Reactive Angular with RxJS

03 – Using `Payload` in @ngrx/store

Peter Kassenaar –
info@kassenaar.com

State & Store abstraction

Updating state with arbitrary content, often called *payload*

OLD way: 1st gen: 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 = '';
}
```

OLD way: 2nd gen: define Action Creators

- Create Constants for Actions...
- Create classes that use these contstants and add optional payload to the constructor

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

// *** Action constants
// These are the strings for the acti
export const INCREMENT = '[COUNTER] -
export const DECREMENT = '[COUNTER] -
export const RESET = '[COUNTER] - res
```

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

NEW way: 3rd gen: add creator function

- **Step 1**

- Pass a *payload creator function* to `createAction()` as the second parameter.
- Optional: give payload an initial value

```
// 1. Increment with an optional payload as property
export const increment = createAction(
  'COUNTER - increment',
  //  2. Dispatching a default value '0', if no number is provided,
  // otherwise, dispatch an object with payload key/value pair
  (num = 0) => ({payload: num})
);
```

Step 2 – update reducer to use the payload

- Add payload to the reducer function

```
// Internal variable/function with reducer.
const reducer = createReducer(initialState,
  on(increment, (state, {payload}) => state + payload),
  on(decrement, state => state - 1),
  on(reset, state => initialState)
);

// The actual reducer function
export const counterReducer = (state = initialState, action: Action) => {
  return reducer(state, action);
};
```

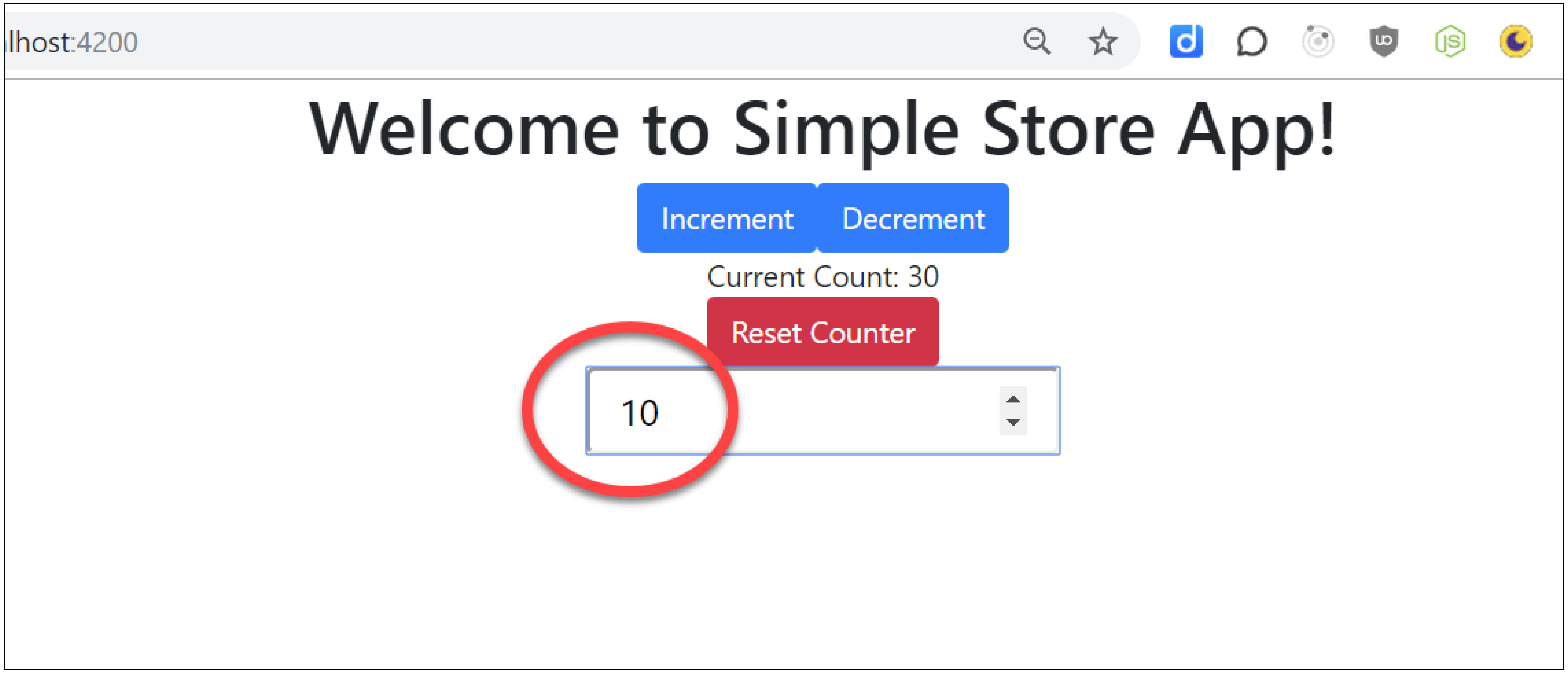
Step 3 – Update the Component

Update the dispatchers if a payload is provided with the action

```
// dispatch actions for the store. They are imported above
increment(num: number) {
  // if a number is provided, dispatch the increment action wi
  // that number (casted to a number);
  if (num) {
    this.store.dispatch(increment(num));
  } else {
    this.store.dispatch(increment(1));
  }
}
```

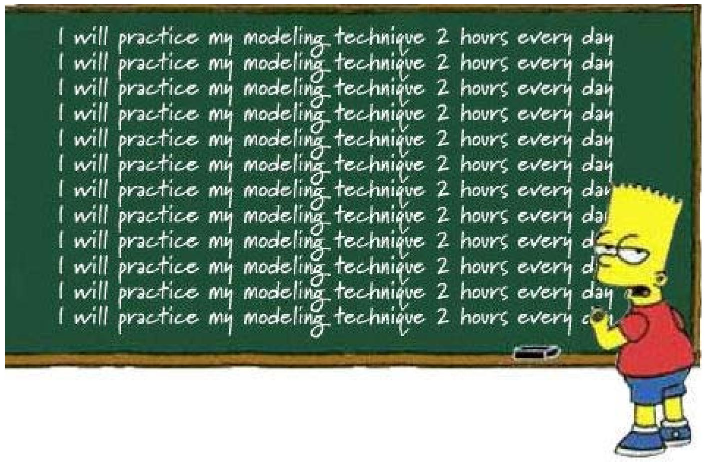
With optional payload

Result


lhost:4200
Welcome to Simple Store App!
Increment
Decrement
Current Count: 30
Reset Counter
10

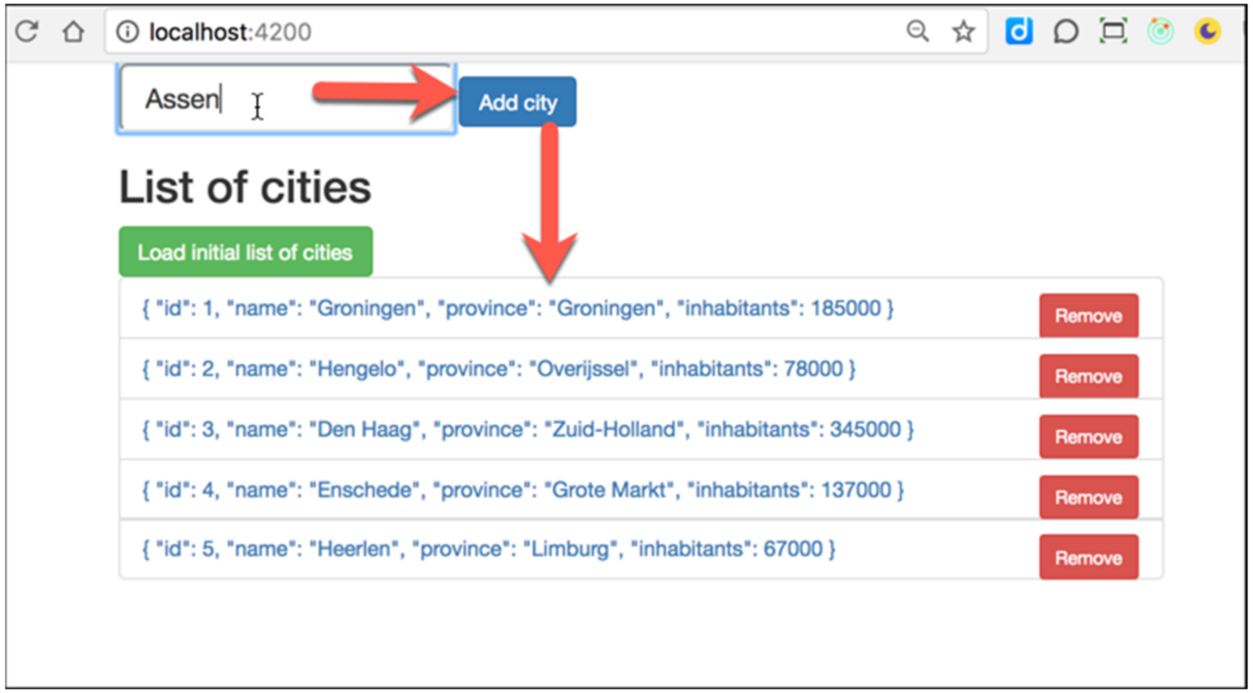
Workshop

- Start from `../203-ngrx-action-payload`
- Create:
  - A new component, holding a textbox
  - Text that is typed in, is send to the store and displayed in the component AND in the counter-component
- Todo: create a new action, reducer, update the module with it and display in the component.
- The UI and logic is (partly) there, but try from scratch first!



Working with complex types

- Real Life Applications – Complex, custom types
- `../207-ngrx-store-complex-types`
- Start with your Component, then Actions, 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 }
```

Short overview

- Cities are provided via a `CityService` – which holds static data
- In the component we have a `cities$: Observable<City[]>`
- In `ngOnInit()` we initialize the store/current state
- The component now calls methods in the service
  - It doesn’t dispatch actions directly anymore

```
…
ngOnInit(): void {
  // Bind observable this.cities$ to  state from the store
  this.cities$ = this.store.pipe(
    select('cities')
  );
}

// load initial cities on mouseclick
loadCities() {
  this.cityService.loadCities();
}
…
```

Cities.actions.ts

- We compose actions that load, update, add and remove cities from the store.
- They are called from the service
- Make sure to understand the `props<>()` call

```
…
// 1. Action for adding one or more cities in an array to the store
export const loadCities = createAction(
  'CITIES - Load cities',
  props<{ cities: City[] }>()
);
…
```

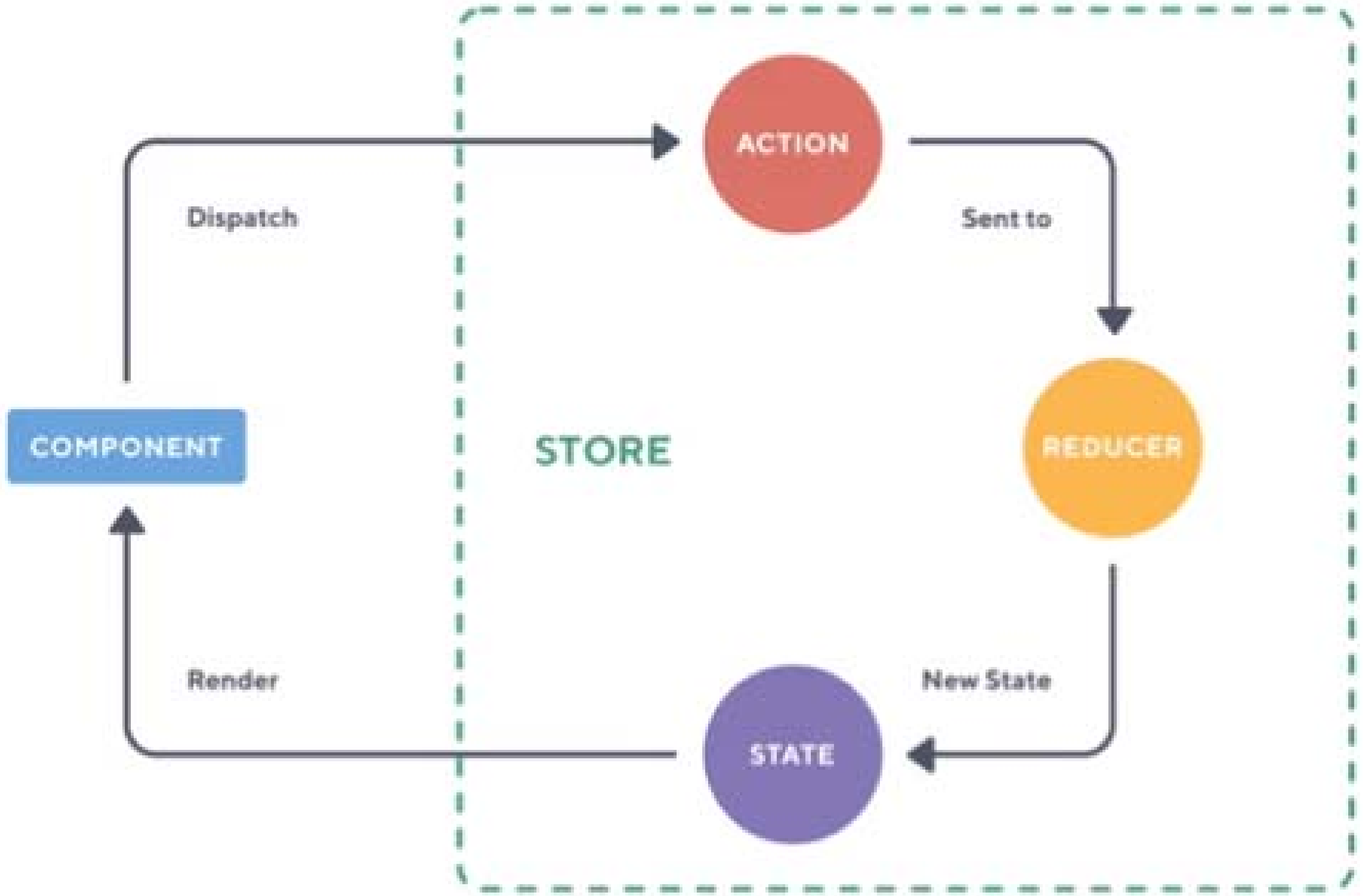
Cities.reducer.ts

- The reducer now works with complex types
- Make sure to understand the rest operator like …payload.cities

```
const reducer = createReducer(
  initialState,
  on(loadCities, (state, payload) =>
                  [...payload.cities]),
  on(removeCity, (state, payload) =>
                  [...state.filter(city => city !== payload.city)])
);
```

REDUX ARCHITECTURE

One-way dataflow



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

Workshop

- Start from `../207-ngrx-store-complex-types`
- Create new Actions and Reducer functions. Goal: Add a new City to the store
  - Update `cities.actions.ts`
  - Update `cities.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 update the name of the City