## Question 3: Component Interaction

1. Options to establish communication

- **Input Binding**

  Data moves from Parent to Child component when you define the **@Input()** in the child component it will receive data from the parent component. Before data is shared we define what type of data is expected in the child component by defining the input type. Type declaration can done either by defining the model using an interface or if the expected input is simple variable types like Boolean, number etc. As shown by the snippet below

  <component-a [data]="data"></component-a>

  @Input() data : any[]

- **Output Binding and EventEmmitter**
  In this case, data moves from Child to parent component that is the opposite of the Input Binding. In the child component we define the **@Output()** decorator. The child component fires an event using an event emmiter to the parent component which is handled by an event handler to execute a function in the parent component.

- **Services**

  This can be used to share data and values by creating a service which can be injected into both Component A and component B.

  The service can access data and services in either of the components.

- **@ViewChild Decorator**

  Sharing data between siblings can be done by using points 1 and 2. First share data between the ***child to parent*** using output decorator and Event Emitter. Once received data in ***parent component share it with another child*** component using Input decorator. Siblings can talk to each other via parent components.

2. Pros and Cons

**Input Binding**

| **Pros** | **Cons** |
|---|---|
| Allows easy use of dump components | Cannot work efficiently if you have a large number of components |
| Easy to implement | Input data is strongly defined |
| | Cannot work efficiently if components are not in the same module |

**Output Binding and EventEmmitter**

| Pros | Cons |
|---|---|
| Easy to implement | Many languages prevent multiple inheritance due to Diamond problem. On the other hands, one class can have multiple decorators for different purposes (e.g. @Component and the deprecated @RouteConfig) |
| information inside decorators is declarative, they define the behaviour of a class, most likely won't change over time and are used by the framework. Class properties and fields are class-specific data, will always be processed and frequently updated, and only are meaningful within the class itself. These two kinds of data should not be mixed together | |

**Services**

| Pros | Cons |
|---|---|
| Can handle multiple components efficiently | Adds complex logic when data is being manipulated by multiple componets |
| Can be used across modules | |

**@ViewChild**

| Pros | Cons |
|---|---|
| All parameters are now located in Typescript. This seems better than mixing/muddling into the html markup. Business/data logic parameters can now be in one place. | Testing such a component is hard since the child component isn't aware of his state and the changes the parent takes care of, you'll have to test them as one. |
| | |

3. Cases solution fits best

**Input Binding**
When you just want to share data from Component A to component B

**Output Binding and EventEmmitter**

When you want to share data from component B to component A.

**Services**

When you want to share data and methods between component A component B and any other components in the application

**@ViewChild Decorator**
When you want to share data between Component A to B.