Time Logging React App

Due to the high level of difficulty in setting up the required project environment for the Time Logging App, we will start this exercise from a boilerplate folder. The project root folder should look like the following:

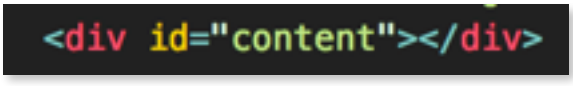
Inside the js folder, lies app.js and some helper modules; most of our work will revolve around these files. The figure below illustrates the content of the js folder.

Noting the lack of node_modules folder, you first need to run npm install to install all dependencies based on the specifications specified in our package.json file. The following screenshot illustrates the package dependencies for this project.

```
{
  "name": "timers_app",
  "version": "1.1.0",
  "scripts": {
    "start": "babel-node server.js"
  },
  "babel": {
    "presets": [
      "react"
    ],
    "plugins": [
      "transform-class-properties"
    ]
  },
  "private": true,
  "dependencies": {
    "babel-cli": "6.22.2",
    "babel-core": "6.22.1",
    "babel-plugin-transform-class-properties": "^6.24.1",
    "babel-preset-react": "^6.24.1",
    "body-parser": "1.14.1",
    "express": "4.13.3",
    "fs": "0.0.2",
    "path": "0.12.7"
  }
}
```

After installing all dependencies using npm install, you can now run npm start to start a Node development server and you should see an empty app titled Timer. The development server is configured to serve our public folder at path /. The server also responds to various other functionalities including POST and GET requests.

If you can recall, we have an index.html file as a template for our React app. Apart from all the obvious sections importing external libraries for React functionalities and styling, we have to also pay attention to a section:



```
<div id="content"></div>
```

This is the part where your app should be mounted, so bear that in mind when you are mounting in your app.js. Also note that since all of the dependencies are manually inlined in the index.html's head tag, there is no need to import them anymore in the your app.js.

App in a Bigger Picture

Before one starts building any app, and the rule is not limited to just web apps, one should always sketch their app up to have a good idea of their app. Production level developers will often sketch up their app idea using Sketch, Adobe Illustrator, Adobe XD, and many other applications, to present to their clients. But often times, pen and paper will do just fine if you're building it in-house.

Visually, you can safely assume there are at least 2 components in play - the TimerList and Timer component. TimerList is holding an array of Timer components, displaying them as illustrated above. This is the simplest possible description of our application.

Based on our sketch of the application UI, TimerList is a container holding our three Timer components. It has a further piece of functionality: the + button at the bottom of the suggests that users should be able to create new Timers by clicking it.

As we saw in the course, Components can be implemented as functions or classes. In software engineering, the term **component** is usually taken to mean a standalone program providing simple functionality that can be combined with other components to form complex programs like a full-blown app. When we design React Components, we should keep this in mind: try to keep the functionality of the components you create simple so that they are both easy to use as easy to understand. With this in mind, we perform a minor edit, shrinking TimerList's responsibility back to just listing Timer components - the functionality to add new Timer components will instead be handled by a new parent component - the TimersDashboard. TimersDashboard will have TimerList and the button as children.

How you name your components is up to you, but having some consistent rules around language as we do here will greatly improve code clarity.

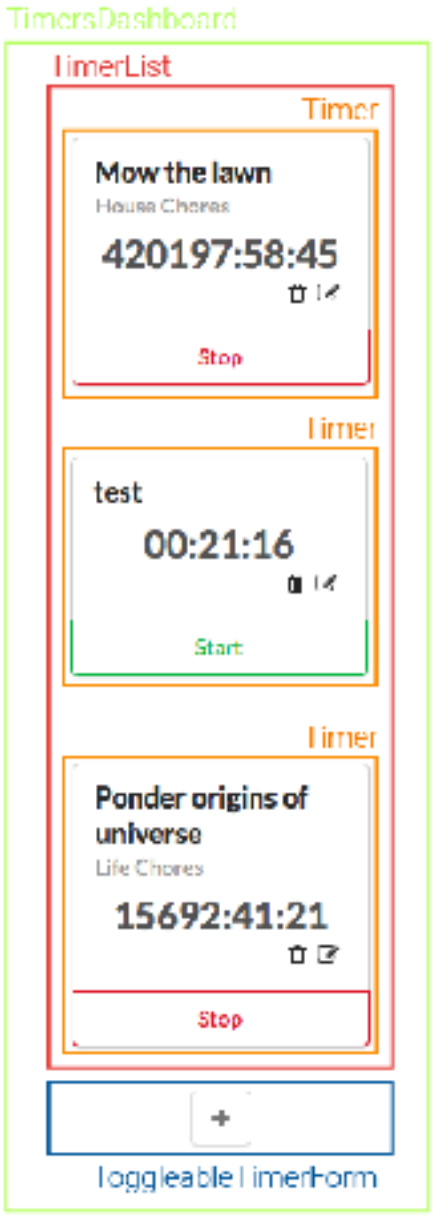
For example, developers can quickly reason that any component they come across that ends in the word "List" simply renders a list of children and no more.

With that in mind, our first sketch of components need to be revised to the following:

Not only does this separation of responsibilities keep components simple, but it often also improves their re-usability. In the future, we can now drop the TimerList component anywhere in the app where we just want to display a list of timers. This component no longer carries the responsibility of also creating timers, which might be a behaviour we want to have for just this dashboard view.

The + button is interesting because it has two distinct representations. When the + button is clicked, the widget transmutes into a form. When the form is closed, the widget transmutes back into a + button.

There are two approaches we could take. The first one is to have the parent component, TimersDashboard decide whether or not to render a + component or a form component based on some piece of stateful data. It could swap between the two children. However, this adds more responsibility to TimerDashboard. The alternative is to have a new child component own the single responsibility of determining whether or not to display a + button or a create timer form. We'll call it ToggleableTimerForm. As a child, it can either render the component TimerForm or the HTML markup for the + button.



The updated sketch of components are as illustrated in the following:

You can see that going through the process of sketching and identifying overburdened components gives us a good practice to plan ahead before starting to actually writing code.

Now that we have a sharp eye in that, we should be able to see another issue over here.

The Timer itself has a fair bit of functionality. It can transform into an edit form, delete itself, and start and stop itself. Do we need to break this up? And if so, how?

Exercise: Figure out how would you group the components to represent this functionality in the Timer Component.

Displaying a timer and editing a timer are indeed two distinct UI elements. They should be two distinct React components. Like ToggleableTimerForm, we need some container component that renders either the timer's face or its edit form depending on if the timer is being edited.

We call this EditableTimer. The child of EditableTimer will then be either a Timer component or the edit form component. The form for creating and editing timers is very similar, so let's assume that we can use the component TimerForm in both contexts:

As for the other functionality of the timer, like the start and stop buttons, it's a bit tough to determine at this point whether or not they should be their own components. We can trust that the answers will be more apparent after we've written some code.

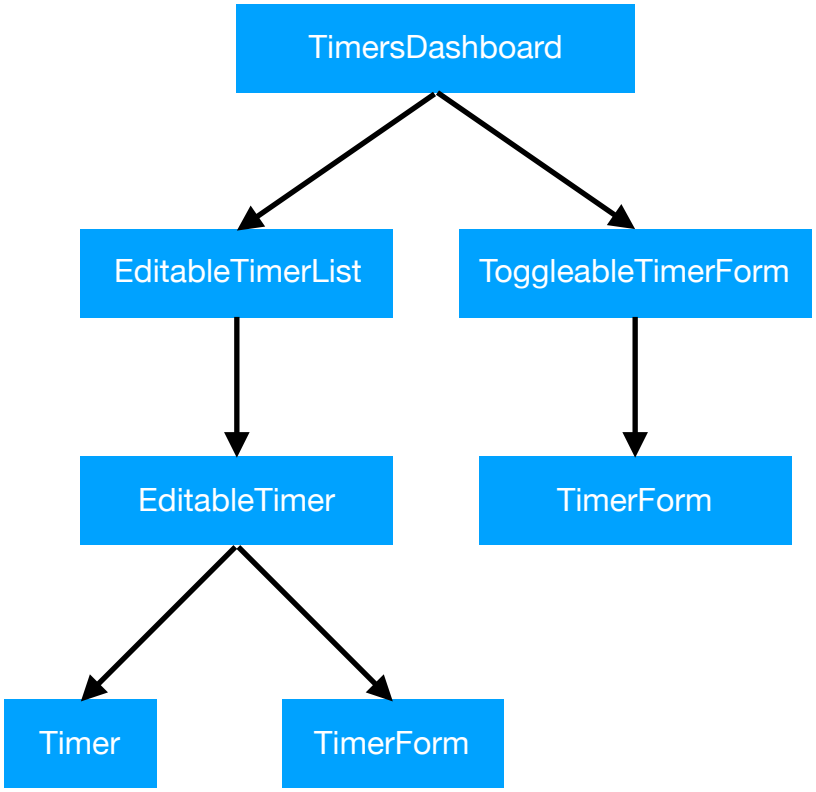
Working back up the component tree, we can see that the name TimerList would be a misnomer. It really is a EditableTimerList, and everything else looks good.

So we have our final component hierarchy, with some ambiguity around the final state of the timer component.

Let's see the revised sketch of components (Component chart on the next page):

TimersDashboard: Parent Container
- EditableTimerList: Display a list of timer containers
  - EditableTimer: Displays either a timer of a timer's edit form
    - Timer: Displays a given timer
    - TimerForm: Displays a given timer's edit form
- ToggleableTimerForm: Displays a form to create a new timer
  - TimerForm (not displayed): Displays a new timer's create form

Represented as hierarchical tree:



Naturally, our top level component will communicate with a server. The server will be the initial source of state, and React will render itself according to the data the server provides. Our app will also send updates to the server, like when a timer is started.

It will simplify things for us if we start off with static components. Our React component will do little more than render HTML. Clicking on buttons won't yield any behaviour as we will not have wired up any interactivity. This will enable us to lay the framework for the app, getting a clear idea of how the component tree is organised.

Next, we can determine what the state should be for the app and in which component it should live. We'll start off by just hard-coding the state into the components instead of loading it from the server.

At that point, we'll have the data flow from parent to child in place. Then we can add inverse data flow, propagating events from child to parent.

Finally, we'll modify the top level component to have it communicate with the server.

The following lists down a handy framework for developing a React app from scratch:

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

Build a Static Version of the App

TimersDashboard

Let's start off with the TimersDashboard component. All of the React code for this project will be inside the file public/app.js.

Begin by defining a familiar function, render() for the TimersDashboard component.

```
class TimersDashboard extends React.Component {
    render() {
        return (
            <div className="ui three column centered grid">
                <div className="column">
                    <EditableTimerList />
                    <ToggleableTimerForm
                    isOpen={true}
                    />
                </div>
            </div>
        )
    }
}
```

This component renders its two child components nested under div tags. TImersDashboard passes down one prop to ToggleableTimerForm:isOpen. This is used by the child component to determine whether to render a + or timerForm. When ToggleableTimerForm is open, the form is being displayed. The classes here are purely for styling purposes, thanks for Semantic UI.

EditableTimerList

We will define EditableTimerList next. We'll have it render two EditableTimer components. One will end up rendering a timer's face. The other will render a timer's edit form:

```
class EditableTimerList extends React.Component {
    render() {
        return (
            <div id="timers">
                <EditableTimer
                    title='Learn React'
                    project='Web Domination'
                    elapsed='8986300'
                    runningSince={null}
                    editFormOpen={true}
                />
            </div>
        )
    }
}
```

We're passing five props to each child component. The key difference between the two EditableTimer components is the value being set for editFormOpen. We'll use this Boolean to instruct EditableTimer which sub-component to render.

EditableTimer

EditableTimer returns either a Timerform or a Timer based on the props editFormOpen. Create a EditableTimer component that renders a TimerForm component and passes down title and project if it's open, else renders a Timer component and passes down elapsed and runningSince.

TimerForm

We'll build an HTML form that will have two input fields. The first input field is for the title and the second is for the project. It also has a pair of buttons at the bottom:

Look at the input tags, we're specifying that they have type of text and then we are using the React property defaultValue. When the form is used for editing as it is here, this sets the fields to the current values of the timer as desired.

Later, we'll use TimerForm again within ToggleableTimerForm for creating timers. ToggleableTimerForm will not pass TImerForm any props. this.props.title and this.props.project will therefore return undefined and the fields will be left empty.

At the beginning of render(), before the return statement, we define a variable submitText. This variable uses the presence of this.props.title using a ternary operator to determine what text the submit button at the bottom of the form should display. If title is present, we know we're editing an existing timer, so it displays Update. Otherwise, it displays Create.

With all of this logic in place, TimerForm is prepared to render a form for creating a new timer or editing an existing one.

ToggleableTimerForm

Recall that this is a wrapper component around TimerForm. It will display either a + or a TimerForm. Right now, it accepts a single prop, isOpen, from its parent that instructs its behaviour:

```
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>
            )
        }
    }
}
```

As noted earlier, TImerForm does not receive any props from ToggleableTimerForm. As such, its title and project fields will be rendered empty.

The return statement under the else block is the markup to render a + button. You could make a case that this should be its own React component (say PlusButton), but at present we'll keep the code inside ToggleableTimerForm.

Timer

Time for the Timer component.

```
class Timer extends React.Component {
    render() {
        const elapsedString = helpers.renderElapsedString(this.props.elapsed);
        return (
            <div className="ui centered card">
                <div className="content">
                    <div className="header">{this.props.title}</div>
                    <div className="meta">{this.props.project}</div>
                    <div className="center aligned description">
                        <h2>{elapsedString}</h2>
                    </div>
                    <div className="extra content">
                        <span className="right floated edit icon">
                            <i className="edit icon" />
                        </span>
                        <span className="right floated trash icon ">
                            <i className="trash icon" />
                        </span>
                    </div>
                    <div className="ui bottom attached blue basic button">Start</div>
                </div>
            </div>
        )
    }
}
```

elapsed in this app is in milliseconds. This is the representation of the data that React will keep. This is good representation for machines, but we want to show our carbon-based user a more readable format.

We use a function defined in helpers.js, renderElapsedString(). You can pop open that file if you're curious about how it's implemented. The string it render is in the format 'HH:MM:SS'.

With all of the components defined, the last step before we can view our static app is to ensure we call ReactDOM's render(). Render the TimersDashboard component into our DOM at id content.

Determine What Should be Stateful

In order to bestow our app with interactivity, we must evolve it from its static existence to a mutable one. The first step is determining what, exactly, should be mutable. We can apply criteria to determine 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.
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.

Applying the Criteria

TimersDashboard — isOpen boolean for ToggleableTimerForm
Stateful. The data is defined here. It changes over time. And it cannot be computed from other state or props.

EditableTimerList — Timer properties
Stateful. The data is defined in this component, changes over time, and cannot be computed from other state or props.

EditableTimer — editFormOpen for a given timer
Stateful. The data is defined in this component, changes over time, and cannot be computed from other state or props.

Timer — Timer properties
Not stateful. Properties are passed down from the parent.

TimerForm
We might be tempted to conclude that TimerForm doesn't manage any stateful data, as title and project are props passed down from the parent. However, as we'll see, forms are special state managers in their own right.

So outside of TimerForm, we've identified our stateful data:
- The list of timers and properties of each timer
- Whether or not the edit form of a timer is open
- Whether or not the create form is open.

Determine in Which Component Each Piece of State Should Live

While the data we've determined to be stateful might live in certain components in our static app, this does not indicate the best position for it in our stateful app. Our next task is to determine the optimal place for each of our three discrete pieces of state to live.

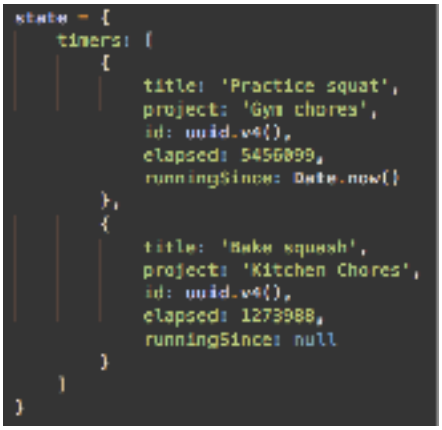
For each piece of state:
- Identify every component that renders something based on that state
- Find a common owner component (a single component above all the components that need the sate in the hierarchy)
- Either the common owner or another component higher up the hierarchy should own the state
- 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.

Hard-Code Initial States

We're now well prepared to make our app stateful. At this stage, we won't yet communicate with the server. Instead, we'll define our initial states within the components themselves. This means hard-coding a list of timers in the top-level component, TimersDashboard. For our two other pieces of state, we'll have the components' forms closed by default.

After we've added initial state to a parent component, we'll make sure our props are properly established in its children.

Start by modifying TimersDashboard to hold the timer data directly inside the component. The following shows the timer data to hold, or you can make some of your own.



Make sure you pass down the timers state down to EditableTimerList as props. For the id property, we're using a library called uuid. We've loaded this library in index.html. We use uuid.v4() to randomly generate a Universally Unique Identifier for each item. A UUDI string looks something like this: 2030efbd-a32f-4fcc-8637-7c410896b3e3

Receiving Props in EditableTimerList

EditableTimerList receives the list of timers as a prop, timers. Modify that component to use those props, mapping a EditableTimer for every single item in the timers prop.

Adding state to EditableTimer

In the static version of the app, EditableTimer relied on editFormOpen as a prop to be passed down from the parent. We decided that this state could actually live here in the component itself.

Set the initial value of editFormOpen to false, which means that the form starts off as closed. Also, pass down the id property down the chain.

Timer remains Stateless

If you look at Timer, you'll see that it does not need to be modified. It has been using exclusively props and is so far unaffected by our refactor.

Adding state to ToggleableTimerForm

We know that we'll need to tweak ToggleableTImerForm as we've assigned it some stateful responsibility. We want to have the component manage the state isOpen. Because this state it isolated to this component, let's also add our app's first bit of interactivity while we're here.

Initialize the state to a closed state. Next, define a function that will toggle the state of the form to open. Make sure custom methods for components are written using arrow functions, else you will need to specifically bind this to the component's this in the constructor method.

Edit the necessary parts to reflect the usage of our new state to control the open form state. Let the button invoke the custom function that you've defined just now using the onClick property.

Adding state to TimerForm

Initialize state at the top of the component, giving it two fields, title and project. The entries for both title and project naturally comes from the component's props, but it has to come with a safety mechanism to revert to a blank string if the props are non-existent.

defaultValue only sets the value of the input field for the initial render. Instead of using defaultValue, we can connect our input fields directly to our component's state using value. We could do so by changing the type property to text and giving our state to the value property. With this change, our input fields would be driven by state. Whenever the state properties title or project change, our input fields would be updated to reflect the new value.

We still need a way for the user to modify this state. The input field will start off in-sync with the component's state. But the moment the user makes a modification, the input field will become out-of-sync with the component's state.

We can fix this by using React's onChange attribute for input elements. Like onClick for button or a elements, we can set onChange to a function. Whenever the input field is changed, React will invoke the function specified. Set the onChange attributes on both input fields to handleTitleChange and handleProjectChange functions that we'll define next.

handleTitleChange should change the timer's title to the target value using setState. handleProjectChange should change the timer's project to the target value using setState.

When React invokes the function passed to onChange, it invokes the function with an event object. We call this argument e. The event object includes the updated value of the field under target.value. We update the state to the new value of the input field. Using a combination of state, the value attribute, and the onChange attribute is the canonical method we use to write form elements in React.

With TimerForm refactored, we've finished establishing our stateful data inside our elected components. Our downward data pipeline, props, is assembled.

We're ready — and perhaps a bit eager — to build out interactivity using inverse data flow. But before we do, let's save and reload the app to ensure everything is working. We expect to see new example timers based on the hard-coded data in TimersDashboard. We also expect clicking the "+" button toggles open a form:

Add Inverse Data Flow

Children communicate with parents by calling functions that are handed to them via props. We are going to need inverse data flow in two areas:

- TimerForm needs to propagate create and update events (create while under ToggleableTimerForm and update while under EditableTimer). Both events will eventually reach TimersDashboard.
- Timer has a fair amount of behavior. It needs to handle delete and edit clicks, as well as the start and stop timer logic.

TimerForm

To get a clear idea of what exactly TimerForm will require, we'll start by adding event handlers to it and then work our way backwards up the hierarchy.

TimerForm needs two event handlers:

- When the form is submitted (creating or updating a timer)
- When the "Cancel" button is clicked (closing the form)

TimerForm will receive two functions as props to handle each event. The parent component that uses TimerForm is responsible for providing these functions:

- props.onFormSubmit(): called when the form is submitted
- props.onFormClose(): called when the "Cancel" button is clicked

This empowers the parent component to dictate what the behavior should be when these events occur.

Modify the buttons on TimerForm. We'll specify onClick attributes for each passing down handleSubmit and onFormClose respectively. We are going to define these custom functions in a moment.

handleSubmit calls onFormSubmit, passing in a data object with id, title and project attributes. This means id will be undefined for creates, as no id exists yet. Be sure to prepare a mechanism for the submit text to display Update or Create depending on the existence of the id.

ToggleableTimerForm

Let's chase the submit event from TimerForm as it bubbles up the component hierarchy. First, we'll modify ToggleableTimerForm. We need it to pass down two prop-functions to TimerForm, onFormClose() and onFormSubmit().

Functions are just like any other prop. Of most interest here is handleFormSubmit(). Remember, ToggleableTimerForm is not the manager of timer state. TimerForm has an event it's emitting, in this case the submission of a new timer. ToggleableTimerForm is just a proxy of this message. So, when the form is submitted, it calls its own prop-function props.onFormSubmit(). We'll eventually define this function in TimersDashboard.

handleFormSubmit() accepts the argument timer. Recall that in TimerForm this argument is an object containing the desired timer properties. We just pass that argument along here. After invoking onFormSubmit(), handleFormSubmit() calls setState() to close its form.

TimersDashboard

We've reached the top of the hierarchy, TimersDashboard. As this component will be responsible for the data for the timers, it is here that we will define the logic for handling the events we're capturing down at the leaf components.

The first event we're concerned with is the submission of a form. When this happens, either a new timer is being created or an existing one is being updated. We'll use two separate functions to handle the two distinct events:

- handleCreateFormSubmit() will handle creates and will be the function passed to ToggleableTimerForm
- handleEditFormSubmit will handle updates and will be the function passed to EditableTimerList

Both functions travel down their respective component hierarchies until they reach TimerForm as the prop onFormSubmit(). Let's start with handleCreateFormSubmit, which inserts a new timer into our timer list state.

We create the timer object with helpers.newTimer(). You can peek at the implementation inside of helpers.js. We pass in the object that originated down in TimerForm. This object has title and project properties. helpers.newTimer() returns an object with those title and project properties as well as a generated id.

The next line calls setState(), appending the new timer to our array of timers held under timers. We pass the whole state object to setState().

We've finished wiring up the create timer flow from the form down in TimerForm up to the state managed in TimersDashboard. Save app.js and reload your browser. Toggle open the create form and create some new timers.

Updating Timers

We need to give the same treatment to the update timer flow. However, as you can see in the current state of the app, we haven't yet added the ability for a timer to be edited. So we don't have a way to display an edit form, which will be a prerequisite to submitting one.

To display an edit form, the user clicks on the edit icon on a Timer. This should propagate an event up to EditableTimer and tell it to flip its child component, opening the form.

Adding Editability to Timer

To notify our app that the user wants to edit a timer we need to add an onClick attribute to the span tag of the edit button. We anticipate a prop-function, onEditClick().

Updating EditableTimer

Now we're prepared to update EditableTimer. Again, it will display either the TimerForm (if we're editing) or an individual Timer (if we're not editing). Let's add event handlers for both possible child components. For TimerForm, we want to handle the form being closed or submitted. For Timer, we want to handle the edit icon being pressed. We pass these event handlers down as props.

EditableTimer handles the same events emitted from TimerForm in a very similar manner as ToggleableTimerForm. This makes sense. Both EditableTimer and ToggleableTimerForm are just intermediaries between TimerForm and TimersDashboard. TimersDashboard is the one that defines the submit function handlers and assigns them to a given component tree.

Like ToggleableTimerForm, EditableTimer doesn't do anything with the incoming timer. In handleSubmit(), it just blindly passes this object along to its prop-function onFormSubmit(). It then closes the form with closeForm().

We pass along a new prop to Timer, onEditClick. The behavior for this function is defined in handleEditClick, which modifies the state for EditableTimer, opening the form.

Updating EditableTimerList

Moving up a level, we make an addition to EditableTimerList to send the submit function from TimersDashboard to each EditableTimer. EditableTimerList doesn't need to do anything with this event so again we just pass the function on directly.

Defining onEditFormSubmit() in TimersDashboard

Last step with this pipeline is to define and pass down the submit function for edit forms in TimersDashboard. For creates, we have a function that creates a new timer object with the specified attributes and we append this new object to the end of the timers array in the state.

For updates, we need to hunt through the timers array until we find the timer object that is being updated. As mentioned in the last chapter, the state object cannot be updated directly. We have to use setState(). Therefore, we'll use map() to traverse the array of timer objects. If the timer's id matches that of the form submitted, we'll return a new object that contains the timer with the updated attributes. Otherwise we'll just return the original timer. This new array of timer objects will be passed to setState(). Be sure to pass the edit form submit handler down to EditableTimerList.

Note that we can call map() on this.state.timers from within the JavaScript object we're passing to setState(). This is an often used pattern. The call is evaluated and then the property timers is set to the result.

Inside of the map() function we check if the timer matches the one being updated. If not, we just return the timer. Otherwise, we use Object#assign() to return a new object with the timer's updated attributes.

Remember, it's important here that we treat state as immutable. By creating a new timers object and then using Object#assign() to populate it, we're not modifying any of the objects sitting in state.

As we did with ToggleableTimerForm and handleCreateFormSubmit, we pass down handleEditFormSubmit as the prop onFormSubmit. TimerForm calls this prop, oblivious to the fact that this function is entirely different when it is rendered underneath EditableTimer as opposed to ToggleableTimerForm.

Both of the forms are wired up! Save app.js, reload the page, and try both creating and updating timers. You can also click "Cancel" on an open form to close it.

The Big Picture

With the length exercise, you should know by now how to pass down data through props and inversely propagate events from children to parents to control states.

With this, wire up the delete button using your own knowledge.

Adding Timing Functionality

Create, update, and delete (CRUD) capability is now in place for our timers. The next challenge: making these timers functional.

There are several different ways we can implement a timer system. The simplest approach would be to have a function update the elapsed property on each timer every second. But this is severely limited. What happens when the app is closed? The timer should continue "running."

This is why we've included the timer property runningSince. A timer is initialized with elapsed equal to 0. When a user clicks "Start", we do not increment elapsed. Instead, we just set runningSince to the start time.

We can then use the difference between the start time and the current time to render the time for the user. When the user clicks "Stop", the difference between the start time and the current time is added to elapsed. runningSince is set to null.

Therefore, at any given time, we can derive how long the timer has been running by taking Date.now() - runningSince and adding it to the total accumulated time (elapsed). We'll calculate this inside the Timer component.

For the app to truly feel like a running timer, we want React to constantly perform this operation and re-render the timers. But elapsed and runningSince will not be changing while the timer is running. So the one mechanism we've seen so far to trigger a render() call will not be sufficient. Instead, 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.

Adding a forceUpdate() Interval to Timer

helpers.renderElapsedString() accepts an optional second argument, runningSince. It will add the delta of Date.now() - runningSince to elapsed and use the function millisecondsToHuman() to return a string formatted as HH:MM:SS.

We will establish an interval to run forceUpdate() after the component mounts.

In componentDidMount(), we use the JavaScript function setInterval(). This will invoke the function forceUpdate() once every 50 ms, causing the component to re-render. We set the return of setInterval() to this.forceUpdateInterval.

In componentWillUnmount(), we use clearInterval() to stop the interval this.forceUpdateInterval. componentWillUnmount() is called before a component is removed from the app. This will happen if a timer is deleted. We want to ensure we do not continue calling forceUpdate() after the timer has been removed from the page. React will throw errors.

Try It Out

Save app.js and reload. The first timer should be running.
We've begun to carve out the app's real utility! We need only wire up the start/stop button and our server-less app will be feature complete.

Add Start and Stop Functionality

The action button at the bottom of each timer should display "Start" if the timer is paused and "Stop" if the timer is running. It should also propagate events when clicked, depending on if the timer is being stopped or started.

We could build all of this functionality into Timer. We could have Timer decide to render one HTML snippet or another depending on if it is running. But that would be adding more responsibility and complexity to Timer. Instead, let's make the button its own React component.

Add Timer Action Events to Timer

Let's modify Timer, anticipating a new component called TimerActionButton. This button just needs to know if the timer is running. It also needs to be able to propagate two events, onStartClick() and onStopClick(). These events will eventually need to make it all the way up to TimersDashboard, which can modify runningSince on the timer.

Create handlers to handle the start click and stop click. Then inside render(), declare TimerActionButton at the bottom of the outermost div, passing in the 3 props, timerIsRunning, onStartClick, and onStopClick. timerIsRunning should hold the Boolean for TimerActionButton, returning false if runningSince is null, and true if otherwise. onStartClick and onStopClick should hold the handlers that you've created just now.

Create TimerActionButton

Create the TimerActionButton component:

```
class TimerActionButton extends React.Component {
    render() {
        if (this.props.timerIsRunning) {
            return (
                <div className="ui bottom attached red basic button"
                onClick={this.props.onStopClick}>
                Stop
                </div>
            )
        } else {
            return (
                <div className="ui bottom attached green basic button"
                onClick={this.props.onStartClick}>
                Start
                </div>
            )
        }
    }
}
```

We render one HTML snippet or another based on this.props.timerIsRunning. Run these events up the component hierarchy, all the way up to timersDashboard where we're managing state.

Try It Out

Save app.js, reload, and behold! You can now create, update, and delete timers as well as actually use them to time things.