

Component interaction

{@a top}

This cookbook contains recipes for common component communication scenarios in which two or more components share information. {@a toc}

See the .

{@a parent-to-child}

Pass data from parent to child with input binding

`HeroChildComponent` has two *input properties*, typically adorned with `@Input()` decorator.

The second `@Input` aliases the child component property name `masterName` as `'master'`.

The `HeroParentComponent` nests the child `HeroChildComponent` inside an `*ngFor` repeater, binding its `master` string property to the child's `master` alias, and each iteration's `hero` instance to the child's `hero` property.

The running application displays three heroes:

Test it

E2E test that all children were instantiated and displayed as expected:

Back to top

{@a parent-to-child-setter}

Intercept input property changes with a setter

Use an input property setter to intercept and act upon a value from the parent.

The setter of the `name` input property in the child `NameChildComponent` trims the whitespace from a name and replaces an empty value with default text.

Here's the `NameParentComponent` demonstrating name variations including a name with all spaces:

Test it

E2E tests of input property setter with empty and non-empty names:

Back to top

{@a parent-to-child-on-changes}

Intercept input property changes with *ngOnChanges()*

Detect and act upon changes to input property values with the `ngOnChanges()` method of the `OnChanges` lifecycle hook interface.

You might prefer this approach to the property setter when watching multiple, interacting input properties.

Learn about `ngOnChanges()` in the Lifecycle Hooks chapter.

This `VersionChildComponent` detects changes to the `major` and `minor` input properties and composes a log message reporting these changes:

The `VersionParentComponent` supplies the `minor` and `major` values and binds buttons to methods that change them.

Here's the output of a button-pushing sequence:

Test it

Test that *both* input properties are set initially and that button clicks trigger the expected `ngOnChanges` calls and values:

Back to top

```
{@a child-to-parent}
```

Parent listens for child event

The child component exposes an `EventEmitter` property with which it **emits** events when something happens. The parent binds to that event property and reacts to those events.

The child's `EventEmitter` property is an *output property*, typically adorned with an `@Output()` decorator as seen in this `VoterComponent`:

Clicking a button triggers emission of a `true` or `false`, the boolean *payload*.

The parent `VoteTakerComponent` binds an event handler called `onVoted()` that responds to the child event payload `$event` and updates a counter.

The framework passes the event argument—represented by `$event`—to the handler method, and the method processes it:

Test it

Test that clicking the *Agree* and *Disagree* buttons update the appropriate counters:

Back to top

Parent interacts with child using *local variable*

A parent component cannot use data binding to read child properties or invoke child methods. Do both by creating a template reference variable for the child element and then reference that variable *within the parent template* as seen in the following example.

{@a countdown-timer-example} The following is a child `CountdownTimerComponent` that repeatedly counts down to zero and launches a rocket. The `start` and `stop` methods control the clock and a countdown status message displays in its own template.

The `CountdownLocalVarParentComponent` that hosts the timer component is as follows:

The parent component cannot data bind to the child's `start` and `stop` methods nor to its `seconds` property.

Place a local variable, `#timer`, on the tag `<countdown-timer>` representing the child component. That gives you a reference to the child component and the ability to access *any of its properties or methods* from within the parent template.

This example wires parent buttons to the child's `start` and `stop` and uses interpolation to display the child's `seconds` property.

Here, the parent and child are working together.

```
{@a countdown-tests}
```

Test it

Test that the seconds displayed in the parent template match the seconds displayed in the child's status message. Test also that clicking the *Stop* button pauses the countdown timer:

Back to top

```
{@a parent-to-view-child}
```

Parent calls an `@ViewChild()`

The *local variable* approach is straightforward. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component *itself* has no access to the child.

You can't use the *local variable* technique if the parent component's *class* relies on the child component's *class*. The parent-child relationship of the components is not established within each components respective *class* with the *local variable* technique. Because the *class* instances are not connected to one another, the parent *class* cannot access the child *class* properties and methods.

When the parent component *class* requires that kind of access, *inject* the child component into the parent as a *ViewChild*.

The following example illustrates this technique with the same Countdown Timer example. Neither its appearance nor its behavior changes. The child `CountdownTimerComponent` is the same as well.

The switch from the *local variable* to the *ViewChild* technique is solely for the purpose of demonstration.

Here is the parent, `CountdownViewChildParentComponent`:

It takes a bit more work to get the child view into the parent component *class*.

First, you have to import references to the `ViewChild` decorator and the `AfterViewInit` lifecycle hook.

Next, inject the child `CountdownTimerComponent` into the private `timerComponent` property using the `@ViewChild` property decoration.

The `#timer` local variable is gone from the component metadata. Instead, bind the buttons to the parent component's own `start` and `stop` methods and present the ticking seconds in an interpolation around the parent component's `seconds` method.

These methods access the injected timer component directly.

The `ngAfterViewInit()` lifecycle hook is an important wrinkle. The timer component isn't available until *after* Angular displays the parent view. So it displays 0 seconds initially.

Then Angular calls the `ngAfterViewInit` lifecycle hook at which time it is *too late* to update the parent view's display of the countdown seconds. Angular's unidirectional data flow rule prevents updating the parent view's in the same cycle. The application must *wait one turn* before it can display the seconds.

Use `setTimeout()` to wait one tick and then revise the `seconds()` method so that it takes future values from the timer component.

Test it

Use the same countdown timer tests as before.

Back to top

{@a bidirectional-service}

Parent and children communicate using a service

A parent component and its children share a service whose interface enables bi-directional communication *within the family*.

The scope of the service instance is the parent component and its children. Components outside this component subtree have no access to the service or their communications.

This `MissionService` connects the `MissionControlComponent` to multiple `AstronautComponent` children.

The `MissionControlComponent` both provides the instance of the service that it shares with its children (through the `providers` metadata array) and injects that instance into itself through its constructor:

The `AstronautComponent` also injects the service in its constructor. Each `AstronautComponent` is a child of the `MissionControlComponent` and therefore receives its parent's service instance:

Notice that this example captures the `subscription` and `unsubscribe()` when the `AstronautComponent` is destroyed. This is a memory-leak guard step. There is no actual risk in this application because the lifetime of a `AstronautComponent` is the same as the lifetime of the application itself. That *would not* always be true in a more complex application.

You don't add this guard to the `MissionControlComponent` because, as the parent, it controls the lifetime of the `MissionService`.

The *History* log demonstrates that messages travel in both directions between the parent `MissionControlComponent` and the `AstronautComponent` children, facilitated by the service:

Test it

Tests click buttons of both the parent `MissionControlComponent` and the `AstronautComponent` children and verify that the history meets expectations:

[Back to top](#)

@reviewed 2021-09-17