COMPONENT COMMUNICATIONS

@Input() & @Output()

component needs to **receive** values

input()

needs to **communicate** values, emit **events** via: o**utput()**

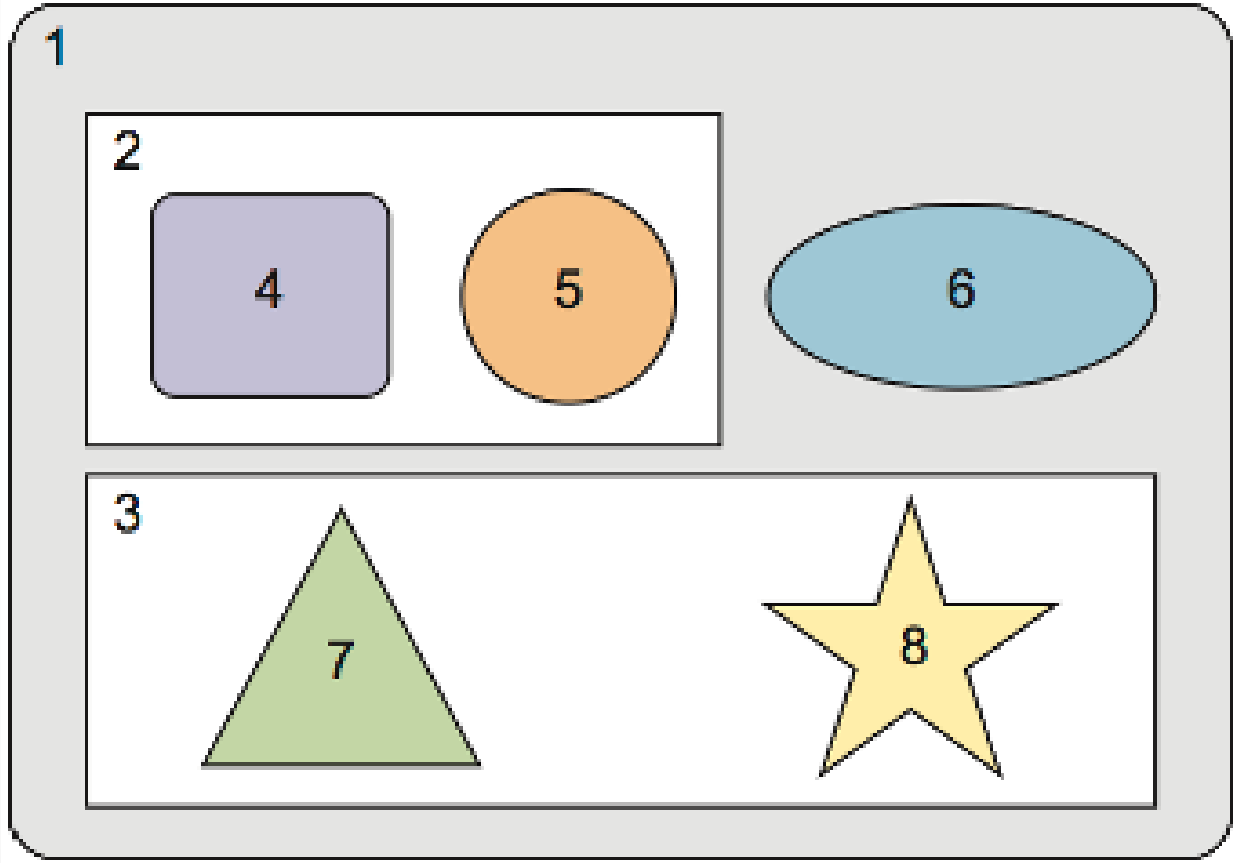
THE MEDIATOR DESIGN PATTERN

communication when components have **common parent**

communication when components **do not have common parent**

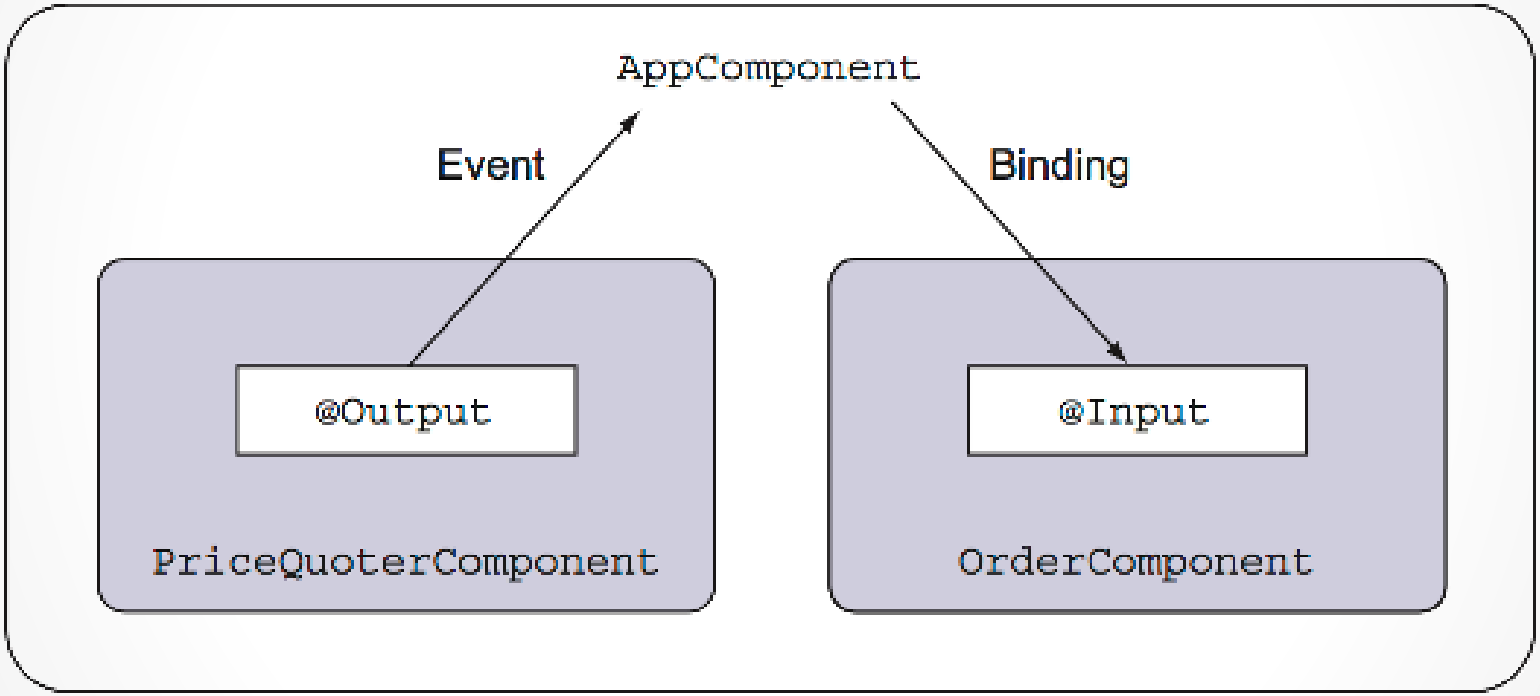
THE MEDIATOR DESIGN PATTERN

common parent



THE MEDIATOR DESIGN PATTERN

common parent



THE MEDIATOR DESIGN PATTERN

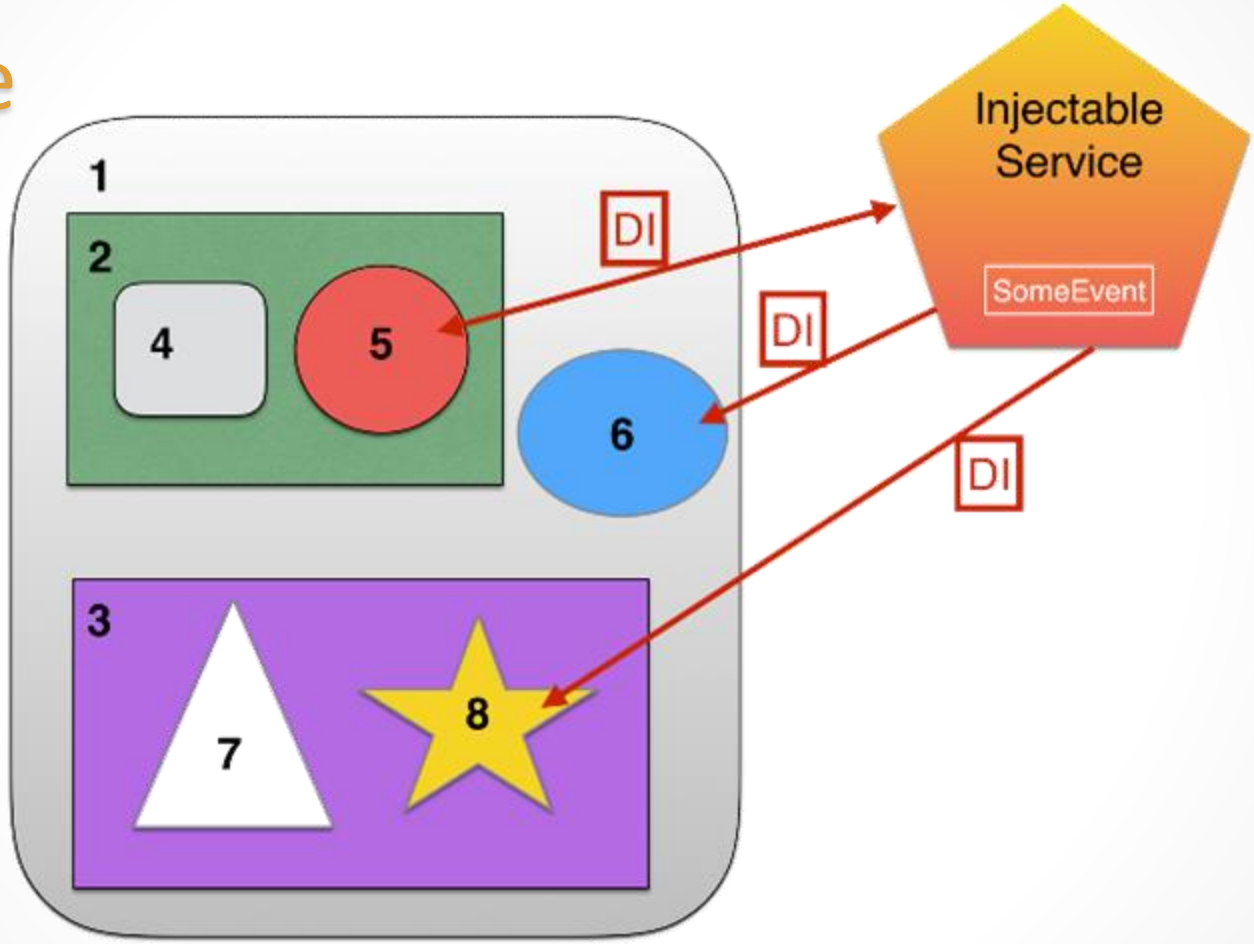
service

components **don't have the same parent** or aren't displayed at the same time

use an **injectable service** as a **mediator**

THE MEDIATOR DESIGN PATTERN

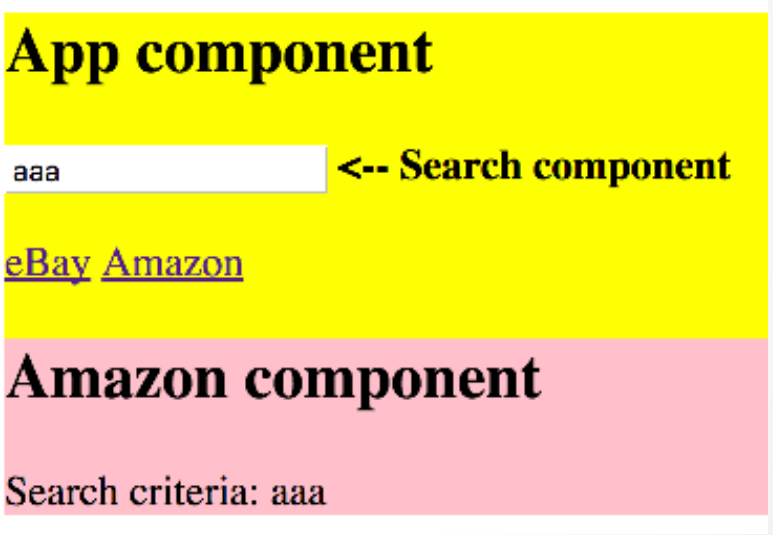
service



THE MEDIATOR DESIGN PATTERN

service

Exemplo:



THE MEDIATOR DESIGN PATTERN

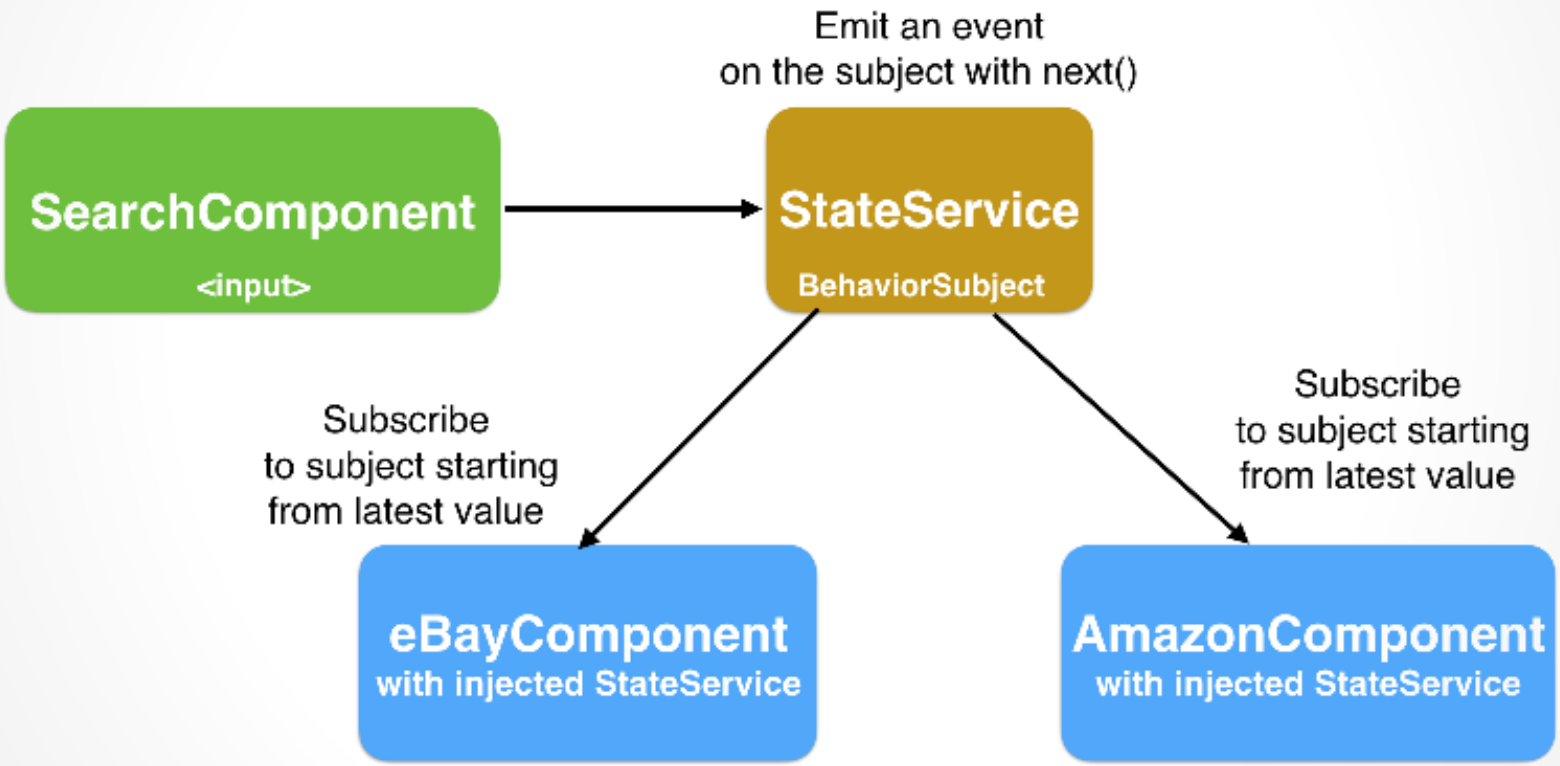
service

implement **two features** here:

1. **Implement communication between** the search, eBay, and Amazon **components**.
2. **Implement state management**

THE MEDIATOR DESIGN PATTERN

service



RxJS – Subject() e BehaviorSubject()

Observable is just a function

does **not** have any **state**

Each **Subscription** receive is **own value**

RxJS – Subject() e BehaviorSubject()

BehaviorSubject (or **Subject**)
stores observer details,
runs the code only once and
gives the result to all observers

Exemplo …

RxJS – Subject() e BehaviorSubject()

```
// Observable
let randomNumGenerator1 = Rx.Observable.create(observer => {
   observer.next(Math.random());
});

let observer1 = randomNumGenerator1
      .subscribe(num => console.log('observer 1: '+ num));

let observer2 = randomNumGenerator1
      .subscribe(num => console.log('observer 2: '+ num));

// OUTPUT
"observer 1: 0.7184075243594013"
"observer 2: 0.4127185021133610 3"
```

RxJS – Subject() e BehaviorSubject()

```
// ------ BehaviorSubject/ Subject

let randomNumGenerator2 = new Rx.BehaviorSubject(0);
randomNumGenerator2.next(Math.random());

let observer1Subject = randomNumGenerator2
      .subscribe(num=> console.log('observer subject 1: '+ num));

let observer2Subject = randomNumGenerator2
      .subscribe(num=> console.log('observer subject 2: '+ num));

// Output
"observer subject 1: 0.8034263165479893"
"observer subject 2: 0.8034263165479893"
```

RxJS – Subject()

```
// Regular Subject

let subject = new Subject();

subject.next("b");

subject.subscribe((value) => {
  console.log("Subscription got", value); // Subscription wont get
                                          // anything at this point
});

subject.next("c"); // Subscription got c
subject.next("d"); // Subscription got d
```

RxJS – BehaviorSubject()

```
// Behavior Subject

// a is an initial value. if there is a subscription
// after this, it would get "a" value immediately
let bSubject = new BehaviorSubject("a");

bSubject.next("b");

bSubject.subscribe((value) => {
  console.log("Subscription got", value); // Subscription got b,
                                          // ^ This would not happen
                                          // for a generic observable
                                          // or generic subject by default

});

bSubject.next("c"); // Subscription got c
bSubject.next("d"); // Subscription got d
```

THE MEDIATOR DESIGN PATTERN

service

Voltar ao Exemplo:

Interaction With Servers Using HTTP

HTTPClient

Working with the **HttpClient** service

Creating a simple web server using the Node and Express frameworks

Developing an **Angular client that communicates with the Node server**

Intercepting HTTP requests and responses

HTTPClient

Browser-based web apps run **HTTP requests asynchronously**, so the UI remains responsive

Asynchronous HTTP requests can be implemented using callbacks, promises, or observables

HTTPCLient

Angular supports HTTP communications via the **HttpClient** service from the **@angular/common/http** package

HTTPClient

HttpClient have

get(), **post()**, **put()**, **delete()** and many other methods return an **Observable**

```
this.httpClient.get<Product>('/product/123')
.subscribe(
    data => console.log(`id: ${data.id} title: ${data.title}`),
    (err: HttpErrorResponse) => console.log(`Got error: ${err}`)
);
```

HTTPClient

By **default**, **HttpClient** expects the data in JSON format, and the data is automatically converted into **JavaScript objects**

use the **responseType** option, for other

```
this.httpClient
    .get<string>('/my_data_file.txt', {responseType: 'text'})
```

HTTPClient

Reading a JSON File

Example…

Iniciar projecto “client” dentro da pasta “HTTP”