[header] → [sign-in-and-sign-up] + [shop]

[homepage] → [directory] → [menu-item]

[shop] → [collection-overview] → [collection-preview] → [collection-item]
→ [collection] → [collection-item]

[sign-in-and-sign-up] → [sign-in] + [sign-out] → [form-input] + [custom-button]

[cart-icon] → [cart-dropdown] → [cart-item]

[checkout] → [checkout-item] + [stripe-button]

Workflow:
1. create project

2. homepage: [homepage] → [directory] → [menu-item]
(state in directory component)

3. hat routing: withRouter

4. ShopPage: [shop] → [collection-preview] → [collection-item]
(state in shop component)

5. Header: [header] → [sign-in-and-sign-up] + [shop]
(no state)

6. Firebase-Sign in & Sign out
[sign-in-and-sign-up] → [sign-in] + [sign-out] → [form-input] + [custom-button]
(state in App.js/Sign in)
(Form Input 和 custom button 都只是改变了输入 style，没有的话程序也完全 work, App.js 里有 current user，仅传进了 header， sign in 用 state 来处理输入，仅用 auth 处理 sign in with google)

--push user information into firebase: createUserProfileDocument: if user doesn't exist create user in the database. Return the Ref
--load the user information into Current user state
--sign up component
--complete sign in

7. Redux
--create store: store.js & root-reducer.js & index.js
--**user**: user.actions.js & user.reducer.js & user.types.js → app.js & header.component.jsx
(import {connect} , {action to use} → delete corresponding state → {action} = this.props 如果是方法就是 mapDispatchToProps，如果只是 state 就是 mapStateToProps) 都可以 this.props.

--**cart**: cart.actions.js & cart.reducer.js & cart.types.js → **cart-icon.component.jsx** (**cart-dropdown.component.jsx**) & header.component.jsx

Add an ADD-TO-CART button to collection-item
Add Add-Item functionality: addItem → Improvement: multiple items (cart.utils)
cart-item & use cart redux in cart-dropdown
cart.selectors: **reselect** help save some rerenders off of our components
user.selectors

8. Checkout Page: [cart-dropdown] → [cart-item]
[checkout] → [checkout-item]

9. Clear item: clearItemFromCart
Remove item: removeItem
10. Local Storage: redux-persist

11. Move Directory & Collection (shop.data.js) state into Redux

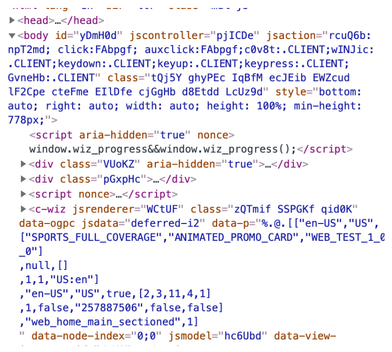
12. [collection] → [collection-item]
Improvement : [shop] → [collection-overview] → [collection-preview]
(shop 中没有 state， serve as routing direction)

13. Improve: make shop data an object (turn array searching to HashMap searching)

14. Stripe

[Learned]
Traditionally:
HTML: display the text on web pages
CSS: control styles
JavaScript: allows to have some interactivity in a Website

Developer tools 里面看到的就是 DOM，JavaScript modify the DOM, remove or update elements



```
▶<head>…</head>
▼<body id="yDmH0d" jscontroller="pjICDe" jsaction="rcuQ6b:
npT2md; click:FAbpgf; auxclick:FAbpgf;c0v8t:.CLIENT;wINJic:
.CLIENT;keydown:.CLIENT;keyup:.CLIENT;keypress:.CLIENT;
GvneHb:.CLIENT" class="tQj5Y ghyPEc IqBfM ecJEib EWZcud
lF2Cpe cteFme EIlDfe cjGgHb d8Etdd LcUz9d" style="bottom:
auto; right: auto; width: auto; height: 100%; min-height:
778px;">
    <script aria-hidden="true" nonce>
    window.wiz_progress&&window.wiz_progress();</script>
  ▶<div class="VUoKZ" aria-hidden="true">…</div>
  ▶<div class="pGxpHc">…</div>
  ▶<script nonce>…</script>
  ▶<c-wiz jsrenderer="WCtUF" class="zQTmif SSPGKf qid0K"
  data-ogpc jsdata="deferred-i2" data-p="%.@.[["en-US","US",
  ["SPORTS_FULL_COVERAGE","ANIMATED_PROMO_CARD","WEB_TEST_1_0
  _0"]
  ,null,[]
  ,1,1,"US:en"]
  ,"en-US","US",true,[2,3,11,4,1]
  ,1,false,"257887506",false,false]
  ,"web_home_main_sectioned",1]
  " data-node-index="0;0" jsmodel="hc6Ubd" data-view-
```

Different Browsers
jQuery: allow developers to easily interact with DOM (document object model) across browsers
has a unified easy API

Massive web applications
Backbone.js: library helps to organize JavaScript files, as JavaScript files getting bigger and bigger

Single Page Application: load application code once instead of making new request
(focus more on JavaScript) use JavaScript to display new things
(interact with the application without communicating with the server anymore)

Angular: allow developers to build large applications by forming these containers that well wrapped your project (MVM)

When it comes to massive programming application, Angular can't handle that good

React:

1. Don't touch DOM, I'll do it
Traditionally, libraries do imperative, directly change individual parts of your app→ hard to see relationships between events and all these edge cases (the change instruction will be released one by one)

React use declarative paradigm, where developers only need to tell React how the page should look like, then React itself will take care of the DOM updates

2. Building websites like Lego blocks

Reusable components between projects, small components

3. Unidirectional data flow

React components === function, state === function parameters. → these will create a virtual DOM (JavaScript version of DOM) like showing in the react developer tools, which contains the component, which is not actual in HTML language, while the real DOM is in HTML language.
Easy to debug code

4. UI, the rest is up to u- only a UI library

Angular gives the developers all the tools necessary to build an application
While React is only a UI library, but with this key concept, we can build the application across platform with different robot??? like React Native, React DOM.

Before Really do a React Application, what should we decide??

1. Components
2. State and where it lives
3. what changes when state changes

Create React App

Helps developers to get started with building react projects very quickly
Reason to use: React is actually written in a version of JavaScript that is further along than the version of JavaScript that our browsers understand, convert the react code we write into older version of JavaScript and HTML that our browser is able to understand. (web pack and Babel)

Create React App helps us spin up a project that already has all configuration done for us

Babel: make sure our JavaScript will work on all the browsers with any version
Webpack: is a module bundler, allows us writing a modular code.
Buddle everything up together and optimize it for production.
The build folder is what we will use to actually deploy our application.

Division

Class components: get access to state
State: a JavaScript object with properties that we can access at any point inside of our class
Super(): calls the method on the component class, give access to this.state, import the extend class method to current class

{} : is introduced by JSX which indicates anything between those curly braces is a JavaScript Code
setState(): allows us to change state
onClick: on every single HTML element, get called whenever this element get clicked, and the render method gets called again since the state changed

Life Cycle Methods: Methods that get called at different stages of when this component gets rendered.

Component Did Mount: json format is a format that JavaScript understand and can use

Props.children: is the thing that between the component brackets
onChange: is a character in input, the input changes fire the onChange with whatever function we pass to it//// get target.value gives us the value user type in in the input box!!

Never run setState in render it will cause mistakes, because every time we set state we will cause rerendering.   -------- 注：发现其实是可行的

Arrow function automatically bind this in arrow function to the place where this arrow function was defined in the first place

Event handler 里面，如果有（）,相当于这个 function 被 call 了

Browser can only read CSS, and create-react-app will convert Sass to CSS

看 state 是不是要在 upper level 是看这个 component 会不会受到其他 component 的影响，如果在 upper level，就说明如果一个 child component 改变了，当前的 child component 也需要改变，看有没有这个必要，如果没有这个必要就不必 lifting state

Query is something ask database for a collection

FIREBASE:
QueryReference: firestore.doc('/users/:userId) this will return a document reference for us either to **get** data from the reference or **save** data to that reference

DocumentReference vs CollectionReference:
documentRef: CRUD methods (create, retrieve, update, delete) ➔ documentRef.set(), get(), update(), delete()

(we can add documents to collections using the collectionRef.add())

Get the **<u>snapshotObject</u>** from the referenceObject using .get():
documentRef → documentSnapshot
collectionRef → querySnapshot

documentSnapshot allows us check if a document exists, using **.exists()**, we can use **.data()** to get the actual properties on the object (return a JSON object of the document)

querySnapshot (collectionSnapshot) contains all documents' documentSnapshot objects in that collection/ .empty() / .size() → how many documents are in this collection
-iterate to get all the data in each document:
{collectionsSnapshot.docs.map(doc => doc.data())}

REDUX

Redux allow none of the components hold state, all state are saved in a big store

Pros:
1. Good for managing large state
2. useful for sharing data between components
3. Predictable state management using the 3 principles:
   -Single source of truth
   -State is read only
   -Changes using pure functions

Redux 可以直接把一个 component 需要的相应 state 传递到相应的 component 里

Reselector:
make sure that cart dropdown component is not getting rerendered whenever the state changes that unrelated to the cart items
managed to help save some rerenders off of our components

Session Storage vs Local Storage
<u>Session Storage</u>: the storage persist when the tab exits, even when we refresh the page, but we will lose all the information once we close the tab.
<u>Local Storage</u>: the information get persist even when we close the browser or tab.

CSS in JS is just using a JavaScript library to render styles
Prevent styles leaking across components due to CSS global namespace (solved using BEM)

(don't have access to all of our selectors because HTML does not support all of the selectors that we want to add to our components ➔ solved by using **styled-components library**)

NoSQL database: we need to enforce ourselves to store the same values across the documents as we can.

HOC higher order component (with-spinner)

Promise fetch oriented API using firebase and existing database to deal with asynchronous
Observer is some piece of code that wraps around this stream of events, essentially has three function calls on it:
-Next: (nextValue) => {}: execute whenever a new event happens
-error: (error) => {} : whenever error occurs
-complete: () => {}: an optional call, occur if the stream if finished

Subscription: is essentially a way for us to tie our code using a listener, subscribing to this observer's stream of events.

Redux Thunk [in .action] a new library allows us to handle asynchronous event handling and firing multiple actions (a piece of middleware allows us to fire functions): similar to mapdispatchtoprops, returns a function that get dispatch in it, so whenever dispatch is called it will fire multiple actions

Allows us to do asynchronous actions

If redux-thunk middleware is enabled, any time you attempt to dispatch a function instead of an object, the middleware will call that function with dispatch method itself as the first argument

We moved our code out into a redux handled asynchronous event process and this pattern is very common pattern in redux especially for asynchronous event handling when it comes to having components that depend on external API to provide it with data

Container pattern: don't render anything , just pass props down to components, although results in more files, it helps keeping concerns separate to each specific component
Compose
Refactoring code using Higher Order Components to keep the code in a clear/concise way

Redux-Saga-library
An alternative and much more popular method of handling **side effect** in redux of asynchronous API requests

Its whole purpose is to run these sagas all concurrently—run them all together in a way that doesn’t block the execution.

A function that conditionally run, and the condition that it depends on whether or not a specific action is coming into saga middleware----can be multiple sagas listening for multiple different actions or the same action, but they are just piece of code do not run until they hear the action that they are listening for.

Side effects: either API calls to back end (asynchronous code) or something that triggers an impure action
Pure function: no matter how many times we call the function as long as the parameters are same, we will get exactly same result
Impure function: external dependency. [like an API call inside of componentDidMount()]

We should move it into sagas any asynchronous activity that happens inside of our applications that is not related to our component state but rather possibly related to the application

Generator Function [function*]: a function resembles async/await, pause execution whenever they see ‘yield’ inside of the function

When calling generator function, we only instantiate this generator object but the execution inside of the function is paused.
generator.next() will resume the execution
we can control when we want to move and continue execution in this function onwards
we replace things with redux saga to handle our asynchronous actions inside of redux

yiled → we are yielding control over this saga back to the saga middleware 将 yield 后面的 action 由 saga middleware 来控制

{ take } has only one argument, waits for this action to happen and we can get the payload of that action. The rest of code does not execute until this take operate.
But!! Only fire once! We can’t get back to the top of the saga again→ using while loop

{ takeEvery } listens for every action of a specific type that we pass to it, creates a non-blocking call in order to not stop our application to continue running either other sagas or whatever the user wants to do ---more like an asynchronous pattern

{ takeLatest } terminate all previous call, only take the last one

{ call } take its first argument, some function, the subsequent arguments will be the parameters that will be passed into this function

{ put } for creating action – like dispatch

{ all } takes an array of sagas

Promise oriented fetch style of using sagas

Hooks a new feature, introduced in February 2019, react 16.8

Why hooks: developers become very confusing about which lifecycle methods to use and keep track of state and state triggers render

Hooks is a way for us to write functional components but gain access to new functionality that was previously only available to us if we wrote class component (we can't write hooks in class component)

Hooks must be called at the very top level of our component

{ useState } -gives us back a array with two values, the first one is the state value we are trying to set, the second is a function allows us to set this property.
-We pass the initial value of this state into useState as parameter
-we are able to use the useState Hooks as many times as possible as we want to instantiate
-when to use: build a feature or a component need local state but don't need any lifecycle method

{ useEffect } -gives us the ability to fire side effects inside of our functional components
-doesn't get back any value
-use a function that gets called whenever the component changes or whenever the component updates/rerenders
→if don't want the useEffect to fire every time the component rerender, pass a second parameter, an array, the properties that this effect listen for
→only one time when application mounted, pass empty array []
-we can actually pass back a function from a function we pass to it → cleanup function → useEffect calls when that component unmounts

(array distructuring : const arr = [1, 2, 3] → const [a, b, c] = arr)

ComponentDidMount

```
//Class
componentDidMount() {
    console.log('I just mounted!');
}

//Hooks
useEffect(() => {
    console.log('I just mounted!');
}, [])
```

ComponentWillUnmount

```
//Class
componentWillUnmount() {
    console.log('I am unmounting');
}

//Hooks
useEffect(() => {
    return () => console.log('I am unmounting');
}, [])
```

ComponentWillReceiveProps

```
//Class
componentWillReceiveProps(nextProps) {
    if (nextProps.count !== this.props.count) {
        console.log('count changed', nextProps.count);
    }
}

//Hooks
useEffect(() => {
    console.log('count changed', props.count);
}, [props.count])
```

Backend to complete a full stripe integration
The secret key of the stripe needs to be hidden on the backend server

NODE: node is an environment that allows us to run JavaScript outside of the browser
REST: representational state transfer
API: application programming interface
Bodyparser.json(): allows any of the request coming in, process their body tag and convert it to Json.
Urlencoded: make sure the URL do not contain anything illegal
Cors: cross origin request, our server and client serves at different origin/port/place, while most web servers will check whether the request origin are from the same place

Context API
Redux library leverages the context API,
Context API essentially is a way for us to store different states and modify those states inside of some separate context object which we would then be able to hook into from components at any level which eliminates

GraphQL
Is actually a server language that wraps around an existing database or server that you can make requests against in a different way from REST API, it just exposes one single endpoint, /graphQL, from this endpoint, we make all of its requests. In order to make a request, we need to pass it either a query or a mutation.

Query: ask for data
Mutation: modify data but essentially they all take a similar shape (like a JSON object).
So, graphQL solve over fetching data, and we don’t have to be aware of all of the different end points, whatever the back end server changes we don’t need to know from our UI, we just need to know the shape of this data, and we can query for certain things