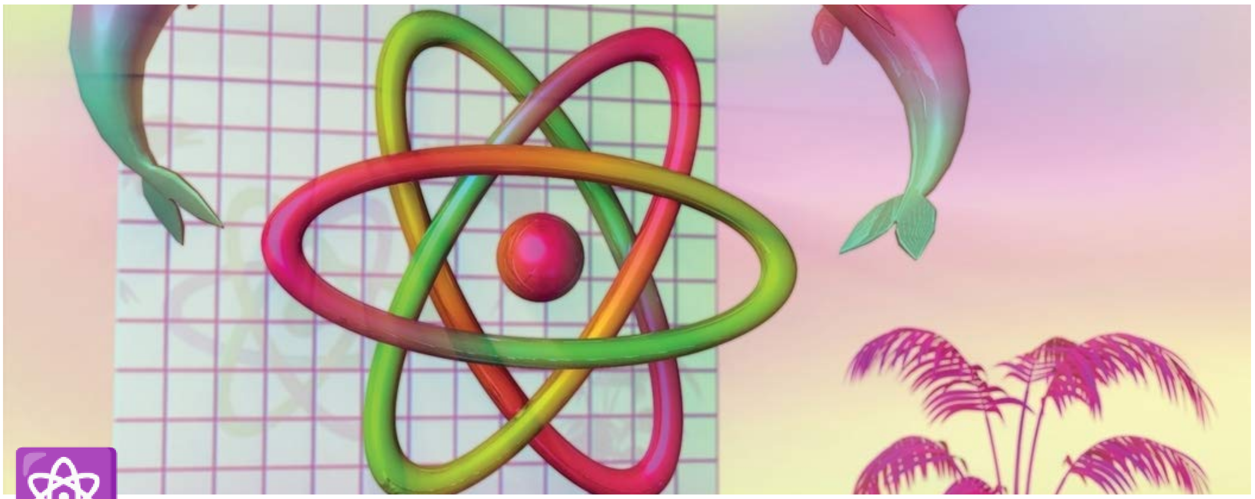


React - Notes

Here are my notes about React, hope they will help you with your programming journey.

Hansu👨‍💻

YT: https://www.youtube.com/channel/UCoSN5_vypjiJ8leuabiWLyg

tiktok: https://www.tiktok.com/@hansu_anders

insta: https://www.instagram.com/hansu_anders/

TIPS:

- type ls -1p to see the project tree
- https://semantic-ui.com/usage/theming.html - like twitters bootstrap
- `npx create-react-app .`

VISUAL STUDIO CODE SHORTCUTS:

- command + - // zoom in zoom out
- option arrows // jump word
- command arrows //jump to start/end

- shift arrows //select text
- command option j //console in **chrome**
- command shift g // and paste the path to file to **go to file**
- shift option f // align code
- control + ` //open terminal in vs code
- COMMAND + SHIFT + N //new incognito window chrome
- control + shift + right arrow //expand the smart selection
- control + shift + left arrow //shrink the smart selection
- .html file → html:5 //create basic template for html file
- npm init -y //just don’t answer the questions during installation
- command + , //open settings in vscode
- command + shift + p //search for files in your vscode project
- select word ⇒ command + d // selects other occurrences of the word, then you can just replace the word with a different name
- select function ⇒ alt + option // to move a block of text up and down

STEPS OF BUILDING A WEB APPLICATION:

1. Break the app into components
2. Build a static version of the app
3. Determine what should be stateful
4. Determine in which component each piece of state should live
5. Hard-code initial states
6. Add inverse data flow
7. Add server communication

BASIC RULES IN REACT.JS:

💡
1. We think about and organize our React apps as components
2. Using JSX inside the render method
3. Data flows from parent to children through props
4. Event flows from children to parent through functions
5. Utilizing React lifecycle methods
6. Stateful components and how state is different from props
7. How to manipulate state while treating it as immutable

Questions

▼ What is Redux?

Redux is a state management paradigm based on Facebooks Flux architecture.

Redux provides a structure for a large state trees and allows you to decouple user interaction in your app from state change.

▼ GraphQL

Powerful, typed, REST API alternative where client describes the data it needs.

▼ Relay

Glue between **GraphQL** and **React**. Relay is data-fetching library that makes it easy to write flexible, performant apps without a lot of data-fetching code.

▼ JavaScript ES6/ES7

ECMAScript or ES - you can think of a ES6 as a 6th version of javascript.

▼ What does framework do when inputs for a component change

Re-renders that component.

▼ What is JSX

JSX (JavaScript
eXtension syntax), a syntax extension for JavaScript written by Facebook. Using JSX enables us to
write the markup for our component views in a familiar, HTML-like syntax. In the end, this JSX
code compiles to vanilla JavaScript. Although using JSX is not a necessity, we'll use it in

this book
as it pairs really well with React.

```
React.createElement('div', {className: 'ui items'},
'Hello, friend! I am a basic React component.'
)
Which can be represented in JSX as:
<div className='ui items'>
Hello, friend! I am a basic React component.
</div>
```

▼ DOM

The Document Object Model (DOM) refers to the browser's HTML tree that makes up a web page.

▼ `ReactDOM.render([what], [where]);`

▼ What does Babel do?

Babel transpiles code to ES5. When its ES6+

▼ nativeHTML vs ReactComponent naming:

In React, native HTML elements always start with a lowercase letter whereas
React component names always start with an uppercase letter.

▼ The only required method in React component:

render() is the only required method for
a React component.

▼ What does braces in_ `id={` `product.id` `}` mean?

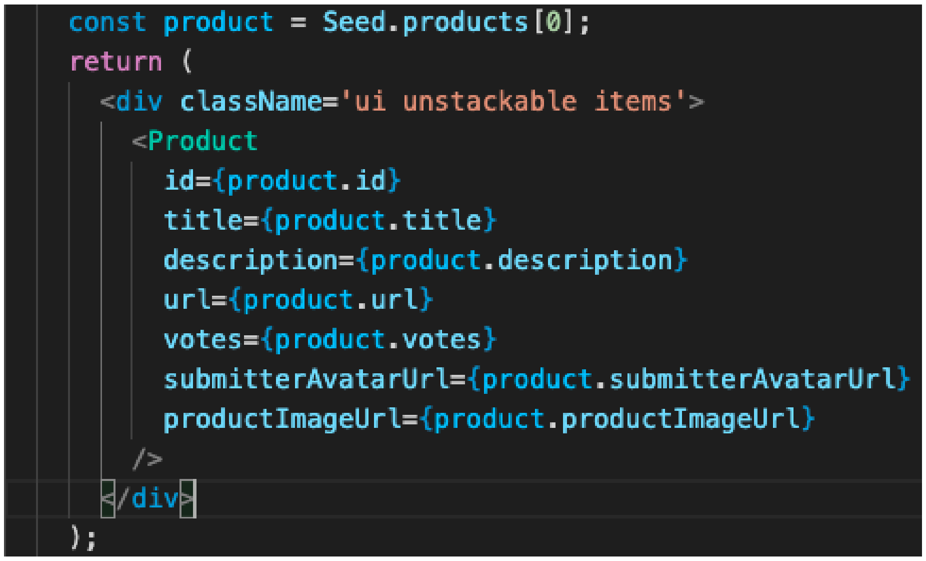
In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is a
JavaScript expression

▼ Why should you choose `let` and `const` instead of `var`

We encourage the use of const and *let* instead of *var*. In addition to the restriction introduced by
const, both *const* and *let* are block scoped as opposed to function scoped. This scoping

can help
avoid unexpected bugs.

▼ How to send `props` to component?



```
const product = Seed.products[0];
return (
  <div className='ui unstackable items'>
    <Product
      id={product.id}
      title={product.title}
      description={product.description}
      url={product.url}
      votes={product.votes}
      submitterAvatarUrl={product.submitterAvatarUrl}
      productImageUrl={product.productImageUrl}
    />
  </div>
);
```

▼ How to retrieve props of the component

{this.props.PROPNAME}

```
<div className='image'>
<img src={this.props.productImageUrl} />
</div>
<div className='middle aligned content'>
<div className='header'>
<a>
  <i className='large caret up icon'/>
</a>
  {this.props.votes}
</div>
<div className='description'>
<a href={this.props.url}>
  {this.props.title}
</a>
<p>
  {this.props.description}
</p>
</div>
```

▼ What are the boundaries of `this` in most cases

of the majority of this book, this will be bound to the *React component class.*
So, when we write this.props inside the component, we’re accessing the props property on the component. When we diverge from this rule in later sections, we’ll point it out.

▼ How to convert an array in json into array in React?

To render multiple products, first we’ll have ProductList generate an array of Product components.
Each will be derived from an individual object in the Seed array. We’ll use map to do so:

```
class ProductList extends React.Component {
  render() {
    const productComponents = Seed.products.map((product) => (
    <Product
      key={'product-' + product.id}
      id={product.id}
      title={product.title}
      description={product.description}
      url={product.url}
      votes={product.votes}
      submitterAvatarUrl={product.submitterAvatarUrl}
      productImageUrl={product.productImageUrl}
    />
));
```

▼ Why should you choose arrow functions?

They are so called anonymous functions, and are much simpler. Compare these 2 functions:

```
const cities = [{ name: 'Cairo', pop: 7764700 },{ name: 'Lagos', pop: 8029200 },];

1.
const pops = cities.map(city => city.pop);
console.log(pops);
// [ 7764700, 8029200 ]
2.
const pops = cities.map(function(city) { return city.pop });

Arrow functions possibilities:

const formattedPopulations = cities.map((city) => {
  const popMM = (city.pop / 1000000).toFixed(2);
  return popMM + ' million';
});
console.log(formattedPopulations);
// [ "7.76 million", "8.03 million" ]
```

```
const formattedPopulations = cities.map(
(city) => ((city.pop / 1000000).toFixed(2) + ' million'));
```

▼ What is the key property in a list?

Note the use of the `key={'product-' +` `product.id` `}` prop.

React uses this special property
to create unique bindings for each instance of the Product component.

The key prop is not
used by our Product component, but by the React framework.

For the time being,
it's enough to note that this property **needs to be unique per React component in a list**.

▼ What does sort do with array besides sorting it?

Mutates the array, it is a dangerous action

▼ Can child modify its props?

While the child **can read** its props, it **can't modify** them. A child does not own its props

In
our app, the parent component ProductList owns the props given to Product.

▼ What is *one-way data flow?*

React favors the
idea of one-way data flow. This means that data changes come from the "top" of the app and are
propagated "downwards" through its various components. Parent is the owner of props of the child. Child cannot changes its props.

▼ What is *immutable*?

In JavaScript, if we treat an array or object as immutable it means we cannot or should not make modifications to it.

▼ How to make a communication between a child and a parent?

We can pass down functions as props too.

We can have the `ProductList` component give each
Product component a function to call when the up-vote button is clicked.

Functions passed down
through props are the canonical manner in which children communicate events with their parent components.

Example:

1.

```
<a onClick={this.handleUpVote}>
<i className='large caret up icon'/>
</a>
```

2.

inside product:

```
handleUpVote(){
this.props.onVote(this.props.id);
}
```

3.

inside productList → giving props to children:

```
onVote={this.handProductUpVote}
```

4.

productList function:

```
handProductUpVote(productId) {
console.log(productId + ' was upvoted.');
}
```

▼ What does `this` return in method in component?

NULL, For the render() function, React binds *this* to the component for us, but for the methods we need to bind it ourselves.

How to bind *this* to the method and make it not return *NULL* but actual *Component*:

```
class MyReactComponent extends React.Component {
  constructor(props) {
    super(props); // always call this first

    // custom method bindings here
    this.someFunction = this.someFunction.bind(this);
  }
}
```

What does `super(props)` do?

`super()` is used to call the parent constructor.

`super(props)` would pass `props` to the parent constructor.

From your example, `super(props)` would call the `React.Component` constructor passing in `props` as the argument.

The React.Component class that
our Product class is extending defines its own constructor(). By calling **super(props)**, we're
invoking that constructor() function first.

Importantly, the *constructor() function defined by React.Component* will bind *this* inside our
`constructor()` to the `component`.

```
this.handleUpVote = this.handleUpVote.bind(this);
Function's bind() method allows you to specify what the this variable inside a function body
should be set to.
What we're doing here is a common JavaScript pattern.
 We're redefining the
component method handleUpVote(), setting it to the same function but bound to this (the
component).
Now, whenever handleUpVote() executes, this will reference the component as
opposed to null.
```

```
class MyReactComponent extends React.Component {
  constructor(props) {
    super(props); // always call this first

    // custom method bindings here
    this.someFunction = this.someFunction.bind(this);
  }
}
```

▼ Why do we use super(props)

To make the `constructor()` function defined by `React.Component` bind `this` inside our constructor to our component.

▼ Does arrow functions automatically bind `this`?

We introduced arrow functions earlier and mentioned that one of their benefits was how they ***bind the*** `this` ***object***.

PART 2

▼ By whom is `state` owned?

Whereas `props` are **immutable** and owned by a component's parent, `state` is **owned by the component**

▼ When will the `component` re-render itself?

Critically, when the state or props of a **component update**, the component will re-render itself.

▼ Is React Component with given set of `state` and `props` always render in the same way?

Every React component is rendered as a **function of** its `this.props` **and** `this.state`. This rendering is deterministic. This means that given a set of `props` and a set of `state`, a React component
will always render a single way

▼ Should you always initialize components with "empty" state?

It's **good** practice to initialize components with "empty" state as we've done here.

▼ Why is this line valid? `class ProductList extends React.Component {`

```
// ...
componentDidMount() {
this.state = Seed.products;
}
// ...
}
```

This is **invalid**.

The only time we can modify the state in this manner is in constructor().
For all state modifications after the initial state, React provides components the method `this.setState()`.

<u>So to sum up:</u>

Never modify state outside of `this.setState().` This function has important hooks around state modification that we would be bypassing.

▼ Should we treat the this.state object as immutable?

We just discussed that we can only make state modifications using `this.setState()`.

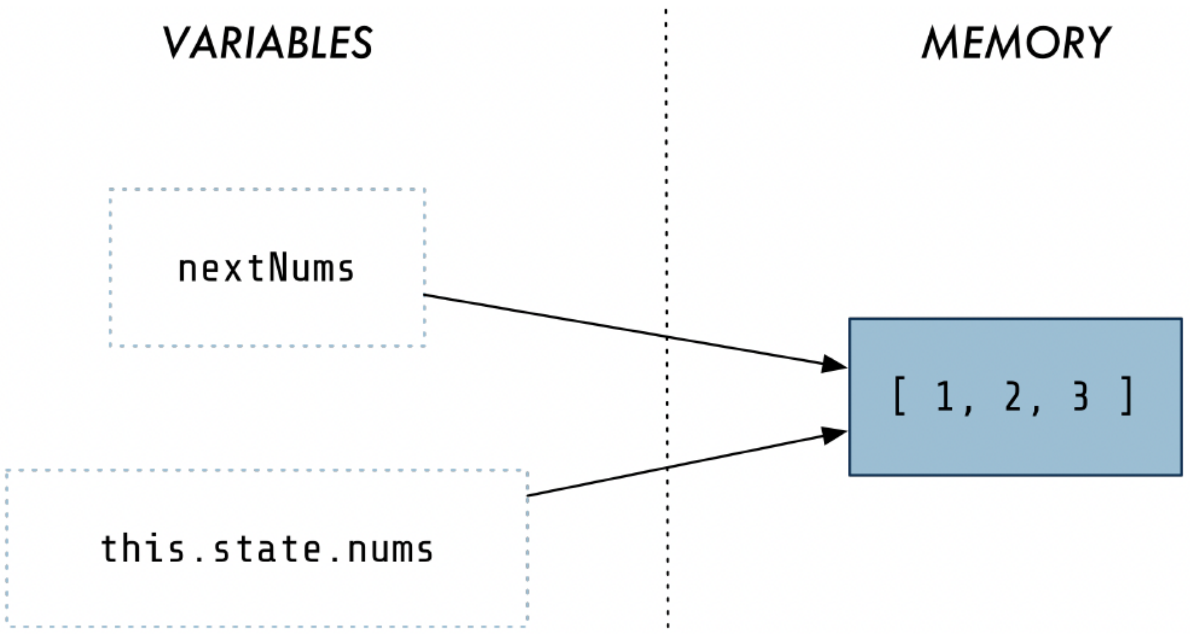
So while a component can update its state, we should treat the `this.state` object as **immutable**.

▼ Can we change `state` of the object with `push`?

As touched on earlier, if we treat an array or object as immutable we never make modifications to it.

`push()` method modifies the original array:

```
const nextNums = this.state.nums.push(4);
this.setState({ nums: nextNums });
```



▼ Can we change `state` of the object with `concat`?

As touched on earlier, if we treat an array or object as immutable we never make modifications to it.

We can use Array's `concat()`. `concat()` creates a new array that contains the elements of the array it was called on followed by the elements passed in as arguments.

```
console.log(this.state.nums);
// [ 1, 2, 3 ]
const nextNums = this.state.nums.concat(4);
console.log(nextNums);
// [ 1, 2, 3, 4]
console.log(this.state.nums);
// [ 1, 2, 3 ] <-- Unmodified!


//example of usage in react
createTimer = (timer) => {
  const t = helpers.newTimer(timer);
  this.setState({
    timers: this.state.timers.concat(t),
  });
};
```

▼ Can we change `state` of the object with `map`?

`map()` to traverse the products array. Importantly, `map()` returns a **new array** as opposed to modifying the array `this.state.products`.

Wrong:

```
handleProductUpVote(productId) {
  const products = this.state.products;
  products.forEach((product) => {
  if (product.id === productId) {
    product.votes = product.votes + 1;
  }
});
this.setState({
  products: products,
});
}
```

When we initialize products to `this.state.products`, **product** references the same array in memory as `this.state.products`

Good:

```
// Inside ProductList
handleProductUpVote(productId) {
  const nextProducts = this.state.products.map((product) => {
  if (product.id === productId) {
    return Object.assign({}, product, {
    votes: product.votes + 1, //cannot modify product.votes += 1, cuz it still
    //references the original array
    });
  } else {
    return product;
  }
  });
  this.setState({
    products: nextProducts,
  });
}
```

▼ When to use `Object.assign()`?

We use `Object#assign()` to create new objects as opposed to modifying existing ones.

`Object#assign()` accepts any number of objects as arguments. When the function receives two
arguments, it copies the properties of the second object onto the first, like so:

```
const coffee = { };
const noCream = { cream: false };
const noMilk = { milk: false };
Object.assign(coffee, noCream);
// coffee is now: `{ cream: false }`
Object.assign(coffee, noMilk);
// coffee is now: `{ cream: false, milk: false }`
```

It is idiomatic to pass in three arguments to `Object.assign()` as we do above.

The first argument is a new JavaScript object, the one that `Object#assign()` will ultimately return.

The second is the object whose properties we'd like to build off of.

The last is the changes we'd like to apply:

```
const coffeeWithMilk = Object.assign({}, coffee, { milk: true });
// coffeeWithMilk is: { cream: false, milk: true }
// coffee was not modified: { cream: false, milk: false }
//Object.assign() is a handy method for working with immutable
//JavaScript objects.
```

▼ What is ES2015

ES2015 is just another name used for ES6

▼ Stages of JavaScript

Beyond ES7, proposed JavaScript features can exist in various stages. A feature can be an
experimental proposal, one that the community is still working out the details for ("stage 1").
Experimental proposals are at risk of being dropped or modified at any time. Or a feature might
already be "ratified," which means it will be included in the next release of JavaScript ("stage 4").

▼ How to avoid constructor and make `this` available in function instantly?

With the transform-class-properties plugin, we can write `handleUpVote` as an arrow function.
This will ensure this inside the function is bound to the component, as expected:

```
class Product extends React.Component {
handleUpVote = () => (
this.props.onVote(this.props.id)
);
render() {
//Using this feature, we can drop constructor(). There is no need for the manual binding call.
```

▼ Provide alternative way to define the initial state of component than using constructor

Instead of:

```
class ProductList extends React.Component {
constructor(props) {
super(props);
this.state = {
products: [],
};
this.handleProductUpVote = this.handleProductUpVote.bind(this);
}
```

Do this:

property initializers

```
class ProductList extends React.Component {
state = {
products: [],
};
```

Arrow function to make `this` to bound the component

```
handleProductUpVote = (productId) => {
const nextProducts = this.state.products.map((product) => {
if (product.id === productId) {
return Object.assign({}, product, {
votes: product.votes + 1,
});
} else {
return product;
}
});
this.setState({
products: nextProducts,
});
}
```

▼ For what do we use property initializers?

1. We can use arrow functions for custom component methods (and avoid having to bind `this`)
2. We can define the initial state outside of `constructor()`

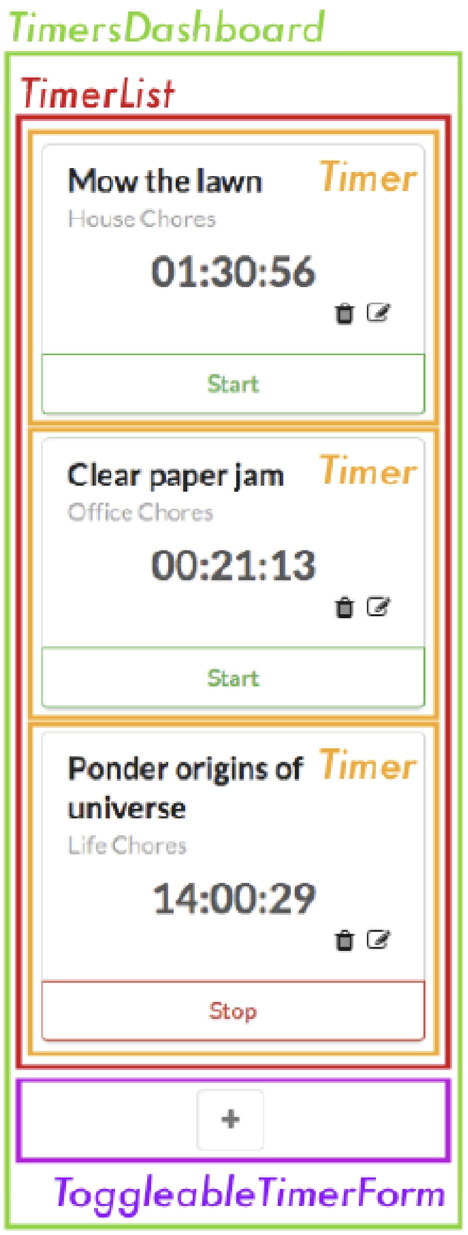
▼ Name the most basic grounds of the React app

1. We think about and organize our React apps as components
2. Using JSX inside the render method
3. Data flows from parent to children through props
4. Event flows from children to parent through functions
5. Utilizing React lifecycle methods
6. Stateful components and how state is different from props
7. How to manipulate state while treating it as immutable

▼ What is *single responsibility principle*?

To think about components as you would with functions or objects.

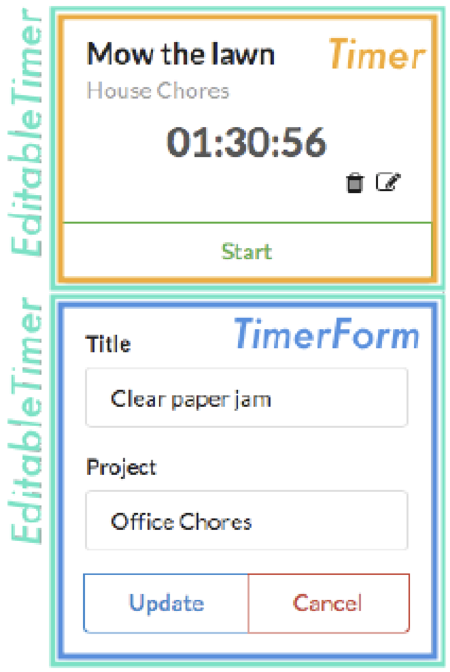
A component should, ideally, **only be responsible for one piece of functionality**.

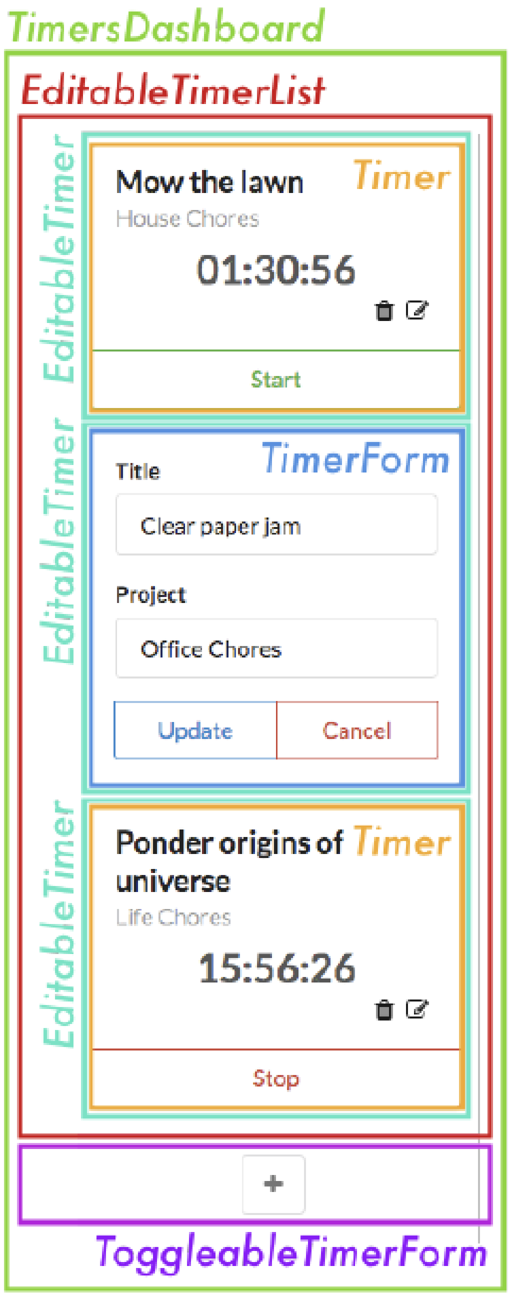


▼ What are the advantages of separating the responsibilities of components?

It improves **re-usability**, you can drop the TimerList component anywhere in the app where you want to display list of timers. And the add button can be accessible only when you log in or something.

▼ Example of breaking app into components


EditableTimer
Timer
Mow the lawn
House Chores
01:30:56
Start
EditableTimer
TimerForm
Title
Clear paper jam
Project
Office Chores
Update
Cancel



▼ **How should you create properties of the component that is responsible for editing/creating objects**

pass every property filled and when it is the 'creation mode' pass empty inputs.

▼ **How would you create a button that pops up a form?**

```
class ToggleableTimerForm extends React.Component {
render() {
if (this.props.isOpen) {
  return (
    <TimerForm />
  );
} else {
  return (
    <div className='ui basic content center aligned segment'>
      <button className='ui basic button icon'>
      <i className='plus icon' />
      </button>
    </div>
  );
  }
}
}
```

▼ What is `span` tag?

https://www.w3schools.com/tags/tag_span.asp

- The `<span>` tag is an inline container used to mark up a part of a text, or a part of a document.
- The `<span>` tag is easily styled by CSS or manipulated with JavaScript using the class or id attribute.
- The `<span>` tag is much like the <div> element, but <div> is a block-level element and `<span>` is an inline element.

▼ What does `ReactDOM` function do?

```
ReactDOM.render(
  <TimersDashboard />,
  document.getElementById('content')
);
```

Again, we specify with `ReactDOM#render()` which React component we want to render and where in our HTML document (*index.html*) to render it.

In this case, we're rendering *TimersDashboard* at the *div* with the *id* of content.

▼ How to detect if data should be stateful?

1. Is it passed in from a parent via props? If so, it probably isn't state.
A lot of the data used in our child components are already listed in their parents. This criterion helps
us de-duplicate.
For example, "timer properties" is listed multiple times. When we see the properties declared in
EditableTimerList, we can consider it state. But when we see it elsewhere, it's not.

2. Does it change over time?

If not, it probably isn't state.
This is a key criterion of stateful data: it changes.

3. Can you compute it based on any other state or props in your component?

If so, it's not
state.
For simplicity, we want to strive to represent state with as few data points as possible.

▼ Where should you store state data in you app?

💡 For each piece of state:
• Identify every component that renders something based on that state.
• Find a common owner component (a single component above all the components
that need the state in the hierarchy).
• Either the common owner or another component higher up in the hierarchy should
own the state.
• If you can't find a component where it makes sense to own the state, create a new
component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

▼ How to create *unique id's*?

For the `id` property, we're using a library called **uuid**.

We load this library in `index.html`.

We use `uuid.v4()` to randomly generate a Universally Unique IDentifier for each item.
A **UUID** is a string that looks like this:
2030efbd-a32f-4fcc-8637-7c410896b3e3

▼ Props vs. state

With your renewed understanding of React's state paradigm, let's reflect on props again.

Remember, props are state's immutable accomplice.

What existed as mutable state in Timers-Dashboard is passed down as immutable props to EditableTimerList.

We talked at length about what qualifies as state and where state should live.

Mercifully, we do not
need to have an equally lengthy discussion about props.

Once you understand state, you can see
how props act as its one-way data pipeline.

State is managed in some select parent components
and then that data flows down through children as props.

If state is updated, the component managing that state re-renders by calling `render()`.

This, in turn, causes any of its children to re-render as well. And the children of those children. And on and on down the chain.
Let's continue our own march down the chain.

▼ What does this line mean? `title: this.props.title || '',`

If TimerForm is
creating a new timer as opposed to editing an existing one, those props would be undefined. In that
case, we initialize both to a blank string ('').

▼ Why separate `handleCreateFormSubmit()` and `createTimer()`?

While not strictly required, the idea here is that we have one function for handling the event
`(handleCreateFormSubmit())` and another for performing the operation of creating a timer `(createTimer()).`

This separation follows from the **Single Responsibility Principle** and enables us to call `createTimer()` from elsewhere if needed.

▼ What is CRUD

Create, update, and delete

▼ How to force a React component to re-render?

We can use React's `forceUpdate()` method. This forces a React component to re-render.

We can call it on an interval to yield the smooth appearance of a live timer.

▼ How to run a function every *X* ms.

```
componentDidMount() {
this.forceUpdateInterval = setInterval(() => this.forceUpdate(), 50);
}
```

`setInterval().` This will invoke the
function `forceUpdate()` once every 50 ms, causing the component to re-render.

`setInterval()` returns a unique interval ID. You can pass this interval ID to `clearInterval()` at any time to halt the interval.

`clearInterval()` to stop. usage:

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

▼ What function is called when before a component is removed from the app?

`componentWillUnmount()` is called before a component is removed from the app.

▼ Wouldn’t it be more efficient if we did not continuously call forceUpdate()
on timers that are not running?

Indeed, we would save a few cycles. But it would not be worth the added code complexity. React will call render() which performs some inexpensive operations in JavaScript. It will then compare this result to the previous call to render() and see that nothing has changed.
It stops there — it won’t attempt any DOM manipulation.

▼ What is `!!` for?

```
<TimerActionButton
timerIsRunning={!!this.props.runningSince}
```

We use `!!` here to derive the boolean prop `timerIsRunning` for TimerActionButton. `!!` returns false when runningSince is null.

▼ Methodology review

1. **Break the app into components**
   We mapped out the component structure of our app by examining the app’s working UI. We
   then applied the single-responsibility principle to break components down so that each had
   minimal viable functionality.
2. **Build a static version of the app**
   Our bottom-level (user-visible) components rendered HTML based on static props, passed
   down from parents.
3. **Determine what should be stateful**
   We used a series of questions to deduce what data should be stateful. This data was represented
   in our static app as props.
4. **Determine in which component each piece of state should live**
   We used another series of questions to determine which component should own each piece of

state. `TimersDashboard` owned timer state data and `ToggleableTimerForm` and `EditableTimer`
both held state pertaining to whether or not to render a `TimerForm`.

5. **Hard-code initial states**
   We then initialized state-owners' state properties with hard-coded values.
6. **Add inverse data flow**
   We added interactivity by decorating buttons with `onClick` handlers. These called functions
   that were passed in as props down the hierarchy from whichever component owned the
   relevant state being manipulated.
7. **Add server communication**

▼ When we go to localhost:3000/ what does the server return to us?

returns index.html

`$ curl -X GET localhost:3000/api/timers`

The `-X` flag specifies which HTTP method to use. It should return a response that looks a bit like this

▼ How to start a timer using curl

```
$ curl -X POST \
-H 'Content-Type: application/json' \
-d '{"start":1456468632194,"id":"a73c1d19-f32d-4aff-b470-cea4e792406a"}\ '\
localhost:3000/api/timers/start

we need to send start (timerstamp)
and id of the timer to start
```

▼ What does -H flag mean in curl

Sets a header for our HTTP request, Content-Type. We're informing the server that the body of the request is JSON.

▼ What does -d flag mean

Sets the body of pur request. Inside the single quotes is the JSON data.

▼ Whats does ‘\’ backslash do in command prompt on linux and macOS?

Note that the backslash \ above is only used to break the command out over multiple lines for readability. This only works on macOS and Linux. Windows users can just type it out as one long string.

▼ What does jq do?

macOS and Linux users: If you want to parse and process JSON on the command line, we highly recommend the tool “jq.”
You can pipe curl responses directly into jq to have the response pretty-formatted:

```
curl -X GET localhost:3000/api/timers | jq '.'

You can also do some powerful manipulation of JSON,
 like iterating over all objects in the response and
returning a particular field. In this example,
 we extract just the id property of every object
in an array:

curl -X GET localhost:3000/api/timers | jq '.[] | { id }'
```

▼ Why is client.getTimers wrong?

When we call `client.getTimers(),` the network request is made asynchronously. The function call itself is not going to return anything useful:

// Wrong
// `getTimers()` does not return the list of timers

const timers = client.getTimers();

Instead, we can pass `getTimers(`) a success `function. getTimers()` will invoke that function after it hears back from the server if the server successfully returned a result. getTimers() will invoke the function with a single argument, the list of timers returned by the server:

```
// Passing getTimers() a success function
client.getTimers((serverTimers) => (
// do something with the array of timers, serverTimers
));
```

▼ What happens after calling `getTimers()`

`getTimers()` is invoked, it fires off the request to the server and then returns control flow immediately. The execution of our program does not wait for the server's response.

`getTimers()` is called an asynchronous function.

▼ How is the success function that we pass to the `getTimers` called?

callback

When you finally hear back from the server, if it's a successful response, invoke this function." This asynchronous paradigm ensures that execution of our JavaScript is not blocked by I/O

▼ How to ensure good data on client?

We also do one other interesting thing in componentDidMount. We use setInterval() to ensure `loadTimersFromServer()` is called every 5 seconds.

▼ Who is the master holder of state?

The server is considered the master holder of state. Our client is a mere replica. This becomes incredibly powerful in a multi-instance scenario. If you have two instances of your app running — in two different tabs or two different computers — changes in one will be pushed to the other within five seconds.

▼ Older alternative to fetch?

XMLHttpRequest

▼ What are the arguments for fetch?

The path to the resource we want to fetch

• An object of request parameters

By default, Fetch makes a GET request

▼ How to tell the server that we will accept json only?

```
accept application/json
```

```
function getTimers(success) { return fetch('/api/timers', {
headers: {
Accept: 'application/json',
},
}).then(checkStatus)
.then(parseJSON)
.then(success);
```

▼ What does fetch return?

Fetch returns a promise.

allows you to chain .`then`() statements.

We pass each .`then`() statement a function.

What we're essentially saying here is: "Fetching the timers from /api/timers then check the status code returned by the server.

Then, extract the JavaScript object from the response.

Then, pass that object to the success function

▼ What does property method mean in fetch?

method: The HTTP request method. `fetch`() defaults to a GET request, so we specify we'd like a **POST** here

▼ -=- body

The body of our HTTP request, the data we're sending to the server.

▼ Example of fetch post

```
function startTimer(data) {
return fetch('/api/timers/start', {
method: 'post',
body: JSON.stringify(data),
headers: {
'Accept': 'application/json',
'Content-Type': 'application/json',
},
}).then(checkStatus);
```

▼ Why do we still manually make the state change within React? Can’t we just inform the server of the action taken and then update state based on the server, the source of truth?

Again, this is valid. However, the user experience will leave something to be desired.

Right now, clicking the start/stop button gives **instantaneous** feedback because the state **changes locally** and React immediately re-renders.

If we waited to hear back **from the server**, there might be a noticeable **delay** between the action

this is called **optimistic updating**

▼ How does complete map method look like?

`arr.map(function(element, index, array){  }, this);`

The callback `function()`
 is called on each array element, and the `map()`
 method always passes the current `element`
, the `index`
 of the current element, and the whole `array`
 object to it.

▼ How to run script on first load of the page

```
componentDidMount() {
this.forceUpdateInterval = setInterval(() => this.forceUpdate(), 50);
}
```

▼ Does `setState` return a promise?

setState does not return a promise.

setState has a callback.

```
this.setState({
    ...this.state,
    key: value,
}, () => {
    //finished
});
```