# Fundamentals

## Components & Templates: Component interaction

This cookbook contains recipes for common component communication scenarios in which two or more components share information.

**See the**[live example](https://angular.io/generated/live-examples/component-interaction/stackblitz.html) / [download example](https://angular.io/generated/zips/component-interaction/component-interaction.zip).

## Pass data from parent to child with input binding

HeroChildComponent has two input properties, typically adorned with [@Input decorations](https://angular.io/guide/template-syntax#inputs-outputs).

component-interaction/src/app/hero-child.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input) } from '@angular/core';

import { Hero } from './hero';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-child',

template: `

<h3>{{hero.name}} says:</h3>

<p>I, {{hero.name}}, am at your service, {{masterName}}.</p>

`

})

export class HeroChildComponent {

@[Input](https://angular.io/api/core/Input)() hero: Hero;

@[Input](https://angular.io/api/core/Input)('master') masterName: string;

}

The second @[Input](https://angular.io/api/core/Input) aliases the child component property name masterName as 'master'.

The HeroParentComponent nests the child HeroChildComponent inside an \*[ngFor](https://angular.io/api/common/NgForOf) repeater, binding its master string property to the child's master alias, and each iteration's hero instance to the child's hero property.

component-interaction/src/app/hero-parent.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

import { HEROES } from './hero';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-parent',

template: `

<h2>{{master}} controls {{heroes.length}} heroes</h2>

<app-hero-child \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"

[hero]="hero"

[master]="master">

</app-hero-child>

`

})

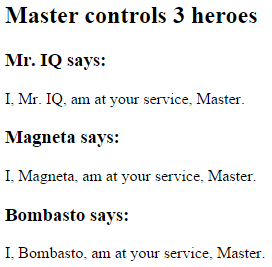
export class HeroParentComponent {

heroes = HEROES;

master = 'Master';

}

The running application displays three heroes:



### Test it

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

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

let \_heroNames = ['Dr IQ', 'Magneta', 'Bombasto'];

let \_masterName = 'Master';

it('should pass properties to children properly', function () {

let parent = element.all(by.tagName('app-hero-parent')).get(0);

let heroes = parent.all(by.tagName('app-hero-child'));

for (let i = 0; i < \_heroNames.length; i++) {

let childTitle = heroes.get(i).element(by.tagName('h3')).getText();

let childDetail = heroes.get(i).element(by.tagName('p')).getText();

expect(childTitle).toEqual(\_heroNames[i] + ' says:');

expect(childDetail).toContain(\_masterName);

}

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## 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.

component-interaction/src/app/name-child.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-name-child',

template: '<h3>"{{name}}"</h3>'

})

export class NameChildComponent {

private \_name = '';

@[Input](https://angular.io/api/core/Input)()

set name(name: string) {

this.\_name = (name && name.trim()) || '<no name set>';

}

get name(): string { return this.\_name; }

}

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

component-interaction/src/app/name-parent.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-name-parent',

template: `

<h2>Master controls {{names.length}} names</h2>

<app-name-child \*[ngFor](https://angular.io/api/common/NgForOf)="let name of names" [name]="name"></app-name-child>

`

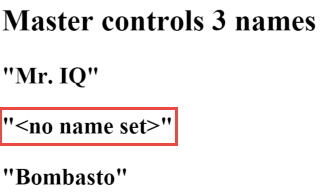
})

export class NameParentComponent {

// Displays 'Dr IQ', '<no name set>', 'Bombasto'

names = ['Dr IQ', ' ', ' Bombasto '];

}



### Test it

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

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should display trimmed, non-empty names', function () {

let \_nonEmptyNameIndex = 0;

let \_nonEmptyName = '"Dr IQ"';

let parent = element.all(by.tagName('app-name-parent')).get(0);

let hero = parent.all(by.tagName('app-name-child')).get(\_nonEmptyNameIndex);

let displayName = hero.element(by.tagName('h3')).getText();

expect(displayName).toEqual(\_nonEmptyName);

});

it('should replace empty name with default name', function () {

let \_emptyNameIndex = 1;

let \_defaultName = '"<no name set>"';

let parent = element.all(by.tagName('app-name-parent')).get(0);

let hero = parent.all(by.tagName('app-name-child')).get(\_emptyNameIndex);

let displayName = hero.element(by.tagName('h3')).getText();

expect(displayName).toEqual(\_defaultName);

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## Intercept input property changes with ngOnChanges()

Detect and act upon changes to input property values with the ngOnChanges() method of the [OnChanges](https://angular.io/api/core/OnChanges) lifecycle hook interface.

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

Learn about ngOnChanges() in the [Lifecycle Hooks](https://angular.io/guide/lifecycle-hooks) chapter.

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

component-interaction/src/app/version-child.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input), [OnChanges](https://angular.io/api/core/OnChanges), [SimpleChange](https://angular.io/api/core/SimpleChange) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-version-child',

template: `

<h3>[Version](https://angular.io/api/core/Version) {{major}}.{{minor}}</h3>

<h4>Change log:</h4>

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let change of changeLog">{{change}}</li>

</ul>

`

})

export class VersionChildComponent implements [OnChanges](https://angular.io/api/core/OnChanges) {

@[Input](https://angular.io/api/core/Input)() major: number;

@[Input](https://angular.io/api/core/Input)() minor: number;

changeLog: string[] = [];

ngOnChanges(changes: {[propKey: string]: [SimpleChange](https://angular.io/api/core/SimpleChange)}) {

let log: string[] = [];

for (let propName in changes) {

let changedProp = changes[propName];

let to = JSON.stringify(changedProp.currentValue);

if (changedProp.isFirstChange()) {

log.push(`Initial value of ${propName} set to ${to}`);

} else {

let from = JSON.stringify(changedProp.previousValue);

log.push(`${propName} changed from ${from} to ${to}`);

}

}

this.changeLog.push(log.join(', '));

}

}

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

component-interaction/src/app/version-parent.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-version-parent',

template: `

<h2>Source code version</h2>

<button (click)="newMinor()">New minor version</button>

<button (click)="newMajor()">New major version</button>

<app-version-child [major]="major" [minor]="minor"></app-version-child>

`

})

export class VersionParentComponent {

major = 1;

minor = 23;

newMinor() {

this.minor++;

}

newMajor() {

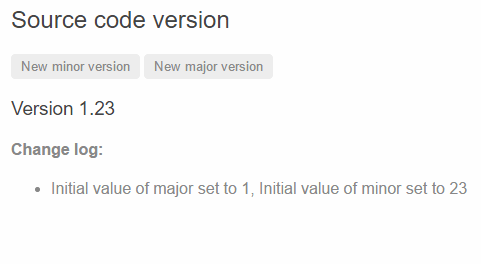
this.major++;

this.minor = 0;

}

}

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:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

// Test must all execute in this exact order

it('should set expected initial values', function () {

let actual = getActual();

let initialLabel = '[Version](https://angular.io/api/core/Version) 1.23';

let initialLog = 'Initial value of major set to 1, Initial value of minor set to 23';

expect(actual.label).toBe(initialLabel);

expect(actual.count).toBe(1);

expect(actual.logs.get(0).getText()).toBe(initialLog);

});

it('should set expected values after clicking \'Minor\' twice', function () {

let repoTag = element(by.tagName('app-version-parent'));

let newMinorButton = repoTag.all(by.tagName('button')).get(0);

newMinorButton.click().then(function() {

newMinorButton.click().then(function() {

let actual = getActual();

let labelAfter2Minor = '[Version](https://angular.io/api/core/Version) 1.25';

let logAfter2Minor = 'minor changed from 24 to 25';

expect(actual.label).toBe(labelAfter2Minor);

expect(actual.count).toBe(3);

expect(actual.logs.get(2).getText()).toBe(logAfter2Minor);

});

});

});

it('should set expected values after clicking \'Major\' once', function () {

let repoTag = element(by.tagName('app-version-parent'));

let newMajorButton = repoTag.all(by.tagName('button')).get(1);

newMajorButton.click().then(function() {

let actual = getActual();

let labelAfterMajor = '[Version](https://angular.io/api/core/Version) 2.0';

let logAfterMajor = 'major changed from 1 to 2, minor changed from 25 to 0';

expect(actual.label).toBe(labelAfterMajor);

expect(actual.count).toBe(4);

expect(actual.logs.get(3).getText()).toBe(logAfterMajor);

});

});

function getActual() {

let versionTag = element(by.tagName('app-version-child'));

let label = versionTag.element(by.tagName('h3')).getText();

let ul = versionTag.element((by.tagName('ul')));

let logs = ul.all(by.tagName('li'));

return {

label: label,

logs: logs,

count: logs.count()

};

}

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## Parent listens for child event

The child component exposes an [EventEmitter](https://angular.io/api/core/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](https://angular.io/api/core/EventEmitter) property is an output property, typically adorned with an [@Output decoration](https://angular.io/guide/template-syntax#inputs-outputs) as seen in this VoterComponent:

component-interaction/src/app/voter.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [EventEmitter](https://angular.io/api/core/EventEmitter), [Input](https://angular.io/api/core/Input), [Output](https://angular.io/api/core/Output) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-voter',

template: `

<h4>{{name}}</h4>

<button (click)="vote(true)" [disabled]="didVote">Agree</button>

<button (click)="vote(false)" [disabled]="didVote">Disagree</button>

`

})

export class VoterComponent {

@[Input](https://angular.io/api/core/Input)() name: string;

@[Output](https://angular.io/api/core/Output)() voted = new [EventEmitter](https://angular.io/api/core/EventEmitter)<boolean>();

didVote = false;

vote(agreed: boolean) {

this.voted.emit(agreed);

this.didVote = true;

}

}

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.

component-interaction/src/app/votetaker.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-vote-taker',

template: `

<h2>Should mankind colonize the Universe?</h2>

<h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>

<app-voter \*[ngFor](https://angular.io/api/common/NgForOf)="let voter of voters"

[name]="voter"

(voted)="onVoted($event)">

</app-voter>

`

})

export class VoteTakerComponent {

agreed = 0;

disagreed = 0;

voters = ['Narco', 'Celeritas', 'Bombasto'];

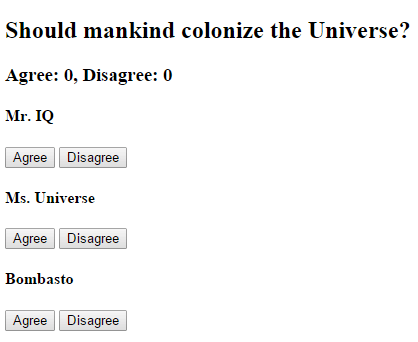
onVoted(agreed: boolean) {

agreed ? this.agreed++ : this.disagreed++;

}

}

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:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should not emit the event initially', function () {

let voteLabel = element(by.tagName('app-vote-taker'))

.element(by.tagName('h3')).getText();

expect(voteLabel).toBe('Agree: 0, Disagree: 0');

});

it('should process Agree vote', function () {

let agreeButton1 = element.all(by.tagName('app-voter')).get(0)

.all(by.tagName('button')).get(0);

agreeButton1.click().then(function() {

let voteLabel = element(by.tagName('app-vote-taker'))

.element(by.tagName('h3')).getText();

expect(voteLabel).toBe('Agree: 1, Disagree: 0');

});

});

it('should process Disagree vote', function () {

let agreeButton1 = element.all(by.tagName('app-voter')).get(1)

.all(by.tagName('button')).get(1);

agreeButton1.click().then(function() {

let voteLabel = element(by.tagName('app-vote-taker'))

.element(by.tagName('h3')).getText();

expect(voteLabel).toBe('Agree: 1, Disagree: 1');

});

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## Parent interacts with child via local variable

A parent component cannot use data binding to read child properties or invoke child methods. You can 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.

The following is a child CountdownTimerComponent that repeatedly counts down to zero and launches a rocket. It has start and stop methods that control the clock and it displays a countdown status message in its own template.

component-interaction/src/app/countdown-timer.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [OnDestroy](https://angular.io/api/core/OnDestroy), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-countdown-timer',

template: '<p>{{message}}</p>'

})

export class CountdownTimerComponent implements [OnInit](https://angular.io/api/core/OnInit), [OnDestroy](https://angular.io/api/core/OnDestroy) {

intervalId = 0;

message = '';

seconds = 11;

clearTimer() { clearInterval(this.intervalId); }

ngOnInit() { this.start(); }

ngOnDestroy() { this.clearTimer(); }

start() { this.countDown(); }

stop() {

this.clearTimer();

this.message = `Holding at T-${this.seconds} seconds`;

}

private countDown() {

this.clearTimer();

this.intervalId = window.setInterval(() => {

this.seconds -= 1;

if (this.seconds === 0) {

this.message = 'Blast off!';

} else {

if (this.seconds < 0) { this.seconds = 10; } // reset

this.message = `T-${this.seconds} seconds and counting`;

}

}, 1000);

}

}

The CountdownLocalVarParentComponent that hosts the timer component is as follows:

component-interaction/src/app/countdown-parent.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

import { CountdownTimerComponent } from './countdown-timer.component';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-countdown-parent-lv',

template: `

<h3>Countdown to Liftoff (via local variable)</h3>

<button (click)="timer.start()">Start</button>

<button (click)="timer.stop()">Stop</button>

<div class="seconds">{{timer.seconds}}</div>

<app-countdown-timer #timer></app-countdown-timer>

`,

styleUrls: ['../assets/demo.css']

})

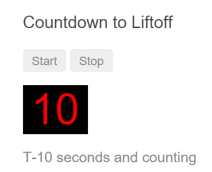
export class CountdownLocalVarParentComponent { }

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

You can 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 we see the parent and child working together.



### 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:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('timer and parent seconds should match', function () {

let parent = element(by.tagName(parentTag));

let message = parent.element(by.tagName('app-countdown-timer')).getText();

browser.sleep(10); // give `seconds` a chance to catchup with `message`

let seconds = parent.element(by.className('seconds')).getText();

expect(message).toContain(seconds);

});

it('should stop the countdown', function () {

let parent = element(by.tagName(parentTag));

let stopButton = parent.all(by.tagName('button')).get(1);

stopButton.click().then(function() {

let message = parent.element(by.tagName('app-countdown-timer')).getText();

expect(message).toContain('Holding');

});

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## Parent calls an @ViewChild()

The local variable approach is simple and easy. 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 an instance of the parent component class must read or write child component values or must call child component 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](https://angular.io/guide/component-interaction#countdown-timer-example) example. Neither its appearance nor its behavior will change. The child [CountdownTimerComponent](https://angular.io/guide/component-interaction#countdown-timer-example) 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:

component-interaction/src/app/countdown-parent.component.ts

content\_copyimport { [AfterViewInit](https://angular.io/api/core/AfterViewInit), [ViewChild](https://angular.io/api/core/ViewChild) } from '@angular/core';

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

import { CountdownTimerComponent } from './countdown-timer.component';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-countdown-parent-vc',

template: `

<h3>Countdown to Liftoff (via [ViewChild](https://angular.io/api/core/ViewChild))</h3>

<button (click)="start()">Start</button>

<button (click)="stop()">Stop</button>

<div class="seconds">{{ seconds() }}</div>

<app-countdown-timer></app-countdown-timer>

`,

styleUrls: ['../assets/demo.css']

})

export class CountdownViewChildParentComponent implements [AfterViewInit](https://angular.io/api/core/AfterViewInit) {

@[ViewChild](https://angular.io/api/core/ViewChild)(CountdownTimerComponent)

private timerComponent: CountdownTimerComponent;

seconds() { return 0; }

ngAfterViewInit() {

// Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...

// but wait a [tick](https://angular.io/api/core/testing/tick) first to avoid one-time devMode

// unidirectional-data-flow-violation error

setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);

}

start() { this.timerComponent.start(); }

stop() { this.timerComponent.stop(); }

}

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](https://angular.io/api/core/ViewChild) decorator and the [AfterViewInit](https://angular.io/api/core/AfterViewInit) lifecycle hook.

Next, inject the child CountdownTimerComponent into the private timerComponent property via the @[ViewChild](https://angular.io/api/core/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 app has to 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](https://angular.io/guide/component-interaction#countdown-tests) as before.

[Back to top](https://angular.io/guide/component-interaction#top)

## Parent and children communicate via 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.

component-interaction/src/app/mission.service.ts

content\_copyimport { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

import { Subject } from 'rxjs';

@[Injectable](https://angular.io/api/core/Injectable)()

export class MissionService {

// Observable string sources

private missionAnnouncedSource = new Subject<string>();

private missionConfirmedSource = new Subject<string>();

// Observable string streams

missionAnnounced$ = this.missionAnnouncedSource.asObservable();

missionConfirmed$ = this.missionConfirmedSource.asObservable();

// Service message commands

announceMission(mission: string) {

this.missionAnnouncedSource.next(mission);

}

confirmMission(astronaut: string) {

this.missionConfirmedSource.next(astronaut);

}

}

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:

component-interaction/src/app/missioncontrol.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

import { MissionService } from './mission.service';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-mission-control',

template: `

<h2>Mission Control</h2>

<button (click)="announce()">Announce mission</button>

<app-astronaut \*[ngFor](https://angular.io/api/common/NgForOf)="let astronaut of astronauts"

[astronaut]="astronaut">

</app-astronaut>

<h3>History</h3>

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let event of history">{{event}}</li>

</ul>

`,

providers: [MissionService]

})

export class MissionControlComponent {

astronauts = ['Lovell', 'Swigert', 'Haise'];

history: string[] = [];

missions = ['Fly to the moon!',

'Fly to mars!',

'Fly to Vegas!'];

nextMission = 0;

constructor(private missionService: MissionService) {

missionService.missionConfirmed$.subscribe(

astronaut => {

this.history.push(`${astronaut} confirmed the mission`);

});

}

announce() {

let mission = this.missions[this.nextMission++];

this.missionService.announceMission(mission);

this.history.push(`Mission "${mission}" announced`);

if (this.nextMission >= this.missions.length) { this.nextMission = 0; }

}

}

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:

component-interaction/src/app/astronaut.component.ts

content\_copyimport { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input), [OnDestroy](https://angular.io/api/core/OnDestroy) } from '@angular/core';

import { MissionService } from './mission.service';

import { Subscription } from 'rxjs';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-astronaut',

template: `

<p>

{{astronaut}}: <strong>{{mission}}</strong>

<button

(click)="confirm()"

[disabled]="!announced || confirmed">

Confirm

</button>

</p>

`

})

export class AstronautComponent implements [OnDestroy](https://angular.io/api/core/OnDestroy) {

@[Input](https://angular.io/api/core/Input)() astronaut: string;

mission = '<no mission announced>';

confirmed = false;

announced = false;

subscription: Subscription;

constructor(private missionService: MissionService) {

this.subscription = missionService.missionAnnounced$.subscribe(

mission => {

this.mission = mission;

this.announced = true;

this.confirmed = false;

});

}

confirm() {

this.confirmed = true;

this.missionService.confirmMission(this.astronaut);

}

ngOnDestroy() {

// prevent memory leak when component destroyed

this.subscription.unsubscribe();

}

}

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 app because the lifetime of a AstronautComponent is the same as the lifetime of the app 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:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should announce a mission', function () {

let missionControl = element(by.tagName('app-mission-control'));

let announceButton = missionControl.all(by.tagName('button')).get(0);

announceButton.click().then(function () {

let history = missionControl.all(by.tagName('li'));

expect(history.count()).toBe(1);

expect(history.get(0).getText()).toMatch(/Mission.\* announced/);

});

});

it('should confirm the mission by Lovell', function () {

testConfirmMission(1, 2, 'Lovell');

});

it('should confirm the mission by Haise', function () {

testConfirmMission(3, 3, 'Haise');

});

it('should confirm the mission by Swigert', function () {

testConfirmMission(2, 4, 'Swigert');

});

function testConfirmMission(buttonIndex: number, expectedLogCount: number, astronaut: string) {

let \_confirmedLog = ' confirmed the mission';

let missionControl = element(by.tagName('app-mission-control'));

let confirmButton = missionControl.all(by.tagName('button')).get(buttonIndex);

confirmButton.click().then(function () {

let history = missionControl.all(by.tagName('li'));

expect(history.count()).toBe(expectedLogCount);

expect(history.get(expectedLogCount - 1).getText()).toBe(astronaut + \_confirmedLog);

});

}

// ...