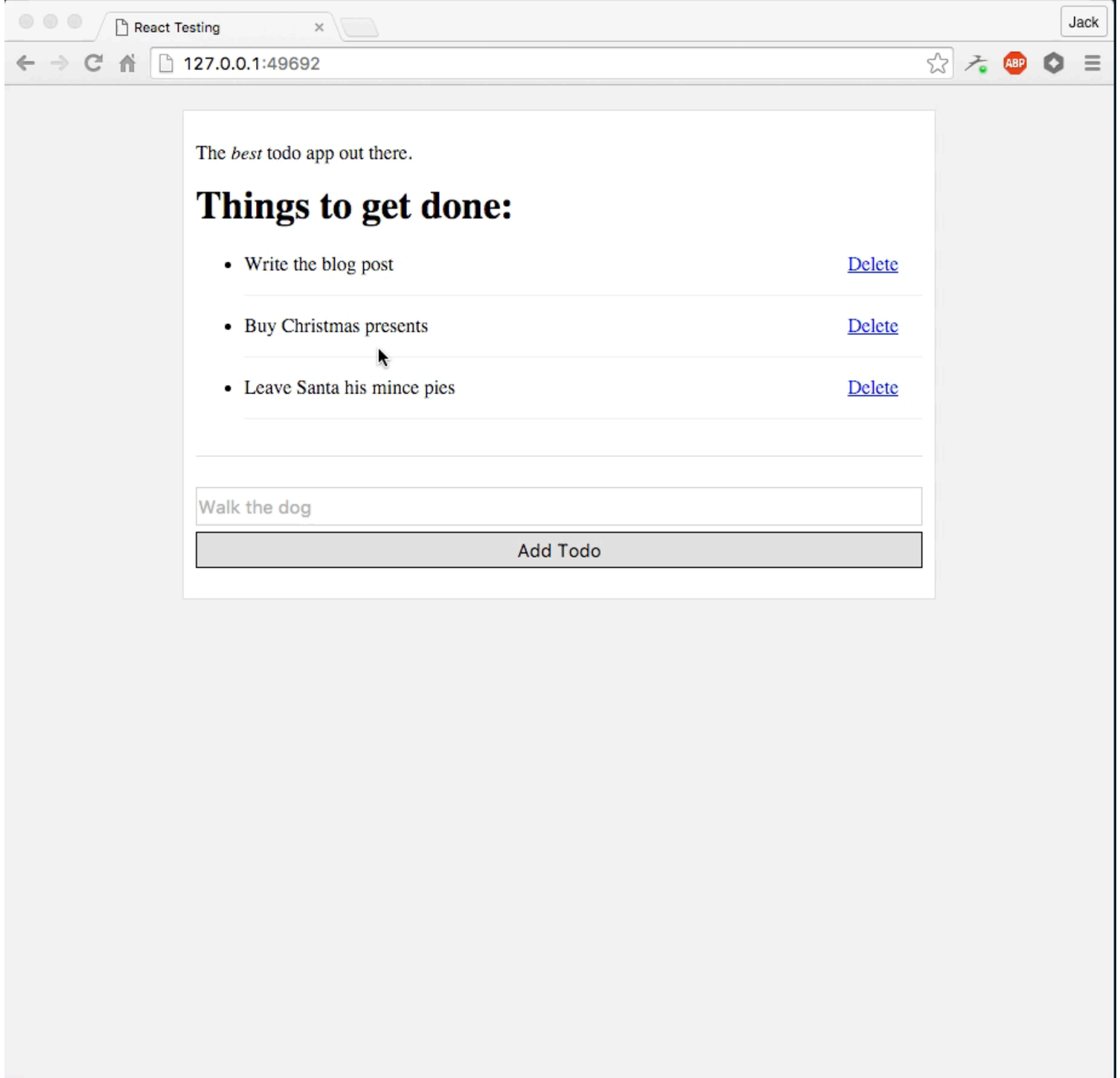
Managing Data with React and Redux

@Jack_Franklin
@pusher

- Code, notes, etc: **github.com/jackfranklin/react-redux-talk**
- Slides (after talk): **speakerdeck.com/jackfranklin**
- I'll tweet them all: **twitter.com/jack_franklin**


React Testing
Jack
127.0.0.1:49692
The *best* todo app out there.
Things to get done:
Write the blog post
Delete
Buy Christmas presents
Delete
Leave Santa his mince pies
Delete
Walk the dog
Add Todo

```
▼ <AppComponent>
  ▼ <div>
    ▼ <Todos>
      ▼ <div>
        ▶ <p>…</p>
          <h1>Things to get done:</h1>
        ▼ <ul className="todos-list">
          ▼ <li>
            ▶ <Todo todo={id: 1, name: "Write the blog post", done: false} doneChange=doneChange() deleteTodo=deleteTodo()>…</Todo>
            </li>
          ▼ <li>
            ▶ <Todo todo={id: 2, name: "Buy Christmas presents", done: false} doneChange=doneChange() deleteTodo=deleteTodo()>…</Todo>
            </li>
          ▼ <li>
            ▶ <Todo todo={id: 3, name: "Leave Santa his mince pies", done: false} doneChange=doneChange() deleteTodo=deleteTodo()>…</Todo
            </li>
          </ul>
        ▶ <AddTodo onNewTodo=onNewTodo()>…</AddTodo>
        </div>
      </Todos>
```

In the beginning

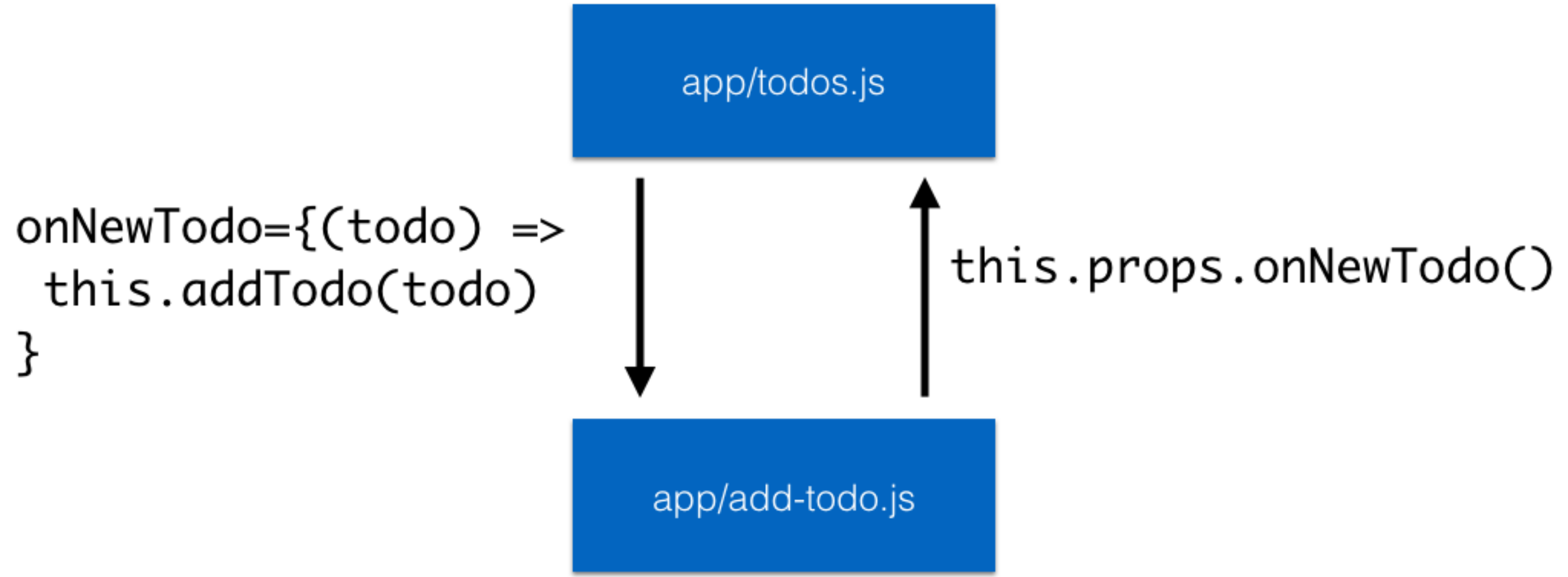
app/todos.js

```
export default class Todos extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: [
        { id: 1, name: 'Write the blog post', done: false },
        { id: 2, name: 'Buy Christmas presents', done: false },
        { id: 3, name: 'Leave Santa his mince pies', done: false },
      ]
    }
  }
  ...
}
```

app/todos.js

```
render() {
  return (
    <div>
      <p>The <em>best</em> todo app out there.</p>
      <h1>Things to get done:</h1>
      <ul className="todos-list">{ this.renderTodos() }</ul>
      <AddTodo onNewTodo={(todo) => this.addTodo(todo)} />
    </div>
  )
}
```

```
<AddTodo onNewTodo={(todo) => this.addTodo(todo)} />
```



Parent component contains all state.

Child components are given functions to call to tell the parent component of the new state.

app/todos.js contained the logic for updating the state from some user input.

```
constructor(props) {...}

addTodo(todo) {
  const newTodos = this.state.todos.concat([todo]);
  this.setState({ todos: newTodos });
}

...

render() {...}
```

But then as this component grew I pulled out the business logic into standalone JavaScript functions:

app/todos.js

```
constructor(props) {...}

addTodo(todo) {
  this.setState(addTodo(this.state, todo));
}

...

render() {...}
```

State functions can take the current state and produce a new state.

```
export function deleteTodo(state, id) {
  return {
    todos: state.todos.filter((todo) => todo.id !== id)
  };
}
```

This is effectively a very, very basic Redux (but worse in many ways!).

This is fine for small applications, but it tightly couples components and makes refactoring or restructuring components trick and makes refactoring or restructuring components tricky.

The more data you have, the more difficult it is to manage as different components can edit different pieces of data.

If you split the data up across components, you no longer have a single source of truth for your application's data.

It's tricky to track down what caused the data to change, and where it happened.

grep setState

As your application grows you need some process and structure around your data.

But don't use Redux by default! For smaller apps you'll probably find yourself quite content without.

Redux

The three principles of Redux.

- Single Source of Truth: all data is stored in one object.
- State is read only: nothing can directly mutate the state.
- The state is manipulated by pure functions: no external data can affect them.

Building a Redux application

```
import { createStore } from 'redux';

function counter(state, action) {
  ...
}

const store = createStore(counter);

console.log('Current state', store.getState());
```

- **store**: the object that holds our state
- **action**: an object sent to the store in order to manipulate the store's data
- **reducer**: a function that takes the current state, an action and produces the new state.

First: define your actions.

```
{ type: 'INCREMENT' }
```

```
{ type: 'DECREMENT' }
```

Second: define how your reducer should deal with those actions:

```
function counter(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

Third: define what to do if you don't have any state:

```
function counter(state, action) {
  if (!state) state = 0;
  ...
}

// OR
function counter(state = 0, action) {
  ...
}
```

Fourth: create your store and **dispatch** some actions:

```
const store = createStore(counter);

store.dispatch({ type: 'INCREMENT' });

console.log('Current state', store.getState());
// => 1
```

What makes this good?

- The main logic is contained in a function, abstracted away from the store. It's easy to modify, follow and test.
- Nothing ever manipulates the state, all manipulation is done via actions.
- Actions are just plain objects; they can be logged, serialised, repeated, and so on.
- Our reducer is pure - the state is never

Adding Redux to a React application

```
npm install --save redux react-redux
```

First, let's decide what our state will look like:

```
{
  todos: [
    { id: 1, name: 'buy milk', done: false },
    ...
  ]
}
```

Secondly, let's define the actions.

- `{ type: 'ADD_TODO', name: '...' }`
- `{ type: 'DELETE_TODO', id: ... }`
- `{ type: 'TOGGLE_TODO', id: ... }`

Thirdly, let's define the reducer function that will deal with these actions.

```
export default function todoAppReducers(
  state = { todos: [] },
  action
) {
  ...
};
```

We can first deal with **ADD_TODO:**

```
switch (action.type) {
  case 'ADD_TODO':
    const todo = {
      name: action.name,
      id: state.todos.length,
      done: false
    }

    return {
      todos: state.todos.concat([todo])
    }
}
```

And then **DELETE_TODO:**

```
case 'DELETE_TODO':
  return {
    todos: state.todos.filter((todo) => todo.id !== action.id)
  }
```

And finally **TOGGLE_TODO:**

```
case 'TOGGLE_TODO':
  const todos = state.todos.map((todo) => {
    if (todo.id === action.id) {
      todo.done = !todo.done;
    }

    return todo;
  });

  return { todos };
```

We've just modelled most of our business logic without having to deal with UI interactions or anything else.

This is one of the biggest pluses to using Redux.

Now let's create a store and connect our components.

By default a component **does not** have access to the store.

Components that do are known as "smart" components.

Components that do not are known as "dumb" components.

app/index.js

```
import React from 'react';
import { render } from 'react-dom';
import Todos from './todos';

class AppComponent extends React.Component {
  render() {
    return <Todos />;
  }
}

render(
  <AppComponent />,
  document.getElementById('app')
);
```

We'll firstly create a store and **connect** our application.

```
// other imports skipped
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import todoAppReducers from './reducers';

const store = createStore(todoAppReducers);

class AppComponent extends React.Component {...}

render(
  <Provider store={store}>
    <AppComponent />
  </Provider>,
  document.getElementById('app')
);
```

All the **Provider** does is make components in our application able to connect to the store if we give them permission.

You only need to wrap your top level component in the **Provider,** and once it's done you can mostly forget about it.

We now have a store and our top level component has been given access to it.

Now we need to give our components access to the store so they can render the data to it.

The **Todos** component needs access to the todos in the store so it can render the individual todos.

We can use the **connect** function from react-redux to do this.

app/todos.js

```
import { connect } from 'react-redux';

class Todos extends React.Components {...};

const ConnectedTodos = connect((state) => {
  return {
    todos: state.todos
  }
})(Todos);
```

The **connect** function takes a React component and produces a new component that will be given access to parts of the state as props.

This lets us strictly control which parts of our state each component has access to.

It also provides **this.props.dispatch**, which is used to dispatch actions, which we'll see later.

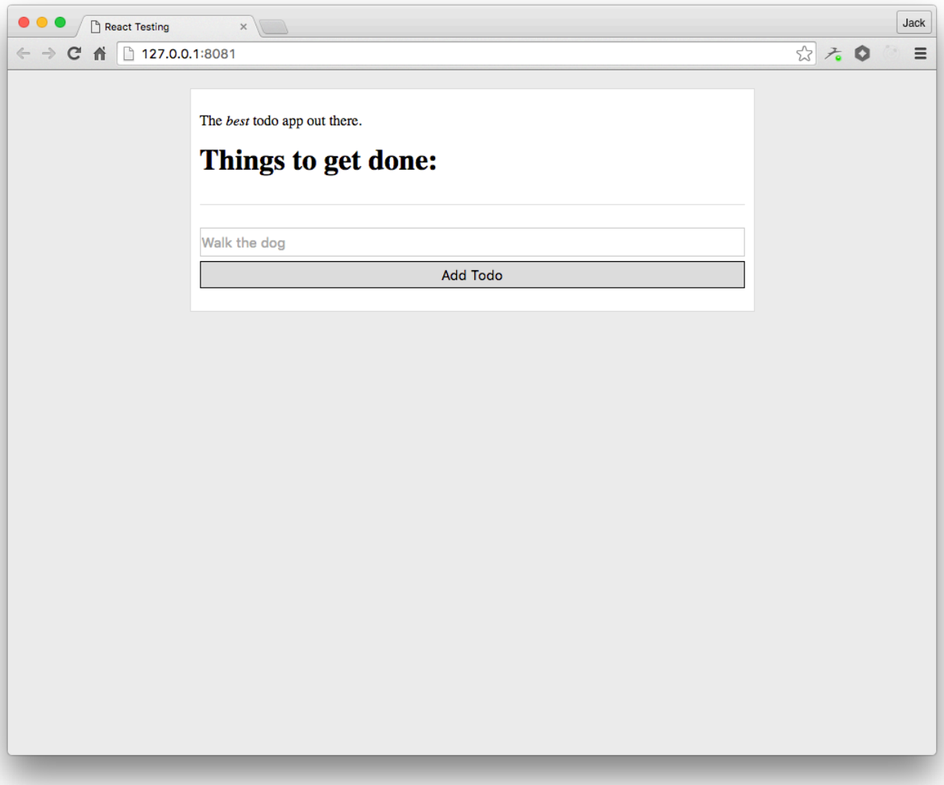
So now within our **Todos** component we can swap **this.state.todos** to **this.props.todos**.

We can also get rid of all the functions for managing state, and stop passing them through to child components.

```
class Todos extends React.Component {
  renderTodos() {
    return this.props.todos.map((todo) => {
      return <li key={todo.id}><Todo todo={todo} /></li>;
    });
  }

  render() {
    return (
      <div>
        <ul className="todos-list">{ this.renderTodos() }</ul>
        <AddTodo />
      </div>
    )
  }
}
```

Notice how much cleaner this is, and how our component is purely focused on presentation.

Now let's hook up **AddTodo** so we can create new todos.



AddTodo doesn't need to access any data in the store but it does need to dispatch actions, so it too must be connected.

```
class AddTodo extends React.Component {...};

const ConnectedAddTodo = connect()(AddTodo);

export default ConnectedAddTodo;
```

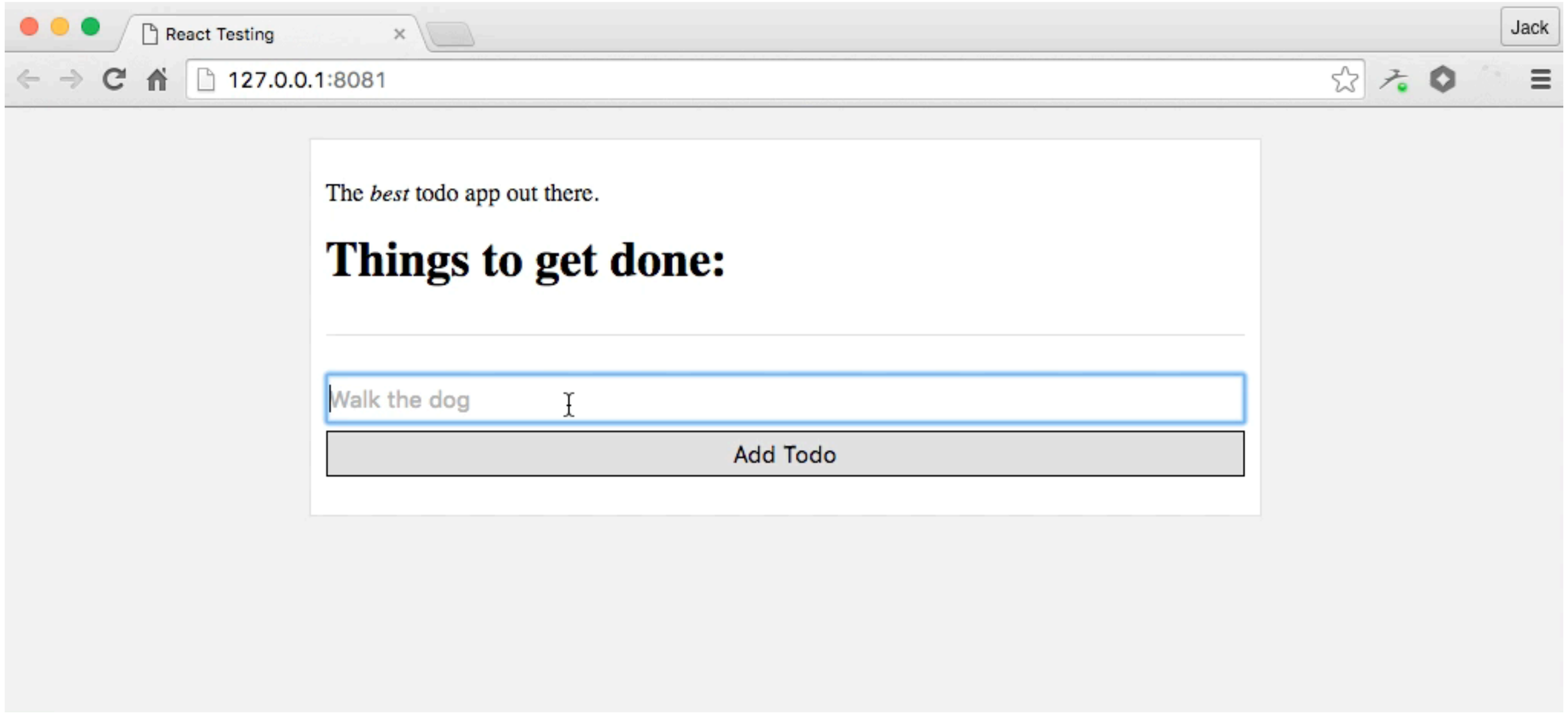
The old **addTodo** method:

```
addTodo(e) {
  e.preventDefault();
  const newTodoName = this.refs.todoTitle.value;
  if (newTodoName) {
    this.props.onNewTodo({
      name: newTodoName
    });

    this.refs.todoTitle.value = '';
  }
}
```

The new one:

```
addTodo(e) {
  e.preventDefault();
  const newTodoName = this.refs.todoTitle.value;
  if (newTodoName) {
    this.props.dispatch({
      name: newTodoName,
      type: 'ADD_TODO'
    });
    ...
  }
}
```

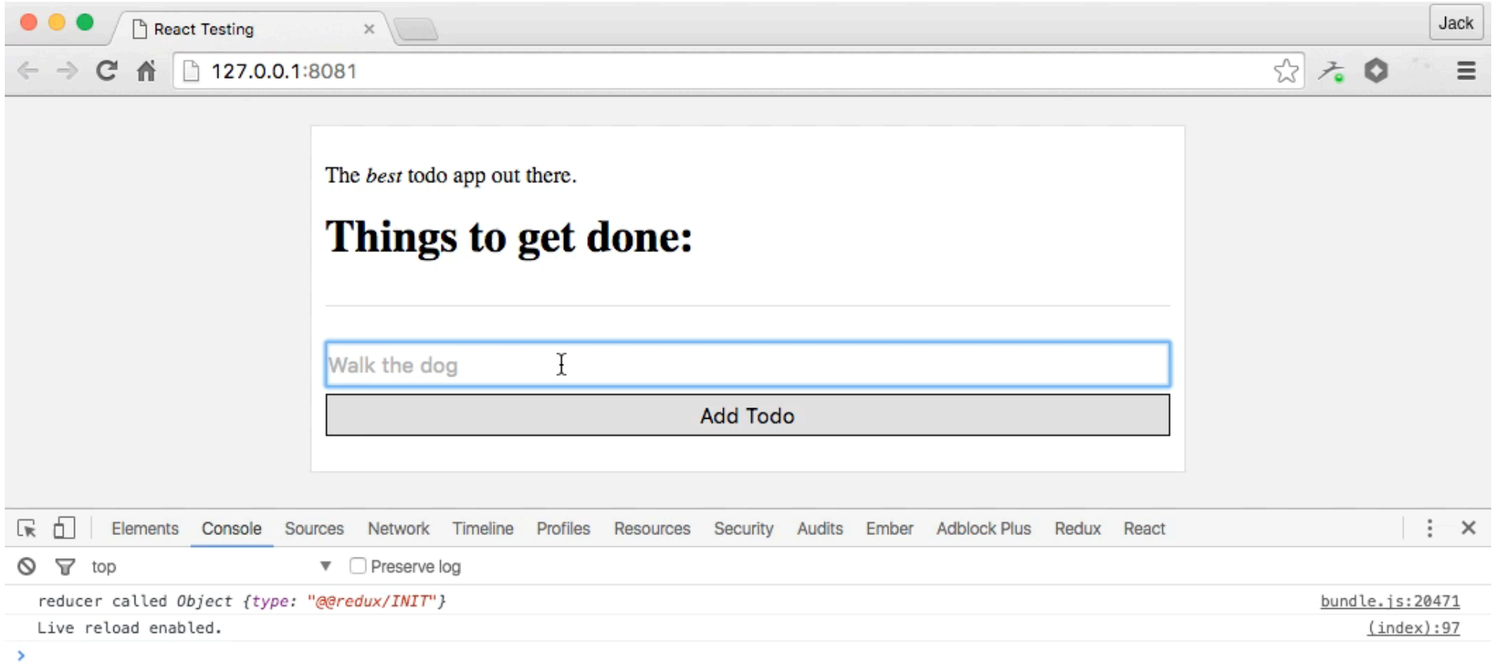
And we're now using Redux to add Todos!



Finally, we can update the **Todo** component to dispatch the right actions for toggling and deleting.

```
toggleDone() {
  this.props.dispatch({
    type: 'TOGGLE_TODO',
    id: this.props.todo.id
  });
}

deleteTodo(e) {
  this.props.dispatch({
    type: 'DELETE_TODO',
    id: this.props.todo.id
  });
}
```

But there's a problem!



Mutation!

Redux expects you to never mutate anything, and if you do it can't always correctly keep your UI in sync with the state.

We've accidentally mutated...

In our reducer...

```
case 'TOGGLE_TODO':
  const todos = state.todos.map((todo) => {
    if (todo.id === action.id) {
      todo.done = !todo.done;
    }

    return todo;
  });

  return { todos };
```

```
todo.done = !todo.done;
```

A quick rewrite...

```
case 'TOGGLE_TODO':
  const todos = state.todos.map((todo) => {
    if (todo.id === action.id) {
      return {
        name: todo.name,
        id: todo.id,
        done: !todo.done
      }
    }

    return todo;
  });
```

And it all works, as does deleting a todo. We're fully Reduxed!

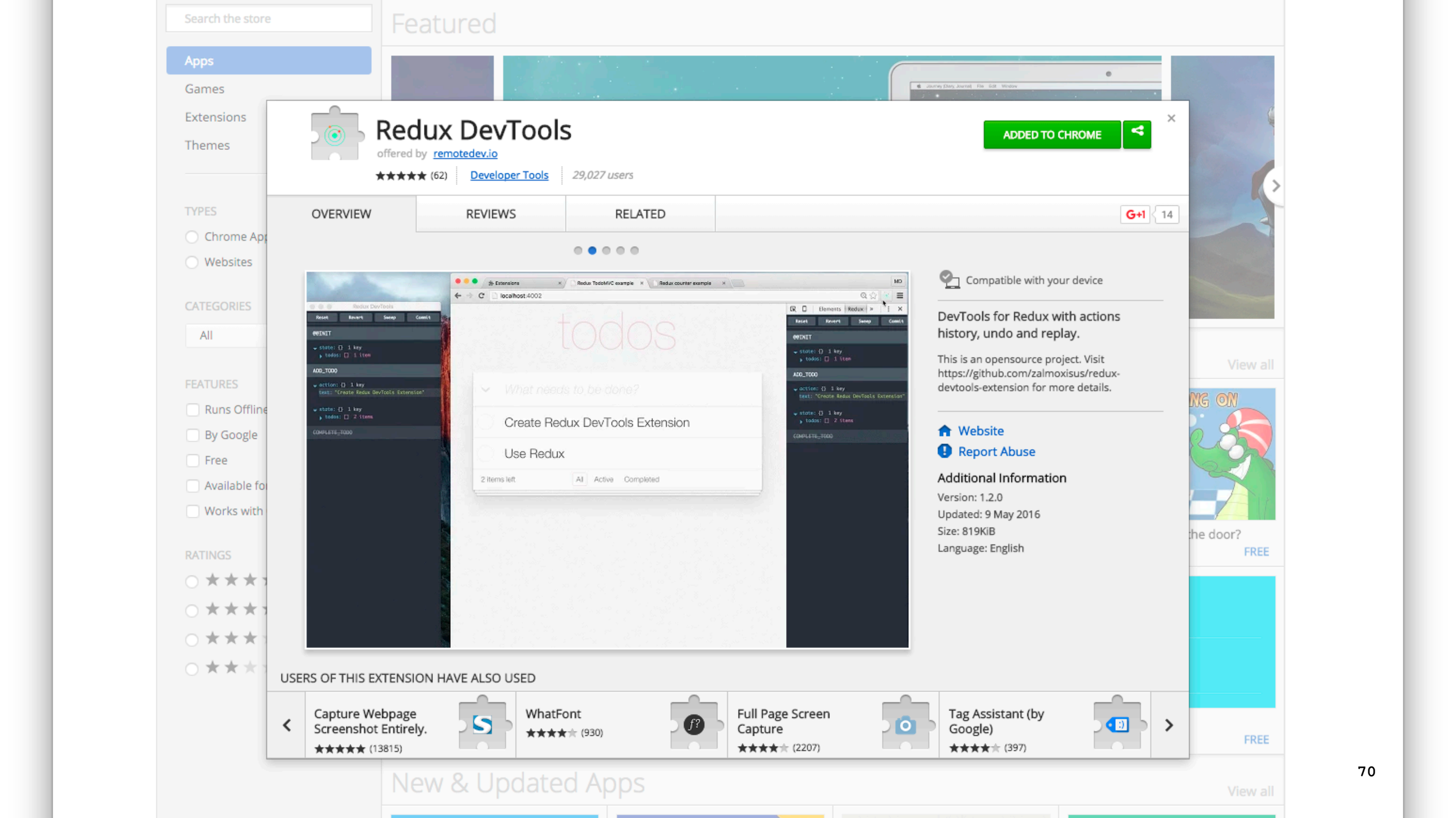
Deep breath...

That probably felt like a lot of effort, but the good news is once you've set Redux up you are set.

1. Decide the shape of your state.
2. Decide the actions that can update the state.
3. Define your reducers that deal with actions.
4. Wire up your UI to dispatch actions.
5. Connect your components to the store to allow them to render state.

Still to come

1. Debugging Redux
2. Better Redux Reducers
3. Middlewares
4. Async actions


Search the store
Apps
Games
Extensions
Themes
TYPES
Chrome App
Websites
CATEGORIES
All
FEATURES
Runs Offline
By Google
Free
Available fo
Works with
RATINGS
Featured
Redux DevTools
offered by remotedev.io
(62)
Developer Tools
29,027 users
ADDED TO CHROME
OVERVIEW
REVIEWS
RELATED
G+1
14
todos
What needs to be done?
Create Redux DevTools Extension
Use Redux
2 items left
All
Active
Completed
Compatible with your device
DevTools for Redux with actions history, undo and replay.
This is an opensource project. Visit https://github.com/zalmoxisus/redux-devtools-extension for more details.
Website
Report Abuse
Additional Information
Version: 1.2.0
Updated: 9 May 2016
Size: 819KiB
Language: English
USERS OF THIS EXTENSION HAVE ALSO USED
Capture Webpage Screenshot Entirely.
(13815)
WhatFont
(930)
Full Page Screen Capture
(2207)
Tag Assistant (by Google)
(397)
View all
FREE
New & Updated Apps
View all

If you're not using Chrome you can still use the devtools but it takes a bit more effort.

See: https://github.com/gaearon/redux-devtools

Firstly, install the plugin in Chrome.

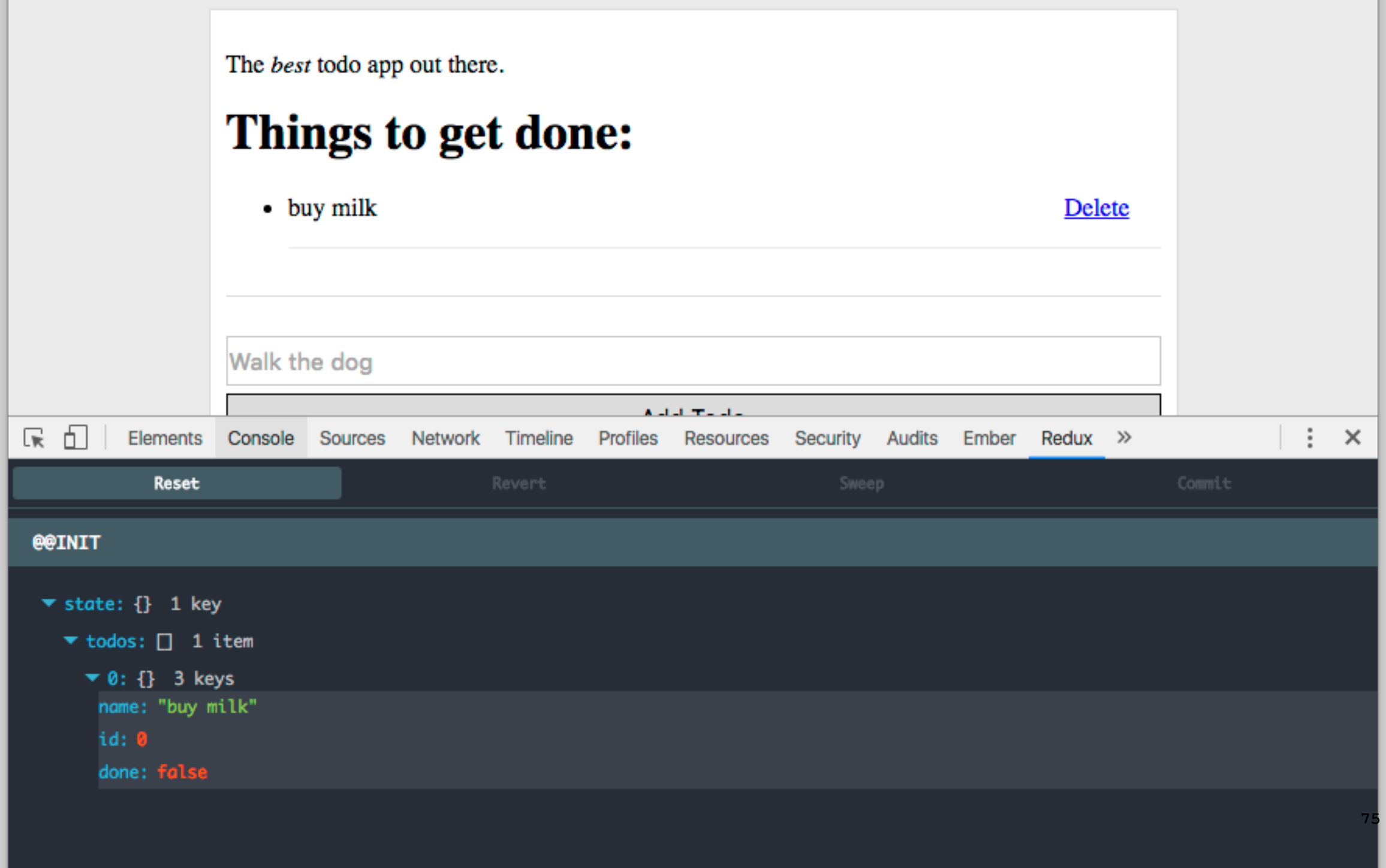
Secondly, update the call to **createStore:**

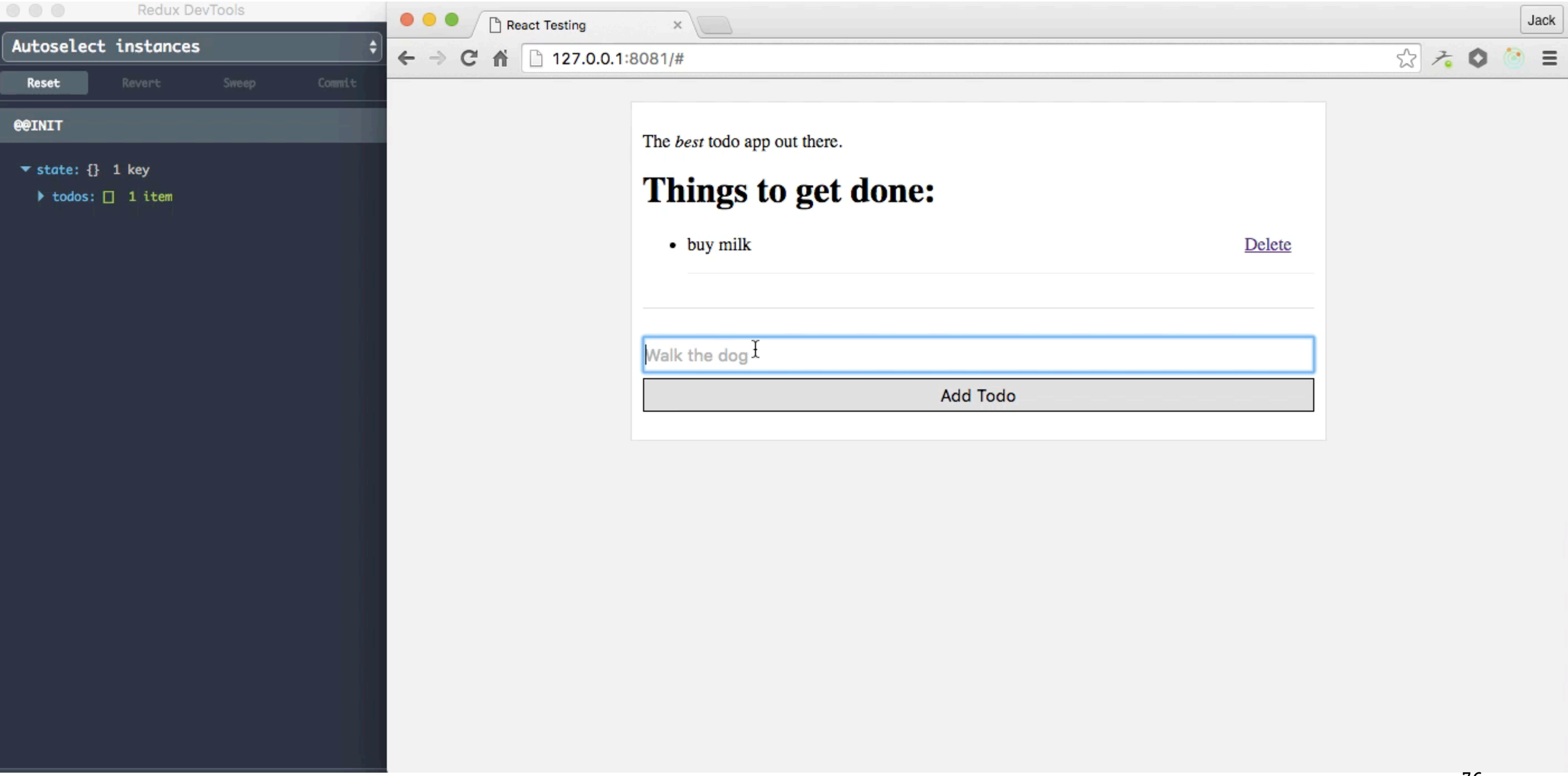
createStore(reducers, initialState, enhancer).

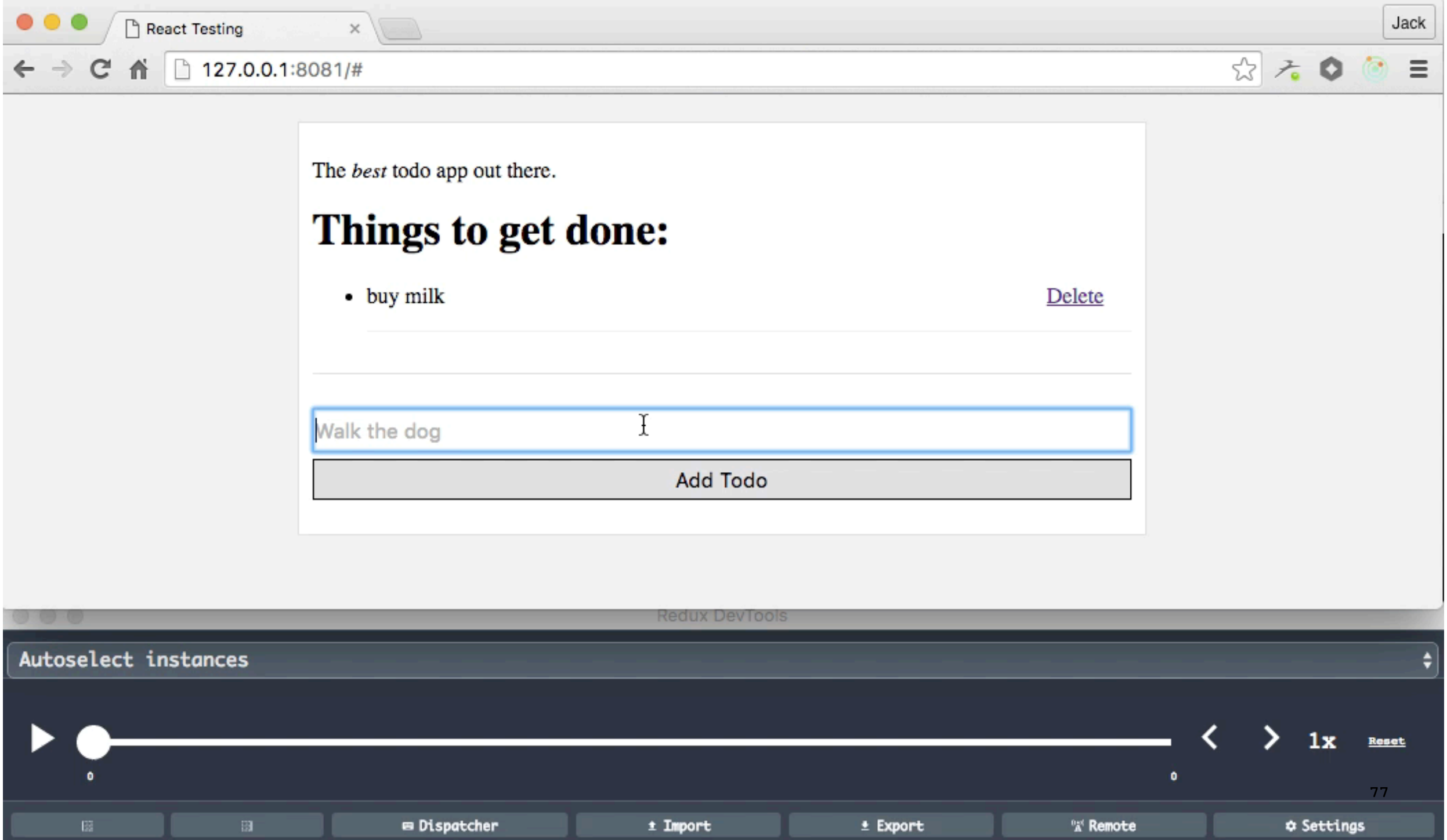
Enhancer: a function that enhances the store with middleware or additional functionality. We'll see this again later.

```
const store = createStore(
  todoAppReducers,
  undefined,
  window.devToolsExtension ? window.devToolsExtension() : undefined
);
```

We leave the **initialState** as undefined because our reducer deals with no state.


The *best* todo app out there.
Things to get done:
buy milk
Delete
Walk the dog
Elements
Console
Sources
Network
Timeline
Profiles
Resources
Security
Audits
Ember
Redux
Reset
Revert
Sweep
Commit
@@INIT
state: {} 1 key
todos: 1 item
0: {} 3 keys
name: "buy milk"
id: 0
done: false


Redux DevTools
Autoselect instances
Reset
Revert
Sweep
Commit
@@INIT
state: {} 1 key
todos: 1 item
React Testing
Jack
127.0.0.1:8081/#
The best todo app out there.
Things to get done:
buy milk
Delete
Walk the dog
Add Todo


React Testing
Jack
127.0.0.1:8081/#
The *best* todo app out there.
Things to get done:
buy milk
Delete
Walk the dog
Add Todo
Redux DevTools
Autoselect instances
1x
Reset
0
0
77
Dispatcher
Import
Export
Remote
Settings

Better Reducers

Imagine our TODO app now needs to have a user log in first, and our state will now keep track of the user that's logged in.

Our new state will look like:

```
{
  todos: [
    { id: 1, name: 'buy milk', done: false },
    ...
  ],
  user: {
    id: 123,
    name: 'Jack'
  }
}
```

Next, let's define some new actions:

```
{ type: 'LOG_USER_IN', id: ..., name: '...' }
{ type: 'LOG_USER_OUT' }
```

And then our reducer needs to be updated. Firstly, now we have two keys in our state, we should update the reducers we have to not lose any keys they don't deal with.

Before:

```
case 'DELETE_TODO':
  return {
    todos: state.todos.filter((todo) => todo.id !== action.id)
  }
```

After:

```
case 'DELETE_TODO':
  return Object.assign({}, state, {
    todos: state.todos.filter((todo) => todo.id !== action.id)
  });
```

And now we can add reducers for the new user actions:

```
case 'LOG_USER_IN':
  return Object.assign({}, state, {
    user: { id: action.id, name: action.name }
  });

case 'LOG_USER_OUT':
  return Object.assign({}, state, {
    user: {}
  });
```

```
export default function todoAppReducers(
  state = { todos: [initialTodo], user: {} },
  action
) {
  switch (action.type) {
    case 'ADD_TODO':
      const todo = {
        name: action.name,
        id: state.todos.length,
        done: false
      }

      return Object.assign({}, state, {
        todos: state.todos.concat([todo])
      });

    case 'DELETE_TODO':
      return Object.assign({}, state, {
        todos: state.todos.filter((todo) => todo.id !== action.id)
      });

    case 'TOGGLE_TODO':
      const todos = state.todos.map((todo) => {
        if (todo.id === action.id) {
          return {
            name: todo.name,
            id: todo.id,
            done: !todo.done
          }
        }

        return todo;
      });

      return Object.assign({}, state, { todos });

    case 'LOG_USER_IN':
      return Object.assign({}, state, {
        user: { id: action.id, name: action.name }
      });

    case 'LOG_USER_OUT':
      return Object.assign({}, state, {
        user: {}
      });

    default:
      return state;

  }
};
```

Our reducer is huge, and deals with two different data sets:

- User
- Todos

In a larger app this will quickly become impossible to manage.

Instead we split into multiple reducers who are each resopnsible for a specific key in the state.

Each of these reducers is **only given their part of the state.**

```
function userReducer(user = {}, action) {
  switch (action.type) {
    case 'LOG_USER_IN':
      return {
        id: action.id,
        name: action.name
      }
    case 'LOG_USER_OUT':
      return {};

    default:
      return user;
  }
}
```

```
function todoReducer(todos = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      ...

    case 'DELETE_TODO':
      return todos.filter((todo) => todo.id !== action.id);

    case 'TOGGLE_TODO':
      ...

    default:
      return todos;
  }
}
```

And our main reducer function becomes:

```
export default function todoAppReducers(state = {}, action) {
  return {
    todos: todoReducer(state.todos, action),
    user: userReducer(state.user, action)
  }
};
```

Now as our state grows we'll add new functions for each key.

Turns out this pattern is so useful that Redux provides a method to do it for us: **combineReducers.**

Before:

```
export default function todoAppReducers(state = {}, action) {
  return {
    todos: todoReducer(state.todos, action),
    user: userReducer(state.user, action)
  }
};
```

After:

```
import { combineReducers } from 'redux';
...

const todoAppReducers = combineReducers({
  todos: todoReducer,
  user: userReducer
});

export default todoAppReducers;
```

Deep breath again!

Middlewares

You might be familiar with Rack Middlewares, NodeJS / Express middlewares, and so on.

It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer

-- http://redux.js.org/docs/advanced/Middleware.html

A middleware provides a function that is given the **store** object.

It should return another function that is called with **next,** the function it should call when it is finished.

That function should return another function that is called with the current action, **action.**

WHAT?!

```
const myMiddleware = function(store) {
  return function(next) {
    return function(action) {
      // your logic goes here
      // this is the dispatch function
      // that each middleware can return
    }
  }
}

const myMiddleware = store => next => action => {
  // your logic here
}
```

Each middleware replaces the **dispatch** function with its own function, and Redux chains them together correctly.

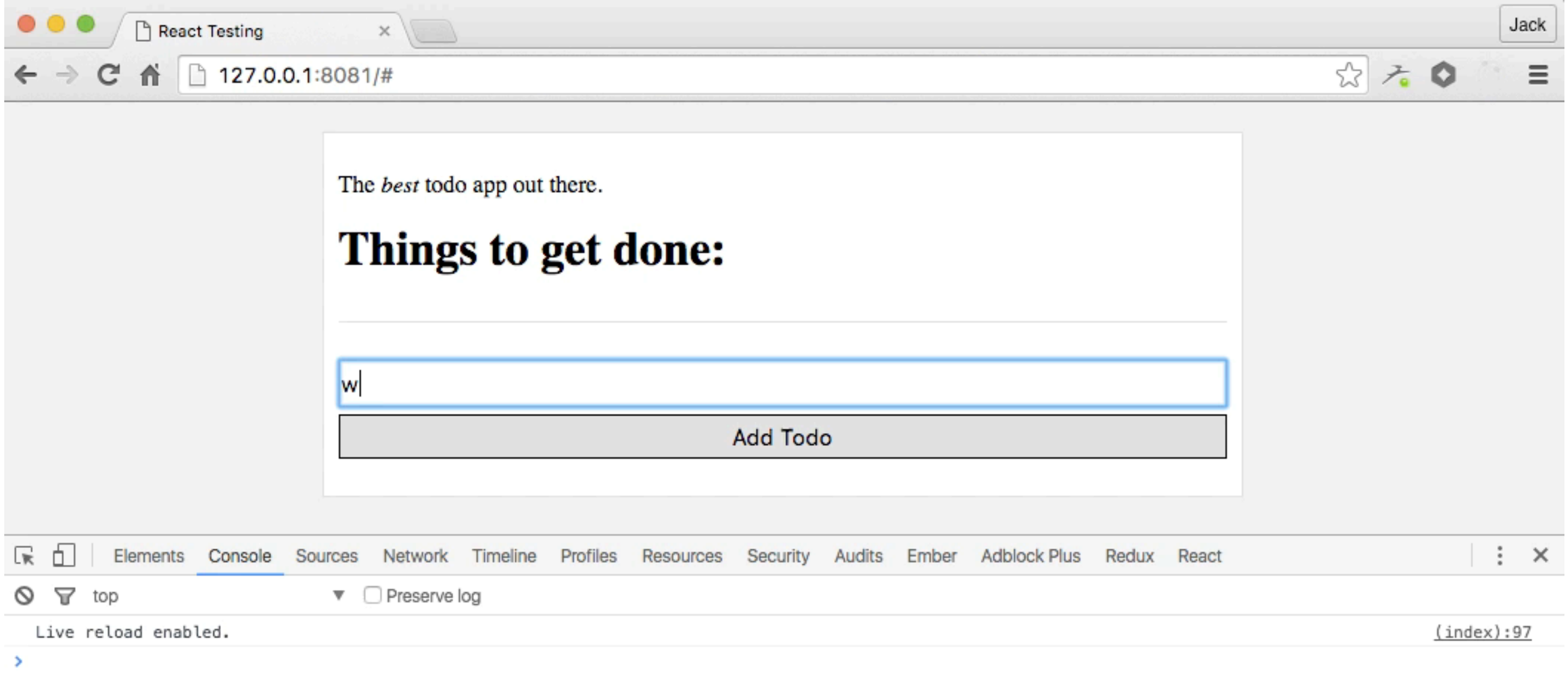
Example: logging each action

app/middlewares.js

```
export const logMiddleware = store => next => action => {
  console.log('MIDDLEWARE: About to dispatch', action);

  return next(action);
};
```

app/index.js

```
import { createStore, applyMiddleware } from 'redux';

...

const store = createStore(
  todoAppReducers,
  undefined,
  applyMiddleware(logMiddleware)
);
```


React Testing
Jack
127.0.0.1:8081/#
The *best* todo app out there.
Things to get done:
w
Add Todo
Elements
Console
Sources
Network
Timeline
Profiles
Resources
Security
Audits
Ember
Adblock Plus
Redux
React
top
Preserve log
Live reload enabled.
(index):97

But wait, we lost the devtools code!

compose

```
const store = createStore(
  todoAppReducers,
  undefined,
  compose(
    applyMiddleware(logMiddleware),
    window.devToolsExtension ? window.devToolsExtension() : f => f
  )
);
```

Final deep breath!

Async Actions

Up until now we've dealt purely with synchronous actions, but often your action will be async, most commonly, data fetching.

We could write our own middleware or logic to deal with this, but one already exists: **redux-thunk.**

```
npm install --save redux-thunk
```

With thunk, functions can either return an action or another function that can dispatch actions.

The HTTP request cycle:

- Make the request, and dispatch action.
- Get the data back, and dispatch an action with that data.
- Request errors, dispatch an action with information about the error.

When you want to dispatch an action to cause an HTTP request you actually want to trigger multiple actions.

```
export default function thunkMiddleware({ dispatch, getState }) {
  return next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState);
    }

    return next(action);
  };
}
```

Fake API:

```
export function fetchTodos() {
  return new Promise((resolve, reject) => {
    resolve({
      todos: [{
        id: 1,
        name: 'Buy Milk',
        done: false
      }]
    })
  });
}
```

We need some new information on the state:

```
{
  todos: ...,
  user: ...,
  isFetching: true / false
}
```

And some new actions:

```
{ type: 'REQUEST_TODOS_INIT' },
{ type: 'REQUEST_TODOS_SUCCESS', todos: [...] }
// and one for error handling
// if this were real
```

```
function isFetchingReducer(isFetching = false, action) {
 switch (action.type) {
   case 'REQUEST_TODOS_INIT':
     return true;

   case 'REQUEST_TODOS_SUCCESS':
     return false

   default:
     return isFetching
 }
}
```

```
const todoAppReducers = combineReducers({
  todos: todoReducer,
  user: userReducer,
  isFetching: isFetchingReducer
});
```

Action Creators

app/action-creators.js

```
export function fetchTodosAction() {
  return (dispatch) => {
  }
}
```

app/todos.js

```
import { fetchTodosAction } from './action-creators';


class Todos extends React.Component {
  componentWillMount() {
    this.props.dispatch(fetchTodosAction());
  }
  ...
}
```

First, dispatch **REQUEST_TODOS_INIT:**

```
export function fetchTodosAction() {
  return (dispatch) => {
    dispatch({ type: 'REQUEST_TODOS_INIT' });
  }
}
```

Then, fetch and dispatch **REQUEST_TODOS_SUCCESS:**

```
fetchTodos().then((data) => {
  dispatch({
    type: 'REQUEST_TODOS_SUCCESS',
    todos: data.todos
  });
});
```

```
import { fetchTodos } from './fake-api';

export function fetchTodosAction() {
  return (dispatch) => {
    dispatch({ type: 'REQUEST_TODOS_INIT' });

    fetchTodos().then((data) => {
      dispatch({
        type: 'REQUEST_TODOS_SUCCESS',
        todos: data.todos
      });
    });
  }
}
```

app/todos.js

Within render:

```
{ this.props.isFetching && <p>LOADING...</p> }
```

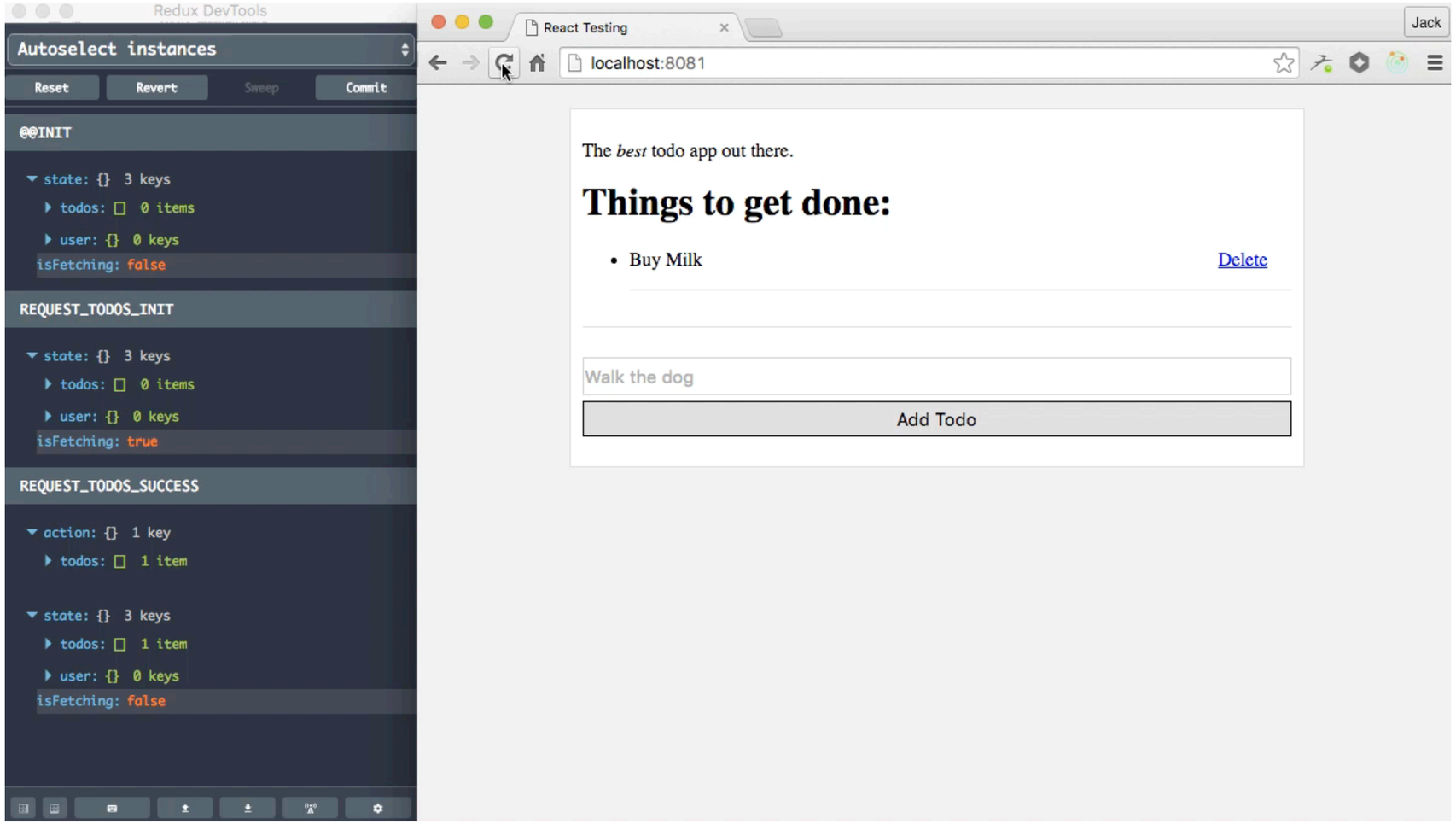
Allow it access:

```
const ConnectedTodos = connect((state) => {
  return {
    todos: state.todos,
    isFetching: state.isFetching
  };
})(Todos);
```

Now we need our **todosReducer** to deal with the success action and use the data.

```
case 'REQUEST_TODOS_SUCCESS':
  return action.todos;
```

Et voila (the delay is me for effect!):



And with that we now have support for async!

Housekeeping Actions

The strings for our actions are cropping up in a lot of places.

Better to have them as constants that can be exported.

```
export const LOG_USER_IN = 'LOG_USER_IN';
export const LOG_USER_OUT = 'LOG_USER_OUT';
```

That way you can keep things in sync easier.

Use action creators for creating actions:

```
export function addTodo(name) {
  return {
    type: ADD_TODO,
    name
  }
};
```

For more info:

- http://redux.js.org/docs/basics/Actions.html
- http://redux.js.org/docs/recipes/ReducingBoilerplate.html

You've made it!

Redux is tough to get started with but the benefits in a large application are huge.

- Code, notes, etc: **github.com/jackfranklin/react-redux-talk**
- Slides (after talk): **speakerdeck.com/jackfranklin**
- I'll tweet them all: **twitter.com/jack_franklin**

Thanks :)

- @Jack_Franklin
- javascriptplayground.com