Angular Advanced
@ngrx/store – Action Creators

Peter Kassenaar –
info@kassenaar.com

State & Store abstraction

Abstracting actions, using models, services and interfaces

OLD way: (V2.0.0) define actions in an object

```
// city.actions.ts
// An object, holding all possible actions on the store
export const ACTIONS = {
    ADD_CITY   : 'ADD_CITY',
    REMOVE_CITY: 'REMOVE_CITY',
    EDIT_CITY  : 'EDIT_CITY'
};
```

```
addCity(city: HTMLInputElement) {
   // add city to store
   this.store.dispatch({type: ACTIONS.ADD_CITY, payload: city.value});
   city.value = '';
}
```

Create City Model & AppState interface

- This is not mandatory, but gives you better type checking

```
export class City {
   constructor(public id: number       = -1,
            public name: string     = 'unknown',
            public province: string = 'unknown',
            public  inhabitants?: number) {

   }
}
```

```
//appstate.ts
import {City} from './city.model';

export interface AppState{
   cities: City[]
}
```

Use a service as a middle man

By using a service, we can abstract `Http` later on and we don't have our components talking directly to the `Store` or `Http`.

```
// city.service.ts
@Injectable()
export class CityService {
   cities: Observable<City[]>;

   constructor(private store: Store<AppState>) {
      this.cities = this.store.select(s => s.cities);
   }
}
```

V2.0.0: combine reducers to AppReducer

- Often, you'll have multiple reducers in your application
- Combine them into a single reducer using `combineReducers`.
- This allows for picking the reducer you need in any given situation
- You can't initialize the store with multiple reducers. If you have more, you need to combine them into a single reducer.
- See also this blog by Kwinten Pisman: https://blog.kwintenp.com/combinereducers-enhanced/

example appReducer

```
// appReducer, combine reducers to be supplied in app.module.ts.
export function appReducer(state: AppState, action: Action) : AppState{
   const reducer = combineReducers({
      cities: citiesReducer,
      selectedCity: selectedCityReducer
   });
   return reducer(state, action);
}
```

```
export interface AppState{
   cities: City[];
   selectedCity: City;
}
```

```
@NgModule({
   …,
   imports      : [
      BrowserModule,
      StoreModule.provideStore(appReducer)
   ],
   …
})
```

“Making an interface for the application state really helped me to understand how reducers fit into the application. In our **AppState** interface, you can see that we are dealing with a single object that has an **cities** collection and a **selectedCity** property which holds a single **City** object.

If we needed to add additional functionality, the store would just expand with new key value pairs to accommodate the updated model.”

http://onehungrymind.com/build-better-angular-2-application-redux-ngrx/

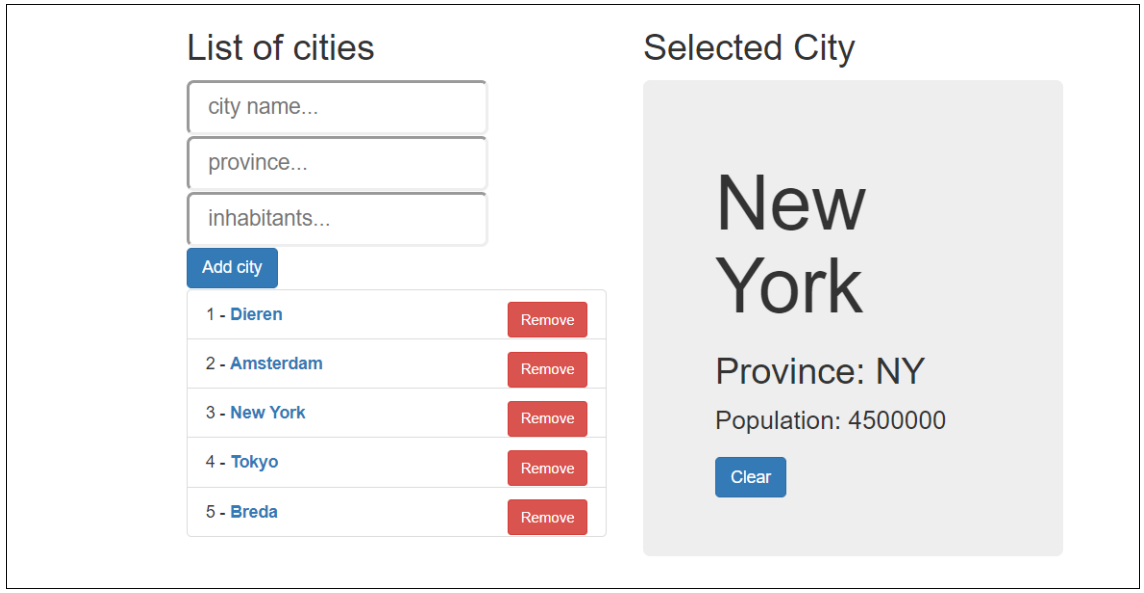
Using the AppState from the store

Subscribe to particular reducer, for instance

```
this.selectedCity$ = this.store.select(s => s.selectedCity)
```

This is the same as:

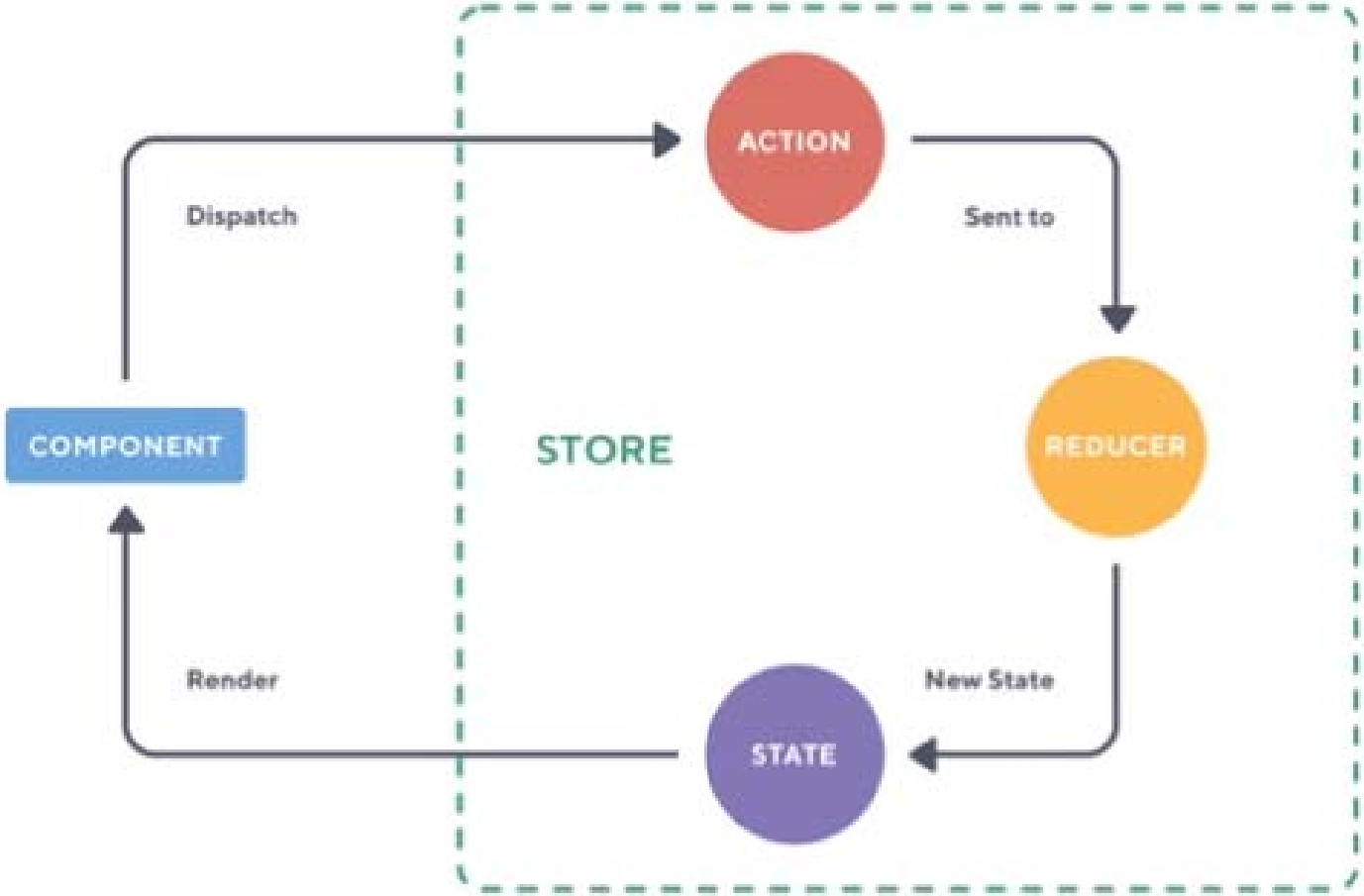
```
this.selectedCity$ = this.store.select('selectedCity')
```



../210-ngrx-store-abstractions

REDUX ARCHITECTURE

One-way dataflow



https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3788532

Action Creators

Store V4.0.0 and up: create constants and classes for actions

Step 1 – create the Action Constants

- Create `Constants` for Actions…
  a) to produce more readable output
  b) Benefit from static typing

```
// counter.action.ts
// import Action interface for static typing later on
import {Action} from '@ngrx/store';

// *** Action constants
// These are the strings for the action
export const INCREMENT = '[COUNTER] - increment';
export const DECREMENT = '[COUNTER] - decrement';
export const RESET = '[COUNTER] - reset';
```

Action Creators

- Create a class for each action…
  - Which implements `Action`
  - Defines a `type` property with the constant of your choice
  - In the constructor you define your own, optional `payload` property

```
// *** Action Creators
export class CounterIncrement implements Action {
  readonly type = INCREMENT;
  constructor(public payload?: number) {}
}
```

You now can define a specific type for every `payload`

Export type

- Not mandatory, but seen very often (and considered best practice):
  - Export a new type `All` or *YourName*`Action`, of the types you just created.
  - Again, gives you nice intellisense and type safety in the reducers

```
//export action types, so they can be used in the reducers
export type CounterActions = CounterIncrement | CounterDecrement | CounterReset;


//OR: simply call the type All:
export type All = CounterIncrement | CounterDecrement | CounterReset;
```

Step 2 – create reducers to use the Actions

- Optional
  - Create constants for `initialState`
  - and for the type that the reducer returns (in this case a `number`, but it can be a custom object or `interface`)

```
// counter.ts - a simple reducer, now with abstracted Counter Actions
import * as fromActions from '../actions/couter.actions';

// Optional: create initial State.
export const initialState: number = 0;

// Optional: create an interface as the return type for the reducer.
export interface counterState{

}
```

Build the reducer

- Create switch statement to manipulate the state

```
// counter.ts
export function counterReducer(state = initialState,
                               action: fromActions.CounterAction): counterState {
   switch (action.type) {
      case fromActions.INCREMENT:
         return action.payload ? state + action.payload : state + 1;

      case fromActions.DECREMENT:
         return state - 1;

      case fromActions.RESET:
         return 0;

      default:
         return state;
   }
}
```

Edit the Component

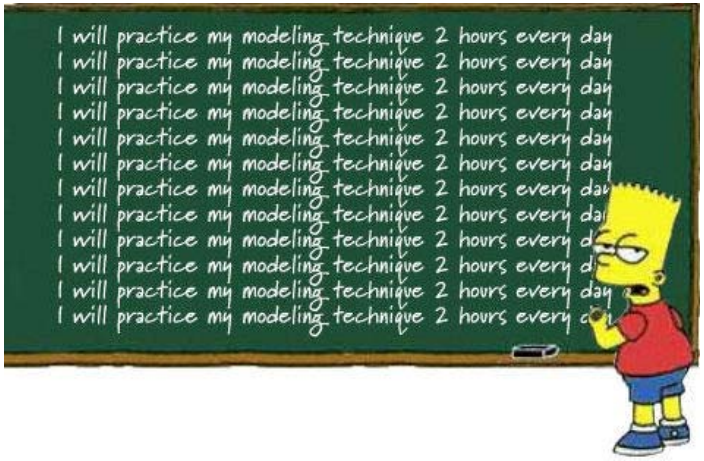
```
// app.component.ts
…
import {AppState} from './appState';
import * as fromActions from './actions/couter.actions’;

…
export class AppComponent implements OnInit {
  counter$: Observable<number>;

  constructor(private  store: Store<AppState>) {}

  ngOnInit() {
    this.counter$ = this.store.select('counter');
  }

  increment() {
    this.store.dispatch(new fromActions.CounterIncrement());
  }

  decrement() {
    this.store.dispatch(new fromActions.CounterDecrement());
  }

  reset() {
    this.store.dispatch(new fromActions.CounterReset());
  }

  // Add a specific number to the counter in the store
  addNumber(txtNumber: string) {
    this.store.dispatch(new fromActions.CounterIncrement(+txtNumber));
  }
}
```

New instance of Action Creator class

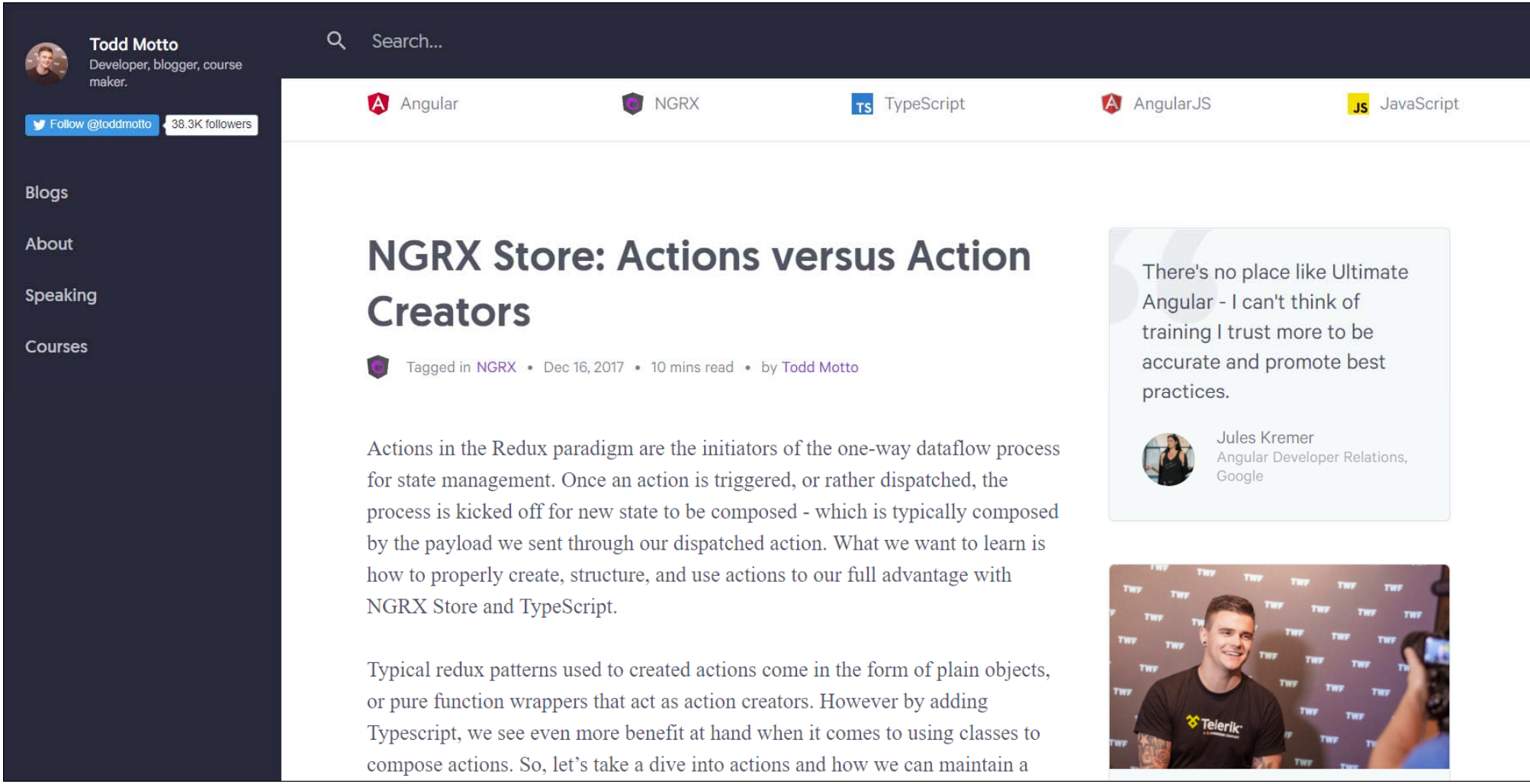
With optional payload

Workshop

- Start from `../202-ngrx-action-creators`
- Create your own Action Creator. Goal: multiply the current `counter` with a given number, typed in a textbox
  - Edit `counter.action.ts`
  - Edit `couter.reducer.ts`
  - Edit component so the user can type a multiplier in a textbox, which is handled by dispatching a new action to the reducer.



More info



https://toddmotto.com/ngrx-store-actions-versus-action-creators

Action reducers

Provide the `ActionReducerMap<T>` with your reducer map for added type checking.

```
import { ActionReducerMap } from '@ngrx/store';
import * as fromAuth from './auth';

export interface State {
  auth: fromAuth.State;
}

export const reducers: ActionReducerMap<State> = {
  auth: fromAuth.reducer
};
```

Typed Actions

Use strongly typed actions to take advantage of TypeScript's compile-time checking.

```
// counter.actions.ts
import { Action } from '@ngrx/store';

export enum CounterActionTypes {
  INCREMENT = '[Counter] Increment',
  DECREMENT = '[Counter] Decrement',
  RESET = '[Counter] Reset'
}

export class Increment implements Action {
  readonly type = CounterActionTypes.INCREMENT;
}

export class Decrement implements Action {
```

https://github.com/ngrx/platform/blob/master/docs/store/actions.md#action-reducers