# Redux

*"… predictable state container for JavaScript apps."* -- [Redux docs]()

Zuzana Dankovčíková

# Why do we need Redux?

We have already solved many problems of state management by

- treating data as **immutable objects** and
- having most of the **data stored in the root component**.

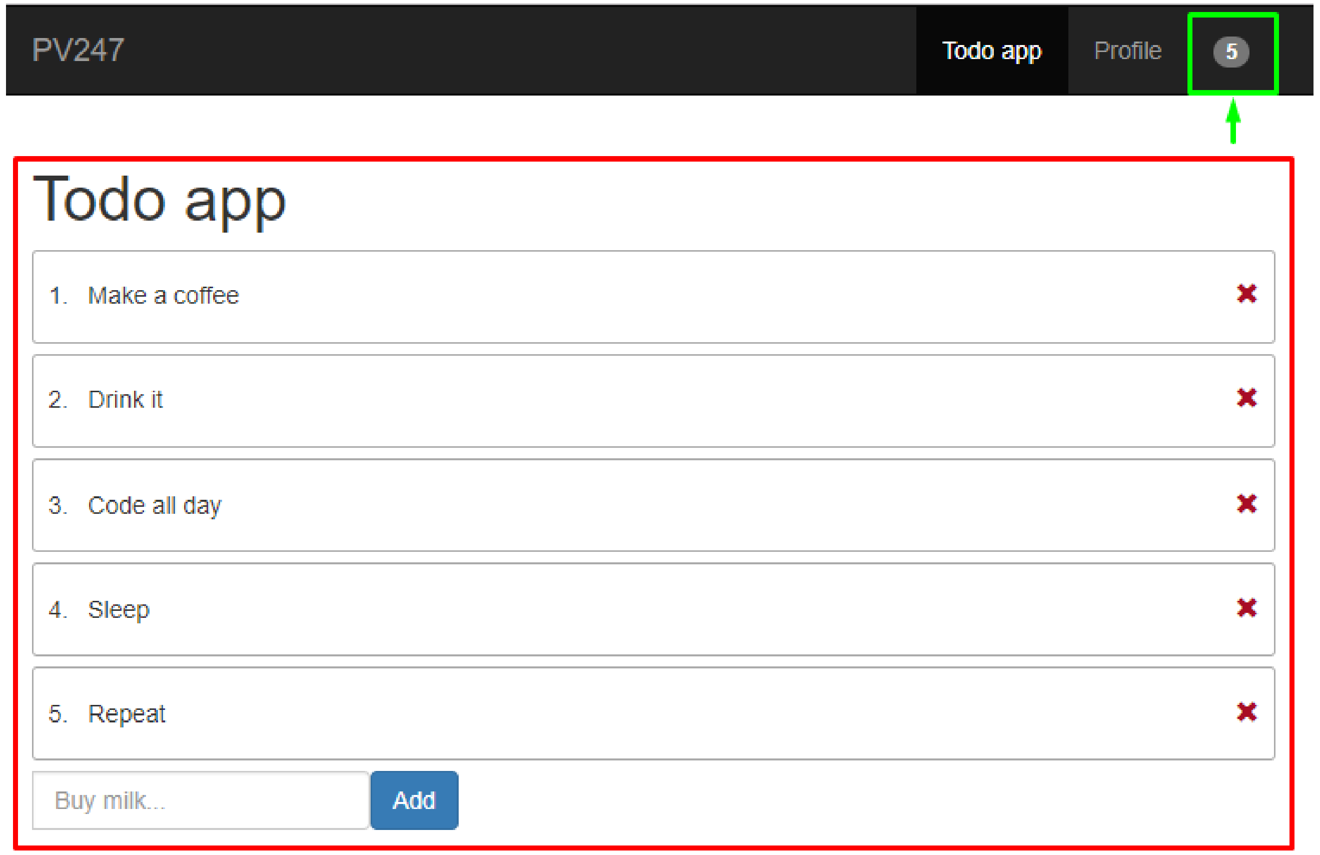
# Problem 1: What is “root component”

**New feature request:**

→ Displaying number of TODOs in the navigation bar?

→ **“Unrelated” components dependent on the same data.**

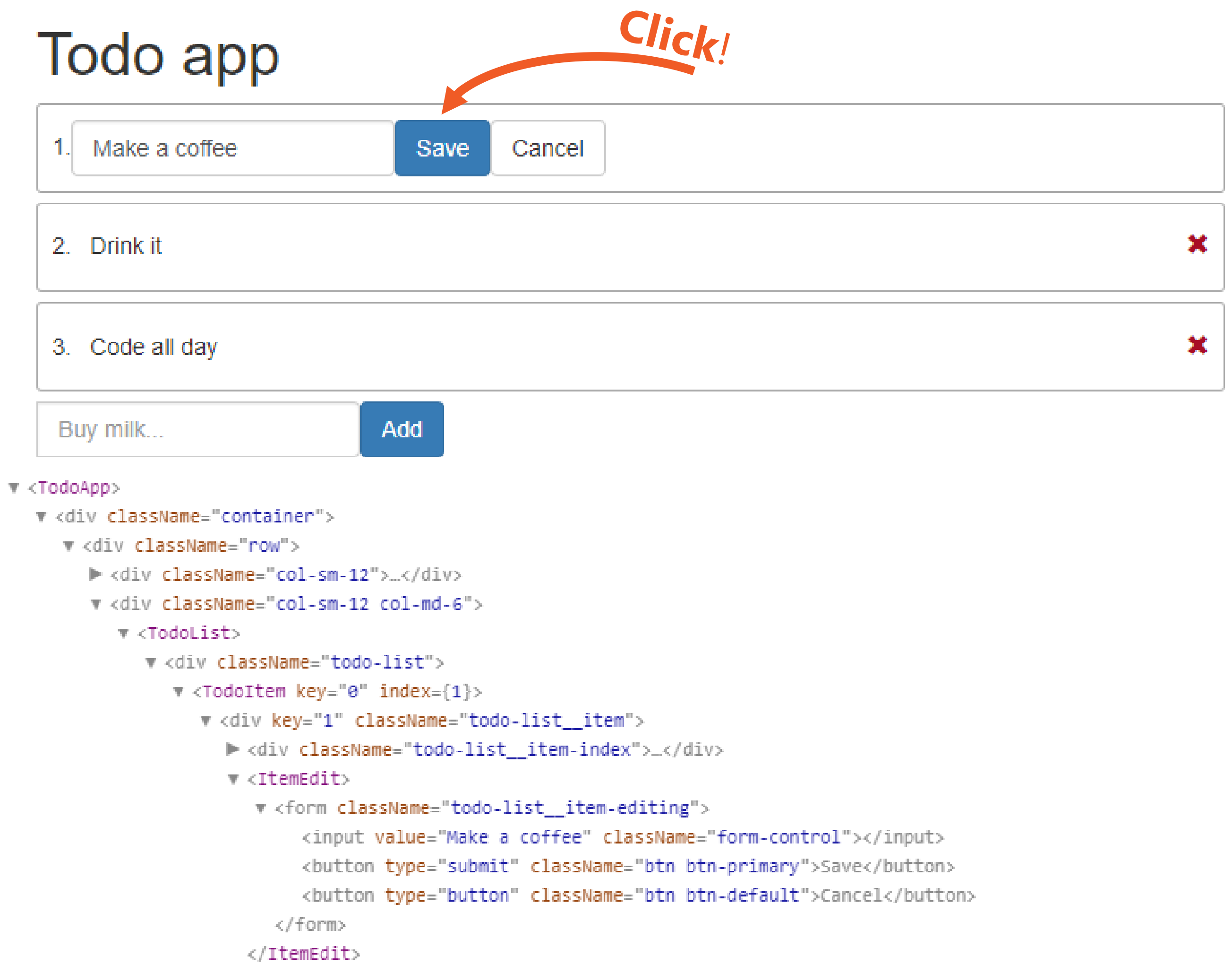
→ **Lifting state up**. But until when? How to make it scalable?



Todo app

1. Make a coffee
2. Drink it
3. Code all day
4. Sleep
5. Repeat

# Problem 2: Callbacks chain



Todo app

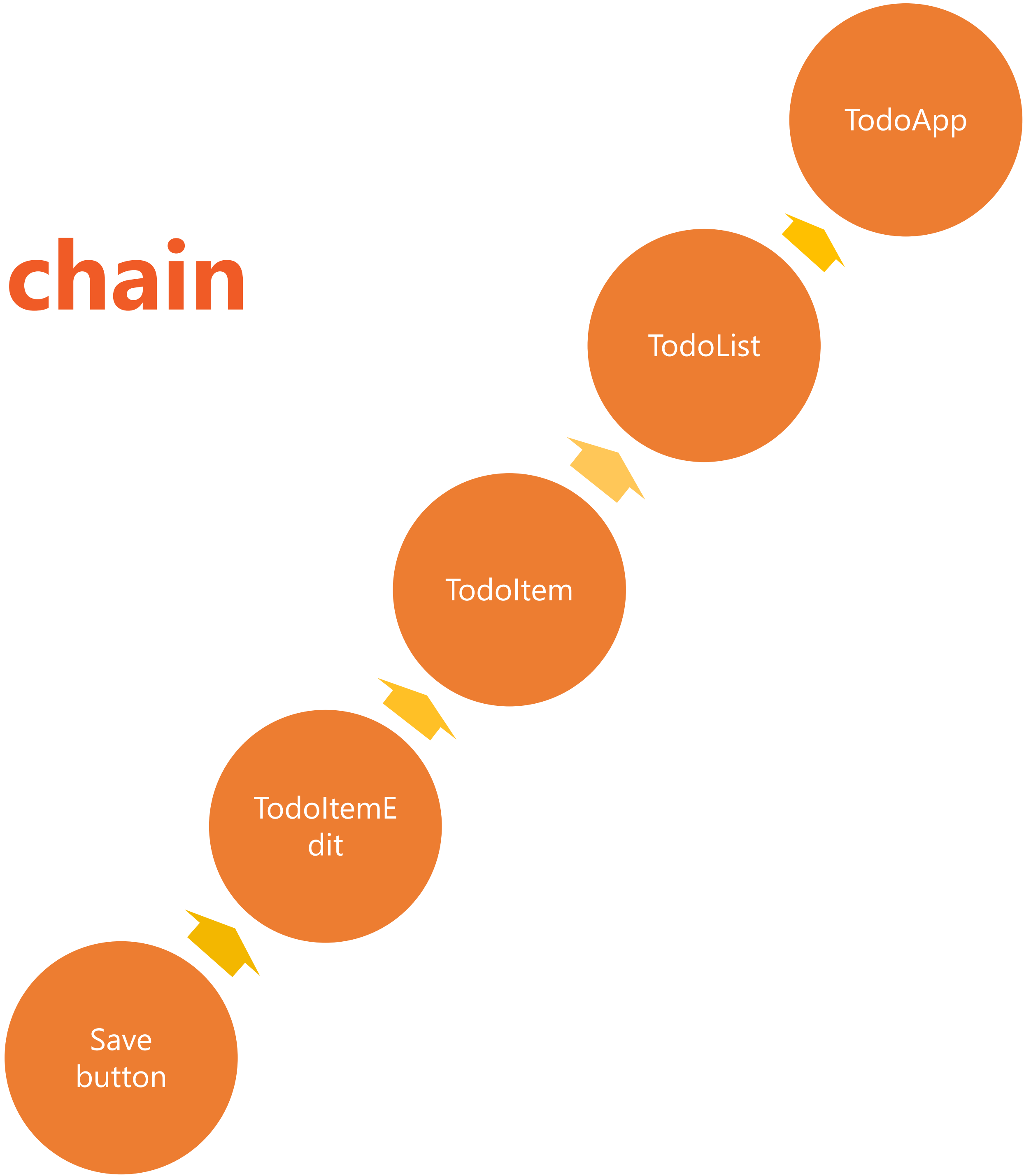
1. Make a coffee Save Cancel

2. Drink it ✖

3. Code all day ✖

Buy milk... Add

```
▼ <TodoApp>
  ▼ <div className="container">
    ▼ <div className="row">
      ▶ <div className="col-sm-12">…</div>
      ▼ <div className="col-sm-12 col-md-6">
        ▼ <TodoList>
          ▼ <div className="todo-list">
            ▼ <TodoItem key="0" index={1}>
              ▼ <div key="1" className="todo-list__item">
                ▶ <div className="todo-list__item-index">…</div>
                ▼ <ItemEdit>
                  ▼ <form className="todo-list__item-editing">
                      <input value="Make a coffee" className="form-control"></input>
                      <button type="submit" className="btn btn-primary">Save</button>
                      <button type="button" className="btn btn-default">Cancel</button>
                    </form>
                  </ItemEdit>
```

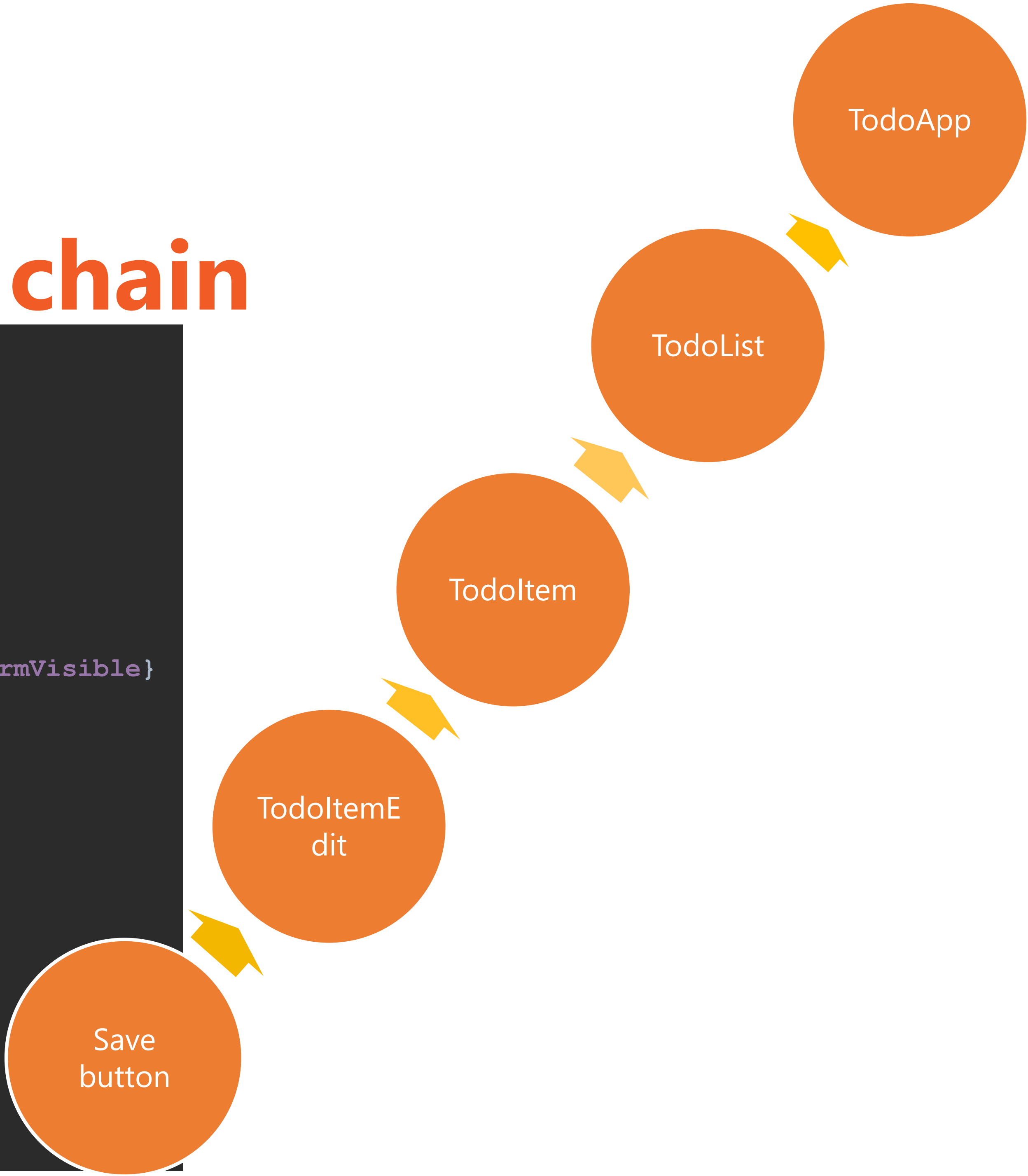


## Problem 2: Callbacks chain

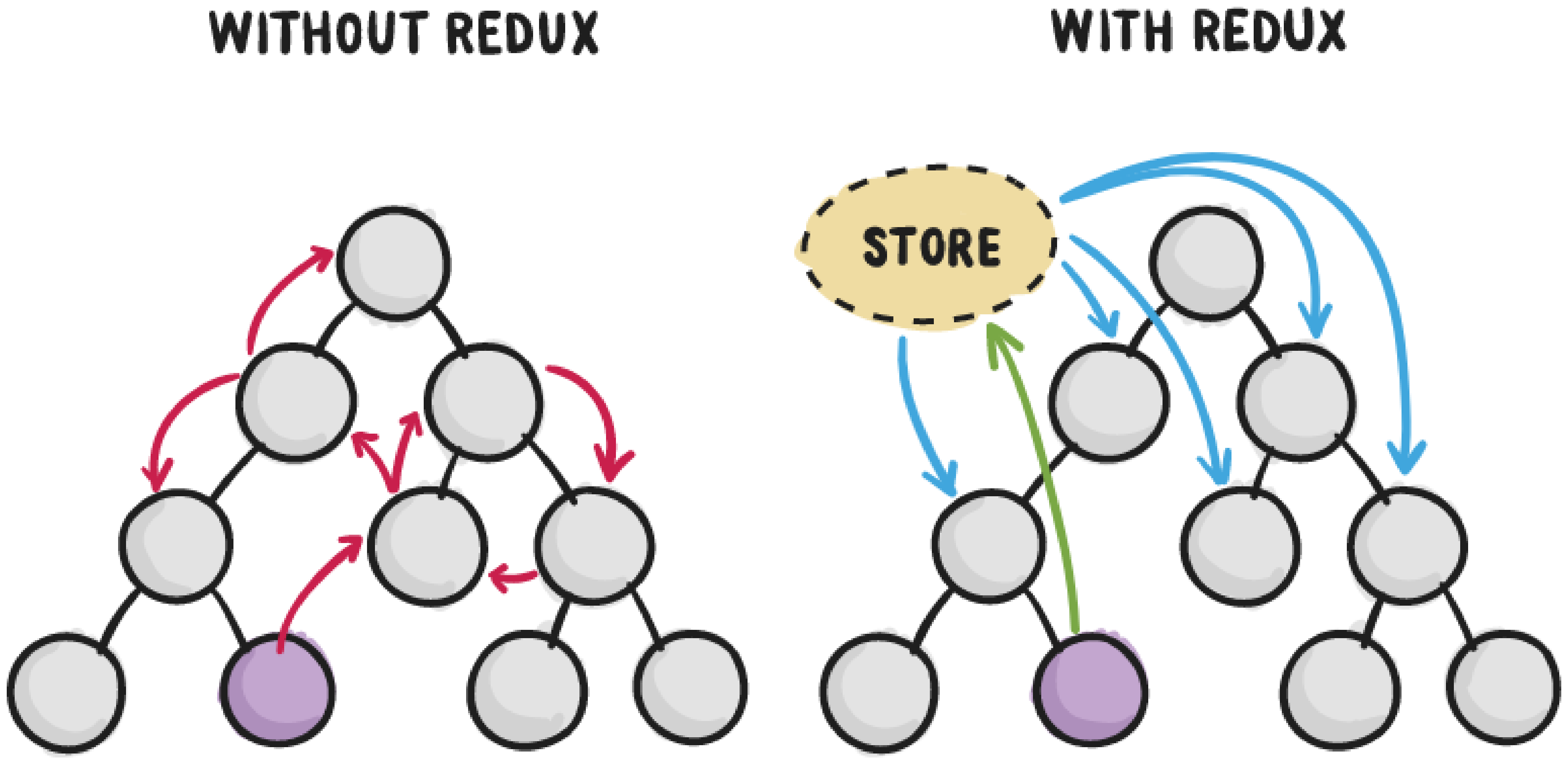
```
class TodoApp extends React.Component {

    // other methods
    // ...

    render() {
        return (
            <TodoList
                list={this.state.list}
                editedItemId={this.state.editedItemId}
                createNewFormVisible={this.state.createNewFormVisible}
                isDragging={this.state.isDragging}
                onDelete={this._deleteItem}
                onExpand={this._startEditing}
                onCancel={this._cancelEditing}
                onSave={this._updateItem}
                onReorder={this._moveItem}
                onCreateNewClick={this._showCreateNewForm}
                onCreateCancel={this._hideCreateNewForm}
                onCreate={this._createNewItem}
                onDragStarted={this._itemDragStarted}
                onDragEnded={this._itemDragEnded}
            />
        );
    }
}
```



Kentico



● COMPONENT INITIATING CHANGE

## Motivation

**Complex state management made easy**

- **Scalable** state management
- **Deterministic** and easily traceable changes
- **State is decoupled from presentation** (won’t break with every UI change)
- Better **dev tools** than console.log()
- Better **testability**

## 3 Principles of Redux

**Single source of truth:**
"The whole state of your app is stored in an object tree inside a single *store*."

**State is read-only:**
"The only way to change the state tree is to emit an *action*, an object describing what happened."

**Changes are made with pure functions:**
“To specify how the actions transform the state tree, you write pure *reducers*."

# Building blocks

**Action**

- describes UI changes

**Store**

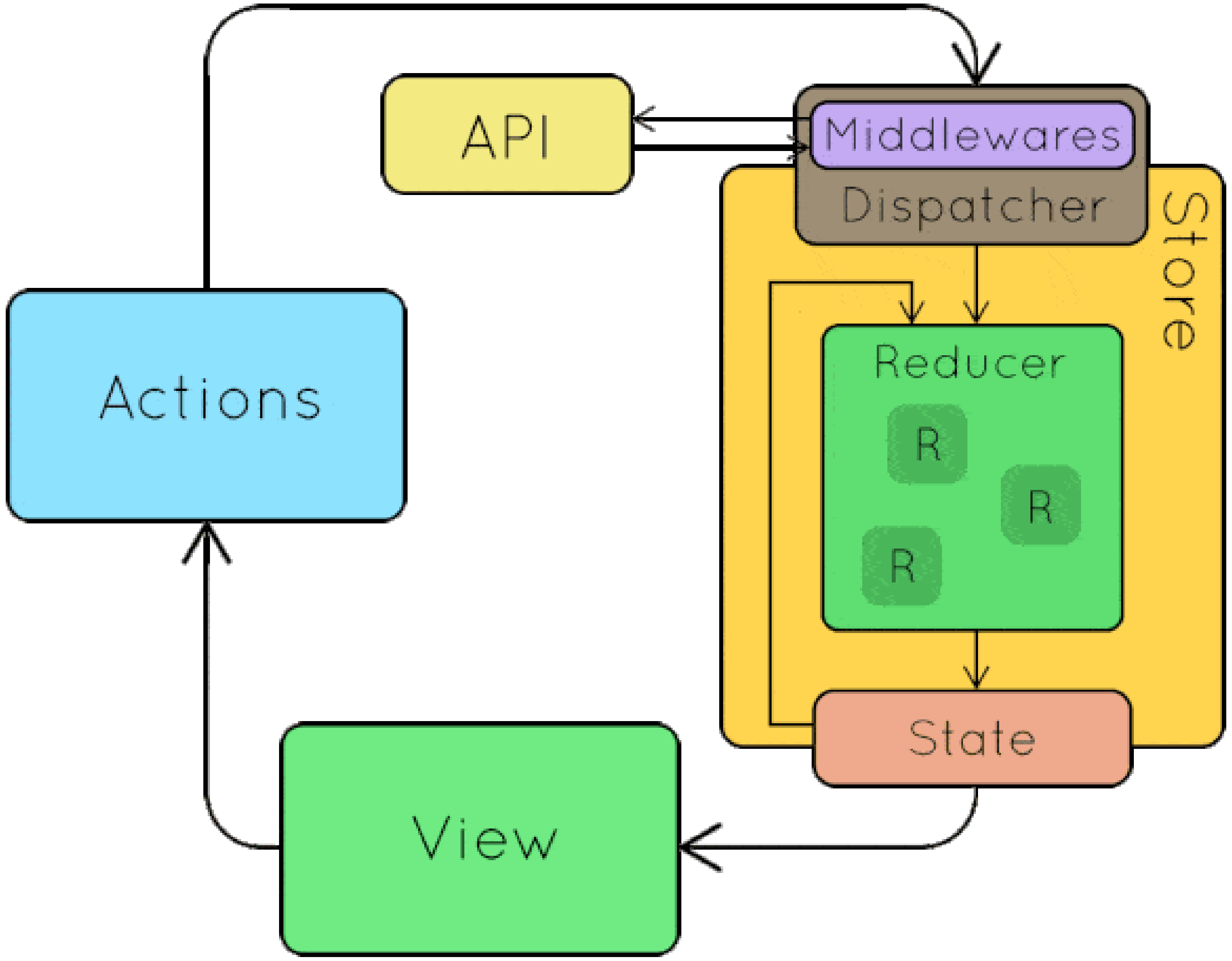
- receives action via dispatcher
- calls root reducer

**Reducer**

- (prevState, action) => newState

**View**

- gets notified about state change
- re-renders with new data



# Actions & Action creators

"**Actions** are payloads of information that send data from your application to your store. They are the *only* source of information for the store."

A new developer can go through all defined actions and immediately see the entire API – all the user interactions that are possible in your app.

**Action -** simple JS objects describing data change

```
{
  type: 'TODO_APP_ITEM_CREATE',
  payload: {
    id: 42,
    text: 'Buy milk'
  }
}
```

**Action creator -** helper function for creating actions

```
const createItem = (text) => ({
  type: TODO_APP_ITEM_CREATE,
    payload: {
      id: uuid(),
      text: text
    }
});
```

# Reducers

Action describes WHAT has happened, reducer specifies **HOW the state should change**

- **1 root reducer** that can be composed from many others
- Pure function **`(prevState, action) => nextState`**

What is a **pure function**? `(args) => result`

- It does not make outside network or database calls.
- Its return value depends solely on the values of its parameters.
- Its arguments should be considered "immutable" (must not be changed)
- **Calling a pure function with the same set of arguments will always return the same value.**

Kentico

## Pure or impure?

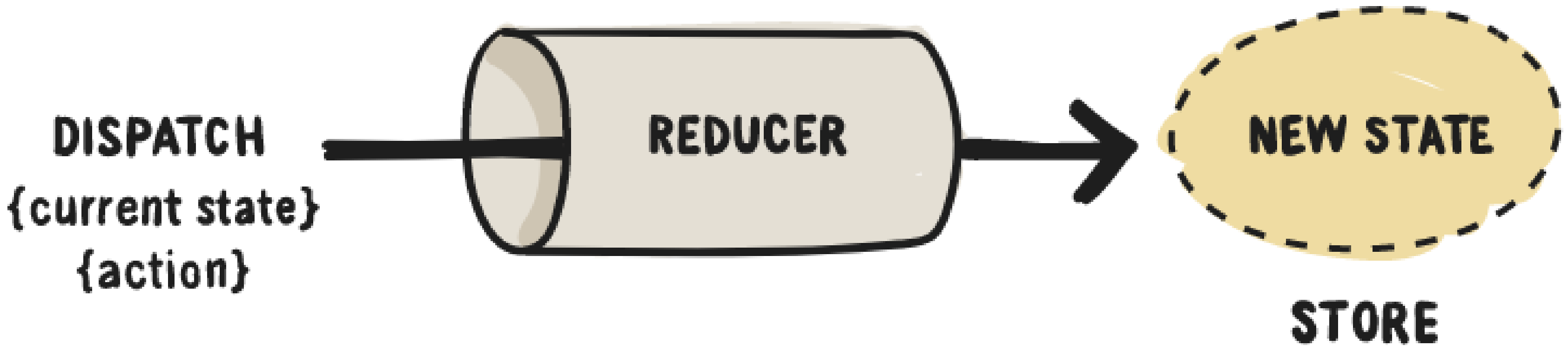
```
const getMagicNumber = () => Math.random();
```

```
const time = () => new Date().toLocaleTimeString();
```

```
const addFive = (val) => val + 5;
```

```
var count = 0;
const increaseCount = (val) => count += val;
```

# Reducers

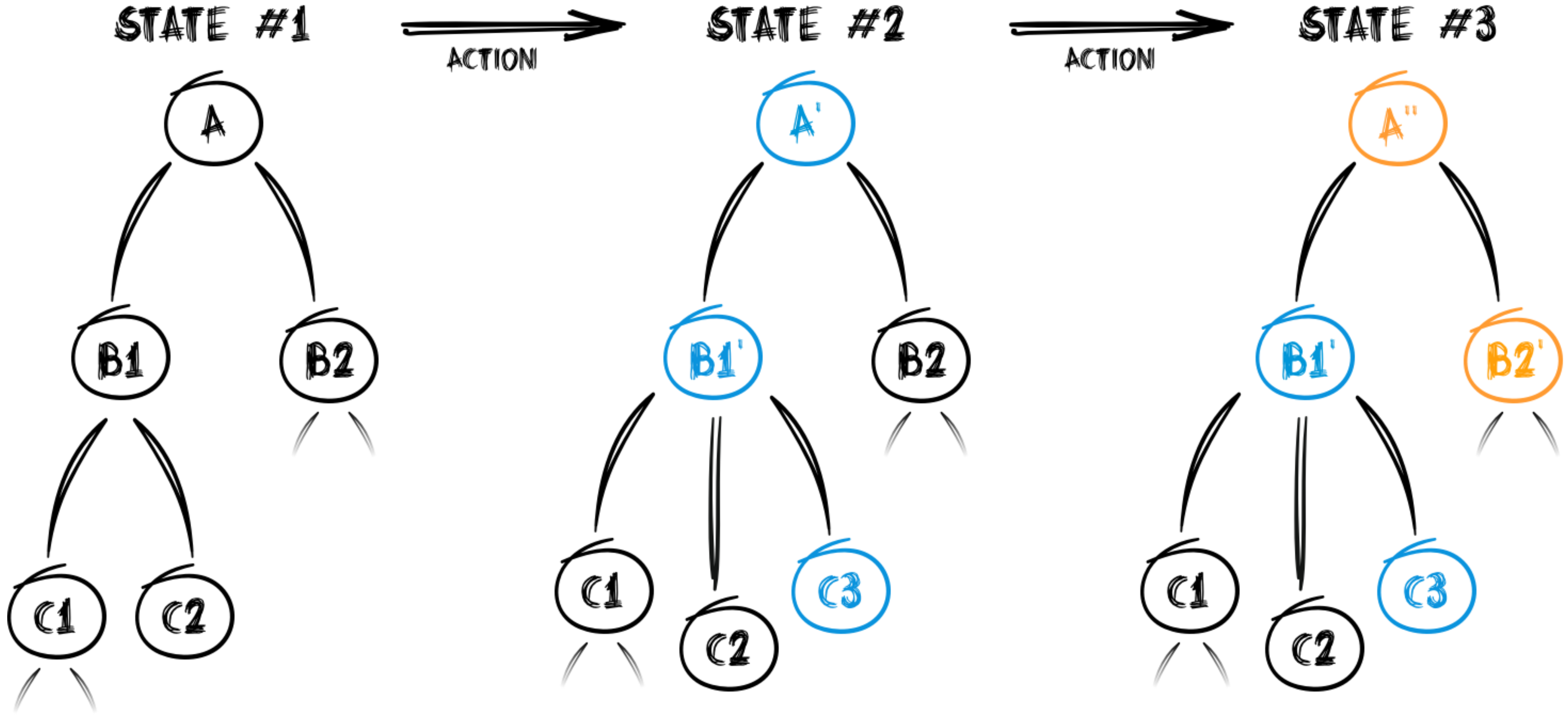


Previous state argument

- Specify default value
- Return same reference for irrelevant action type

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

## Reducer composition



## Store

**Single store for whole app managed by Redux** (we only provide a root reducer)

- Holds application state;
- Allows access to state via **getState**();
- Allows state to be updated via **dispatch(action**);
- Registers listeners via **subscribe(listener**);
- Handles unregistering of listeners via the function returned by subscribe(listener).

-- Redux docs

## Minimalistic API

- **createStore(rootReducer)**

- store.**getState**()
- store.**dispatch**(action)
- store.**subscribe**(listener)

- **combineReducers**({…})

- What is the **store lifecycle**?

→ initial call to reducer + call on every dispatched action

# Moving state to the Redux store

**GOAL**: No internal state in TodoApp.jsx

? How do we inject state to TodoApp component?
? How do we subscribe to changes?

## React-redux integration

You can connect your existing app to the store by hand.
But you would loose many optimizations react-redux package brings.

Use react-redux library instead:

1. Wrap your root component in **`<Provider>`**
2. Connect components to redux store
   - `connect(mapStateToProps, mapDispatchToProps)(Component)`

## Should all components be stateless?

*"How much" state should we move to the redux store?*

**Does your state influence more components in your application?**

→ (and the common parent is way up in the hierarchy)

→ move state to redux store

→ `TodoApp.jsx` – rendering number of items in navbar

→ `TodoItem.jsx` – if you want just one item to be editable at a time

**Is the state well encapsulated and local for the component?**

→ It can stay in the stateful component.

→ `TodoItemEdit.jsx` – temporary value of the input field

Kentico

## What about our props explosion?

```
<TodoList
    list={this.state.list}
    editedItemId={this.state.editedItemId}
    createNewFormVisible={this.state.createNewFormVisible}
    isDragging={this.state.isDragging}
    onDelete={this._deleteItem}
    onExpand={this._startEditing}
    onCancel={this._cancelEditing}
    onSave={this._updateItem}
    onReorder={this._moveItem}
    onCreateNewClick={this._showCreateNewForm}a
    onCreateCancel={this._hideCreateNewForm}
    onCreate={this._createNewItem}
    onDragStarted={this._itemDragStarted}
    onDragEnded={this._itemDragEnded}
/>
```

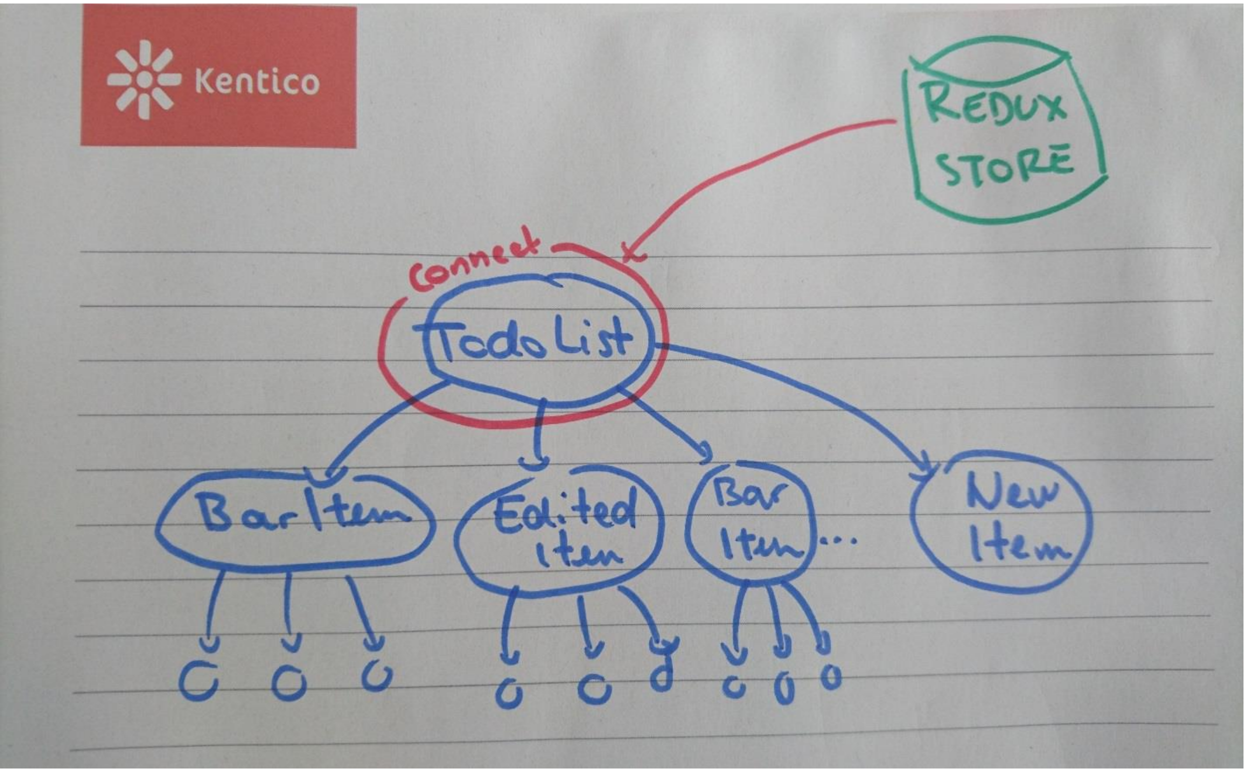
```
<TodoList
    list={this.props.list}
    editedItemId={this.props.editedItemId}
    createNewFormVisible={this.props.isCreateNewFormOpen}
    isDragging={this.props.isDragging}
    onDelete={this.props.onDelete}
    onExpand={this.props.onStartEditing}
    onCancel={this.props.onCancelEditing}
    onSave={this.props.onUpdate}
    onReorder={this.props.onMove}
    onCreateNewClick={this.props.onCreateNewClick}
    onCreateCancel={this.props.onCreateNewCancel}
    onCreate={this.props.onCreateNew}
    onDragStarted={this.props.onDragStarted}
    onDragEnded={this.props.onDragEnded}
/>
```

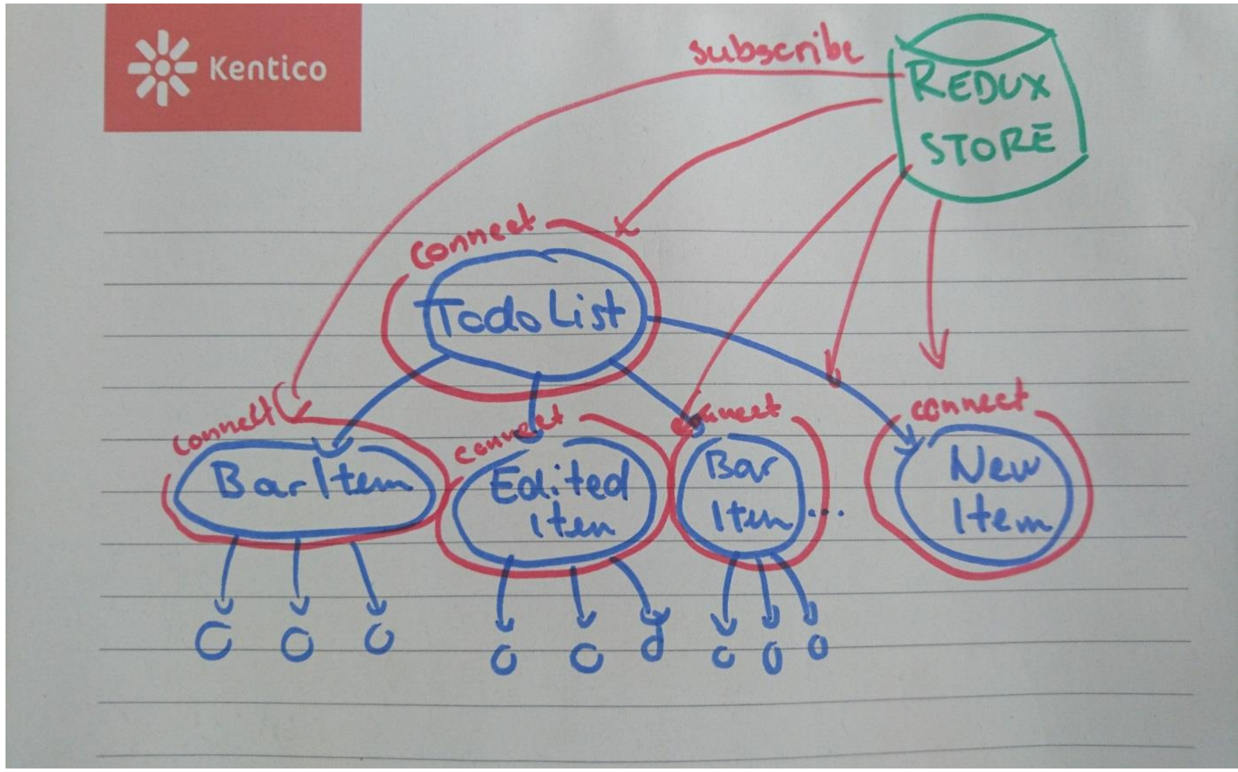
## Performance

Which components are re-rendered when we edit one todo item?
→ Whole app is re-rendered

How to fix this?

# Connecting more components





# Connecting more components to store

```
<TodoList
    list={this.state.list}
    editedItemId={this.state.editedItemId}
    createNewFormVisible={this.state.createNewFormVisible}
    isDragging={this.state.isDragging}
    onDelete={this._deleteItem}
    onExpand={this._startEditing}
    onCancel={this._cancelEditing}
    onSave={this._updateItem}
    onReorder={this._moveItem}
    onCreateNewClick={this._showCreateNewForm}a
    onCreateCancel={this._hideCreateNewForm}
    onCreate={this._createNewItem}
    onDragStarted={this._itemDragStarted}
    onDragEnded={this._itemDragEnded}
/>
```

```
<TodoList
    list={this.props.list}
    editedItemId={this.props.editedItemId}
    createNewFormVisible={this.props.isCreateNewFormOpen}
    onCreateNewClick={this.props.onCreateNewClick}
/>
```

# 3 Principles of Redux - recap

**Single source of truth:**
"The whole state of your app is stored in an object tree inside a single *store*."

**State is read-only:**
"The only way to change the state tree is to emit an *action*, an object describing what happened."

**Changes are made with pure functions:**
“To specify how the actions transform the state tree, you write pure *reducers*."

## Benefits

**State** described as plain object and arrays:
- Inject initial state during server rendering
- Persist to and load from localStorage
- UI is function of state (state -> UI -> deterministic behavior)
- Immutability (React performance)

**State changes** described as plain objects
- Replaying the history (reproducing bugs)
- Pass actions over network in collaborative environments (Google Docs, Trello live updates)
- Implementing undo
- Awesome tooling

**State modification** as pure functions
- Testability
- Hot reloading

**3rd party modules integration (middleware, libs that need to store state...)**

## Drawbacks

- **Boilerplate & Verbosity**

-> have a look at Repatch

- **"One huge object"**

-> pretty much eliminated by reducer composition and ImmutableJS

- **"Component state vs Redux store" dillema**

-> see #1287 and: "*Do whatever is less awkward.*"

# Be declarative

**Action** describes **what** has happened, **reducer** decides **how** to react

```
const editedItemId = (state = null, action) => {
  switch(action.type) {
    case TODO_LIST_ITEM_START_EDITING:
      return action.payload.id;

    case TODO_LIST_ITEM_CANCEL_EDITING:
    case TODO_LIST_ITEM_UPDATE:
    case TODO_LIST_ITEM_DELETE:
      return null;

    default:
      return null;
  }
};
```

```
dispatch({
    type: 'SET_EDITED_ITEM_ID',
    payload: {
        id: 42
    }
});



dispatch({
    type: 'CLEAR_EDITED[illegible]D'
});
```

## Task

```
git clone https://github.com/KenticoAcademy/PV247-2018.git
cd PV247-2018
git checkout -b solution-1 redux-task-1
cd 05-redux
npm install
npm start
```

## Task

1. **Implement removeTodo action**
    a) Action type
    b) Action creator
    c) Handling in reducer
    d) Connect TodoItem
2. **Implement # of todos in the navigation**
    a) Component capable of rendering number
    b) Connect component and pass number of todos
    c) Render container component in app menu
3. **[Bonus] Make sure only one item at a time can be editable**
    a) You need to store editedItemId in store (todoApp)

## Redux vol 2. - advanced stuff

- Normalization, memorization, selectors…
- Optimizing performance
- Async action - communicating with API
- How to cleverly structure your state