Using NgRx 4 to Manage State in Angular Applications

Victor Savkin Follow

Jul 6, 2017 · 11 min read

Victor Savkin is a co-founder of nrwl.io, providing Angular consulting to enterprise teams. He was previously on the Angular core team at Google, and built the dependency injection, change detection, forms, and router modules.

. . .

Managing state is a hard problem. We need to coordinate multiple backends, web workers, and UI components, all of which update the state concurrently.

What should we store in memory and what in the URL? What about the local UI state? How do we synchronize the persistent state, the URL,

and the state on the server? All these questions have to be answered when designing the state management of our applications.

In this article I will cover six types of state, the typical mistakes we make managing them in Angular applications, and how to use NgRx to fix the mistakes.

Note: This is a reworked version of "Managing State in Angular Apps", where I replaced my artisan redux impl with NgRx.

Types of State

A typical web application has the following six types of state:

- Server state
- Persistent state
- The URL and router state
- Client state
- Transient client state
- Local UI state

Let's examine them in detail.

The server state is stored, unsurprisingly, on the server and is provided via, for example, a REST endpoint. The persistent state is a subset of the server state stored on the client, in memory. Naively, we can treat the persistent state as a cache of the server state. In real applications though this doesn't work as we often want to apply optimistic updates to provide a better user experience.

The client state is not stored on the server. A good example is the filters used to create a list of items displayed to the user. The items

themselves are stored in some database on the server, but the values of the filters are not.

> *Recommendation: It's a good practice to reflect both the persistent and client state in the URL.*

Applications often have state that is stored on the client, but is not represented in the URL. For instance, YouTube remembers where I stopped the video. So next time I start watching it, it will resume the video from that moment. Since this information is not stored in the URL, if I pass a link to someone else, they will start watching from the beginning. **This is transient client state.**

Finally, individual components can have local state governing small aspects of their behavior. Should a zippy be expanded? What should be the color of a button? **This is local UI state.**

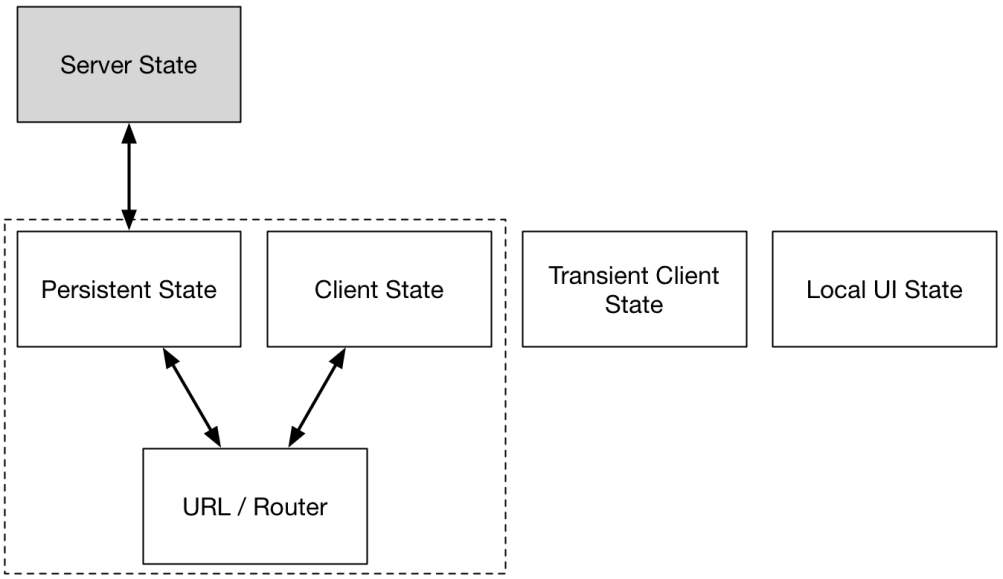
> *Recommendation: When identifying the type of state, ask yourself the following questions: Can it be shared? What is its lifetime.*

State Synchronization

The persistent state and the server state store the same information. So do the client state and the URL. Because of this we have to synchronize them. And the choice of the synchronization strategy is one of the most important decisions we make when designing the state management of our applications.

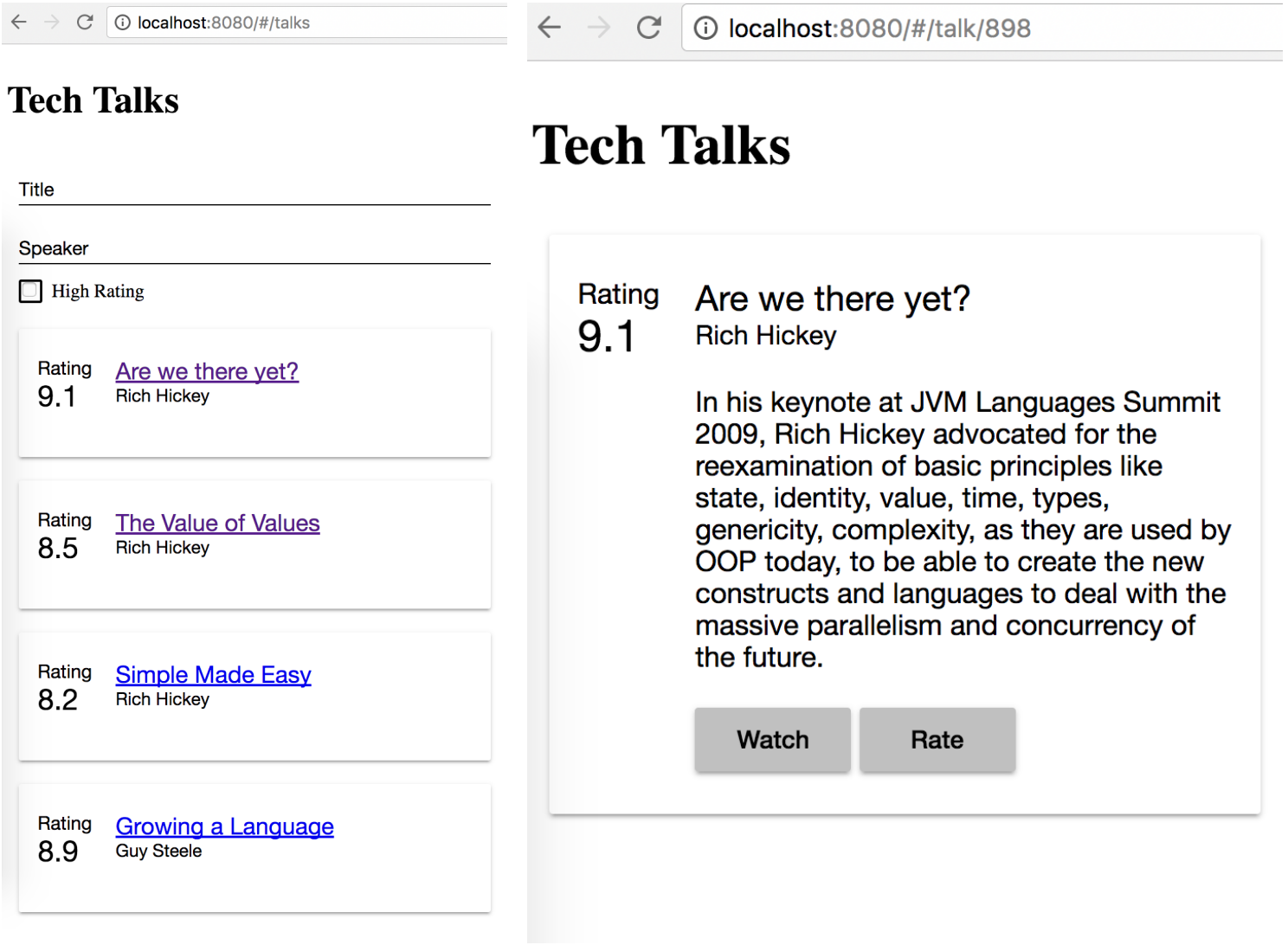
Can we make some of this synchronization synchronous? What has to be asynchronous? Or using the distributed systems terminology: should we use strict or eventual consistency?

These, among others, are the questions we are going to explore in this article.



Example

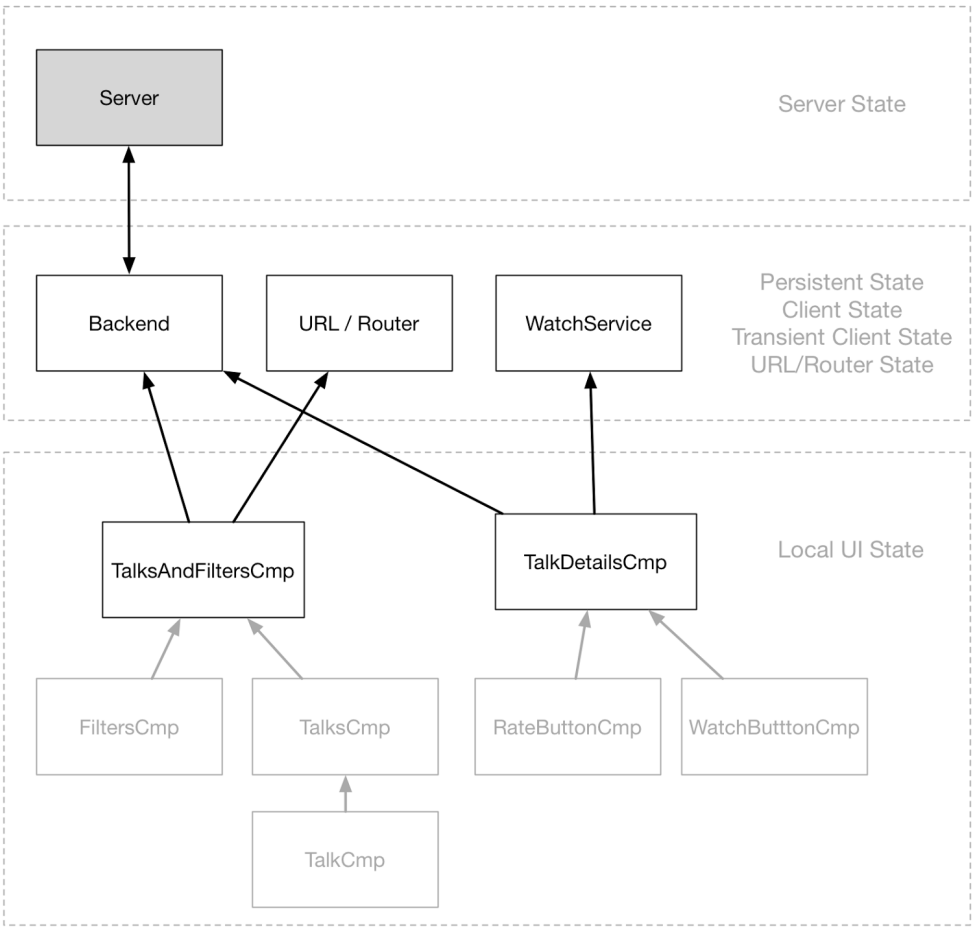
Let's start with an example of what seems to be a reasonably built system. This is an application that shows a list of talks that the user can filter, watch, and rate.



The application has two main routes: one that displays a list of talks, and the other one showing detailed information about each talk.

```
RouterModule.forRoot([
  { path: 'talks',  component: TalksAndFiltersCmp },
  { path: 'talk/:id', component: TalkDetailsCmp }
])
```

This is a rough sketch of this application's architecture.



This is the application's model.

```
type Talk = {
  id: number;
  title: string;
  speaker: string;
  description: string;
  yourRating: number;
  rating: number;
}

type Filters = {
```

And these are the two main components.

```
@Component({
  selector: 'app-cmp',
  templateUrl: './talks-and-filters.html',
  styleUrls: ['./talks-and-filters.css']
})
class TalksAndFiltersCmp {
  constructor(public backend: Backend) {}

  handleFiltersChange(filters: Filters): void {
```

```
@Component({
  selector: 'talk-details-cmp',
  templateUrl: './talk-details.html',
  styleUrls: ['./talk-details.css']
})
class TalkDetailsCmp {
  talk: Talk;

  constructor(private backend: Backend, public watchServic
    route.params.mergeMap(p => this.backend.findTalk(+p['i
  }

  handleRate(newRating: number): void {
    this.backend.rateTalk(this.talk.id, newRating);
```

Both the components do not do any actual work themselves, and instead delegate to *Backend* and *WatchService*.

```
@Injectable()
class Backend {
  _talks: {[id:number]: Talk} = {};
  _list: number[] = [];

  filters: Filters = {speaker: null, title: null, minRatin

  constructor(private http: Http) {}

  get talks(): Talk[] {
    return this._list.map(n => this._talks[n]);
  }

  findTalk(id: number): Observable<Talk> {
    return of(this._talks[id]);
  }

  rateTalk(id: number, rating: number): void {
    const talk = this._talks[id];
    talk.yourRating = rating;
    this.http.post(`/rate`, {id: talk.id, yourRating: rati
  }

  changeFilters(filters: Filters): void {
    this.filters = filters;
    this.refetch();
```

Any time the filters change, *Backend* will refetch the array of talks. So when the user navigates to an individual talk, *Backend* will have the needed information in memory.

The implementation of *WatchService* is very simple.

```
class WatchService {
  watched: {[k:number]:boolean} = {};

  watch(talk: Talk): void {
    console.log("watch", talk.id);
    this.watched[talk.id] = true;
  }

  isWatched(talk: Talk): boolean {
```

Source Code

You can find the source code of the application here.

Types of State

Let's see what manages each type of state.

- *Backend* manages the persistent state (the talks) and the client state (the filters).
- The router manages the URL and the router state.
- *WatchService* manages the transient client state (watched talks).
- The individual components manage the local UI state.

Problems

At a first sight, the implementation looks reasonable: the application logic is handled in the services, the methods are small, and the code looks well-written. But if we look deeper, we will find a lot of problems.

Syncing Persistent and Server State

First, when loading a talk's details, we call *Backend.findTalk*, which reads the data from the in-memory collection. This works when the

user starts with the list view and then navigates around. If the user, however, initially loads the talk details URL, the collection will be empty, and the application will fail. We can workaround it by checking if the collection has the right talk, and if it does not, fetching the information about that talk from the server.

```
class Backend {
  //...
  findTalk(id: number): Observable<Talk> {
    if (this._talks[id]) return of(this._talks[id]);

    const params = new URLSearchParams();
    params.set("id", id.toString());
    return this.http.get(`${this.url}/talk/`, {search: par
```

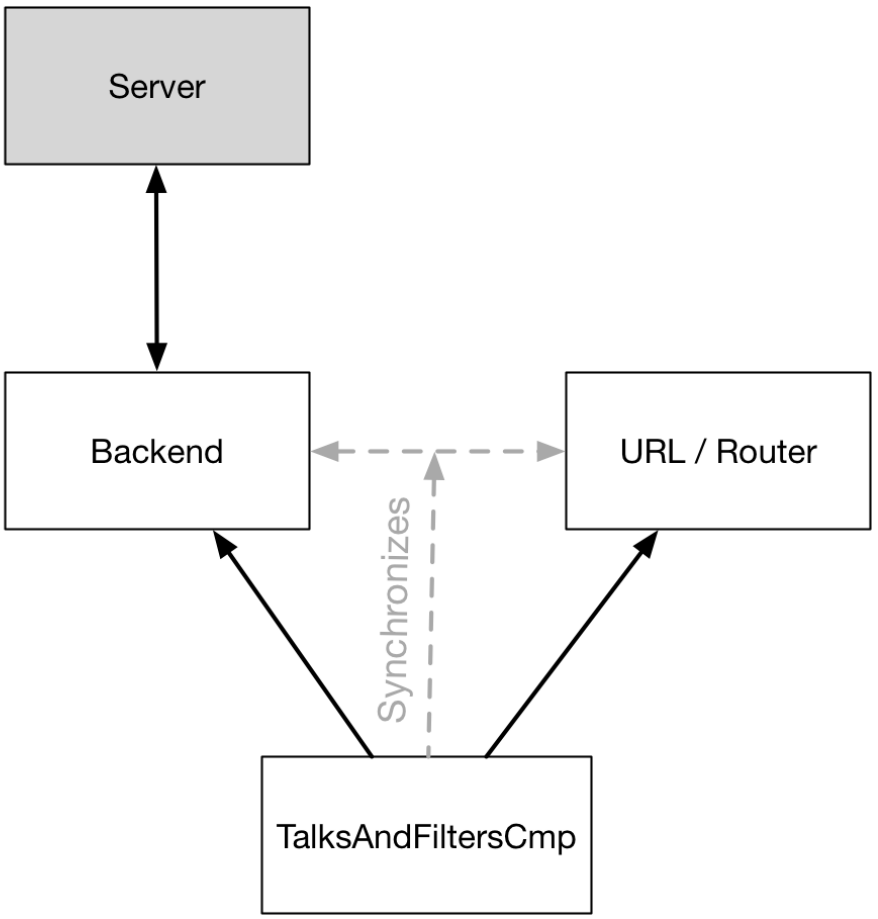
Second, the rate method optimistically updates the passed-in talk object to get a better user experience. The problem is that it doesn't handle errors: if the server fails to update the talk, the client will show incorrect information. Let's fix it by resetting the rating to *null*.

```
class Backend {
  //...
  rateTalk(talk: Talk, rating: number): void {
    talk.yourRating = rating;
    this.http.post(`${this.url}/rate`, {id: talk.id, yourR
      talk.yourRating = null;
      throw e;
    }).forEach(() => {});
```

After these changes, the persistent state and the server state are synced properly.

Syncing URL and Client State

We can also see that changing the filters doesn't update the URL. We can fix it by manually syncing the two.



```
import {paramsToFilters, filtersToParams} from './utils';

@Component({
  selector: 'app-cmp',
  templateUrl: './talks-and-filters.html',
  styleUrls: ['./talks-and-filters.css']
})
class TalksAndFiltersCmp {
  constructor(public backend: Backend, private router: Rou
    route.params.subscribe((p:any) => {
      // the url changed => update the backend
      this.backend.changeFilters(paramsToFilters(filters))
    });
  }
```

Technically this works (the URL and the client state are in sync), but this solution is problematic.

- We call *Backend.refetch* twice. A filter change runs refetch. But it also causes a navigation that will eventually result in another refetch.
- We synchronize the router and *Backend* asynchronously. This means that the router cannot reliability get any information from Backend. Similarly, *Backend* cannot reliability get anything from the router or the URL—it just may not be there.
- We do not handle the case when a router guard blocks navigation. We would update the client state regardless, as if the navigation succeeded.
- Our solution is ad-hoc. We managed to synchronize the router and Backend for this particular route. If we added a new one, we would have to duplicate the logic.

- Finally, our model is mutable. This means that we can update it without updating the URL. This is a common source of errors.

Mistakes

This is a tiny application, and we found so many problems with it. Why was it so hard? What mistakes did we make?

- **We did not separate the state management from the computation and services.** *Backend* talks to the server, and also manages state. Same goes for *WatchService*.
- **We did not clearly define the synchronization strategy of the persistent state and the server.** Even after our changes, the solution seems ad-hoc and not holistic.
- **We did not clearly define the synchronization strategy of the client state and the URL.** Since there are no guards and refetch is idempotent, our fix worked, but it is not a sustainable solution.
- **Our model is mutable**, which makes ensuring any sort of guarantees difficult.

Source Code

You can find the source code of this step here.

Introducing NgRx

Now, let's fix these issues in a more holistic way, and to do so let's refactor our application to use NgRx.

Learn NgRx Basics

I won't explain NgRx in this article as there is a lot of information about it online already. Check out the following posts and talks for more information:

- Talk: Tackling State in Angular Applications
- Tackling State
- Comprehensive Introduction to @ngrx/store

Defining All Building Blocks

We shall start with defining the state of our application.

```
type Talk = { id: number, title: string, speaker: string,
type Filters = { speaker: string, title: string, minRating
type AppState = { talks: { [id: number]: Talk }, list: num
type State = { app: AppState }; // this will also contain

// state
const initialState: State = {
  app: {
    filters: {speaker: "", title: "", minRating: 0},
    talks: {},
```

Then, all the actions our application can perform.

```
type TalksUpdated = { type: 'TALKS_UPDATED', payload: { tal
type TalkUpdated = { type: 'TALK_UPDATED', payload: Talk };
type Watch = { type: 'WATCH', payload: { talkId: number } }
type TalkWatched = { type: 'TALK_WATCHED', payload: { talkI
type Rate = { type: 'RATE', payload: { talkId: number, rati
type Unrate = { type: 'UNRATE', payload: { talkId: number,
```

Note the *RouterAction<State>* action, which we are going to use to make sure our URL and client-state are in sync.

Next, the effects class.

```
class TalksEffects {
  // @Effect() navigateToTalks = ...
  // @Effect() navigateToTalk = ...
  // @Effect() rateTalk = ...
  // @Effect() watchTalk = ...

  constructor(private actions: Actions, private store: Stor
```

Effects classes are where we fetch data, update local storage, and perform other side effects.

Then, the reducer.

```
function appReducer(state: AppState, action: Action): AppS
  switch (action.type) {
    case 'TALKS_UPDATED': // ...
    case 'TALK_UPDATED': // ...
    case 'RATE': // ...
    case 'UNRATE': // ...
    case 'TALK_WATCHED': // ...

    default: return state;
```

Reducer functions are where we manipulate non-local state.

And, finally, we wire everything up in the application module.

```
@NgModule({
  imports: [
    //...
    StoreModule.forRoot({app: appReducer}, {initialState})
    EffectsModule.forRoot([TalksEffects]),
    StoreRouterConnectingModule
  ],
```

Fixing Router and Client-State Synchronization

The first problem we will tackle is the synchronization of the persistent state with the server state and the client state with the URL. And to do that we need to rethink how we interact with the router.

Let's look the issues with the current design once more:

- If the router needs some information from *Backend*, it cannot reliably get it.
- If *Backend* needs something from the router or the URL, it cannot reliably get it.
- If a router guard rejects navigation, the client state would be updated as if the navigation succeeded.
- The synchronization is ad-hoc. If we add a new route, we will have to reimplement the synchronization code there as well.

Router as the Source of Truth

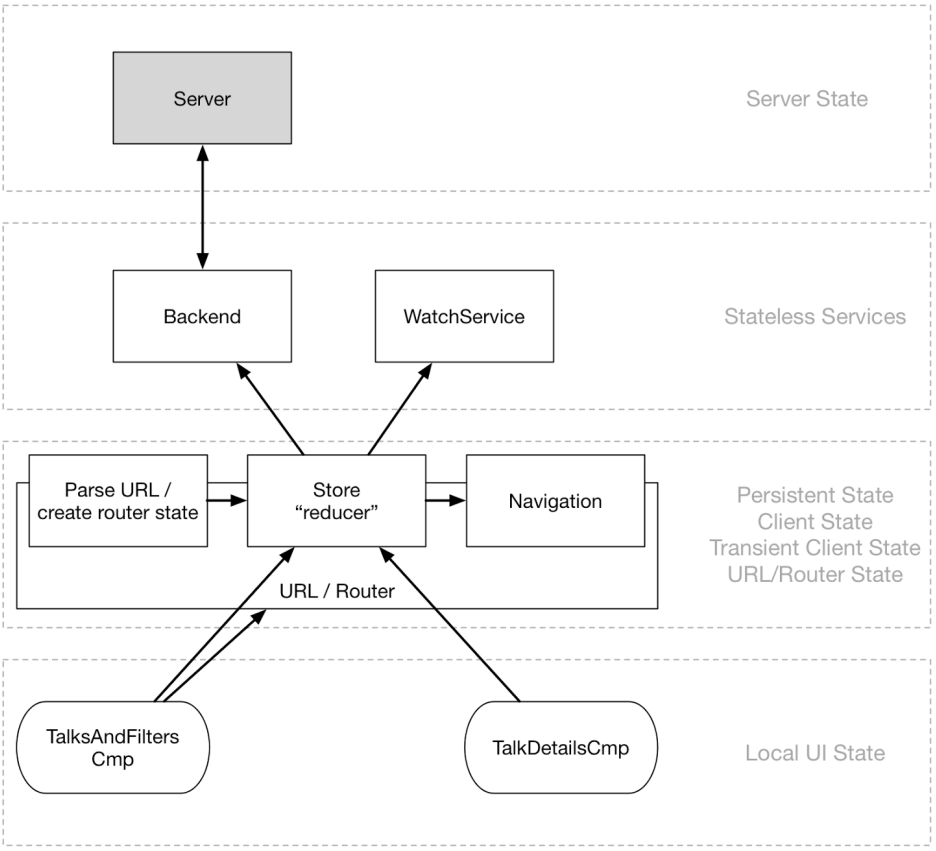
There are many ways to approach the synchronization of the @ngrx/Store and the router. One way is to build a generic library synchronizing the store with the router. It won't solve all of the

problems, but at least the synchronization won't be ad-hoc. Another way is to make navigation part of updating the store. And finally we can make updating the store part of navigation. Which one should we pick?

Since the user can always interact with the URL directly, we should treat the router as the source of truth and the initiator of actions. In other words, the router should invoke the reducer, not the other way around.

Rule 1: Always treat Router as the source of truth

In this arrangement, the router parses the URL and creates a router state snapshot. It then invokes the reducer with the snapshot, and only after the reducer is done it proceeds with the navigation.



And that's what *StoreRouterConnectingModule* does. It will set up the router in such a way that right after the URL gets parsed and the future router state gets created, the router will dispatch a *RouterAction*. Now we just need to handle this action in *TalksEffects*.

```
class TalksEffects {
  @Effect() navigateToTalks = this.actions.ofType(ROUTER_N
    map(firstSegment).
    filter(s => s.routeConfig.path === "talks").
    switchMap((r: ActivatedRouteSnapshot) => {
      const filters = createFilters(r.params);
      return this.backend.findTalks(filters).map(resp => (
    }).catch(e => {
      console.log('Network error', e);
      return of();
    });
```

Let me walk you through it step by step.

First, we get the actions navigating to *talks*.

```
this.actions.ofType(ROUTER_NAVIGATION).
    map(firstSegment).
    filter(s => s.routeConfig.path === "talks")
```

Then, for each of those, we make a request to the backend, and then map the response to a *TALKS_UPDATED* action.

```
  switchMap((r: ActivatedRouteSnapshot) => {
    const filters = createFilters(r.params);
    return this.backend.findTalks(filters).map(resp => ({type
  });
```

Finally, we handle errors by printing them.

```
  catch(e => {
    console.log('Network error', e);
    return of();
  }.
```

Currently, *TalksEffects* has some boilerplate, which we can extract into a helper function.

```
  class TalksEffects {
    @Effect() navigateToTalks = this.handleNavigation('talks'
      const filters = createFilters(r.params);
      return this.backend.findTalks(filters).map(resp => ({ty
    });

    ''
```

Handling TALKS_UPDATED in the Reducer

Since *TalksEffects* is handling our data fetching, what's left is to create a new state object in the reducer.

```
function appReducer(state: AppState, action: Action): AppS
  switch (action.type) {
    case 'TALKS_UPDATED': {
      return {...state, ...action.payload};
    }
    case  'TALK_UPDATED': // ...
    case 'RATE': // ...
    case 'UNRATE': // ...
    case 'TALK_WATCHED': //
```

Analysis

- The reducer can reliably use the new URL and the new router state in its calculations.
- Router guards and resolvers can use the new state created by the reducer, again reliably.
- The solution is also holistic: it is done once. We don't need to worry about syncing the two states when introducing new routes.

Handling Optimistic Updates

Previously, we had an ad-hoc strategy for handling optimistic updates. We can do better than that.

Let's introduce a separate action called *UNRATE*, to handle the case when the server rejects and update.

```
@Injectable()
class TalksEffects {
  @Effect() navigateToTalks = //...
  @Effect() navigateToTalk = //...

  @Effect() rateTalk = this.actions.ofType('RATE').
    switchMap((a: Rate) => {
      return this.backend.rateTalk(a.payload.talkId, a.pay
        switchMap(() => of()). // do nothing when it succe
        catch(e => {
          console.log('Error', e);
          return of({type: 'UNRATE', payload: {talkId: a.p
```

Now, we just need to handle the *RATE* and *UNRATE* actions in our reducer.

```
function appReducer(state: AppState, action: Action): AppS
  switch (action.type) {
    case 'TALKS_UPDATED': // ...
    case  'TALK_UPDATED': // ...

    case 'RATE': {
      const talks = {...state.talks};
      talks[action.payload.talkId].rating = action.payload
      return {...state, talks};
    }
    case 'UNRATE': {
      const talks = {...state.talks};
      talks[action.payload.talkId].rating = null;
```

> *Rule 2: Optimistic updates require separate actions to deal with errors.*

Immutable Data

Note, we changed our model to be immutable. This has a lot of positive consequences.

> *Rule 3: Use immutable data for persistent and client state*

Updated Components

This refactoring simplified our components. Now they only query the state and dispatch actions.

```
@Component({
  selector: 'talk-details-cmp',
  templateUrl: './talk-details.component.html',
  styleUrls: ['./talk-details.component.css']
})
class TalkDetailsComponent {
  talk: Talk;
  isWatched: boolean;

  constructor(private route: ActivatedRoute, private store
    store.select('app').subscribe(t => {
      const id = (+route.snapshot.paramMap.get('id'));
      this.talk = t.talks[id];
      this.isWatched = t.watched[id];
    });
  }

  handleRate(newRating: number): void {
    this.store.dispatch({
      type: 'RATE',
      payload: {
        talkId: this.talk.id,
        rating: newRating
```

Updated Services

After these changes, both *Backend* and *WatchService* became stateless.

```
class WatchService {
  watch(talk: Talk): void {
    console.log("watch", talk.id);
  }
```

Analysis

- **State management and computation/services are separated. The reducer is the only place where we manipulate non-local state.** Talking to the server and watching videos are handled by stateless services.
- **We no longer use mutable objects for persistent and client state.**
- We have a new strategy for syncing the persistent state with the server. We use *UNRATE* to handle errors.

To make it clear our goal was not to use NgRx. It is easy to use NgRx and still mix computation and state management, not handle errors and optimistic updates, or use mutable state. NgRx enables us to fix all of these, but it is not a panacea and not the only way to do it.

> *Rule 4: NgRx should the means of achieving a goal, not the goal*

Also note that we didn't touch any local UI state during this refactoring. Because the local UI state is almost never your problem. Components can have mutable properties that no one else can touch—that's not what we should focus our attention on.

Source Code

You can find the final version of the application here.

Summary

We started with a simple application. Its implementation look reasonable: the functions were small, the code looked well-written. But when we examinded it closer, we noticed a lot of problems. Many of

which are not easy to notice if you don't have a trained eye. We fixed some of them, but many still remained, and the solution was not holistic.

The application had these issues because we did not think through its state management strategy. Everything was ad-hoc. And when dealing with concurrent distributed systems, ad-hoc solutions quickly break down.

We embarked on refactoring the application. We switched to using NgRx and immutable data. This wasn't our goal. Rather this was the means of achieving some of our goals. To solve the rest of the problems, we implemented a strategy connecting the reducer and its store to the router.

And we discovered a few useful rules on the way.

The main takeaway is you should be deliberate about how you manage state. It is a hard problem, and hence it requires careful thinking. Do not trust anyone saying they have "one simple pattern/library" fixing it —that's never the case.

Decide on the types of state, how to manage them, and how to make sure the state is consistent. Be intentional about your design.

. . .

Victor Savkin is a co-founder of Nrwl—Enterprise Angular Consulting.

. . .

. . .

If you liked this, click the ♡ below so other people will see this here on Medium. Follow @victorsavkin to read more about Angular.