`@Input()` and `@Output()` properties

Contents ›

`@Input()` and `@Output()` allow Angular to share data between the parent context and child directives or components. An `@Input()` property is writable while an `@Output()` property is observable.

> See the live example / download example for a working example containing the code snippets in this guide.

Consider this example of a child/parent relationship:

```
<parent-component>
  <child-component></child-component>
</parent-component>
```

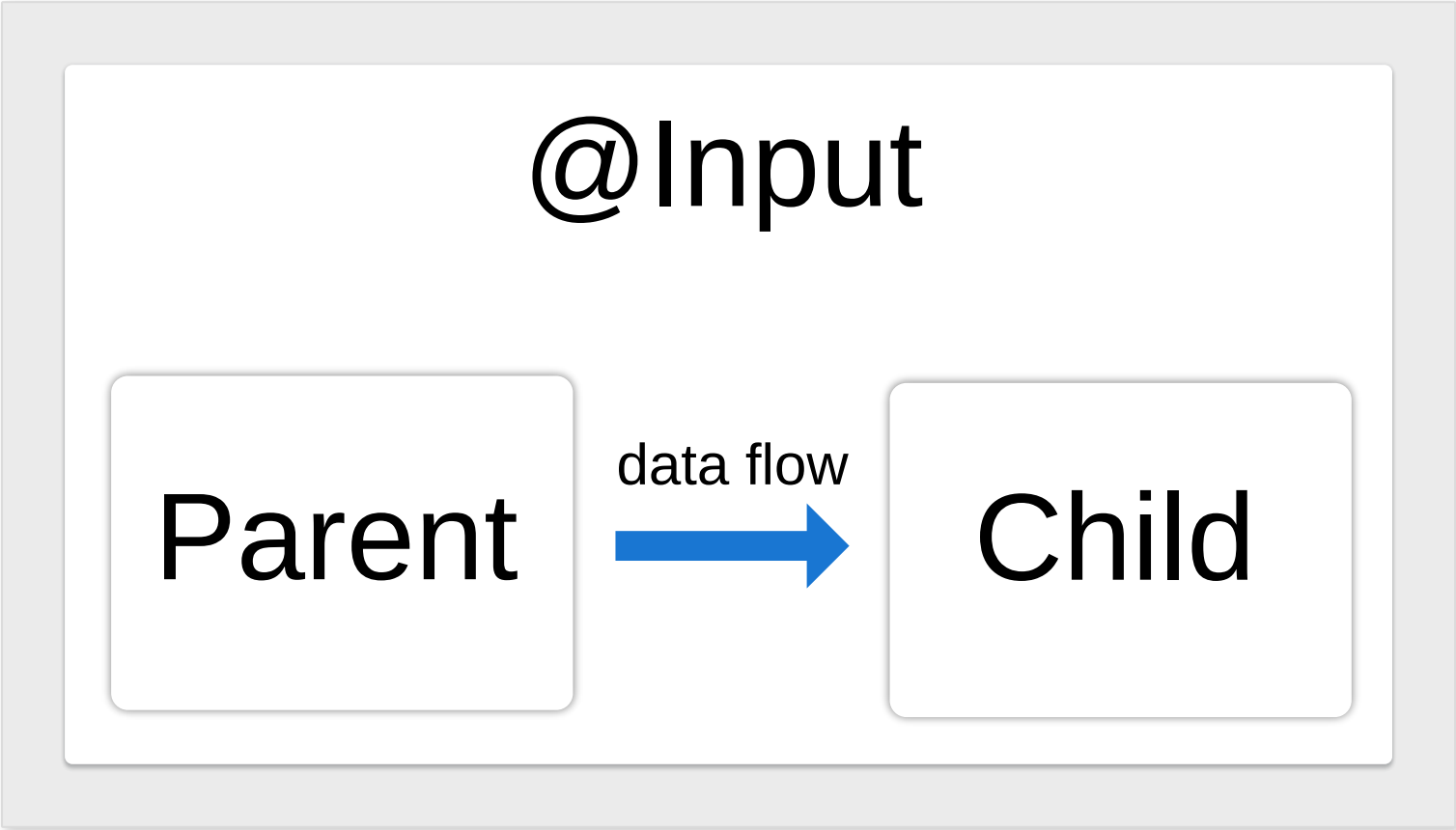
Here, the `<child-component>` selector, or child directive, is embedded within a `<parent-component>`, which serves as the child's context.

`@Input()` and `@Output()` act as the API, or application programming interface, of the child component in that they allow the child to communicate with the parent. Think of `@Input()` and `@Output()` like ports or doorways—`@Input()` is the doorway into the component allowing data to flow in while `@Output()` is the doorway out of the component, allowing the child component to send data out.

> ### `@Input()` and `@Output()` are independent
>
> Though `@Input()` and `@Output()` often appear together in apps, you can use them separately. If the nested component is such that it only needs to send data to its parent, you wouldn't need an `@Input()`, only an `@Output()`. The reverse is also true in that if the child only needs to receive data from the parent, you'd only need `@Input()`.

How to use `@Input()`

Use the `@Input()` decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component. It helps to remember that the data flow is from the perspective of the child component. So an `@Input()` allows data to be input *into* the child component from the parent component.



To illustrate the use of `@Input()`, edit these parts of your app:

- The child component class and template
- The parent component class and template

In the child

To use the `@Input()` decorator in a child component class, first import `Input` and then decorate the property with `@Input()`:

src/app/item-detail/item-detail.component.ts

```
import { Component, Input } from '@angular/core'; // First, import Input
export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}
```

In this case, `@Input()` decorates the property `item`, which has a type of `string`, however, `@Input()` properties can have any type, such as `number`, `string`, `boolean`, or `object`. The value for `item` will come from the parent component, which the next section covers.

Next, in the child component template, add the following:

src/app/item-detail/item-detail.component.html

```
<p>
  Today's item: {{item}}
</p>
```

In the parent

The next step is to bind the property in the parent component's template. In this example, the parent component template is `app.component.html`.

First, use the child's selector, here `<app-item-detail>`, as a directive within the parent component template. Then, use property binding to bind the property in the child to the property of the parent.

src/app/app.component.html

```
<app-item-detail [item]="currentItem"></app-item-detail>
```

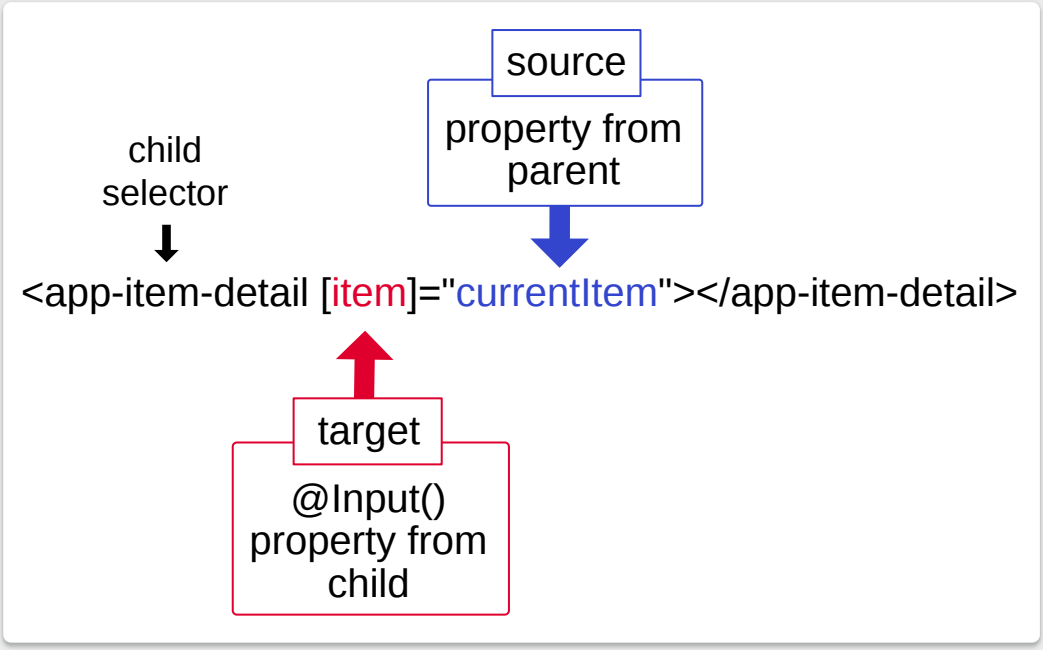
Next, in the parent component class, `app.component.ts`, designate a value for `currentItem`:

src/app/app.component.ts

```
export class AppComponent {
  currentItem = 'Television';
}
```

With `@Input()`, Angular passes the value for `currentItem` to the child so that `item` renders as `Television`.

The following diagram shows this structure:



The target in the square brackets, `[]`, is the property you decorate with `@Input()` in the child component. The binding source, the part to the right of the equal sign, is the data that the parent component passes to the nested

component.

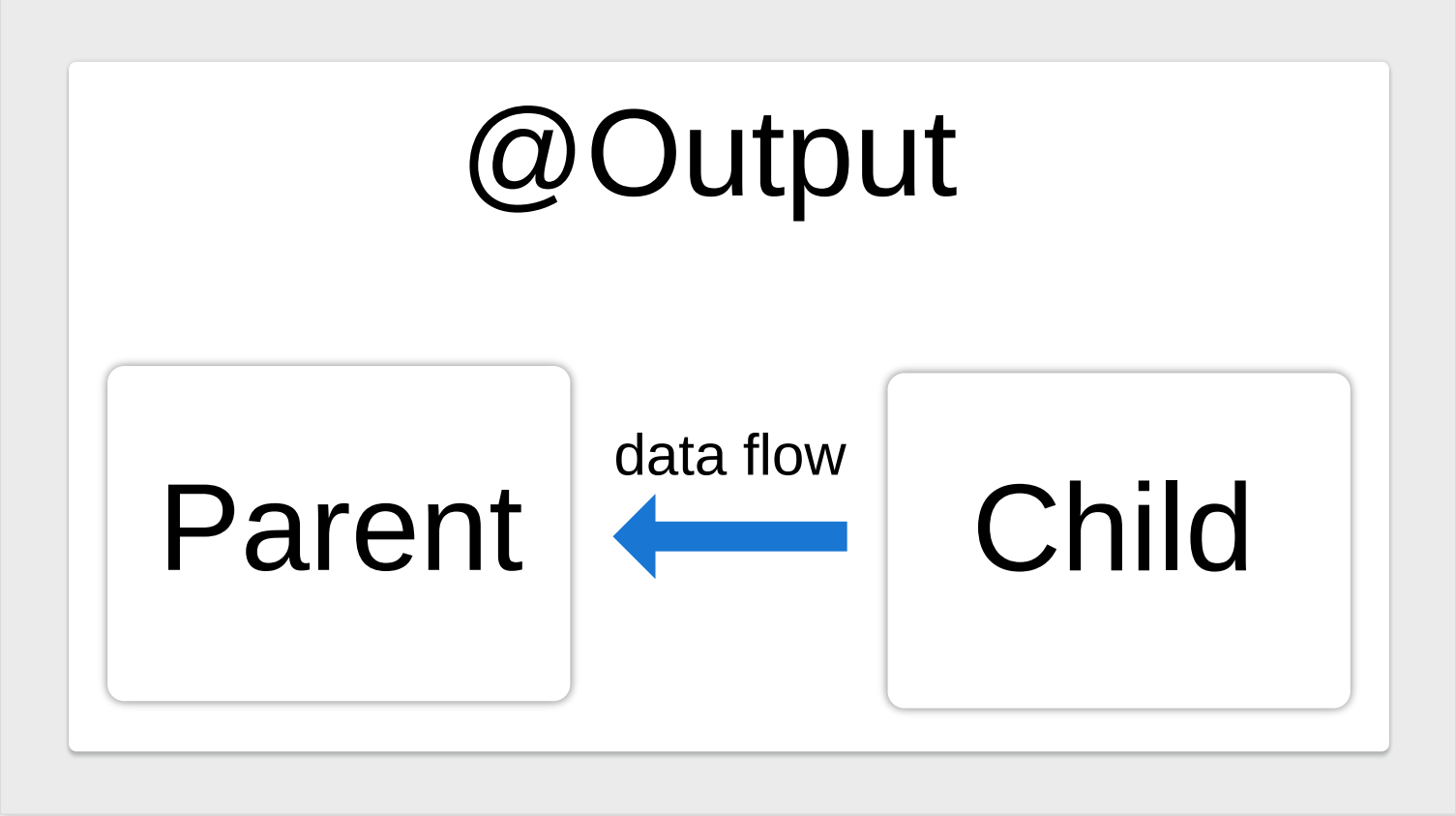
The key takeaway is that when binding to a child component's property in a parent component—that is, what's in square brackets—you must decorate the property with `@Input()` in the child component.

> ### `OnChanges` and `@Input()`
>
> To watch for changes on an `@Input()` property, use `OnChanges`, one of Angular's lifecycle hooks. `OnChanges` is specifically designed to work with properties that have the `@Input()` decorator. See the `OnChanges` section of the Lifecycle Hooks guide for more details and examples.

How to use `@Output()`

Use the `@Output()` decorator in the child component or directive to allow data to flow from the child *out* to the parent.

An `@Output()` property should normally be initialized to an Angular `EventEmitter` with values flowing out of the component as events.



Just like with `@Input()`, you can use `@Output()` on a property of the child component but its type should be `EventEmitter`.

`@Output()` marks a property in a child component as a doorway through which data can travel from the child to the parent. The child component then has to raise an event so the parent knows something has changed. To raise an event, `@Output()` works hand in hand with `EventEmitter`, which is a class in `@angular/core` that you use to emit custom events.

When you use `@Output()`, edit these parts of your app:

- The child component class and template
- The parent component class and template

The following example shows how to set up an `@Output()` in a child component that pushes data you enter in an HTML `<input>` to an array in the parent component.

> The HTML element `<input>` and the Angular decorator `@Input()` are different. This documentation is about component communication in Angular as it pertains to `@Input()` and `@Output()`. For more information on the HTML element `<input>`, see the W3C Recommendation.

In the child

This example features an `<input>` where a user can enter a value and click a `<button>` that raises an event. The `EventEmitter` then relays the data to the parent component.

First, be sure to import `Output` and `EventEmitter` in the child component class:

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

Next, still in the child, decorate a property with `@Output()` in the component class. The following example `@Output()` is called `newItemEvent` and its type is `EventEmitter`, which means it's an event.

src/app/item-output/item-output.component.ts

```
@Output() newItemEvent = new EventEmitter<string>();
```

The different parts of the above declaration are as follows:

- `@Output()`—a decorator function marking the property as a way for data to go from the child to the parent
- `newItemEvent`—the name of the `@Output()`
- `EventEmitter<string>`—the `@Output()`'s type
- `new EventEmitter<string>()`—tells Angular to create a new event emitter and that the data it emits is of type string. The type could be any type, such as `number`, `boolean`, and so on. For more information on `EventEmitter`, see the EventEmitter API documentation.

Next, create an `addNewItem()` method in the same component class:

src/app/item-output/item-output.component.ts

```
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();
```

```
  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

The `addNewItem()` function uses the `@Output()`, `newItemEvent`, to raise an event in which it emits the value the user types into the `<input>`. In other words, when the user clicks the add button in the UI, the child lets the parent know about the event and gives that data to the parent.

In the child's template

The child's template has two controls. The first is an HTML `<input>` with a template reference variable , `#newItem`, where the user types in an item name. Whatever the user types into the `<input>` gets stored in the `#newItem` variable.

src/app/item-output/item-output.component.html

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

The second element is a `<button>` with an event binding. You know it's an event binding because the part to the left of the equal sign is in parentheses, `(click)`.

The `(click)` event is bound to the `addNewItem()` method in the child component class which takes as its argument whatever the value of `#newItem` is.

Now the child component has an `@Output()` for sending data to the parent and a method for raising an event. The next step is in the parent.

In the parent

In this example, the parent component is `AppComponent`, but you could use any component in which you could nest the child.

The `AppComponent` in this example features a list of `items` in an array and a method for adding more items to the array.

src/app/app.component.ts

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

The `addItem()` method takes an argument in the form of a string and then pushes, or adds, that string to the `items` array.

In the parent's template

Next, in the parent's template, bind the parent's method to the child's event. Put the child selector, here `<app-item-output>`, within the parent component's template, `app.component.html`.

src/app/app.component.html

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

The event binding, `(newItemEvent)='addItem($event)'`, tells Angular to connect the event in the child, `newItemEvent`, to the method in the parent, `addItem()`, and that the event that the child is notifying the parent about is to be the argument of `addItem()`. In other words, this is where the actual hand off of data takes place. The `$event` contains the data that the user types into the `<input>` in the child template UI.

Now, in order to see the `@Output()` working, add the following to the parent's template:

```
<ul>
  <li *ngFor="let item of items">{{item}}</li>
</ul>
```

The `*ngFor` iterates over the items in the `items` array. When you enter a value in the child's `<input>` and click the button, the child emits the event and the parent's `addItem()` method pushes the value to the `items` array and it renders in the list.

`@Input()` and `@Output()` together

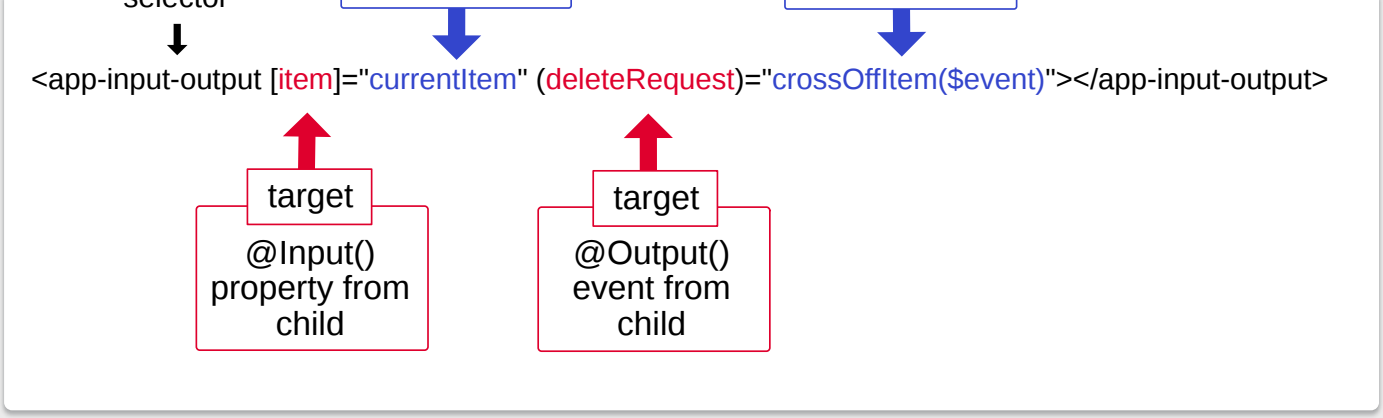
You can use `@Input()` and `@Output()` on the same child component as in the following:

src/app/app.component.html

```
<app-input-output [item]="currentItem" (deleteRequest)="crossOffItem($event)"></app-input-output>
```

The target, `item`, which is an `@Input()` property in the child component class, receives its value from the parent's property, `currentItem`. When you click delete, the child component raises an event, `deleteRequest`, which is the argument for the parent's `crossOffItem()` method.

The following diagram is of an `@Input()` and an `@Output()` on the same child component and shows the different parts of each:



As the diagram shows, use inputs and outputs together in the same manner as using them separately. Here, the child selector is `<app-input-output>` with `item` and `deleteRequest` being `@Input()` and `@Output()` properties in the child component class. The property `currentItem` and the method `crossOffItem()` are both in the parent component class.

To combine property and event bindings using the banana-in-a-box syntax, `[()]`, see Two-way Binding.

`@Input()` and `@Output()` declarations

Instead of using the `@Input()` and `@Output()` decorators to declare inputs and outputs, you can identify members in the `inputs` and `outputs` arrays of the directive metadata, as in this example:

src/app/in-the-metadata/in-the-metadata.component.ts

```
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
inputs: ['clearanceItem'],
outputs: ['buyEvent']
// tslint:enable: no-inputs-metadata-property no-outputs-metadata-property
```

While declaring `inputs` and `outputs` in the `@Directive` and `@Component` metadata is possible, it is a better practice to use the `@Input()` and `@Output()` class decorators instead, as follows:

src/app/input-output/input-output.component.ts

```
@Input() item: string;
@Output() deleteRequest = new EventEmitter<string>();
```

See the Decorate input and output properties section of the Style Guide for details.

If you get a template parse error when trying to use inputs or outputs, but you know that the properties do indeed exist, double check that your properties are annotated with `@Input()` / `@Output()` or that you've declared them in an `inputs/outputs` array:

```
Uncaught Error: Template parse errors:
Can't bind to 'item' since it isn't a known property of 'app-item-detail'
```

Aliasing inputs and outputs

Sometimes the public name of an input/output property should be different from the internal name. While it is a best practice to avoid this situation, Angular does offer a solution.

Aliasing in the metadata

Alias inputs and outputs in the metadata using a colon-delimited (:) string with the directive property name on the left and the public alias on the right:

src/app/aliasing/aliasing.component.ts

```
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
inputs: ['input1: saveForLaterItem'], // propertyName:alias
outputs: ['outputEvent1: saveForLaterEvent']
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
```

Aliasing with the `@Input()`/`@Output()` decorator

You can specify the alias for the property name by passing the alias name to the `@Input()`/`@Output()` decorator. The internal name remains as usual.

src/app/aliasing/aliasing.component.ts

```
@Input('wishListItem') input2: string; //  @Input(alias)
@Output('wishEvent') outputEvent2 = new EventEmitter<string>(); //  @Output(alias)
propertyName = ...
```


Super-powered by Google ©2010-2020. Code licensed under an MIT-style License. Documentation licensed under CC BY 4.0.

Version 10.0.10-local+sha.b32126c335.