Component Communication

Dan Wahlin
WAHLIN CONSULTING

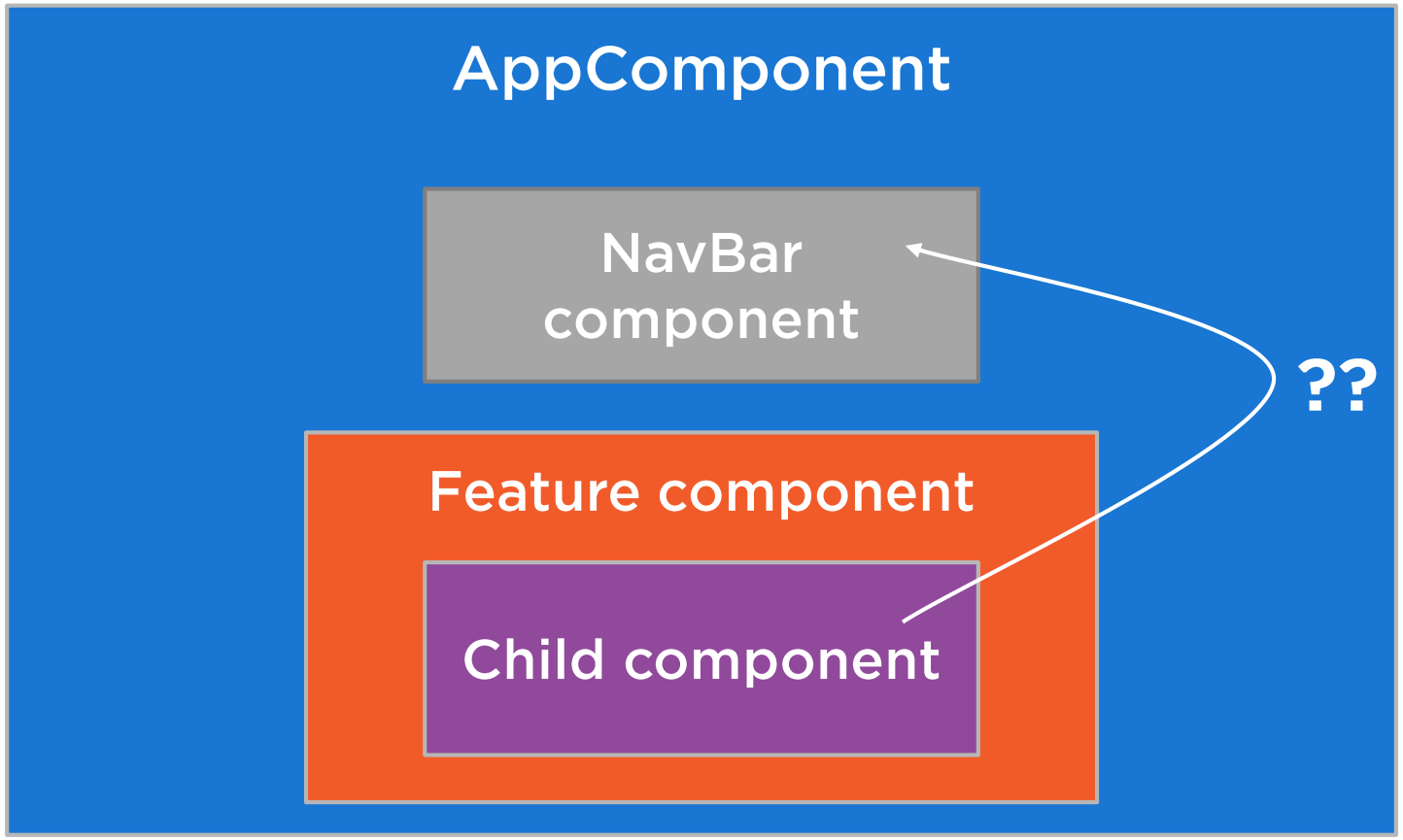
@danwahlin www.codewithdan.com

Module Overview

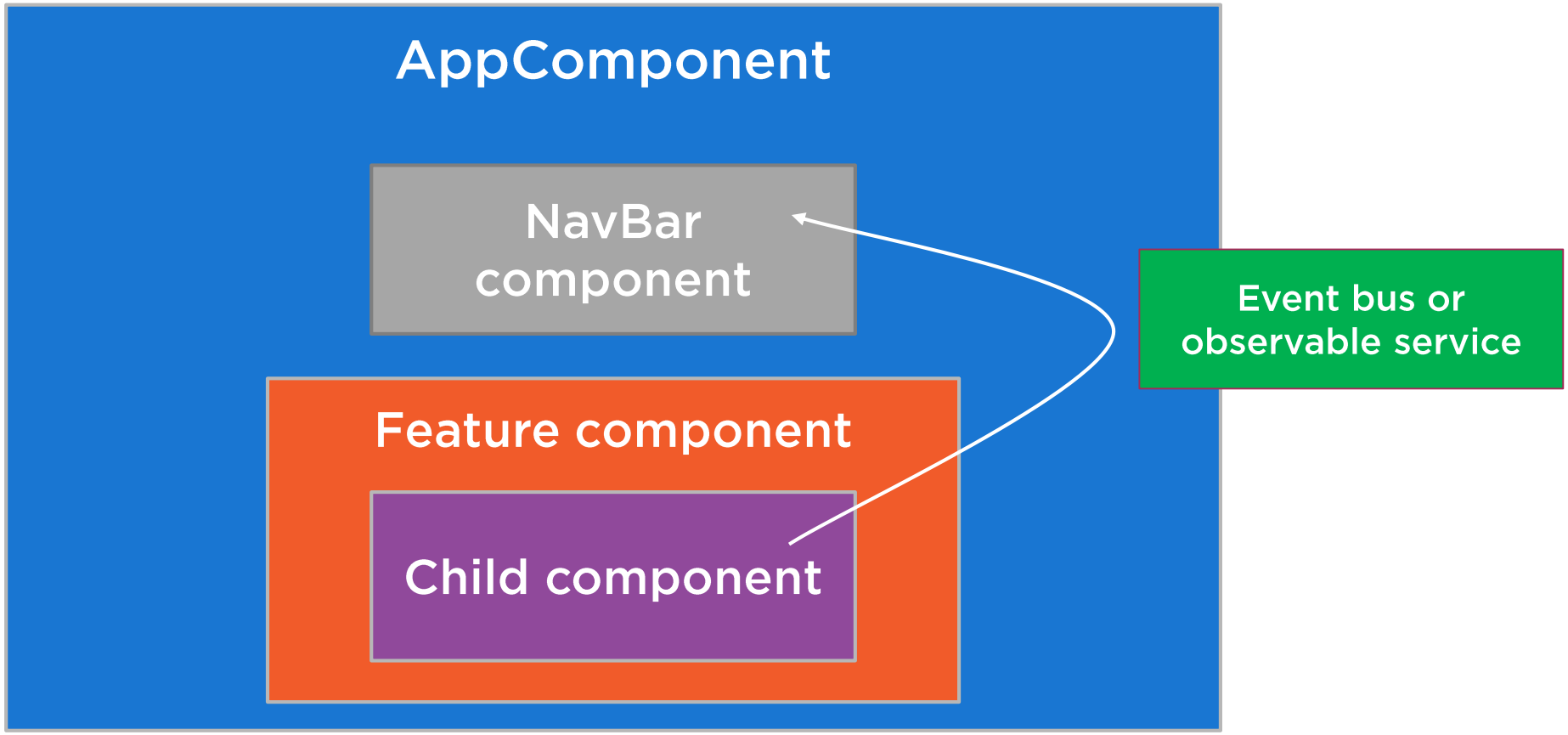
- Component communication
- Understanding RxJS subjects
- Creating and using an event bus service
- Creating and using an observable service
- Unsubscribing from observables

Component Communication

The Need for Component Communication



Component Communication Options



Event Bus vs. Observable Service

Event Bus	Observable Service
Mediator pattern	Observer pattern
Angular service acts as the middleman between components	Angular service exposes observable directly to components
Components don't know where data is coming from by default	Components know where data is coming from
Loosely coupled	Not as loosely coupled as event bus
Relies on subject/observable	Relies on subject/observable

Understanding RxJS Subjects

RxJS Subjects

Subject

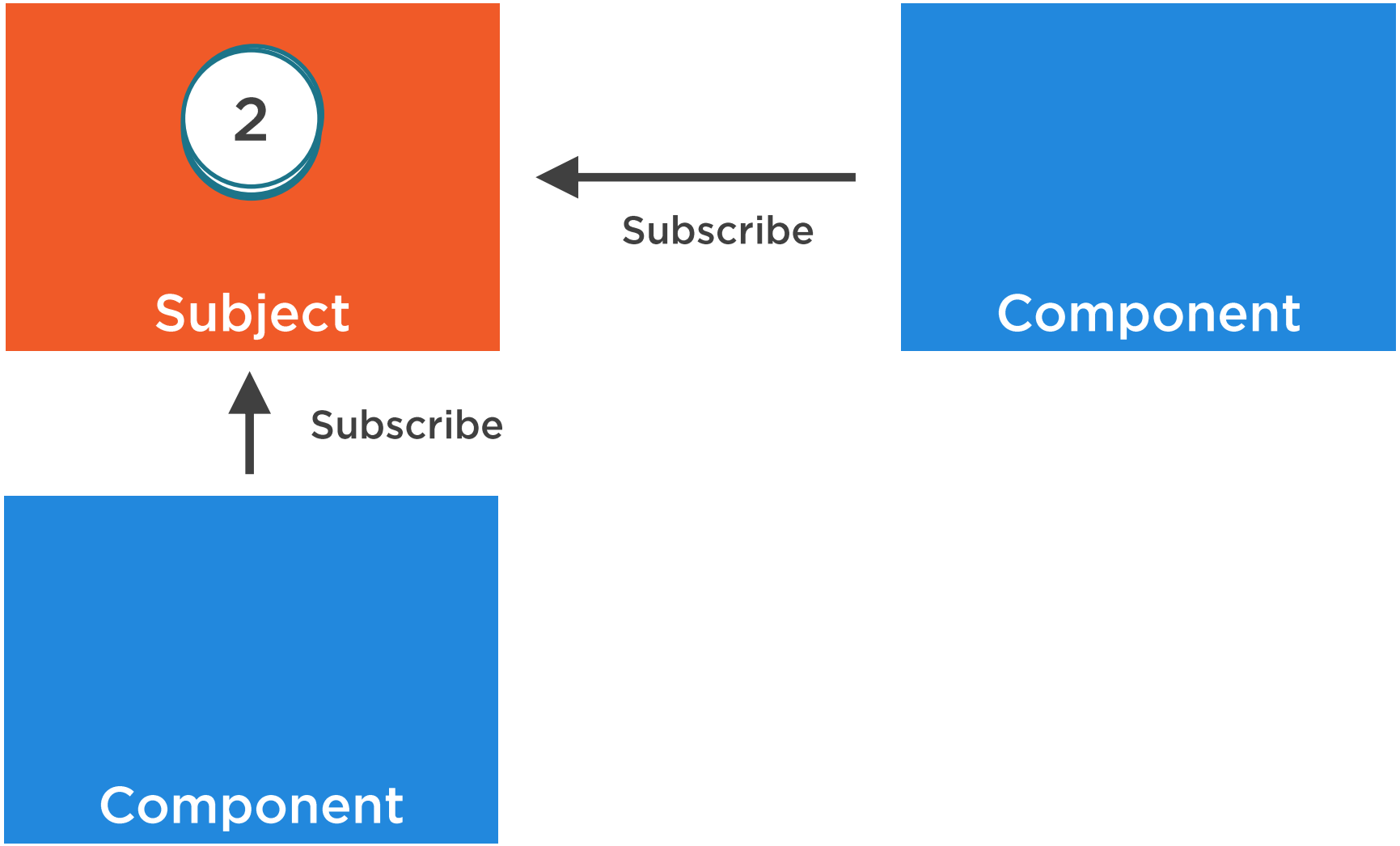
BehaviorSubject

ReplaySubject

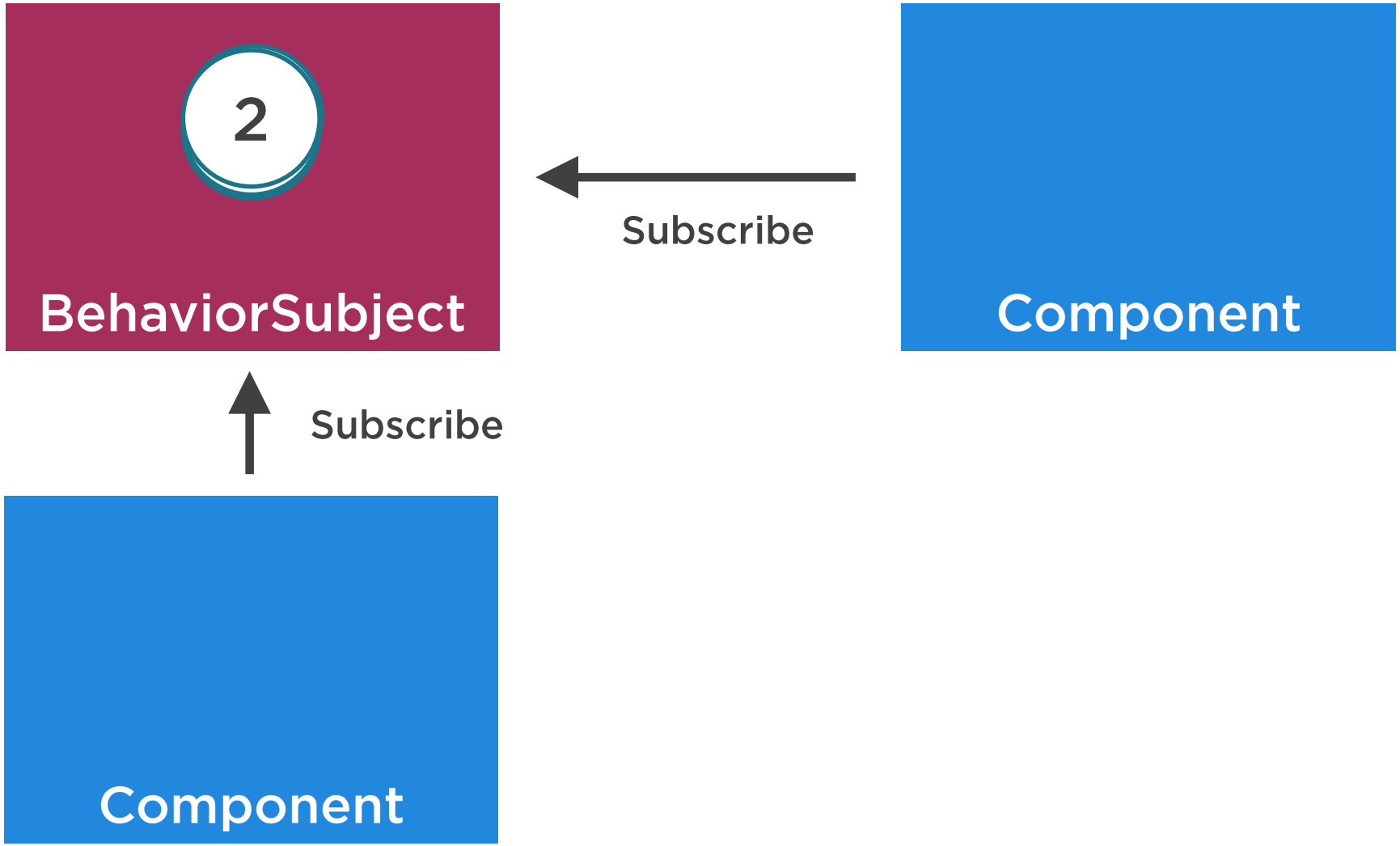
AsyncSubject

http://reactivex.io/documentation/subject.html

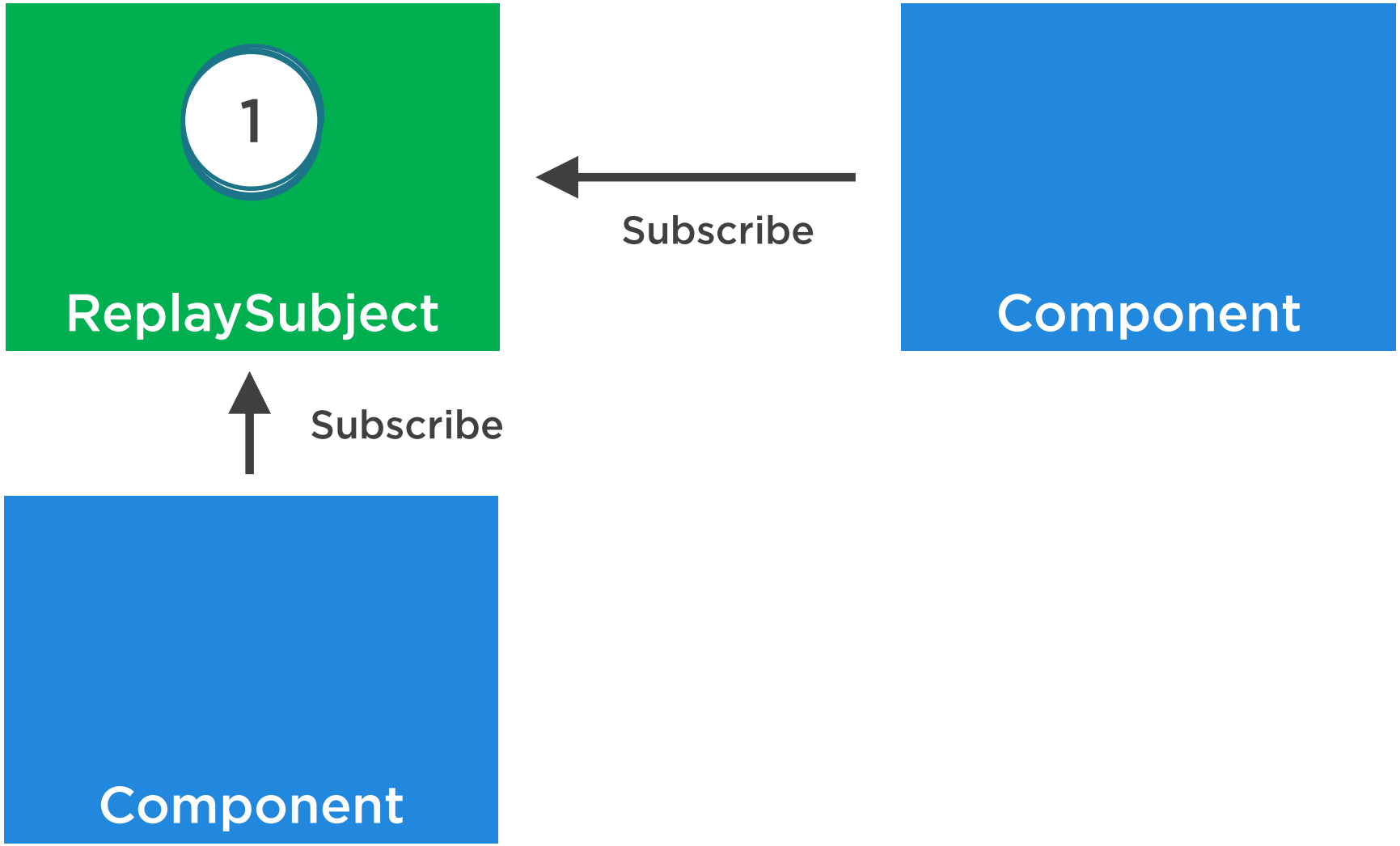
Using Subject



Using BehaviorSubject



Using ReplaySubject



Using AsyncSubject

Send data to subscribed observers. Any previously emitted data is not sent to new observers.

Subject

Send last data value to new observers.

BehaviorSubject

All previously sent data can (optionally) be “replayed” to new observers.

ReplaySubject

Emits the last value (and only the last value) to observers when the sequence is completed.

AsyncSubject

RxJS Subjects in Action – Part 1

RxJS Subjects in Action – Part 2

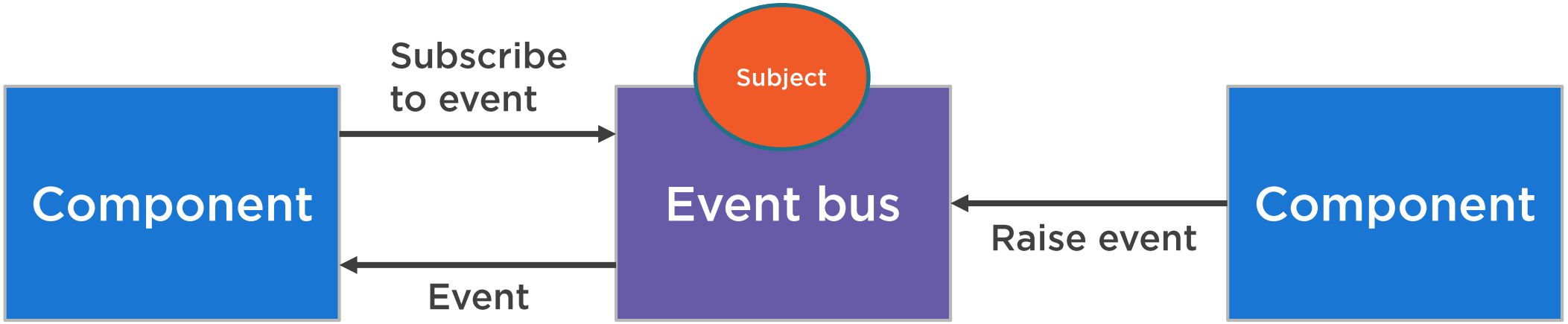
Creating an Event Bus Service

Event Bus

Event bus can send data between multiple components

Follows the mediator pattern

Uses RxJS Subject



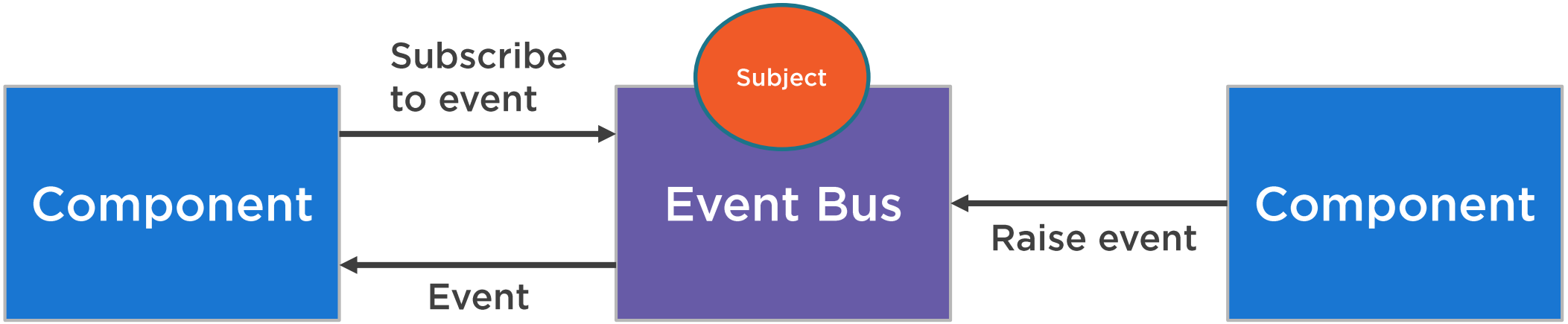
Using an Event Bus Service

Event Bus

Event bus can send data between multiple components

Follows the mediator pattern

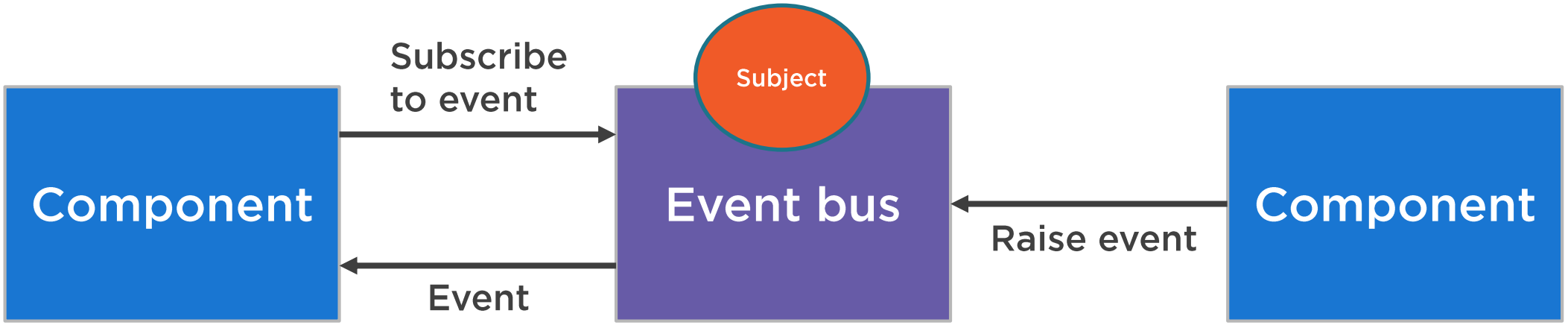
Uses RxJS Subject



Event Bus Pros and Cons

Pros	Cons
Simple to use - call emit() or on()	Who is triggering the event?
Loosely coupled communication	Loosely coupled events can make maintenance more challenging (due to above)
Lightweight	Must remember to unsubscribe

Event Bus Communication



Creating an Observable Service

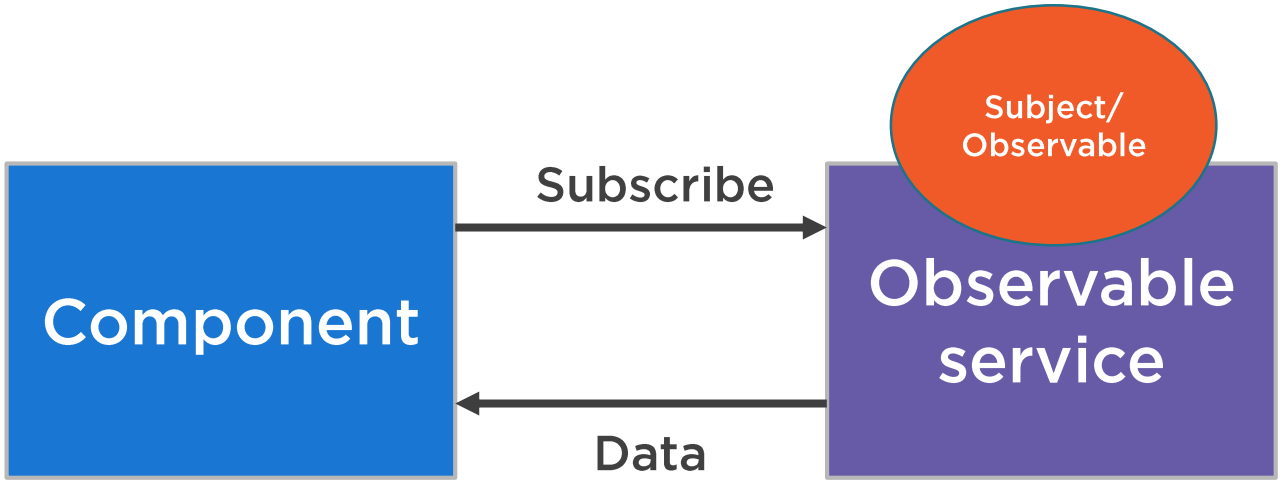
Observable Services

Observable services can send data to observers/subscribers

Follows the observer pattern

Provides a simple way to keep multiple observers (components, services) up-to-date

Service can use RxJS Subject objects and observables



Using an Observable Service

Observable Service Pros and Cons

Pros	Cons
Simple to use – subscribe/unsubscribe	Not as loosely coupled as an event bus (coupling between observable and observer)
Data source is known (simplifies maintenance)	Subject variations can be challenging to master
Easy to share data between different layers of an application	Must remember to unsubscribe

Unsubscribing from Observables

Unsubscribing from Observables

ngOnDestroy/
unsubscribe

AutoUnsubscribe
decorator

SubSink

```
export class MyComponent implements OnInit, OnDestroy {

  eventbusSub: Subscription;

  constructor(private eventbus: EventBusService) { }

  ngOnInit() {
    this.eventbusSub = this.eventbus.on(Events.CustomerSelected, (cust => this.customer = cust));
  }

  ngOnDestroy() {
    if (this.eventbusSub) {
      this.eventbusSub.unsubscribe();
    }
  }
}
```

Unsubscribing in ngOnDestroy

Each observable subscription must be assigned to a property

Subscribe from observables in ngOnDestroy

```
@AutoUnsubscribe()
export class MyComponent implements OnInit, OnDestroy {

  eventbusSub: Subscription;

  constructor(private eventbus: EventBusService, private dataService: DataService) { }

  ngOnInit() {
    this.eventbusSub = this.eventbus.on(Events.CustomerSelected, (cust => this.customer = cust));
  }

  ngOnDestroy() {
    // No need to manually unsubscribe
  }
}
```

Must add ngOnDestroy()

Using the AutoUnsubscribe Decorator

Component subscriptions can automatically be unsubscribed

https://github.com/NetanelBasal/ngx-auto-unsubscribe

```
export class MyComponent implements OnInit, OnDestroy {

  subs = new SubSink();

  constructor(private eventbus: EventBusService, private obsService: ObsService) { }

  ngOnInit() {
    this.subs.sink = this.eventbus.on(Events.CustomerSelected, (cust => this.customer = cust));
    this.subs.sink = this.obsService.subscribe(...);
  }

  ngOnDestroy() {
    this.subs.unsubscribe();
  }
}
```

Using SubSink

Add multiple subscription objects to subsink and call its unsubscribe() function in ngOnDestroy()

https://github.com/wardbell/subsink

Summary

RxJS subjects provide a flexible way to communicate between components

BehaviorSubject returns the last emitted value to new subscribers

An event bus can be used for loosely coupled communication

An observable service exposes uses a subject to expose an observable

Unsubscribe from observables (pick a technique that works for you)