Angular Service Layers: Redux, RxJs and Ngrx Store - When to Use a Store And Why ?

Angular 2 is just out and we are all starting to use it in our projects, and its such an improvement towards the previous version. The View layer is simpler to learn and use than ever.

But the service layer (also known as the data layer) which is really the functional heart of the application offers many options:

- How should we structure the service layer ?
- Should we use a store ?
- Should we use Redux ?
- Should we use plain RxJs ?
- What about NgRx Store ?

One thing that is getting very popular in the Angular 2 world are store solutions.

They originated in the React world and went through the usual technological adoption curve: mass adoption, realization that its not the ultimate solution for everything, and then settling in using it in certain situations but not others.

Why are stores so popular in React ?

Why have stores become so popular in the React world, is there a specific reason or is it due to a combination of reasons ? Do those reasons also apply to the Angular world, or are there alternative solutions ? What problems do stores solve ?

Did you notice that there is a lot of information on store solutions, but limited information about when should we use them and why ? Let's go over these questions.

Table of Contents

In this post we will cover the following topics:

- Conclusions and Suggestions

When to use Redux or stores in general ?

Stores have originated in the Redux world, so that would be one of the best places to look first, and then take it from there.

Let's take the react-howto guide to the React ecosystem, what are the recommendations ? Here is an important quote:

> *You've probably heard of Flux. There's a ton of misinformation about Flux out there. A lot of people sit down to build an app and want to define their data model, and they think they need to use Flux to do it. This is the wrong way to adopt Flux.*

There is also this well known post by the creator of Redux - You Might Not Need Redux, which can be applied to any store solution in general.

And then there is also this other statement in the React How-To, that seems to apply equally to original Flux, Redux, Ngrx Store or any store solution in general:

> *You'll know when you need Flux. If you aren't sure if you need it, you don't need it.*

Based on this, it looks like stores are not recommended for systematic use by some of their original creators. In these posts we get the impression that the creators seem to fear that stores are seen as a one size fits all solution.

But then we run into posts like this one - I Always Seem to Need Redux

Somehow even though stores are recommended with caution by their own creators, they still where adopted at scale in the React world.

Why could that be ? Let's try to answer that.

When is it recommended to use Flux, or Redux ?

If we dig deeper in the docs of the React How-To, we get to a couple of indications of when we benefit from Flux:

> *React components are arranged in a hierarchy. Most of the time, your data model also follows a hierarchy. In these situations Flux doesn't buy you much. Sometimes, however, your data model is not hierarchical. When your React components start to receive props that feel extraneous, or you have a small number of components starting to get very complex, then you might want to look into Flux.*

Also if we dig into the issues we get these recommendations as well. A store-like architecture is recommended if:

> *You have a piece of data that needs to be used in multiple places in your app, and passing it via props makes your components break the single-responsibility principle (i.e. makes their interface make less sense)*

But also there is this scenario:

> *There are multiple independent actors (generally, the server and the end-user) that may mutate that data*

So there are a couple of situations where its suggested to use a store solution together with React. So let's see how does this fit into Angular.

Stores and applications with concurrent updates

If we base ourselves only on the last part, only a small number of apps namely applications with server push requirements would benefit from Flux. Because that's usually when we have multiple actors updating the same data, and that is the case of the original Facebook counter issue that originated Flux.

Note that we don't need to have server push to fall into this situation, long-polling with `setInterval` or modifying the data inside `setTimeout` would lead us to the same scenario: multiple actors editing concurrently the same data.

We can safely say that many applications don't have this problem, right ? It's an important problem that we need to design towards if present, but do most applications have it ? Probably not, only a certain class of apps.

But then why is Redux so universally adopted in the React world ? That leaves the other reason provided.

What is the most frequent problem that Redux solves ?

Redux also solves the "extraneous props" issue. And that has got to be the one of main reasons why Redux is so popular in the React world.

What could "props feels extraneous" mean in Angular terms? Props are the equivalent of the `@Input()` member variables of an Angular component.

So this means that Redux helps us cope with situations where we are passing inputs to components up the component tree using `@Input()`, but those inputs feel extraneous, as not part of the application at that point.

For example, we are passing something 5 or 10 levels up the component tree. The leaves of the tree know what to do with it, but for all the components in the middle the input feels extraneous and makes that component less reusable and more tied to the application. But that is just one example.

Extraneous props, what else can it mean?

The extraneous props issue seems to be a component inter-communication issue.

There are situations where components are dependent on each other at completely different points in the component tree, and passing inputs 10 levels up the tree and callback functions 10 levels down the tree then 5 levels up another branch is not scalable in complexity.

These are other cases when this happens:

- pass data deep down the tree, and react to events several levels up the component tree
- Another issue is, we have sibling components in the tree that are interdependent, and that represent different view for the

> same data on the screen, like a list of folders with unread messages, and a total unread messages counter on the page header.

There are many more examples. If we only had props or `@Input()` as a component communication mechanism we would run into trouble very quickly. Passing only inputs to components won't scale in complexity.

These scenarios are actually very common, so there is our answer.

Why is Redux so popular in the React world ?

Probably because it also solves the extraneous props issue: which means it provides a solution for more complex component interaction scenarios.

This is a fundamental problem without which we cannot build larger applications, and Redux solves it.

Almost all non-trivial applications have these scenarios, it really does not take a large application, most typical enterprise applications will have some sort of complex component intercommunication scenario.

Why does Redux work well in those cases ?

If we try to solve those scenarios with event buses like AngularJs `$scope.broadcast()`, we will easily end up with event soup scenarios, where the events chain themselves in unexpected ways, and it becomes hard to reason about the application.

This is because an event can very easily be turned instead into a command, causing the emitter to know about the internals of the receiver. Plus there is the possibility of chaining events together accidentally.

Redux looks like an event bus, but its not. Actually a Redux store is a combination of the Command and the Observable patterns. What we do with the store is, we send it a command object known as an action:

```
store.dispatch({
    type: 'REFRESH_MESSAGES'
});
```

We dispatch an action into the store, and the store will operate on the data inside the store. But the emitter of the action does not know what the store will do with it.

We could also dispatch another action from a completely different part of the application:

```
store.dispatch({
    type: 'MARK_MESSAGE_AS_READ',
    payload: {
        messageId: 103
    }
});
```

The store would process it and update the list of messages. The messages are then sent to any parts of the application that need it. But

the receiving end does not know what triggered the generation of the new data:

- a new message arrived from the backend
- a refresh was requested
- a message is marked as read

So what does this have to do with decoupling and scaling in complexity ?

How stores allow decoupled component interaction

The components consuming the new version of the data (maybe a message list and a counter) do not know about what caused the data to change, much like when we subscribe to an RxJs Observable we don't know what triggered the value emission, we only know that we have a new value.

The consuming components have subscribed themselves to the store, like if they had subscribed to an RxJs Observable. This pattern works well because we would have to go out of our way to turn the emitted data into a command, while with event buses that is extremely easy to do.

What about server push ?

Let's now say the server is also pushing new data constantly, new messages. The data is also pushed via a dispatch action:

```
1  store.dispatch({
2      type: 'DISPLAY_NEW_MESSAGE',
```

```
    payload: {
        messageId: 104,
        userId: 3000,
        status: 'UNREAD',
        text: 'Hello World !'
    }
});

```

In all cases, a new list of messages is received and rendered, either into a list of messages or a counter of unread messages. The result of the rendering will be consistent: we will not have a list of messages which are all read and a counter saying that there are 3 unread messages.

This situation is when a store shines

A store is an ideal solution for this problem of editable data and multiple actors, but let's imagine that the data is not being pushed from the server. In that case, we only have the component interaction and coordination problem, but we don't have the possibility of race conditions.

In that case the problem that we are trying to solve is simply component interaction at multiple disconnected places of the component tree, right ?

We no longer need a solution for editing the same data by multiple concurrent actors. And this leads to an important feature of Redux and stores in general.

Stores are a compound solution for multiple problems, not just one

We can see with this example see that stores are a multi-responsibility solution:

- they solve the problem of component interaction via the Observable pattern
- they provide a client side cache if needed, to avoid doing repeated Ajax requests
- They provide a place to put temporary UI state, as we fill in a large form or want to store search criteria in a search form when navigating between router views
- and they solve the problem of allowing modification of client side transient data by multiple actors

Stores are not a solution for only one of those problems, they solve all of them.

What is the problem with a multi-responsibility solution ?

One potential problem with that is that is that those problems don't always come together: you might want to solve one but not the other. Not every application has the same constraints as Facebook: its the biggest web application of the world with 1.8 *Billion* users.

Let's say that your application is your typical enterprise application with less than 100 users: you have limited use for a client side cache, and likely no server push requirements.You might have server push but the data is mostly read-only.

In that case you probably don't benefit from a store architecture (more on that later).

Also you might need to cover a complex component interaction scenario without needing to store the data in-memory for that. The important part here is that these problems don't always come together: they come together for a very particular class of applications but not others.

Does Redux avoid state-related problems ?

Its important to be aware that it doesn't and neither do other global stores solutions in general, because with Redux we are creating a big global application-level state: the store is an application wide singleton service.

The problem with global application state is not the way that its created, its the fact that it exists. Its very easy to create subtle errors due to the fact that we forgot to clean it up. It really does not change much the fact that we created the state using pure reducer functions only, or if the global state is immutable.

All of that helps, but we have still created global application state, and the main problem is still there: it exists and we need to clean it up at all in all the right places, and that does not scale well in complexity.

But if needed, there is nothing wrong with global state: some user data data needed everywhere, why not load it once and put it in a singleton service?

What is the best way to deal with global state ?

The best way to avoid global application state is to not create it unless its necessary, which many times is not. Modern applications do tend to

need more state than before: like for example where do we keep the last search results for a given search form as we navigate through the app?

We don't want to repeat a search each time when we go back from a detail to the master table, even though we triggered a router navigation.

Can we use temporary local state ?

The ideal situation for these situations would be to be able to create a state that is local only to that interaction with that particular master-detail setup, and to make it so that it cleans itself up automatically after use.

And this what Angular allows us to do as we will see in a moment.

Are there alternative solutions in the Angular world, other than a store ?

In Angular we have a whole set of built-solutions to handle complex component interaction scenarios. The core of all those solutions is the Angular Dependency Injection system:

- We can inject services deep in the component tree if we want to, have a look at Angular 2 Smart Components vs Presentation Components: What's the Difference, When to Use Each and Why?)
- We can even inject components or services into each other if we feel they are inherently tightly coupled
- We can created shared data services that might or might not store the data

But that is just to start. Let's go back the the master detail scenario: we can create a non global service and associate it to a section of the page only, using the hierarchical injector. This means the service and its eventual state would clean itself up transparently.

Creating Local state that cleans itself up

Lets say that we have navigated to a section of the application containing the messages list, and that we click on the list and and we go to the detail of the message.

This is the top-level component of that route:

```
@Component({
  selector: 'messages-container',
  providers: [MessagesService],
  template: `
        Messages Master Detail Container:
        <router-outlet></router-outlet>
  `
})
export class MessagesContainerComponent {

      ...

}

```

Notice the `MessagesService` in the `providers` property. What does this mean ? It means that the service is not an application wide singleton. So if we wanted to keep the search results of the master in memory while we open and close multiple details, the `MessagesService` would be an ideal place to put it instead of a global store. Why ?

Because this instance of `MessagesService` is local to the `MessagesContainerComponent` and its siblings. It can only be injected there and not anywhere else in the application.

You could also create a `MessagesTableService` and inject it at the level of table, use it to load and paginate data and have multiple tables side by side, each with its own instance of `MessagesTableService`.

The great thing about these local services visible only by a subset of the component tree is that they clean themselves up together with the associated component as we navigate away from its route.

The local stateful service could be implemented for example as an Observable Data Service.

Angular 2 and Stores - A Frequent option?

As we can see, in Angular we have a number of inter-component communication mechanisms available to us, not just `@Input()`, also we have a mechanism for creating and disposing automatically of local state.

In Angular we don't necessarily benefit from a store to solve those problems, there are many other built-in solutions.

Many times a store is added to an application to get an observable-like API to allow for certain component interactions: why not simply use an observable ?

Adding a store is an important constraint to the overall architecture of the application, and it implies the creation of a large amount of global

application state. If there are better alternatives built-in that don't imply this, why not consider them instead ?

Using a Store has a Cost

The store does solve the problem of component interaction, but it also creates the need for managing state in your application, which might otherwise not exist when using other solutions.

This might mean that in Angular a store solution would be much less often useful than in React? Actually also in React after an initial period other solutions where seeked as we will see.

There are other arguments usually mentioned to support the choice of a store solution: performance, testability, tooling and the ability to keep the application predictable and simple to reason about. Let's cover these one by one, starting with the last.

Unidirectional Data Flow

Unidirectional data flow is an important property that we hear about both in React and Angular 2: its referred to a property that is looked for in applications, that ensures that they are predictable and easy to reason about.

Unidirectional Data Flow in React

In the original talks of Flux, unidirectional data flow is described as the following: the user triggers an action, its gets dispatched to the stores which generate a new model and send it to the view.

But the view cannot itself dispatch further actions while rendering, nor another action can be dispatched if the dispatch for an action is already

ongoing.

Avoiding this scenario looks like one of the main goals of Flux based on the original presentation, have a look here. Another reference is made here.
Also take a look at the original Flux dispatcher code here where the check is made mentioned in the talks.

UI predictability in React and Flux seems to be aimed to be achieved mostly by putting a beneficial constraint on the data layer: prevent chained dispatches.

Redux and Unidirectional Data Flow

Its important to be aware that Redux does not protect against the chained dispatch scenario mentioned in the original talks of Flux. With Redux we can trigger another dispatch from the subscribe method while in original Flux if an action was already being dispatched we could not trigger another.

So the need for enforcing an unidirectional data flow does not seem to be one of the main reasons why Redux is so widely adopted, because by design and at least according to the definition provided in the original Flux talks, it does not prevent the chained dispatch problem.

Maybe because its too constraining and in practice it does not happen a lot ?

Unidirectional Data Flow in Angular

In Angular we also see unidirectional data flow mentioned as a property that allows us to reason about the application in a predicable way.

But there the concern seems a bit different although related: its not about putting a constraint on the data layer, the data layer can have any form.

Unidirectional data flow in Angular is described as making sure that the view cannot update itself. What does that mean ?

Unidirectional Data flow and Rendering in Development Mode

When the rendering starts, we go through the component tree in one sweep and a component cannot during rendering give different results on a second pass or modify a parent component.

Basically the act of evaluating the expressions in the templates or triggering certain component lifecycle methods cannot itself trigger further changes in the view, creating a situation similar to the multi-step digest cycle of AngularJs, which sometimes lead to unpredictable results.

Breaking Angular Unidirectional Data Flow

Imagine you are printing a random number to the screen: if you try to calculate it via a component getter method and pass it to a template expression, we will break the application in development mode, because you don't get the same result in the second top to bottom sweep:

```
@Component({
  selector: 'messages-container',
  template: `

       {{randomNumber}}
  `
})
```

```
export class MessagesContainerComponent {

      ...


     get randomNumber() {
          return Math.random();
     }

}

```

Try it, you should get:

```
Expression has changed after it was checked
```

So it looks like to ensure a predicable rendering behavior in the UI and preventing the view from updating itself, we don't necessarily need to adopt a store-like architecture.

Let's go over another common reason presented to use a store: improved performance and next let's go over testability and tooling.

Stores and Performance

Sometimes stores are mentioned as a way to make an application more performant, because we can make the state immutable using something like ImmutableJs or Deep Freeze, and we can then use `OnPush` change detection everywhere.

The Angular 2 Change Detection mechanism is out of the box blazing fast and behaves very intuitively. By default only what we use in the

template as expressions is used to detect changes, all the rest is ignored (have a look at this post).

`OnPush` is really an optimization that only a few applications will likely benefit from, like applications that load a lot of data (and how much data can we load that will still be useful for the user), or applications that run in very constrained devices.

Its safe to say that most applications don't fall under those categories (given current smartphones). But if we still need `OnPush`, we can simply use it without a store, especially if our data is mostly read-only.

If the application is a real time dashboard of some sort like a chart dashboard, its probably better to either throttle the data or some other solution. We can even detach a branch of the UI from change detection and throttle its rendering.

The main point here is that adding a store does mean that we will make an application more performant or easier to optimize, because we can optimize the change detection system in a completely independent way from the store - the two things can be used together but are not inherently linked.

Another common point for adoption of a store architecture is testability, let's look into that, last point before getting to the demo of the tooling.

Stores and Testability

One of the main benefits often presented to introduce a store is that it will improve the testability of the application.

Its true that reducer functions are easy to test, but the application itself is not made more testable by introducing a store, anymore than its testable because we inject dependencies via the dependency injection system instead of creating them directly inside components.

Let's say that an application does not have a lot of data modification or concurrent modifications of data by the server and the user: that application probably does not need a store, and introducing it would not make it more testable.

But last and certainly not least, we get to a huge benefit - the tooling.

Stores and Tooling

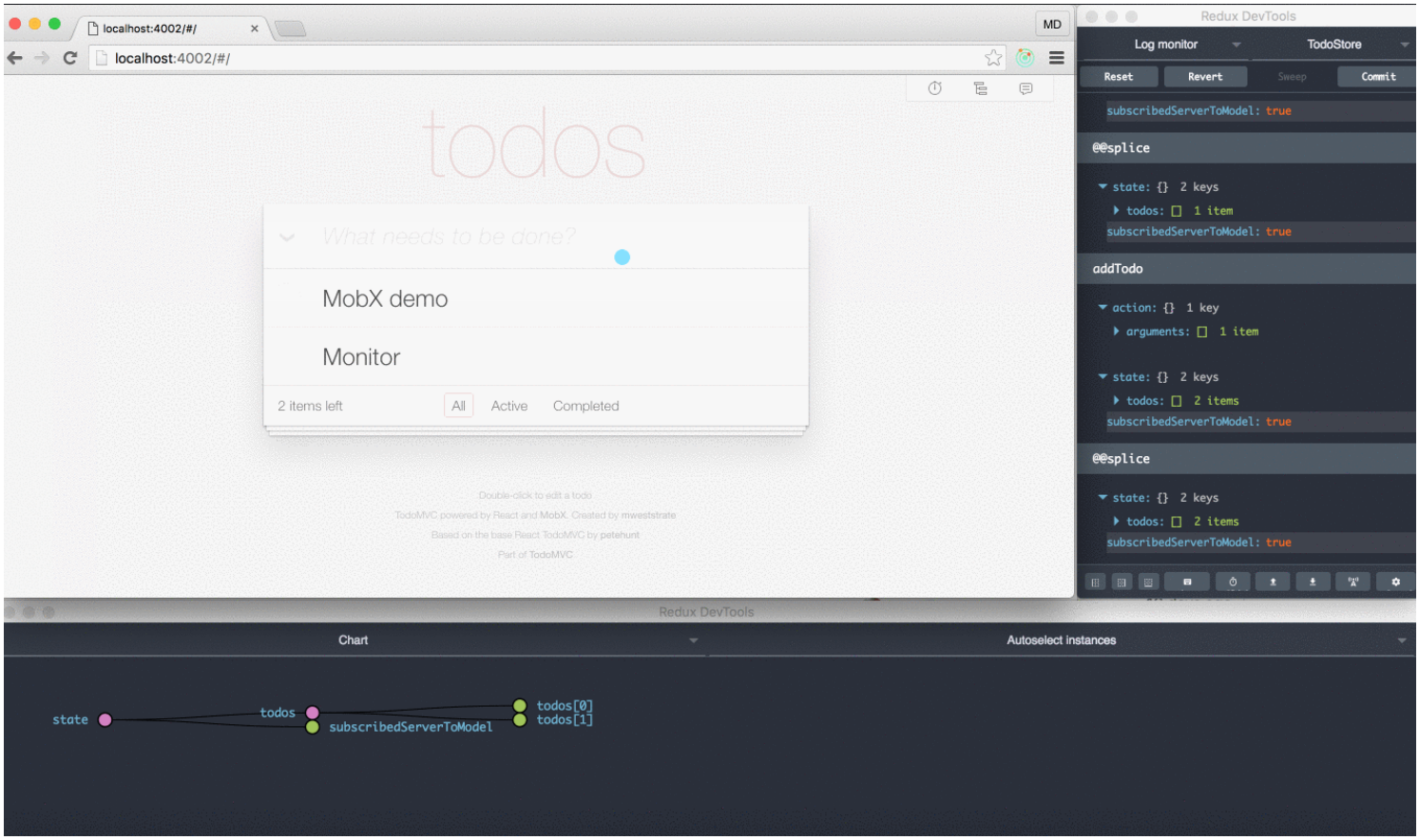
One of the biggest reasons for using a store its the tooling ecosystem it provides. The tooling is amazing, time traveling debugging, being able to attach a store state to a bug report and hot reloading those are huge features.

These tools are amazing, but it looks like these days Redux is not a must use in new React applications, so how does that work in terms of tooling ? ### A frequent alternative to Redux After a period of initial adoption of mostly Redux, many React applications are being built using MobX, which is a variation of the Observable pattern. We can see this description in the docs: > MobX adds observable capabilities to existing data structures like objects, arrays and class instances. This can simply be done by annotating your class properties with the @observable decorator (ES.Next). And here is a small code sample of what it looks like:

```
class Todo {
    id = Math.random();
    @observable title = "";
    @observable finished = false;
}

```

If you saw the video above on the NgRx Dev Tools, does this look familiar ? Take a look, there are some also [developer tools] (https://github.com/zalmoxisus/mobx-remotedev) like the redux dev tools for Mobx:



Actually the Mobx dev tools use also the same browser plugin. Based on this example, it seems that to have this advanced type of tooling we don't necessarily need to adopt a store architecture. All we need to do is write our application using an Observable library with a good or evolving tooling ecosystem. #### Tools in other related ecosystems Another ecosystem related to the notions of streams and of the observable pattern and its variations its the CycleJs ecosystem. Here is a view on Flux, Redux and a demo of some Developer Tools in the CycleJs ecosystem. Just before, we have a great discussion about adding tooling while not bringing a pre-defined architecture with it (by [@andrestaltz](https://twitter.com/andrestaltz)).

Then the tooling is demonstrated at the end, note that many of this advanced tooling is in general a work in progress across ecosystems.

The most important thing to keep in mind here is that it looks like there are other ways of obtaining great tooling without adopting a store architecture.

Conclusions

It could well be that Store architectures initially became popular in the React world because they solved a couple of fundamental problems that React as being just the View did not provide (by design) a solution for out of the box:

- provide an observable-like pattern for decoupled component interaction
- provide a client container for temporary UI state
- provide a cache for avoiding excessive HTTP requests
- provide a solution for concurrent data modification by multiple actors
- provide a hook for tooling

Then half a year to one year later the ecosystem evolved to adopt stores in only certain application types and not others. The same thing might be happening now in the Angular world, and the outcome could be the same. Let's keep an eye on the tooling, one of the main features of RxJs 5 will be improved debugability.

Suggestions

So what does this all mean, if you are choosing an architecture for your application, what to do then ? The original quote of the React How-To still seems like good advise:

> *You'll know when you need Flux. If you aren't sure if you need it, you don't need it*

Here is a suggestion: unless you have a concurrent data modification scenario, consider starting to build your application with some plain RxJs services, leveraging local services and the dependency injection system.

Then if the need arises, we can always refactor part of the application into a store if a use case comes up.

On the other hand if we have a concurrent data modification scenario in a part of our application, we might as well use a store from the beginning, because that is a great solution for that situation.