Clone the workshop repo

https://github.com/etiennedi/react-workshop-01

Let’s go! …Tips for Lesson 01

- any html element in React will accept a style prop
  - styles are an object, such as { width: ‘100%’ }
- Remember the JSX Syntax for Components and Props:
  - <Component propName={value} />

Stateless functional vs class-based components

You already know this syntax:

```
const FirstComponent = () => (
  <div>
    <h1>Hello world!</h1>
  </div>
);
```

- it's fast
- it's short
- it's stateless
- **always prefer this option!**
  - in fact, there is a common linter rule to check for this

This will do the same:

```
import React, { Component } from 'react';

class SecondComponent extends Component {
  render() {
    return (
      <div>
        <h1>Hello world!</h1>
      </div>
    );
  }
}
```

When to use class-based components

- required if the component must be stateful (**try to avoid stateful components!**).
- required when you want to access lifecycle methods
- needs a **render()** method
  - identical to a FSC's return

```
import React, { Component } from 'react';

class SecondComponent extends Component {
  render() {
    return (
      <div>
        <h1>Hello world!</h1>
      </div>
    );
  }
}
```

How to introduce state into a component

- **Important**: We haven't talked about state management yet. You will find out soon that there are better ways to manage state (Redux).

- **Step 1**: create an initial state on the constructor using this.state.

- **this.state is immutable**. Outside of the constructor you cannot modify it like this.state.money = 120;

```
constructor(props) {
  super(props);

  this.state = {
    money: 100,
  }
}
```

Step2: Updating state

- Remember: **React is declarative**. If you were to modify state directly (imperatively), how would React know to re-render changed components.
- Instead you must call **this.setState()**
  - it will update the state (by creating a new state)
  - and trigger a new render
  - any component which depends on this state will update
  - because setState triggers a re-render, you cannot call it from inside a render method … unless you like infinite loops.

```
increaseMoney() {
  this.setState({
    money: this.state.money + 10,
  })
}
```

Let’s go! …Tips for Lesson 02

- class methods called from DOM events aren’t bound to the class
  - ‘this’ does not refer to the class instance by default
  - tip: Use the class constructor to bind methods to the class.

- constructors in react components
  - need to accept props as an argument
  - need to call the parent constructor with the props (i.e. super(props))

- To output JS variables inside of JSX use single curly braces, i.e. { foo }
- Tip: A click handler can just be a class method.

Validating props

- React offers **propTypes** for props to make developing components easier

- this differs from standard types (such as TypeScript, Flow) because it knows about React Types, such as element or DOM node.

```
11  import { PropTypes } from 'react';
12
13  const propTypes = {
14    foo: PropTypes.string,
15  }
16
17  FirstComponent.propTypes = propTypes;
18
```

Default Props

- To avoid unnecessary conditionals, you can specify default props

- If a prop is not specified, the prop's value will be what you set in defaultProps

```
const defaultProps = {
  foo: 'I am a default string'
}

FirstComponent.defaultProps = defaultProps;
```

Testing React is easy.

Let’s start with unit tests.

Karma/Mocha vs Jest

- **Mocha**
  - most used testing library
  - works good, but not state-of-the-art
  - common describe / it syntax
- **Karma**
  - automatic tests runner and watcher for Mocha.
- **Jest**
  - backed by Facebook (same as React)
  - super-fast (multiple worker processes)
  - great mocking/stubbing abilities (no need for sinon.js, …)
  - amazing watcher abilities (e.g. only files changed since last commit)
  - same basic syntax as mocha (describe / it)
  - Jasmine-based assertion syntax
  - for list of all assertions/expectations see: https://facebook.github.io/jest/docs/expect.html

We’ll go with Jest.

How can we render react components in tests?

- Enzyme (by airbnb) offers amazing rendering abilities

```
const component = shallow(
    <LikeButton />
)
```

- we can now call all kinds of API methods on component.

Shallow rendering vs. mounting

- **Shallow** rendering
  - ideal for **unit tests**
  - only renders the current component
  - child-components are placed in code, but not rendered themselves
  - no full DOM support
  - no lifecycle hooks called
- **Mounting**
  - ideal for **integration tests**
  - mounts the entire component subtree
  - renders all child components
  - DOM support
  - lifecycle hooks called

Enzyme API samples

```
1  component.props() // returns object of all props
2
3  component.prop('prop') // returns a single prop
4
5  component.simulalte('change') // simulates a change event
6
7  component.setProps({someProp: 'value'}) // update the someProp with 'value'
8
9  component.setState({someState: 'value'}) // same as setProps, but for state
10
11 // and many more!
```

See full API at http://airbnb.io/enzyme/docs/api/shallow.html

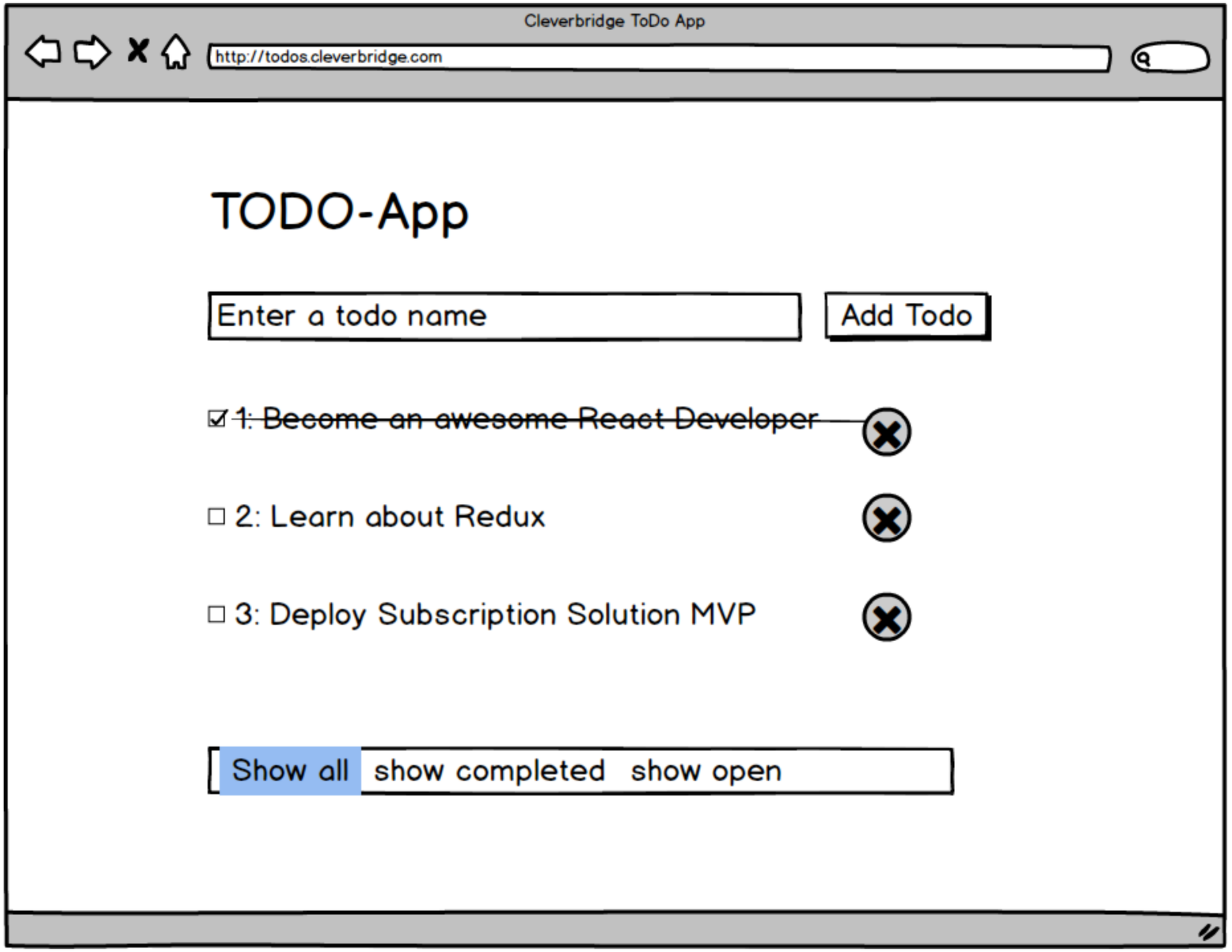
A sample test.

```
// SampleComponent.spec.js

import React from 'react';
import { shallow } from 'enzyme';

import SampleComponent from './SampleComponent';

describe('sample component', ()=> {
    it('should be awesome', ()=> {
        const component = shallow(
            <SampleComponent awesomeness={9001} />
        );

        expect(component.prop('awesomeness')).toBeGreaterThan(9000)
    })
})
```

```
// SampleComponent.js

import React from 'react';

const SampleComponent = ({awesomeness}) => {
    return (
        <div>My Awesomeness is {awesomeness}</div>
    )
}

export default SampleComponent;
```

Let’s build an app.

“Every time you build a todo app, a puppy dies”


Cleverbridge ToDo App
http://todos.cleverbridge.com
TODO-App
Enter a todo name
Add Todo
1: Become an awesome React Developer
2: Learn about Redux
3: Deploy Subscription Solution MVP
Show all
show completed
show open

Forms in React

- **Three ways to build a form:**
  - **native html form** with submit event
    - very simple, great for our example
  - **controlled React** components
    - added functionality, such as real-time updates
  - **“managed forms”**, such as Redux Form
    - huge feature set (explicit form state, sync/async validation, dirty state, …)

```
<form onSubmit={/*submitHandler*/}
    <input
        type="text"
        name="some-field"
    />
    <button>Submit</button>
</form>
```

Let’s use native forms for now.

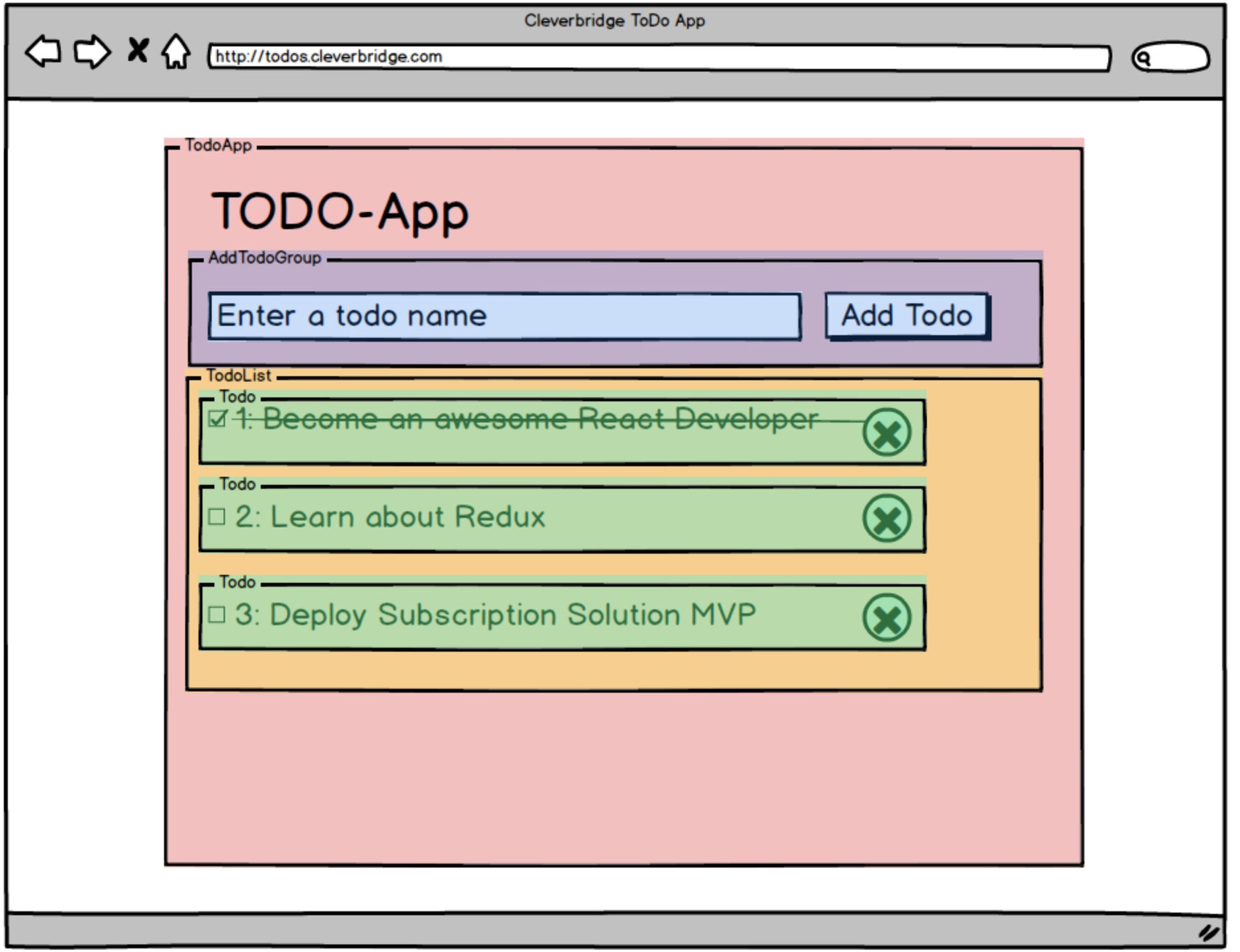
- html form element has an onSubmit event
- native DOM event
- you can access each form element from within the event
- don’t forget to prevent default action (page reload)

```
<form onSubmit={
    (e)=>{
        e.preventDefault();
        console.log(e.target.someField.value);
    }
} >
    <input
        type="text"
        name="someField"
    />
    <button>Submit</button>
</form>
```

How to display multiple items

- split into two components
  - one ‘list' component
  - one ‘item’ component
- React relies on functional JS
  - use Array.map() to access each array item.
  - ES6 syntax makes this quick and easy.

```
1   import SingleItem from './SingleItem';
2
3   const ItemsList = ({items}) => (
4       <div>
5           {
6               items.map((item, index) => (
7                   <SingleItem
8                       item={item}
9                       key={index}
10                  />
11              ))
12          }
13      </div>
14  );
15  export default ItemsList;
16
```

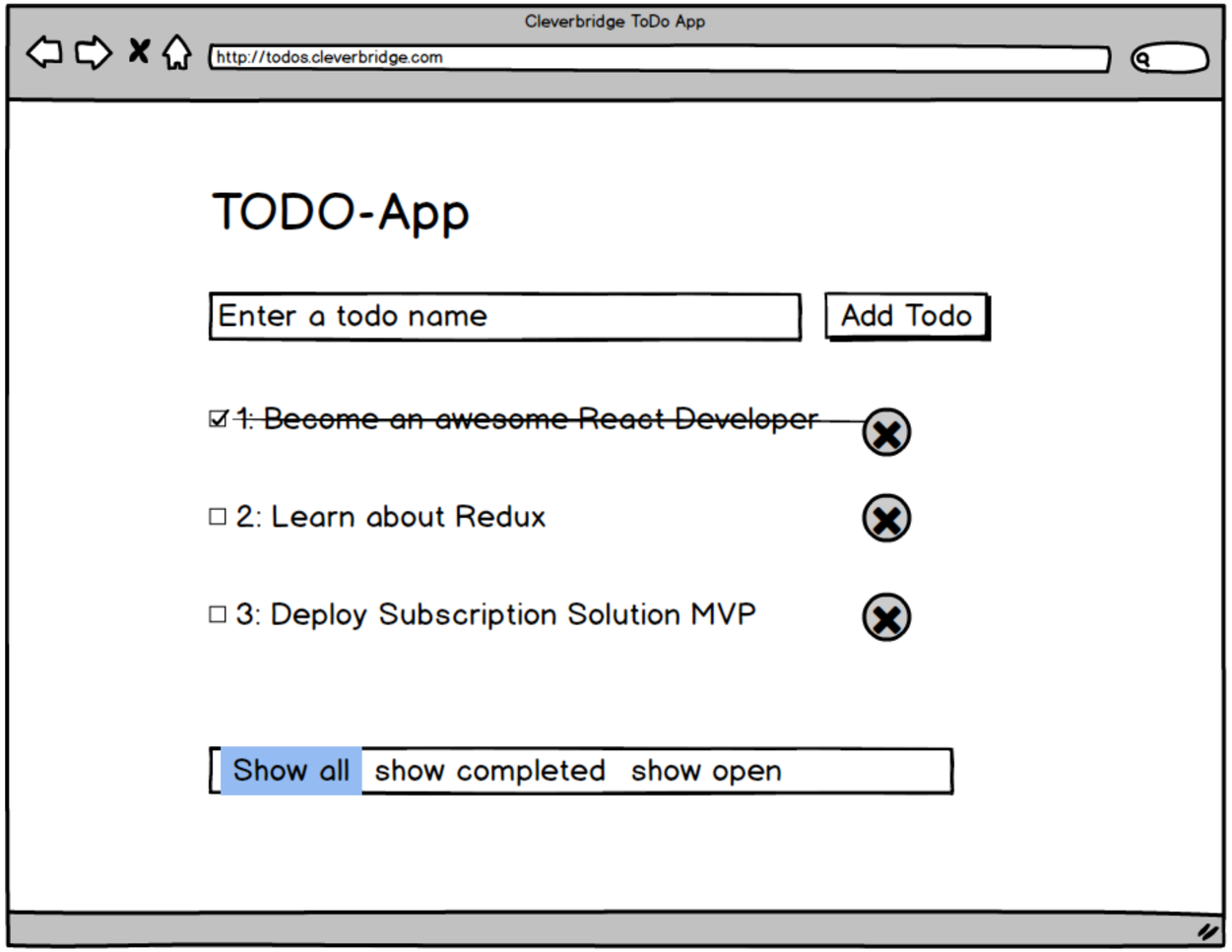

Cleverbridge ToDo App
http://todos.cleverbridge.com
TodoApp
TODO-App
AddTodoGroup
Enter a todo name
Add Todo
TodoList
Todo
1: Become an awesome React Developer
Todo
2: Learn about Redux
Todo
3: Deploy Subscription Solution MVP

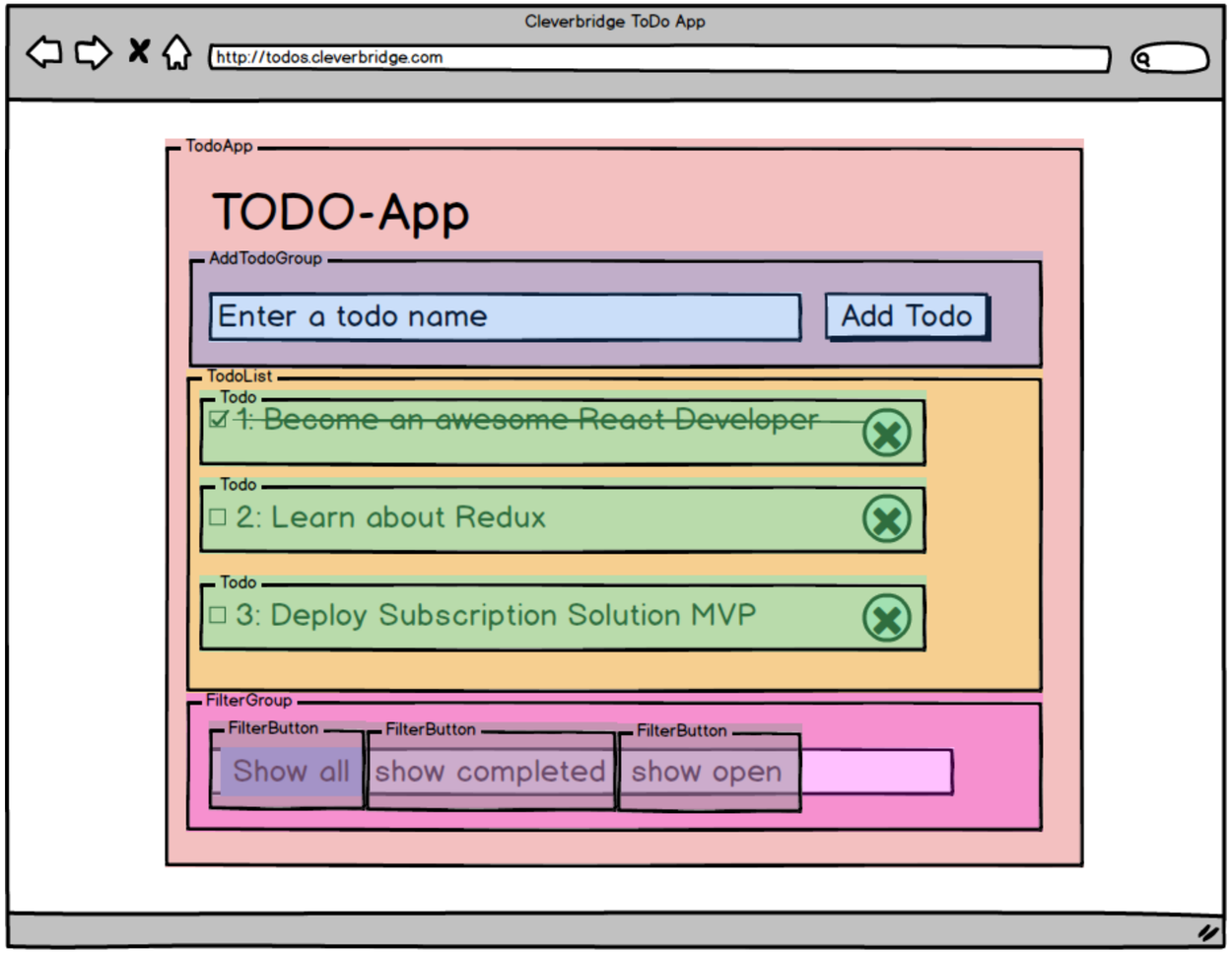
Let’s go … Tips for Lesson 05:

- To use the ‘for’ attribute on the ‘label’ element, you must use ‘htmlFor’ in React. (You will need this for the checkbox.)
- To use the ‘class’ attribute on any html element, you must use ‘className’ in React.
- Use functional JS methods, such .map(), .filter(), …
- Don’t worry about assigning IDs to your todos. Simply reference them by their array index. We’ll show a better way soon.
- **Reminder: State is immutable.** Do not (accidentally) mutate state directly.

We can do better than that.

Let’s improve our app.


Cleverbridge ToDo App
http://todos.cleverbridge.com
TODO-App
Enter a todo name
Add Todo
☑ 1: Become an awesome React Developer
☐ 2: Learn about Redux
☐ 3: Deploy Subscription Solution MVP
Show all
show completed
show open


Cleverbridge ToDo App
http://todos.cleverbridge.com
TodoApp
TODO-App
AddTodoGroup
Enter a todo name
Add Todo
TodoList
Todo
☑ 1: Become an awesome React Developer
Todo
☐ 2: Learn about Redux
Todo
☐ 3: Deploy Subscription Solution MVP
FilterGroup
FilterButton
Show all
FilterButton
show completed
FilterButton
show open

Referencing the array index won't work anymore.

- We can add an id property.
- It's easiest to turn the array into an object.
- We can now make use of a lot of ES7 spread operator magic.
- Array function can still be used with Object.keys()

```
// old - harder to access
const todos = [
    {
        title: '',
        completed: '',
    }
]

// new easier to access
const todos = {
    '238049203423' : {
        id: '238049203423',
        title: '',
        completed: '',
    }
}

```

Three dots change everything. (ES7 Spread Operator)

- Basic usage:
  - Spread the content of one object into another.
- Advantages:
  - super simple syntax
  - impossible to mutate object (always creates copies)
  - very clean code possible on large state updates (will become more relevant when we talk about Redux)

```
1  const obj1 = { apples: 1, oranges: 2 }
2
3  const obj2 = { bananas: 3 }
4
5  const fruits = { ...obj1, ...obj2 }
6
7  // result
8  // { apples: 1, oranges: 2, bananas: 3 }
9
```

(ES6) Object destructuring is the same in reverse.

- you can destructure any object into its parts using this syntax.
- you can even desctructure only some parts.
  - by using the spread operator (…) inside object destructuring you can collect all not explicitly mentioned properties.
  - *Hint: This is an excellent way to "delete" an object property without mutating the original object or having to make a manual copy. Think of "otherFruits" as "fruitsWithoutApples".*

```
10 const fruits = {
11     apples: 1,
12     oranges: 2,
13     bananas: 3,
14 };
15
16 const { apples, oranges, bananas } = fruits;
17
18 // apples === 1
19 // oranges === 2
20 // bananas === 3
```

```
22 const { apples, ...otherFruits } = fruits;
23
24 // apples === 1
25 // otherFruits == { oranges: 2, bananas: 3 }
26
```

Let’s **spread** some magic into our Todo app.

Pun intended.

Let’s go. Tips for Lesson 6.

- You don’t need to update the data state (this.state.todos) when changing filters.
  - simply add a filter state (such as this.state.filter === ‘all’, etc.)
  - instead of passing this.state.todos to the TodosList component, you can pass a method call which returns a sorted list. (This is the declarative way - “*Don’t put in the state, what you can calculate from the state*”)
- To generate (mostly) unique IDs, you could use a timestamp, such as Date.now().toString(). (That’s how we did it.)

Alternative Solution with controlled component.

The default solution in branch “lesson-06” uses the native html form submit event, as we discussed in the slides. To see an alternative with a “controlled component”, check out the branch “lesson-06-with-controlled-component”

Redux

Redux is a **predictable state** container for JavaScript apps.

Redux concepts

- State is centralised in a single “store”.
- State is immutable.
- A new state can be generated with pure functions called “reducers”.
- Anyone can subscribe to state changes (push vs. pull)
- Most of redux “usage” is pure javascript.
  - Rather a concept than a library.

Photo credit: erikras (https://github.com/erikras/ducks-modular-redux/blob/master/migrate.jpg) and Airwolfhound (https://www.flickr.com/photos/24874528@N04/3453886876/)

Updating the state with actions

- To update the state you have to dispatch an “action”
  - an action is very similar to an event
  - each action must have a “type” property
  - all other properties are up to you

Sample action:

```
const action = {
    type: 'ADD_TODO',
    text: 'I am the sample text for an ADD_TODO action',
}
```

Sample State generated by redux:

```
const state = {
    todos: [
        {
            text: 'I am the sample text for an ADD_TODO action',
            completed: false,
        }
    ],
};
```

How do we create the state from actions?

- The function to create the new state from these arguments is called a **reducer:**
  - the old state
  - an action describing the desired change
- Reducers must be pure functions. (Predictable and side-effect-free).
- All reducers will be called with all actions, but not all reducers must react to an action.
- The reducer owns a part of the state
  - and is responsible to create its initial state
  - to create any new state for this part.

The “todos” reducer

```
1  function todos(state = [], action) {
2      switch (action.type) {
3          case 'ADD_TODO':
4              return [
5                  ...state,
6                  { text: action.text, completed: false }
7              ];
8          case 'TOGGLE_TODO':
9          /*
10           put logic here to create new state with
11           todo toggled
12         */
13         default:
14             return state;
15     }
16 }
```

Redux data flow

Redux is independent of React. The view can be any view library.

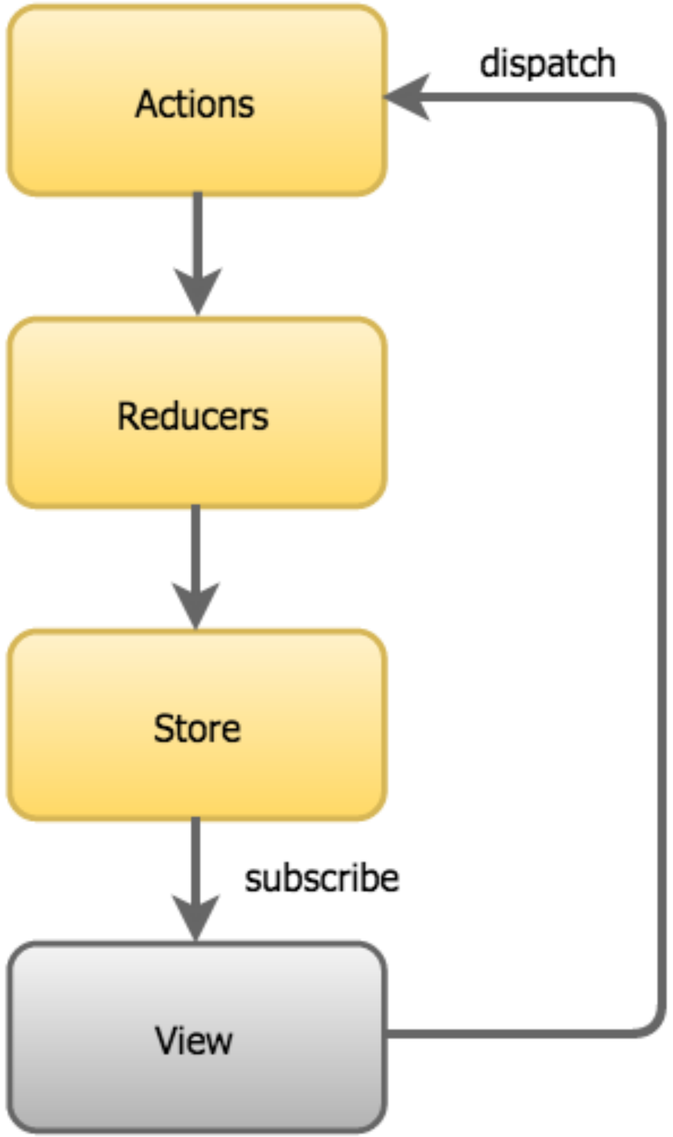
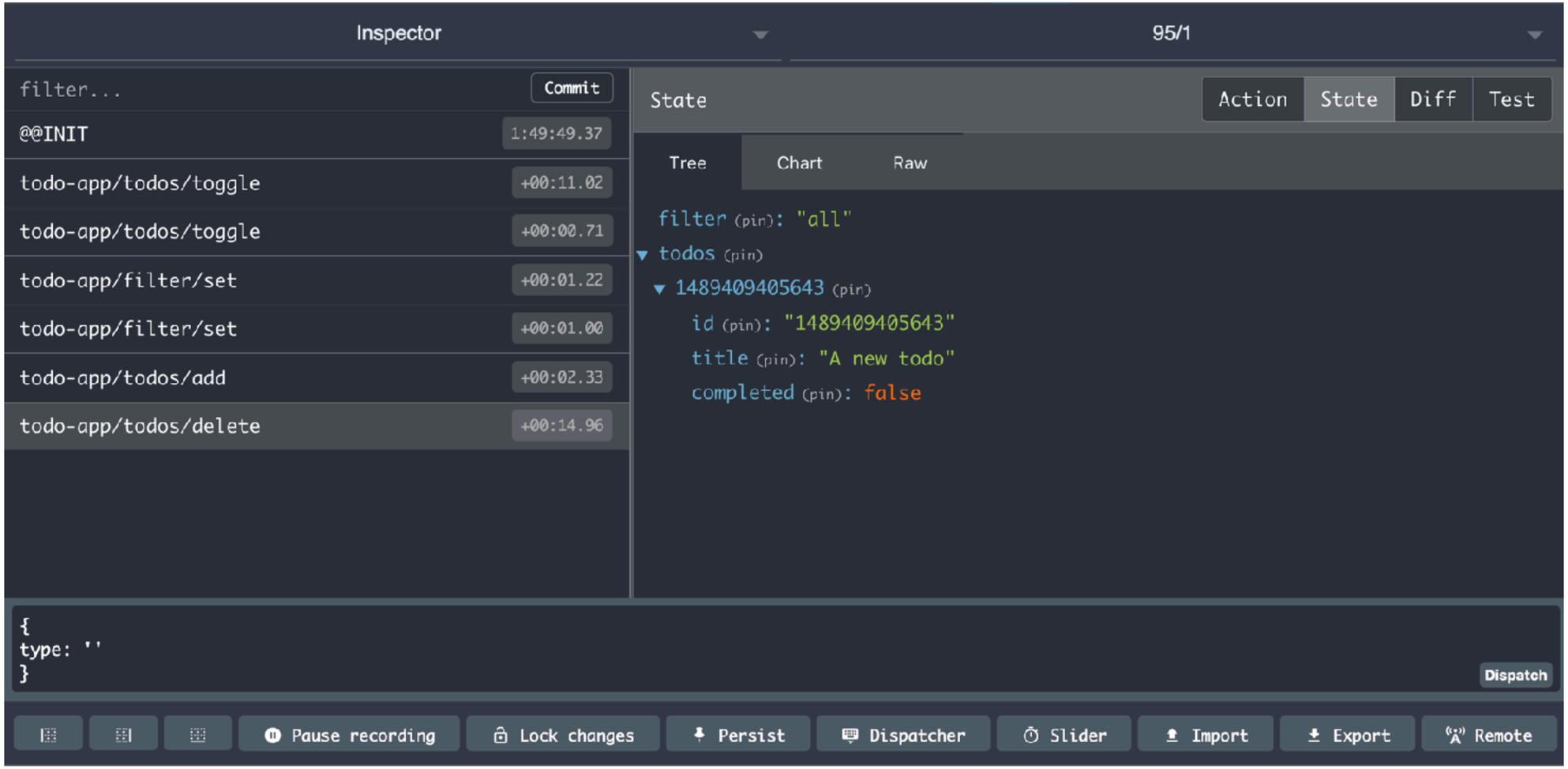


Image source:
https://medium.com/google-developer-experts/angular-2-introduction-to-redux-1cf18af27e6e#.vnerjmvnb

Organizing redux

https://github.com/erikras/ducks-modular-redux



Easy debugging with Redux Dev Tools Chrome extension

https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

Combining Redux and React

- Each component which wants to **read the state** or **dispatch actions** must be "**connected**".
  - Connected Components are sometimes called "containers" which can be a bit misleading.
- **connect()** is a **higher-order component** from **react-redux**
- it takes two arguments to configure the part of the state it is connected to.

Standard Component:

```
1 const SomeComponent = () => {
2     /* return standard React component logic here */
3 }
4
5 export default SomeComponent;
```

Connected Component:

```
1  import { connect } from 'react-redux';
2
3  const SomeComponent = () => {
4      /* return standard React component logic here */
5  }
6
7  export default connect(
8      /* connect configuration goes here */
9  )(SomeComponent);
10
11
```

Configuring connect()

- Connect uses currying/partial application for its configuration
  - this basically just means that you have to call it twice
- The first call takes to arguments:
  - mapStateToProps
  - mapDispatchToProps

```
6   const mapStateToProps = (state) => ({
7       /* see next slide */
8   })
9
10  const mapDispatchToProps = (dispatch) => ({
11      /* see next slide */
12  })
13
14  export default connect(
15      mapStateToProps,
16      mapDispatchToProps
17  )(SomeComponent);
```

mapStateToProps

- this function has the **entire redux state** available
- it **returns an object**
  - each object property is available to the react component as a prop.
- All you have to do is "**map the state to the props**".

```
6  const mapStateToProps = (state) => ({
7      products: state.products
8  })
9
```

mapDispatchToProps

- redux provides "**bindActionCreators**" which makes sure that action creators are always bound to the dispatch method
  - all you need to do to dispatch an action, is call the bound action creator function.
- **mapDispatchToProps** maps all bound action creator to the components props (similar to mapStateToProps).

```
10  import { bindActionCreators } from 'redux';
11  import {
12      addProduct as addProductActionCreator,
13      deleteProduct as deleteProductActionCreator
14  } from './ducks/product';
15
16  const mapDispatchToProps = (dispatch) => {
17      return bindActionCreators({
18          addProduct: addProductActionCreator,
19          deleteProduct: deleteProductActionCreator,
20          /* ... */
21      }, dispatch)
22  }
```

Instead of tips … let’s do some coding together:

- lets assume we have
  - a products reducer
  - with an addProduct action creator
- We now want to create a connected component.

Tipps for Lesson 07

- Redux docs: http://redux.js.org
- Redux ducks pattern: https://github.com/erikras/ducks-modular-redux

A nicer solution with flow and linting.

The standard solution for lesson 07 can be found in the branch “lesson-07”, a nicer solution is in the branch “lesson-07-with-flow-and-lint”

We can now use awesome libraries to make our life easier.

Let **redux-form** handle all form logic from now on.

“The best way to manage your form state in Redux.”

- Redux-Form transfers all your form state into the redux state
- It has its own reducer
- All form manipulations (key events, submit events, etc.) are dispatched as redux actions.
- Validation, form status (dirty vs. pristine, touched vs untouched, …) and normalizing is now super-easy

reduxForm is a higher-order component, similar to connect():

```
32  export default reduxForm(
33    {
34      form: 'yourForm',
35    },
36  )(YourComponent);
```

How to use redux-form (1/2)

- Instead of rendering input fields directly, you render redux-form's "**Field**" component.
- It takes a "**component**" prop to specify the rendering component
  - can be a native html "component" such as input
  - can be your own presentational component.
  - it passes all handlers (onChange, onFocus, onBlur, …) to the DOM component.
- It must have a unique "**name**" prop, which acts as an internal id in redux-form.
- any other props will be passed down to the render component (e.g. type for input)

```
39  export const ContactForm = ({ handleSubmit }) => (
40    <form onSubmit={handleSubmit}>
41      <Field
42        name="name"
43        component="input"
44      />
45    </form>
46  );
47
48
49  export default reduxForm(
50    {
51      form: 'contact',
52    },
53  )(ContactForm);
54
```

How to use redux-form (2/2)

- Because reduxForm() is **a Higher-Order-Component** (HOC), it can inject props into your original component
  - the most important prop is the **handleSubmit** prop
  - you must make sure your form calls it on submit.
  - even if you have your own submit function, the built-in redux-form handleSubmit must always be called!

```
39  export const ContactForm = ({ handleSubmit }) => (
40    <form onSubmit={handleSubmit}>
41      <Field
42        name="name"
43        component="input"
44      />
45    </form>
46  );
47
48
49  export default reduxForm(
50    {
51      form: 'contact',
52    },
53  )(ContactForm);
54
```

How to configure redux-form

- The **reduxForm** function takes a configuration object in it's first call
  - you must specify a unique "**form**" identifier as a string
  - you can add a lot of configuration (see http://redux-form.com/6.5.0/docs/api/ReduxForm.md/)
  - most likely, you will want to use an "**onSubmit**" prop

```
48
49 export default reduxForm(
50   {
51     form: 'contact',
52     onSubmit: (values) => { /* do something with the values */}
53   },
54 )(ContactForm);
55
```

“All of these configuration options may be passed into **`reduxForm()`** at "design time" or passed in as props to your component at runtime.”

design time

```
// ContactForm.jsx
export default reduxForm(
  {
    form: 'contact',
    onSubmit: (values) => { /* do something with the values */}
  },
)(ContactForm);
```

```
// OtherComponentWhichusesTheForm.jsx
const OtherComponentWhichusesTheForm = () => (
    <ContactForm />
)

export default OtherComponentWhichusesTheForm;
```

runtime

```
// ContactForm.jsx
export default reduxForm()(ContactForm);
```

```
// OtherComponentWhichusesTheForm.jsx
const OtherComponentWhichusesTheForm = () => (
    <ContactForm
        name="contact"
        onSubmit={ values => { /* do something with the values */ }}
    />
)

export default OtherComponentWhichusesTheForm;
```

Let’s go. Tips for Lesson 8.

- You don’t need your own rendering components just yet, simply pass **native html elements** (as strings) to the “**component**” prop of the “**Field**” component.

- You can use the redux-devtools extension, to see what redux-form does in the background

  - This will give you a first impression, why redux-form can be a huge timesaver. (Hint: There is a lot happening in the background).

So far, all we have is a form.

Let’s use some cool redux-form features, to make it a much better form.

Validating fields has become much easer since redux-form v6.3.0

- Before you could only validate the entire form, now you can validate single fields
- The “**validate**” props takes a validation function.
  - it is called with the field value
  - it must return undefined if the validation passes
  - it must return a string containing the error message if the validation fails

```
54  const checkUsername = (name = '') => {
55      if (name.toLowerCase() === 'John ForbiddenName') {
56          return 'You have entered the forbidden name';
57      }
58      return undefined;
59  }
60
61  const SomeForm = () => (
62      <Field
63          name="username"
64          component="input"
65          validate={checkUsername}
66      />
67  )
```

How will we know if a field has an error?

- So far we've been using standard html components as render components in "**Field**"
- If we want to display errors we must use our own render component
- Each component passed to the component prop in "Field" will receive the following props:
  - input
  - meta

```
props.input === {
    /* all change handlers and properties the html input needs,
     * for example: */
    onChange,
    value,
    /* ... */
}

props.meta === {
    /* all kinds of meta information we can use to determine the
     * status of the current field, for example: */
    dirty,
    touched,
    error,
    /* ... */
}
```

What can we do with input and meta?

- “input” can be passed to the native html “input” element
  - it contains everything React needs to turn the component into a controlled component
  - meta information can be consumed by your own component
    - e.g. render a red box if meta.error !== undefined

```
props.input === {
    /* all change handlers and properties the html input needs,
     * for example: */
    onChange,
    value,
    /* ... */
}

props.meta === {
    /* all kinds of meta information we can use to determine the
     * status of the current field, for example: */
    dirty,
    touched,
    error,
    /* ... */
}
```

Let’s go. Tips for Lesson 9.

- You will need to either write your own validators (or use any of the validation libraries available, such as Joi,…)

- To make sure you actually see the error you must replace the default html rendering component (such as “input”) with your own component.

  - Your own component must of course also contain an input element

  - You should pass the input prop (available through redux-from) to the input field. You can use the spread operator for this.

  - Everything available in the “input” and “meta” props is listed under “Props” (about half way down) on http://redux-form.com/6.5.0/docs/api/Field.md/

Routing in Single-Page-Applications

Single-Page is misleading.

Server-Side-Routing vs. Client-Side-Routing

Server-Side

- part of web server (express, nginx, apache, …)
- a route change is a “hard” page reload in browser
  - slow
  - very noticeable (blank page in between)

Client-Side

- part of (React) app
- Server-Side Router must pass URL through to client (History API Fallback)
- a route change does not reload the page
  - extremely fast
  - hardly noticeable
  - animatable

React Router v4

- https://reacttraining.com/react-router/web/
- Declarative Routing with React components
- Easy to use, example says all.
- Exact matching vs. part-matching vs. strict matching.

```
1   import React from 'react'
2   import {
3     BrowserRouter as Router,
4     Route,
5     Link
6   } from 'react-router-dom'
7
8   const BasicExample = () => (
9     <Router>
10      <div>
11        <ul>
12          <li><Link to="/">Home</Link></li>
13          <li><Link to="/about">About</Link></li>
14          <li><Link to="/topics">Topics</Link></li>
15        </ul>
16
17        <hr/>
18
19        <Route exact path="/" component={Home}/>
20        <Route path="/about" component={About}/>
21        <Route path="/topics" component={Topics}/>
22      </div>
23    </Router>
24  )
```

What happens when you hit refresh on specific route?

- By default, the server-side router tries to match the URL and will most likely not find anything
- Instead we must tell the server, to forward all routes to the client
  - In webpack-dev-server config we can set historyApiFallback to true
  - In express (dist environment) we can install a historyApiFallback package.

```
12    devServer: {
13      contentBase: path.join(__dirname, 'dist'),
14      compress: true,
15      port: 9000,
16      historyApiFallback: true,
17    },
```

What’s necessary for production readyness?

- A production build process (simply use webpack -p) to build the app into the ‘dist’ folder
  - There are plenty of webpack plugins for optimising your bundles
- We should generate an HTML entry point template dynamically. Use https://github.com/jantimon/html-webpack-plugin#writing-your-own-templates
  - Make sure your app root div exists
- A Dockerfile (node basis image, add files, start an express server)
  - If Client-Side-Routing is desired you need to install/use API History Fallback

API calls are asynchronous side effects.

And we should treat them that way.

Three standard “side effect libraries”

- Redux Thunk
- Redux Saga
- Redux Observables

Redux Thunk

Redux-Saga

Thunk is simple, but limited

- action creators don't have to be synchronous anymore
  - they can handle promises now
  - instead of returning an action, return a function which returns an action (at some point)
- this way they can also do API calls
- but … is this really the right place to do API calls?

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

function increment() {
  return {
    type: INCREMENT_COUNTER
  };
}

function incrementAsync() {
  return dispatch => {
    setTimeout(() => {
      // Yay! Can invoke sync or async actions with `dispatch`
      dispatch(increment());
    }, 1000);
  };
}
```

Saga is basically event-management.

- UI dispatched event
- Saga listens to event
- Saga does an api call
- Saga dispatches a new event with a payload
- Reducer can create a new state
- Everybody is happy

```
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from '...'

// worker Saga: will be fired on USER_FETCH_REQUESTED actions
function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "USER_FETCH_FAILED", message: e.message});
   }
}
```