Angular Advanced
@ngrx/store – Action Creators

Peter Kassenaar –
info@kassenaar.com

State & Store abstraction

Abstracting actions, using models, services and interfaces

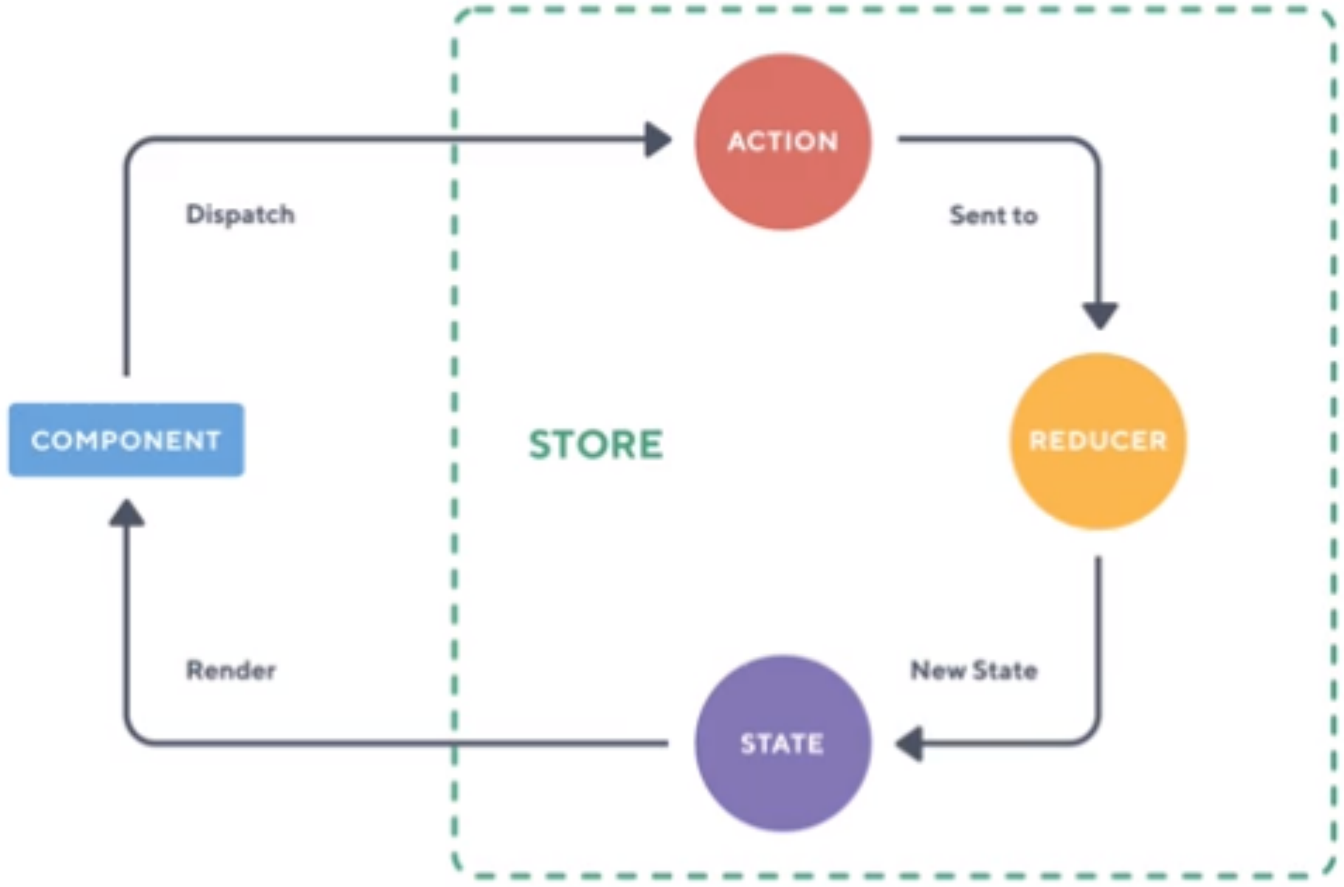
OLD way: (V2.0.0) define actions in an object

```
// 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 = '';
}
```

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/counter.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

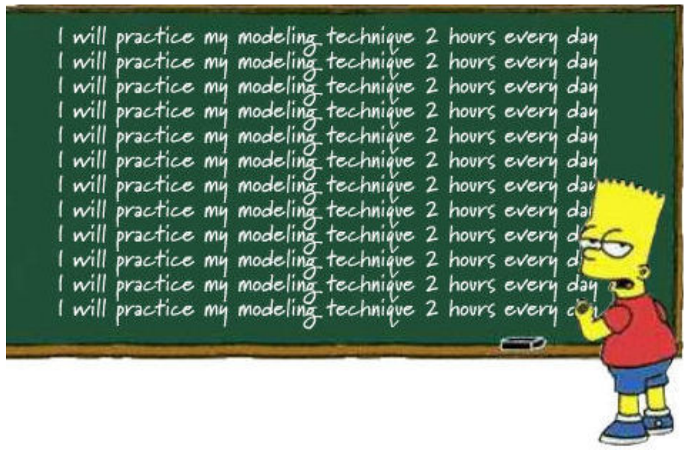
```
// app.component.ts
…
import {AppState} from './appState';
import * as fromActions from './actions/counter.actions’;

…
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

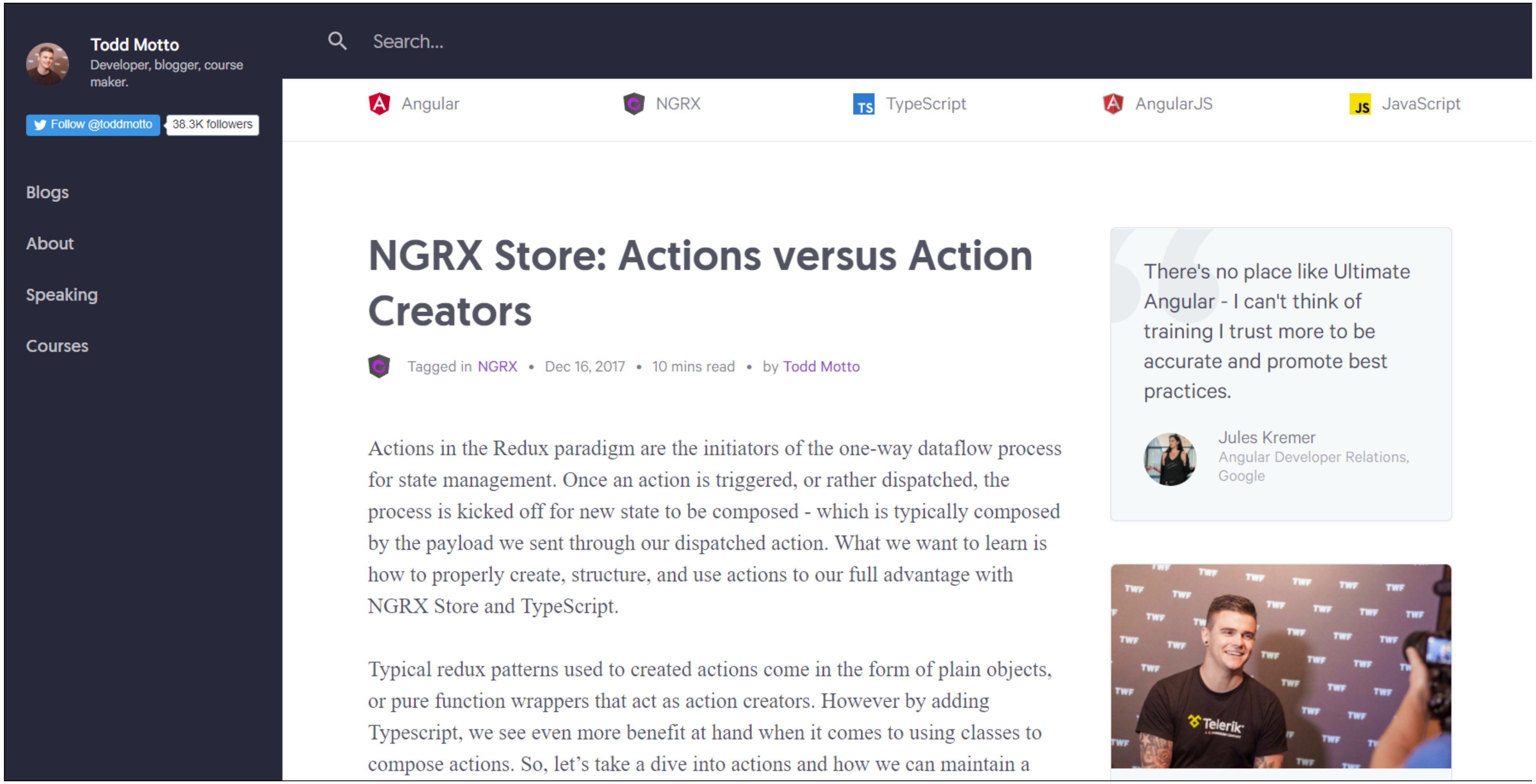
With optional `payload`

Workshop

- Start from `../205-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



NGRX Store: Actions versus Action Creators

Tagged in NGRX • Dec 16, 2017 • 10 mins read • by Todd Motto

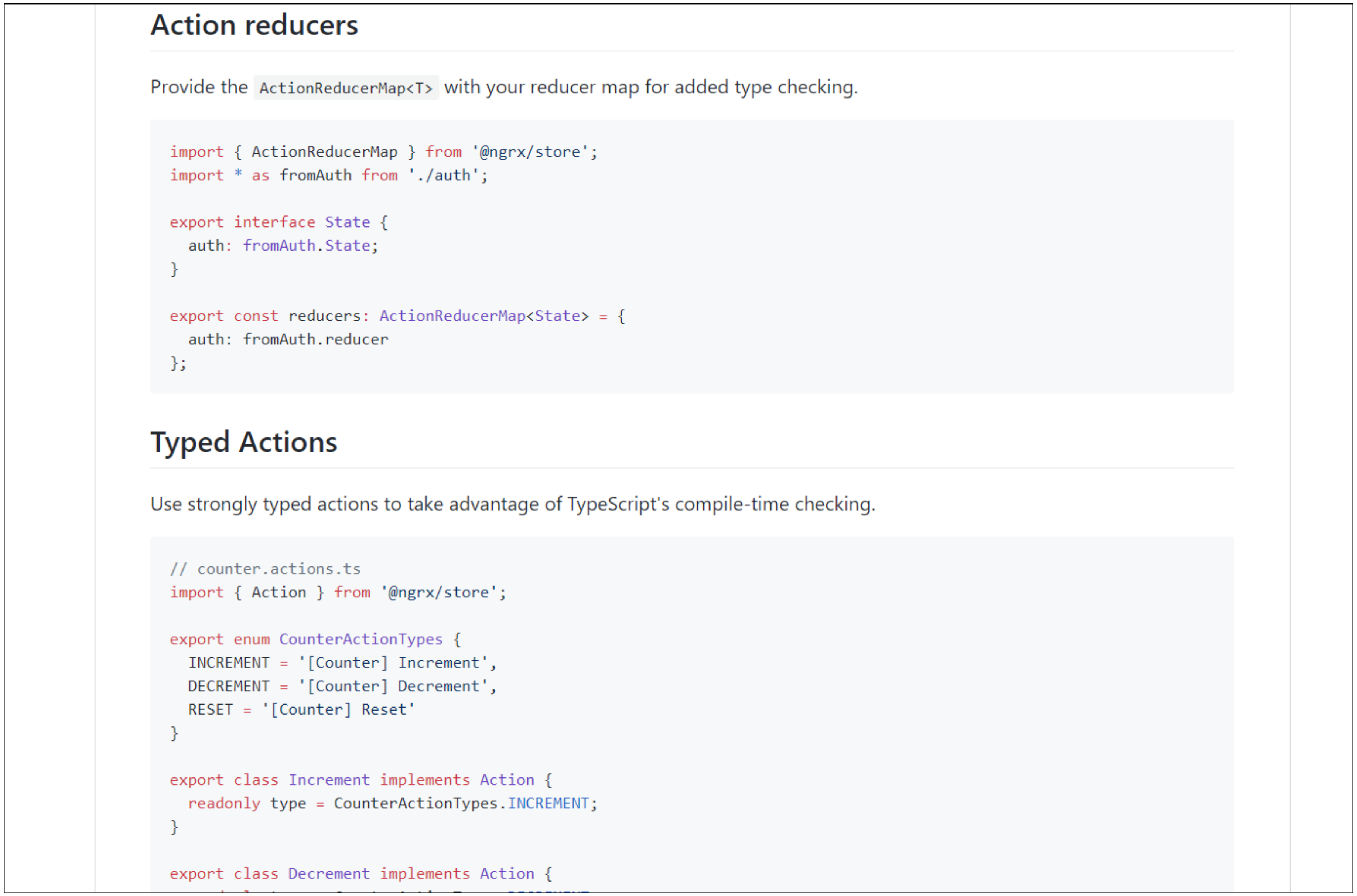
Actions in the Redux paradigm are the initiators of the one-way dataflow process for state management. Once an action is triggered, or rather dispatched, the process is kicked off for new state to be composed - which is typically composed by the payload we sent through our dispatched action. What we want to learn is how to properly create, structure, and use actions to our full advantage with NGRX Store and TypeScript.

Typical redux patterns used to created actions come in the form of plain objects, or pure function wrappers that act as action creators. However by adding Typescript, we see even more benefit at hand when it comes to using classes to compose actions. So, let's take a dive into actions and how we can maintain a

There's no place like Ultimate Angular - I can't think of training I trust more to be accurate and promote best practices.

Jules Kremer
Angular Developer Relations, Google

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

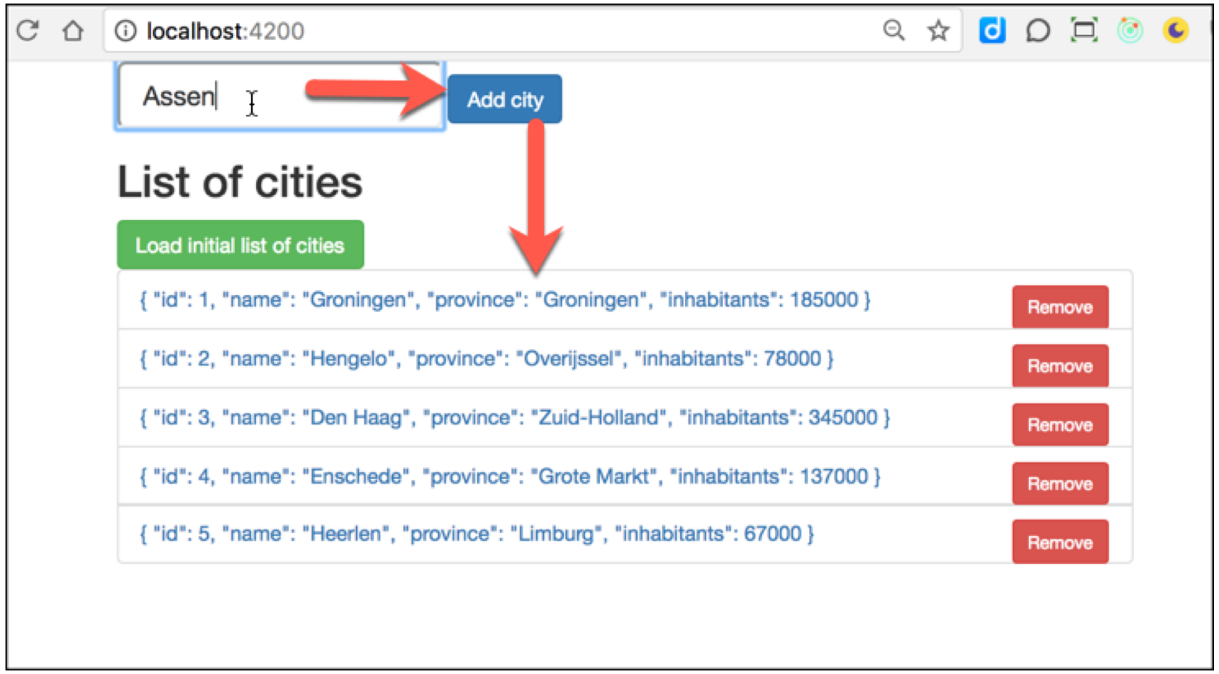
Use strongly typed actions to take advantage of TypeScript's compile-time checking.

```
// 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

Working with complex types

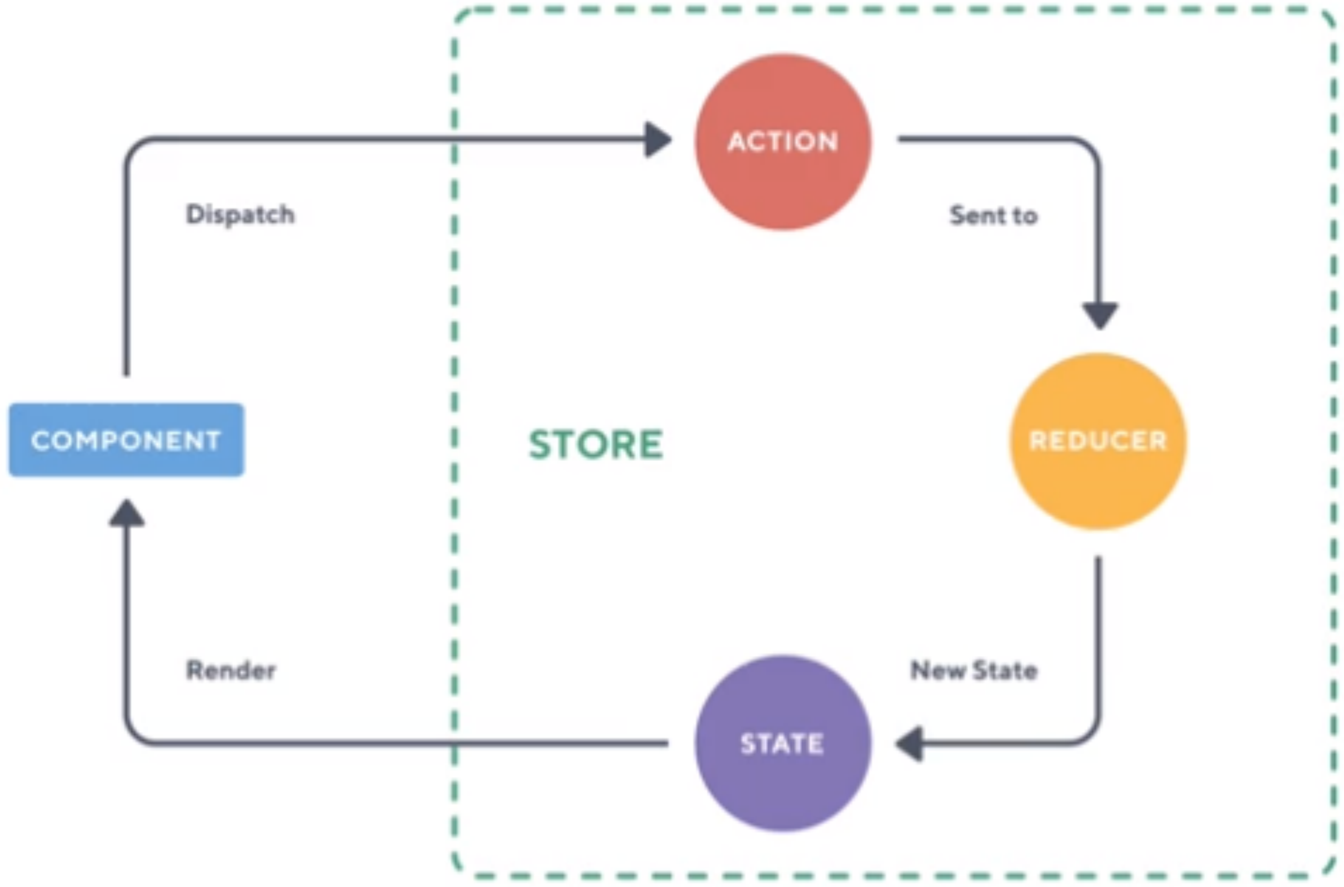
- Real Life Applications – Complex, custom types
- `../210-ngrx-store-complex-types`
- Start with your Actions, then work Clockwise in the diagram



{ "id": 1, "name": "Groningen", "province": "Groningen", "inhabitants": 185000 }
{ "id": 2, "name": "Hengelo", "province": "Overijssel", "inhabitants": 78000 }
{ "id": 3, "name": "Den Haag", "province": "Zuid-Holland", "inhabitants": 345000 }
{ "id": 4, "name": "Enschede", "province": "Grote Markt", "inhabitants": 137000 }
{ "id": 5, "name": "Heerlen", "province": "Limburg", "inhabitants": 67000 }

REDUX ARCHITECTURE

One-way dataflow



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

Workshop

- Start from `../210-ngrx-store-complex-types`
- Create new Actions and Reducer functions. Goal: Add a new City to the store
  - Check and/or Edit `city.actions.ts`
  - Update `city.reducer.ts`
  - Edit component so the user can type a city in the textbox, dispatched to the Reducer and added to the store.
  - Optional Goal: Create an Edit Action, so the user can edit the name of the clicked City