Kentico

Redux

"... predictable state container for JavaScript apps." -- Redux docs

Zuzana Dankovčíková

Kentico

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**.

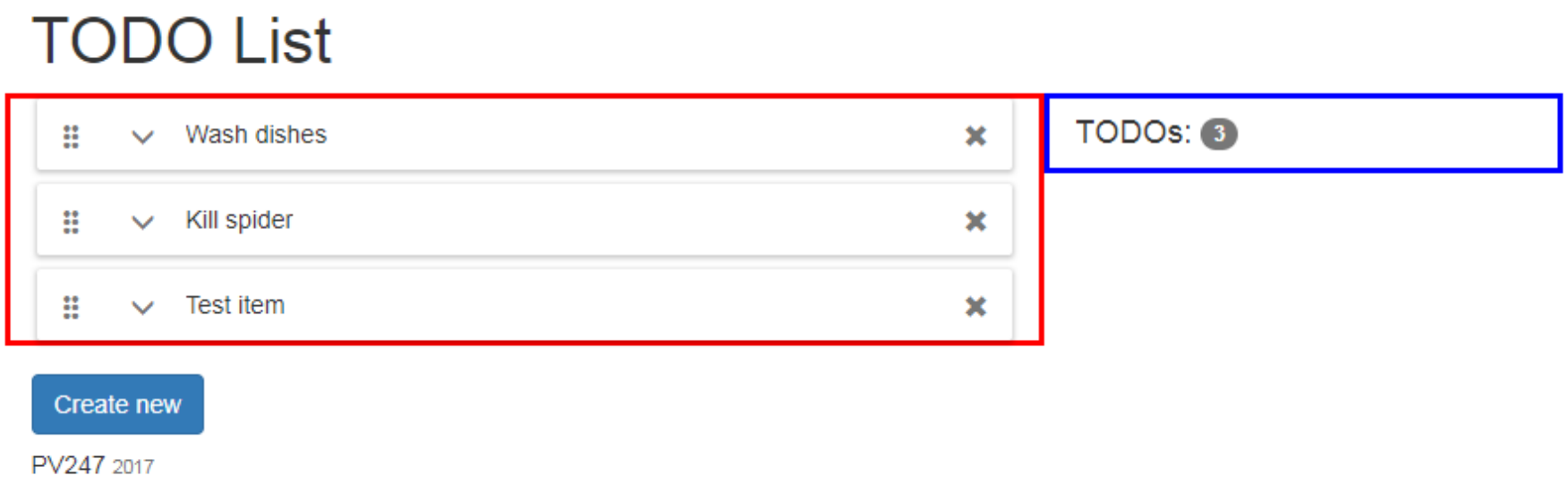
Problem 1: What is “root component”

New feature request:

→ Displaying number of TODOs next to the avatar of the logged-in user?

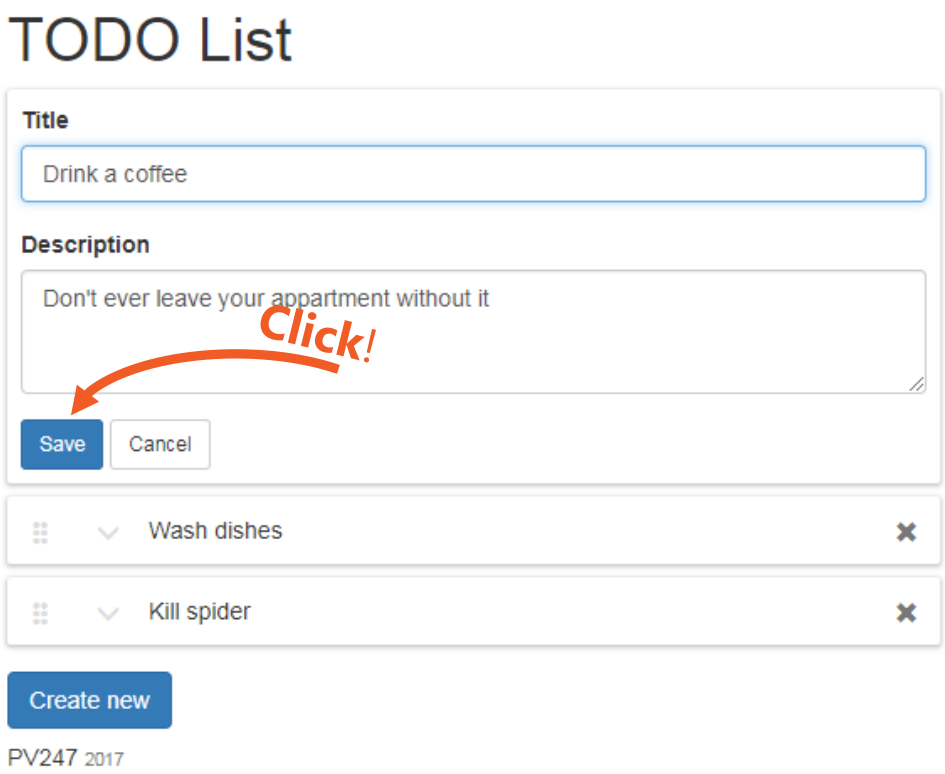
→ **“Unrelated” components dependent on the same data.**

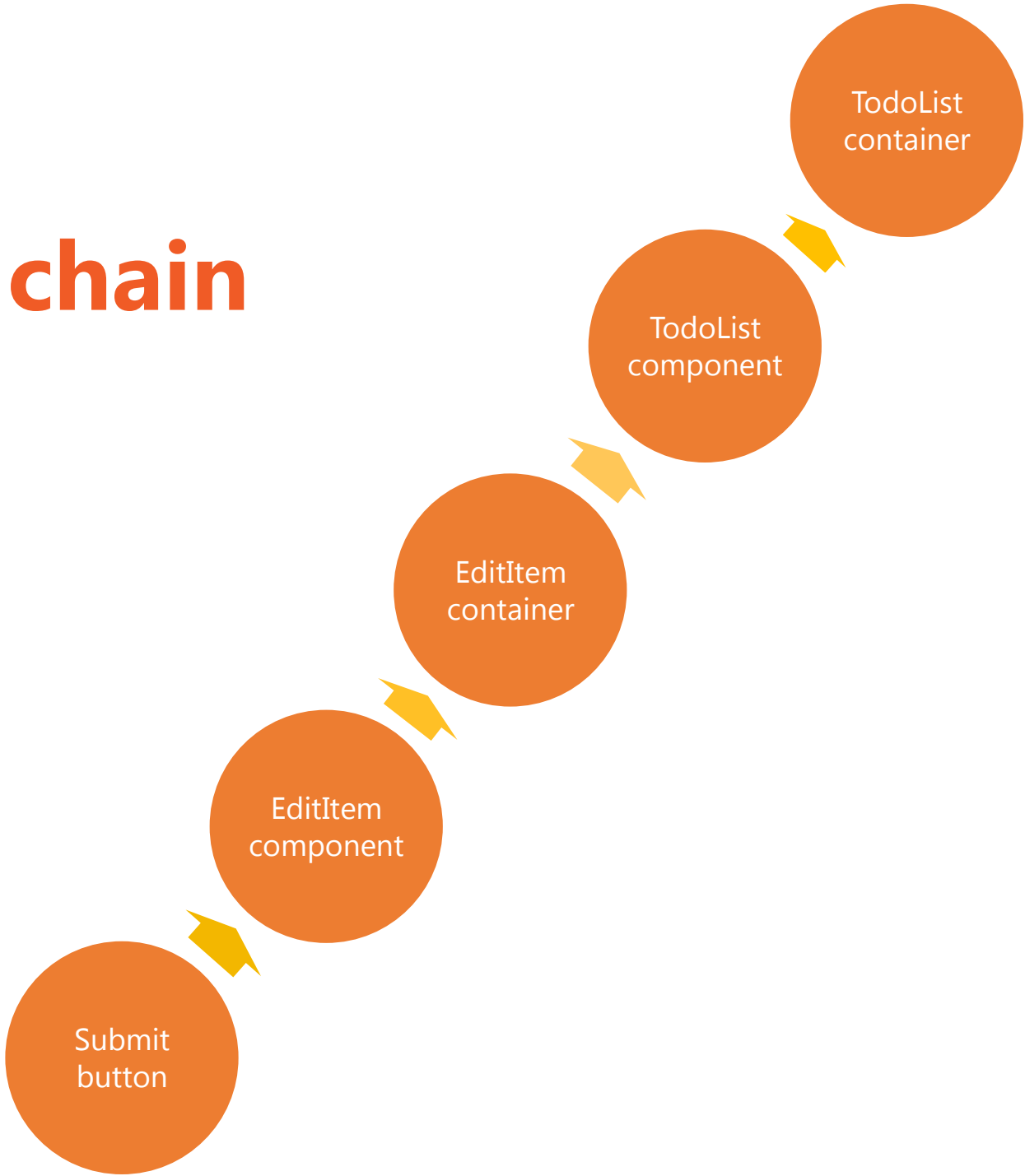
→ [Lifting state up](). But until when? How to make it scalable?



Kentico

Problem 2: Callbacks chain

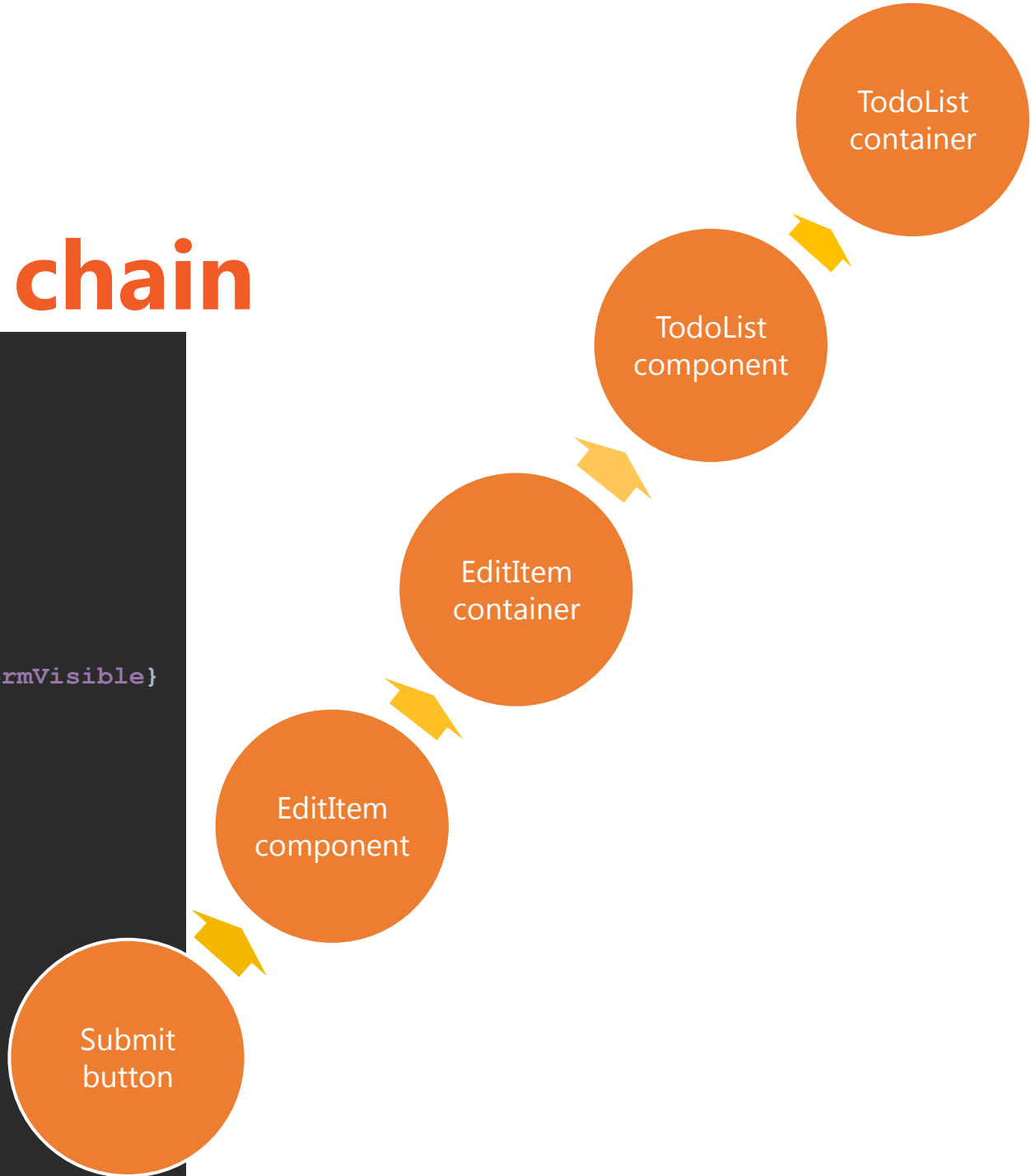




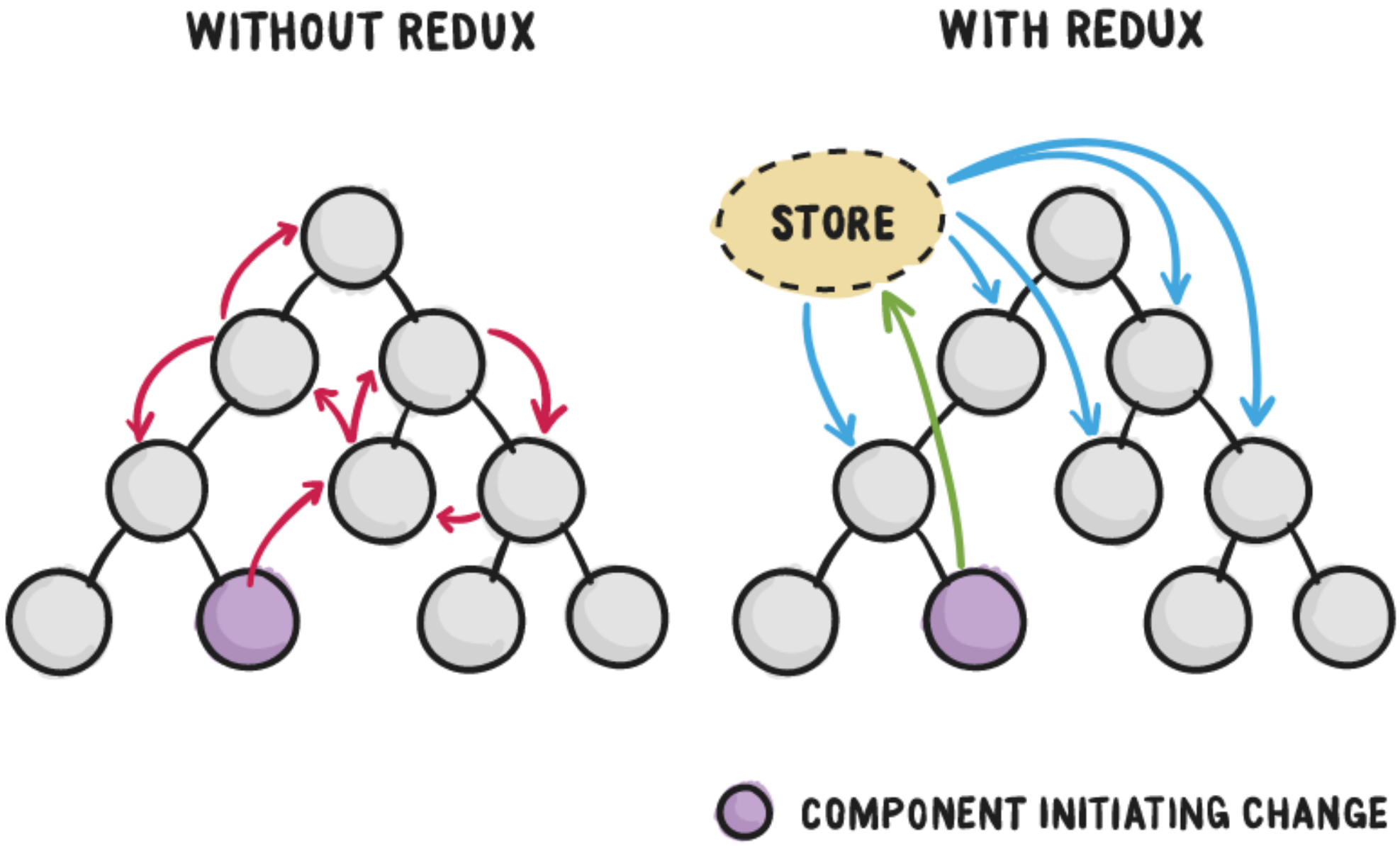
Kentico

Problem 2: Callbacks chain

```
class TodoListContainer extends React.Component {

    // other methods
    // ...

    render() {
        return (
            <TodoListComponent
                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


WITHOUT REDUX
WITH REDUX
STORE
COMPONENT INITIATING CHANGE

Kentico

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**

Kentico

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*."

Kentico

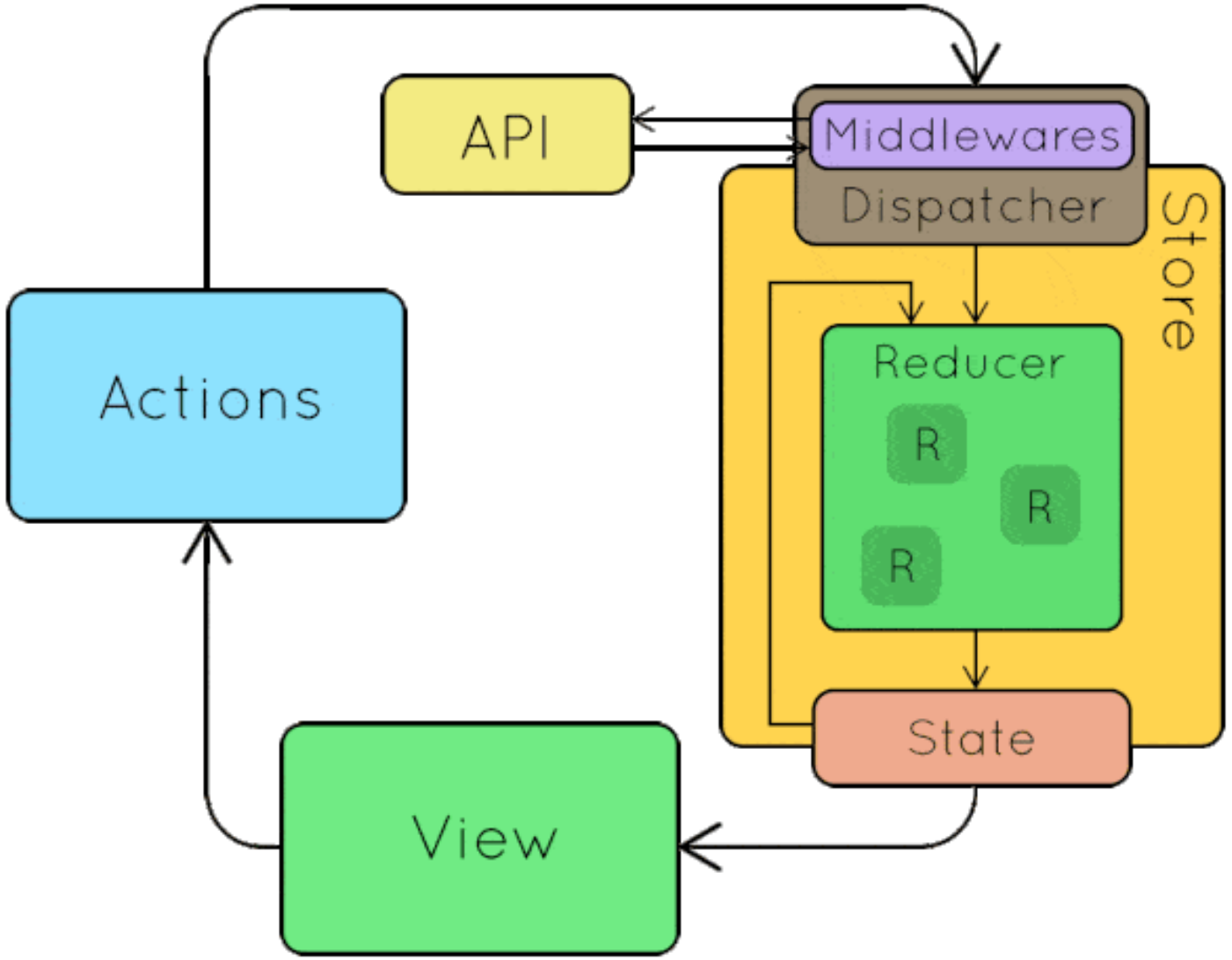
Building block

Action
- describes UI changes

Store
- receives action via dispatcher
- calls root reducer

Reducer
- (prevState, action) => newStat

View
- gets notified about state chan
- rerenders 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 user interactions that are possible in your app.

Action - simple JS objects describing data change

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

Action creator - helper function for creating actions

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

Kentico

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;
cosnt increaseCount = (val) => count += val;
```

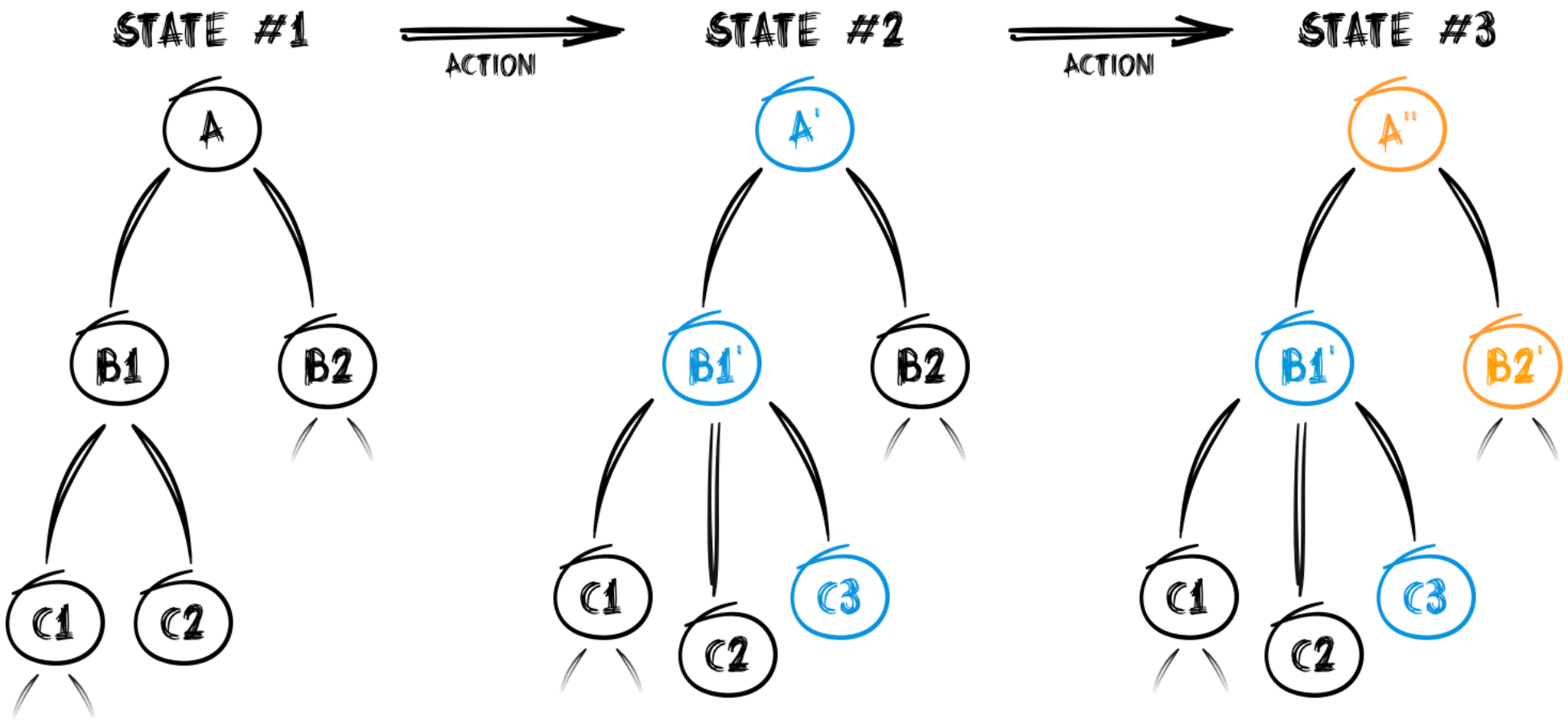
Reducers

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



Kentico

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

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)`

Moving more state to the Redux store

All state from the root component shall be moved to the store

- New actions,
- New reducers
- No internal state in TodoList.jsx

→ the old container is basically useless

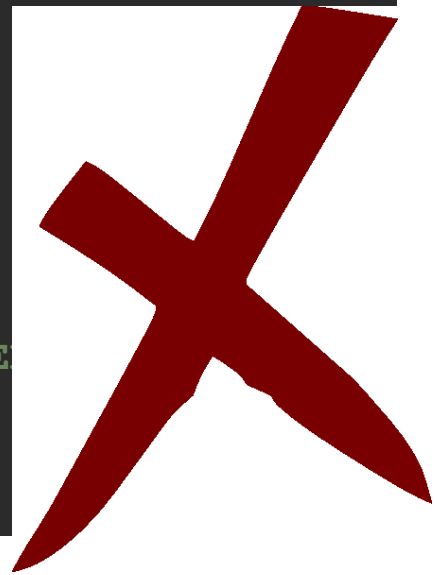
Kentico

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
    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_E
});
```



Kentico

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
→ `TodoList.jsx`

Is the state well encapsulated and local for the component?

→ It can stay in the stateful component.
→ `TodoListEditedItem.jsx`

Kentico

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...)

Kentico

Drawbacks

- **Boilerplate & Verbosity**

-> have a look at Repatch

- **"One huge object"**

-> pretty much eliminated by reducer composition and ImmutableJS

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

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

Kentico

3 Principles of Redux - revised

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*."

Part 2

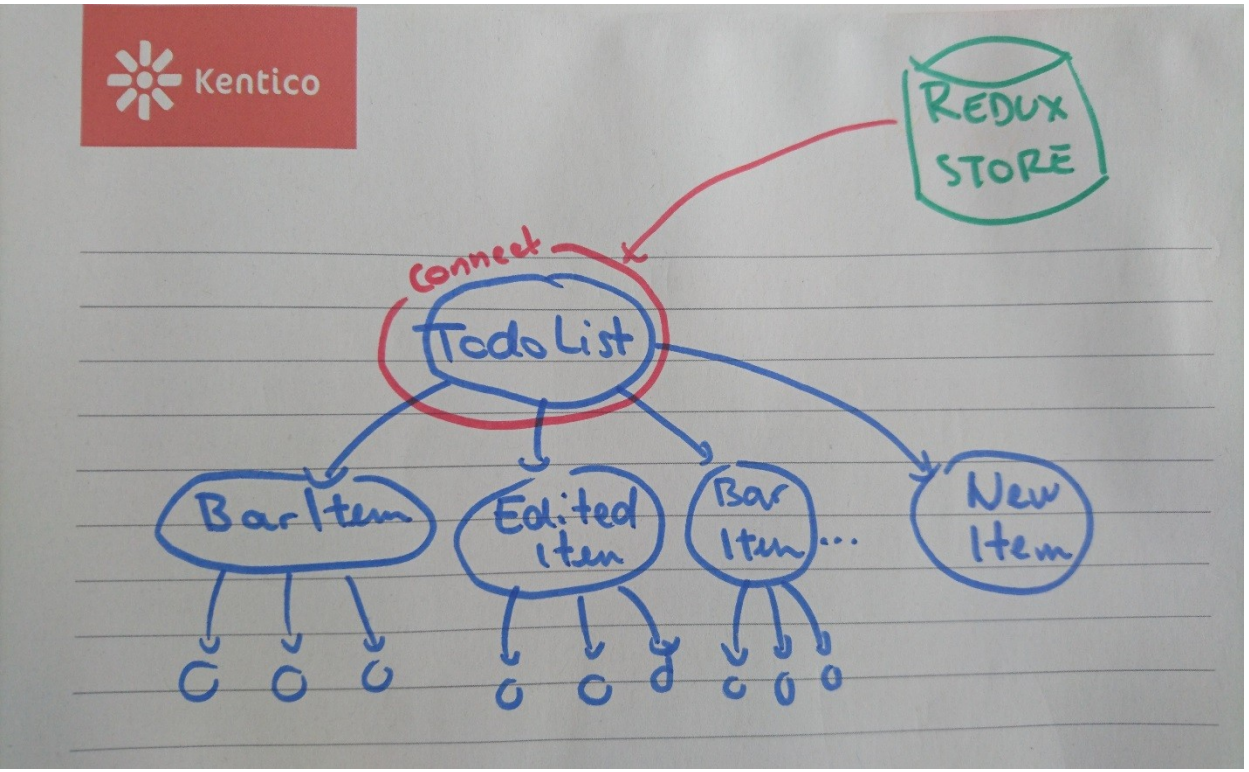
Kentico

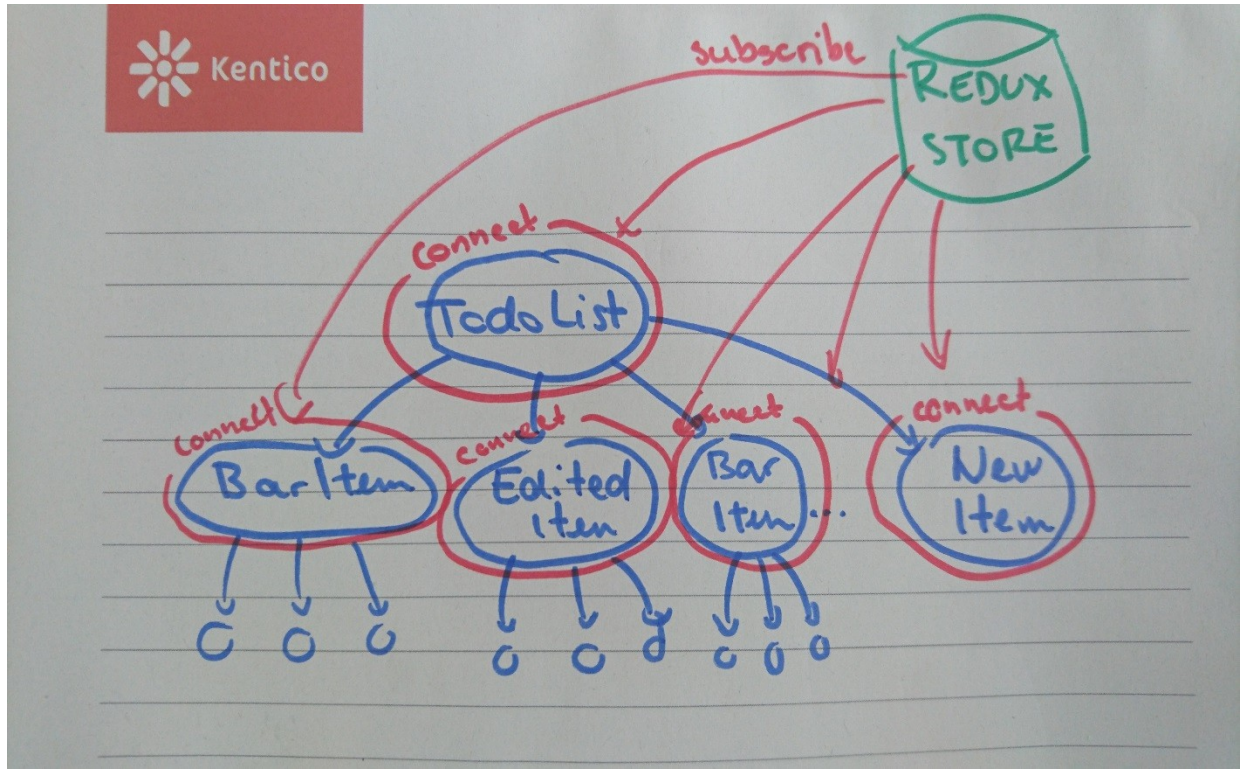
What about our props explosion?

```
<TodoListComponent
    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}
/>
```

```
<TodoListComponent
    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}
/>
```

Connecting more components





Connecting more components to store

```
<TodoListComponent
    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}
/>
```

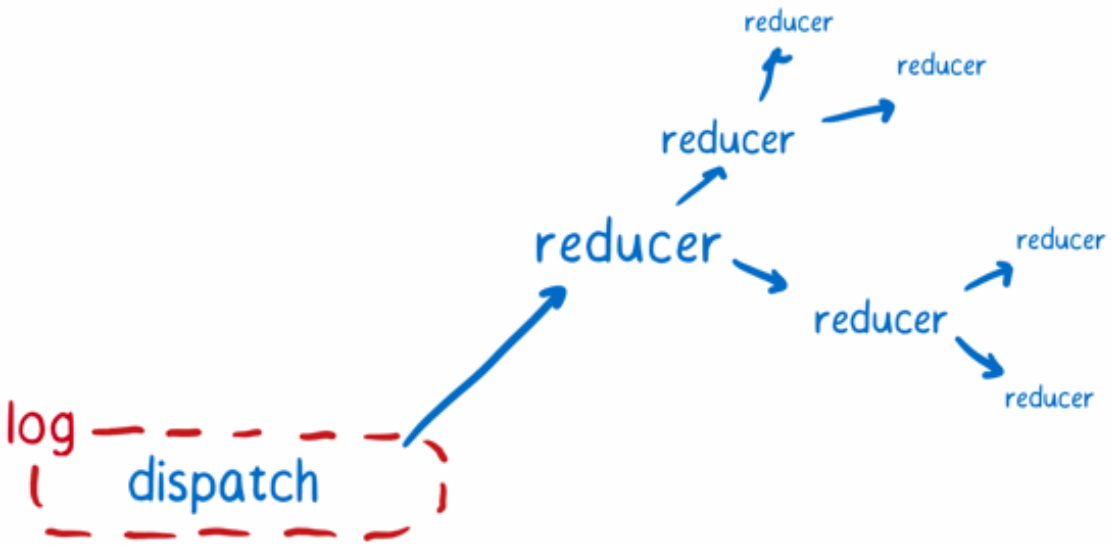
```
<TodoListComponent
    list={this.props.list}
    editedItemId={this.props.editedItemId}
    createNewFormVisible={this.props.isCreateNewFormOpen}
    onCreateNewClick={this.props.onCreateNewClick}
/>
```

Middleware

One of the greatest things about Redux is its modularity

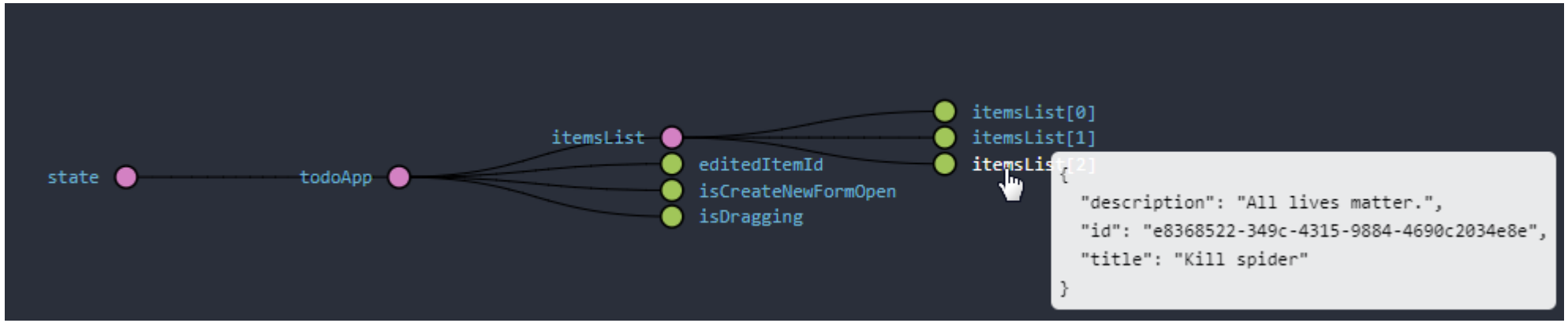
```
createStore(app, initialState, applyMiddleware(...middleware));
```

- Logging
- Complex actions (Thunk, promise)
- devTools
- …



Redux-devtools

- All your actions and state visualized
- You can replay history
- Install chrome extension
- See kentico cloud or kiwi.com



Kentico

Redux-thunk

Where to handle side-effects in Redux app?
(**async code** (API communication), **data generation** like `new Date()` or `Math.random()`)

- Components?
- Reducers?
- Action creators?

→ “thunk” action creators

Kentico

Thunk actions

"In computer programming, a **thunk** is **a subroutine used to inject an additional calculation into another subroutine**. Thunks are primarily used to **delay a calculation** until it is needed, or to **insert operations at the beginning or end of the other subroutine**."

-- Wikipedia

Function that can dispatch other actions:

```
export const saveItems = () =>
    (dispatch, getState) => {
        dispatch(savingStarted());
        setTimeout(() => {
            const items =  JSON.stringify(getState().todoApp.itemsList.toJS());
            localStorage.setItem('todoList', items);
            dispatch(savingFinished());
        }, 1000);
    };
```

Saving items to localStorage

Getting rid of Dummy `TodoList` container

New component `<SavingStatus />`

- Displays saving status
- Watches for changes in `todoList` part of state
- On changes dispatches a thunk action to save items

✓ Install redux-thunk

✓ Define `savingStarted` & `savingFinished` action types and creators

✓ Introduce reducer with saving flag

✓ Create `SavingStatus` component

✓ Wrap component in a container (`list` data & `save` callback)

Kentico

Data normalization

```
Immutable.List<Item>
```

vs.

```
Immutable.Map<id, Item> & Immutable.List<ic
```

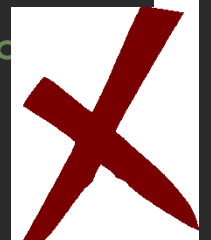
Data should be stored in a normalized form (same as in relation DB)

- ✓ **Easier manipulation** – reducers (entity vs collection)
- ✓ **No duplication** (for complex nested objects)

Kentico

Data normalization

```
{
  itemsWithAuthors: [
    {
      id: '1',
      title: 'Buy milk',
      author: { id: '410237', name: 'Suzii' },
    },
    {
      id: '2',
      title: 'Learn Redux',
      author: { id: '410237', name: 'Suzii' },
    },
    {
      id: '3',
      title: 'Be awesome',
      author: { id: '325335', name: 'Slavo
    },
  ],
};
```



```
{
  authors: {
    byId: {
      '410237': { id: '410237', name: 'Suzii' },
      '325335': { id: '325335', name: 'Slavo' },
    },
  },
  items: {
    allIds: ['1', '2', '3'],
    byId: {
      '1': {
        id: '1',
        title: 'Buy milk',
        author: '410237',
      },
      '2': {
        id: '2',
        title: 'Learn redux',
        author: '410237',
      },
      '3': {
        id: '3',
        title: 'Be awesome',
        author: '325335',
      },
    },
  },
}
```

Normalizing todo list

We replace the itemsList with data structure:

```
{
  items: {
    allIds: [], // list of ids
    byId: {}, // map of items indexed by id
  }
}
```

```
▼ todoApp (pin)
  ▼ items (pin)
    ▼ allIds (pin)
        0 (pin): "8b803c50-05a2-4a15-b752-2f08d70f14ac"
        1 (pin): "3f762052-9a42-4a5d-865f-1023d79ed0b4"
    ▼ byId (pin)
      ▼ 8b803c50-05a2-4a15-b752-2f08d70f14ac (pin)
          id (pin): "8b803c50-05a2-4a15-b752-2f08d70f14ac"
          title (pin): "Wash dishes"
          description (pin): "Not again!"
      ▼ 3f762052-9a42-4a5d-865f-1023d79ed0b4 (pin)
          id (pin): "3f762052-9a42-4a5d-865f-1023d79ed0b4"
          title (pin): "Kill spideraaaaaa"
          description (pin): "All lives matter"
    editedItemId (pin): null
    isCreateNewFormOpen (pin): false
    isDragging (pin): false
    isSaving (pin): false
```

Memoization

What do we pass to TodoList container?

Two options:

- Both `byId` and `allIds`
- We create list of item in container and do not need to change the component at all

```
const getListOfItems = (items) => items.allIds.map(id => items.byId.get(id)).toList();
```

But we are creating new instance of list every time `mapStateToProps` is called

→ ANY change in state,

→ The component is ALWAYS rerendered

→ **MEMOIZE**

```
const getListOfItemsMemoized = memoizee(getListOfItems);
```

Kentico

Unit testing

Action creators:

- Very easy to test, however, most of the times unnecessary

Thunk action creators:

- If you inject your dependencies → easy to test

Reducers:

- Pure functions → super-easy to test

MapStateToProps/Selectors (reselect library)

- Should be a pure function mapping data from store to another data structure → easy to test

Kentico

Interesting libraries, concepts

Redux is widely used in the community and there are tons of other packages that work with it.

Integration with React: react-redux
React router: react-router-redux
Forms: redux-form
Computing derived data: reselect
Memoizing: memoizee
Normalizing data from server: normalizr
Middleware: redux-logger, redux-thunk

And lots more...

Kentico

Alternatives

Flux

- "It is cool that you are inventing better Flux by not doing Flux at all." – reduxjs.org
- More stores, dispatcher entity, action handlers

RePatch

- Redux with less boilerplate

MobX

- Functional reactive programming

Others

- There are new libraries every day

Sources

http://redux.js.org
https://css-tricks.com/learning-react-redux/
https://code-cartoons.com/a-cartoon-intro-to-redux-3afb775501a6

Questions?

zuzanad@kentico.com
410237@mail.muni.cz