React Redux

01.02.2017

Konstantin Sabadir

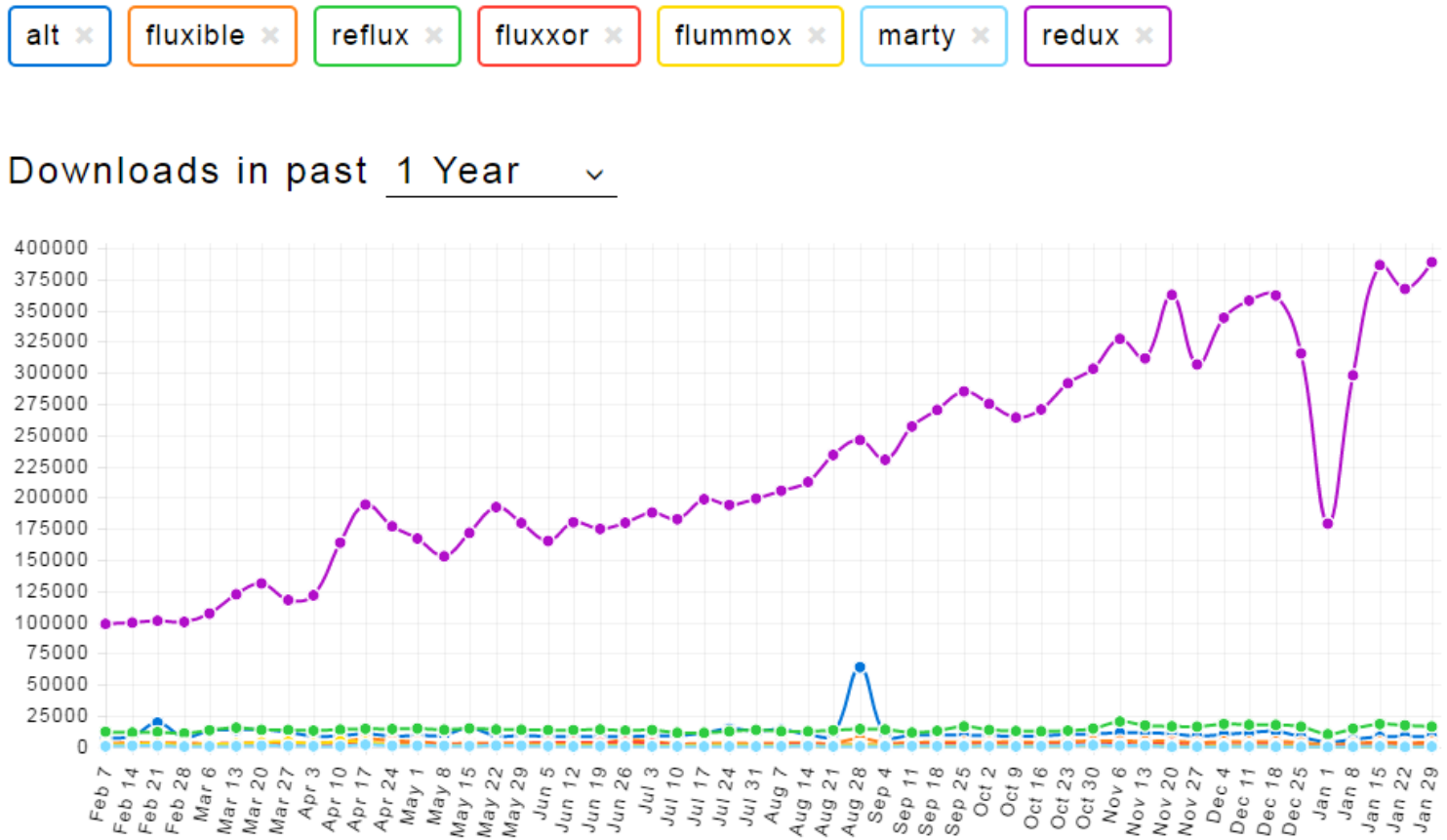
Warum Redux?

Redux ist ein App-Zustands-Container, inspiriert durch Flux und Elm. Einsteigen in die Technologie ist sehr einfach und gleichzeitig ist bringt sie großes Nutzen.

- Hat Unterstützung für Hot Reloading und Time Travel Debugger.
- Frei den Fehlern und Komplexität von Flux - durch Nutzung eines einzelnen Containers und der sog. Reducers.
- Fördert die Nutzung von immutabler Datenstrukturen, was sowohl direkten Einfluss auf Effizienz hat, als auch das Debuggen vereinfacht.
- Es kann mit einer beliebigen Bibliothek oder Framework verwendet werden – ausser React auch mit Angular oder vue.js.
- Ist frei von komplizierten Abhängigkeiten zwischen Daten, die bei größerer Anzahl der Container (wie in Alt.js) auftreten.
- Ist populär und wird von der Mehrheit der Teams gewählt, die Apps in React entwickeln.

Popularität

Redux ist heutzutage der De-facto-Standard in der React-Welt.



Three Principles

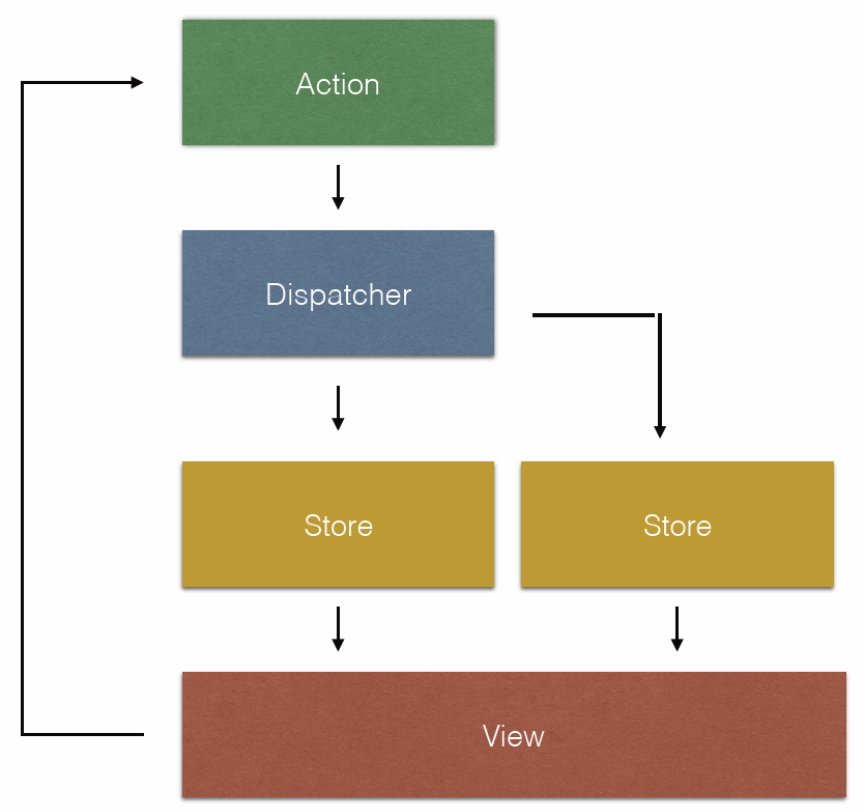
- **Single source of truth**
  Der Zustand der Anwendung wird zentral über einen einzigen *Store* gemanagt.

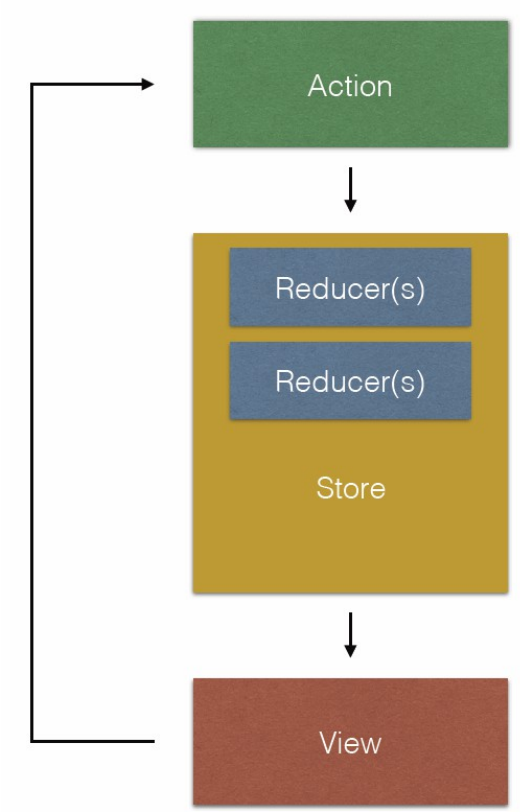
- **State is read-only**
  Die Anwendung reagiert auf Benutzerinteraktionen, indem sie mit einem Action-Creator eine Action erzeugt, die anzeigt, was gerade passiert ist. Mit einem Reducer wird dann abhängig von der Action aus dem alten Zustand ein neuer Zustand erzeugt.

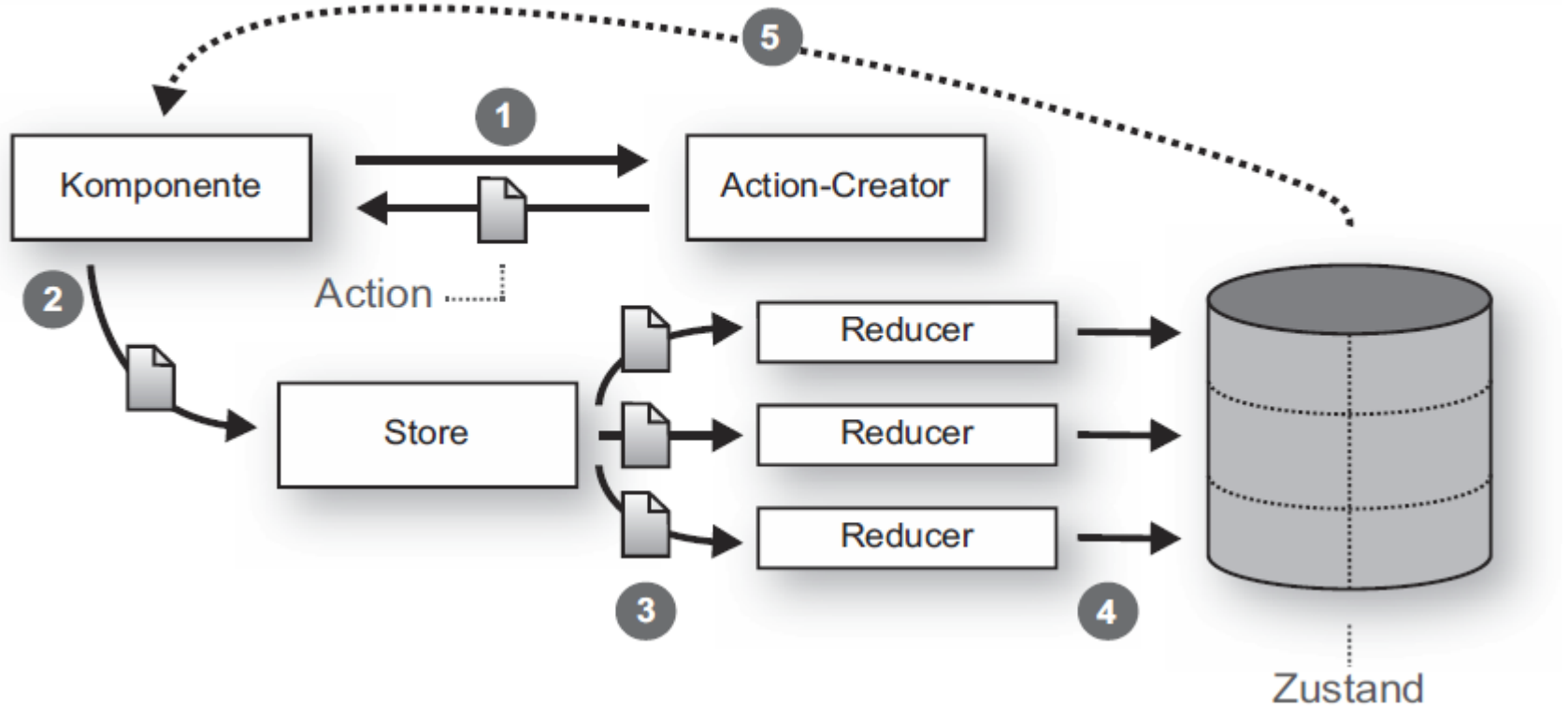
- **Changes are made with pure functions**
  Reducer sind seiteneffektfreie Funktionen: Sie erhalten zwei Eingabeparameter (den derzeitigen Anwendungszustand sowie die Action, die gerade ausgelöst wurde) und erzeugen daraus eine Ausgabe (den neuen Anwendungszustand).

Flux oder nicht?

Flux (links) und Redux (rechts)







Eine Komponente erzeugt mit einem Action-Creator eine Action (1) und übergibt diese zur Verteilung an den zentralen Store (2). Der Store gibt die Action an alle bekannten Reducer weiter (3). Die Reducer verarbeiten die Action und geben jeweils einen neuen Teilzustand als Ergebnis der Verarbeitung zurück. Die Ergebnisse aller Reducer werden im neuen Gesamtzustand abgelegt (4). Der neue Gesamtzustand wird der Komponente über Properties (5) übergeben. Die Komponente rendert sich mit den aktualisierten Properties.

Beispiel (TODOs): State

State (global, hierarchisch):

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Beispiel (TODOs): Reducer kombinieren

Den Zustand dürfen nur die Reducer ändern und zwar nur ihren Teilbaum.
In unserem Beispiel haben wir zwei Reducer, „todos“ und „visibilityFilter“, die wie folgt kombiniert werden (im ROOT-Reducer):

reducers/index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})

export default todoApp
```

Beispiel (TODOs): Reducer

Reducer sind seiteneffektfreie Funktionen: Sie erhalten zwei Eingabeparameter (den derzeitigen Anwendungszustand sowie die Action, die gerade ausgelöst wurde) und erzeugen daraus eine Ausgabe (den neuen Anwendungszustand).

reducers/todos.js

```
function todos(state = [], action) {
 switch (action.type) {
 case 'ADD_TODO':
  return state.concat([{ text: action.text, completed: false }]);
 case 'TOGGLE_TODO':
  return state.map((todo, index) =>
   action.index === index ?
    { text: todo.text, completed: !todo.completed } :
    todo
 )
 default:
  return state;
 }
}

export default todos
```

Beispiel (TODOs): Actions/Action Creators

Actions sind einfach die Payload-Objekte, die an die Reducer übergeben werden (z.B. { type: 'TOGGLE_TODO', id }).

Action-Creator sind die Funktionen, die die Actions erzeugen (z.B. die Funktion „toggleTodo“ im Beispiel unten).

In der Komponente kann man die Action wie folgt triggern:
store.dispatch(toggleTodo(3))

actions/index.js

```
export const toggleTodo = (id) => {
  return {
    type: 'TOGGLE_TODO',
    id
  }
}
```

Beispiel (TODOs): Container Component

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}
const mapStateToProps = (state) => {
  return { todos:
      getVisibleTodos(state.todos, state.visibilityFilter)  }
}
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Beispiel (TODOs): mapDispatch-/mapStateToProps

mapStateToProps/mapDispatchToProps sind Helper-Funktion von Redux.

mapDispatchToProps mappt die Funktionen zu den Properties der Komponente.

mapStateToProps mappt die Daten aus dem State zu den Properties der Komponente.

containers/VisibleTodoList.js

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Presentational and Container Components

Presentational Components	Container Components	
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Redux vs Flux (vom Dan Abramov)

Reducer Composition

- Flux makes it unnatural to reuse functionality across stores.
- In Flux, stores are flat, but in Redux, reducers can be nested via functional composition, just like React components can be nested.
- Can you imagine plugging Undo/Redo into a Flux app being two lines of code? Hardly. With Redux, it is.

Server Rendering

- Since there is just a single store (managed by many reducers), you don't need any special API to manage the (re)hydration

Redux vs Flux (vom Dan Abramov)

Developer Experience

- It is possible to change reducer code on the fly or even “change the past” by crossing out actions, and see the state being recalculated.
- Hot Reloading, Time Travel Debugger, undo/redo.

Ecosystem

- Redux provides a few extension points such as middleware.
- It was designed with use cases such as logging, support for Promises, Observables, routing, immutability dev checks, persistence, etc, in mind.

Redux vs Flux (vom Dan Abramov)

Simplicity

- Redux preserves all the benefits of Flux (recording and replaying of actions, unidirectional data flow, dependent mutations)
- Redux adds new benefits (easy undo-redo, hot reloading) without introducing Dispatcher and store registration.
- Unlike most Flux libraries, Redux API surface is tiny.

Links

- http://redux.js.org
- https://egghead.io/courses/getting-started-with-redux
- https://speakerdeck.com/rstankov/react-and-redux
- http://stackoverflow.com/questions/32461229/why-use-redux-over-facebook-flux
- https://github.com/xgrommx/awesome-redux
- https://www.dpunkt.de/leseproben/12388/4_Flux-Architektur%20am%20Beispiel%20von%20Redux%20(Kapitelauszug).pdf