Making your web application easier to reason about with unidirectional data flow.

October 8th, 2017 – Kenneth Yeung

@kennethyeung815

kenneth.yeung@ncino.com

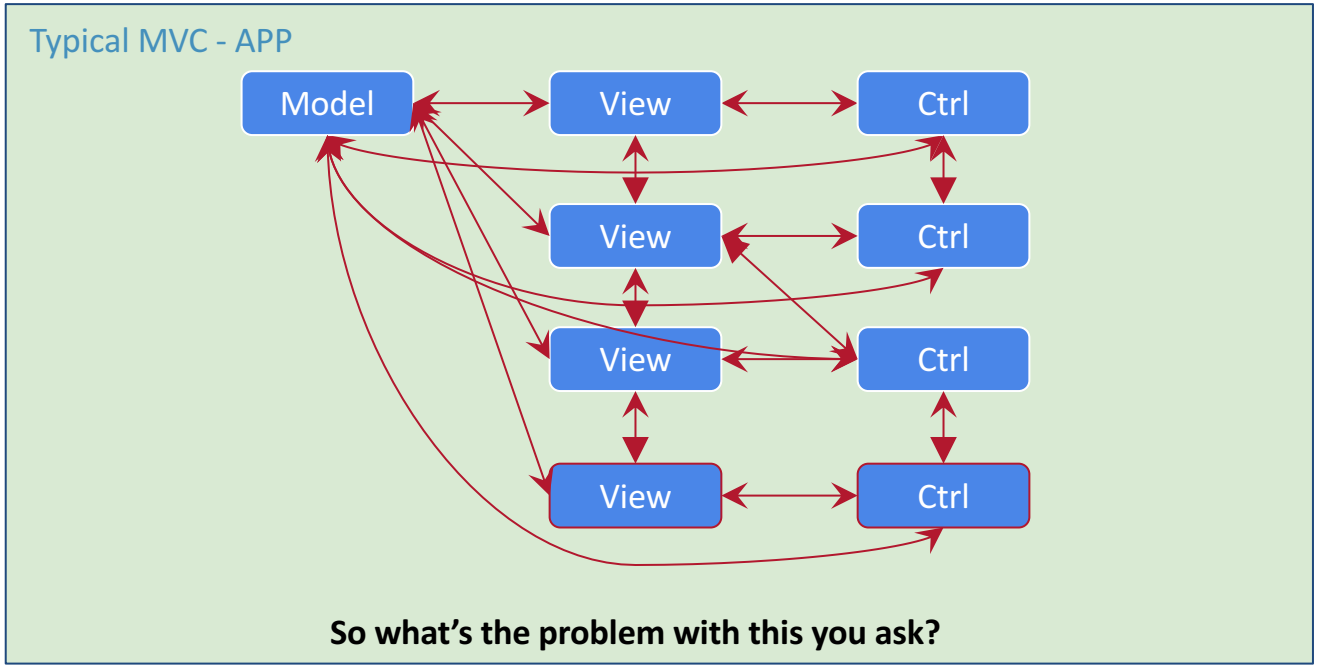
Hustle in Comfort

Agenda

- The need for unidirectional data flow?
- What is redux?
- When & why should you use it?
- NgRx in parts
- Demo powerful debugging tools

Model View Controller

Let’s take a look at traditional MVC.

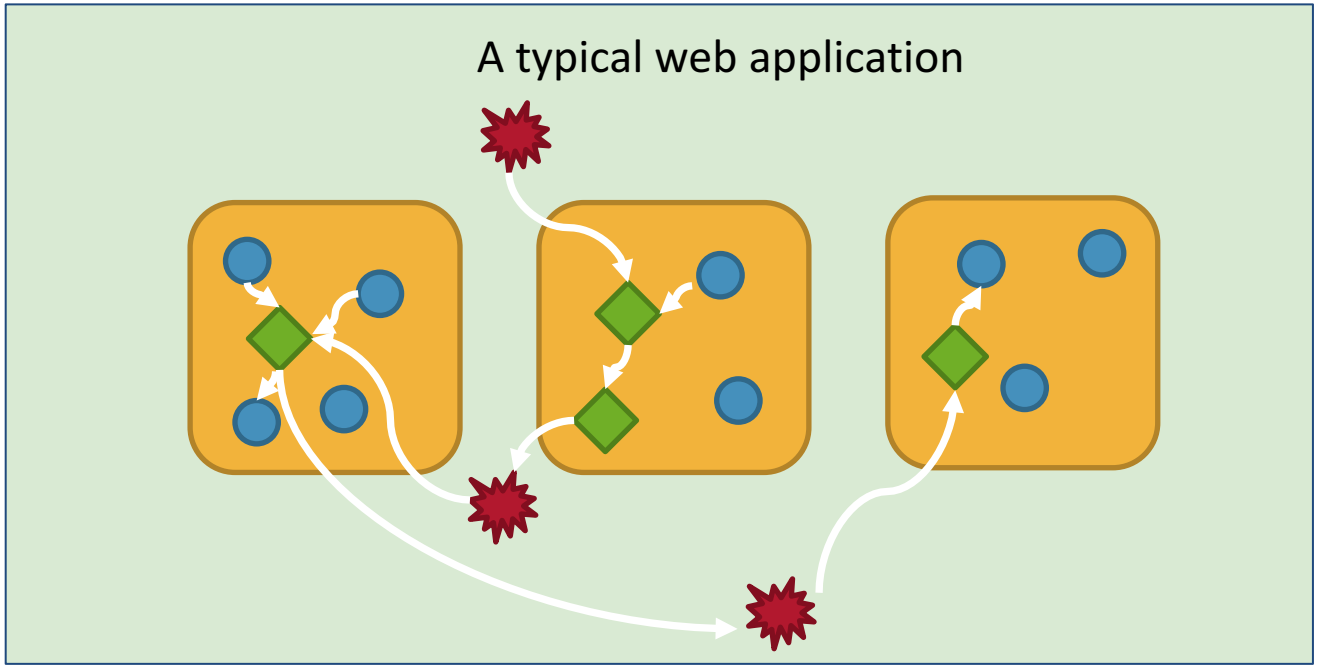


Model View Controller

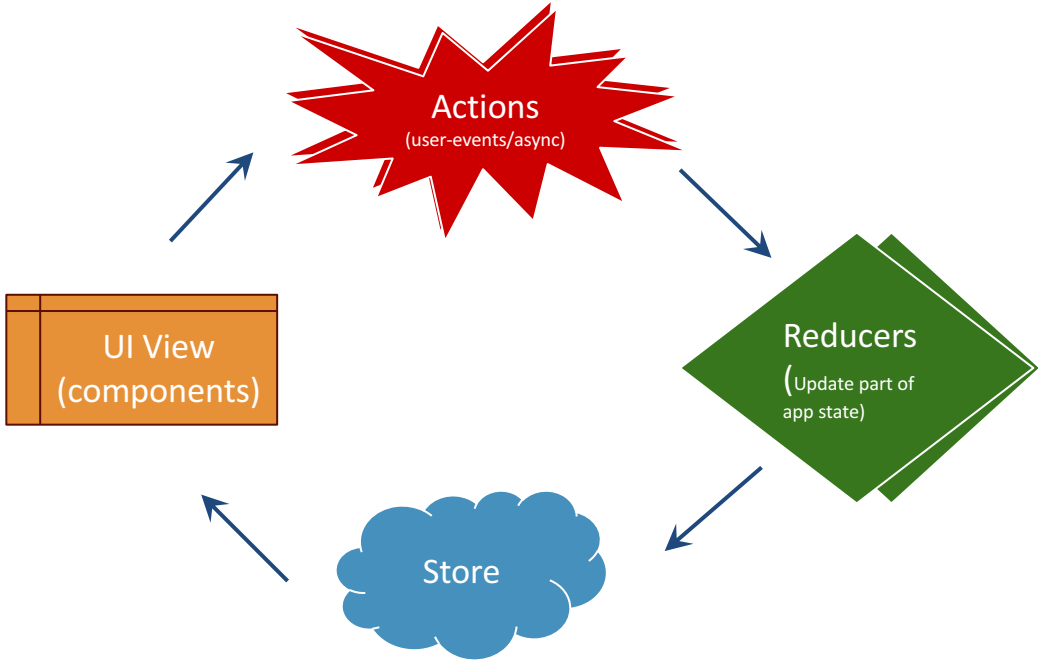
- **Problem?**
  - Difficult to understand
  - Difficult to manage state
  - Difficult to debug
  - Requires multiple controllers and directives to manage and represent state changes over time
  - Cascading effect – One component can triggers updates to multiple different components

What we do as developers?

- We ~~create~~ manage complexity!



Unidirectional Data Flow (Redux)



Unidirectional Data flow

- What is unidirectional data flow
  - Pattern for handling state in your application.
- Why invest the time to learn yet another pattern
  - Benefits
    - Easier to reason about
      - Shorten development time
      - Promotes standardization and increase in code quality
    - Easier to test
    - Easier to debug
    - Saves marriages.

History of unidirectional data flow

- 2014 – Flux
- 2015 – Redux
- 2016 – ngRx

Unidirectional Data flow

- When to use ngRx/Redux?
  - Medium to Large SPA
  - In applications that have complex data flows

Unidirectional Data flow

- Redux - 3 principles
  - The whole state of your app is stored in an object tree inside a single *store*.
  - The only way to change the state tree is to dispatch an *action*, an object describing what happened.
  - To specify how the actions transform the state tree, you write pure *reducers*.

Parts of NgRx

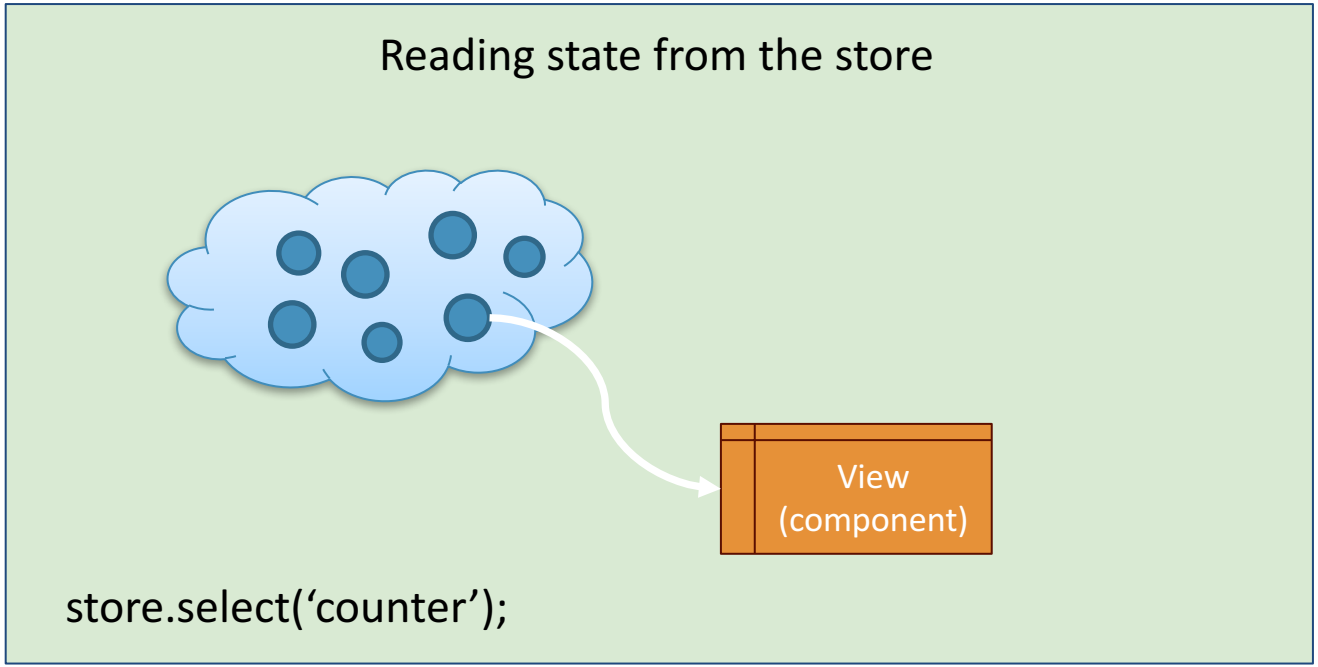
- 3 main parts of NgRx
  - store
  - actions
  - reducers

Parts of NgRx Store

- **Store**
  - Consolidate all our state into a single object called a store.
  - Similar to a traditional database, the store represents the point of data storage for an application
  - Your store can be thought of as a client side ‘single source of truth’, (Client side database)
  - Your store at any point could supply a complete representation of your application.

Parts of NgRx

- How do you get state from the store?



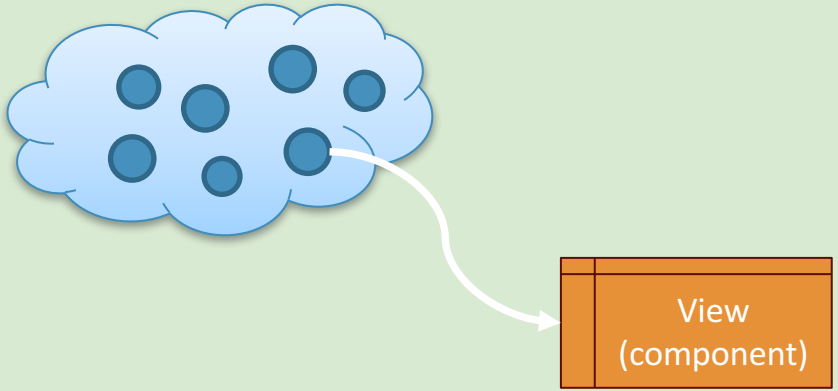
Parts of NgRx

- **Observables**
  - Stream of zero, one or multiple values **over any amount of time.**
  - They are Lazy. Observables will not generate values via an underlying producer until they are subscribed to
  - Can be Unsubscribed from. Underlying producer can be told to stop and torn down.
  - Observables will become a ECMAScript standard

Parts of NgRx

- How do you get state from the store?

Reading state from the store




```
counter$ = store.select('counter');
```

Parts of NgRx Actions

- **Actions**
  - Are simple JavaScript objects.
  - They describe what happened.
  - All events and user interaction that would cause a state update must be expressed in the form of an action
  - Describes something has (or should) happen, but they **don't specify how** it should be done.

Parts of NgRx

Actions

- Actions

```
// Increment Action
{
    type: 'INCREMENT'
}


// Increment By Action
{
    type: 'INCREMENT_BY',
    payload: 2
}
```

```
export interface Action {
  type: string;
  payload?: any;
}
```

- **Reducers**
  - Reducers are **pure functions**
  - What is a pure function?
    - A function that has a return value that relies only on it's input arguments.
    - Does not modify the values passed to them
    - Can call other pure functions.
    - No side effects, network or database calls.
    - You can treat it as a black box

- Pure functions vs impure functions

```
// pure function
function increment(count) {
    return count + 1;
}

// impure function
let count = 0;
function increment() {
    count = count + 1;
}
```

- Pure functions vs impure functions

```
// pure function
function square(x){
    return x * x;
}

// impure function
function square(x){
    updateInDatabase(x);
    return x * x;
}
```

- Testing impure vs pure functions

```
// testing impure
count = 0
increment();
expect(count).toBe(1);

// testing pure
expect(increment(0)).toBe(1);
```

- Pure functions are easy to test!

```
// testing impure
// Spy on and call fake updateInDatabase()
spyOn(myApp,"updateInDatabase").and.callFake(function() {
    return "Database updated!";
});
expect(square(3)).toBe(9);

// testing pure
expect(square(3)).toBe(9);
```

Parts of ngRx

Reducers

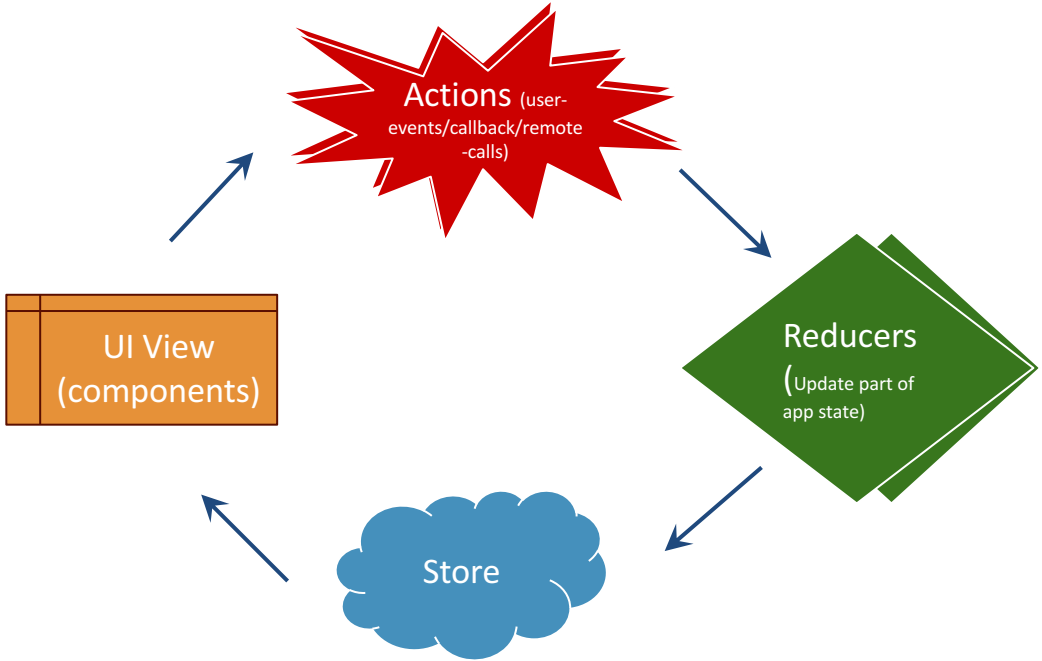
- **Reducers**
  - Accepts two arguments
    - the **previous state**
    - **action** with a type and optional data (payload) associated with the event.
  - Returns a new state

```
export interface Reducer<T> {
  (state: T, action: Action): T;
}
```

- Reducers

```
export function counterReducer(state: number = 0, action: Action) {
    switch (action.type) {
        case INCREMENT:
            return state + 1;
        case INCREMENT_BY:
            return state + action.payload;
        case DECREMENT:
            return state - 1;
    }
}
```

Parts of NgRx (Recap)



RxJS

- Reactive Extensions for Javascript
  - **RxJS** is a library that allows us to easily create and manipulate streams of events and data. This makes developing complex asynchronous code much easier.
- @ngrx – Redux implementation supercharged with RxJS
  - Lightweight, interoperable reactive services and components for Angular

Debugging Tools

-

DEMO TIME!!!

Resources

- This demo's github repo
  - https://github.com/kennethyeung815/ngrxDemo
- Chrome Redux DevTools
  - https://tinyurl.com/redux-devtools
- NgRx – exampleApp
  - https://github.com/ngrx/platform/tree/master/example-app
- RxJS 5 Thinking Reactively | Ben Lesh
  - http://reactivex.io/rxjs/
  - https://youtu.be/3LKMwkuK0ZE
- Reactive Angular2 with ngRx | Rob Womald
  - https://youtu.be/mhA7zZ23Odw
- Getting Started with Redux | Dan Abramov
  - http://redux.js.org/
  - https://egghead.io/courses/getting-started-with-redux