NGRX

THERE IS A REDUCER IN MY SOUP

About me

Google Developer Expert
Telerik Developer Expert
Digital McKinsey
@chris_noring

NGRX IS:

An Angular implementation of Redux

Why do we care about Redux?

If you experience the following symtoms you might need Redux:

- A feeling that the state is spread out
- There are issues with updating state
- Some things keep changing the state but you don't know who or what
- Un an explainable itch

Solution: a single source of truth with reducers guarding state change. Also enhanced predictability with immutable data structures

CORE CONCEPTS

- Store, our data store
- Reducer, a function that takes state + action and produces a new state
- Selector, selects a slice of state
- Action, an intention of state change
- Action creator, produces an intention that may include data

Typical Store content, just an object

```
{
  counter: 'a value',
  jedis: [{ id: 1, name: 'Yoda' }],
  selectedJedi: { id: 1, name: 'Yoda' }
}
```

REDUCER

NEW STATE = STATE + ACTION

Let's take some reducing examples:

BANANA + MILK + ICE CREAM =

MILKSHAKE

PIZZA + HAMBURGER + ICECREAM =

STOMACH ACHE

Mathematical function, immutable, just a calculation

```
//mutating
var sum = 3;
function add(a) { sum += a; return sum; }

add(5); // 8
add(5); // 13

// immutable
function computeSum(a,b) { return a + b; }

computeSum(1,1); // 2
computeSum(1,1); // 2
```

Immutable = Predictability

A reducer looks like the following:

```
function reducer(state, action) { /* implementation */ }

state, previous/initial state

action = {
  type: 'my intent, e.g ADD_ITEM',
  payload: { /* some kind of object */ }
}
```

state + 1, instead of state +=1, immutable

```
function counterReducer(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
}
```

Usage

```
let initialState = 0;
let state = counterReducer(initialState,{ type: 'INCREMENT' })
// 1
state = counterReducer(state, { type: 'INCREMENT' })
// 2
function counterReducer(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
}
```

A list reducer

```
function jediReducer(state = [], action) {
  switch(action.type) {
    case 'ADD_JEDI':
      return [ ...state, action.payload];
    case 'REMOVE_JEDI':
      return state.filter(
        jedi => jedi.id !== action.payload.id);
    default:
      return state;
  }
}
```

Usage, list reducer

```
let state = jediReducer([], {
  type: 'ADD_JEDI',
  payload: { id: 1, name: 'Yoda' }
});
// [{ id: 1, name: 'Yoda' }]

state = jediReducer(state, {
  type: 'REMOVE_JEDI',
  payload: { id: 1 }
});
// []
```

An object reducer

```
let initialState = {
  loading: false,
  data: [],
  error: void 0
};
function productsReducer(state = initialState, action) {
  switch(action.type) {
    case 'FETCHING_PRODUCTS':
      return { ...state, loading: true };
    case 'FETCHED_PRODUCTS':
      return { ...state, data: action.payload, loading: false
    case 'FETCHED_PRODUCTS_ERROR':
      return { ...state, error: action.payload, loading: false
  }
}
```

Build a new object based on old + change with ...spread

Object reducer, usage

```
let state = productsReducer(initialState, {
  type: 'FETCHING_PRODUCTS'
});
try {
  let products = await getProducts(); // from an endpoint;
  state = productsReducer(state, {
    type: 'FETCHED_PRODUCTS',
    payload: products
  });
} catch (error) {
  state = productsReducer(state, {
    type: 'FETCHED_PRODUCTS_ERROR',
    payload: error
  });
}
```

Handling an AJAX case, can show spinner when, loading = true

A simple store

```
class Store {
  constructor(initialState) { this.state = initialState; }
  dispatch(action) {
    this.state = calc(this.state, action);
  }
  calc(state, action) {
    return {
      counter: counterReducer(state.counter, action),
      jedis: jediReducer(state.jedis, action)
    }
  }
}
```

Usage, store

```
let store = new Store({ counter: 0, jedis: [] });
store.dispatch({ type: 'INCREMENT' });
// { counter: 1, jedis: [] }
store.dispatch({
  type: 'ADD_JEDI',
  payload: { id: 1, name: 'Yoda' }
});
// { counter: 1, jedis: [{ id: 1, name: 'Yoda' }] }

store.dispatch({
  type: 'REMOVE_JEDI',
  payload: { id: 1 }
});
// { counter: 1, jedis: [] }
```

Action, an object with a property 'type' and 'payload'

'type' = intent

'payload' = the change

```
{ type: 'ADD_JEDI', payload: { id: 1, name: 'Yoda'  } }
```

Action creator, function that creates action

```
const addJedi = (id, name) =>
({ type: 'ADD_JEDI', payload: { id, name } });

addJedi(1, 'Yoda');
// { type:'ADD_JEDI', payload: { id: 1, name: 'Yoda' } }

//usage
store.dispatch(addJedi(1, 'Yoda'));
```

Selector, slice of state

```
class Store {
  constructor(initialState) { ... }
  dispatch(action) { ... }
  calc(state, action) { ... }
  select(fn) {
    return fn(state);
  }
}
```

Selector, definitions

```
const getCounter = (state) => state.counter;
const getJedis = (state) => state.jedis;
```

NGRX

OVERVIEW OF LIBRARIES

- @ngrx/store, the store
- @ngrx/store-devtools, a debug tool that helps you track dispatched actions
- @ngrx/router-store, lets you put the routing state in the store
- @ngrx/effects, handles side effects
- @ngrx/entites, handles records
- @ngrx/schematics

STORE

WHERE THE STATE LIVES

INSTALLATION AND SET UP

```
npm install @ngrx/store --save

// file 'app.module.ts'
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './counter.reducer';
@NgModule({
  imports: {
    StoreModule.forRoot({
      counter: counterReducer
    })
  }
})
export class AppModule {}
```

SHOW DATA FROM STORE

```
// app-state.ts
export interface AppState {
  counter: number;
}
// some.component.ts
@Component({
  template: ` {{ counter$ | async }} `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store.select('counter');
  }
}
```

SHOW DATA FROM STORE, SELECTOR FUNCTION

```
// app-state.ts
export interface AppState {
  counter: number;
}
// some.component.ts
@Component({
  template: ` {{ counter$ | async }} `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store
    .select( state => state.counter);
  }
}
```

DISPATCH DATA

```
@Component({
  template: `
  {{ counter$ | async }}
  <button (click)="increment()">Increment</button>
  <button (click)="decrement()">Decrement</button>
  `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store.select('counter');
  }
  increment() { this.store.dispatch({ type: 'INCREMENT' }); }
  decrement() { this.store.dispatch({ type: 'DECREMENT' }); }
}
```

DISPATCH DATA, WITH PAYLOAD, TEMPLATE

```
@Component({
  template: `
  <input [(ngModel)]="newProduct" />
  <div *ngFor="let product of products$ | async">
    {{ product.name }}
    <button (click)="remove(product.id)">Remove</button>
  </div>
  <button (click)="add()">Add</button>
  `
})
```

CLASS BODY

```
export class SomeComponent {
  products$, id = 0;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store.select('products');
  }
  remove(id) { this.state.dispatch({
    type: 'REMOVE_PRODUCT', payload: { id } })
  }
  add() {
    this.state.dispatch({
      type: 'ADD_PRODUCT',
      payload: { id : this.id++, this.name: newProduct  }
    })
  }
}
```

When our app grows, we can't have all the state in StoreModule.forRoot({})

•

•

Solution is using StoreModule.forFeature()
Let's also find a way to organize files

Set up store in a feature module, StoreModule.forFeature('feature',{})

```
interface CombinedState {
  list: Product[];
  item: Product;
};
const combinedReducers: ActionReducerMap<CombinedState> = {
  list: listReducer,
  item: itemReducer
};
@NgModule({
  imports: [
    StoreModule.forFeature<CombinedState, Action>(
      'products',
      combinedReducers
    )]
  });
```

Organizing your files, Domain approach

```
/feature
  feature.component.ts
  feature.selector.ts
  feature.actions.ts
  feature.reducer.ts
```

Organizing your files, Ruby on Rails approach

```
/feature
  feature.component.ts
/feature2
  feature2.component.ts
/reducers
  feature.reducer.ts
  feature2.reducer.ts
  index.ts
/selectors
  feature.selector.ts
  feature2.selector.ts
  index.ts
/actions
  feature.actions.ts
  feature2.actions.ts
```

Whatever approach you go for, consistency is key

STORE DEVTOOLS

DEBUG LIKE A PRO

INSTALLATION AND SET UP

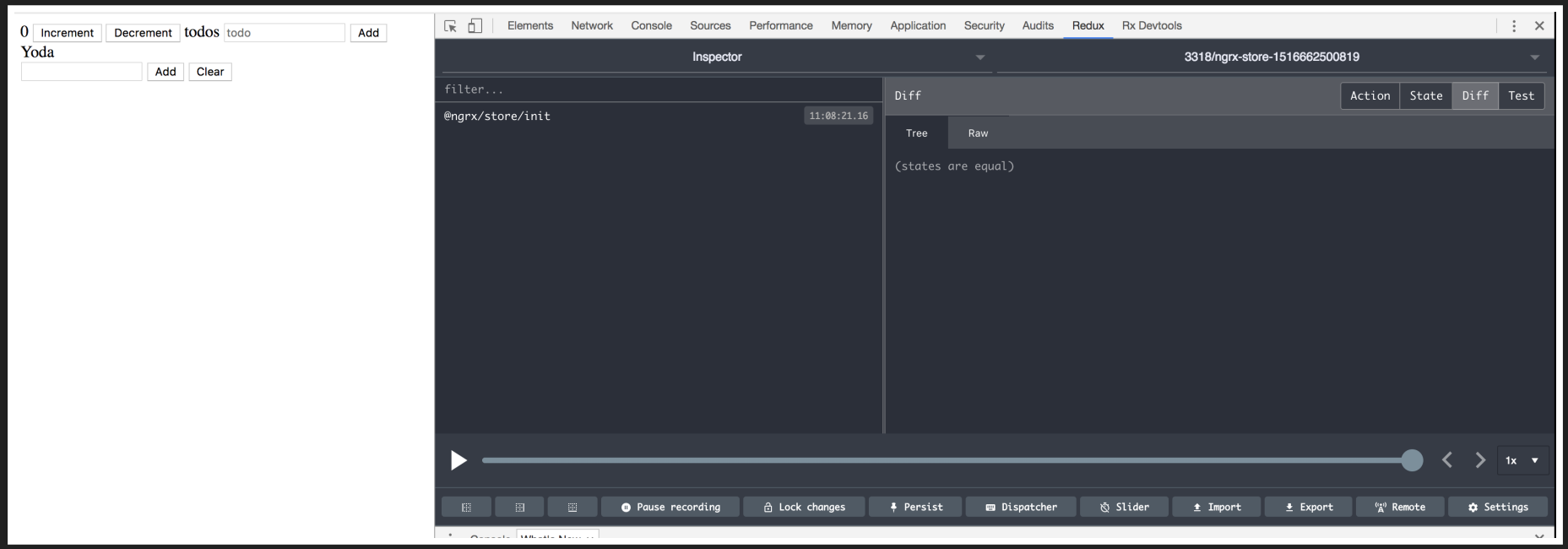
Install lib on NPM and download chrome extension on http://extension.remotedev.io/

```
npm install @ngrx/store-devtools

import { StoreDevtoolsModule } from '@ngrx/store-devtools';

@NgModule({
  imports: [
    StoreDevtoolsModule.instrument({
      maxAge: 25 // Retains last 25 states
    })]
})
```

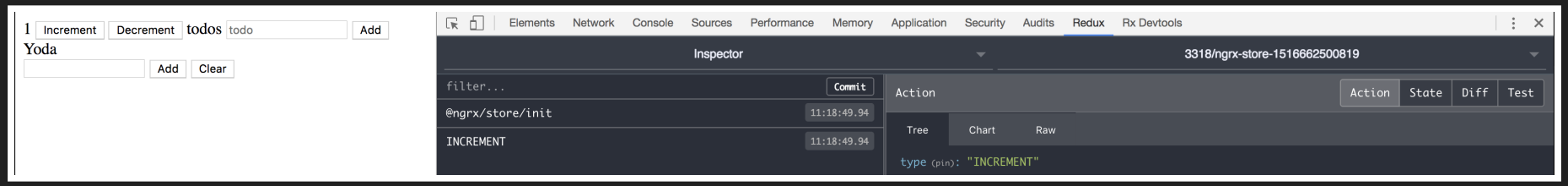
What can we do with it?

- See dispatched actions
- Undo dispatched actions
- Use time travel debugging and move through time, back and forth with a gauge

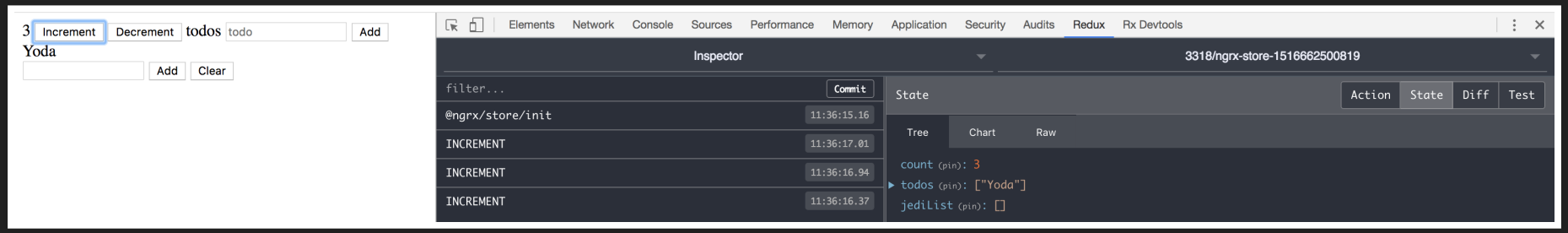
Initial view, we see actions as well as different tabs



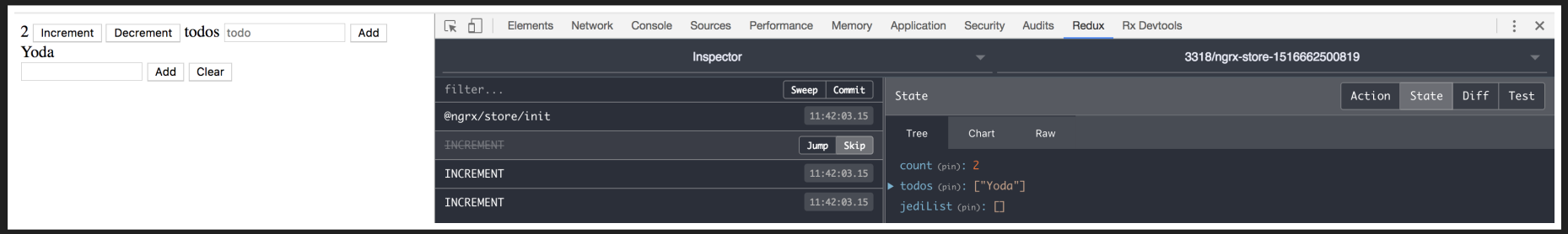
Here we can see the detail of an action, what type etc.



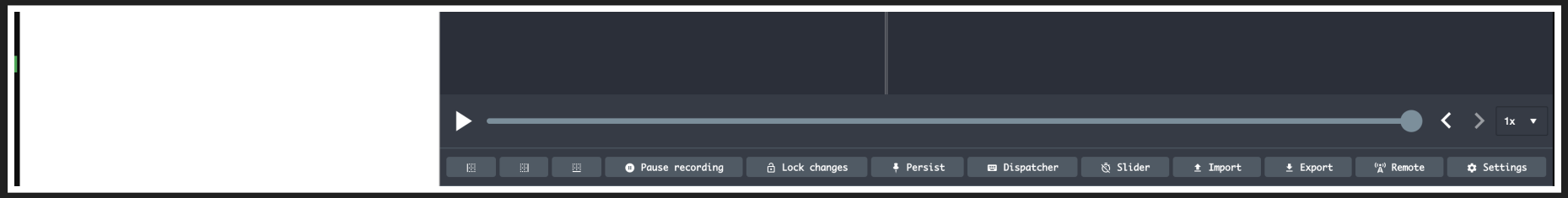
It records all of our actions, here multiple dispatched actions



Here we are undoing/skipping an action, store is recalculated



This gauge enables us to travel through time, back and forth



ROUTER STORE

ENABLING US TO PUT THE ROUTER STATE IN THE STORE

We want to accomplish the following:

- Save the route state to our store
- Customize whats gets saved down

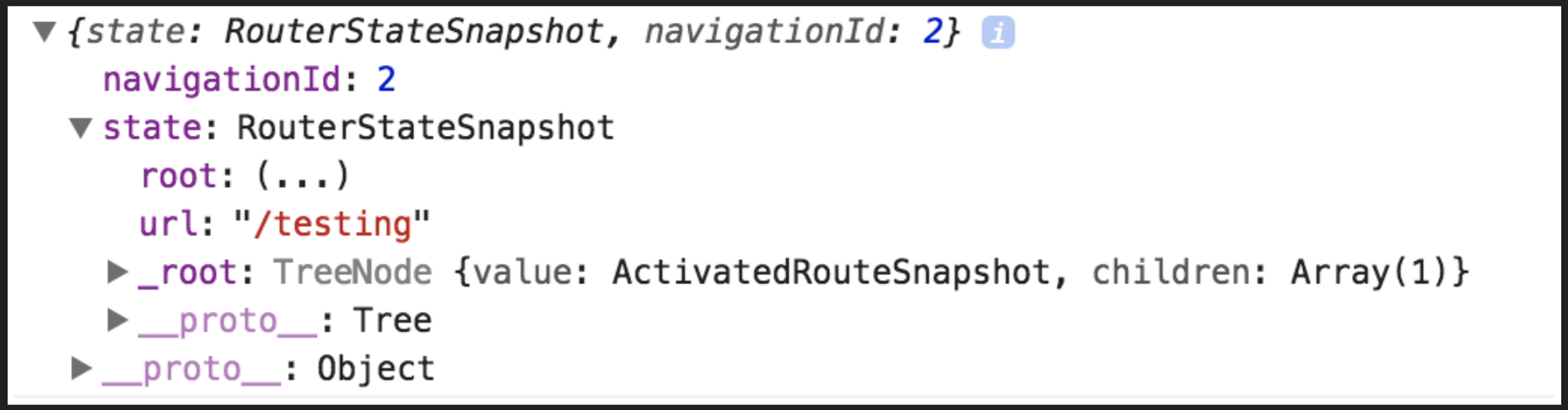
INSTALLATION AND SET UP

```
npm install @ngrx/router-store

import { StoreRouterConnectingModule } from '@ngrx/router-stor
@NgModule({
  imports: [
    StoreModule.forRoot({
      router: routerReducer // this is where our route state g
    }),
    StoreRouterConnectingModule.forRoot({
      stateKey: 'router' // name of reducer key
    })
  ]
})
```

We can listen to this 'router' state like any other state

```
@Component({
  template: ``
})
export class SomeComponent {
  constructor(private state: State<AppState>) {
    // updates every time we route
    this.state.select('router')
    .subscribe(data => console.log(data));
  }
}
```

Might be a bit hard to read



LET'S BUILD OUR OWN ROUTER STATE

The following is of interest:

- The url
- The router parameters
- The query parameters

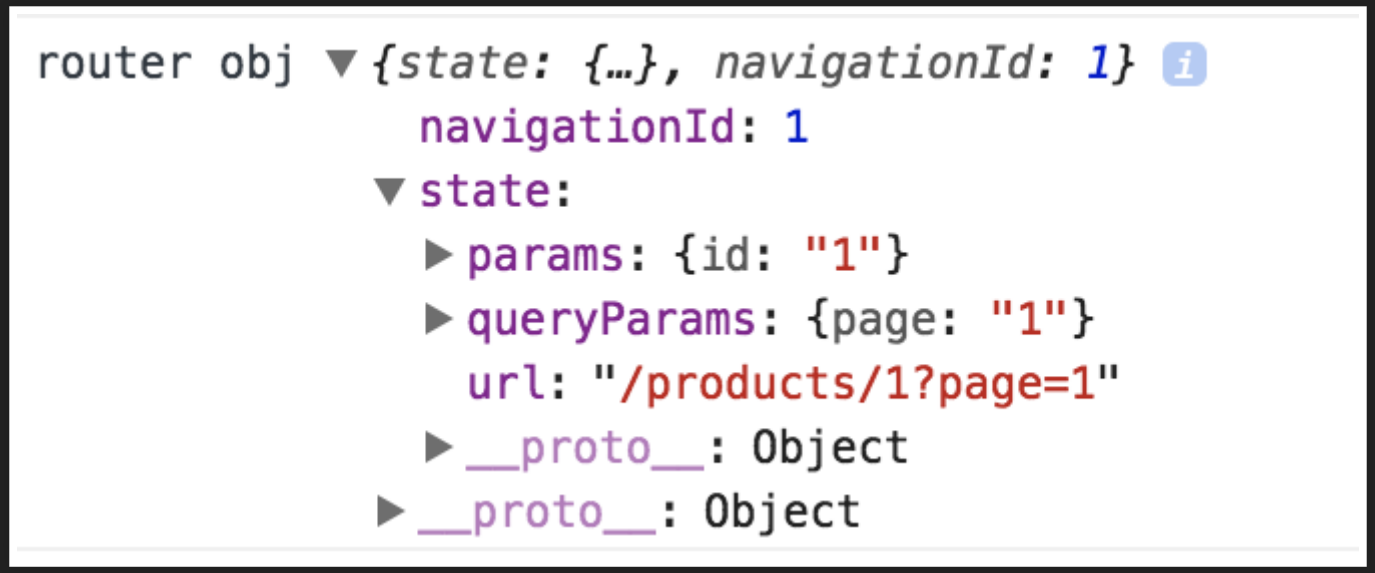
Define a serializer, that saves url, queryParams and router params

```
interface MyState {
  url: string;
  queryParams;
  params;
}

export class MySerializer implements
RouterStateSerializer<MyState> {
  serialize(routerState: RouterStateSnapshot): MyState {
    console.log('serializer');
    console.log('complete router state', routerState);
    const { url, root: { queryParams, firstChild: { params } } }
    return { url, queryParams, params };
  }
}
```

Provide this as the 'new' serializer

```
@NgModule({
  providers: [{
    provide: RouterStateSerializer,
    useClass: MySerializer
  }]
})
```

This is what the router state looks like now, only saves exactly what we want



EFFECTS

HANDLING SIDE EFFECTS

Objective: We want to ensure we can carry out things like accessing resources over the network.

INSTALLATION AND SETUP

```
npm install @ngrx/effects

import { EffectsModule } from '@ngrx/effects';
@NgModule({
  EffectsModule.forRoot([ ... my effects classes ])
})
```

What kind of behaviour do we want?

- Set a loading flag, show spinner
- Do AJAX call
- Show fetched data or error
- Set loading flag to false, hide spinner

NGRX approach

```
try {
  store.dispatch({ type: 'FETCHING_DATA' })
  // state: { loading: true }

  const data = await getData(); // async operation
  store.dispatch({ type: 'FETCHED_DATA', payload: data });
  // state: { loading: false, data: {/* data from endpoint */}
} catch (error) {
  store.dispatch({
    type: 'FETCHED_DATA_ERROR',
    payload: error
  });
}
```

My first effect

```
@Injectable()
export class ProductEffects {
  @Effect()
  products$: Observable<Action> = this.actions$.pipe(
    ofType(FETCHING_PRODUCTS),
    switchMap(
      // ensure we dispatch an action the last thing we do
      action => of({ type: "SOME_ACTION" })
    )
  );

  constructor(private actions$: Actions<Action>, private http:
    console.log("product effects init");
  }
}
```

My first effect - calling HTTP

```
@Injectable()
export class ProductEffects {
  @Effect()
  products$: Observable<Action> = this.actions$.pipe(
    ofType(FETCHING_PRODUCTS), // listen to this action
    switchMap(action =>
      this.http
        .get("data/products.json")
        .pipe(
          delay(3000),
          map(fetchProductsSuccessfully), // success
          catchError(err => of(fetchError(err))) // error
        )
    )
  );
```

ENTITES

REDUCE THAT BORING BOILER PLATE

Install and set up

```
npm install @ngrx/entity

import {
  EntityState,
  createEntityAdapter,
  EntityAdapter } from "@ngrx/entity";
// set up the adapter
const adapter: EntityAdapter<User> =
  createEntityAdapter<User>();
// set up initial state
const initial = adapter.getInitialState({
  ids: [],
  entities: {}
});
```

Install and set up

```
/*
use the adapter methods
for specific cases like adapter.addOne()
*/
function userReducer(
  state = initial,
  action: ActionPayload<User>
) {
  switch (action.type) {
    case "ADD_USER":
      return adapter.addOne(action.payload, state);
    default:
      return state;
  }
}
```

What else can it do for us?

- addOne: Add one entity to the collection
- addMany: Add multiple entities to the collection
- addAll: Replace current collection with provided collection
- removeOne: Remove one entity from the collection
- removeMany: Remove multiple entities from the collection
- removeAll: Clear entity collection
- updateOne: Update one entity in the collection
- updateMany: Update multiple entities in the collection

Further reading:
https://github.com/ngrx/platform/tree/master/docs/ent

SCHEMATICS

BE LAZY AND SCAFFOLD :)

It is a scaffolding library that helps us scaffold out NGRX features

Schematics can help us scaffold the following:

- Action
- Container
- Effect
- Entity
- Feature
- Reducer
- Store

Install and set up

```
// install schematics and prerequisits
npm install @ngrx/schematics --save-dev
npm install @ngrx/{store,effects,entity,store-devtools} --save

// we might need this one as well
npm install --save @angular/cli@latest
```

Scaffold, initial state set up, forRoot({}) and instrument()

```
ng generate store State --root --module app.module.ts --collec

// result
@NgModule({
  declarations: [ ... ],
  imports: [
    BrowserModule,
    StoreModule.forRoot(reducers, { metaReducers }),
    !environment.production ? StoreDevtoolsModule.instrument()
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Scaffold, setting up effects for the App

```
ng generate effect App --root --module app.module.ts --collect

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    ...
    EffectsModule.forRoot([AppEffects])
  ],
  providers: [],
  bootstrap: [ ... ]
})
export class AppModule { }
```

Scaffold, create Action, generates action and test

```
ng generate action User --spec

export enum UserActionTypes {
  UserAction = '[User] Action'
}

export class User implements Action {
  readonly type = UserActionTypes.UserAction;
}

export type UserActions = User;
```

Scaffold, create a component, with store injected

```
ng generate container TodoList

@Component({
  selector: 'app-todo-list',
  templateUrl: './todo-list.component.html',
  styleUrls: ['./todo-list.component.css']
})
export class TodoListComponent implements OnInit {
  constructor(private store: Store<any>) { }
  ngOnInit() {
  }
}
```

DO IT YOURSELF

BUILD YOUR OWN NGRX

What requirements do we have?

- Should be possible to dispatch actions
- State should update when we dispatch
- It should be possible to subscribe to a slice of state
- We should support side effects

BEHAVIOURSUBJECT

SUPPORTS A STATE, CAN BE SUBSCRIBED TO

BehaviorSubject

```
// ctor value = inital value
let subject = new BehaviorSubject({ value: 1 })

subject.subscribe(data => console.log('data', data));
// { value: 1 }
// { prop: 2}

subject.next({ prop: 2 });
```

Merging states with .scan()

```
let subject = new BehaviorSubject({ value: 1 }) // {} inital
subject
  .scan((acc, value) =>({ ...acc, ...value }))
  .subscribe(data => console.log('data', data));

subject.next({ prop: 2 }); // { value: 1, prop: 2 }
```

Implementing the store

```
class Store extends BehaviorSubject {
  constructor(initialState = {}) {
    super(initialState);
    this.listenerMap = {};

    this.dispatcher = new Subject();
    this.dispatcher
    .scan((acc, value) =>({ ...acc, ...value }))
    .subscribe(state => super.next(state));
  }

  dispatch(newState) {
    this.dispatcher.next(newState);
  }
}
```

Usage, store

```
store = new Store();
store.subscribe(data => console.log('state', data));
store.dispatch({ val: 1 });
store.dispatch({ prop: 'string' });
// { val: 1, prop: 'string' }
```

What about slice of state?

```
class Store extends BehaviorSubject {
  constructor(initialState = {}) {
    ...
  }
  dispatch(newState) {
    this.dispatcher.next(newState);
  }
  select(slice) { return this.map[slice] }
  selectWithFn(fn) { return this.map(fn) }
}
```

We need to improve the core implementation, enter storeCalc()

```
const storeCalc = (state, action) => {
  return {
    counter: countReducer(state.counter, action),
    products: productsReducer(state.products, action)
  }
};
```

A retake on our dispatch(), old state, getValue() + action = new state

```
dispatch(action) {
  const newState = storeCalc(this.getValue(),action);
  this.dispatcher.next(newState);
}
```

LETS' TALK ABOUT EFFECTS

- We need to be able to signup to specific actions
- We need to be able to carry out side effects

First let's set up subscription in the store

```
class Store {
  constructor() { ... }
  dispatch() { ... }
  select() { ... }
  effect(listenToAction, listener) {
    if(!this.listenerMap.hasOwnProperty(listenToAction)) {
      this.listenerMap[listenToAction] = [];
    }
    this.listenerMap[listenToAction].push( listener );
  }
}
```

Then ensure the effect happens in dispatch()

```
class Store {
  constructor() { ... }
  dispatch() {
    const newState = storeCalc(this.getValue(),action);
    this.dispatcher.next(newState);
    // tell our listeners this action.type happened
    if(this.listenerMap[action.type]) {
      this.listenerMap[action.type].forEach(listener => {
        listener(this.dispatch.bind(this),action);
      });
    }
  }
  select() { ... }
  effect(listenToAction, listener) { ... }
}
```

use our new effect() method

```
let store = new Store();
store.effect('INCREMENT' ,async(dispatch, action) => {
  // side effect
  let products = await getProducts();
  // side effect
  let data = await getData();
  // dispatch, if we want
  dispatch({ type: 'INCREMENT' });
})
store.dispatch({ type: 'DECREMENT' });
```

SUMMARY

We learned how to:

- Grasp the basics of Redux
- NGRX building blocks
- Use the store
- Leverage the dev tools and its Redux plugin
- Store our router state and transform it
- How we handle side effect like AJAX calls
- Remove boiler plate with Entity
- How to be even lazier with the scaffold tool Schematics
- Upgrading ourselves to Ninja level by learning how to implement NGRX

Further reading:

- Free video course, https://platform.ultimateangular.com/courses/ngrx-st effects
- Redux docs, https://redux.js.org/docs
- Probably the best homepage on it, Brian Troncone, https://gist.github.com/btroncone/a6e4347326749f93

Buy my book (please :)):
https://www.packtpub.com/web-development/architecting-angular-applications-flux-redux-ngrx

Thank you for listening