**Component Interaction**

**Parent to Child: Sharing Data via Input**

This is probably the most common and straightforward method of sharing data.

It works by using the [**@Input() decorator**](https://angular.io/docs/ts/latest/api/core/index/Input-interface.html) to allow data to be passed via the template.

parent.component.html

<p>parent</p>

<app-child [childMessage]="parentMessage"></app-child>

parent.component.ts

import {Component} from '@angular/core'

@Component({

  selector:'app-parent',

  templateUrl:'./parent.component.html',

  styleUrls:['./parent.component.css']

})

export class appParent{

  parentMessage="message from parent";

}

child.component.html

<p>child ===> {{childMessage}}</p>

child.component.ts

import { Component, Input } from "@angular/core";

@Component({

  selector: "app-child",

  templateUrl: "./child.component.html",

  styleUrls: ["./child.component.css"]

})

export class appChild {

  @Input() childMessage: string;

}

app.component.html

<app-parent></app-parent>

app.module.ts

import { NgModule } from "@angular/core";

import { BrowserModule } from "@angular/platform-browser";

import { FormsModule } from "@angular/forms";

import { AppComponent } from "./app.component";

import { HelloComponent } from "./hello.component";

import { appParent } from "../parent/parent.component";

import { appChild } from "../child/child.component";

@NgModule({

  imports: [BrowserModule, FormsModule],

  declarations: [AppComponent, HelloComponent, appParent, appChild],

  bootstrap: [AppComponent]

})

export class AppModule {}

**Child to Parent: Sharing Data via ViewChild**

[**ViewChild**](https://angular.io/api/core/ViewChild) allows a one component to be injected into another, giving the parent access to its attributes and functions.

One caveat, however, is that child won’t be available until after the view has been initialized.

This means we need to implement the AfterViewInit lifecycle hook to receive the data from the child.

parent.component.html

<p>sample</p>

<app-child></app-child>

{{message}}

parent.component.ts

import { Component, ViewChild, AfterViewInit } from "@angular/core";

import { appChild } from "../child/child.component";

@Component({

  selector: "app-parent",

  templateUrl: "../parent/parent.component.html",

  styleUrls: ["../parent/parent.component.css"]

})

export class appParent implements AfterViewInit {

  @ViewChild(appChild) child;

  message: string;

  ngAfterViewInit() {

    this.message= this.child.msg;

  }

}

child.component.html

<p>sample</p>

child.component.ts

import { Component } from "@angular/core";

@Component({

  selector: "app-child",

  templateUrl: "../child/child.component.html",

  styleUrls: ["../child/child.component.css"]

})

export class appChild {

  msg = "hello sakthi";

}

app.module.ts

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { HelloComponent } from './hello.component';

import {appParent} from './parent/parent.component'

import {appChild} from './child/child.component'

@NgModule({

  imports:      [ BrowserModule, FormsModule ],

  declarations: [ AppComponent, HelloComponent,appParent,appChild ],

  bootstrap:    [ AppComponent ]

})

export class AppModule { }

**Child to Parent: Sharing Data via Output() and EventEmitter**

Another way to share data is to emit data from the child, which can be listed to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entires, and other user events.

In the parent, we create a function to receive the message and set it equal to the message variable.

In the child, we declare a messageEvent variable with the Output decorator and set it equal to a new event emitter. Then we create a function named sendMessage that calls emit on this event with the message we want to send. Lastly, we create a button to trigger this function.

The parent can now subscribe to this messageEvent that’s outputted by the child component, then run the receive message function whenever this event occurs.

parent.component.html

Message: {{message}}

<app-child (messageEvent)="receiveMessage($event)"></app-child>

parent.component.ts

import { Component } from "@angular/core";

@Component({

  selector: "app-parent",

  templateUrl: "../parent/parent.component.html",

  styleUrls: ["../parent/parent.component.css"]

})

export class appParent {

  message: string;

  receiveMessage($event) {

    this.message = $event;

  }

}

child.component.html

<div>

  <button (click)="sendMessage()">Send Item</button>

</div>

child.component.ts

import { Component, Output, EventEmitter } from "@angular/core";

@Component({

  selector: "app-child",

  templateUrl: "../child/child.component.html",

  styleUrls: ["../child/child.component.css"]

})

export class appChild {

  message = "Hello Sakthi";

  @Output() messageEvent = new EventEmitter<String>();

  sendMessage() {

    this.messageEvent.emit(this.message);

  }

}

app.module.ts

import { NgModule } from "@angular/core";

import { BrowserModule } from "@angular/platform-browser";

import { FormsModule } from "@angular/forms";

import { AppComponent } from "./app.component";

import { HelloComponent } from "./hello.component";

import { appParent } from './parent/parent.component';

import { appChild } from './child/child.component';

@NgModule({

  imports: [BrowserModule, FormsModule],

  declarations: [AppComponent, HelloComponent, appParent, appChild],

  bootstrap: [AppComponent]

})

export class AppModule {}

**Unrelated Components: Sharing Data with a Service**

When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should you a shared service. When you have data that should aways been in sync, I find the **[RxJS BehaviorSubject](https://xgrommx.github.io/rx-book/content/subjects/behavior_subject/index.html)** very useful in this situation.

You can also use a regular RxJS Subject for sharing data via the service, but here’s why I prefer a BehaviorSubject.

* It will always return the current value on subscription - there is no need to call **onnext**
* It has a **getValue()** function to extract the last value as raw data.
* It ensures that the component always receives the most recent data.

In the service, we create a private BehaviorSubject that will hold the current value of the message. We define a currentMessage variable handle this data stream as an observable that will be used by the components. Lastly, we create function that calls next on the BehaviorSubject to change its value.

The parent, child, and sibling components all receive the same treatment. We inject the DataService in the constructor, then subscribe to the currentMessage observable and set its value equal to the message variable.

Now if we create a function in any one of these components that changes the value of the message. when this function is executed the new data it’s automatically broadcast to all other components.