State management in Angular with observable store services

Published: January 3, 2018 (last update: March 20, 2019)

tl;dr

Observable stores are a state management solution for Angular apps implemented using RxJS to mimic Redux architecture. This article explains how to create, use and test these observable store services.

Effective state management in front-end development is a challenge, especially in larger and more complex single page applications. Right now, Redux is probably the most popular way of managing state. It is based on a few main ideas:

- One source of truth (app state).
- State is modified in a "pure" way via reducers.
- Reducers are invoked by emitting events to them.
- Interested entities are notified about state updates.

At my day job we have a client facing dashboard application built as a hybrid Angular app (running AngularJS and Angular side by side). AngularJS part of the app stores some state in components' controllers and other in global services (implementing a pub-sub pattern). Every feature manages its state in a slightly different way because there are no clear conventions set about state management. As a consequence, the more features we add, the harder it becomes to ensure the state stays consistent across all components and services.

The process of upgrading to Angular gave us the opportunity to rethink how we tackle state management in the app. We didn't want to introduce another layer of

complexity by adding a state management library to the codebase. New Angular framework, TypeScript, new build system and hybrid app bootstrap already brought a lot of additional complexity to the mix. Instead, we used the ideas from Redux to create **a state management solution that leverages Angular's (and RxJS's) features** to do its job.

One could argue that developing a custom solution for state management introduces additional complexity to the codebase too. It would be naive to dismiss such claims. The difference though is in how much of this complexity is added by developing features using Redux versus observable store pattern. The solution we developed is a really stripped down version of Redux. It does not "prescribe" how to handle async actions, how to combine reducers, how to implement middleware etc. Its only role is to provide a simple API to update state object and to subscribe to its updates. Stores are otherwise just good ol' Angular service classes.

This article explains how one can use the observable store pattern we developed to manage state in Angular apps. The solution was inspired by the following article from Angular University: How to build Angular apps using Observable Data Services.

To showcase the usage of observable stores we'll build a simple app called *Coffee election* that lets its users vote for their favorite type of coffee and add their own coffee type to the list of candidates. The source code is available on GitHub: github.com/georgebyte/coffee-election.

Edit (March 20, 2019):

Many readers have been asking about how to best integrate observable store pattern into an app. I've published an article about scalable Angular app architecture which uses observable stores to manage state. The article and accompanying example app should give you a better understanding of how to use the ideas from this article to build production ready front-end apps. It's available here: Scalable Angular app architecture.

Abstract `Store` class (get it from npm: rxjs-observable-store)

At the core of observable store pattern is the abstract `Store` class. It leverages RxJS to achieve data flow similar to Redux. It is implemented like this:

store.ts

```
import {Observable, BehaviorSubject} from 'rxjs';

export class Store<T> {
    state$: Observable<T>;
    private _state$: BehaviorSubject<T>;

    protected constructor (initialState: T) {
        this._state$ = new BehaviorSubject(initialState);
        this.state$ = this._state$.asObservable();
    }

    get state (): T {
        return this._state$.getValue();
    }

    setState (nextState: T): void {
        this._state$.next(nextState);
    }
}
```

The store's state (`_state$`) is a RxJS `BehaviorSubject` . Changing the state means pushing new state object into the `_state$` stream via the `setState` method. Interested entities can subscribe to state updates by subscribing to the `state$` property. It is also possible to get the current state via the `state` property without subscribing to state updates.

`Store` class provides a **unified interface** for all features' store services to extend. In the next section we'll have a look at how to use the abstract `Store` class to implement an example feature store service.

Features' stores

Feature specific stores are Angular `Injectable` s extending the abstract `Store` class:

coffee-election.store.ts

```
@Injectable()
export class CoffeeElectionStore extends Store<CoffeeElectionState>
  ...
}
```

In the code snippet above note the `CoffeeElectionState` type used when extending the `Store` . Specifying `CoffeeElectionState` as the store type adds correct type definitions to the generic store.

`CoffeeElectionState` is a class representing state object with initial values. In the *Coffee election* example app it looks like this:

coffee-election-state.ts

```
export class CoffeeElectionState {
  candidates: {name: string, votes: number}[] = [];
}
```

One last thing to do to make this simple example work is to add a `super` call to `CoffeeElectionStore` 's constructor in order to correctly initialize the state when creating a new instance of `CoffeeElectionStore` :

coffee-election.store.ts

```
constructor () {
  super(new CoffeeElectionState());
}
```

With the above code in place, each instance of `CoffeeElectionStore` has a way of setting its state and getting the current state or an observable of the state. To make it more useful, additional methods to modify the state (similar to Redux reducers) should be added:

coffee-election.store.ts

```ts
@Injectable()
export class CoffeeElectionStore extends Store<CoffeeElectionState>
  constructor () {
    super(new CoffeeElectionState());
  }

  addVote (candidate: {name: string, votes: number}): void {
    this.setState({
      ...this.state,
      candidates: this.state.candidates.map(c => {
        if (c === candidate) {
          return {...c, votes: c.votes + 1};
        }
        return c;
      })
    });
  }

  addCandidate (name: string): void {
    this.setState({
      ...this.state,
      candidates: [...this.state.candidates, {name: name, votes: 0}
    });
  }
}
```

In the example above `CoffeeElectionStore` 's functionality was extended by defining `addVote` and `addCandidate` methods. In essence, these methods modify the state by pushing new state objects into the observable `state$` stream via the `setState` method.

Note how it is **impossible to modify the state without notifying listeners about the change**. This characteristic of observable stores makes them a perfect fit for

implementing one-way data flow in Angular apps - much like with Redux or a similar state management library.

Using injectable store services

App's state could all be stored in a single global state object. But as the app grows, so does the state object and it can quickly become too big to easily extend it with new features. So instead of storing the whole state in one place, it is better to **split the state into smaller chunks**. A good way to split the properties is to group them by feature and extract these groups into separate state objects, managed by corresponding stores.

There are two types of stores that emerge from splitting:

- global stores that contain globally used state,
- component stores that contain the states used by a single component.

To set up a **store containing global state** accessed by different services and components, the store is listed as a provider in a module's providers list (root app module or a feature specific module). This way Angular adds a new global provider to its dependency injector. The state in global stores will be available until the page is reloaded.

app.module.ts

```
@NgModule({
  ...
  providers: [ExampleGlobalStore],
})
export class AppModule {
  ...
}
```

Note that **many global stores can be defined** as providers in app's modules, each managing its own subset of global state. The codebase stays much more maintainable this way, since each store follows the principle of single responsibility.

To use a global store in different parts of the app, the store needs to be defined as their dependency. This way Angular injects the **same instance** of a global store (defined as singleton provider in `AppModule` or any other module) into every component/ service depending on it.

example.component.ts

```
@Component({ ... })
export class ExampleComponent {
  constructor (private exampleGlobalStore: ExampleGlobalStore) {
    // ExampleComponent has access to global state via exampleGloba
  }
}
```

Not all state needs to be global though. **Component specific state** should only exist in memory if a component is using it. Once user navigates to a different view and the component is destroyed, its state should be cleaned-up too. This can be achieved by adding the store to a list of component's providers. This way we get "self-cleaning" stores that are kept in memory as long as components using them are kept in memory.

example.component.ts

```
@Component({
  ...
  providers: [ExampleComponentStore],
})
export class ExampleComponent {
  ...
}
```

Private component stores are used in the same way as global stores by defining them as dependencies in the components' constructors. The key difference is that these **component specific stores are not singletons**. Instead, Angular creates a new instance of the store each time a component depending on it is created. As a consequence, multiple instances of the same component can be present in the

DOM at the same time, each one of them having its own store instance with its own state.

Subscribing to state updates in components and services

Once a store instance is injected into a component or service, this component/service can subscribe to state updates. In the example of `coffee-election` component, subscribing to state updates looks like this:

coffee-election.component.ts

```
@Component({ ... })
export class CoffeeElectionComponent implements OnInit {
  constructor (private store: CoffeeElectionStore) {}

  ngOnInit () {
    this.store.state$.subscribe(state => {
      // Logic to execute on state update
    });
  }
}
```

It is also possible to only subscribe to updates of a subset of state:

```
this.store.state$
  .map(state => state.candidates)
  .distinctUntilChanged()
  .subscribe(candidates => {
    // Logic to execute on state.candidates update
  });
```

Note that these **subscriptions must be cleaned up** before a component is destroyed in order to prevent memory leaks. We won't go into details about unsubscribing in this article. Check out this topic on Stack Overflow to learn more.

Subscribing to state updates in components' templates

In case a component doesn't execute any logic on state update and it only serves as a proxy to pass the state to its template, Angular provides a nice shortcut to subscribe to state updates directly from templates via the `async` pipe. `ngFor` in the example below will redraw a list of candidates every time the state is updated.

coffee-election.component.html

```
<ul>
  <li *ngFor="let candidate of (store.state$ | async).candidates">
    <span>{{ candidate.name }}</span>
    <span>Votes: {{ candidate.votes }}</span>
    <button (click)="store.addVote(candidate)">+</button>
  </li>
</ul>
```

These subscriptions to state updates via `async` pipes are **automatically cleaned up** by the framework upon destroying the component.

Unit testing the store

Testing state modifying store methods is pretty straightforward. It consists of three steps:

1. Creating an instance of the tested store and setting up mocked initial state.
2. Calling a store's method the test is testing.
3. Asserting the method updated the state correctly.

In practice unit tests to test the store from the *Coffee election* example look like this:

coffee-election.store.spec.ts

```
describe('CoffeeElectionStore', () => {
  let store: CoffeeElectionStore;

  const MOCK_CANDIDATES = [{name: 'Test candidate 1', votes: 0}, {n

  beforeEach(() => {
```

```
    TestBed.configureTestingModule({
      providers: [CoffeeElectionStore]
    });

    store = new CoffeeElectionStore();
    store.setState({
      candidates: MOCK_CANDIDATES
    });
  });

  it('should correctly add a vote to a candidate', () => {
    store.addVote(MOCK_CANDIDATES[1]);
    expect(store.state.candidates[0].votes).toBe(0);
    expect(store.state.candidates[1].votes).toBe(6);
  });

  it('should correctly add a candidate', () => {
    store.addCandidate('Test candidate 3');
    expect(store.state.candidates[2].name).toBe('Test candidate 3')
  });
});
```

Conclusion

The purpose of this article was to present how one can leverage the built in features of Angular framework to implement a simple yet powerful state management solution. The provided *Coffee election* example app is very simple, but the concepts it demonstrates can be used to successfully manage state in much bigger and more complex apps. At Zemanta we used observable store services to implement a rather complex feature and since the experiment worked out great we will continue to use such stores in our app going forward.

If you are interested in how to use the ideas from this article to build production ready front-end apps, you should check out my article about Scalable Angular app

architecture.

Edit (March 4, 2018):

Some readers pointed out that different state management libraries (e.g. ngrx) provide the same functionality as observable store services and were wondering why one may use observable store pattern instead of these libraries.

Observable store pattern I described here is a much simpler solution in my opinion.

I used ngrx in the past and I think it is a really good library for state management. But it also takes longer to learn it, because of all the features it supports.

As it turned out in our case at Zemanta, one may not need a full blown state management library to manage state even in larger applications. This "stripped down" implementation of Redux pattern covers pretty much all of Zemanta's app use cases without introducing much additional complexity. I believe less complexity comes from the fact that observable store services heavily depend on Angular features (dependency injection, async pipes etc.) to do a lot of heavy lifting (e.g. cleaning-up unused state when components are destroyed, creating new instances of stores when needed etc.).

...

PS: Let's connect on Twitter (I have no product to push on you and my feed stays clean and interesting 😇).

Name

ooo-2 • 2 years ago

The cleanest and most lightweight Redux impl. I have ever seen. Thanks a lot! The cool thing, you can also mutate store slices if you want. Mutable stores make sence for real-time apps with big, frequently updated data or for apps which data are not reflected in UI components, but rendered into the canvas (e.g. with WebGL).

11 ^ | v • Reply • Share ›

M • a year ago

Good tutorial.
Just wondering how I would set this up in a way where I can make http requests and then add that information into the store to be used in templates.

^ | v • Reply • Share ›

George Byte Mod ➚ M • a year ago

Hi, I believe my other article will help you on this one: https://georgebyte.com/scal...

^ | v • Reply • Share ›

Stephane • a year ago

I have implemented that neat data store but I get an ExpressionChangedAfterItHasBeenCheckedError message in the console when, from within the ngOnInit method, I add items in an array set as state in the data store, later traversed with an ngFor loop in the view. The items are added in the data store and the view async-pipe observable of the ngFor receives them fine. I've been looking for the root cause for some time :-) I guess it keeps me busy here quarantined in the South of France.

^ | v • Reply • Share ›

Stephane ➚ Stephane • a year ago

I could solve my issue by using an explicit `subscribe()` call in the controller instead of using the piped `async` keyword in the view, as well as do an explicit call to the change detection `this.changeDetector.detectChanges();` in the `subscribe()` block.

See the SO question for further information.

^ | v • Reply • Share ›

George Byte Mod ➚ Stephane • a year ago

George Byte Mod › Stephanie • a year ago

Hi, I honestly can't figure out what's going on here. We've never run into this error with my team when using observable stores in our Angular app. We were using Angular v8, so this might be the difference. Is there any chance you could clone your codebase, downgrade Angular to v8 and test if the error is still present?
Also, could you maybe provide the minimal source code of a store that reproduces this issue and a snippet of how the items are added in that data store?

^ | v • Reply • Share ›

Jerry • a year ago

While upgrading the typescript version and angular (version 8) to the latest along with other packages the rx-js-observable store is causing an issue in my application. Here is the error details: node_modules\rxjs-observable-store\node_modules\ts-toolbelt\out\types\src\Function\Length.d.ts(4,10): Error TS2440: Build:Import declaration conflicts with local declaration of 'Length'.

It will be great if you can update the ts-toolbelt package to the latest and fix the error, So that I can continue using the observable store in my angular app

^ | v • Reply • Share ›

George Byte Mod ➚ Jerry • a year ago

Hi, thanks for reporting the issue. I released version 2.0.2 of rxjs-observable-store with updated npm dependencies. Hopefully this fixes the error.

^ | v • Reply • Share ›

Jerry ➚ George Byte • a year ago

Thanks for the quick update.

The issue which I reported seems to be fixed in this new version. Unfortunately It's throwing a new error now. Here is the details: ERROR in ../node_modules/rxjs-observable-store/lib/store.d.ts:8:9 - error TS1086: An accessor cannot be declared in an ambient context.
8 get state(): S;

As this was not there in the previous version I think it's something in new version causing this

think it's something in new version causing this issue.

Reply • Share ›

George Byte Mod ➚ Jerry • a year ago

Seems like I updated TS to a version unsupported by Angular 8. In the version 2.0.4 of the package I downgraded TS to 3.5.3 and tested the package in a newly created Angular 8 app. It should work now.

1 Reply • Share ›

Jerry ➚ George Byte • a year ago

Great! Now it's working fine...

Reply • Share ›

Ajit Abraham • 2 years ago

Hi Jure, I am a newbie to observables.

From your example, I created a simple store with couple of properties like name, age etc (not an array like the candidate array in your example)

In the state.ts, the purpose of the state() method is to return the current value of the store.

But I *am* able to change the value of the store by
this.store.name = "i have changed the name"
in the coffee-election.store.ts
and all the subscribers do get the changed value!!!

Even for something similar to your candidate array object, I could push in a new value like this :
this.state.names.push({name:newName});

I don't know what I am doing wrong - but I am thinking, if I am able to change the value of the store in this way, why do I need the next() method?

Can you replicate the above in your code?

Regards

Ajit

Reply • Share ›

George Byte Mod ➚ Ajit Abraham • 2 years ago • edited

Hey!

Hey!

I believe what happened in your case is that the rendered list of candidates re-rendered after `this.state.candidates.push({name:newName})` mutated the state directly (by reference). Angular's change detection detected the change and re-rendered the template with one additional candidate. This is how Angular works out of the box, but you can only detect changes in your templates' code using Angular's change detection.
If you for example wanted to execute a callback method in a component/service on state updates (subscribing to updates via `this.store.state$.subscribe(state => {...})`), the callback wouldn't get executed because the state was "silently" updated. And that's why calling `next()` is required - to notify all components/services subscribed to state updates via `this.store.state$.subscribe()` about the change and trigger their callbacks.

Hopefully this answered your question.

 • Reply • Share ›

Ajit Abraham ➚ George Byte • 2 years ago

Hello Jure,
Thanks for your answer.
So, what **@ooo-2** posted earlier was correct. Your store is a mutated store.

Well in my case - so far - I have yet to deal with concepts like callback methods - so I am happy with the way your code works currently.
Thanks once again for sharing your code.

Ajit

 • Reply • Share ›

George Byte Mod ➚ Ajit Abraham
• 2 years ago

You are correct, the state in store is not immutable. It's up to the developer to update the state in an immutable way.
Check out this comment where we discussed about a similar topic:
http://disq.us/p/1sjnxri.

 • Reply • Share ›

Ajit Abraham ➚ George Byte
• 2 years ago • edited

Thanks Jure

Thanks Bure.
As I mentioned earlier - The way it works currently, is perfect for me (until my code gets complex in future) as I had some trouble understanding the spread operator for its use in the next() method.

To make sure that only the CoffeeElectionStore can change the store, I changed the get state().... to, protected get state().... so that there is a single class from where the mutation could be made (is that called 'single source of truth' ?).

Ajit

^ | v • Reply • Share ›

George Byte Mod ➚ Ajit Abraham
• 2 years ago

I encountered a problem with making the state update method protected - it is harder to setup mocked state in tests since you can't call the method in tests directly:

```
const testStore = new TestStore();
testStore.setState({}); // Can't call a
protected method from abstract class
(Store<t>) like this
```

That's why I kept it public.

^ | v • Reply • Share ›

Ajit Abraham ➚ George Byte
• 8 months ago • edited

I realised the folly in keeping the method protected. So as shown in your code, I have kept it public

Also, the map and distinctUntilChanged in no longer available in Observable. It has been moved to operators

This is what I did

```
import { map, distinctUntilChanged } from
'rxjs/operators';
this.store.state$
.pipe((map(state => state.candidates),
```

```
distinctUntilChanged())
.subscribe(candidates => {
// Logic to execute on state.candidates
update
});
```

Its working for me. But please point out any gotchas.
Thanks

Ajit

 • Reply • Share ›


This comment was deleted.

George Byte Mod ➚ Guest • 2 years ago • edited

In a previous version I had it implemented via TypeScript getters which called `this._state$.asObservable()` and returned the result (https://github.com/jurebajt...).
The problem with that approach was that too many instances of observables were created because each access of `store.state$` property created a new observable. Subscriptions that should be made to the same observable were messed up because of `store.state$` always referencing a different observable instance. This was especially problematic when using the `async` pipe in Angular templates. The fix was to always return the same internal instance so that the reference always stays the same. And yes, `asObservable` is used just to prohibit publishing new values directly into `_state$` subject.

 • Reply • Share ›

NetanelBasal • 2 years ago

Or just use Akita :) https://netbasal.gitbook.io...

 • Reply • Share ›

Nicolas Forney ➚ NetanelBasal • 2 years ago

Akita is a really interesting tool. Great work by the way ;). But it's more about willing to add a new framework/pattern to your existing app/s or go with what is available with the core angular dependencies. Who knows if you will still maintain Akita in 2 years ;).

 • Reply • Share ›

George Byte Mod ➚ Nicolas Forney • 2 years ago

My thoughts exactly, thanks Nicolas for answering this one :)

Reply • Share ›

NetanelBasal ➔ George Byte
• 2 years ago

Thanks. I will maintain it because it's in use at my company. Using the approach you described is helpful for small applications. In medium-large organizations, you need patterns, plugins, etc. It couldn't scale, IMHO.

Reply • Share ›

George Byte Mod ➔ NetanelBasal
• 2 years ago • edited

I'll admit I still need to check out Akita more in depth to make any meaningful argument. But I'm not sure I agree this approach can't scale. I believe because it leans heavily on Angular framework, it can scale pretty good to support development of larger SPAs. At least this is the case at my company where we primarily use it too build our main dashboard which isn't the most basic app. Sure, we also had to develop an architecture that works for us (described here: https://jurebajt.com/scalab...), but for state management we don't need much more. When used "correctly", Angular provides a really solid foundation for scalable apps almost out of the box.

Reply • Share ›

doroncy • 2 years ago

Very nice implementation, i created something similar as well, and i wanted to know if now about a year after you created this design pattern, are you still use it? have you found some problematic issues with this pattern? also i am dealing with an issue that some of the stores needs to be invalidated (refresh their data) due to some other changes in different services, have you had such issues? and if so how did you solved it. thanks again

Reply • Share ›

George Byte Mod ↗ doroncy • 2 years ago • edited

Hello!
Yes, we still use this pattern at my company and haven't had any major issues with it. There are some cases where code can get a bit repetitive, because observable stores provide a very simple api and one needs to implement the rest of the functionality. Bigger state management libraries abstract this away, which has its benefits as well as its downsides.

If I understand your issue correctly, there are two ways to solve is:
- the store that should refresh its data can subscribe to state observable (if available) from the other store/service and refresh its data in subscription callback,
- in case the other service doesn't have an observable state, the store could expose a method to trigger data reload which the service will be able to call when appropriate.

I prefer the first approach as it introduces a bit less coupling between different parts of the codebase. A service from the second option has to know that after it executes some code it also needs to call a method from another place, which has nothing to do with it. This becomes even more problematic, when methods from multiple other stores need to be called.

In my opinion, the best solution in this case is to create a new core service with observable state. Its "state" is just a stream of updates. When a service from your example changes, it should also push an update into the stream of this new core service. The store from your example would then be subscribed to these updates (the core service's observable state) and would refresh its data in subscription callback. Doing it this way, there is still some coupling present, but this kind of coupling is less problematic in my opinion.

 | • Reply • Share ›

Doron Cyngiser ↗ George Byte • 2 years ago

Hi, Thanks for the quick reply.
I handled it in a very similar approach, i created a globalBus service which have only an rxjs subject and and enum of the valid event types, some of the service methods push new events in that globalBus (for example: employeeUpdated, etc...)

globalBus (for example: employeeUpdated, etc...) and now other services in the application can inject the globalBus and react to specific global events. the combination of both functional and reactive events approach with your style of mini stores is really powerful and i think fit to most applications.

Reply • Share ›

Bipin Shrestha • 2 years ago

Very useful! Gonna give this a try. Thanks a lot! :)

Reply • Share ›

Dheeraj Roy • 2 years ago

Thanks a lot. This is very useful. I spent hours learning redux to get in confusion whether it is worth implementing for a mid-sized app.

Reply • Share ›

Peter Kristensen • 2 years ago

Will definitely try this out! I, too, feel that the full RxJs store is often too complex and adds an extra layer instead utilizing what Angular already provides us with. But it depends on the complexity and type of app, I believe... Anyways, this is awesome and thank you for writing this article :)

Reply • Share ›

George Byte Mod → Peter Kristensen • 2 years ago

Thank you for your kind words :) I completely agree and don't want to sell observable stores described in this article as the silver bullet for state management. Sometimes a simpler approach like this will get the job done with less complexity and other times a fully-featured library will solve the challenges one faces in a better way. We as engineers have to assess different options and choose the most suitable one (and be prepared to maybe replace it later, if new needs arose).

1 Reply • Share ›

richard roma • 2 years ago

Hi Jure, I'm new to Angular 2+ and am exploring options for project architectures. I like this state management, very lightweight but helps organizing the code without the learing curve and overhead of a big library. But when I use it from npm I have to call super instead of this to access the state members. Any idea what I'm missing?

Reply • Share ›

George Byte Mod ➚ richard roma • 2 years ago

Hey, it is hard for me to guess what could be wrong just from this description. Can you perhaps share some code where this happens? I never had any issues like this when using the abstract store from npm.

Reply • Share ›

salah Alhaddabi • 2 years ago

Dear Jure, extremely nice article.

Would you please give us hints on how to implement side effects using this very effective approach of yours??

Regards,
Salah.

Reply • Share ›

George Byte Mod ➚ salah Alhaddabi • 2 years ago • edited

Hello! Thanks for your comment.

What kind of side effects do you have in mind?

I think the notion of side effect isn't necessarily applicable in the context of observable stores - there's no need (in my opinion) to handle side effects in observable stores in any special way. Store does whatever it needs to do, and at the end it updates the state. Whenever the state update happens, subscribers now about it and can react.

As an example, here's how I would handle a fetch data "side effect" in loadData store method:
https://gist.github.com/jur...

Hopefully this simplified example will help you. I'm currently in the process of writing a longer blog post about how to create a scalable Angular app architecture with observable stores at its core. Accompanying GitHub repo should provide all the insights needed to understand how to use observable store pattern. Stay tuned :)

Reply • Share ›