# Angular: RxJS for Reactive Programming

## Reactive Programming with RxJS and Signals

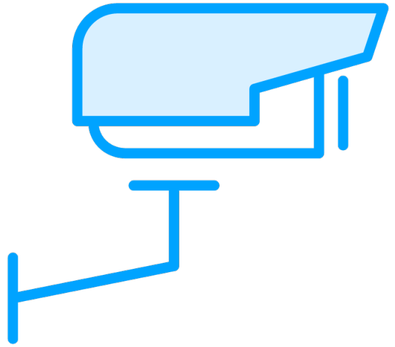

**Deborah Kurata**

Developer | Content Creator | MVP | GDE

@deborahkurata | https://www.youtube.com/@deborah_kurata

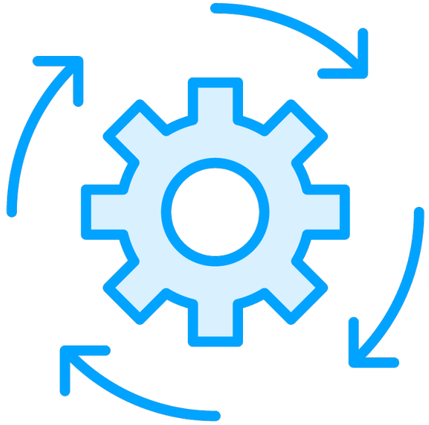
# Reactive Programming

**How do RxJS and signals help us with reactive programming?**



**Observe** events (actions) and state (data)

Sign up to receive **notifications** when event occurs or state changes



**React** to the notification

Applications are more **responsive** and provide a better user experience

# Reactive Programming


**Signals**
Watch and react to data changes

**RxJS**
Watch and react to events


**RxJS**
Issue HTTP requests, react, and process responses

# Course Introduction

**Why RxJS is still important in Angular**

**Observables vs. signals**

**RxJS terms, concepts, and syntax**

**Use Cases:**

- Manage a stream of keyboard events
- Compose data using observables and `rxResource()`
- Implement a search feature using debouncing and transformations

**Integrate observables into an Angular application**

# Reactive Extensions for JavaScript (RxJS)

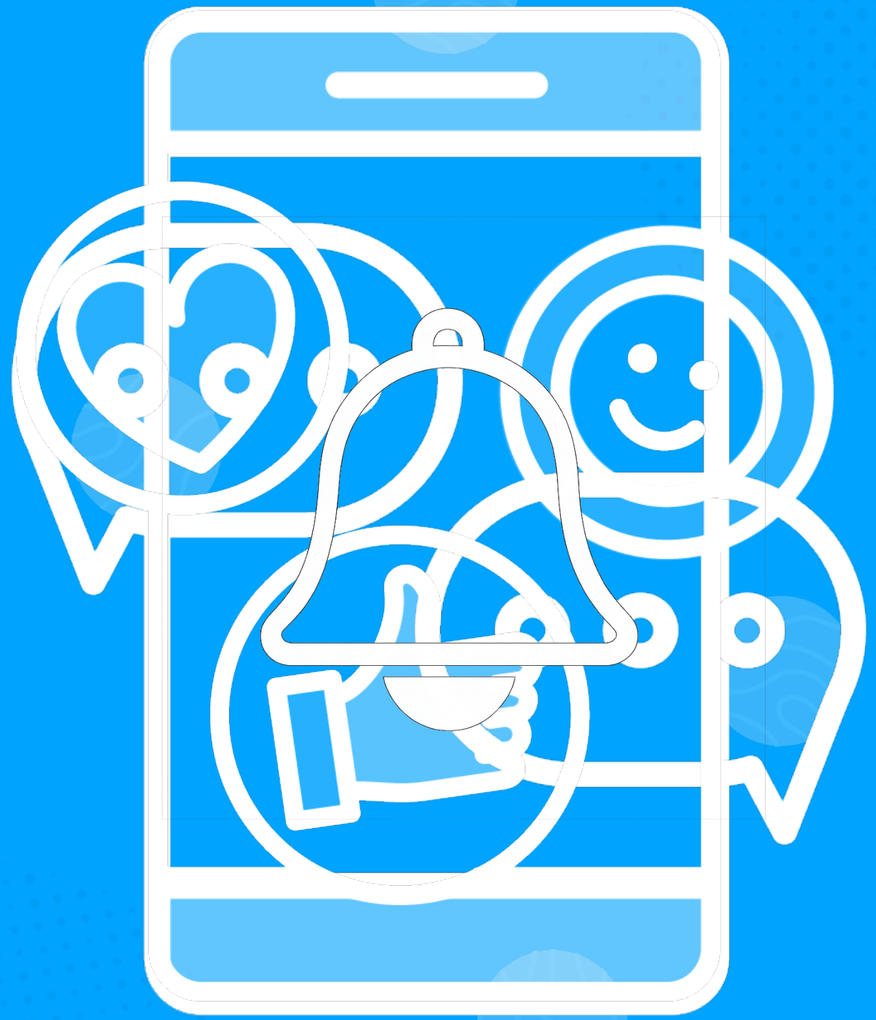
Popular library for reactive programming

Observe events and data streams over time

Subscribe to receive a notification

React to the notification

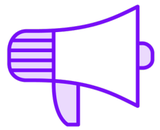
# Observable



Signal =

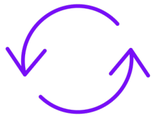
data value

# Signals



Provide a notification when data changes

Help us write more reactive code



Improve change detection

16+

Available since version 16

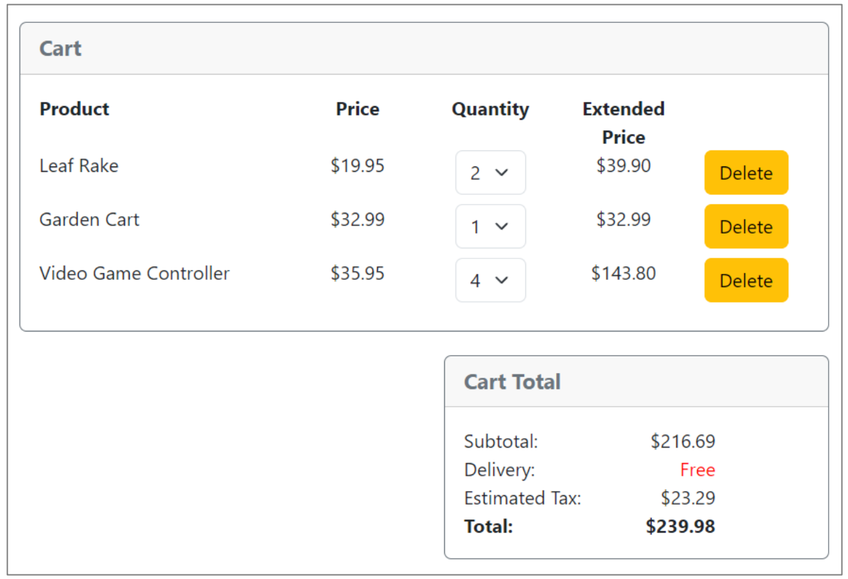
# Reactive Programming

## Signals

**Observe state** (variable values)

**Notified** when that state changes

**React** to those notifications

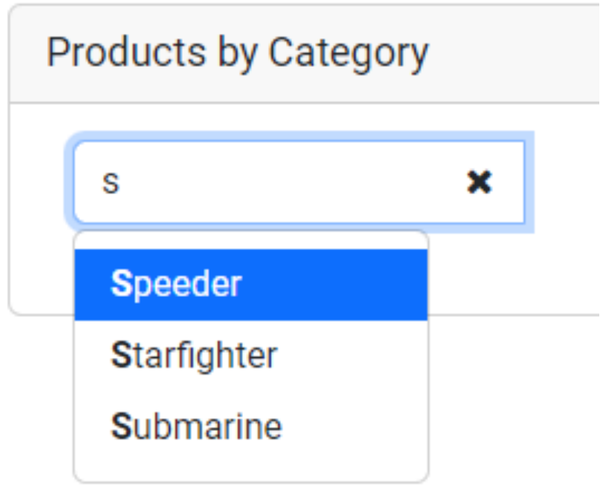



## RxJS Observables

**Observe events or data that arrive over time**

**Notified** when the event occurs, or data arrives

**React** to those notifications



# RxJS + Signals: Better Together


Handling asynchronous operations such as HTTP requests

Retrieving and merging related sets of data

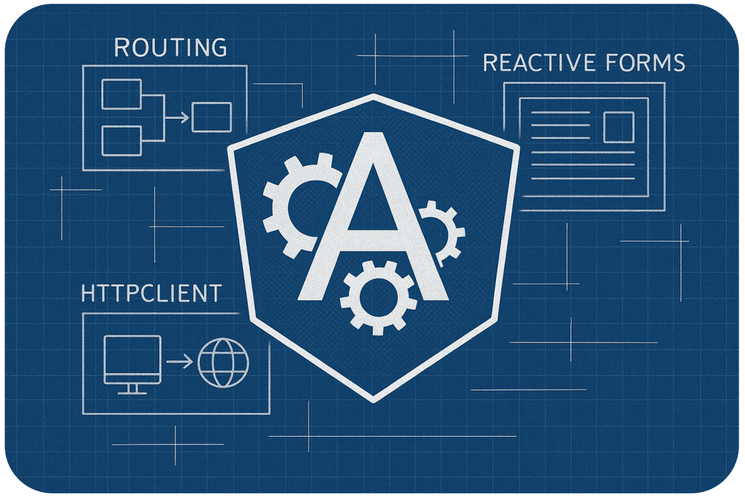
Managing data after it's been retrieved + computed signals

Binding in the template for better change detection

With signals and `httpResource()` ...

# Is there still a place for RxJS in an Angular app?

# RxJS Plays an Important Role in Angular




Built into several core features

Handles events and asynchronous streams of data

Operators provide powerful data composition and transformation

# RxJS Is Integrated into Core Features


**Routing events**

```
this.route.paramMap
this.route.data
this.router.events
```

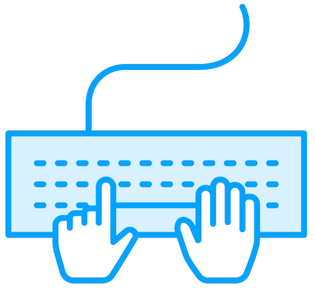
**Reactive forms events**

```
this.productForm.valueChanges
```


**HttpClient response notifications**

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.url);
}
```

# RxJS Handles Events + Async Data Streams



Source

'r'

Observable

Subscriber

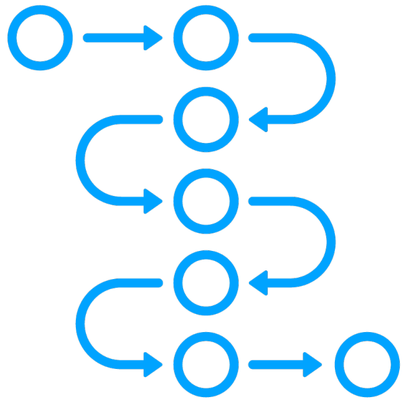
Component

**Subscribes to receive notifications and reacts**

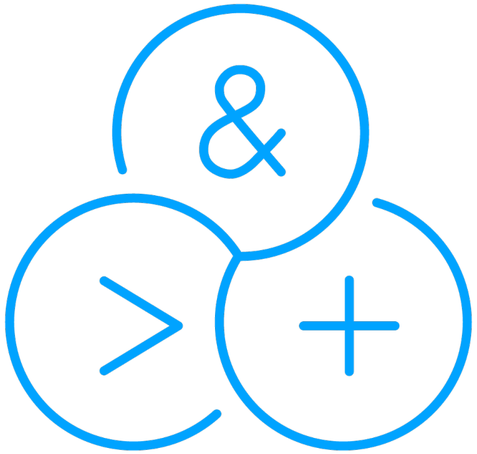
Subscription

'R'

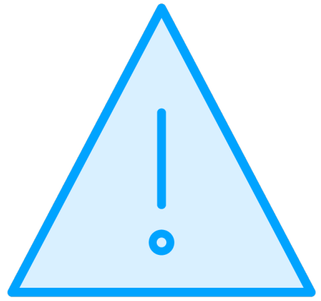
# RxJS Operators Provide Powerful Features



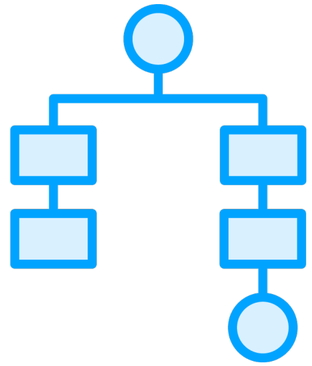
Work with the data before it is emitted




100+ operators



Provide error handling



Handle multiple complex data sets

# Observe and React to Events through Time

**Start:** `subscribe()`

**As an observer**

- Next item, process it
- Error occurred, stop and handle it
- Complete, you're done

**Item passes through a set of operations**

- Transform
- Filter
- Modify
- …

**Stop:** `unsubscribe()`

# Observable, Observer, Subscription

```
this.sub = this.apples$.subscribe({
  next: apple => console.log(`Value emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples`)
});
```

```
this.sub.unsubscribe();
```

Observables + signals are better together

# When should you use which?

# Use Signals for Managing State

```
<select [(ngModel)]="selectedProduct">…</select>

<div> {{ selectedProduct().price }} </div>

<input type='number' [(ngModel)]='quantity'>

<div> {{ total() }} </div>
```

```
selectedProduct = signal<Product | undefined>(undefined);
quantity = signal(1);

total = computed(() =>
     this.selectedProduct().price * this.quantity());
```

# Use Signals for Managing State

```
selectedProduct = signal<Product | undefined>(undefined);
quantity = signal(1);

total = computed(() =>
     this.selectedProduct().price * this.quantity());
```

**Hold state**

**Improve change detection**

**Define derived values**

# An Observable Does Not Hold State

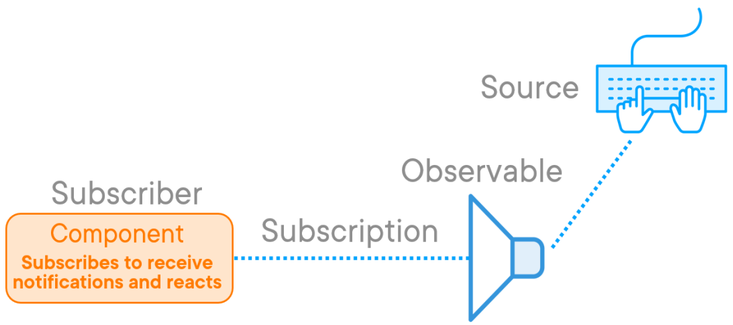
```
this.sub = this.apples$.subscribe({
  next: apple => console.log(`Value emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples`)
});
```

Has no value until a value is provided from the source

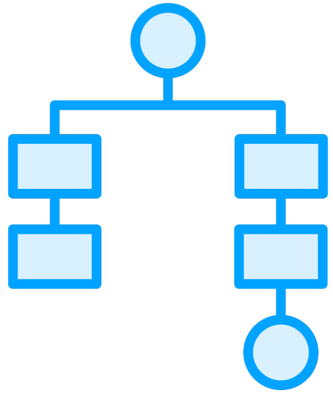
Emits that value

Once emitted, the value is gone

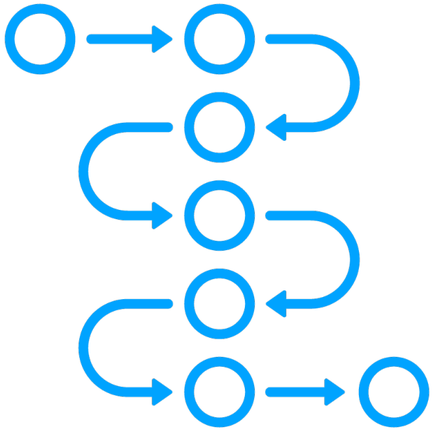
# Use an Observable to...




**Listen for and react to events**



**Compose complex data streams**



**Operate on emitted values**

# When to Use a Signal vs. an Observable?

When working with state, a signal is great!

For an event-driven stream, an observable is keen!