Data Architecture in Angular 4

An Overview of Data Architecture

Managing data can be one of the trickiest aspects of writing a maintainable app. There are tons of ways to get data into your application:

- AJAX HTTP Requests
- Websockets
- Indexdb
- LocalStorage
- Service Workers
- etc.

The problem of data architecture addresses questions like:

- How can we aggregate all of these different sources into a coherent system?
- How can we avoid bugs caused by unintended side-effects?
- How can we structure the code sensibly so that it's easier to maintain and on-board new team members?
- How can we make the app run as fast as possible when data changes?

For many years MVC was a standard pattern for architecting data in applications: the Models contained the domain logic, the View displayed the data, and the Controller tied it all together. The problem is, we've learned that MVC doesn't translate directly into client-side web applications very well.

There has been a renaissance in the area of data architectures and many new ideas are being explored. For instance:

- **MVW / Two-way data binding**: *Model-View-Whatever* is a term used[72] to describe Angular 1's default architecture. The `$scope` provides a two-way data-binding - the whole application shares the same data structures and a change in one area propagates to the rest of the app.
- **Flux**[73]: uses a unidirectional data flow. In Flux, Stores hold data, Views render what's in the Store, and Actions change the data in the Store. There is a bit more ceremony to setup Flux, but the idea is that because data only flows in one direction, it's easier to reason about.
- **Observables**: Observables give us streams of data. We subscribe to the streams and then perform operations to react to changes. RxJs[74] is the most popular reactive streams library for

[72] See: Model View Whatever

[73] https://facebook.github.io/flux/

[74] https://github.com/Reactive-Extensions/RxJS

JavaScript and it gives us powerful operators for composing operations on streams of data.

There are a lot of variations on these ideas. For instance:

- Flux is a pattern, and not an implementation. There are **many** different implementations of Flux (just like there are many implementations of MVC)
- Immutability is a common variant on all of the above data architectures.
- Falcor[75] is a powerful framework that helps bind your client-side models to the server-side data. Falcor often used with an Observables-type data architecture.

Data Architecture in Angular 4

Angular 4 is extremely flexible in what it allows for data architecture. A data strategy that works for one project doesn't necessarily work for another. So Angular doesn't prescribe a particular stack, but instead tries to make it easy to use whatever architecture we choose (while still retaining fast performance).

The benefit of this is that you have flexibility to fit Angular into almost any situation. The downside is that you have to make your own decisions about what's right for your project.

Don't worry, we're not going to leave you to make this decision on your own! In the chapters that follow, we're going to cover how to build applications using some of these patterns.

[75] http://netflix.github.io/falcor/