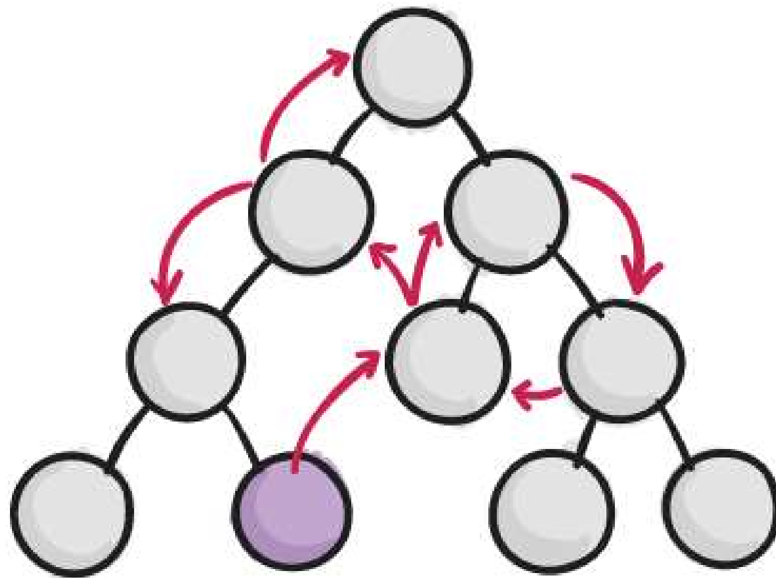
Redux vol. 2

Zuzana Dankovčíková

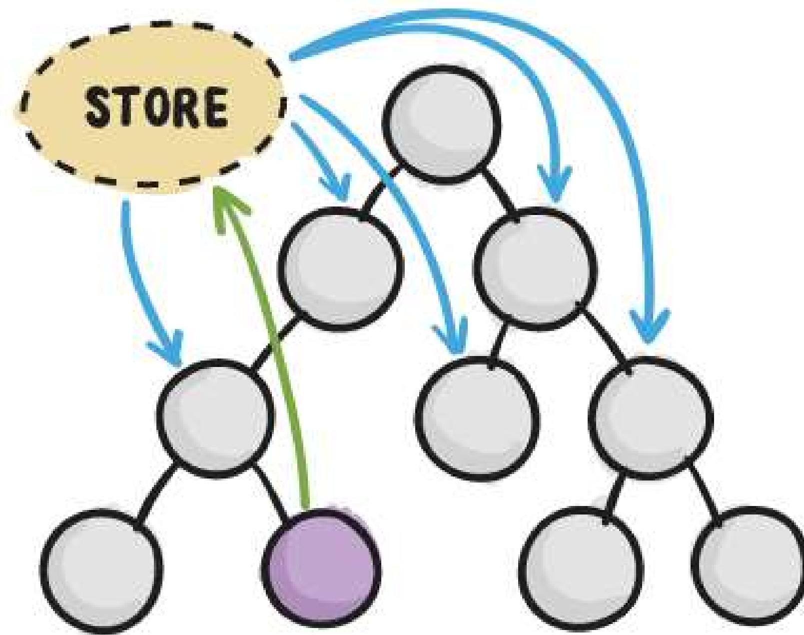
https://redux.js.org/

Kentico

WITHOUT REDUX



WITH REDUX



Kentico

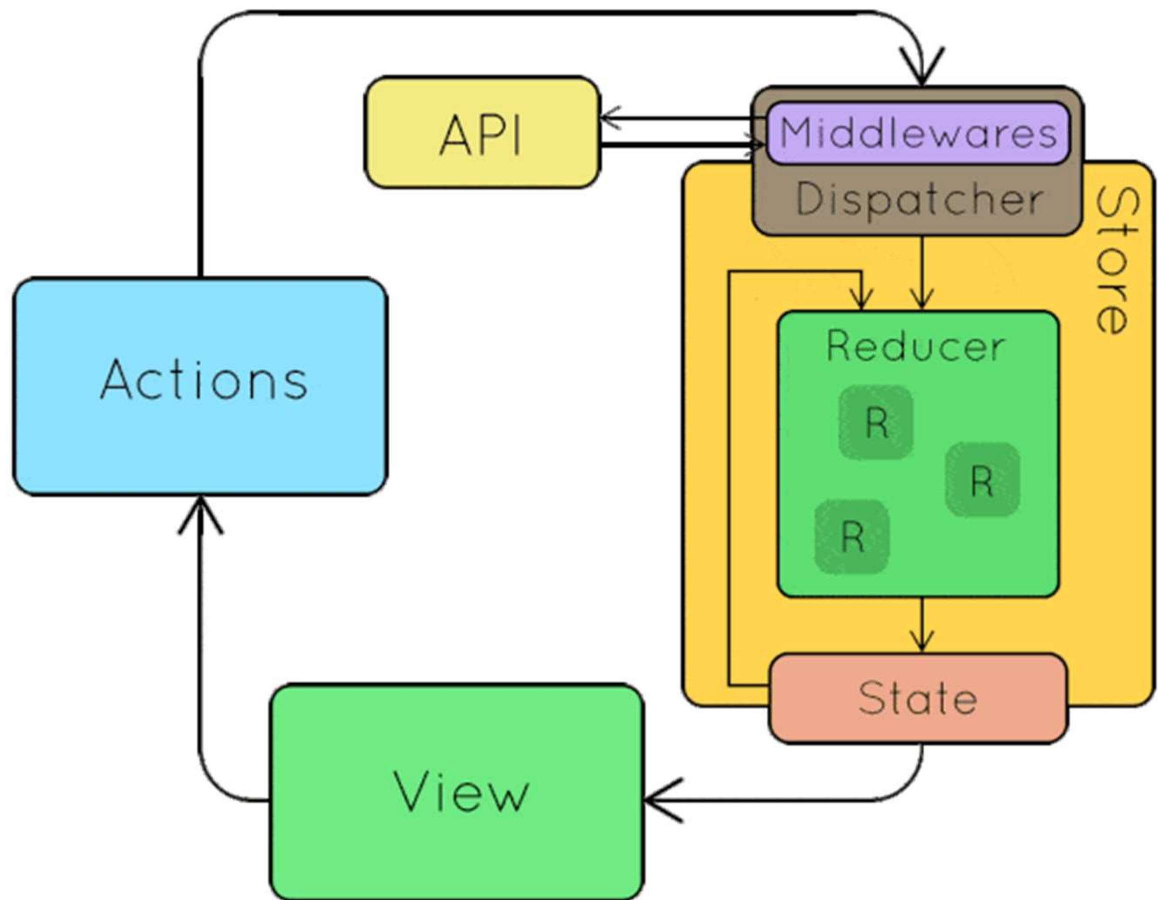
Building blocks

Action
- describes UI changes

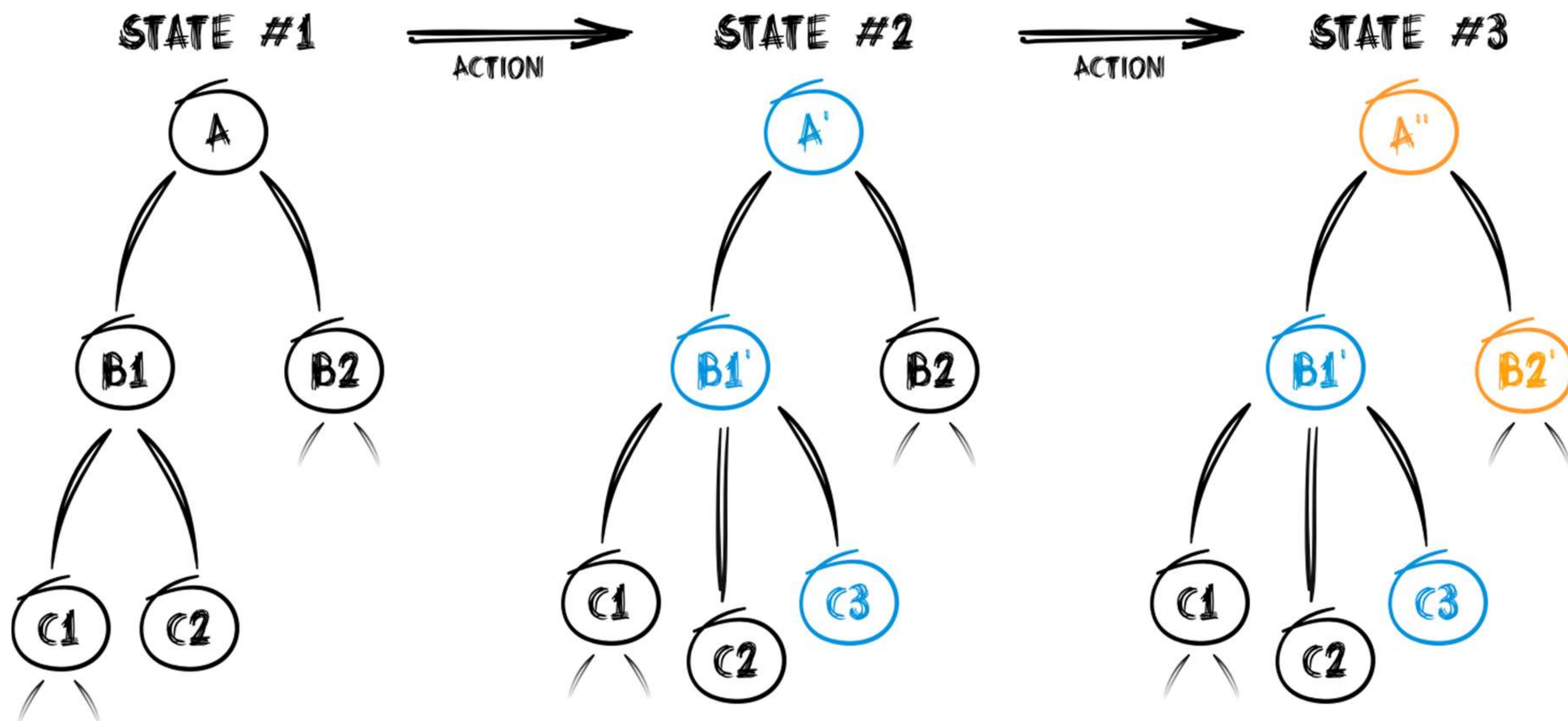
Store
- receives action via dispatcher
- calls root reducer

Reducer
- (prevState, action) => newState

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



Reducer composition



Today

- Middleware & devtools
- Normalization, memoization, selectors…
- Async action, communicating with API
- Exercises

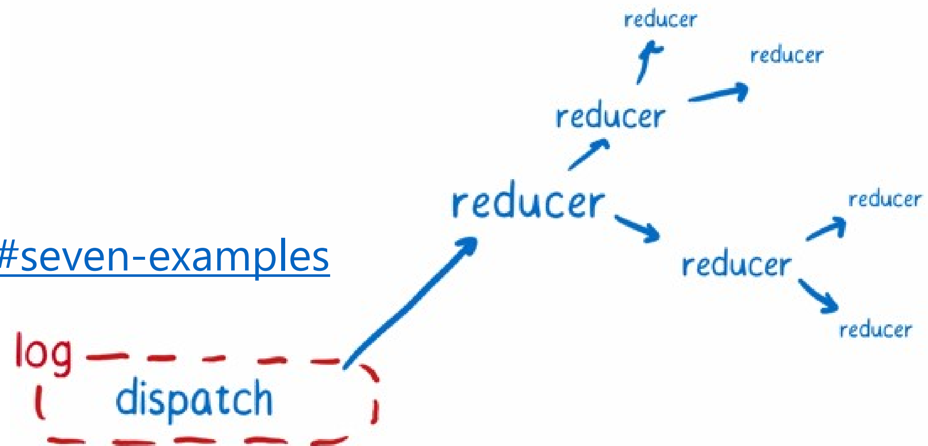
Middleware

One of the greatest things about Redux is its modularity

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

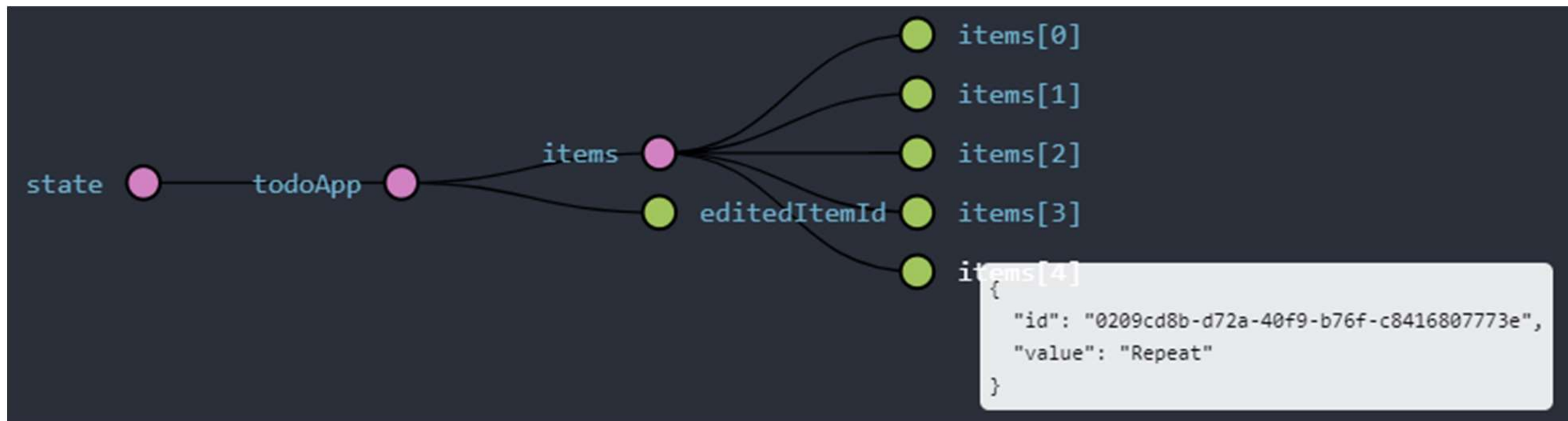
- Logging
- Complex actions (Thunk, promise)
- devTools
- …

https://redux.js.org/advanced/middleware#seven-examples



Redux-devtools

- All your actions and state visualized
- You can replay history, export & inject state
- Install [chrome extension]
- See [kentico cloud] or [kiwi.com]



Kentico

Data normalization

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

vs.

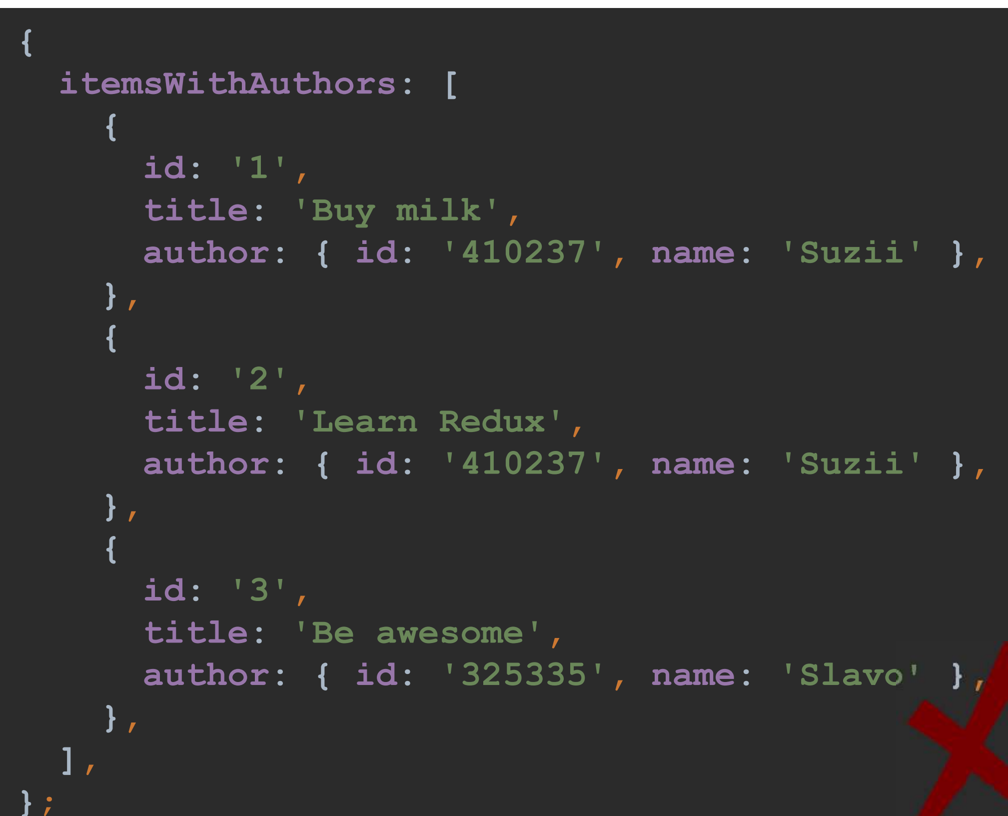
```
Immutable.Map<id, Item> & Immutable.List<id>
```

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



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): "2cba6d20-10ca-4e34-b282-befe81405e24"
        1 (pin): "f008cb8e-0bc4-48f6-bd82-b1a06801ef72"
    ▼ byId (pin)
      ▼ 2cba6d20-10ca-4e34-b282-befe81405e24 (pin)
          id (pin): "2cba6d20-10ca-4e34-b282-befe81405e24"
          value (pin): "Make a coffee"
      ▼ f008cb8e-0bc4-48f6-bd82-b1a06801ef72 (pin)
          id (pin): "f008cb8e-0bc4-48f6-bd82-b1a06801ef72"
          value (pin): "Code all day"
  editedItemId (pin): null
```

And refactor containers structure:

- TodoList
- TodoItem
- NewTodo

React ‘key’ p

Use **unique** identifier for each item rendered in list

https://reactjs.org/docs/lists-and-keys.html

The identifier for one specific item should not change.

Delete action:
Index vs Id matters.

```
{this.props.todoIds.map((item, index) =>
  <TodoItemContainer
    key={Math.random()}
    item={item}
  />
)}
```

```
{this.props.todoIds.map((item, index) =>
  <TodoItemContainer
    key={index}
    item={item}
/>
)}
```

```
{this.props.todoIds.map((item, index) =>
  <TodoItemContainer
    key={item.id}
    item={item}
/>
)}
```

Selectors

New functionality:

- Toggling todos (isComplete flag)
- VisibilityFilter component (state stored in redux)

Motivation:

- We never want to store duplicate or derived data in redux store
- How do we display only items that match the Visibility filter?
- We calculate the data for render on demand

→ **Selectors**

```
const getVisibleTodoIds = (state: IState): Immutable.List<Uuid> => {…}
```

Generally, should be placed next to reducers.

Memoization

Selecting list of visible todos TodoList component on each state change.

```
const getVisibleTodoIds = (state) => state.todoApp.allIds.filter(…).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 getVisibleTodoIdsMomoized = memoizee((state) => {…})
```

Why isn't it working?

- We state changes all the time, we memorize on a wrong argument.

```
const getVisibleTodoIdsMomoized = memoizee((visibilityFilter, allIds, byId) => {…})
```

Kentico

Memoization (generally)

Computation-heavy tasks, dynamic programming...
React? -> preventing unnecessary re-renders of PureComponents when props are computed

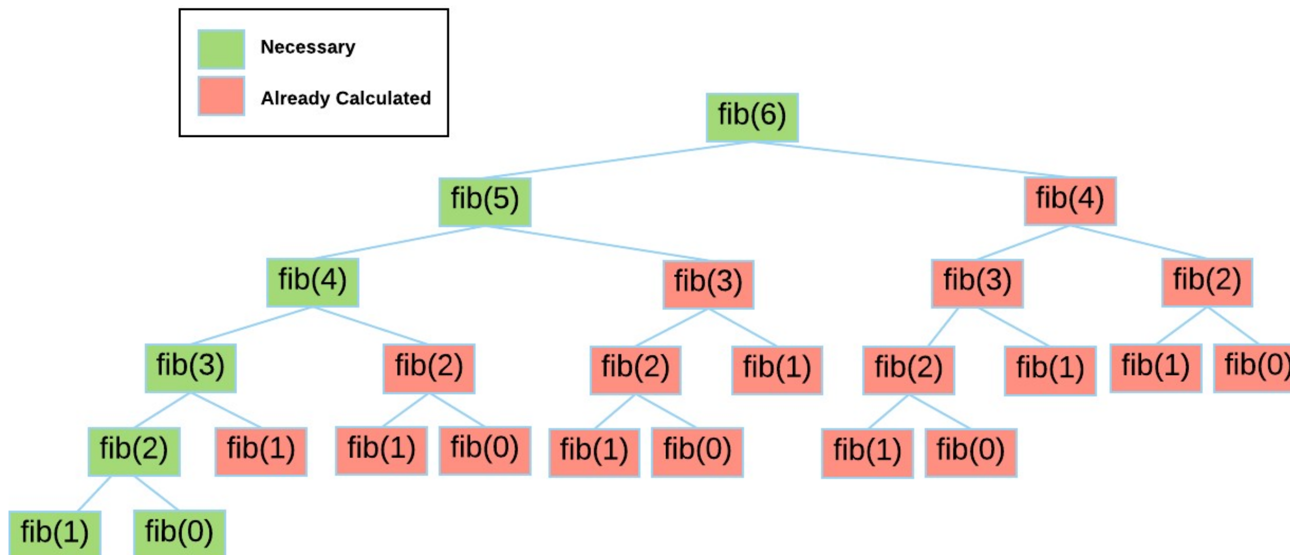


Fig: Fibonacci Number Recursive Implemention

Reselect

```
import { createSelector } from 'reselect';
```

```
const getVisibleTodoIds = createSelector(
  [
    state => state.todoApp.visibilityFilter,
    state => state.todoApp.items.allIds,
    state => state.todoApp.items.byId,
  ],
  (visibilityFilter, allIds, byId) => {
    switch (visibilityFilter) {
      case TodoFilter.All:
        return allIds;

      case TodoFilter.Done:
        return allIds.filter((id: Uuid) => byId.get(id).isCompleted).toList();

      case TodoFilter.Todo:
        return allIds.filter((id: Uuid) => !byId.get(id).isCompleted).toList();

      default:
        throw new Error(`Unknown value of visibility filter '${visibilityFilter}'`);
    }
  });
```

Communicating with API

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

Thunk actions (redux-thunk)

"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

https://redux.js.org/advanced/middleware#seven-examples

Function that can dispatch other actions:

```
export const loadTodos = () =>
  async (dispatch) => {
    dispatch(loadingStarted());
    const todos = await getTodos();
    dispatch(loadingSuccess(todos));
  };
```

For async actions generally 3 separate actions

```
'TODO_APP_LOADING_STARTED'
'TODO_APP_LOADING_SUCCESS'
'TODO_APP_LOADING_FAILED'
```

Showing loader

We want to display loading while items are being fetched via API:

1. isLoading reducer
2. Pass isLoading flag to TodoApp.tsx component
3. In TodoApp.tsx
   - if isLoading then displayLoader

Updating Todo via API

- Action types
- Thunk action + simple actions for start/success/fail

```
export const updateItem = (id: Uuid, text: string) =>
  async (dispatch, getState) => {
    dispatch(updateTodoStarted());

    const oldTodo = getState().todoApp.items.byId.get(id);
    const todo = await updateTodoApi({ ...oldTodo, text });

    dispatch(updateTodoSuccess(todo));
  };
```

- Update reducers to react to new actions
- Give container new action to be dispatched

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

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…

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

React ContextAPI
- https://daveceddia.com/context-api-vs-redux/

Sources

http://redux.js.org
https://css-tricks.com/learning-react-redux/
https://code-cartoons.com/a-cartoon-intro-to-redux-3afb775501a6
https://medium.com/@ohansemmanuel/how-to-eliminate-react-performance-issues-a16a250c0f27

Questions?

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

Project-related questions?

Task

```
git clone https://github.com/KenticoAcademy/PV247-2018.git
cd PV247-2018
git checkout -b solution-2 redux-task-2
cd 05-redux
npm install
npm start
```

Kentico

Task

1. **Implement createItem() via API** (similar to updateItem)
   a) Action types
   b) Thunk action creator
   c) Handling in reducer
   d) Connecting NewTodo
2. **Implement toggleItem() via API** (reuse basic actions from updateTodo)
   a) Implement new thunk actions dispatching same stuff as updateItem does
   b) Remove old action type
   c) Connect TodoItem correctly
3. **[Bonus] Show loader while performing async action**
   a) Reacting to all *_START and *_SUCCESS actions with some reducer
   b) Simple solution can reuse isLoading reducer
4. **[Bonus] Implement deleteCompletedItems() via API**
5. **[Bonus] Make TodosCountBadge display count of not completed items**