    process.exit(1)

  }

}

export default connectDB

到服務器引入

import connectDB from './config/db.js'

使用元件

connectDB() //需要放在dot.config()後面

強調提示訊息

執行npm install colors不需引入即可使用

但只能在console.log字串後連接樣式

  console.log(`Server is on ${process.env.NODE\_ENV} mode on port ${PORT}.`.bgGreen.italic)

  console.error(`Error: ${error.message}`.red.bold)

建立資料庫模型

資料庫模型中共用資料的格式

    user: {

      type: mongoose.Schema.Types.ObjectId, //\*

      required: true,

      ref: 'User',

    },

日期資料

paidAt: {

      type: Date,

    },

資料筆數不固定時用陣列儲存

資料筆數固定、不會再增加的資料用物件格式

時間戳記跟資料庫是同一層級

  {

    timestamps: true, //\*

  },

將data資料中products的\_id刪除，因為資料庫會自動添加

建立使用者資料

執行npm install bcryptjs並引入

import bcrypt from 'bcryptjs'

替設定的密碼加密

    password: bcrypt.hashSync('123456', 10),

上傳資料到資料庫

利用try catch格式做分流

上傳資料時先清空所有資料，避免重複

    await Order.deleteMany()

    await User.deleteMany()

    await Product.deleteMany()

上傳資料庫

    const createUsers = await User.insertMany(users)

    const adminUser = createUsers[0].\_id

    //遍歷商品並加入管理者id

    const sampleProducts = products.map((product) => {

      return { ...product, user: adminUser }

    })

    //上傳商品

    await Product.insertMany(sampleProducts)

清空資料庫

const destoryData = async () => {

  try {

    await Order.deleteMany()

    await Product.deleteMany()

    await User.deleteMany()

    console.log('Data Successfully Destoryed'.bgYellow)

    process.exit()

  } catch (error) {

    console.error(`${error.message}`.bgRed)

    process.exit(1)

  }

}

node流程控制

if (process.argv[2] === '-d') { //如果第二個參數是-d

  destoryData()

} else {

  importData()

}

配置運行腳本

    "data:import": "node backend/seeder",

    "data:destory": "node backend/seeder -d"

配置路由器

在服務器建立產品的路由器

import productRoutes from './routes/productRoutes.js'

將api改成middleware

表示此前綴的api會分流到productRoutes管理

app.use('/api/products', productRoutes)

建立路由器

執行npm install express-async-handler並引入，讓路由自動分流錯誤error handling

import AsyncHandler from 'express-async-handler'

使用AsyncHandler將第二個參數包裹

router.get(

  '/',

  AsyncHandler(async (req, res) => {

    const products = await Product.find({})

    res.json(products)

  }),

)

用Postman測試api 使用chrome的開發工具

增加新的collection後可以新增資料夾管理api

save儲存常用的api

增加api環境可以避免重複輸入api前綴

需在api前面增加 {{URL}} 代替前綴

錯誤管理

在底層新增middleware資料夾並建立errorMiddleware.js

找不到的錯誤

const notFound = (req, res, next) => {

  const error = new Error(`Not found - ${req.originalUrl}`) //打印出錯誤的url

  res.status(404)

  next(error) //將錯誤往下傳

}

服務器錯誤

const errorHandler = (err, req, res, next) => {

  const statusCode = res.statusCode === 200 ? 500 : res.statusCode

  res.status(statusCode)

  res.json({

    message: err.message,

    // stack: process.env.NODE\_ENV === 'production' ? null : err.stack,

  })

}

導出

export { notFound, errorHandler }

到服務器獲取並增加其middleware

app.use(notFound)

app.use(errorHandler)

因此路由器的錯誤可以直接寫成

      throw new Error('Product not found...')

而不需每次都使用json標示錯誤

      res.status(404).json({ message: 'Product not found...' })

使用redux

到前端執行npm install redux react-redux redux-thunk redux-devtools-extension

在前端資料夾的最上層建立redux資料夾並新增store.js

import { createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import { composeWithDevTools } from 'redux-devtools-extension' //使用redux開發工具

const reducer = combineReducers({}) //合併多個組件到store

const initialState = {}

const middleware = [thunk]

//讓store執行異步操作需要redux-thunk作為middleware

//而執行middleware需要引入applyMiddleware

const store = createStore(

  reducer,

  initialState,

  composeWithDevTools(applyMiddleware(...middleware)),

)

export default store

到index.js引入

import { Provider } from 'react-redux'

import store from './redux/store'

將App標籤以Provider包裹並傳入store功能

  <Provider store={store}>

    <App />

  </Provider>,

在redux資料夾建立constants資料夾並新增productConstants.js儲存處理商品的指令(通常大寫)

export const PRODUCT\_LIST\_REQUEST = 'PRODUCT\_LIST\_REQUEST'

export const PRODUCT\_LIST\_SUCCESS = 'PRODUCT\_LIST\_SUCCESS'

export const PRODUCT\_LIST\_FAIL = 'PRODUCT\_LIST\_FAIL'

最後批次引入到productReducer.js

在redux資料夾建立reducers資料夾並新增productReducers.js處理商品相關的流程

export const productListReducer = (state = { products: [] }, action) => {

  const { type, payload } = action //傳入動作類型

  switch (type) {

    case PRODUCT\_LIST\_REQUEST:

      return { loading: true, products: [] }

    case PRODUCT\_LIST\_SUCCESS:

      return { loading: false, products: payload }

    case PRODUCT\_LIST\_FAIL:

      return { loading: false, error: payload }

    default:

      return state

  }

}

state狀態複雜的情況下返回狀態的方法

      const newItem = payload

      const existItem = state.cartItems.find(

        (item) => item.productID === newItem.productID,

      )

      if (existItem) {

        return {

          ...state,

          cartItems: state.cartItems.map( //\*找到一樣的商品，換成後來傳入的物件

            (item) => (item.productID === existItem.productID ? newItem : item),

          ),

        }

      } else { //\*要返回跟原狀態一樣的格式

        return { ...state, cartItems: [...state.cartItems, newItem] }

      }

完成後批次導入到store.js

import { productListReducer } from './reducers/productReducers'

將此reducer放到combineReducers中

const allReducer = combineReducers({

  productList: productListReducer,

})

※productList為頁面指令，非常重要

在redux資料夾建立actions資料夾並新增productAction.js派發產品相關的操作指令

異步action操作指令及錯誤控制

export const listProducts = () => { //異步action值為函數

  return async (dispatch) => {

    try {

      dispatch({ type: PRODUCT\_LIST\_REQUEST })

      const { data } = await axios.get('/api/products')

      dispatch({

        type: PRODUCT\_LIST\_SUCCESS,

        payload: data,

      })

    } catch (error) {

      dispatch({

        type: PRODUCT\_LIST\_FAIL,

        payload:

          error.response && error.response.data.message

            ? error.response.data.message

            : error.message,

      })

    }

  }

}

目前頁面資料要交給redux處理，因此axios、useState已經不需要

引入action及兩個redux hook

import { listProducts } from '../redux/actions/productAction'

import { useDispatch, useSelector } from 'react-redux'

useDispatch負責派發指令

  const dispatch = useDispatch()

在元件中使用dispatch action可以即時更改狀態

onChange={(e) => {

  dispatch(addToCart(item.productID, Number(e.target.value)),

    )}

useSelector負責取得狀態

  const data = useSelector((state) => {

    return state.productList //productList就是在store設定的reducer指令

  })

  const { loading, products, error } = data //取得資料

useEffect根據依賴改變就派發指令渲染頁面

  useEffect(() => {

    dispatch(listProducts())

  }, [dispatch]) //\*表示派發指令改變就渲染頁面(可以放入多種依賴)

根據不同狀態做不同顯示

      {loading ? (

        <Loader />

      ) : error ? (

        <Message variant="danger">{error}</Message>

      ) : (

        <Row>...</Row>

      )}

從路由在組件獲得鉤子

          <Route path="/cart/:id?" component={CartPage} /> //?代表可選擇性的傳值

路由組件有match location history可以使用

const CartPage = ({ match, location, history }) => {

match取得傳送的值

  const productID = match.params.id

location取得指定的參數

  const qty = Number(location.search.slice(5))

history用來指定路徑

    history.push('/login?redirect=shipping')

在異步action的回調中可以得到getState

export const addToCart = (id, qty) => {

  return async (dispatch, getState) => {

在回調中以getState取得物件的狀態(找到該reducer)儲存資料

    // console.log(getState) //取得狀態的函數

    // console.log(getState()) //執行函數

    // console.log(getState().cart) //找到該reducer

    // console.log(getState().cart.cartItems) //該reducer中cart的狀態

    localStorage.setItem('cartItems', JSON.stringify(getState().cart.cartItems))

可以在異步action中將資料狀態放到本地儲存

到store.js取得本地儲存的資料

const cartItemsFromStorage = localStorage.getItem('cartItems')

  ? JSON.parse(localStorage.getItem('cartItems'))

  : []

將資料放到初始狀態

const initialState = {

  cart: { cartItems: cartItemsFromStorage },

}

route controller 簡潔化routes.js檔案

建立controller資料夾新增controller.js檔

import AsyncHandler from 'express-async-handler'

import Product from '../../models/productModel.js'

const getProducts = AsyncHandler(async (req, res) => {

  const products = await Product.find({})

  res.json(products)

})

導出

export { getProducts, getProductByID }

在routes.js檔就可以寫成這樣

router.route('/').get(getProducts)

router.route('/:id').get(getProductByID)

建立user Authentication

先利用postman測試路由是否可以用post request傳送req.body資料

在userModel.js自訂密碼比對方法

userSchema.methods.matchPassword = async function (enterPassword) {

  return await bcrypt.compare(enterPassword, this.password)

}

安裝jsonwebtoken製作token讓傳送的資料攜帶

export const authUser = AsyncHandler(async (req, res) => {

  const { email, password } = req.body

  const user = await User.findOne({ email })

  if (!user) {

    res.status(401).send({ message: 'User not found.' })

  } else {

    if (await user.matchPassword(password)) {

      //產生token

      const tokenObject = { \_id: user.\_id } //必須為物件

      const token = jwt.sign(tokenObject, process.env.JWT\_SECRET, {

        expiresIn: '30d',

      })

      res.json({ //傳送資料

        \_id: user.\_id,

        name: user.name,

        email: user.email,

        isAdmin: user.isAdmin,

        token: token, //攜帶token

      })

    } else {

      res.status(401).send({ message: 'Invalid email or password' })

    }

  }

})

在私人路由上建立路由閘門

設定私人路由並用postman測試

server.js的路由器

app.use('/api/users', userRoutes)

導入路由再設定controller並製作路由守衛

router.route('/profile').get(protect, getUserProfile) //放入路由守衛

路由守衛

import AsyncHandler from 'express-async-handler'

import jwt from 'jsonwebtoken'

import User from '../models/userModel.js'

export const protect = AsyncHandler(async (req, res, next) => {

  let token

  if (

    req.headers.authorization &&

    req.headers.authorization.startsWith('Bearer')

  ) {

    try {

      token = req.headers.authorization.split(' ')[1]

      const decode = jwt.verify(token, process.env.JWT\_SECRET)

      //   console.log(decode) //可以得到封裝進去的\_id

      req.user = await User.findById(decode.\_id).select('-password')

      //傳到controller不需要password

      next()

    } catch (error) {

      console.error(error)

      res.status(401)

      throw new Error('Fail! Not authorized token.')

    }

  }

  if (!token) {

    res.status(401)

    throw new Error('Sorry, No token to authorize.')

  }

})

私密路由的controller

export const getUserProfile = AsyncHandler(async (req, res) => {

  console.log(req.user) //可以得到middleware設定的user資料

  const user = await User.findById(req.user.\_id) //利用得到的id搜尋資料庫

  if (user) {

    res.json({

      \_id: user.\_id,

      name: user.name,

      email: user.email,

      isAdmin: user.isAdmin,

    })

  } else {

    res.status(404)

    throw new Error('User not found.')

  }

})

到postman設定token環境

到test欄位增加此配置pm.environment.set("TOKEN", pm.response.json().token)

意為TOKEN欄位放入postman得到json格式的回應，但只取token的部分

增加user資料及加密

到userModel.js設置加密程序

userSchema.pre('save', async function (next) {

  if (this.isModified('password') || this.isNew) { //修改密碼或薪資料都會進入加密

    const salt = await bcrypt.genSalt(10)

    this.password = await bcrypt.hash(this.password, salt)

    next()

  } else {

    next()

  }

})

route增加路由

router.route('/').post(registerUser)

controller設定增加資料流程

export const registerUser = AsyncHandler(async (req, res) => {

  const { name, email, password } = req.body

  const existUser = await User.findOne({ email })

  if (existUser) { //user是否已經存在

    res.status(400)

    throw new Error('User already exists')

  }

  const user = await User.create({ //創造使用者

    name,

    email,

    password,

  })

  const tokenObject = { \_id: user.\_id } //必須為物件

  const token = jwt.sign(tokenObject, process.env.JWT\_SECRET, {

    expiresIn: '30d',

  })

  if (user) {

    res.status(201).json({

      \_id: user.\_id,

      name: user.name,

      email: user.email,

      isAdmin: user.isAdmin,

      token: token, //攜帶token

    })

  } else {

    res.status(400)

    throw new Error('Invalid user data')

  }

})

金流系統

設定後端api並存入金流資料

export const updateToPaid = AsyncHandler(async (req, res) => {

  const order = await Order.findById(req.params.id)

  if (order) {

    //設定金流完成要存入資料庫的資料

    order.isPaid = true //付款狀態改為true

    order.paidAt = Date.now()

    //資料來自paypal

    order.paymentResult = {

      id: req.body.id,

      status: req.body.status,

      update\_time: req.body.update\_time,

      email\_address: req.body.payer.email\_address,

    }

    const updateOrder = await order.save() //\*儲存資料

    res.json(updateOrder)

  } else {

    res.status(404)

    throw new Error({ message: 'Order not found.' })

  }

})

路由

router.route('/:id/pay').put(protect, updateToPaid)

設定reducer

export const orderPayReducer = (state = {}, action) => {

  const { type, payload } = action

  switch (type) {

    case ORDER\_PAY\_REQUEST:

      return { loadingPay: true } //\*區分同一個畫面的其他reducer狀態

    case ORDER\_PAY\_SUCCESS:

      return { loadingPay: false, successPay: true } //不需要payload

    case ORDER\_PAY\_FAIL:

      return { loadingPay: false, errorPay: payload }

    case ORDER\_PAY\_RESET: //\*

      return {}

    default:

      return state

  }

}

設定action

export const payOrder = (orderID, result) => {

  return async (dispatch, getState) => {

    try {

      dispatch({ type: ORDER\_PAY\_REQUEST })

      const { token } = getState().userLogin.userInfo

      const config = {

        headers: {

          'Content-Type': 'application/json', //\* put request需要

          Authorization: 'Bearer ' + token, //私密路由需要

        },

      }

      //修改資料庫資料使用put request

      const { data } = await axios.put(`/api/orders/${orderID}/pay`, result, config)

      dispatch({

        type: ORDER\_PAY\_SUCCESS, //通知reducer並得到successPay渲染畫面

        payload: data,

      })

    } catch (error) {

      dispatch({

        type: ORDER\_PAY\_FAIL,

        payload:

          error.response && error.response.data.message //\*測試值

            ? error.response.data.message

            : error.message,

      })

    }

  }

}

使用paypal模擬金流操作

建立好paypal帳戶後，選擇sandbox模式到create new app創建系統

輸入系統名稱proshop建立後得到client ID並放入.env檔

到server.js設置請求client ID的api

app.get('/api/config/paypal', (req, res)=>{

  res.send(process.env.PAYPAL\_CLIENT\_ID)

})

回到訂單頁面order page

以axios取得設置的client ID並動態加載script金流識別到文件頁面

  useEffect(() => { //放在useEffect

    const addPaypalScript = async () => {

      const { data: clientID } = await axios.get('/api/config/paypal')

  //動態加載script金流識別文件到頁面文件

  const script = document.createElement('script')

      script.type = 'text/javascript'

      script.src = `https://www.paypal.com/sdk/js?client-id=${clientID}` //來自paypal SDK

      script.async = true

      script.onload = () =>{ //script載入時將SDK設定為true(默認為false)

        setSDK(true)

      }

      document.body.appendChild(script) //動態加載到文件頁面

    }

    if (!order || order.\_id !== orderID || successPay) {

      dispatch({ type: ORDER\_PAY\_RESET }) //\*先將付款狀態清空，避免沒有得到successPay重新整理

      dispatch(getOrderDetails(orderID)) //獲取頁面資料的reducer

    }else if(!order.isPaid){

      if(!window.paypal){ //如果頁面沒有自動渲染，就強制執行程式取得script

        addPaypalScript()

      }

    }

  }, [orderID, order, dispatch, successPay])

安裝react-paypal-button-v2

取得付款的reducer狀態，以此設定按鈕顯示及結帳狀態

  const { loadingPay, successPay } = useSelector((state) => state.orderPay)

          {!order.isPaid && ( //付款狀態為false才會顯示

            <ListGroup>

              {loadingPay && <Loader />} //一開始要RESET清空不然會一直重整

              {!SDK ? (

                <Loader />

              ) : (

                <PayPalButton

                  amount={order.totalPrice} //需要攜帶結帳金額

                  onSuccess={successPaymentHandler} //動作事件

                />

              )}

            </ListGroup>

          )}

paymentResult是根據paypal-button得到的結果

將結果存入資料庫

  const successPaymentHandler = (paymentResult) => {

    dispatch(payOrder(orderID, paymentResult))

  }