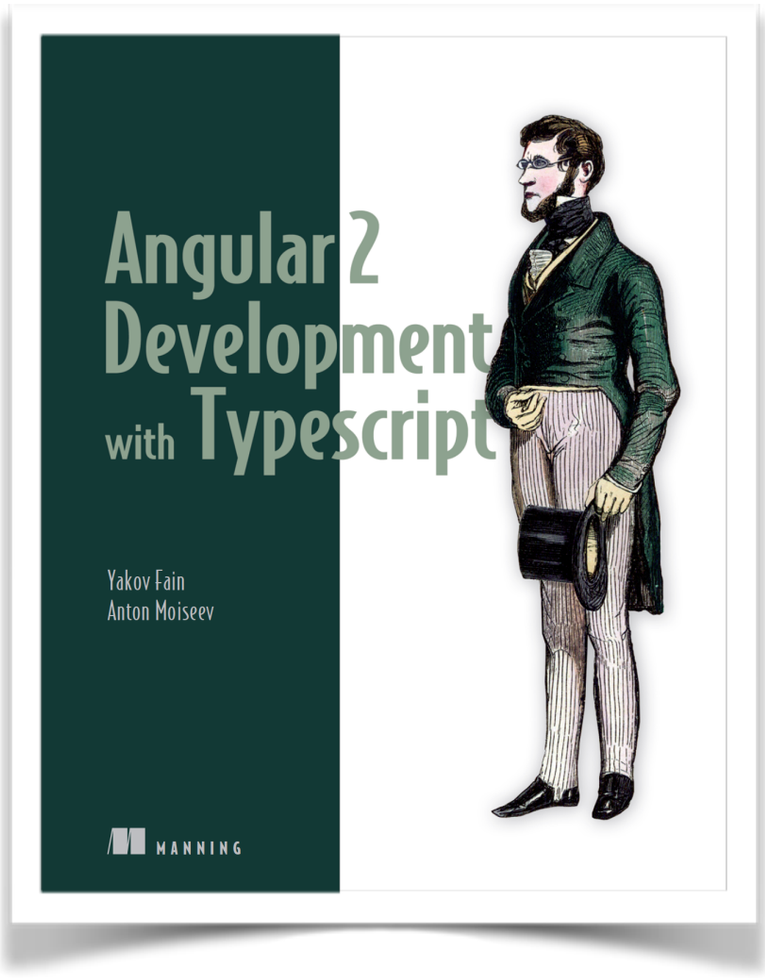
Reactive programming in Angular 2

Yakov Fain

  yfain

About myself

- Solutions Architect at Farata Systems
- Java Champion
- Co-authored the book
  “Angular Development with TypeScript”



50% off at Manning.com
ctwdevnexus17

The Agenda

- Intro to RxJS

  - Observable
  - Observer
  - Operators

- Observables in Angular 2

  - Forms
  - Http
  - Router

RxJS 5

- Github repo:
  https://github.com/ReactiveX/rxjs

- CDN:
  https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js

- Installing RxJS in the npm-based projects:

  npm install rxjs

- Documentation:
  http://reactivex.io/rxjs

Main RxJS players

- **Observable** - a producer of sequences of values
- **Observer** - a consumer of observable values
- **Subscriber** - connects observer with observable
- **Operator** - en-route value transformation

Data Flow

Data Source

Data Flow

Observable

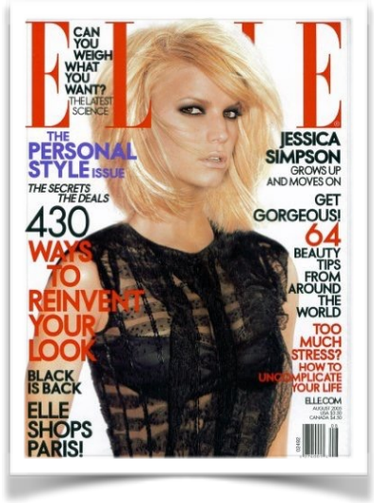
Data Source

Data Flow

Observable

Data Source

next



Data Flow

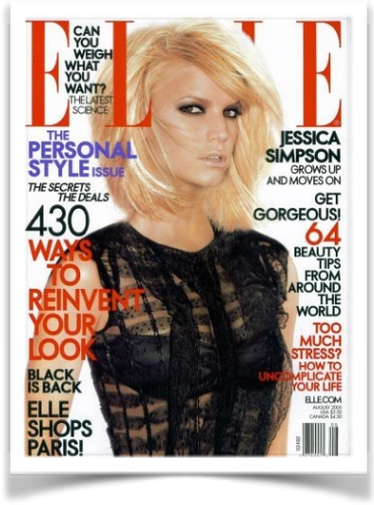
Observer

Subscriber

Observable

Data Source

next



Data Flow

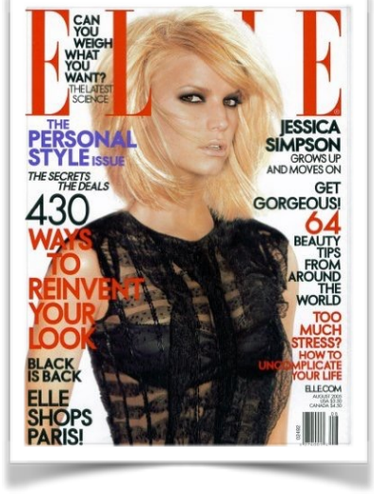
Observer

Subscriber

Observable

Data Source

next



next()

error()

complete()

An Observable allows:

- Subscribe/unsubscribe to its data stream
- Emit the next value to the observer
- Notify the observer about errors
- Inform the observer about the stream completion

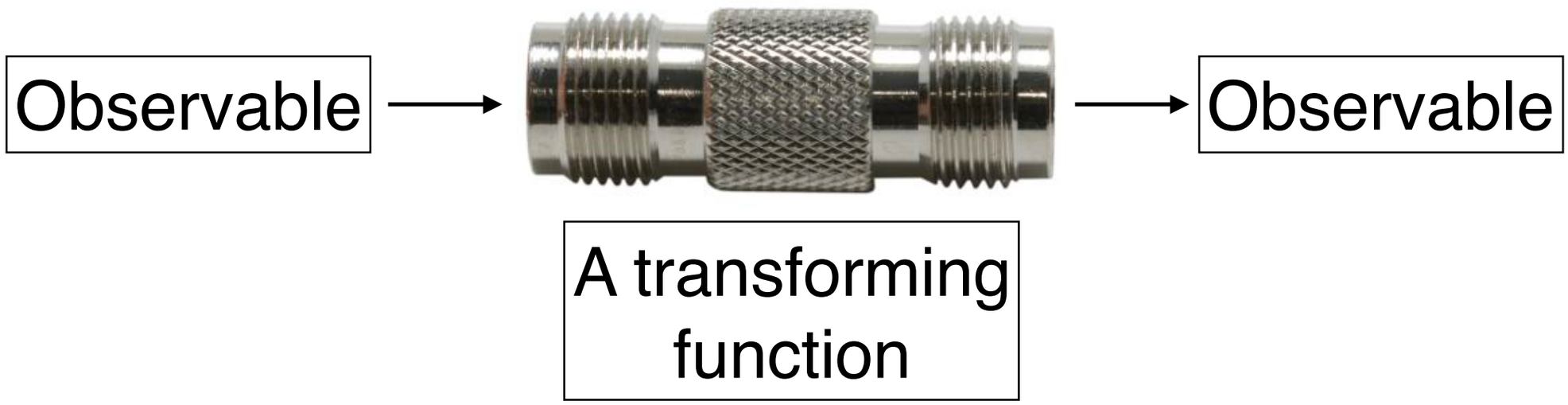
An Observer provides:

- A function to handle the next value from the stream
- A function to handle errors
- A function to handle end-of-stream

Creating an Observable

- Observable.create() - returns Observable that can invoke methods on Observer
- Observable.from() - converts an array or iterable into Observable
- Observable.fromEvent() - converts an event into Observable
- Observable.fromPromise() - converts a Promise into Observable
- Observable.range() - returns a sequence of integers in the specified range

An Operator



Let’s have a beer

```
let beers = [
    {name: "Stella", country: "Belgium", price: 9.50},
    {name: "Sam Adams", country: "USA", price: 8.50},
    {name: "Bud Light", country: "USA", price: 6.50},
    {name: "Brooklyn Lager", country: "USA", price: 8.00},
    {name: "Sapporo", country: "Japan", price: 7.50}
];
```

An Operator


```
observableBeers
        .filter(beer => beer.price < 8))
```

Observable Beer

Creating the Observable

```
observableBeers = Rx.Observable.from(beers)
    .filter(beer => beer.price < 8)
    .map(beer => beer.name + ": $" + beer.price);

observableBeers
    .subscribe(
        beer => console.log(beer),
        err  => console.error(err),
        ()   => console.log("Streaming is over")
    );
```

Observable Beer

Operators

Observer

```
observableBeers = Rx.Observable.from(beers)
    .filter(beer => beer.price < 8)
    .map(beer => beer.name + ": $" + beer.price);

observableBeers
    .subscribe(
      beer => console.log(beer),
      err => console.error(err),
      () => console.log("Stream is over")
    );
```

Observable Beer

No streaming yet

Streaming begins

```
observableBeers = Rx.Observable.from(beers)
    .filter(beer => beer.price < 8)
    .map(beer => beer.name + ": $" + beer.price);

observableBeers
    .subscribe(
        beer => console.log(beer),
        err  => console.error(err),
        ()   => console.log("Streaming is over")
    );
```

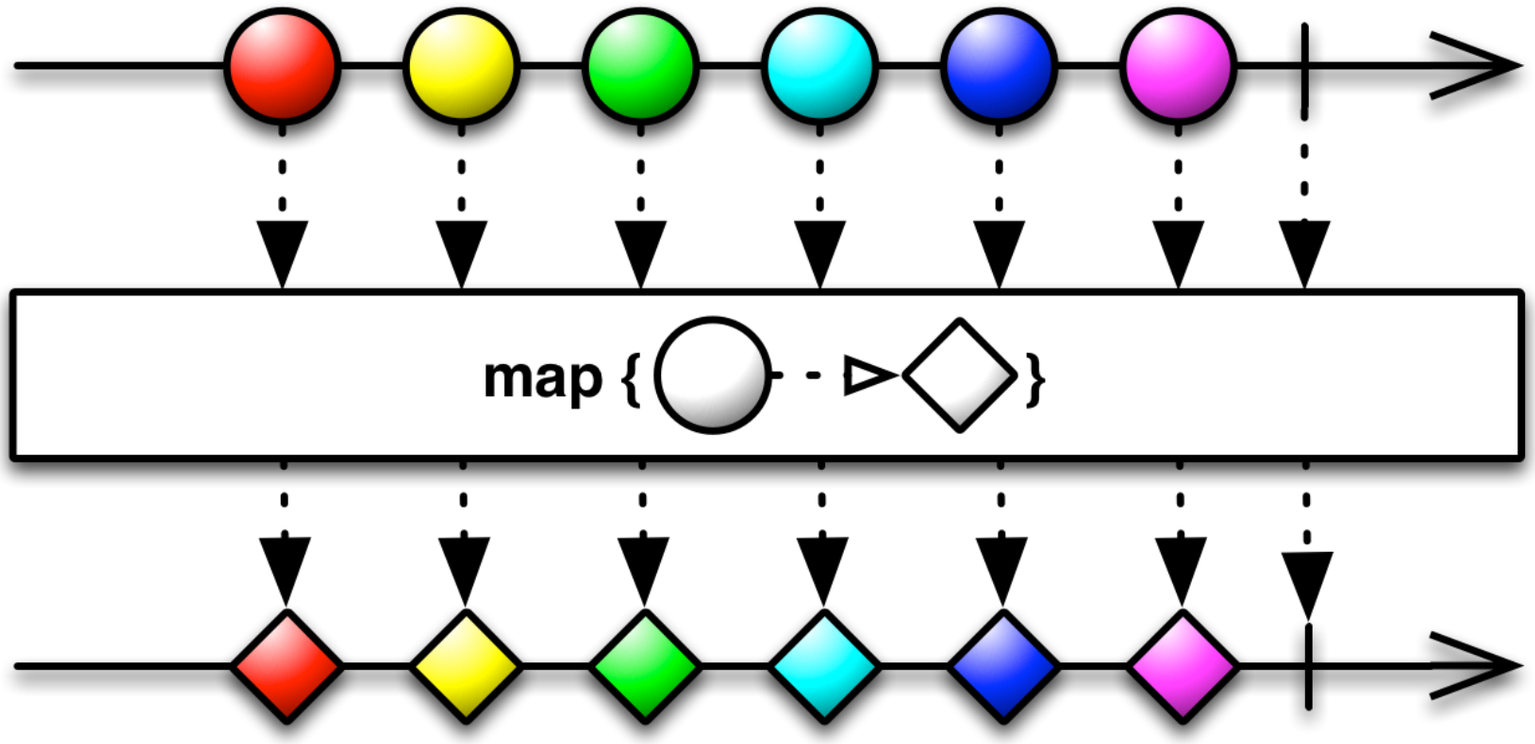
Demo

http://bit.ly/2kogm42

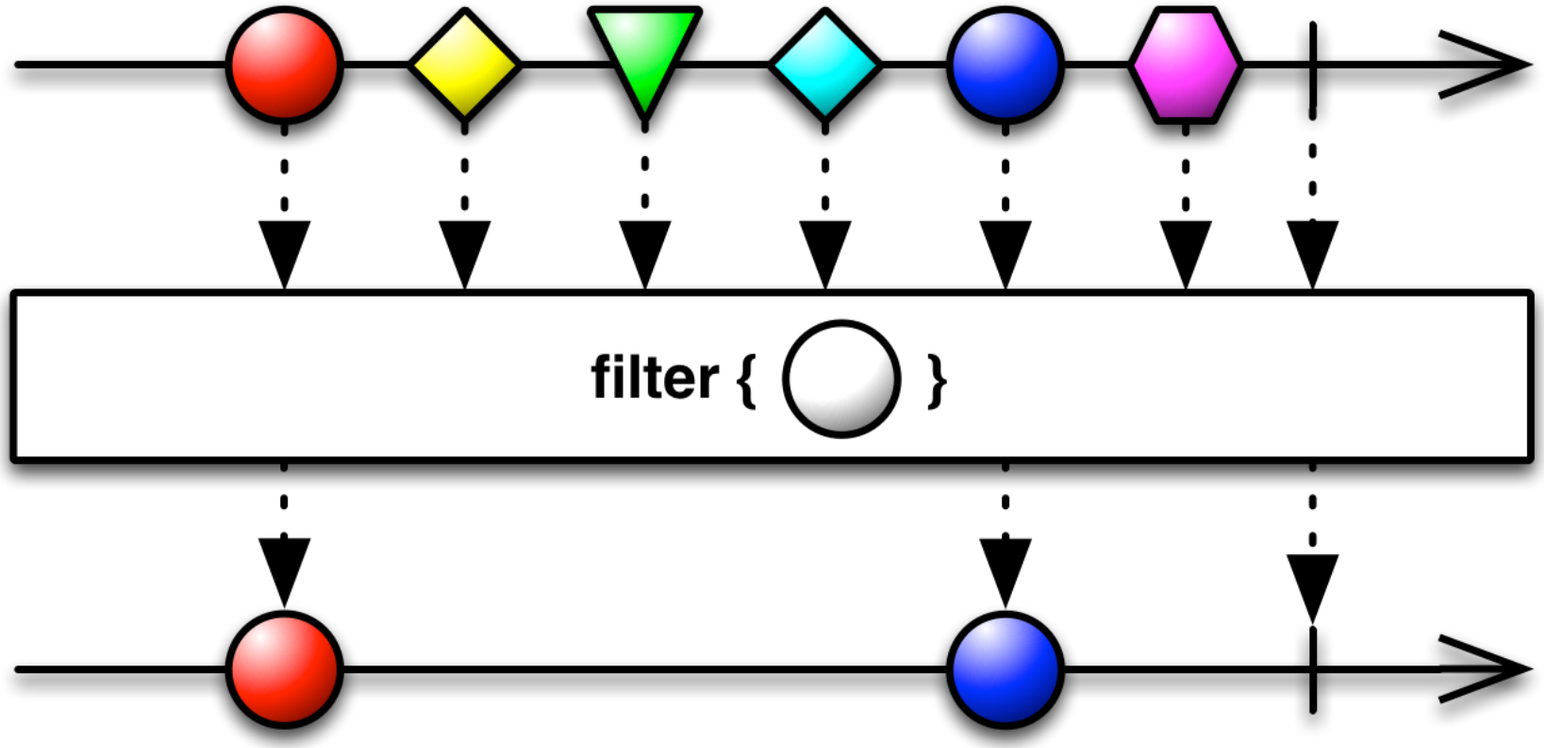
http://bit.ly/2jm69aM

Marble Diagrams

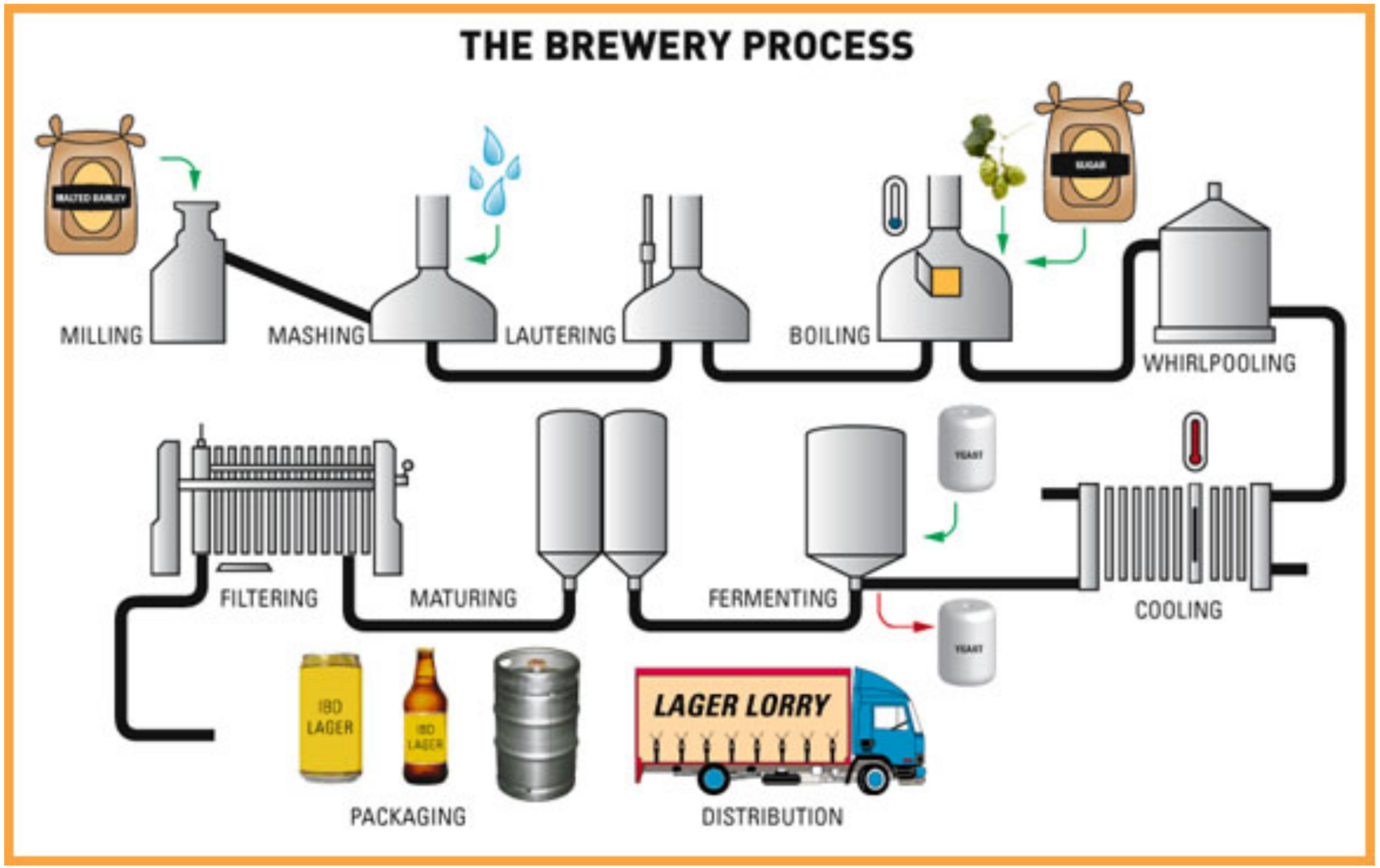
Observable map(function){}



Observable filter(function){}



RX: the data moves across your algorithm



Operator

Operator

Operator

Operator

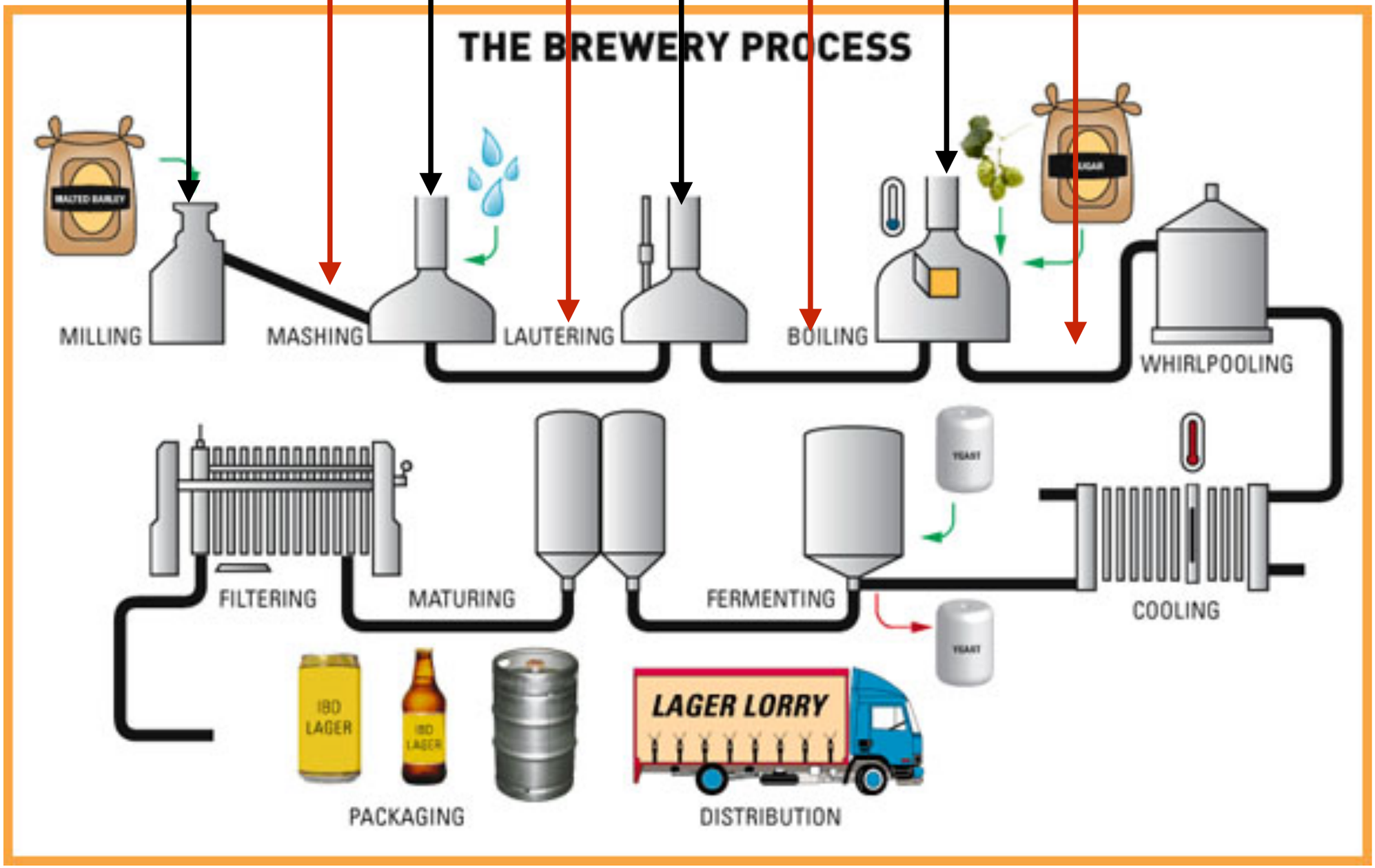
Observable

Observable

Observable

Observable

Observable



Error-handling operators

- **error()** is invoked by the Observable on the Observer.
- **catch()** - intercepts the error in the subscriber **before the observer gets it**. You can handle the error and re-subscribe.
- **retry(n)** - retry immediately up to n times
- **retryWhen(fn)** - retries as prescribed by the argument

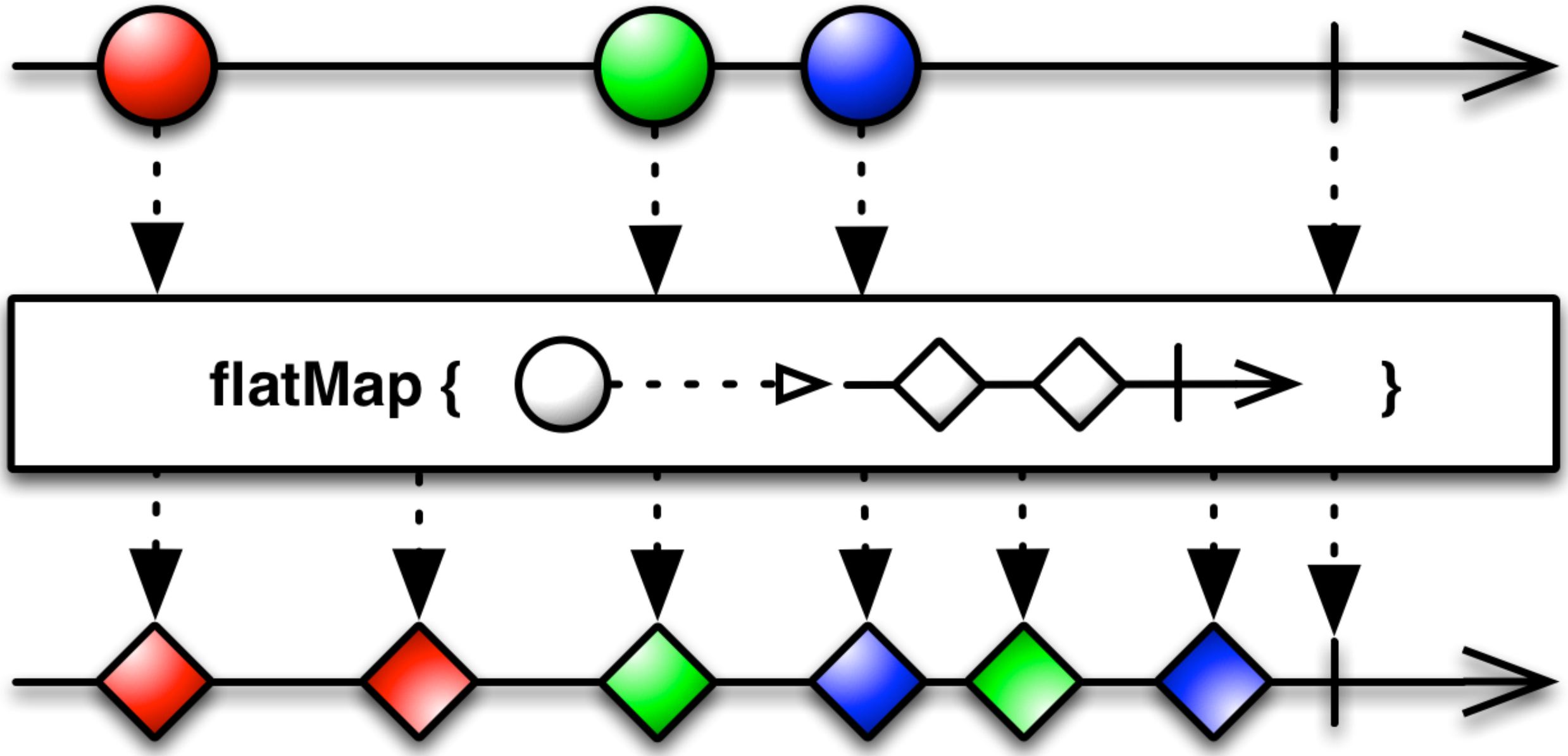
Failover with catch()

```
// Declaring
function getData(): Observable {…} // data source 1

function getCachedData(): Observable {…} // data source 2

function getDataFromAnotherService(): Observable {…} // data source 3

//Invoking and subscribing
getData()
    .catch(err => {

        if (err.status === 500){
            console.error("Switching to streaming cached beer data");
            return getCachedData();

        } else{
            console.error("Switching to another beer service");
            return getDataFromAnotherService();
        }

    })
    .map(beer => beer.name + ", " + beer.country)
    .subscribe(
        beer => console.log("Subscriber got " + beer)
    );
```

Failover

plunker: http://bit.ly/2jXY9ha

flatMap



Operator flatMap

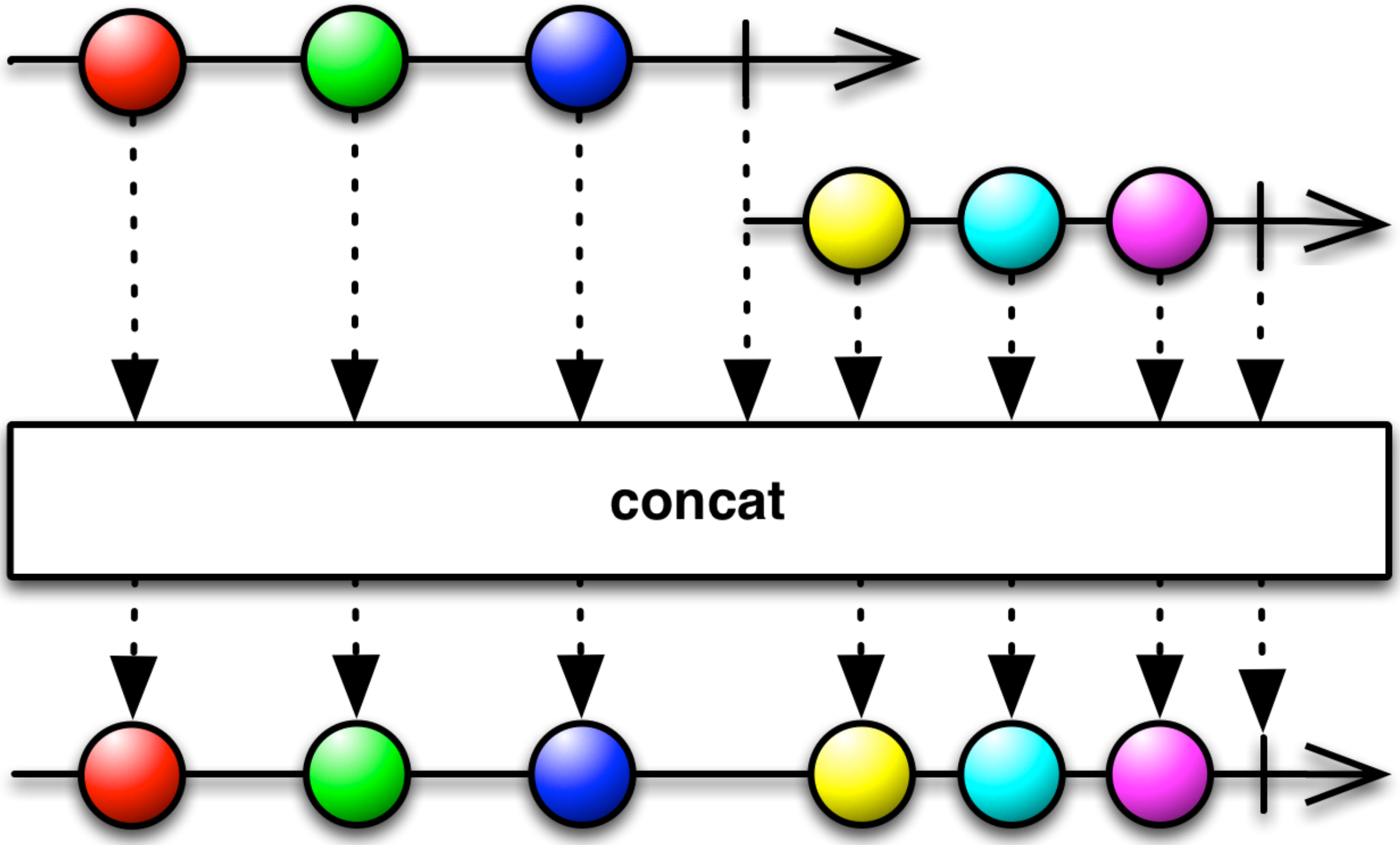
- Handles every value emitted by an observable as another observable

- Auto-subscribes to the internal observable and unwraps it

```
function getDrinks() {

    let beers = Rx.Observable.from([
        {name: "Stella", country: "Belgium", price: 9.50},
        {name: "Sam Adams", country: "USA", price: 8.50},
        {name: "Bud Light", country: "USA", price: 6.50}
    ]);

    let softDrinks = Rx.Observable.from([
        {name: "Coca Cola", country: "USA", price: 1.50},
        {name: "Fanta", country: "USA", price: 1.50},
        {name: "Lemonade", country: "France", price: 2.50}
    ]);

    return Rx.Observable.create( observer => {

            observer.next(beers);         // pushing the beer pallet (observable)
            observer.next(softDrinks);    // pushing the soft drinks pallet (observable)
        }
    );
}

getDrinks()
    .flatMap(drinks => drinks)            // unloading drinks from pallets
    .subscribe(
        drink => console.log("Subscriber got " + drink.name + ": " + drink.price )
    );
```

plunker http://bit.ly/2jZgc6T

concat



Operator concat

Subscribe to the next observable only when the previous completes. It's useful for a sequential processing, e.g. HTTP requests.

```
// Emulate HTTP requests
let fourSecHTTPRequest = Rx.Observable.timer(4000).mapTo('First response');

let oneSecHTTPRequest = Rx.Observable.timer(1000).mapTo('Second response');

Rx.Observable
   .concat(fourSecHTTPRequest, oneSecHTTPRequest)
   .subscribe(res => console.log(res));
```

plunker http://bit.ly/2keEoiI

RxJS operators

- bindCallback
- bindNodeCallback
- create
- defer
- empty
- from
- fromEvent
- fromEventPattern
- fromPromise
- generate
- interval
- never
- of
- repeat
- repeatWhen
- range
- throw
- timer

- catch
- retry
- retryWhen

- count
- max
- min
- reduce

- buffer
- bufferCount
- bufferTime
- bufferToggle
- bufferWhen
- concatMap
- concatMapTo
- exhaustMap
- expand
- groupBy
- map
- mapTo
- mergeMap
- mergeMapTo
- mergeScan
- pairwise
- partition
- pluck
- scan
- switchMap
- switchMapTo
- window
- windowCount
- windowTime
- windowToggle
- windowWhen

- debounce
- debounceTime
- distinct
- distinctKey
- distinctUntilChanged
- distinctUntilKeyChanged
- elementAt
- filter
- first
- ignoreElements
- audit
- auditTime
- last
- sample
- sampleTime
- single
- skip
- skipUntil
- skipWhile
- take
- takeLast
- takeUntil
- takeWhile
- throttle
- throttleTime

- cache
- multicast
- publish
- publishBehavior
- publishLast
- publishReplay
- share

- combineAll
- combineLatest
- concat
- concatAll
- exhaust
- forkJoin
- merge
- mergeAll
- race
- startWith
- switch
- withLatestFrom
- zip
- zipAll

- do
- delay
- delayWhen
- dematerialize
- finally
- let
- materialize
- observeOn
- subscribeOn
- timeInterval
- timestamp
- timeout
- timeoutWith
- toArray
- toPromise

- defaultIfEmpty
- every
- find
- findIndex
- isEmpty

http://reactivex.io/rxjs/manual/overview.html#categories-of-operators

Observables in Angular

Code samples: https://github.com/yfain/observables

Observables in Forms

An input field: FormControl

- `valueChanges` - the value of the form control changes

```
this.searchInput.valueChanges.subscribe(…);
```

- `statusChanges` - status of the form control (valid/invalid)

```
this.password.statusChanges.subscribe(…);
```

Observable Events in Angular forms

```
@Component({
    selector: "app",
    template: `
      <h2>Observable events demo</h2>
      <input type="text" placeholder="Enter stock" [formControl]="searchInput">
    `
})
class AppComponent {

    searchInput: FormControl;

    constructor(){
        this.searchInput = new FormControl('');

        this.searchInput.valueChanges
            .debounceTime(500)
            .subscribe(stock => this.getStockQuoteFromServer(stock));
    }

    getStockQuoteFromServer(stock) {

        console.log(`The price of ${stock} is ${100*Math.random().toFixed(4)}`);
    }
}
```

Observable

Demo

main-formcontrol

Http and Observables

```
class AppComponent {

  products: Array<string> = [];

  constructor(private http: Http) {

    this.http.get('/products')
        .map(res => res.json())
        .subscribe(
          data => {
            this.products=data;
          },
          err =>
            console.log("Can't get products. Error code: %s, URL: %s ",
                                                  err.status, err.url),
          () => console.log('Product(s) are retrieved')
        );
  }
}
```

Observer

async pipe

```
@Component({
  selector: 'http-client',
  template: `<h1>All Products</h1>
  <ul>
    <li *ngFor="let product of products | async">
       {{product.title}}
    </li>
  </ul>
  <h2>{{errorMessage}}</h2>
  `})
class AppComponent {

  products: Observable<Array<string>>;
  errorMessage: string;

  constructor(private http: Http) {

    this.products = this.http.get('/products')
        .map(res => res.json())
        .catch( err => {
          this.errorMessage =`Can't get product details from ${err.url},
                                                  error ${err.status} `;
          return Observable.empty();
        });
  }
}
```

The switchMap operator

RxJS 5, official doc:

http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap

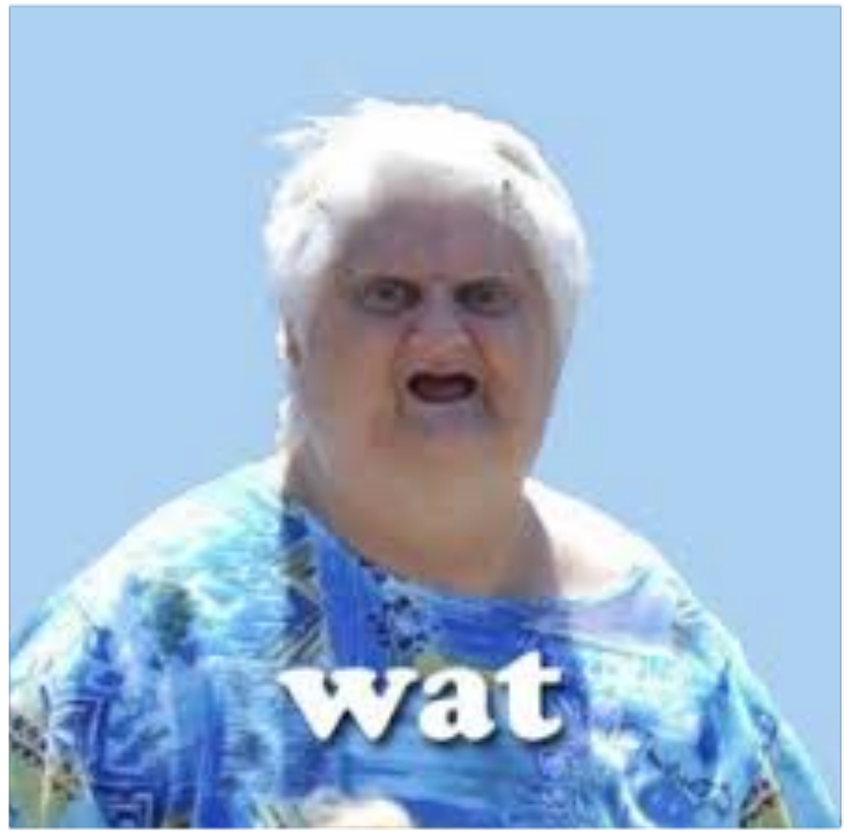
Returns an Observable that emits items based on applying a function that you supply to each item emitted by the source Observable, where that function returns an (so-called "inner") Observable. Each time it observes one of these inner Observables, the output Observable begins emitting the items emitted by that inner Observable. When a new inner Observable is emitted, switchMap stops emitting items from the earlier-emitted inner Observable and begins emitting items from the new one. It continues to behave like this for subsequent inner Observables.

The switchMap operator

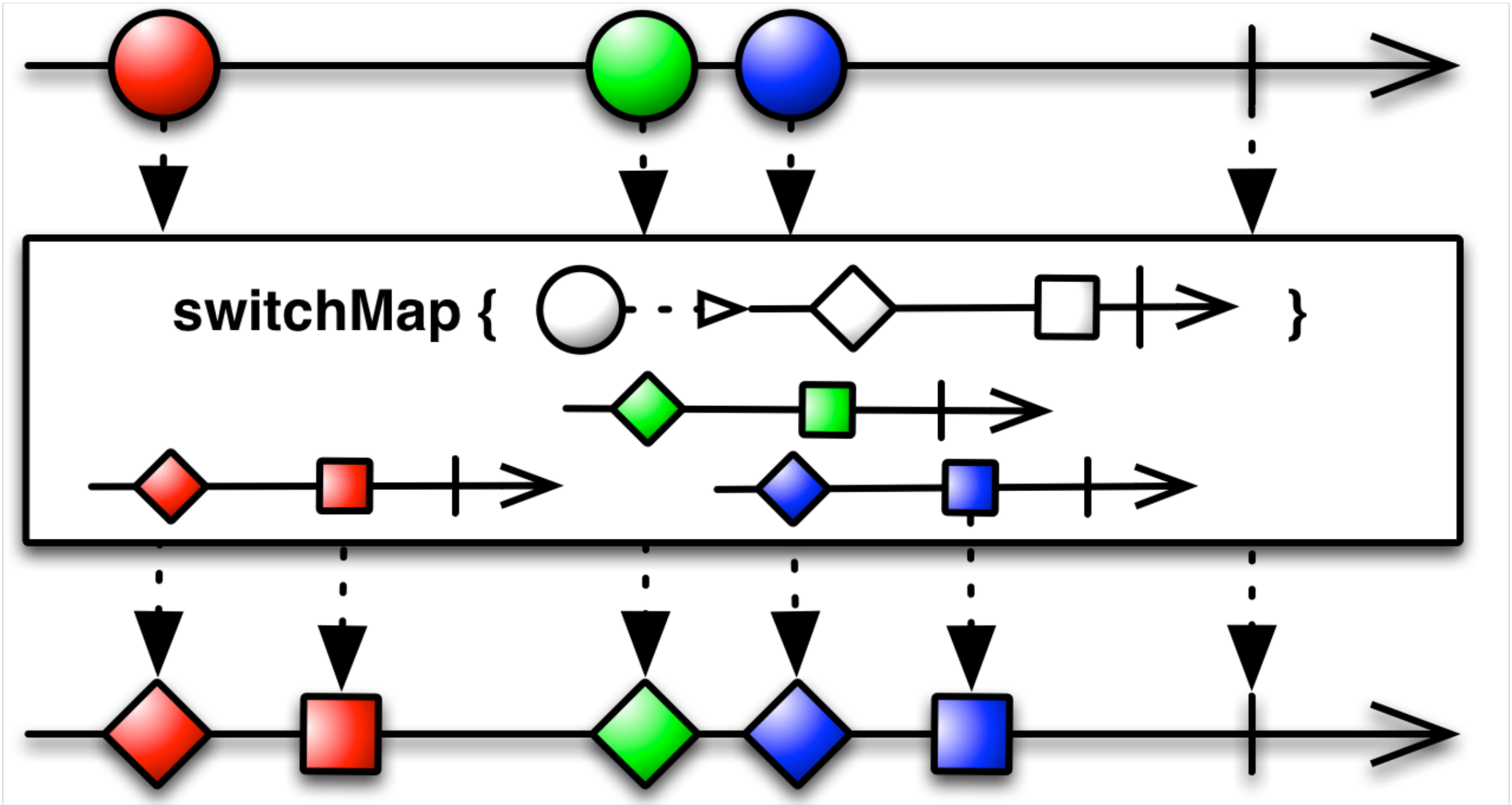
RxJS 5, official doc:

http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap

Returns an Observable that emits items based on applying a function that you supply to each item emitted by the source Observable, where that function returns an (so-called "inner") Observable. Each time it observes one of these inner Observables, the output Observable begins emitting the items emitted by that inner Observable. When a new inner Observable is emitted, switchMap stops emitting items from the earlier-emitted inner Observable and begins emitting items from the new one. It continues to behave like this for subsequent inner Observables.



When a picture worth a thousand words



http://reactivex.io/documentation/operators/images/switchMap.png

The switchMap operator in English

- An outer observable emits the data and switches over to the inner observable for processing

- If the outer observable emits the new data while the inner one is still processing, the inner observable is terminated

- The inner observable starts processing the newly emitted data

- **Example:** A user types in a field (outer observable) and the HTTP requests are being made (inner observable) as the user types

Killing HTTP requests with switchMap

```
<input type="text" placeholder="Enter city" [formControl]="searchInput">

…

this.searchInput.valueChanges
        .debounceTime(200)
        .switchMap(city => this.getWeather(city))
        .subscribe(
            res => {
                this.temperature =
                    `Current temperature is  ${res.main.temp}F, ` +
                    `humidity: ${res.main.humidity}%`;
            },
            err => console.log(`Can't get weather. Error code: %s, URL: %s`, err.message, err.url)
        );
}

getWeather(city): Observable<Array> {
  return this.http.get(this.baseWeatherURL + city + this.urlSuffix)
    .map(res => res.json());
}
```

Outer Obs.

Inner Obs.

Demo

main-http

Observables in the Router

Receiving params in ActivatedRoute

- Inject `ActivatedRoute` into a component to receive route params during navigation

- Use `ActivatedRoute.snapshot` to get params once

- Use `ActivatedRoute.param.subscribe()` for receiving multiple params over time

Subject: Observable + Observer

Can emit values and allows to subscribe to them

```
@Component({
  selector: "app-root",
  template: `
       <h3>Using Subject for emitting/subscribing to keyup and input events</h3>
      <input type="text" placeholder="Start typing"
            (input)="mySubject.next($event)" (keyup)="myKeySubject.next($event)">
    `
})
export class AppComponent {

  mySubject: Observable<Event> = new Subject();              // Observable for any events

  myKeySubject: Observable<KeyboardEvent> = new Subject(); // Observable for keyboard events

  constructor(){

    this.myKeySubject.subscribe(({type, key}) => console.log(`Event: ${type} key: ${key}`));

    this.mySubject.subscribe(({type, target}) => console.log(
                  `Event: ${type} value:  ${(<HTMLInputElement>target).value}`));
  }
}
```

Sharing an Observable

```
@Component({
  selector: "app-root",
  template: `
       <h3>Sharing Observable between subscribers to keyup and input events</h3>
      <input type="text" placeholder="Start typing"
            (input)="mySubject.next($event)"
            (keyup)="mySubject.next($event)">

      <br> Subscriber to input events got {{inputValue}}
      <p>
      <br> Subscriber to input events got {{keyValue}}
    `
})
export class AppComponent {

  keyValue: string;
  inputValue: string;

  mySubject: Observable<Event> = new Subject().share(); // Single Observable for any events

  constructor(){

    // Subscriber 1
    this.mySubject
      .filter(({type}) => type==="keyup")
      .map(e => (<KeyboardEvent>e).key)
      .subscribe((value) => this.keyValue=value);

    // Subscriber 2
    this.mySubject
      .filter(({type}) => type==="input")
      .map(e => (<HTMLInputElement>e.target).value)
      .subscribe((value) => this.inputValue=value);
  }
}
```

Accessing native elements with ElementRef

```
@Component({
    selector: "app",
    template: `
       <h2>Sharing the same stream</h2>
      <input #myinput type="text" placeholder="Start typing" >

      <br> Subscribing to each value: {{data1}}
      <br> Subscribing to 3-second samples: {{data2}}
    `})
class AppComponent {

    @ViewChild('myinput') myInputField: ElementRef;

    data1: string;
    data2: string;

    ngAfterViewInit(){

        let keyup$: Observable = Observable.fromEvent(this.myInputField.nativeElement, 'keyup');

        let keyupValue$ = keyup$
            .map(event => event.target.value)
            .share();

        // Subscribe to each keyup
        keyupValue$
            .subscribe(value => this.data1 = value);

        // Subscribe to 3-second samples
        keyupValue$
            .sample(Observable.interval(3000))
            .subscribe(value => this.data2 = value);
    }
}
```

Using ElementRef
is not recommended

Demo

main-subject
main-subject-shared

Subscribing to EventEmitter

Angular:

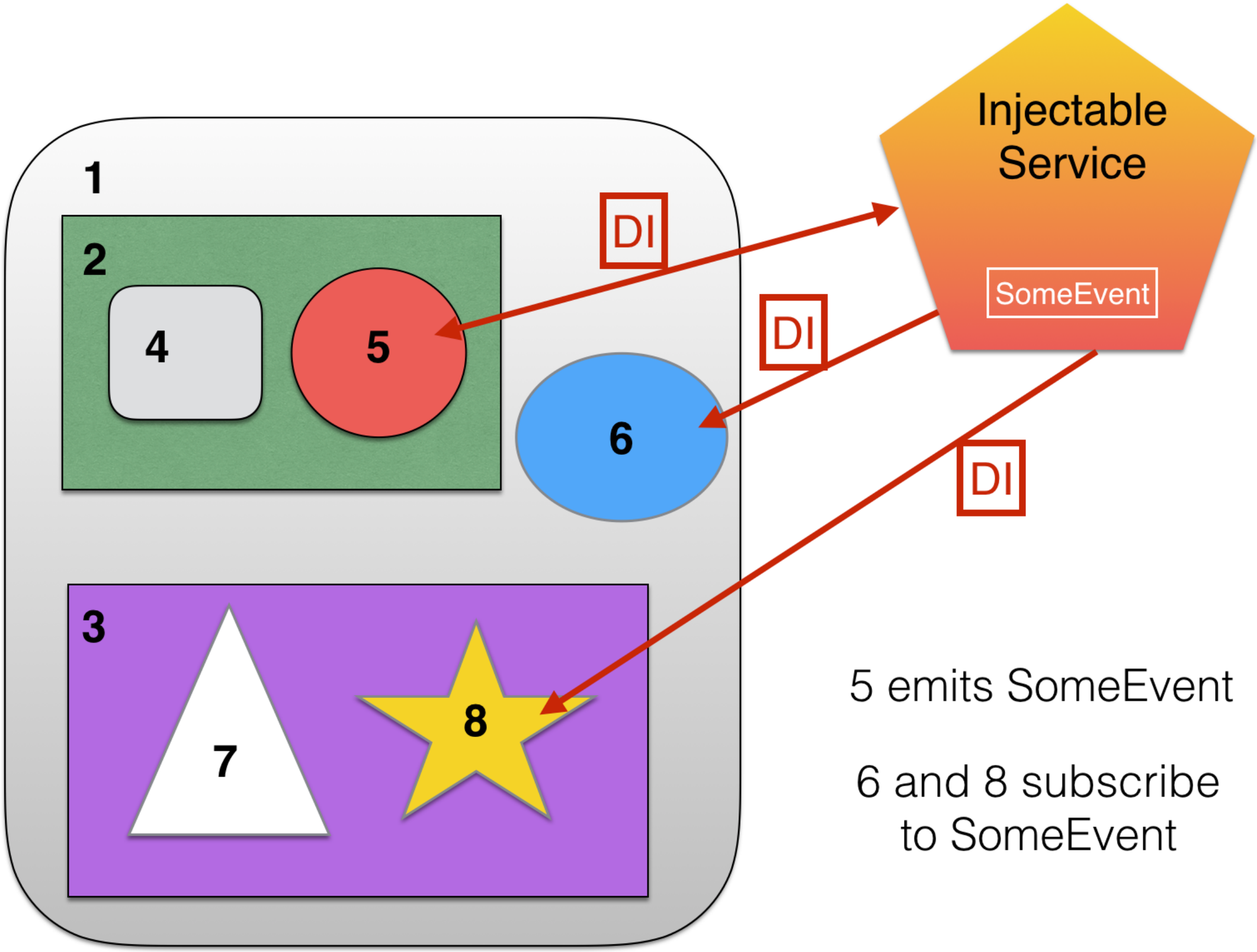
```
export declare class EventEmitter<T> extends Subject<T> {}
```

Has Observer and Observable

Your app:

```
myEvent: EventEmitter<string> = new EventEmitter();

myEvent.emit("Hello World");

…

myEvent.subscribe(event => console.log(" Received " + event);
```

Injectable service as Mediator



5 emits SomeEvent

6 and 8 subscribe
to SomeEvent

Subscribing to EventEmitter

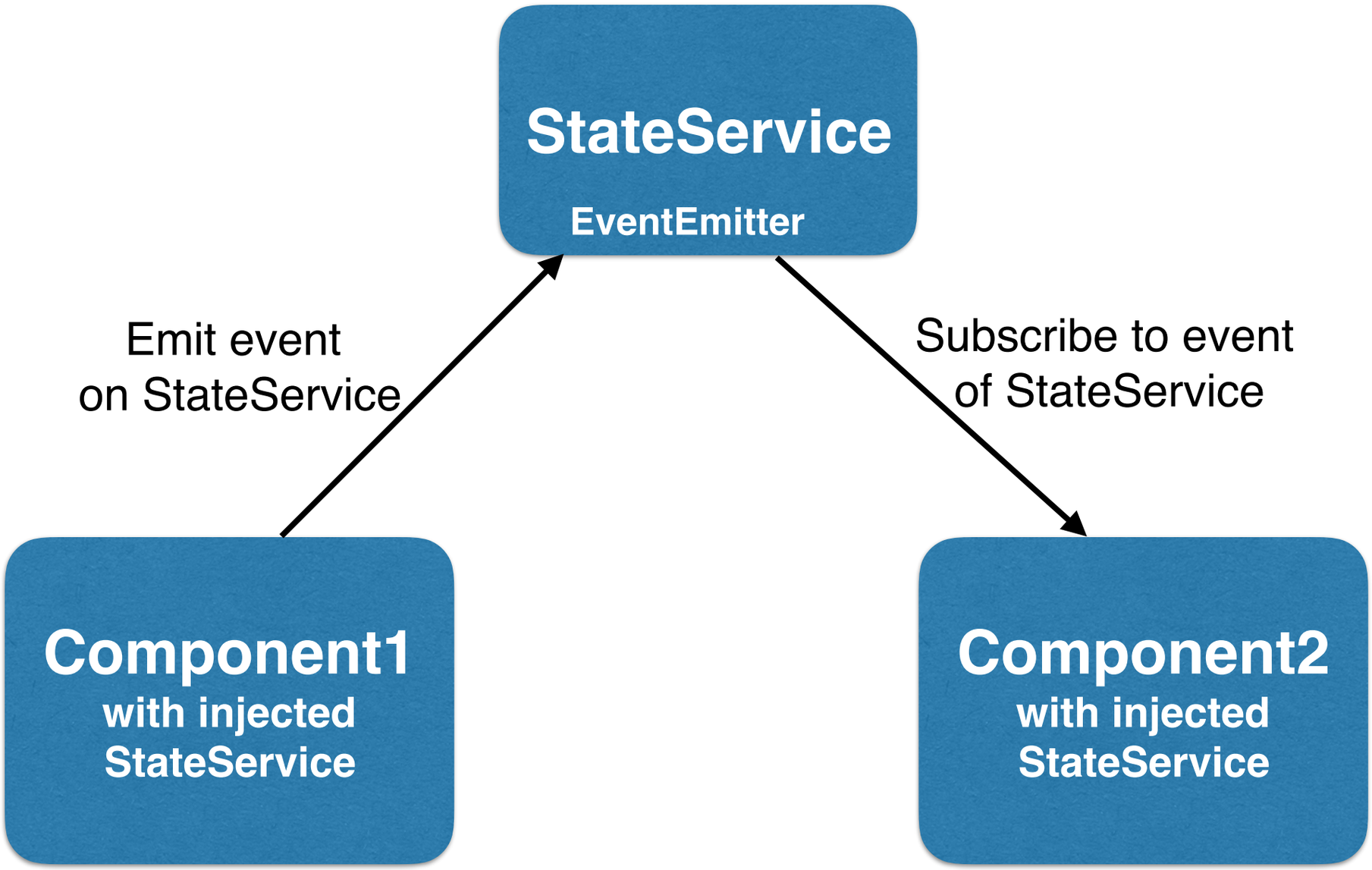
Angular:

```
export declare class EventEmitter<T> extends Subject<T> {}
```

Your app:

```
myEvent: EventEmitter<string> = new EventEmitter();

myEvent.emit("Hello World");

…

myEvent.subscribe(event => console.log(" Received " + event);
```

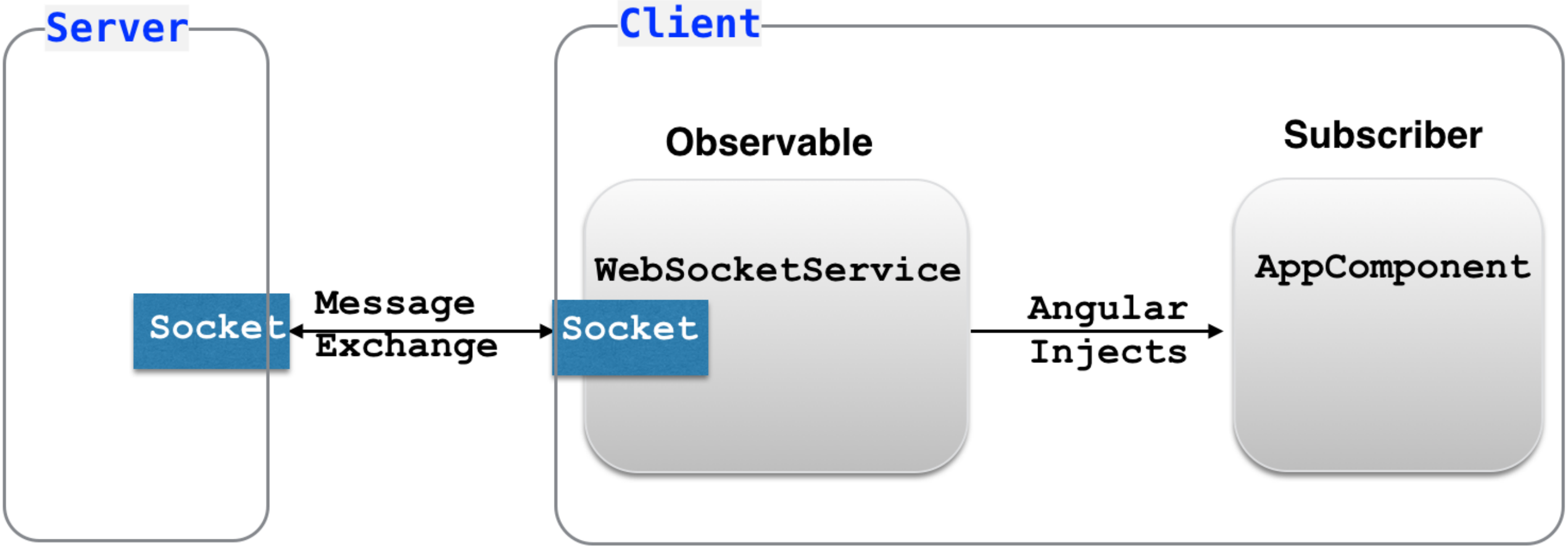
Mediator, DI, Events, and Observables



Demo

main-mediator-service

WebSockets and Observables



Wrapping WebSocket into Observable

```
import {Observable } from 'rxjs/Observable';

export class BidService{

    ws: WebSocket;

    createObservableSocket(url:string): Observable{

        this.ws = new WebSocket(url);

        return new Observable(
                        observer => {

                          this.ws.onmessage = (event) => observer.next(event.data);

                          this.ws.onerror   = (event) => observer.error(event);

                           this.ws.onclose  = (event) => observer.complete();
                        });
    }

}
```

Subscribing to WebSocket's messages

```
@Component({ … })
class BidComponent {
  newBid: Bid;

  constructor(private wsService: BidService) {

      this.wsService.createObservableSocket("ws://localhost:8085")
        .map(res => JSON.parse(res))
        .subscribe(
            data => {

              this.newBid = data;
              this.newBid.bidTime= Date.parse(data.bidTime);
              console.log(this.newBid);
            },
            err => console.log( err),
            () =>  console.log( 'The bid stream is complete')
        );
  }

}
```

Demo

1. Open http_websocket_samples

2. systemjs.config: bids/bid-component.ts

3. npm run tsc

4.npm run bidServer

5.http://localhost:8000

Summary

- Everything is an observable
- No data is pushed to you until you subscribe
- Chain the operators to pre-process the observable data before it gets to the subscriber
- Angular offers you ready-to-use observables in multiple components and services
- You can wrap the data pushed to your app into an Observable

Thank you!

- Code samples:
  https://github.com/yfain/observables
- Our company:
  faratasystems.com
- My blog:
  yakovfain.com
- Twitter: @yfain