**1)-Component Interaction:**

Angular components can communicate with each other in parent-child or sibling relationships.

**A)-Parent-to-child** communication: Use input properties to pass data from parent to child

**components. @Input** decorator is used in the child component to receive the data.

**B)-Child-to-parent** communication: Emit events from the child component **using @Output**

and EventEmitter. The parent component listens to these events and responds accordingly.

Parent Component:

<app-child [inputData]="dataFromParent" (outputEvent)="handleChildEvent($event)"></app-child>

Child Component:

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

@Component({

selector: 'app-child',

template: `

<button (click)="emitEvent()">Send Event to Parent</button>

`

})

export class ChildComponent {

@Input() inputData: string;

@Output() outputEvent = new EventEmitter<string>();

emitEvent() {

this.outputEvent.emit('Child component event');

}

}

**2)-Service Communication:**

Services provide a central place to share data and functionality across different

parts of your application.Components can inject the same service and use it to

exchange data and trigger actions.Services can be used to hold shared state

or provide methods for inter-component communication.

**3)-RxJS Observables:**

RxJS observables can be used to create data streams that allow components and

services to communicate asynchronously.

Observables provide a powerful mechanism for handling events, data updates,

and asynchronous operations in a reactive manner.

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

import { Observable, Subject } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class DataService {

private dataSubject = new Subject<string>();

sendData(data: string) {

this.dataSubject.next(data);

}

getData(): Observable<string> {

return this.dataSubject.asObservable();

}

}

Component A:

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

import { DataService } from './data.service';

@Component({

selector: 'app-component-a',

template: `

<button (click)="sendData()">Send Data to Component B</button>

`

})

export class ComponentA {

constructor(private dataService: DataService) {}

sendData() {

this.dataService.sendData('Data from Component A');

}

}

Component B:

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

import { DataService } from './data.service';

@Component({

selector: 'app-component-b',

template: `

<p>{{ receivedData }}</p>

`

})

export class ComponentB {

receivedData: string;

constructor(private dataService: DataService) {

this.dataService.getData().subscribe(data => this.receivedData = data);

}

}

**4)-State Management Libraries:**

Libraries like NgRx and Akita provide advanced state management solutions for

Angular applications.

They use observables to manage application state, enabling efficient

communication between components and predictable data flow.

**5)-Input and Output Properties:**

Components can communicate through input properties (bindings) and output properties (events).

Input properties allow parents to pass data to child components.

Output properties allow child components to emit events to their parent components.

**6)-ViewChild and ContentChild:**

**@ViewChild and @ContentChild** decorators allow you to access child elements or

components in a parent component.

This is useful for interacting with specific child components or

elements and calling their methods.

**7)-Local Variables and Template Reference Variables:**

Local variables can be used to reference elements or components directly within a template.

Template reference variables (#varName) can be used to reference HTML elements or

Angular components in templates.

**8)-Route Parameters and Query Parameters:**

Components can communicate through route parameters and query parameters when navigating between routes.

Components can read route parameters and query parameters to determine context and data.

**9)-HTTP Requests and APIs:**

Components can communicate with external APIs to fetch and update data using HTTP requests.

Services are often used to encapsulate API calls and provide data to components.

Third-Party Libraries and APIs:

Components can communicate with third-party libraries and APIs to integrate functionality such as maps, charts, or social media sharing.