ngrx store

Friday, March 20, 2020 1:57 PM

1. Overview

Introduction

With the advent of Angular 2, new patterns, best practices, and libraries empowered by new framework features and functionality are emerging. One such library, championed by Angular developer advocate Rob Wormald (huge props to @MikeRyan52 as well), is @ngrx/store. @ngrx/store builds on the concepts made popular by Redux, a popular state management container in the React community, and supercharges it with the backing of RxJS. The result is a tool and philosophy that will revolutionize your applications and development experience.

Core Concepts

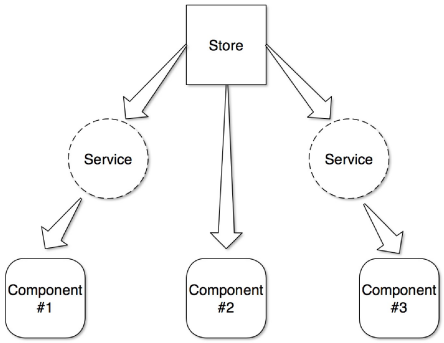
Before we dive into the nuts and bolts of what makes @ngrx/store work, let's first take a high-level look at the core concepts. Each application built around @ngrx/store will contain three main pieces, reducers, actions, and a single application store. Let's take a moment to explore each of these concepts.

Store

Like a traditional database represents the point of record for an application, your store can be thought of as a client side *'single source of truth'*, or database. By adhering to the [1] *store contract* when designing your application, a snapshot of store at any point will supply a complete representation of relevant application state. This becomes extremely powerful when it comes to reasoning about user interaction, debugging, and in the context of Angular 2+, performance.

1 Single, immutable state tree updated only through explicitly defined and dispatched actions

Centralized, Immutable State



Reducers

The second integral part of a @ngrx/store application is a **reducer**. A [2] reducer is a [3] pure function, accepting two arguments, the previous state and an **action** with a type and optional data (payload) associated with the event. Using the previous analogy, if the store is to be thought of as your client side database, reducers can be considered the tables in said database. Reducers represent sections, or slices of state within your application and should be structured and composed accordingly.

2 Reducer Interface

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

3 A function whose return value is determined only by its input values, with no observable side-effects.

Sample Reducer

```
export const counter: Reducer<number> = (state: number = 0, action: Action) => {
    switch(action.type){
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
};
```

Actions

@ngrx/store encompasses our application state and reducers output sections of that state, but how do we communicate to our reducers when state needs to be updated? That is the role of [4] **actions**. Within an @ngrx/store application, all user interaction that would cause a state update must be expressed in the form of actions. All relevant user events are dispatched as actions, flowing through the [5] action pipeline defined by @ngrx/store, before a new representation of state is output. This process occurs each time an action is dispatched, leaving a complete, serializable representation of application state changes over time.

4 Action Interface

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

5 Dispatched Action Pipeline

Sample Actions

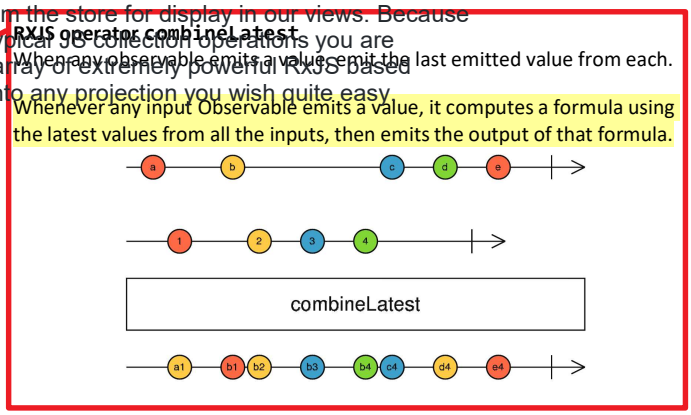
```
// simple action without a payload
dispatch({type: 'DECREMENT'});
// action with an associated payload
dispatch({type: ADD_TODO, payload: {id: 1, message: 'Learn ngrx/store', completed: true}})
```

Projecting Data

Finally, we need to extract, combine, and project data from the store for display in our views. Because the store itself is an observable, we have access to the typical JS collection operations you are accustom to (map, filter, reduce, etc.) along with a wide-array of extremely powerful RxJS based observable operators. This makes slicing up store data into any projection you wish quite easy.

State Projection

```
// most basic example, get people from state
store.select('people')

// combine multiple state slices
Observable.combineLatest(
  store.select('people'),
  store.select('events'),
  (people, events) => {
    //projection here
})
```



Angular 2 Approach

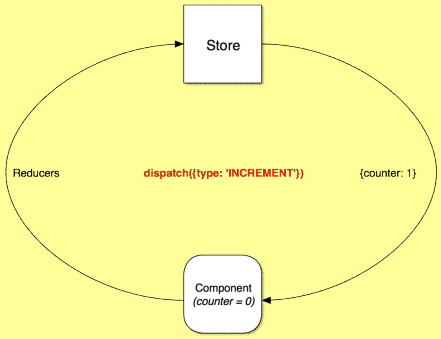
In the previous section I mentioned *abiding by the store contract* when developing your application. What exactly does this mean in the context of the typical Angular setup and workflow which you have grown accustomed? Let's take a look.

@ngrx/store promotes the idea of [7] one-way data flow (or cycle) and explicitly dispatched actions. All state updates are handled above your components in store, delegated to reducers. The only way to

initiate a state update in your application is through dispatched actions, corresponding to a particular reducer case. This not only makes reasoning about state changes in your application easier, as updates are centralized, it leaves a clear audit trail in case of error.

6 Two-Way Data Binding - REMOVED

7 One Way Data-Flow



Non-Store Counter Example
(demo)

```
@Component({
    selector: 'counter',
    template: `
    <div class="content">
        <button (click)="increment()">+</button>
        <button (click)="decrement()">-</button>
        <h3>{{counter}}</h3>
    </div>
    `
})
export class Counter{
    counter = 0;

    increment(){
        this.counter += 1;
    }

    decrement(){
        this.counter -= 1;
    }
}
```

Store Counter Example
(demo)

```
@Component({
    selector: 'counter',
    template: `
    <div class="content">
        <button (click)="increment()">+</button>
        <button (click)="decrement()">-</button>
        <h3>{{counter$ | async}}</h3>
    </div>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class Counter{
    counter$: Observable<number>;

    constructor(private store : Store<number>){
        this.counter$ = this.store.select('counter')
    }

    increment(){
        this.store.dispatch({type: 'INCREMENT'});
    }

    decrement(){
        this.store.dispatch({type: 'DECREMENT'});
    }
}
```

Angular can detect when component data changes, and then automatically re-render the view to reflect that change.

Angular, at startup time, will patch several low-level browser APIs, such as for example addEventListener, which is the browser function used to register all browser events, including click handlers.

This low-level patching of browser APIs is done by a library shipped with Angular called Zone.js.

From <https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/>

See Also <https://alligator.io/angular/change-detection-strategy/>

See Also <https://blog.angular-university.io/onpush-change-detection-how-it-works/>

See Also <https://netbasal.com/a-comprehensive-guide-to-angular-onpush-change-detection-strategy-5bac493074a4>

Advantages of Store

Why take the time to invest in this particular library, architecture pattern, and learning curve? The

primary advantages to a @ngrx/store-based application are centralized state, performance, testability, and tooling.

Centralized, Immutable State

All relevant application state exists in one location. This makes it easier to track down problems, as a snapshot of state at the time of an error can provide important insight and make it easy to recreate issues. This also makes notoriously hard problems such as undo/redo trivial in the context of a Store application and enables powerful tooling.

Performance

Since state is centralized at the top of your application, data updates can flow down through your components relying on slices of store. Angular 2+ is built to optimize on such a data-flow arrangement, and can disable change detection in cases where components rely on Observables which have not emitted new values. In an optimal store solution this will be the vast majority of your components.

Testability

All state updates are handled in reducers, which are pure functions. Pure functions are extremely simple to test, as it is simply input in, assert against output. This enables the testing of the most crucial aspects of your application without mocks, spies, or other tricks that can make testing both complex and error prone.

Tooling and Ecosystem

A centralized, immutable state also enables powerful tooling. One such example is ngrx developer tools, which provides a history of actions and state changes, allowing for [8] *time travel* during development. The patterns provided by Store also allow for a rich ecosystem of easy to implement middleware. Because store provides an entry point both before and after dispatched actions hit application reducers, problems such as syncing slices of state to local storage, advanced logging, and implementing sagas are easily solved with a quick package include and a few lines of code. This ecosystem will only grow over the coming months.

8 Manipulating the history of dispatched actions and state changes to emulate a point in time of application interaction.

2. Building blocks of @ngrx/store

Before we build a @ngrx/store application, let's first take a look at the RxJS concepts on which @ngrx/store is built. By understanding these concepts first, we can more effectively utilize the library in the future. For a more detailed explanation of each of the topics below, please check out the following additional resources.

- Step-by-Step Async JavaScript with RxJS - John Lindquist
- RxJS Beyond The Basics - Creating Observables From Scratch - André Staltz
- Rx Lessons from Ben Lesh - Ben Lesh
- Introduction to Reactive Programming - André Staltz
- Getting Started With Redux - Dan Abramov
- Asynchronous Programming - The End of the Loop - Jafar Husain
- Using the Async Pipe in Angular 2 - Brian Troncone

Disclaimer: The actual @ngrx/store code by Mike Ryan and Rob Wormald is significantly more robust. These examples are meant to demonstrate the RxJS concepts involved and remove the 'magic' from the library.

Exploration of Subject / Dispatcher

The messenger of Rx, you tell me, I'll tell them...

(demo)

The two pillars of @ngrx/store, the store and dispatcher, both extend RxJS Subjects. Subjects are both Observables and Observers, meaning you can subscribe to a Subject, but can also subscribe a Subject to a source. At a high-level Subjects can be thought of as messengers, or proxies.

Because Subjects are Observers, you can *'next'*, or pass values into the stream directly. Subscribers of that Subject will then be notified of emitted values. In the context of @ngrx/store, these subscribers could be an Angular 2 service, component, or anything requiring access to application state.

Subscribing to a Subject

```
//create a subject
const mySubject = new Rx.Subject();

//add subscribers
const subscriberOne = mySubject.subscribe(val => {
  console.log('***SUBSCRIBER ONE***', val);
});

const subscriberTwo = mySubject.subscribe(val => {
  console.log('***SUBSCRIBER TWO***', val);
});

//publish values to observers of subject
mySubject.next('FIRST VALUE!'); '***SUBSCRIBER ONE*** FIRST VALUE! ***SUBSCRIBER TWO*** FIRST VALUE!'
mySubject.next('SECOND VALUE!'); '***SUBSCRIBER ONE*** SECOND VALUE! ***SUBSCRIBER TWO*** SECOND VALUE!'
```

In @ngrx/store (and Redux), it is convention to dispatch actions to the application store. To maintain this API, the Dispatcher extends Subject, adding a dispatch method as a passthrough to the classic next method. This is used to pass values into the Subject before emitting these values to subscribers.

Extending Subject as Dispatcher

```
/*
redux/ngrx-store has a concept of a dispatcher, or method to send actions to application store
lets extend Rx.Subject with our Dispatcher class to maintain familiar terms.
*/
//inherit from subject
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}

//create a dispatcher (just a Subject with wrapped next method)
const dispatcher = new Dispatcher();

//add subscribers
const subscriberOne = dispatcher.subscribe(val => {
  console.log('***SUBSCRIBER ONE***', val);
});

const subscriberTwo = dispatcher.subscribe(val => {
  console.log('***SUBSCRIBER TWO***', val);
});

//publish values to observers of dispatcher
dispatcher.dispatch('FIRST DISPATCHED VALUE!');
dispatcher.dispatch('SECOND DISPATCHED VALUE!');
```

Exploration of BehaviorSubject / Store

Similar to Subject but, what's the last thing you said?...

(demo)

While vanilla Subjects work perfectly as a 'dispatcher', they have one issue that prevents them from being a good fit for store. When subscribing to a Subject, *only values emitted after the subscription are received*. This is unacceptable in an environment where components will be consistently added and removed, requiring the latest, on-demand sections of state from the application store at the time of subscription.

Subjects Only Receive Values Emitted After Subscription

```
/*
Now that we have a dispatcher, let's create our store to receieve dispatched actions.
*/
class FirstStore extends Rx.Subject{}

const myFirstStore = new FirstStore();

//add a few subscribers
const subscriberOne = myFirstStore.subscribe(val => {
  console.log('***SUBSCRIBER ONE***', val);
```

```
});
const subscriberTwo = myFirstStore.subscribe(val => {
  console.log('***SUBSCRIBER TWO***', val);
});

//For now, lets surpass dispatcher and manually publish values to store
myFirstStore.next('FIRST VALUE!');

/*
Let's add another subscriber.
Since our first implementation of store is a subject, subscribers will only have
visibility into values emitted *AFTER* they subscribe. In this case, subscriber
three will have no knowledge of 'FIRST VALUE!'
*/
const subscriberThree = myFirstStore.subscribe(val => {
  console.log('***SUBSCRIBER THREE***', val);
});
```

Luckily, RxJS offers a convenient extension of Subject to handle this problem, the BehaviorSubject. BehaviorSubject's encapsulate all of the functionality of Subject, but also return the last emitted value to subscribers upon subscription. This means components and services will always have access to the latest (or initial) application state and all future updates.

BehaviorSubject Subscriptions Receive Last Emitted Value

```
/*
Because our components will need to query current state, BehaviorSubject is a more
natural fit for Store. BehaviorSubjects have all the functionality of Subject, but
also allow for an initial value to be set, as well as outputting the last value
received to all observers upon subscription.
*/
class Store extends Rx.BehaviorSubject{
  constructor(initialState : any){
    super(initialState);
  }
}

const store = new Store('INITIAL VALUE!');

//add a few subscribers
const storeSubscriberOne = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER ONE***', val);
});
const storeSubscriberTwo = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER TWO***', val);
});

//For demonstration, manually publish values to store
store.next('FIRST STORE VALUE!');

//Add another subscriber after 'FIRST VALUE!' published
//output: ***STORE SUBSCRIBER THREE*** FIRST STORE VALUE!
const subscriberThree = store.subscribe(val => {
  console.log('***STORE SUBSCRIBER THREE***', val);
});
```

Store + Dispatcher Data Flow

Single state tree and one way data flow in Angular 2...

(demo)

In order to function correctly in the context of @ngrx/store, the dispatcher still needs some work. In a @ngrx/store application, all dispatched actions must be passed through a specific pipeline before a new representation of state is passed into store, to be emitted to all observers. You can think of this as a *factory assembly line*, in this case the stations on the line are pre-middleware -> reducers -> post-middleware -> store.

The creation of this pipeline is handled by passing the dispatcher into store when it is created. The store next method is then overridden in order to delegate all actions first to the *dispatcher pipeline* before passing the new representation of state into store. This also allows the store to be *subscribed directly* to action streams, funneling received actions first through the dispatcher.

For now, the implementation of middleware and reducers will be stubbed out with comments.

Associating the Dispatcher with Store

```
/*
All actions should pass through pipeline before newly calculated state is passed to store.
1.) Dispatched Action
2.) Pre-Middleware
3.) Reducers (return new state)
```

```
4.) Post-Middleware
5.) store.next(newState)
*/
class Dispatcher extends Rx.Subject {
  dispatch(value : any) : void {
    this.next(value);
  }
}

class Store extends Rx.BehaviorSubject {
  constructor(private dispatcher, initialState){
    super(initialState);
    /*
    all dispatched actions pass through action pipeline before new state is passed
    to store
    */
    this.dispatcher
       //pre-middleware
       //reducers
       //post-middleware
       .subscribe(state => super.next(state));
  }
  //delegate store.dispatch first through dispatched action pipeline
  dispatch(value){
    this.dispatcher.dispatch(value);
  }
  //override store next to allow direct subscription to action streams by store
  next(value){
    this.dispatcher.dispatch(value);
  }
}

const dispatcher = new Dispatcher();
const store = new Store(dispatcher, 'INITIAL STATE');
const subscriber = store.subscribe(val => console.log(`VALUE FROM STORE: ${val}`));

/*
All dispatched actions first flow through action pipeline, calculating new state which is
then passed to store. To recap, our ideal behavior:
dispatched action -> pre-middleware -> reducers -> post-middleware -> store.next(newState)
*/
//both methods are same behind the scenes
dispatcher.dispatch('DISPATCHED VALUE!');
store.dispatch('ANOTHER DISPATCHED VALUE!');
const actionStream$ = new Rx.Subject();
/*
Overriding store next method allows us to subscribe store directly to action streams, providing same behavior as
manually calling store.dispatch or dispatcher.dispatch
*/
actionStream$.subscribe(store);
actionStream$.next('NEW ACTION!');
```

What's A Reducer?

Like a snowball rolling down hill, reducers accumulate value through iteration...

(demo)

Reducers are the foundation of any @ngrx/store or Redux-based application; they describe sections of state and potential transformations based on dispatched action types. It is the combination of your reducers that makes up a representation of application state at any given time.

Before we discuss how reducers are created and implemented, let's first look at the reduce function. reduce takes an array of values, running a function against an accumulated value and current value, *reducing* your array into a single value upon completion. You can think of the accumulator as a snowball rolling downhill, gaining mass with each revolution. In the same way, the reduce accumulator is the result of applying your defined function to the current value through iteration.

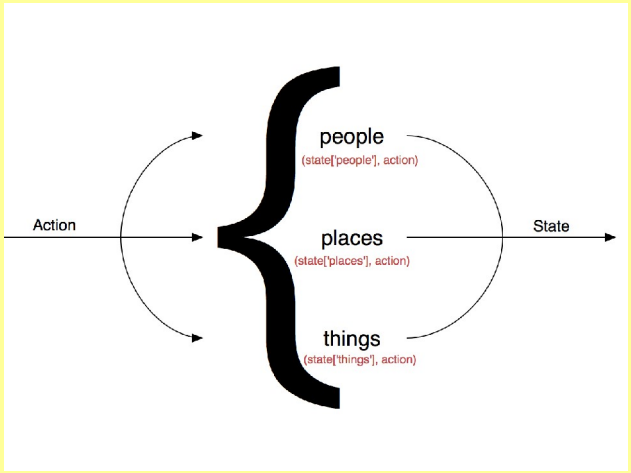
Standard Reduce

```
/*
You can think of reduce as a snowball rolling downhill. Each rotation (or iteration) mass is accumulated until
we reach the bottom. Similarly with reduce, the returned value is passed to the next invocation of the supplied
function until all values in the source array are exhausted. Let's see some examples to solidify this concept.
*/
const numberArray = [1,2,3];
/*
1.) accumulator: 1, current: 2
2.) accumulator: 3, current: 3
Final: 6
*/
const total = numberArray.reduce((accumulator, current) => accumulator + current);
console.log(`***TOTAL***: ${total}`);

//Reduce with objects
const personInfo = [{name: 'Joe'}, {age: 31}, {birthday: '1/1/1985'}];
```

```
/*
1.) accumulator: {name: 'Joe'}, current: {age: 31}
2.) accumulator: {name: 'Joe', age: 31}, current: {birthday: '1/1/1985'}
Final: {name: 'Joe', age: 31, birthday: '1'/1/1985'}
*/
const fullPerson = personInfo.reduce((accumulator, current) => {
 return Object.assign({}, accumulator, current)
});
console.log('***FULL PERSON***:', fullPerson);

//We can also provide an initial value for reduce as a second parameter
const personInfoStart = [{name: 'Joe'}, {age: 31}, {birthday: '1/1/1985'}];
/*
1.) accumulator: {favoriteLanguage: 'JavaScript'}, current: {name: 'Joe'}
2.) accumulator: {favoriteLanguage: 'JavaScript', name: 'Joe'}, current: {age: 31}
3.) accumulator: {favoriteLanguage: 'JavaScript', name: 'Joe', age: 31}, current: {birthday: '1/1/1985'}
Final: {favoriteLanguage: 'JavaScript', name: 'Joe', age: 31, birthday: '1/1/1985'}
*/
const fullPersonStart = personInfo.reduce((accumulator, current) => {
 return Object.assign({}, accumulator, current)
}, {favoriteLanguage: 'JavaScript'});
console.log('***FULL PERSON START:', fullPersonStart);
```

Inspired by Redux, @ngrx/store has the concept of Reducer functions used to manipulate specific slices of state. Reducers accept a state and action parameter, exposing a switch statement (generally, although this could be handled multiple ways) defining action types in which the reducer is concerned. [9] Each time an action is dispatched, each reducer registered to @ngrx/store will be called (through the *root reducer*, created during provideStore at application bootstrap), passed the current state for that state slice (accumulator) and the dispatched action. If the reducer is registered to handle that action type, the appropriate state calculation will be performed and a new representation of state output. If not, the current state for that section will be returned. This is the core of @ngrx/store and Redux state management.

9 Dispatched Action Through Root Reducer



Store / Redux Style Reducer

```
// Redux-Style Reducer
const person = (state = {}, action) => {
  switch(action.type){
    case 'ADD_INFO':
      return Object.assign({}, state, action.payload)
    default:
      return state;
  }
}
const infoAction = {type: 'ADD_INFO', payload: {name: 'Brian', framework: 'Angular'}}
const anotherPersonInfo = person(undefined, infoAction);
console.log('***REDUX STYLE PERSON***: ', anotherPersonInfo);
//Add another reducer
const hoursWorked = (state = 0, action) => {
  switch(action.type){
    case 'ADD_HOUR':
      return state + 1;
    case 'SUBTRACT_HOUR':
      return state - 1;
    default:
      return state;
  }
}
// Combine Reducers Refresher
```

TypeScript - Array reduce()

`reduce()` method applies a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.

Syntax

```
array.reduce(callback[, initialValue]);
```

Parameter Details

- `callback` — Function to execute on each value in the array.
- `initialValue` — Object to use as the first argument to the first call of the callback.

Return Value

Returns the reduced single value of the array.

Example

```
var total = [0, 1, 2, 3].reduce((a, b) => {
      return a + b;
    });
console.log("total is : " + total );
```

```
    default:
      return state;
  }
}
// Combine Reducers Refresher
const myReducers = {person, hoursWorked};
const combineReducers = reducers => (state = {}, action) => {
  return Object.keys(reducers).reduce((nextState, key) => {
    nextState[key] = reducers[key](state[key], action);
    return nextState;
  }, {});
};
/*
This gets us most of the way there, but really want we want is
for the value of firstState and secondState to accumulate
as actions are dispatched over time. Luckily, RxJS offers the
perfect operator for this scenario., to be discussed in next lesson.
*/
const rootReducer = combineReducers(myReducers);
const firstState = rootReducer(undefined, {type: 'ADD_INFO', payload: {name: 'Brian'}});
const secondState = rootReducer({hoursWorked: 10, person: {name: 'Joe'}}, {type: 'ADD_HOURS'});
console.log('***FIRST STATE***:', firstState);
console.log('***SECOND STATE***:', secondState);
```

Example

```
var total = [0, 1, 2, 3].reduce((a, b) => {
      return a + b;
    });
console.log("total is : " + total );
```

Output is as follows:

```
total is : 6
```

From <https://www.tutorialspoint.com/typescript/typescript_array_reduce.htm>

The **Object.keys()** method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

```
const object1 = {a: 'somestring',b: 42,c: false};
console.log(Object.keys(object1)); // expected output: Array ["a", "b", "c"]
```

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys>

Aggregating State with scan

Similar to reduce, but value is accumulated over time...

(scan demo | demo)

scan behaves in a similar fashion to **reduce**, except the accumulator value is maintained over time, or until the observable with scan applied has completed. For instance, as actions are dispatched and new state output, the accumulator in the scan function will always be the last output representation of state. This alleviates the need to maintain a copy of state in store to be passed to our reducers, the scan operator handles this functionality.

Basic scan example

```
const testSubject = new Rx.Subject();
//basic scan example, sum over time starting with zero
const basicScan = testSubject.scan((acc, curr) => acc + curr, 0);
//log accumulated values
const subscribe = basicScan.subscribe(val => console.log('Accumulated total:', val));
//pass values into our test subject, adding to the current sum
testSubject.next(1); //1
testSubject.next(2); //3
testSubject.next(3); //6
const testSubjectTwo = new Rx.Subject();
//scan example building an object over time
const objectScan = testSubjectTwo.scan((acc, curr) => Object.assign({}, acc, curr), {});
//log accumulated values
const subscribe = objectScan.subscribe(val => console.log('Accumulated object:', val));
//pass values into our test subject, adding properties to object
testSubjectTwo.next({name: 'Joe'}); // {name: 'Joe'}
testSubjectTwo.next({age: 30}); // {name: 'Joe', age: 30}
testSubjectTwo.next({favoriteFramework: 'Angular 2'}); // {name: 'Joe', age: 30, favoriteFramework: 'Angular 2'}
```

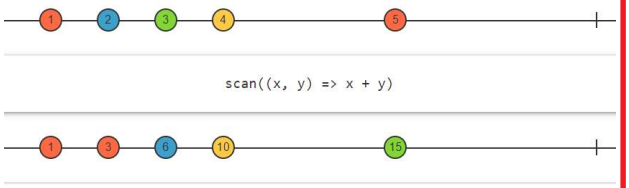
To utilize scan in the application store, it simply needs to be applied as an operator to the dispatcher. All dispatched actions will pass through scan, invoking the combined reducer with current state and action, outputting a new representation of state. The new application state is then next'ed, or pushed to the store, and emitted to all subscribers.

Equipping Store with scan

```
class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       //pre-middleware?
/*
Scan is a reduce over time. In the previous lesson we compared reduce
to a snowball rolling downhill, accumulating mass (or calculated value).
Scan can be thought of similarly, except the hill has no certain end.
The accumulator (in this case, state) will continue to collect until destroyed.
This makes it the ideal operator for managing application state.
*/
       .scan((state, action) => this.reducer(state, action), initialState)
       //post-middleware?
       .subscribe(state => super.next(state));
```

Rx.Observable.prototype.scan([seed], accumulator)

Applies an accumulator function over an observable sequence and returns each intermediate result. The optional seed value is used as the initial accumulator value.



Arguments

1. `[seed]` *(Any)*: The initial accumulator value.
2. `accumulator` *(Function)*: An accumulator function to be invoked on each element.

Return

`(Observable)` : An observable sequence which results from the comonadic bind operation

From <https://xgrommx.github.io/rx-book/content/observable/observable_instance_methods/scan.html>

```
This makes it the ideal operator for managing application state.
*/
      .scan((state, action) => this.reducer(state, action), initialState)
      //post-middleware?
      .subscribe(state => super.next(state));
  }
  //...store implementation
}
```

From <https://xgrommx.github.io/rx-book/content/observable/observable_instance_methods/scan.html>

A comonad is, just like a monad, a mathematical structure in category theory.

From <https://softwareengineering.stackexchange.com/questions/322431/what-is-a-comonad-and-how-are-they-useful>

A comonad provides the means of extracting a single value from it. It does not give the means to insert values. So if you want to think of a comonad as a container, it always comes pre-filled with contents, and it lets you peek at it.

From <https://bartoszmilewski.com/2017/01/02/comonads/>

Managing Middleware with let

let me have the entire observable...

(let demo | store demo)

Middleware has been removed in ngrx/store v2. *It is still worth reading through this section to understand* `Let` *as it can be used with selectors.*

While most operators are passed emitted values from the observable, `let` is handed the *entire* observable. This allows the opportunity to tack on extra operators and functionality, before returning the source observable. While this may seem like a small nuance, it fits perfectly into situations like middleware or selectors (to be discussed later), where the consumer would like to define a composable, reusable block of code to be inserted at a particular slot in an observable chain.

Basic functionality of let

```
const myArray = [1,2,3,4,5];
const myObservableArray = Rx.Observable.from(myArray);
const test = myObservableArray
    .map(val => val + 1)
  //this fails, let behaves differently than most operators
  //.let(val => val + 2)
  .subscribe(val => console.log('VALUE FROM ARRAY: ', val));


const letTest = myObservableArray
  .map(val => val + 1)
  //'let' me have the entire observable
  .let(obs => obs.map(val => val + 2))
  .subscribe(val => console.log('VALUE FROM ARRAY WITH let: ', val));
//let provides flexibility to add multiple operators to source observable then return
const letTestThree = myObservableArray
       .map(val => val + 1)
  //'let' me have the entire observable
  .let(obs => obs
    .map(val => val + 2)
    //also, just return evens
    .filter(val => val % 2 === 0)
  )
  .subscribe(val => console.log('let WITH MULTIPLE OPERATORS: ', val));
//pass in your own function to add operators to observable
const obsArrayPlusYourOperators = (yourAppliedOperators) => {
  return myObservableArray
    .map(val => val + 1)
    .let(yourAppliedOperators)
 };
const addTenThenTwenty = obs => obs.map(val => val + 10).map(val => val + 20);
const letTestFour = obsArrayPlusYourOperators(addTenThenTwenty)
       .subscribe(val => console.log('let FROM FUNCTION:', val));
```

The let operator is a perfect fit for *@ngrx/store* middleware as an entry point is required for the user to add custom functionality *before* or *after* reducers output state. This is the basis for how pre and post middleware is applied in @ngrx/store.

Adding let for a Middleware Entry Point

```
class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    preMiddleware,
    postMiddleware,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       //The let operate accepts the source observable, returning a new observable
       //@ngrx/store composes middleware so you can supply more than 1 function,
       //for our simple example we will accept one pre and post middleware
       //Middleware signature: (obs) => obs
       .let(preMiddleware)
       .scan((state, action) => this.reducer(state, action), initialState)
       .let(postMiddleware)
```

```
      .subscribe(state => super.next(state));
  }
  //...store implementation
}
//create basic middleware that logs actions before reducer, and newly outputted state
const preMiddleware = obs => { return obs.do(val => console.log('ACTION: ', val))};
const postMiddleware = obs => { return obs.do(val => console.log('STATE: ', val))};
...create store supplying middleware
```

To recap up to this point:

- Dispatched actions are next'ed into the dispatcher (Subject)
- The Dispatcher has 3 operators applied:
  - let - Passed an observable of actions
  - scan - Calls each reducer with current state and action, outputting new representation of state
  - let - Passed an observable of state
- The new representation of state is then nexted into store (BehaviorSubject), to be emitted to subscribers

This is the gist of the inner workings of store.

Slicing state with map

I'll take the corner piece...

(demo)

The cornerstone function for projecting data from a collection is **map**. Map applies a specified function to each item, returning a new representation of that item. Because application state is a key/value object map of sections of state, it's simple to provide a helper function to return the requested slice of state based on a string, or any other relevant selector.

Providing Slices of State with map

```
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}
class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    preMiddleware,
    postMiddleware,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       .let(preMiddleware)
       .scan((state, action) => this.reducer(state, action), initialState)
       .let(postMiddleware)
       .subscribe(state => super.next(state));
  }

  //Map makes it easy to select slices of state that will be needed for your components
  //This is a simple helper function to make grabbing sections of state more concise
  select(key : string) {
    return this.map(state => state[key]);
  }
//...store implementation
}
//...create store
//utilizing the store select helper
const subscriber = store
  .select('person')
  .subscribe(val => console.log('VALUE OF PERSON:', val));
```

Managing Updates with distinctUntilChanged

Don't call me until you've changed...

(distinctUntilChanged demo | store demo)

Each view in our application generally is concerned with their own particular slices of state. For performance reasons, we would prefer not to emit new values from the selected state slices unless an update has been made. Luckily for us, RxJS has an operator for exactly this scenario (notice the trend). The distinctUntilChanged operator will only emit when the next value is unique, based on the previously

emitted value. In cases of numbers and strings, this means equal numbers and strings; in the case of objects, if the object reference is the same, a new object will not be emitted.

Utilizing distinctUntilChanged With Basic Values and Objects

```
//only output distinct values, based on the last emitted value
const myArrayWithDuplicatesInARow = new Rx.Observable
  .from([1,1,2,2,3,1,2,3]);

const distinctSub = myArrayWithDuplicatesInARow
       .distinctUntilChanged()
       //output: 1,2,3,1,2,3
       .subscribe(val => console.log('DISTINCT SUB:', val));

const nonDistinctSub = myArrayWithDuplicatesInARow
       //output: 1,1,2,2,3,1,2,3
       .subscribe(val => console.log('NON DISTINCT SUB:', val));
const sampleObject = {name: 'Test'};
const myArrayWithDuplicateObjects = new Rx.Observable.from([sampleObject, sampleObject, sampleObject]);
//only out distinct objects, based on last emitted value
const nonDistinctObjects = myArrayWithDuplicateObjects
  .distinctUntilChanged()
  //output: 'DISTINCT OBJECTS: {name: 'Test'}'
  .subscribe(val => console.log('DISTINCT OBJECTS:', val));
```

Recall that store reducers always have a default case, returning the previous state if dispatched actions are not relevant. This means, when selecting slices of state within your applications, you will not receive updates unless your particular slice has been updated. This aids in making your Store application more efficient.

distinctUntilChanged With Store

```
class Dispatcher extends Rx.Subject{
  dispatch(value : any) : void {
    this.next(value);
  }
}
class Store extends Rx.BehaviorSubject{
  constructor(
    private dispatcher,
    private reducer,
    preMiddleware,
    postMiddleware,
    initialState = {}
  ){
    super(initialState);
    this.dispatcher
       .let(preMiddleware)
       .scan((state, action) => this.reducer(state, action), initialState)
       .let(postMiddleware)
       .subscribe(state => super.next(state));
  }

  /*
    distinctUntilChanged only emits new values when output is distinct, per last emitted value.
    In the example below, the observable with the distinctUntilChanged operator will emit one less
    value then the other with only the map operator applied
  */
  select(key : string) {
    return this
      .map(state => state[key])
      .distinctUntilChanged();
  }
//...store implementation
}
// add reducers...
// configure store...
const subscriber = store
  //with distinctUntilChanged
  .select('person')
  .subscribe(val => console.log('PERSON WITH DISTINCTUNTILCHANGED:', val));
const subscriberTwo = store
  //without distinctUntilChanged, will print out extra time
  .map(state => state.person)
  .subscribe(val => console.log('PERSON WITHOUT DISTINCTUNTILCHANGED:', val));
//dispatch a few actions
dispatcher.dispatch({
  type: 'ADD_INFO',
  payload: {
    name: 'Brian',
    message: 'Exploring Reduce!'
  }
});
//person will not be changed
dispatcher.dispatch({
  type: 'ADD_HOUR'
});
```

3. Walkthrough

The Sample Application

The [10] sample application we will be building is a **simple party planner**. The user should be able to

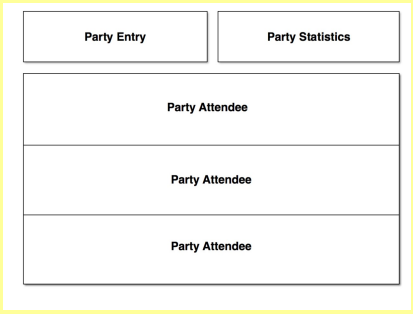
1. Enter a list of attendees and their guests
2. Keep track of who has confirmed attendance
3. Filter attendees by particular criteria
4. View important statistics regarding the event

Throughout the creation of the application we will explore the core concepts of @ngrx/store and discuss popular patterns and best practices.

There are two links provided above each section, *Work Along* and *Completed Lesson*. The *Work Along* link picks up at the beginning of each lesson if you wish to code along as the concepts are presented. Otherwise, the *Completed Lesson* link allows you to start at the end point of the current lesson.

Without further ado, let's get started!

10 Party Planner Application Layout



Setting Up The First Reducer

(Work Along | Completed Lesson)

Reducers are the foundation to your store application. As the application *store* maintains state, reducers are the workhorse behind the manipulation and output of new state representations as actions are dispatched. Each reducer should be focused on a specific section, or slice of state, similar to a table in a database.

Creating a reducer is quite simple once you get used to one common idiom, *never mutate previous state* and *always return a new representation of state when a relevant action is dispatched.* If you are new to store or the Redux pattern this takes some practice to feel natural. Instead of using mutative methods like push, or reassigning values to previously existing objects, you will instead lean on none mutating methods like concat and Object.assign to return new values. Still confused? Let's see what this looks like in practice with our people reducer.

The people reducer needs to handle five actions

- Adding a person (invitee)
- Removing a person (invitee),
- Adding guests to a person (invitee),
- Removing guests from a person (invitee)
- Toggling whether they (invitee and guests) are attending the event

To do this, we will create a reducer function, accepting previous state and the currently dispatched action. We then need to implement a case statement that performs the correct state recalculation when a relevant action is dispatched.

People Reducer

```
Object.assign(target, ...sources)

Parameters

target
```

action. We then need to implement a case statement that performs the correct state recalculation when a relevant action is dispatched.

People Reducer

```
const details = (state, action) => {
  switch(action.type){
    case ADD_GUEST:
      if (state.id === action.payload) {
          return Object.assign({}, state, {guests: state.guests + 1});
      }
      return state;

    case REMOVE_GUEST:
      if (state.id === action.payload) {
          return Object.assign({}, state, {guests: state.guests - 1});
      }
      return state;

    case TOGGLE_ATTENDING:
      if (state.id === action.payload) {
          return Object.assign({}, state, {attending: !state.attending});
      }
      return state;

    default:
      return state;
  }
}

//remember to avoid mutation within reducers
export const people = (state = [], action) => {
  switch(action.type){
    case ADD_PERSON:
      return [
        ...state,
        Object.assign({}, {id: action.payload.id, name: action.payload.name, guests:0, attending: false})
      ];

    case REMOVE_PERSON:
      return state
        .filter(person => person.id !== action.payload);
     //to shorten our case statements, delegate detail updates
     //to second private reducer

    case ADD_GUEST:
      return state.map(person => details(person, action));

    case REMOVE_GUEST:
      return state.map(person => details(person, action));

    case TOGGLE_ATTENDING:
      return state.map(person => details(person, action));
     //always have default return of previous state when action is not relevant

    default:
      return state;
  }
}
```

`Object.assign(target, ...sources)`

Parameters

target
The target object — what to apply the sources' properties to, which is returned after it is modified.

sources
The source object(s) — objects containing the properties you want to apply.

Return value
The target object.

Note: `Object.assign()` does not throw on `null` or `undefined` sources.

Example

```
const obj = { a: 1 };
const copy = Object.assign({}, obj);
console.log(copy); // { a: 1 }
```

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign>

TypeScript - Array filter()

`filter()` method **creates a new array** with all elements that pass the test implemented by the provided function.

Syntax

```
array.filter (callback[, thisObject]);
```

Parameter Details

- callback – Function to test for each element.
- thisObject – Object to use as this when executing callback.

Return Value
Returns created array.

Example

```
function isBigEnough(element, index, array) {
   return (element >= 10);
}

var passed = [12, 5, 8, 130, 44].filter(isBigEnough);
console.log("Test Value : " + passed );
```

Its output is as follows –

```
Test Value :12,130,44
```

From <https://www.tutorialspoint.com/typescript/typescript_array_filter.htm>

Configuring Store Actions

(Work Along | Completed Lesson)

The only way to modify state within a @ngrx/store application is by dispatching actions. Because of this, a log of actions should present a clear, readable, history of user interaction. Actions are generally defined as string constants or as static string values on services encapsulating a particular action type. In the latter case, functions are provided to return an appropriate action given the correct input. These methods, which help standardize your actions while providing additional type safety, are known as *action creators*.

For the case of our starter application we will export a string constant for each application action. These will then be used as the keys to our reducer case statements and the type for every dispatched action.

Initial Actions

```
//Person Action Constants
export const ADD_PERSON = 'ADD_PERSON';
export const REMOVE_PERSON = 'REMOVE_PERSON';
export const ADD_GUEST = 'ADD_GUEST';
export const REMOVE_GUEST = 'REMOVE_GUEST';
```

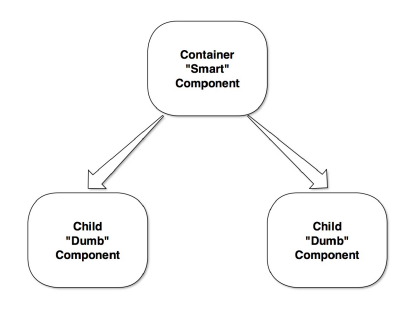
```
export const TOGGLE_ATTENDING = 'TOGGLE_ATTENDING';
```

Utilizing Container Components

(Work Along | Completed Lesson)

Components in your Store application will fall into two categories, [11] smart and dumb. So what does it mean to be a smart component vs a dumb component?

11 Smart (Container) and Dumb Components



Smart, or Container components should be your root level, routable components. These components generally have direct access to store or a derivative. Smart components handle view events and the dispatching of actions, whether through a service or directly. Smart components also handle the logic behind events emitted up from child components within the same view.

Dumb, or Child components are generally for presentation only, relying exclusively on @Input parameters, acting on the received data in an appropriate manner. When relevant events occur in dumb components, they are emitted up to be handled by a parent smart component. Dumb components will make up the majority of your application, as they should be small, focused, and reusable.

The party planner application will need a single container component. This component will be responsible for passing appropriate state down to each child component and dispatching actions based on events emitted by our dumb components, person-input, person-list, and in the future party-stats. For now, we will manually subscribe to store in the constructor, setting updates to a component-level property. In future lessons we will explore how to utilize the AsyncPipe to reduce some of this boilerplate.

Container Component

```
@Component({
    selector: 'app',
    template: `
      <h3>@ngrx/store Party Planner</h3>
      <person-input
        (addPerson)="addPerson($event)"
      >
      </person-input>
      <person-list
        [people]="people"
        (addGuest)="addGuest($event)"
        (removeGuest)="removeGuest($event)"
        (removePerson)="removePerson($event)"
        (toggleAttending)="toggleAttending($event)"
      >
      </person-list>
    `,
    directives: [PersonList, PersonInput]
})
export class App {
    public people;
    private subscription;

    constructor(
     private _store: Store
    ){
      /*
        demonstrating use without the async pipe,
        we will explore the async pipe in the next lesson
      */
      this.subscription = this._store
        .select('people')
        .subscribe(people => {
          this.people = people;
      });
```

Containment

```
    }
    //all state-changing actions get dispatched to and handled by reducers
    addPerson(name){
      this._store.dispatch({type: ADD_PERSON, payload: {id: id(), name})
    }

    addGuest(id){
      this._store.dispatch({type: ADD_GUEST, payload: id});
    }

    removeGuest(id){
      this._store.dispatch({type: REMOVE_GUEST, payload: id});
    }

    removePerson(id){
      this._store.dispatch({type: REMOVE_PERSON, payload: id});
    }

    toggleAttending(id){
      this._store.dispatch({type: TOGGLE_ATTENDING, payload: id})
    }
    /*
      if you do not use async pipe and create manual subscriptions
      always remember to unsubscribe in ngOnDestroy
    */
    ngOnDestroy(){
      this.subscription.unsubscribe();
    }
}
```

Dumb Component - PersonList

```
@Component({
    selector: 'person-list',
    template: `
      <ul>
        <li
          *ngFor="let person of people"
          [class.attending]="person.attending"
        >
           {{person.name}} - Guests: {{person.guests}}
           <button (click)="addGuest.emit(person.id)">+</button>
           <button *ngIf="person.guests" (click)="removeGuest.emit(person.id)">-</button>
           Attending?
           <input type="checkbox" [(ngModel)]="person.attending" (change)="toggleAttending.emit(person.id)" />
           <button (click)="removePerson.emit(person.id)">Delete</button>
        </li>
      </ul>
    `
})
export class PersonList {
    /*
      "dumb" components do nothing but display data based on input and
      emit relevant events back up for parent, or "container" components to handle
    */
    @Input() people;
    @Output() addGuest = new EventEmitter();
    @Output() removeGuest = new EventEmitter();
    @Output() removePerson = new EventEmitter();
    @Output() toggleAttending = new EventEmitter();
}
```

Dumb Component - PersonInput

```
@Component({
    selector: 'person-input',
    template: `
      <input #personName type="text" />
      <button (click)="add(personName)">Add Person</button>
    `
})
export class PersonInput {
    @Output() addPerson = new EventEmitter();

    add(personInput){
      this.addPerson.emit(personInput.value);
      personInput.value = '';
    }
}
```

Utilizing the AsyncPipe

(Work Along | Completed Lesson)

The AsyncPipe is a unique, stateful pipe in Angular 2 meant for handling **both Observables and Promises**. When using the AsyncPipe in a template expression with Observables, the supplied Observable is subscribed to, with emitted values being displayed within your view. This pipe also handles unsubscribing to the supplied observable, saving you the mental overhead of manually cleaning up subscriptions in ngOnDestroy. In a @ngrx/store application, you will find yourself leaning on the AsyncPipe heavily in nearly all of your component views. For a more detailed explanation of exactly how the AsyncPipe works, check out my article *Understand and Utilize the AsyncPipe in Angular 2* or free egghead.io video Using the Async Pipe in Angular 2.

Utilizing the AsyncPipe in our templates is easy. You can pipe any Observable (or promise) through **async** and a subscription will be created, updating the template value on source emission. Because we are using the AsyncPipe, we can also remove the manual subscription from the component constructor and unsubscribe from the ngOnDestroy lifecycle hook. This is now handled for us behind the scenes.

Refactoring to Async Pipe

```
@Component({
    selector: 'app',
    template: `
      <h3>@ngrx/store Party Planner</h3>
      <person-input
        (addPerson)="addPerson($event)"
      >
      </person-input>
      <person-list
        [people]="people | async"
        (addGuest)="addGuest($event)"
        (removeGuest)="removeGuest($event)"
        (removePerson)="removePerson($event)"
        (toggleAttending)="toggleAttending($event)"
      >
      </person-list>
    `,
    directives: [PersonList, PersonInput]
})
export class App {
    public people;
    private subscription;

    constructor(private _store: Store) {
      /*
        Observable of people, utilizing the async pipe
        in our templates this will be subscribed to, with
        new values being displayed in our template.
        Unsubscribe will be called automatically when component
        is disposed.
      */
      this.people = _store.select('people');
    }
    //all state-changing actions get dispatched to and handled by reducers
    addPerson(name){
      this._store.dispatch({type: ADD_PERSON, payload: name})
    }

    addGuest(id){
      this._store.dispatch({type: ADD_GUEST, payload: id});
    }

    removeGuest(id){
      this._store.dispatch({type: REMOVE_GUEST, payload: id});
    }

    removePerson(id){
      this._store.dispatch({type: REMOVE_PERSON, payload: id});
    }

    toggleAttending(id){
      this._store.dispatch({type: TOGGLE_ATTENDING, payload: id})
    }
    //ngOnDestroy to unsubscribe is no longer necessary
}
```

Taking Advantage of ChangeDetection.OnPush

(Work Along | Completed Lesson)

Utilizing a centralized state tree in Angular 2 cannot only bring benefits in predictability and maintability, but also performance. To enable this performance benefit we can utilize the changeDetectionStrategy of OnPush.

The concept behind OnPush is straightforward. When components rely solely on inputs, and those input references do not change, Angular can skip running change detection on that section of the component

tree. As discussed previously, all delegating of state should be handled in smart, or top level components. This leaves the majority of components in our application relying solely on input, safely allowing us to set the ChangeDetectionStrategy to OnPush in the component definition. These components can now forgo change detection until necessary, giving us a free performance boost.

To utilize OnPush change detection in our components, we need to set the changeDetection propery in the @Component decorator to ChangeDetection.OnPush. That's it! Angular will now ignore change detection on our *dumb* components and children of these components until there is a change in their input references.

Updating to ChangeDetection.OnPush

```
@Component({
    selector: 'person-list',
    template: `
      <ul>
        <li
          *ngFor="let person of people"
          [class.attending]="person.attending"
        >
           {{person.name}} - Guests: {{person.guests}}
           <button (click)="addGuest.emit(person.id)">+</button>
           <button *ngIf="person.guests" (click)="removeGuest.emit(person.id)">-</button>
           Attending?
           <input type="checkbox" [(ngModel)]="person.attending" (change)="toggleAttending.emit(person.id)" />
           <button (click)="removePerson.emit(person.id)">Delete</button>
        </li>
      </ul>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
/*
  with 'onpush' change detection, components which rely solely on
  input can skip change detection until those input references change,
  this can supply a significant performance boost
*/
export class PersonList {
    /*
      "dumb" components do nothing but display data based on input and
      emit relevant events back up for parent, or "container" components to handle
    */
    @Input() people;
    @Output() addGuest = new EventEmitter();
    @Output() removeGuest = new EventEmitter();
    @Output() removePerson = new EventEmitter();
    @Output() toggleAttending = new EventEmitter();
}
```

Expanding State

(Work Along | Completed Lesson)

The majority of store applications will be made up of multiple reducers, each managing their own slice of state. For this example we will have two, one to manage party attendees and the other to represent the currently active filter being applied to this list. Let's first write our action constants, specifying the allowed filters to be applied by the user.

It's now time to create the *partyFilter* reducer. For this functionality we have a couple of options. The first would be to simply return a string representation of which filter should be applied. We could then write a method, either in a service or component, that filters the list based on the current active party filter. While this works, it is more extensible to return the function to be applied to the party list based on the current party filter state. In the future, adding more filters is as simple as creating a new case statement to return the appropriate projection function.

Party Filter Reducer

```
import {
  SHOW_ATTENDING,
  SHOW_ALL,
  SHOW_WITH_GUESTS
} from './actions';
//return appropriate function depending on selected filter
export const partyFilter = (state = person => person, action) => {
    switch(action.type){
        case SHOW_ATTENDING:
            return person => person.attending;
        case SHOW_ALL:
            return person => person;
        case SHOW_WITH_GUESTS:
            return person => person.guests;
        default:
            return state;
    }
```

```
};
```

Party Filter Actions

```
//Party Filter Constants
export const SHOW_ATTENDING = 'SHOW_ATTENDING';
export const SHOW_ALL = 'SHOW_ALL';
export const SHOW_WITH_GUESTS = 'SHOW_GUESTS';
```

Party Filter Select

```
import {Component, Output, EventEmitter} from "angular2/core";
import {
  SHOW_ATTENDING,
  SHOW_ALL,
  SHOW_WITH_GUESTS
} from './actions';
@Component({
    selector: 'filter-select',
    template: `
      <div class="margin-bottom-10">
        <select #selectList (change)="updateFilter.emit(selectList.value)">
            <option *ngFor="let filter of filters" value="{{filter.action}}">
                {{filter.friendly}}
            </option>
        </select>
      </div>
    `
})
export class FilterSelect {
    public filters = [
        {friendly: "All", action: SHOW_ALL},
        {friendly: "Attending", action: SHOW_ATTENDING},
        {friendly: "Attending w/ Guests", action: SHOW_WITH_GUESTS}
      ];
    @Output() updateFilter : EventEmitter<string> = new EventEmitter<string>();
}
```

Slicing State for Views

(Work Along | Completed Lesson)

Store can be thought of as your *client-side database*. Because store is the aggregate of state in our application we need to be able to *query* against it, returning relevant state slices and projections. This is an area where the RxJS foundation of store really shines.

To select appropriate slices of state for consumption you can start by using the Rx implementations of classic JavaScript collection operations in which you have grown accustom. Store also provides a helper function, select, which accepts a string or function, applying map and distinctUntilChanged behind the scenes to return an Observable of the appropriate section of state. As your needs progress, RxJS offers a multitude of powerful operators to suit any use-case.

In this example, we are going to slice our state into multiple Observables, representing the party attendees, current filter, attendance and guest count. In the next lesson we will see how to streamline this utilizing popular RxJS operators.

Projecting State Without Combination Operators

```
@Component({
    selector: 'app',
    template: `
      <h3>@ngrx/store Party Planner</h3>
      <party-stats
        [invited]="(people | async)?.length"
        [attending]="(attending | async)?.length"
        [guests]="(guests | async)"
      >
      </party-stats>
      <filter-select
        (updateFilter)="updateFilter($event)"
      >
      </filter-select>
      <person-input
        (addPerson)="addPerson($event)"
      >
      </person-input>
      <person-list
        [people]="people | async"
        [filter]="filter | async"
        (addGuest)="addGuest($event)"
        (removeGuest)="removeGuest($event)"
        (removePerson)="removePerson($event)"
        (toggleAttending)="toggleAttending($event)"
```

```
      >
      </person-list>
    `,
    directives: [PersonList, PersonInput, FilterSelect, PartyStats]
})
export class App {
    public people;
    private subscription;

    constructor(
     private _store: Store
    ){
      this.people = _store.select('people');
      /*
        this is a naive way to handle projecting state, we will discover a better
        Rx based solution in next lesson
      */
      this.filter = _store.select('partyFilter');
      this.attending = this.people.map(p => p.filter(person => person.attending));
      this.guests = this.people
          .map(p => p.map(person => person.guests)
                     .reduce((acc, curr) => acc + curr, 0));
    }
    //...rest of component
}
```

Now that we have all the necessary data, we can pass it down to our "dumb" components to be presented accordingly. Our container component will handle any actions emitted back up the chain, dispatching the appropriate events.

Manually Passing Down and Applying Filters

```
@Component({
    selector: 'person-list',
    template: `
      <ul>
        <li
          *ngFor="let person of people.filter(filter)"
          [class.attending]="person.attending"
        >
           {{person.name}} - Guests: {{person.guests}}
           <button (click)="addGuest.emit(person.id)">+</button>
           <button *ngIf="person.guests" (click)="removeGuest.emit(person.id)">-</button>
           Attending?
           <input type="checkbox" [checked]="person.attending" (change)="toggleAttending.emit(person.id)" />
           <button (click)="removePerson.emit(person.id)">Delete</button>
        </li>
      </ul>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class PersonList {
    @Input() people;
    //for now, we will pass filter down and apply
    @Input() filter;
    @Output() addGuest = new EventEmitter();
    @Output() removeGuest = new EventEmitter();
    @Output() removePerson = new EventEmitter();
    @Output() toggleAttending = new EventEmitter();
}
```

Projecting State for View with combineLatest and withLatestFrom

(Work Along | Completed Lesson)

While building out your @ngrx/store application, many of your components will require slices of state output from multiple reducers. One example of this would be a list that is adjusted by a filter, both managed through store. When the filter is changed, the list needs to be updated to reflect this change. So how do we facilitate this interaction? RxJS provides two operators that you will consistently utilize, combineLatest and withLatestFrom.

The combineLatest operator accepts an unspecified number of observables, emitting the last emitted value from each when any of the provided observables emit. These values are passed to a projection function for you to form the appropriate projection.

combineLatest Example

(demo)

```
//timerOne emits first value at 1s, then once every 4s
const timerOne = Rx.Observable.timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = Rx.Observable.timer(2000, 4000)
```

```
//timerThree emits first value at 3s, then once every 4s
const timerThree = Rx.Observable.timer(3000, 4000)
//when one timer emits, emit the latest values from each timer as an array
const combined = Rx.Observable
.combineLatest(
    timerOne,
    timerTwo,
    timerThree
);
const subscribe = combined.subscribe(latestValues => {
       //grab latest emitted values for timers one, two, and three
       const [timerValOne, timerValTwo, timerValThree] = latestValues;
  /*
       Example:
    timerOne first tick: 'Timer One Latest: 1, Timer Two Latest:0, Timer Three Latest: 0
    timerTwo first tick: 'Timer One Latest: 1, Timer Two Latest:1, Timer Three Latest: 0
    timerThree first tick: 'Timer One Latest: 1, Timer Two Latest:1, Timer Three Latest: 1
  */
  console.log(
    `Timer One Latest: ${timerValOne},
     Timer Two Latest: ${timerValTwo},
     Timer Three Latest: ${timerValThree}`
   );
});
//combineLatest also takes an optional projection function
const combinedProject = Rx.Observable
.combineLatest(
    timerOne,
    timerTwo,
    timerThree,
    (one, two, three) => {
      return `Timer One (Proj) Latest: ${one},
              Timer Two (Proj) Latest: ${two},
              Timer Three (Proj) Latest: ${three}`
    }
);
//log values
const subscribe = combinedProject.subscribe(latestValuesProject => console.log(latestValuesProject));
```

The withLatestFrom operator is similar, except it only combines the last emitted values from the provided observables when the source observable emits. This is useful when your projection depends first on a single source, aided by multiple other sources.

withLatestFrom Example

(demo)

```
//Create an observable that emits a value every second
const myInterval = Rx.Observable.interval(1000);
//Create an observable that emits immediately, then every 5 seconds
const myTimer = Rx.Observable.timer(0, 5000);
//Every time interval emits, also get latest from timer and add the two values
const latest = myInterval
  .withLatestFrom(myTimer)
  .map(([interval, timer]) => {
    console.log(`Latest Interval: ${interval}`);
    console.log(`Latest Timer: ${timer}`);
    return interval + timer;
  });
//log total
const subscribe = latest.subscribe(val => console.log(`Total: ${val}`));
```

Now that we have an understanding of these *combination* operators, we can clean up our previous example by using Observable.combineLatest. Instead of creating a new observable for each statistic, the state from *people* and *partyFilter* can be combined any time either updates, executing our projection function to calculate party statistics and properly filter the people list.

Refactoring Container View with combineLatest

```
@Component({
    selector: 'app',
    template: `
      <h3>@ngrx/store Party Planner</h3>
      <party-stats
        [invited]="(model | async)?.total"
        [attending]="(model | async)?.attending"
        [guests]="(model | async)?.guests"
      >
      {{guests | async | json}}
      </party-stats>
      <filter-select
        (updateFilter)="updateFilter($event)"
      >
      </filter-select>
      <person-input
```

```
        (addPerson)="addPerson($event)"
      >
      </person-input>
      <person-list
        [people]="(model | async)?.people"
        (addGuest)="addGuest($event)"
        (removeGuest)="removeGuest($event)"
        (removePerson)="removePerson($event)"
        (toggleAttending)="toggleAttending($event)"
      >
      </person-list>
    `,
    directives: [PersonList, PersonInput, FilterSelect, PartyStats]
})
export class App {
    public model;

    constructor(
     private _store: Store
    ){
      /*
        Every time people or partyFilter emits, pass the latest
        value from each into supplied function. We can then calculate
        and output statistics.
      */
      this.model = Observable.combineLatest(
          _store.select('people')
          _store.select('partyFilter'),
          (people, filter) => {
          return {
            total: people.length,
            people: people.filter(filter),
            attending: people.filter(person => person.attending).length,
            guests: people.reduce((acc, curr) => acc + curr.guests, 0)
          }
        });
    }
    //...rest of component
}
```

Extracting Selectors for Reuse

(Work Along | Completed Lesson)

Through the course of building your application you will often utilize similar queries, or projections of state in your views. A common way to eliminate the duplication of this logic is to place popular selections into services, injecting these services where needed in your components or other services. While this certainly works, there is another more flexible, composable way to tackle this issue.

Because nothing about these projections is Angular specific, we can export each small query, or [12] selector, independently without the need for Angular service wrapping. Leveraging the let operator, these selectors can then be mixed and matched for the desired result, whether in components, services, or middleware. This toolbox of targeted, composable queries is called the **selector pattern**.

12 Selector Interface

```
interface Selector<T,V> {
  (state: Observable<T>): Observable<V>
}
```

To accomplish this we will create a new file to house our application selectors. We can then extract the projection function being applied in combineLatest to filter people and produce statistics into a *selector*.

Party Model Selector

```
export const partyModel = () => {
  return state => state
    .map(([people, filter]) => {
      return {
            total: people.length,
            people: people.filter(filter),
            attending: people.filter(person => person.attending).length,
            guests: people.reduce((acc, curr) => acc + curr.guests, 0)
          }
    });
};
```

For further demonstration, let's create two more selectors, one to return an observable of party attendees, and another, building on the previous selector, to calculate the percent of people attending based on those invited. This shows how easy it is to compose these small, focused selectors into powerful queries for use in views and middleware.

Percent Attendance Selector

```
export const attendees = () => {
  return state => state
    .map(s => s.people)
    .distinctUntilChanged();
};
export const percentAttending = () => {
  return state => state
    //build on previous selectors
    .let(attendees())
    .map(p => {
      const totalAttending = p.filter(person => person.attending).length;
      const total = p.length;
      return total > 0 ? (totalAttending / total) * 100 : 0;
    });
};
```

Applying selectors is easy. Simply apply the let operator to the appropriate Observable, supplying the selector(s) of your choosing.

Applying Selectors In Container Component

```
export class App {
    public model;

    constructor(
     private _store: Store
    ){
      /*
        Every time people or partyFilter emits, pass the latest
        value from each into supplied function. We can then calculate
        and output statistics.
      */
      this.model = Observable.combineLatest(
            _store.select('people'),
            _store.select('partyFilter')
          )
          //extracting party model to selector
          .let(partyModel());
      //for demonstration on combining selectors
      this.percentAttendance = _store.let(percentAttending());
    }
    //...rest of component
}
```

Implementing a Meta-Reducer

(Work Along | Completed Lesson)

One of the many advantages to a single, immutable state tree is the ease of implementation for generally tricky features like undo/redo. Since the progression of application state is fully traceable through snapshots of store, the ability to walk back through these snap shots becomes trivial. A popular method for implementing this functionality is through *meta-reducers*.

Despite the ominous sound, *meta-reducers* are actually quite simple in theory and implementation. To create a meta-reducer, you wrap a current reducer in a parent reducer, delegating the majority of actions through the wrapped reducer as normal, stepping in only when defined *meta* actions (such as undo/redo) are dispatched. How does this look in practice? Let's see by creating a reset feature for our party planning application, allowing the user to *start from scratch* if they want to enter all new data.

To encapsulate this functionality, we create a factory function that accepts any reducer and returns our reset reducer. When the reset reducer is initialized, we save the initial state of the wrapped reducer for later use. All that is left is to listen for a specific reset action to be dispatched. If RESET_STATE is not dispatched, the action is passed to the wrapped reducer and state returned as normal.
When RESET_STATE is triggered, the stored initial state is returned instead of the result of invoking the wrapped reducer. Undo/redo can be handled similarly, keeping track of previous actions in local state.

Reset Meta-Reducer

```
export const RESET_STATE = 'RESET_STATE';
const INIT = '__NOT_A_REAL_ACTION__';

export const reset = reducer => {
    let initialState = reducer(undefined, {type: INIT});
```

```
    return function (state, action) {
      //if reset action is fired, return initial state
      if (action.type === RESET_STATE){
        return initialState;
      }
      //calculate next state based on action
      let nextState = reducer(state, action);
      //return nextState as normal when not reset action
      return nextState;
  }
}
```

Wrap Reducer On Bootstrap

```
bootstrap(App, [
  //wrap people in reset meta-reducer
  provideStore({people: reset(people), partyFilter})
]);
```

It is worth noting that the store *root reducer* is itself a meta-reducer, created behind the scenes when calling provideStore (by [14] combineReducers). For each dispatched action, the root reducer invokes each application reducer with the previous state and current action, returning an object map of [reducer]: state[reducer].

14 combineReducers

```
export function combineReducers(reducers: any): Reducer<any> {
  const reducerKeys = Object.keys(reducers);
  const finalReducers = {};
  for (let i = 0; i < reducerKeys.length; i++) {
    const key = reducerKeys[i];

    if (typeof reducers[key] === 'function') {
      finalReducers[key] = reducers[key];
    }
  }

  const finalReducerKeys = Object.keys(finalReducers);

  return function combination(state = {}, action) {
    let hasChanged = false;
    const nextState = {};

    for (let i = 0; i < finalReducerKeys.length; i++) {
      const key = finalReducerKeys[i];
      const reducer = finalReducers[key];
      const previousStateForKey = state[key];
      const nextStateForKey = reducer(previousStateForKey, action);

      nextState[key] = nextStateForKey;

      hasChanged = hasChanged || nextStateForKey !== previousStateForKey;
    }

    return hasChanged ? nextState : state;
  };
}
```

From <https://gist.github.com/btroncone/a6e4347326749f938510>