Angular: Effective Component Patterns

Sharing data between components

Erxk Follow

Jun 17, 2019 · 6 min read ★

I have built a sample Angular application displaying four component patterns. These patterns are simple yet effective for sharing data between components. We will use concrete examples of these patterns to illustrate how to use them.

Download the source code on Github 🚀

Test it live on Stackblitz 🚀

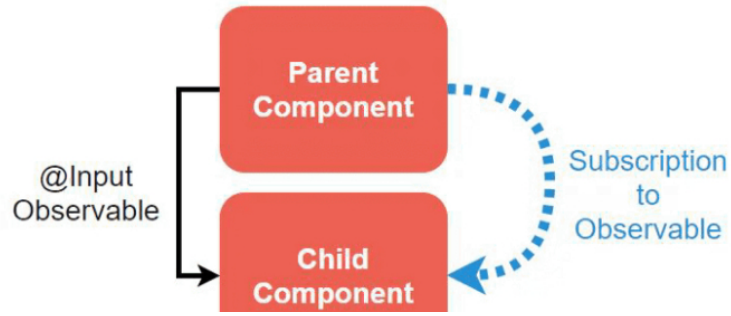
We will cover:

- Passing data from a parent component to a child component
- Passing data between sibling components
- Passing data from one component to an unrelated component.
- Multidirectional data passing between a parent and children components

Parent to Child Pattern

We can implement various patterns based on where our components are in relation to each other in the DOM. Whenever we need to facilitate communication between two components, we can reach for one of these patterns based on their relationship.

A parent-child relationship exists when an element (child) lives inside another element (parent)*. Our parent to child example will be a “Terms of Service” checkbox that will display a tool-tip on hover. When the box is checked, the message for the tool-tip will change.



Subject and Observable — Maintaining and Emitting State Changes

All of our patterns will utilize a `BehaviorSubject/state` and an `Observable/eventStream$` created from the BehaviorSubject. This allows us to maintain state in the BehaviorSubject and emit changes to that state through the Observable. [1.1] [1.2]

Tip: Utilizing a **private BehaviorSubject** and a **public Observable** allows us to **lock down access to our state and prevent excessive modification**.

```
export class ParentComponent implements OnInit {
  private state = new BehaviorSubject({
    enabled: false,
    toolTip: false
  });
  public eventStream$ = this.state.asObservable()

  update(value, command) {
    let update = this.state.value;
    if (command === 'enabled') update.enabled = value;
    if (command === 'toolTip') update.toolTip = value;
    this.state.next(update);
  }
}
```

parent.component.ts hosted with ❤ by GitHub — view raw

parent.component.ts | ToS Checkbox ☑

Now we can **leverage Angular Event Binding with RxJS to trigger updates to our state/data**. Whenever the parent's input is changed or interacts with the mouse, the parent state will be updated. [2]

All of our patterns will leverage Angular Event Binding and RxJS similar to this. The key differences will be how the components receive Observables, or how they pass state changes to each other.

```
<span>Agree to the Terms of Service:</span>
<input #toggle type="checkbox"
(change)="update(toggle.checked, 'enabled')"
(mouseover)="update(true, 'toolTip')"
(mouseleave)="update(false, 'toolTip')">
<app-child [eventStream]="eventStream$"></app-child>
```

parent.component.html hosted with ❤ by GitHub — view raw

parent.component.html | ToS Checkbox ☑

@Input — Receiving Changes in the Child Component

On initialization, our parent will pass its Observable to our child using `@Input`. Once the input has been initialized, **the child will receive any changes to the parent state through our async pipe subscription.** [3]

Utilizing this pattern, our components are now completely reactive. This simplifies working with the components when debugging or making future modifications.

```
<ng-container *ngIf="eventStream$ | async as toggle">
  <div *ngIf="toggle.toolTip">{{getMessage(toggle.enabled)}}</div>
</ng-container>
```

child.component.html hosted with ❤ by GitHub — view raw

child.component.html | Tool-Tip

Alternative Use of Observable and @Input

Instead of passing the entire Observable, we can use an async pipe to unwrap and pass the value from the parent to the child. Use-based on preference or if one suits a particular need better than the other.**

```
<ng-container *ngIf="eventStream$ | async as eventData">
  <span>Agree to the Terms of Service:</span>
<input #toggle type="checkbox"
(change)="update(toggle.checked, 'enabled')"
(mouseover)="update(true, 'toolTip')"
(mouseleave)="update(false, 'toolTip')">
  <app-child [eventData]="eventData"></app-child>
</ng-container>
```

parent.component.html hosted with ❤ by GitHub view raw

parent.component.html

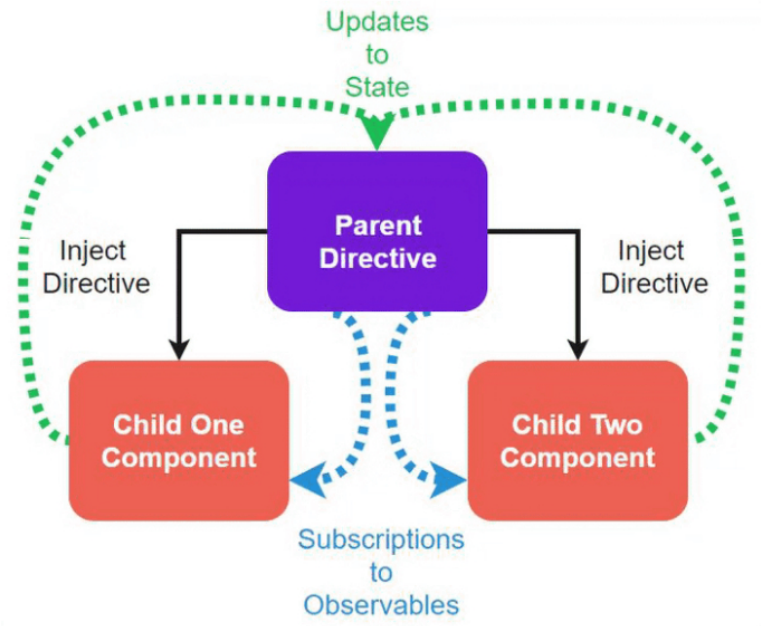
```
<div *ngIf="eventData.toolTip">
  {{getMessage(eventData.enabled)}}
</div>
```

child.component.html hosted with ❤ by GitHub view raw

child.component.html

Siblings Pattern

A sibling-relationship exists when multiple elements share a common parent element. Our sibling components example will be a color-picker that alters the color of a display. In this example, only our color-picker will update state. However, utilizing this pattern, any number of siblings could update state.



Container and Directive — Facilitating Communication

To pass data between our sibling components; we will wrap them in a container, such as div or ng-container, and apply an Angular Directive to the container. **This Directive will act as a parent or intermediary** between the siblings for passing state/data to and from each other.

Similar to our parent-child pattern, we still utilize a private

BehaviorSubject. However, we will expose our method for updating state to the siblings. This allows the siblings to send new data to our BehaviorSubject without directly accessing it.

```
export class ParentDirective {
  private state = new BehaviorSubject('#1e90ff');
  public eventStream$ = this.state.asObservable();

  updateColor(color) {
    this.state.next(color);
  }
}
```

parent.directive.ts hosted with ❤ by GitHub view raw

parent.directive.ts

Injecting the Directive — Accessing State

Our sibling components inject the parent directive [4]. This gives them the ability to reference our Directive's Observable `color$`. To access the value emitted from `color$` we will utilize the async pipe. Ideally, we want to use an async pipe whenever applicable. [5] (My article: Async Pipe Deep Dive)

```
// HTML
<ng-container *ngIf="color$ | async as color">
  <input #picker type="color"
  [value]="color"
  (change)="updateColor(picker.value)">
</ng-container>

// Typescript
export class ChildTwoComponent implements OnInit {

  color$;

  constructor(public directive: ParentDirective) { }

  ngOnInit() {
    this.color$ = this.directive.eventStream$;
  }

  updateColor(val) {
    this.directive.updateColor(val);
  }
}
```

childTwo.component.ts hosted with ❤ by GitHub view raw

child-two.component.ts | child-two.component.html | color-picker

We can add as many siblings as we want. We simply have to add the sibling element to the container and it will be able to send and receive data from the other components.

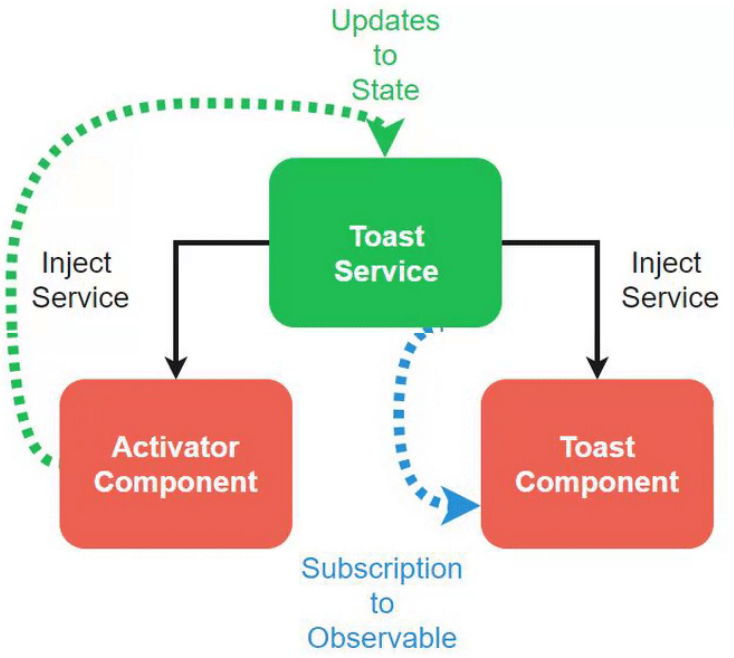
```
<div appParent>
  <app-child-one></app-child-one>
  <app-child-two></app-child-two>
</div>
```

app.component.html hosted with ❤ by GitHub view raw

app.component.html

Unrelated Pattern

Elements that do not share a mutual parent are considered unrelated. Our example will be a button and a toast message that do not share a common parent or wrapper. We will utilize an Angular Service to facilitate data sharing between the two components.



Injecting the Service — Accessing State

This pattern is similar to the sibling pattern. The key difference is that we utilize an Angular Service instead of an Angular Directive. **The advantage of the Service is that we no longer need both components in the same container.**

Compared to a Directive, a potential disadvantage of a Service is the increased complexity if we need multiple instances of the Service. In this scenario, we only need a single instance of the Service.

```
@Injectable()
export class ToastService {

  private bs = new BehaviorSubject(false);
  public eventStream$ = this.bs.asObservable();

  constructor() {}

  public setEnabled() {
    if (!this.bs.value) {
      this.bs.next(true);
      setTimeout(val => {
        this.bs.next(false);
      }, 5000);
    }
  }
}
```

toast.service.ts hosted with ❤ by GitHub view raw

toast.service.ts

Whenever the `ActivatorComponent` makes an update to the `ToastService` the `ToastComponent` will receive the update through the Service. An @Input is not needed in the scenario, as everything is facilitated through the Service.

```
export class ActivatorComponent {

  constructor(private ts: ToastService) { }

  onChange() {
    this.ts.setEnabled();
  }
}
```

activator.component.ts hosted with ❤ by GitHub view raw

activator.component.ts | Button

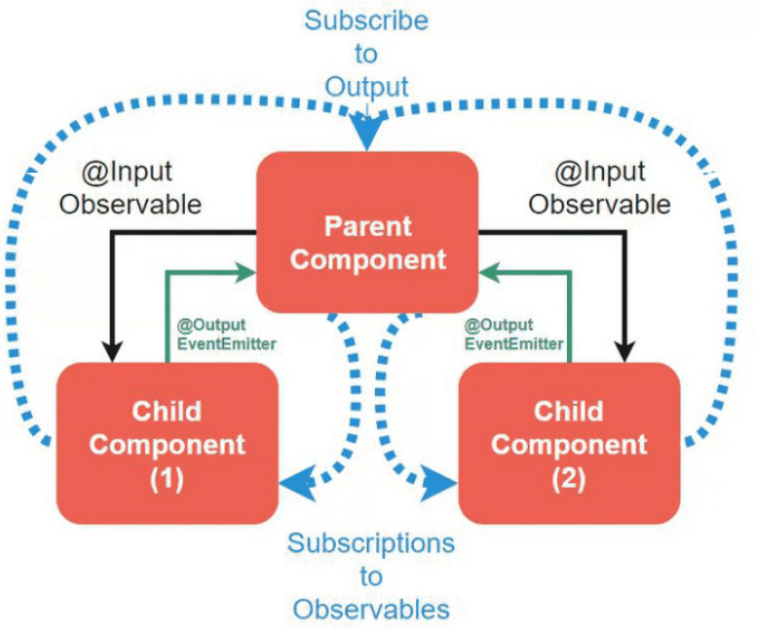
```ts
export class ToastComponent {

  eventStream$

  constructor(private ts: ToastService) {
    this.eventStream$ = ts.eventStream$;
   }

}
```

toast.component.ts hosted with ❤ by GitHub view raw

toast.component.ts | Toast

Multidirectional Data

Our last pattern will feature a parent element with multiple children elements. The parent will be able to pass state/data down to the children and the children will be able to pass state/data back up to the parent.



Subject and Observable — Maintaining and Emitting State Changes

Similar to the parent-child pattern we will utilize a private BehaviorSubject and a public Observable to maintain state and emit changes to the children.

```ts
export class MultiParentComponent implements OnInit {

  private state = new BehaviorSubject('Off');
  public eventStream$: Observable<any> = this.state.asObservable();

  updateStatus(value) {
    // ...
    this.state.next('Process Complete');
    // ...
  }
}
```

multiParent.component.ts hosted with ❤ by GitHub view raw

multi-parent.component.ts

@Output — Receiving Changes in the Parent Component

This is the key difference that makes our pattern multi-directional.

We utilize an @Output to allow our children components to emit events/state up to our parent component. [6]

We will replace our Observable in this scenario with an `EventEmitter`. Using EventEmitter with @Output is common practice and supported by the documentation.

```html
<!-- Parent HTML -->
<h3>{{state.value}}</h3>
<button #startBtn type="button" (click)="state.next('Running')">Start</button>
<app-multi-child [eventStream]="eventStream$" (done)="updateStatus($event)"></app-multi-child>
<app-multi-child [eventStream]="eventStream$" (done)="updateStatus($event)"></app-multi-child>


<!-- app-multi-child HTML -->
<div *ngIf="eventStream$ | async as status">
  <span *ngIf="state.value as value">{{value}}</span>
</div>
```

multiParent.component.html hosted with ❤ by GitHub — view raw

multi-parent.component.html | multi-child.component.html

When our EventEmitter emits a value, the `done($event)` method will be called. `$event` will contain whatever state/data that was emitted from our EventEmitter.

To maintain state and use the value in both the child and the parent, we emit the events on changes to our BehaviorSubject. If we did not need to maintain state, we could omit the BehaviorSubject and use just the EventEmitter.**

```typescript
@Input('eventStream') eventStream$;

state: BehaviorSubject<string> = new BehaviorSubject('');
@Output('done') eventEmitter = new EventEmitter<string>();
// outputStream$ = this.state.asObservable();

ngOnInit() {
  this.state.subscribe(tap(val => {
    this.eventEmitter.emit(val);
  })); // Unsubscribe in ngOnDestroy
 // ...
}
```

multiChild.component.ts hosted with ❤ by GitHub — view raw

multi-child.component.ts

Summary

- **Parent to Child Pattern:** Utilize a Subject and Observable in the parent and an @Input in the child to facilitate sending data from the parent to the child
- **Siblings Pattern:** Subject and Observable live inside a Directive that acts as an intermediary for sending and receiving data between siblings
- **Unrelated Pattern:** Subject and Observable live inside a Service, components do not have to share a parent or container
- **Multidirectional Parent Child Pattern:** Same as the Parent to Child pattern. Adds an @Output to the children components to send state/events back to the parent.

. . .

Resources / References

[1.1] http://reactivex.io/rxjs/manual/overview.html#observable

[1.2] http://reactivex.io/rxjs/manual/overview.html#behaviorsubject

[2] https://angular.io/guide/user-input

[3] https://angular.io/api/core/Input

[4] https://angular.io/guide/dependency-injection

[5] My Article: https://medium.com/better-programming/angular-rxjs-async-pipe-deep-dive-2510b56f793a

[6] https://angular.io/api/core/Output

* — component/element is used interchangeably. When referring to relationships or position in DOM: element. When referring to data usage or code: component
** — Thank you to those who gave feedback

JavaScript Programming Software Engineering Angular Web Development

undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Explore

post your thinking on any topic. Start a blog