

Lab 1: TodoList Application in React

Creation of a basic TodoList React application from scratch - all done within Electrode.

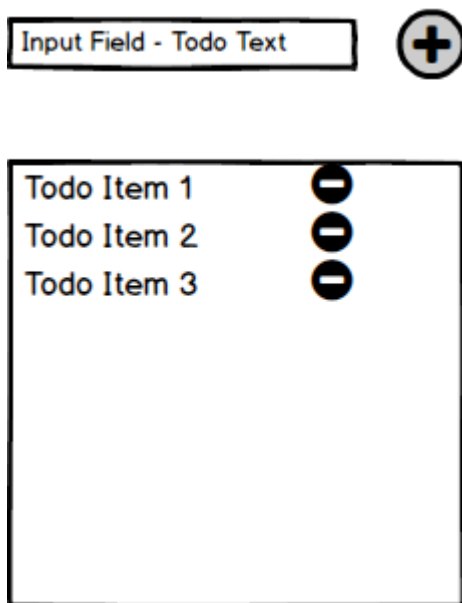
Presentations



- [Presentation](#) 

Lab Overview

In this lab we will be building a simple todo list in React. There is no Redux piece in today's lab. The lab is partially copy-paste but also has some challenging exercises for you. The lab is also a great precursor to the day 2 lab, and will give you a good working reference for how to approach the days 2/3 labs and the assignment, which will be more difficult. This lab also works as a good sanity check of your local dev setup.

Here's a mockup of what our final TodoList might look like:



- **Development Setup Requirements**
 - [Requirements](#) 
- **Required concepts before starting**
 - [ES6 Study Guide](#) 

- [React Study Guide](#)
- **What you will learn**
 - How to use Electrode and how it's built
 - How React components fit together
- **Required**
 - All the Milestones below are required
- **Bonus**
 - None right now

Strategy

We can use our mockup above of the `TodoList` to help us decide how to break the app down into separate React components.

- We'll need a container component that holds everything - including the actual list of todo items. We can call that the **TodoApp**.
- We could say that the input field and add button could be a part of an interactive form component. We can call that **AddTodo**.
- We would also want a component to hold the collection of Todos, and **TodoList** should get that across.
- Each todo item in our list could be it's own component. We'll make a **Todo** component for each entry, and the above list would act as a parent component for these todos.

So, we would have a total of 4 different components:

- The `TodoApp` component, which acts as a container for everything.
- The `AddTodo` component, representing the input field and add button
- The `TodoList` component, representing the list of Todo items
- The `Todo` component, representing each Todo item
- Let's also throw in a very simple `Title` component that will just display a title for the app.

That gives us a total of 5 components.

Functional pieces of our TodoList

Functionally, what would we expect from our Todo list? Here's a list of things our MVP needs:

- `Storage` to save all the todos in our current list (perhaps React state)
- `Add` a new todo to our list
- `Remove` a todo from our list

Before we start creating components, you should make sure you understand the distinction between Function and Class components [↗](#)


Milestone 1: App Setup

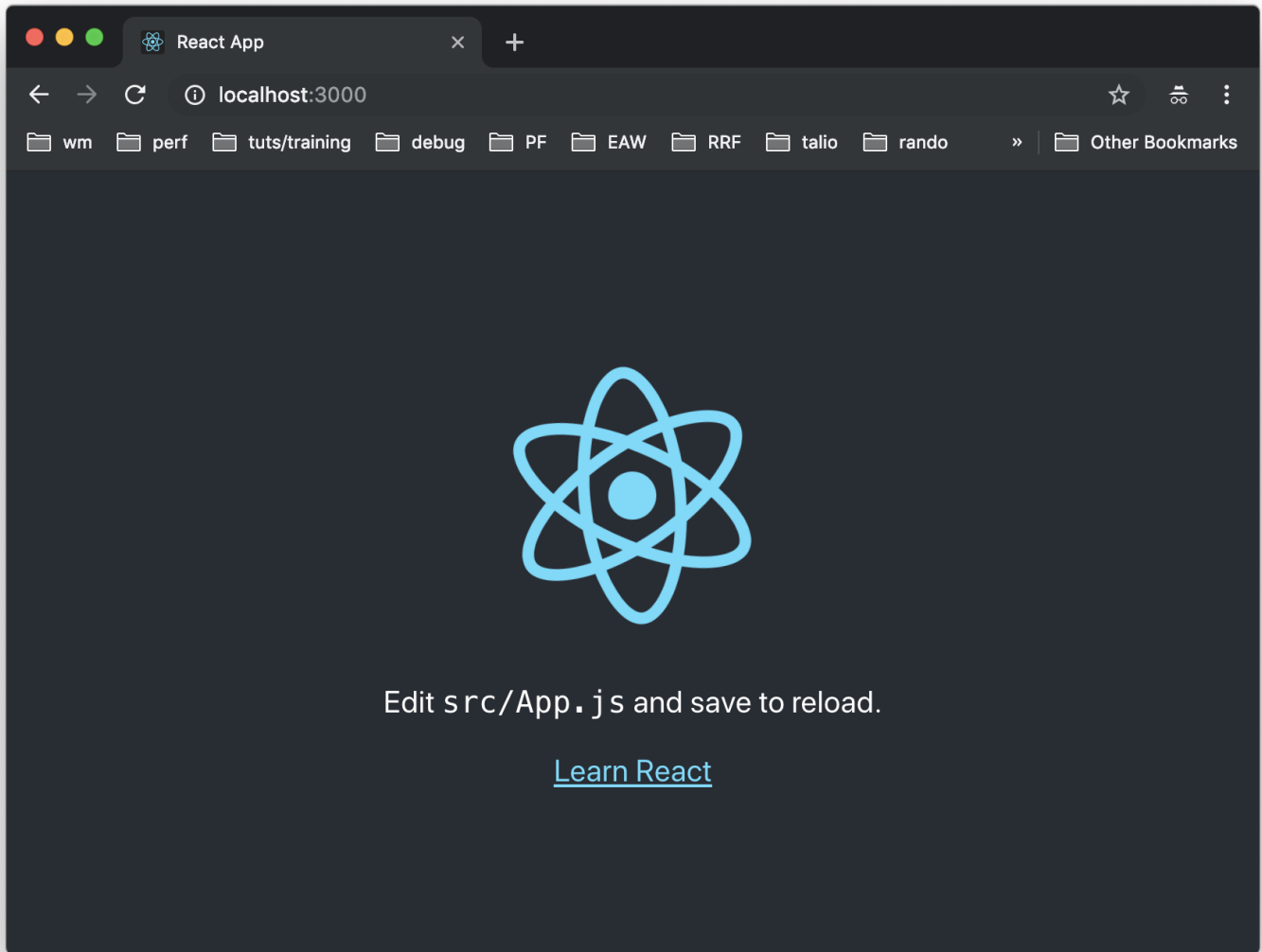
For this lab we'll be using a tool built by Facebook to quickly bootstrap a React app. This will allow us to focus on just React. In later labs we'll introduce Redux, and then the Electrode platform.

Navigate to the location where you want to create the project. Running the command will create a new directory for your project. NPM has bundled the `npx` tool since version 5.2. It runs npm binaries from your project or from the central repository. Run the following command:

```
$ npx create-react-app rrf-todo
```

And npm will check for a local version of the create-react-app binary and if it doesn't find one will pull the binary from the npm repository. Very nice! We now will bootstrap our app with the latest version of the tool and not have to worry about managing a local install.

Once the install completes, change into the project directory and run `npm start`, and if you navigate to `localhost:3000`  you will see a small Demo component that's included as part of the app generator. It will look like this:



You can stop the server for now, and locate the code for that demo component inside the `rrf-todo/src/` directory. Let's clean it up. Delete everything except `index.js`. Next create a file called `todo-app.jsx`. Open `index.js` and change the `App` import statement to

```
import TodoApp from './todo-app.jsx';
```

and delete the `index.css` and `serviceWorker` imports. Finally change the reference to `App` in the `ReactDOM.render` method invocation to `TodoApp`. Now we're ready to build the app.

Milestone 2: Component Boilerplate

Add `TodoApp` component code

Inside your `todo-app.jsx` file copy this code:

```
import React, { Component } from 'react';
import Title from './title';
import AddTodo from './add-todo';
import TodoList from './todo-list';

export default class TodoApp extends Component {
  constructor(props) {
    super(props);

    this.state = {
      todos: [
        { text: " This is a simple todo list app written in React!", id: 1 },
        { text: "Hover over todos and click on the `XX` to delete them!", id: 2 },
        { text: "Add new todos if you like!", id: 3 }
      ]
    };

    this.addTodo = this.addTodo.bind(this);
    this.removeTodo = this.removeTodo.bind(this);
  }

  addTodo(todo) {
    // your code here
  }

  removeTodo(id) {
    // your code here
  }

  render() {
    return (
      <div>
        <Title />
        <AddTodo handleAdd={this.addTodo} />
        <TodoList todos={this.state.todos} handleRemove={this.removeTodo} />
      </div>
    );
  }
}
```

Notice how we bind our methods to **this** . In JavaScript, class methods are not bound by default. See here [🔗](#) for more details.

`TodoApp` is our top level component which will maintain the list of todo items, utilizing React component state. Because we want to store state on the component we chose a Class component.

AddTodo component code

Inside the render method of `TodoApp` , you can see that we use a component called `AddTodo` . Let's add the code for that component. Create a file `add-todo.jsx` and add the following code to it:

```

import React, { Component } from 'react';

class AddTodo extends Component {
  constructor(props) {
    super(props);
    this.state = {
      todoText: ''
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    // your code here
  }

  handleSubmit() {
    // your code here
  }

  render() {
    return (
      <div className="row">
        <input
          type="text"
          value={this.state.todoText}
          placeholder="Add todos here..."
          autoComplete="off"
          onChange={this.handleChange}
          className="column column-60"
        />
        <button className="button button-outline column column-10" onClick={this.handleSubmit}
      > + </button>
      </div>
    );
  }
}

export default AddTodo;

```

This is called a controlled component. The state of the input field is directly managed by React (notice the `value` attribute of the input field is linked to the state of the component).

Add TodoList component code

Next, create the `todo-list.jsx` file:

```
import React from 'react';
import Todo from './todo.jsx';

const TodoList = ({ todos, handleRemove }) => {
  return (
    <div>
      <span><b>Your Todos:</b></span>
      <ul>{
        todos.map(todo =>
          <Todo todo={todo} remove={handleRemove} key={todo.id} />
        )
      }</ul>
    </div>
  );
};

export default TodoList;
```

Because `TodoList` will just **display** a list of `Todos`, it's a good candidate for a presentational component. That is why we created it as a function component using ES6 arrow function syntax.

Add Todo component code

Note that your `TodoList` component imports a `Todos` component. Create `todo.jsx`:

```
import React from 'react';

const Todo = (props) => (
  <li>
    {props.todo.text}
    <span
      onClick={() => {
        // your code here
      }}>
      <b>&nbsp;XX</b>
    </span>
  </li>
);

export default Todo;
```


`Todo` represents a single todo item. It's just a simple presentational component that takes in the `todo` as props and then returns it as part of the markup.


Add Title component

Finally, let's add a simple `Title` component. Create `title.jsx` and create a component that renders our heading. You can use whatever text you like!



Milestone 3: Sanity Check

Make sure your app still loads and doesn't produce any errors.

Milestone 4: Handling User Input

Taking a look at your Todo list application (<http://localhost:3000> ) , you'll notice that the `input` field currently doesn't accept any input!

The reason it doesn't work as expected is because we left the `handleChange` event handler blank inside the `AddTodo` component. This is the first functionality for you to complete – your goal is to get the `input` field to simply accept some input.

You will need to read some more on [Controlled Components](#)  and forms in React to complete that exercise. Note that inside the code we provide the `value` of the input field is set equal to `this.state.value` . Also, make sure that you understand [State in React](#) .

Milestone 5: Add Todos

Once you get the `input` field actually accepting inputs, adding todos still doesn't work. This is because inside the `AddTodo` component, we left the `handleSubmit` method blank.

This is the next step in getting our app to add todos to the list. Notice `handleSubmit` is our callback function for when the `Add` button is clicked.

Some things to keep in mind:

- check for any text in the input field
- give your todo an id, a quick and dirty way for us to generate an ostensibly unique id is to use `new Date().getTime()` which returns the Unix epoch time
- remember what the component is receiving as props
- after adding the todo, what should the input look like?

Milestone 6: Remove Todos

Now that we have the add functionality working, we need to implement the remove functionality so that a user can easily remove a todo item from the list by clicking on the `XX` next to each todo item.

In our boilerplate, we have the `TodoApp` component, which is a parent for the `TodoList` component. From `TodoApp` we pass a function called `removeTodo` into the `TodoList` child component using `props` - it's passed with the name `handleRemove`. Then, `TodoList` passes `handleRemove` to the `Todo` component. Note that `TodoList` doesn't do anything with the `handleRemove` callback - it's really just an intermediary in the passage of the function. So, `removeTodo` gets passed from the `TodoApp` grandparent component down to its grandchild - `Todo`.

Understanding the design

Why was it designed in this way? Why are we passing the callback through props two components down in the hierarchy?

Well, our `TodoApp` component is our stateful component which maintains the list of todo items utilizing React component state. The list of todos must be kept at the `TodoApp` level because both the `AddTodo` and `TodoList` components depend on that data. Since sibling components can't share data we must promote it to a common parent component. Because `TodoApp` contains the data, it is necessary that the handlers for operating on that data—namely adding and deleting todo items—are defined on this component too, where the data they manipulate is in scope.

Challenge details

Inside the `removeTodo` function, you will need to remove a todo assuming that you are passed in the todo's `id`. Be sure to look at the data currently being stored in state to grab the most current list of todos. Create an updated list that excludes the todo we want to delete. Then use `setState` to reset the state to your updated list.

You'll need to appropriately call `remove` from your `Todo` component's `onClick` utilizing props and be sure to pass in the appropriate data.

Milestone 7: Destructuring our props

If you take a look at your `Todo` component you'll notice we just have a simple functional component that takes in some `props` and then returns some markup using the `props` data:

```
// our abbreviated Todo component:
const Todo = props => {
  return (
    //use props here to return some JSX...
    ...
  );
};
```

We could also use ES6 destructuring with our props. Here's an example of what that would look like:

```
//curly braces for destructuring
const Todo = ({prop1, prop2}) => {
  return <h1>Hello, {prop1} {prop2}</h1>;
};
```

Now, your challenge is to implement ES6 destructuring [↗](#) in the `Todo` component to destructure the props passed in. That way, your code will be cleaner, and you won't have to call `props.remove` or `props.todo`. Instead, you can just call `remove` and `todo` directly.

Milestone 8: Styling our components

Our app works great, but it doesn't look great. Let's use CSS Modules [↗](#) to make our interface a little bit easier to use and understand. We can use a CSS framework Milligram (["https://milligram.io/"](https://milligram.io/)), along with some of our own styling rules to prettify our to-do app.

First install Milligram

```
$ npm install milligram
```

Next create a new file `src/todos.module.css` and fill it with some very basic, but noticeable CSS.

```
.addTodos {
  background: red;
}
```

Import Milligram into `index.js`.

```
import 'milligram';
```

Then in `add-todo.jsx` import the css module.

```
import styles from './todos.module.css';
```

You can attach your custom styles to a component within the JSX by specifying the `className` you'd like to assign to that component. Find the button you're using to add a todo, and add the following JSX after your `onClick` handler within the opening tag.

```
className={styles['addTodos']}
```

Refresh the page and confirm that your add todo button is now a horrific red. The CSS Module works, but nobody is going to want to click that!

Milligram will be available as global styles so you can apply class names to your jsx to use them. For example add the following to the button:





```
className="button button-outline column column-10"
```

Now your button should look a lot more button-y.

Take this time to mix in some basic Milligram boilerplate CSS with some of your own custom CSS. Here are a couple of suggestions:

- Make the 'add todos' textbox a bit more prominent. This should be a focal point of your app.
- Make the 'delete todo' button more intimidating. Users should be fully aware of the consequences of clicking it just by looking at it.
- Try to use some of the extra white space on the screen.

Resources

- [Electrode Public Documentation](#) 
- [ES6 Study Guide](#) 
- [React Study Guide](#) 
- [Milligram CSS Homepage](#) 
- [CSS Modules Documentation](#) 