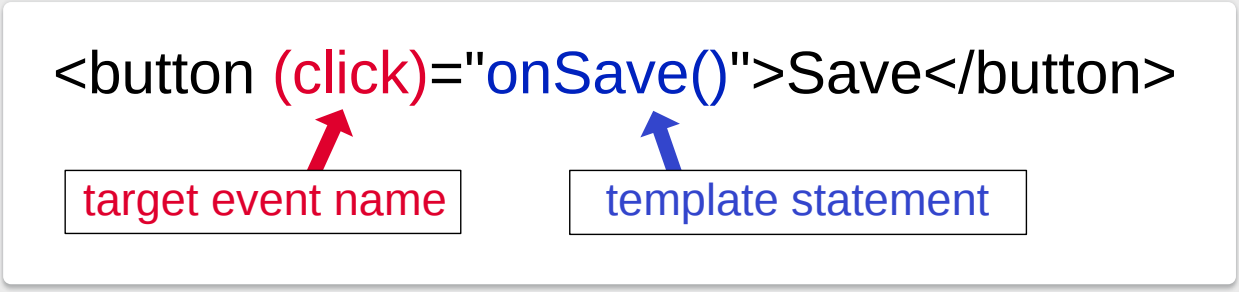
Event binding `(event)`

Contents

Event binding allows you to listen for certain events such as keystrokes, mouse movements, clicks, and touches.

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

Angular event binding syntax consists of a **target event** name within parentheses on the left of an equal sign, and a quoted template statement on the right. The following event binding listens for the button's click events, calling the component's `onSave()` method whenever a click occurs:



Target event

As above, the target is the button's click event.

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
```

Alternatively, use the `on-` prefix, known as the canonical form:

src/app/app.component.html

```
<button on-click="onSave($event)">on-click Save</button>
```

Element events may be the more common targets, but Angular looks first to see if the name matches an event property of a known directive, as it does in the following example:

src/app/app.component.html

```
<h4>myClick is an event on the custom ClickDirective:</h4>
<button (myClick)="clickMessage=$event" clickable>click with myClick</button>
{{clickMessage}}
```

If the name fails to match an element event or an output property of a known directive, Angular reports an “unknown directive” error.

$event and event handling statements

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

The binding conveys information about the event. This information can include data values such as an event object, string, or number named `$event`.

The target event determines the shape of the `$event` object. If the target event is a native DOM element event, then `$event` is a DOM event object, with properties such as `target` and `target.value`.

Consider this example:

src/app/app.component.html

```
<input [value]="currentItem.name"
       (input)="currentItem.name=$event.target.value" >
without NgModel
```

This code sets the `<input>` `value` property by binding to the `name` property. To listen for changes to the value, the code binds to the `input` event of the `<input>` element. When the user makes changes, the `input` event is raised, and the binding executes the statement within a context that includes the DOM event object, `$event`.

To update the `name` property, the changed text is retrieved by following the path `$event.target.value`.

If the event belongs to a directive—recall that components are directives—`$event` has whatever shape the directive produces.

Custom events with `EventEmitter`

Directives typically raise custom events with an Angular EventEmitter. The directive creates an `EventEmitter` and exposes it as a property. The directive calls `EventEmitter.emit(payload)` to fire an event, passing in a message

payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the `$event` object.

Consider an `ItemDetailComponent` that presents item information and responds to user actions. Although the `ItemDetailComponent` has a delete button, it doesn't know how to delete the hero. It can only raise an event reporting the user's delete request.

Here are the pertinent excerpts from that `ItemDetailComponent`:

src/app/item-detail/item-detail.component.html (template)

```
<img src="{{itemImageUrl}}" [style.display]="displayNone">
<span [style.text-decoration]="lineThrough">{{ item.name }}
</span>
<button (click)="delete()">Delete</button>
```

src/app/item-detail/item-detail.component.ts (deleteRequest)

```
// This component makes a request but it can't actually delete a hero.
@Output() deleteRequest = new EventEmitter<Item>();

delete() {
  this.deleteRequest.emit(this.item);
  this.displayNone = this.displayNone ? '' : 'none';
  this.lineThrough = this.lineThrough ? '' : 'line-through';
}
```

The component defines a `deleteRequest` property that returns an `EventEmitter`. When the user clicks *delete*, the component invokes the `delete()` method, telling the `EventEmitter` to emit an `Item` object.

Now imagine a hosting parent component that binds to the `deleteRequest` event of the `ItemDetailComponent`.

src/app/app.component.html (event-binding-to-component)

```
<app-item-detail (deleteRequest)="deleteItem($event)" [item]="currentItem"></app-item-
detail>
```

When the `deleteRequest` event fires, Angular calls the parent component's `deleteItem()` method, passing the *item-to-delete* (emitted by `ItemDetail`) in the `$event` variable.

Template statements have side effects

Though template expressions shouldn't have side effects, template statements usually do. The `deleteItem()` method does have a side effect: it deletes an item.

Deleting an item updates the model, and depending on your code, triggers other changes including queries and saving to a remote server. These changes propagate through the system and ultimately display in this and other views.

Version 10.0.10-local+sha.b32126c335.