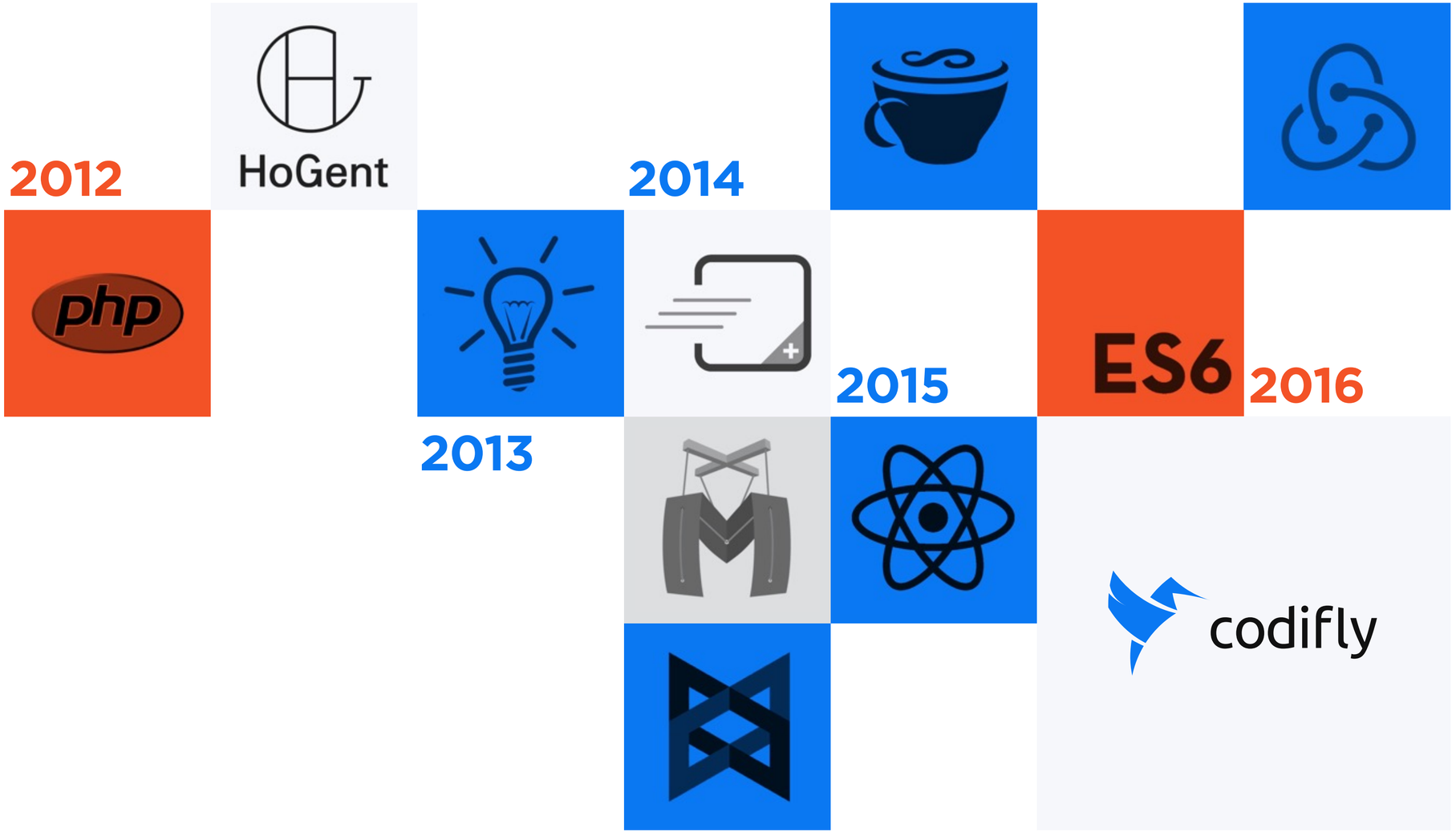
codifly

React, Redux and Building Applications that Scale

BY KRISTOF VAN MIEGEM
Co-Founder Codifly

About me



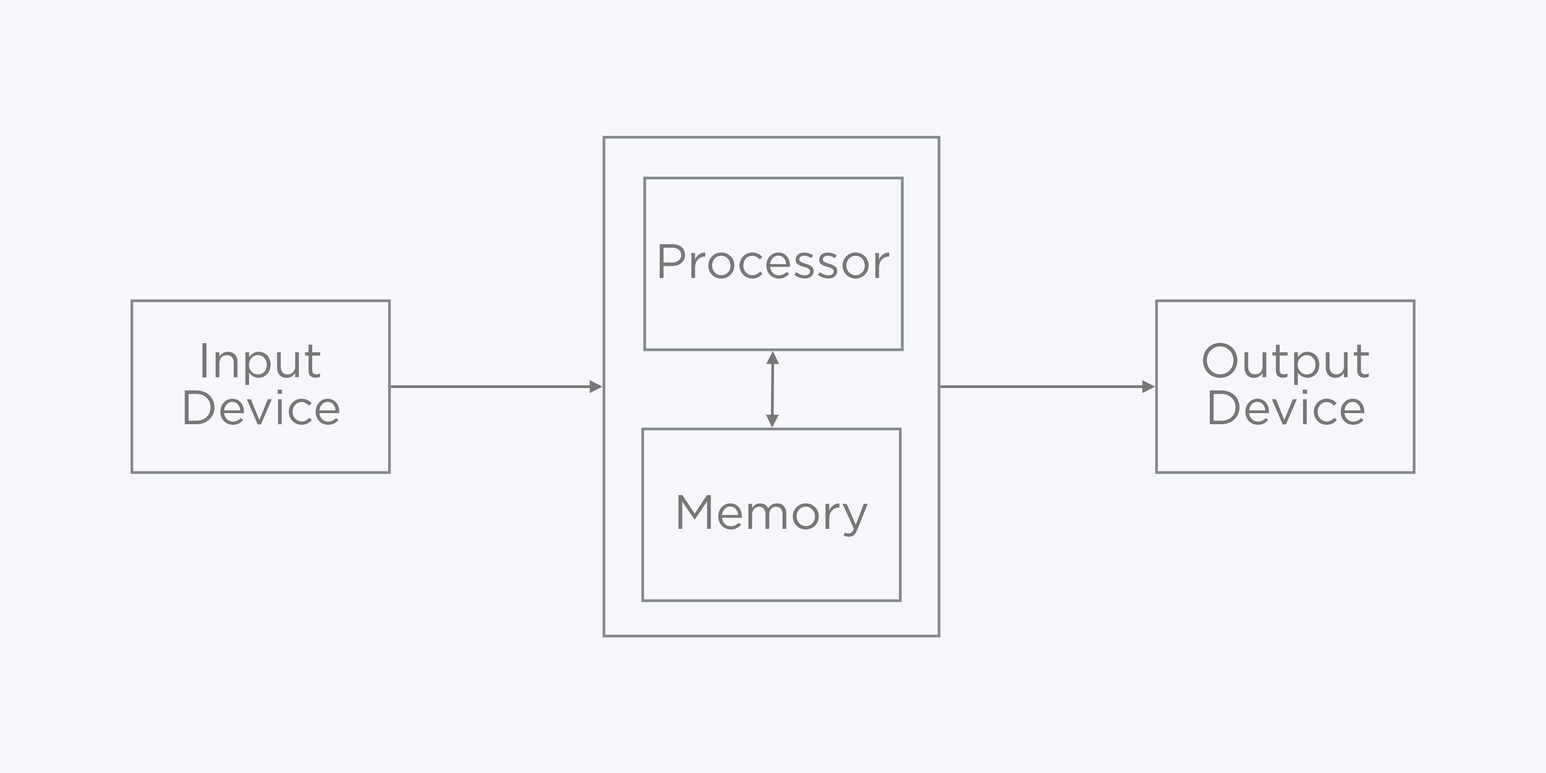
Agenda

- Issues with Traditional Front End Development
- Managing Application State
- Embracing Immutable Data Structures

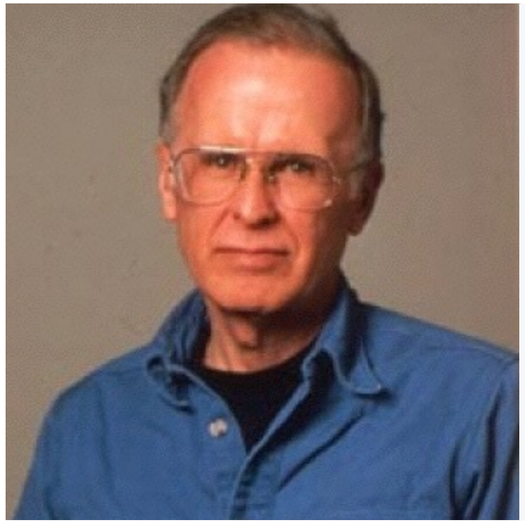
Issues with traditional front-end development

About imperative programming

The Von Neumann Architecture



Assignment is bad, mmkay!



"Can Programming be Liberated from the Von Neumann Style?"

— Paper by John Backus

Led to Functional Programming

Pure Functions

$$f(x) = f(x)$$

React to the Rescue

Structuring JavaScript Apps Done Well

React

“How do you **structure** front end JavaScript Applications?”

React Components

Cohesive units bundling **both the look-and-feel and logic** of the UI:

- Declaratively define how the UI should look like at any point in time
- Contain logic for event handling

Example: A Stateful Counter Component (1/2)

```
class Counter extends Component {

  constructor () {
    super();
    this.onInc = ::this.onInc;
    this.state = { currentValue: 1 };
  }

  onInc () {
    this.setState({
      currentValue:
        this.state.currentValue + 1
    });
  }
```

Example: A Stateful Counter Component (2/2)

```
  render () {
    return (
      <div>

        <div>
          {this.state.currentValue}
        </div>

        <ClickButton
          text='+'
          onClick={this.onInc} />

      </div>
    );
  }

}
```

https://jsfiddle.net/57ooLr66/

Render as a Pure Function

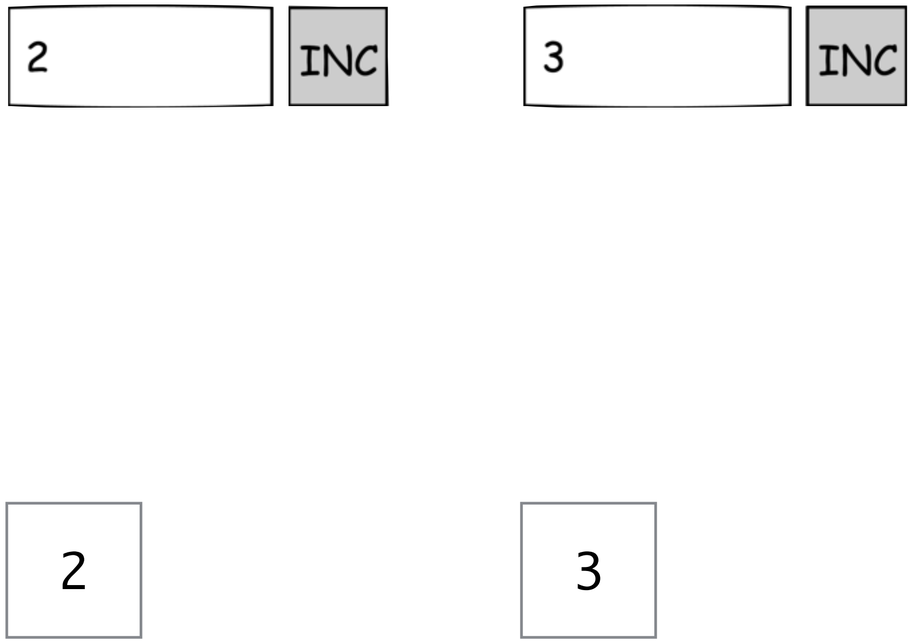
```
render(props, state)
= render(props, state)
```

Managing Application State

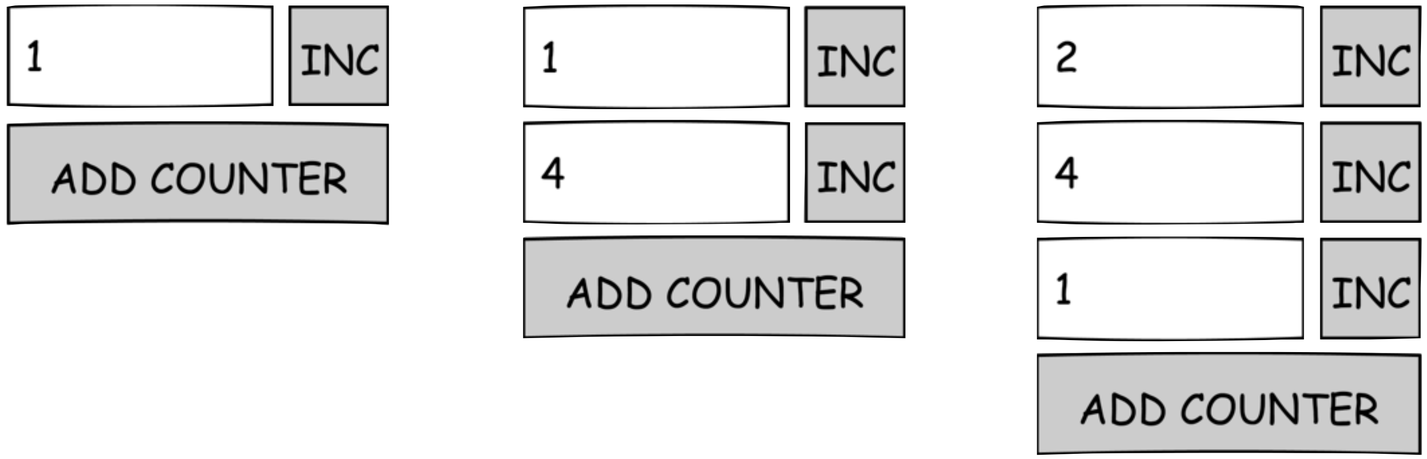
With Redux

Example 1: Counter

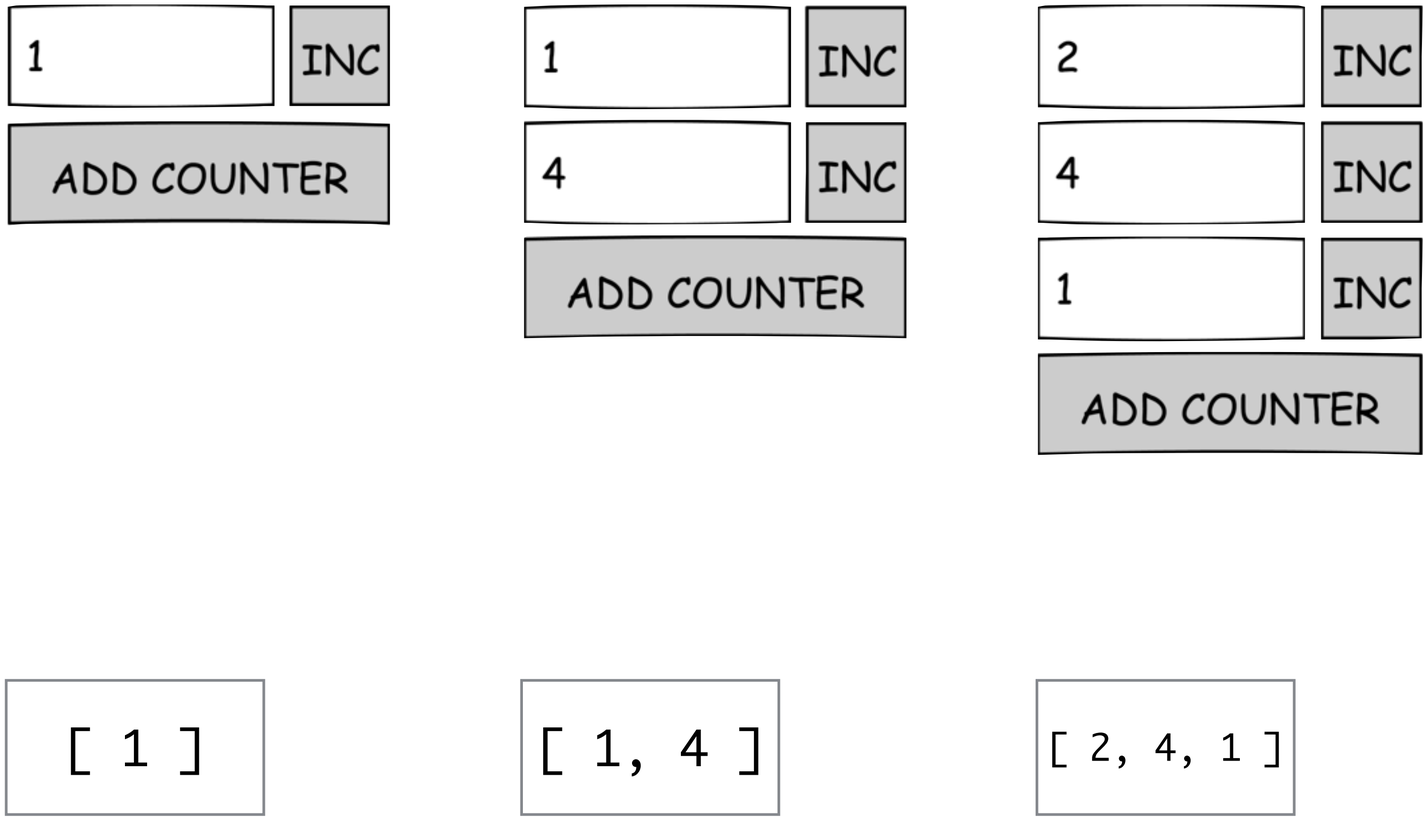
Example 1: Counter



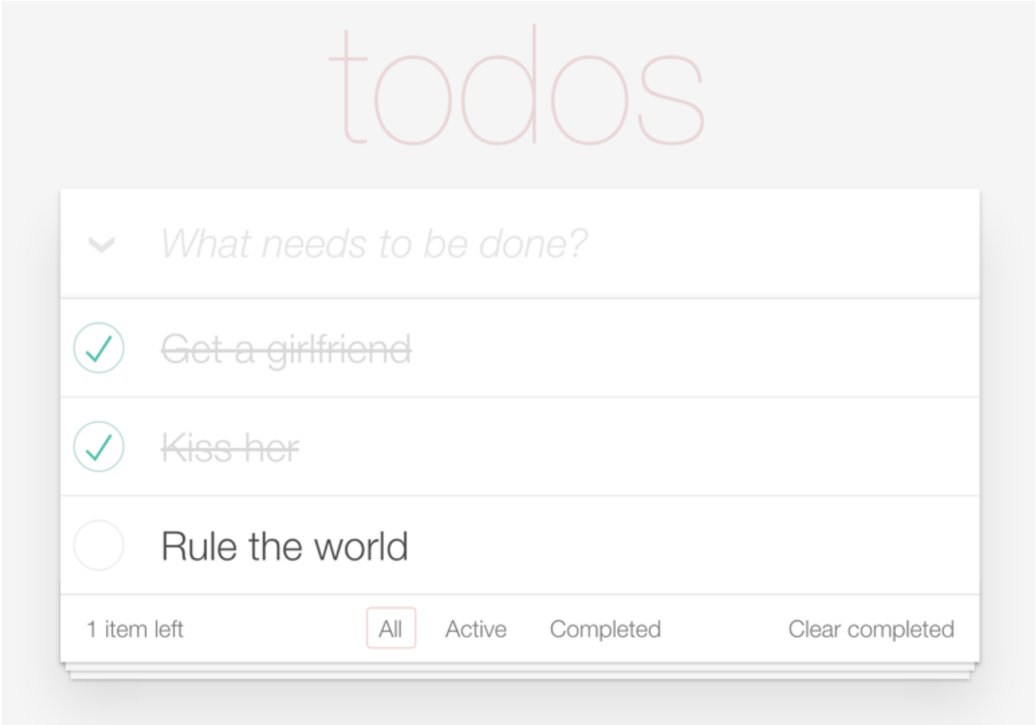
Example 2: Multiple Counters



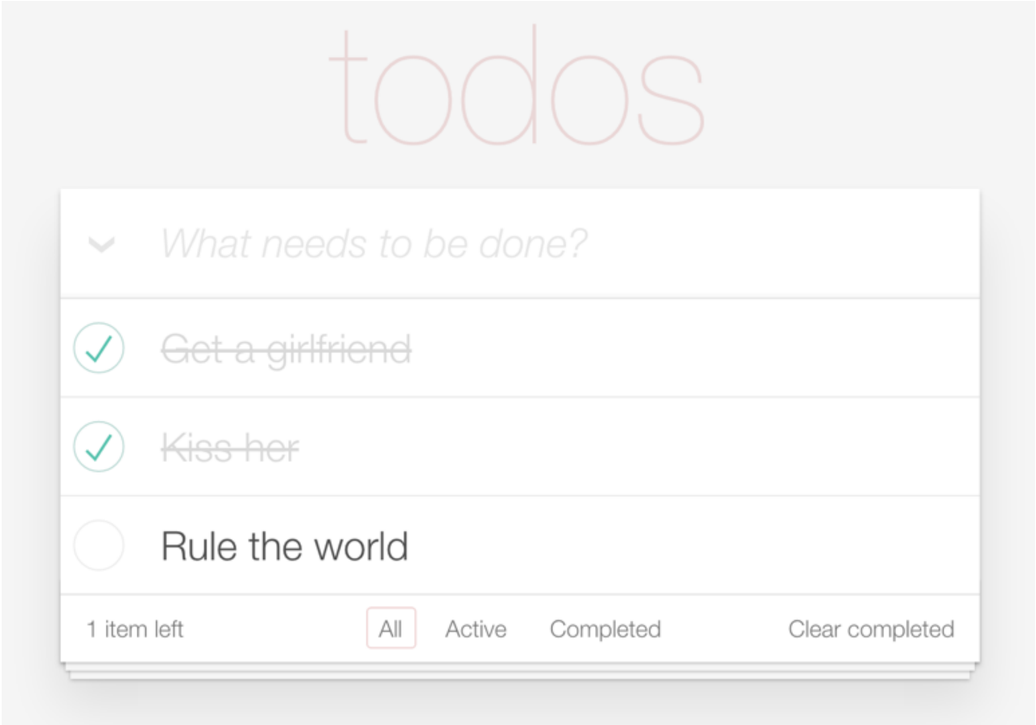
Example 2: Multiple Counters



Example 3: TodoMVC



Example 3: TodoMVC



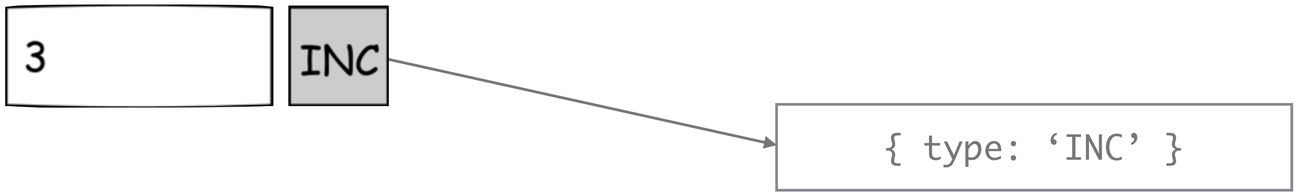

```
{
  todos: [
    { completed: true,
      id: 0,
      text: 'Get a girfriend' },
    { completed: true,
      id: 1,
      text: 'Kiss her' },
    { completed: false,
      id: 2,
      text: 'Rule the world' },
  ],
  visibilityFilter: 'SHOW_ALL'
}
```

State is Read-Only

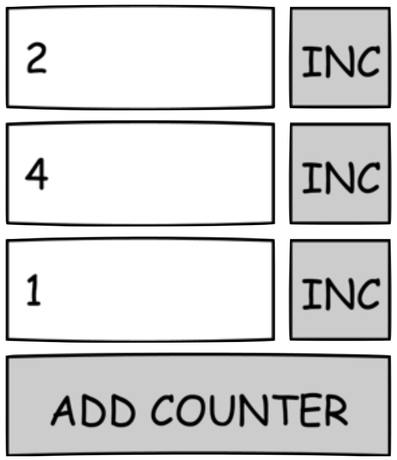
The only way to change the state is to emit an **Action**:

- A plain JavaScript object that describes what happened.
- Expresses an intent to change state

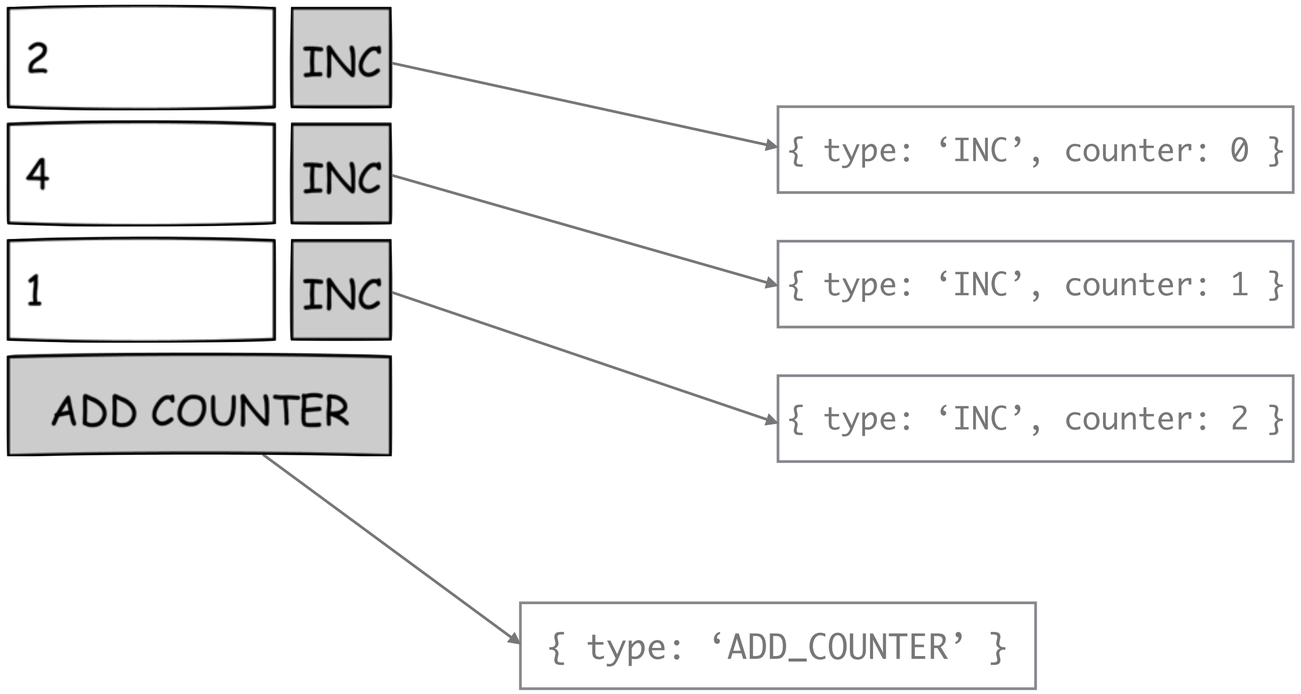
Example 1: Counter



Example 2: Multiple Counters



Example 2: Multiple Counters



Action Creators

```
function inc (counter) {
  return { type: 'INC', counter };
}

function addCounter () {
  return { type: 'ADD_COUNTER' };
}
```

State Changes are Made with Pure Functions

To specify how the state tree is transformed by actions, you write **reducers**:

- Pure functions
- that take the previous state and an action, and return the next state

State Changes are Made with Pure Functions

```
reducer(state, action)
= reducer(state, action)
```

Example: Counter

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INC':
      return state + 1;
    default:
      return state;
  }
};
```

Bringing it All Together: The Store

The store is responsible for:

- Containing the current state of the app
- Allowing state updates
- Notification of listeners

Bringing it All Together: The Store

Store creation

```
let store = createStore(reducer)
```

Triggering intents to change: dispatching actions

```
store.dispatch(actionCreator(…))
```

Listening for state changes

```
store.subscribe(() =>
  console.log(store.getState())
)
```

Full Example

```
// Actions
const inc = () => ({ type: 'INC' });
const dec = () => ({ type: 'DEC' });

// Reducer
const counter = (state = 0, action) => {
  switch (action.type) {
    case ‘INC': return state + 1;
    case ‘DEC': return state - 1;
    default: return state;
  }
};
```

Full Example

```
// View
class Counter extends React.Component {
  render () {
    return (
      <div>
        <h1>{this.props.value}</h1>
        <button onClick={this.props.onIncrement}>
          +
        </button>
        <button onClick={this.props.onDecrement}>
           -
         </button>
      </div>
    );
  }
}
```

Full Example

```
// Store
const store = Redux.createStore(counter);

// Rendering
const render = () => {
  ReactDOM.render(
    <Counter
      value={store.getState()}
      onIncrement={() => store.dispatch(inc())}
      onDecrement={() => store.dispatch(dec())}
      />,
    document.getElementById('container')
  );
};

render();
store.subscribe(render);
```

https://jsfiddle.net/pxktc6pv/

Reselect

- Selectors get or compute data
- Selectors are composable
- Selectors are performant

Selectors

```
import { createSelector } from 'reselect'

const productsSelector =
  state => state.productsInBasket
const vatPercentageSelector =
  state => state.vatPercentage

const totalSelector = createSelector(
  productsSelector,
  vatPercentageSelector,
  (products, vatPercentage) =>
    products.reduce(
      (acc, product) => acc + product.price, 0)
    * (1 + vatPercentage)
)
```

Thunks

```
function inc () {
  return (dispatch) => {
    setTimeout(() => {
      dispatch({ type: 'INC' });
    }, 5000);
  };
}
```

Embracing Immutable Data Structures

For improved reasoning and performance

Numbers are Immutable

$$3 + 5 = 8$$

Arrays are Mutable

```
var a = [3];
a.push(5);


console.log(a);
```

Hypothetical Immutable Array

```
var a = [3];
a.push(5);


console.log(a);
```

```
var a = [3]i;
var a2
  = a.push(4);

console.log(a);
console.log(a2);
```

https://facebook.github.io/immutable-js/

Docs Questions Github

IMMUTABLE

Immutable collections for JavaScript

Immutable data cannot be changed once created, leading to much simpler application development, no defensive copying, and enabling advanced memoization and change detection techniques with simple logic. Persistent data presents a mutative API which does not update the data in-place, but instead always yields new updated data.

Immutable.js provides many Persistent Immutable data structures including: `List`, `Stack`, `Map`, `OrderedMap`, `Set`, `OrderedSet` and `Record`.

Immutable List

```
var a = [3]i;
var a2
  = a.push(4);


console.log(a);
console.log(a2);
```

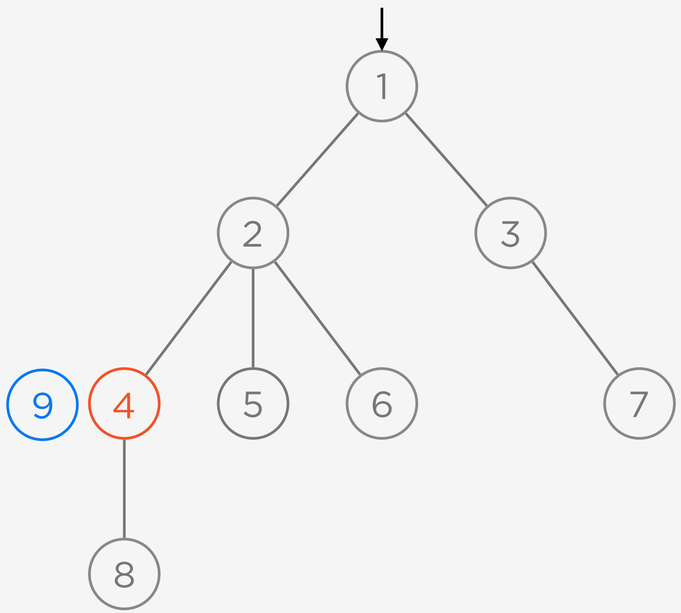
```
var l = List([3]);
var l2
  = list.push(4);


console.log(l.toJS());
console.log(l2.toJS());
```

Updating Immutable Data

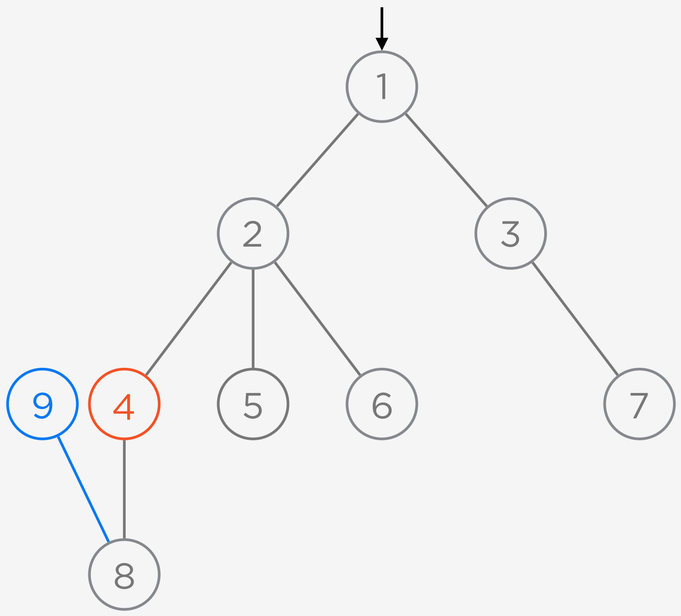
Conceptually, the update process

(1) Deep clone
(2) Perform changes
(3) Return result

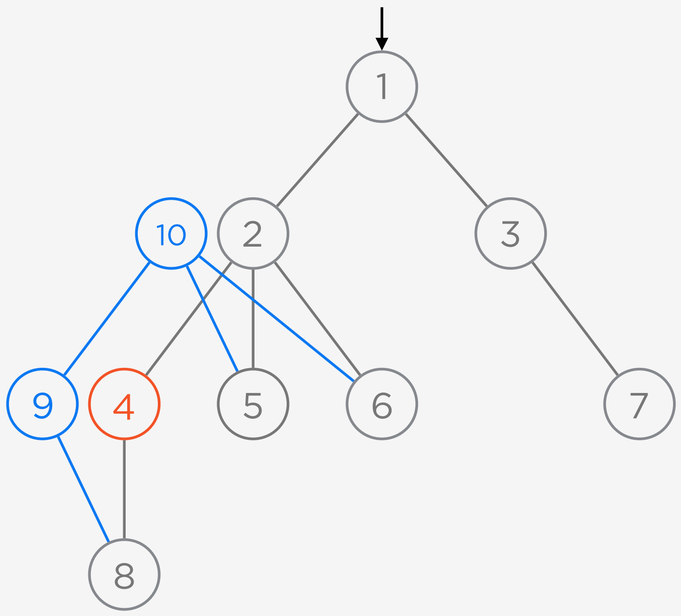
Structural Sharing



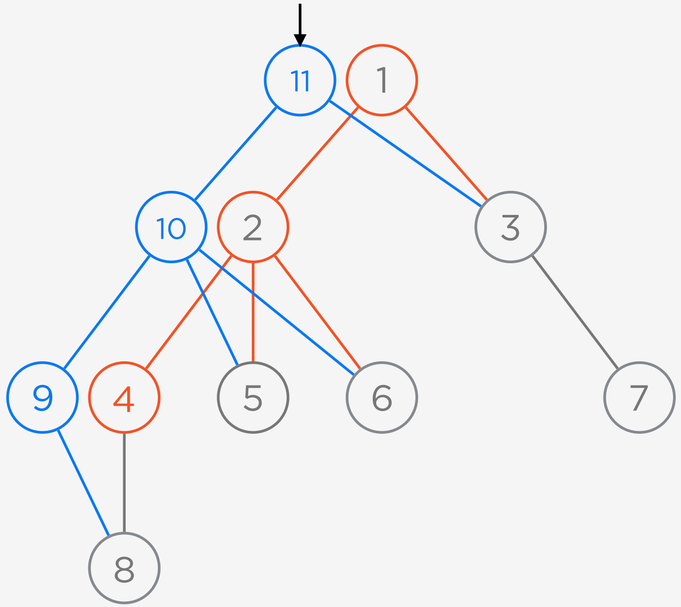
Structural Sharing



Structural Sharing



Structural Sharing



Get and Set

```
const map1 = Immutable.Map({ a: 1, b: 2, c: 3 });
const map2 = map1.set('b', 4);

console.log(map2.count()); // 3
console.log(map1.get('b')); // 2
console.log(map2.get('b')); // 4

const list1 = Immutable.List([ 1, 2 , 3 ]);
const list2 = list1.set(1, 4);
console.log(list2.count()); // 3
console.log(list1.get(1)); // 2
console.log(list2.get(1)); // 4
```

Nested Data Structures

```
var nested1 =
  Immutable.fromJS({ a: { b: [ 1, 2, 3 ] } });
var nested2 =
  nested1.setIn(['a', 'b', 1], 4);

console.log(nested1.getIn(['a', 'b', 1])); // 2
console.log(
  nested2.getIn(['a', 'b']).toJS()); // [ 1, 2, 3 ]
console.log(nested2.getIn(['a', 'b', 1])); // 4
```

codifly

Redux-like state management in Angular 2

BY WOUT DE ROOMS @woutderooms

Web & Mobile Engineer at Codifly

About Me

Agenda

- Observables
- Managing Application State with ngrx/store
- And What About Angular 2?

Observables

A Crash Course

Basic Principles

- Observables are kinda like streams
- Observables are lazy
- Think synchronously over asynchronous flows

Subscription

- Subscribe()
- Unsubscribe()

RxJS operators

- Transformation
- Combination
- Creation

Managing Application State with ngrx/store

Redux-Inspired, but Different

ngrx/store

https://github.com/ngrx/store

Code | Issues 9 | Pull requests 6 | Projects 0 | Pulse | Graphs

RxJS powered state management for Angular2 apps, inspired by Redux

140 commits | 2 branches | 22 releases | 15 contributors

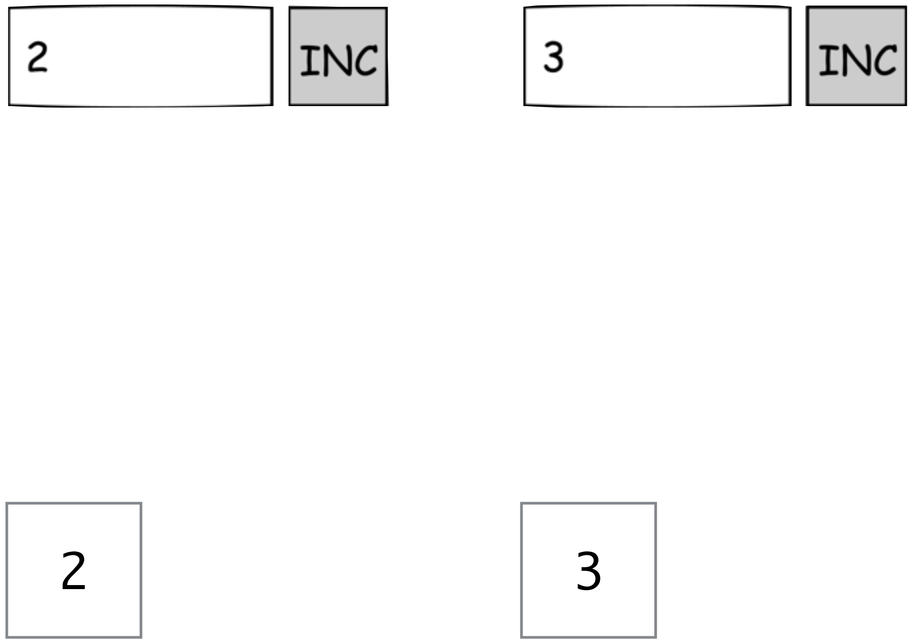
Branch: master | New pull request | Find file | Clone or download

robwormald fix(packaging): ngrx/core fix for umd — Latest commit 6a526ae 3 days ago

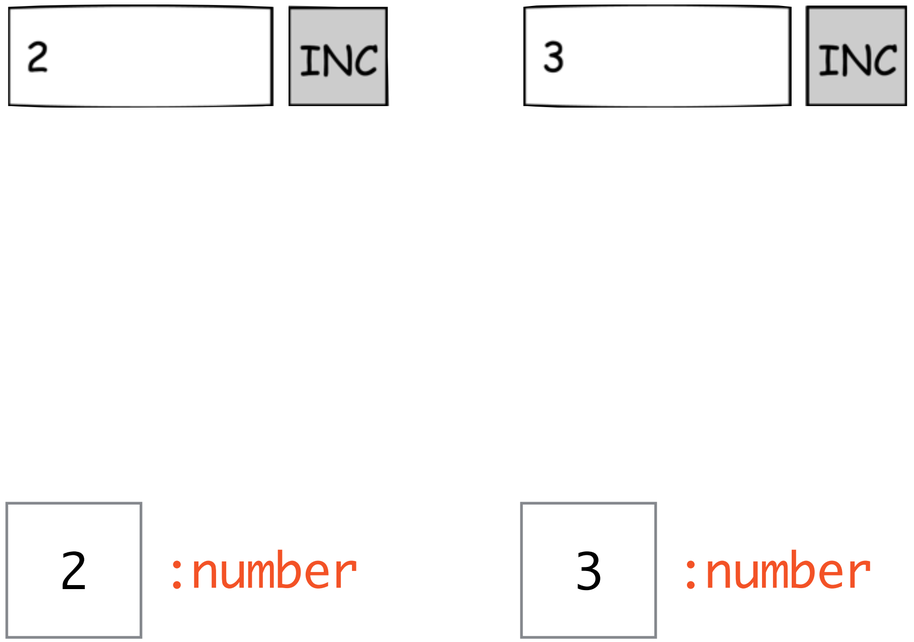
spec	chore: Update build process to emit ESM modules and a UMD bundle (#212)	5 days ago
src	fix(packaging): Use top level Core export of select. (#218)	3 days ago
.editorconfig	chore(build): add .editorconfig	9 months ago
.gitignore	chore: Update build process to emit ESM modules and a UMD bundle (#212)	5 days ago
.npmignore	chore(build): add .npmignore	9 months ago
CONTRIBUTING.md	update npm scripts and contribution guide, bump up package versions (#...	2 months ago
LICENSE	Initial commit	9 months ago

ngrx/store

A Single Source of Truth - Revisited



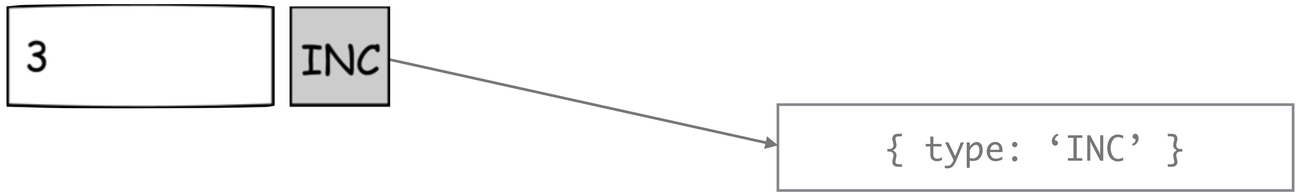
The State Tree: Defining Shape



State is Read-Only - Revisited

The only way to change the state is to emit an **Action**:

- A plain JavaScript object that describes what happened.
- Expresses an intent to change state



Action Creators

```
interface Action {
  type: string,
  payload?: any
}

function addCounter ():Action {
  return { type: 'ADD_COUNTER' };
}
```

State Changes - Revisited

To specify how the state tree is transformed by actions, you write **reducers**:

- Pure functions
- that take the previous state and an action, and return the next state

Example: Counter

```
const counter = (state: number = 0,
action: Action): number => {
  switch (action.type) {
    case 'INC':
      return state + 1;
    default:
      return state;
  }
};
```

Bringing it All Together: The Store

The store is responsible for:

- Containing the current state of the app
- Allowing state updates
- Allowing subscriptions on state changes

Bringing it All Together: The Store

Store creation

```
StoreModule.provideStore({ … })
```

Triggering intents to change: dispatching actions

```
store.dispatch(actionCreator(…))
```

Listening for state changes

```
store.select(…): Observable
```

And What About Angular 2?

By Example

1... 2... 3!

```
// Actions
interface IAction {
  type: string,
  payload?: any
}

const inc = (): IAction => ({ type: 'INC' });
const dec = (): IAction => ({ type: 'DEC' });

// Reducer
export const counterReducer = (state: number = 0,
action: IAction) => {
...
```

1... 2... 3!

```
...
const inc = (): IAction => ({ type: 'INC' });
const dec = (): IAction => ({ type: 'DEC' });

// Reducer
export const counterReducer = (state: number = 0,
action: IAction) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    default: return state;
  }
};

// Component
@Component({
...
```

1... 2... 3!

```
    default: return state;
  }
};

// Component
@Component({
  selector: 'my-app',
  template: `
    <div>
      <h1>{{value | async}}</h1>
      <button (click)='onIncrement()'>+</button>
      <button (click)='onDecrement()'>-</button>
    </div>`
})
export class AppComponent {
  ...
```

1... 2... 3!

```
      ...
    </div>`
})
export class AppComponent {
  value: Observable<number>
  constructor(private _store: Store<any>) {
    this.value = _store.select((s: any) => s.counter);
  }
  onIncrement () {
    this._store.dispatch(inc());
  }
  onDecrement () {
    this._store.dispatch(dec());
  }
}
```

1... 2... 3!

```
// Bootstrapping our store
@NgModule({
  imports:      [ BrowserModule,
                  StoreModule.provideStore({
                    counter: counterReducer
                  })
                ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

http://plnkr.co/edit/BHkYj9X1Tp5bExSsvGkz?p=info

codifly

codifly

Looking for an internship or exciting job?

Contact us
kristof@codifly.be

www.codifly.be