In Angular, communication between components can be achieved using various techniques such as

1.Input and Output properties,

2.ViewChild and ContentChild decorators,

3. services, and

4. event emitters.

Here are examples of each method:

Input ans Output used to send data from child to parent

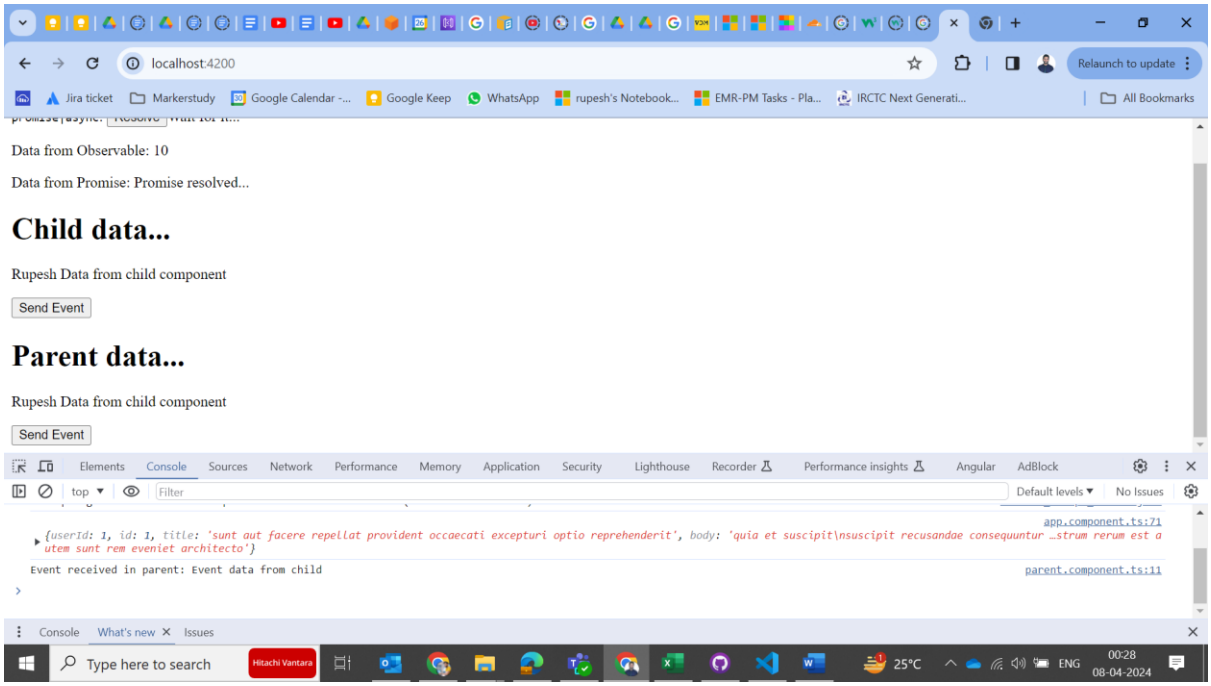
```
export class ChildComponent {
  @Input() data!: string;
  @Output() onEvent: EventEmitter<any> = new EventEmitter<any>();

  sendEvent() {
    this.data = 'Rupesh Data from child component';
    this.onEvent.emit('Event data from child');
  }
```

```
<p>{{ data }}</p>
<button (click)="sendEvent()">Send Event</button>
```

Parent com➔>

```
export class ParentComponent {
  parentData = 'Data from parent';
  handleEvent(eventData: any) {
    console.log('Event received in parent:', eventData);
  }
```

Output:➔



Using ViewChild and ContentChild Decorators:

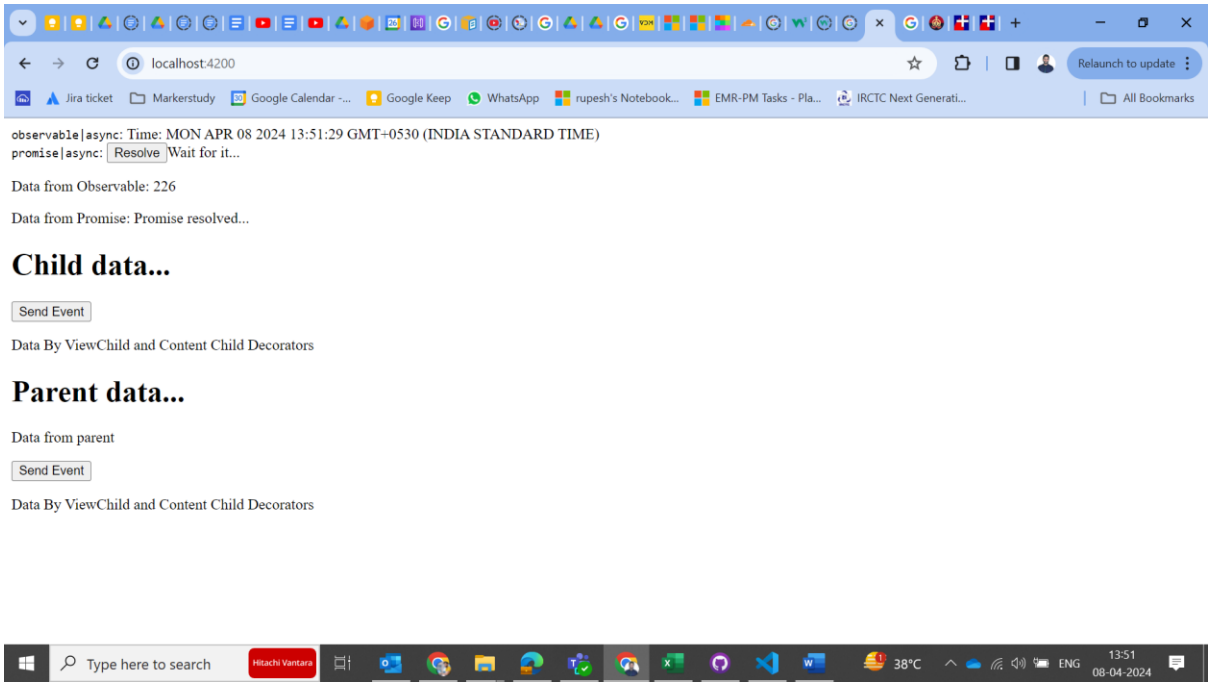
Childhtmpl

```
<p>Data By ViewChild and Content Child Decorators</p>
```

Parent.component.ts

```
export class ParentComponent implements AfterViewInit {
  ngAfterViewInit(): void {
    console.log(this.childComponent);
  }
  @ViewChild(ChildComponent)
  childComponent!: ChildComponent;
```

output:



Using Services:

```
import { Injectable, EventEmitter } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  dataUpdated: EventEmitter<any> = new EventEmitter<any>();

  updateData(data: any) {
    this.dataUpdated.emit(data);
  }
}
```

Parent Component:

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
```

```
  selector: 'parent-component',
  template: `
    <button (click)="updateData()">Update Data</button>
  `
})
export class ParentComponent {
  constructor(private dataService: DataService) {}

  updateData() {
    this.dataService.updateData('New data');
  }
}
```

Child Component:

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'child-component',
  template: `
    <p>{{ data }}</p>
  `
})
export class ChildComponent implements OnInit {
  data: any;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.dataUpdated.subscribe((data: any) => {
      this.data = data;
    });
  }
}
```

Using Event Emitters:

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

@Component({
  selector: 'parent-component',
  template: `
    <child-component (onEvent)="handleEvent($event)"></child-component>
  `
})
export class ParentComponent {
  handleEvent(eventData: any) {
    console.log('Event received in parent:', eventData);
  }
}
```

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

@Component({
  selector: 'child-component',
  template: `
    <button (click)="sendEvent()">Send Event</button>
  `
})
export class ChildComponent {
  @Output() onEvent: EventEmitter<any> = new EventEmitter<any>();

  sendEvent() {
    this.onEvent.emit('Event data from child');
  }
}
```

Each of these methods provides a way to establish communication between Angular components in different scenarios. The choice of method depends on factors such as component relationship, data flow direction, and complexity of communication requirements.

In enterprise applications, the most recommended method for communication between components in Angular depends on various factors such as scalability, maintainability, data flow, and complexity of communication requirements. Here's a breakdown of the most recommended methods and when they are suitable:

Using Services:

- **Recommended for**: Complex data sharing, cross-component communication, state management, and data synchronization.
- **Advantages**:
    - Centralized data management: Services act as a single source of truth for shared data and state.
    - Dependency Injection (DI): Angular's DI system ensures that services are singletons and can be injected into multiple components, ensuring data consistency.
    - Promotes modularity: Encourages separation of concerns by keeping business logic separate from UI components.
- **Considerations**:
    - May introduce complexity: Services are suitable for managing complex state and communication needs, but they require careful design to avoid becoming overly complex.
    - Dependency management: Dependencies between services and components need to be managed effectively to prevent tight coupling.
- **Usage Example**: Managing user authentication, fetching data from APIs, sharing application configuration settings.

Using Input and Output Properties:

- **Recommended for**: Parent-child component communication, passing data from parent to child components.
- **Advantages**:
    - Simple and straightforward: Input and Output properties provide a clear and intuitive way to pass data between components.
    - Angular's change detection handles property binding efficiently.
- **Considerations**:
    - Limited to parent-child hierarchy: Suitable for communication within a component tree, but less convenient for communication between unrelated components.
    - Can become cumbersome for deep component hierarchies or complex data sharing scenarios.
- **Usage Example**: Passing data to child components, handling events in parent components.

Using Services with Observables:

- **Recommended for**: Asynchronous data streams, real-time updates, event-driven communication.
- **Advantages**:
  - Supports asynchronous data handling: Observables provide a powerful mechanism for handling asynchronous data streams and real-time updates.
  - Reactive programming: Allows components to react to changes in data streams efficiently.
  - Promotes loose coupling: Components can subscribe to relevant data streams without directly interacting with each other.
- **Considerations**:
  - Requires understanding of reactive programming concepts: Developers need to be familiar with concepts such as observables, operators, and subscriptions.
  - May require additional setup and configuration compared to simpler communication methods.
- **Usage Example**: Real-time updates from server, handling user interactions with dynamic data.

Using ViewChild and ContentChild:

- **Recommended for**: Accessing child components, querying DOM elements, and interacting with child component instances.
- **Advantages**:
  - Provides direct access to child components and DOM elements, enabling fine-grained control over their behavior.
  - Useful for scenarios where direct manipulation of child components or DOM elements is necessary.
- **Considerations**:
  - Tight coupling: Directly accessing child components or DOM elements can lead to tighter coupling between parent and child components.
  - May violate encapsulation: Exposes internal implementation details of child components to parent components.
- **Usage Example**: Accessing child component methods, querying DOM elements for specific functionality.

Transfer data to component by Service

Message service.ts

```
export class MessageService {
  private messageSubject: Subject<string> = new Subject<string>();
  message$: Observable<string> = this.messageSubject.asObservable();

  sendMessage(message: string) {
    this.messageSubject.next(message);
  }
  constructor() { }
}
```

componentA will send message to B

```
export class ComponentaComponent {
  public message!: string;
  constructor(private dataService: DataService, private messageService:
MessageService) { }

sendMessage() {
    this.messageService.sendMessage(this.message);
  }
}
```

And html A

```
<input type="text" [(ngModel)]="message">
<button (click)="sendMessage()">Send Message</button>
```

Comp B

```
 ngOnInit() {
    this.dataService.data$.subscribe(data => {
      this.data = data;
    });
    this.messageService.message$.subscribe(message => {
      this.messages.push(message);
    });
```

```
  }
```

And CompB Html

```
<div *ngFor="let message of messages">{{ message }}</div>
```