

Create a Component

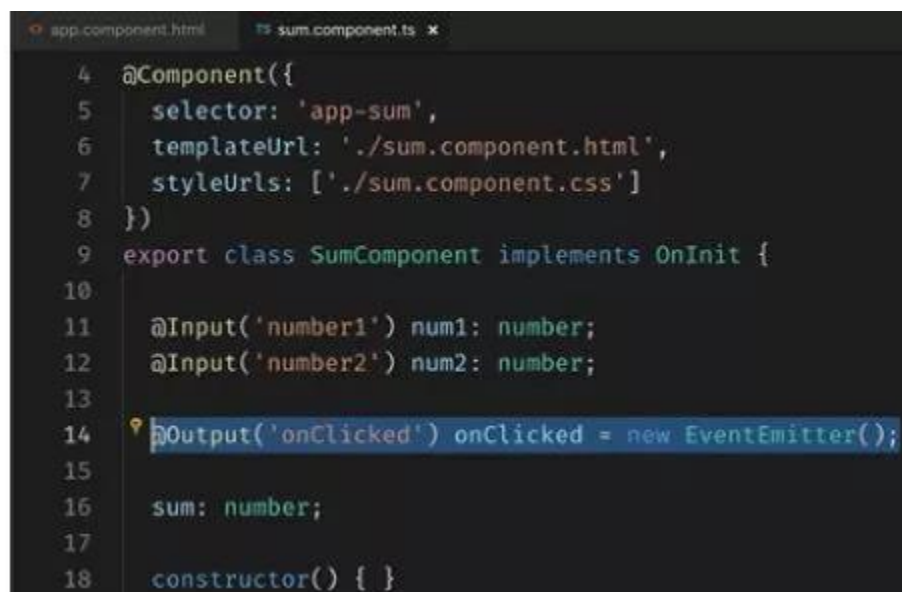
To be able to pass an input to a component, we need to write code inside the component to make it capable of receiving input. **We use the @Input decorator for this purpose. On the other hand, to pass in an input to the component, we simply use property binding.**

Before we start writing the code, let's create a new component using the Angular CLI. You can do this manually as well but why go through all the trouble when we have a more efficient method which is error-proof. Use the following command to create a new component called Sum.

```
ng generate component sum
```

You can also use the shorthand, `ng g c sum` which does the same thing. Once the command is successful, you should see the following messages.

So a folder with a few files has been created in the project folder. Let's look at the folder structure and understand how it has changed in the process.



```
1  app.component.html  sum.component.ts x
4  @Component({
5    selector: 'app-sum',
6    templateUrl: './sum.component.html',
7    styleUrls: ['./sum.component.css']
8  })
9  export class SumComponent implements OnInit {
10
11    @Input('number1') num1: number;
12    @Input('number2') num2: number;
13
14    @Output('onClicked') onClicked = new EventEmitter();
15
16    sum: number;
17
18    constructor() { }
```

The files shown in the green are the ones that have been created. The sum folder with the component files has been created. Along with that, app.module.ts file has been modified as an entry for the new component, the SumComponent has been added to the declarations array of the main NgModule, the AppModule.

```
sum.component.html x
1 <p (click)="clicked()">
2   Sum of {{ num1 }} and {{ num2 }} is {{ sum }}.
3 </p>
```

Now, we are not seeing any changes in the app-routing.module.ts file because we are not using Angular Routing as of now, but if we were, a new route for the new component would have been added to this file as well, automatically.

```
24 clicked(){
25   this.onClicked.emit("User clicked on the p-element");
26 }
```

All the above is done by the Angular CLI, all we did was execute a command. That's how awesome the CLI is. Saves time and ensures no mistakes.

```
16   onClicked() {
17     console.log("User clicked at " + new Date())
18   }
```

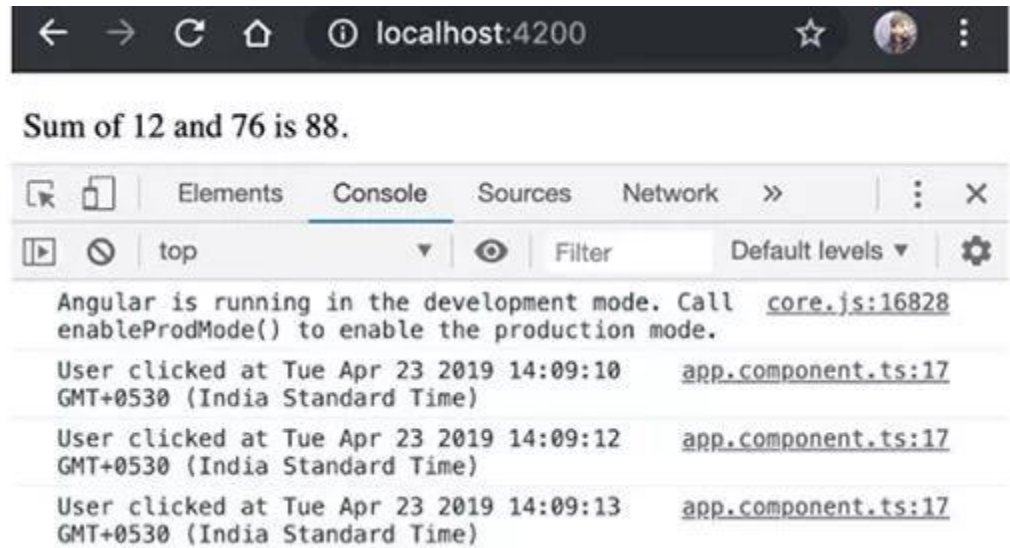
Now that we have the new component ready, let's use the component in the app. Open up the app.component.ts file and replace everything in the file with the following code.

```
<app-sum></app-sum>
```

The app-sum here is the component selector. A component's selector is used to render the component within another component or in the HTML page.

This code will render the SumComponent component in the application. That's actually how we use components within components.

Save everything and you should see the application refresh in the browser and the following preview.



Component Inputs

Now if you open sum.component.html file (which is inside the sum folder), you should see the same HTML code in the file. That confirms that the SumComponent is what we see in the application preview in the browser.

Moving on, let's try to pass in two inputs, both numbers, number1 and number2 to this component. To do that, we simply pass inputs as attributes within the component element.

With the above code, we are simply passing in two inputs, namely number1 and number2, to the component. Please note that we need to use the property binding syntax to ensure that the inputs are interpreted properly.

Next, we need to receive these inputs within the component so that we can use these input values in our code and render out the output. To do that, we use the Input decorator. The Input decorator is a special TS class that allows you to receive inputs within a component.

First, import the Input decorator from the @angular/core package.

```
import { Component, OnInit, Input } from '@angular/core';
```

Next, we define the inputs using the same names that we used while passing in the inputs.

The @Input notation is the use of Input decorator. We are passing in the strings names of the inputs as string arguments to the decorator. Finally, within the SumComponent class, we can use num1 and num2 for number1 and number2, respectively.

Let's render out the sum of these two numbers in the component's HTML code. To do that, first we need to add the numbers and store them in a class variable.

We are writing the code to add the numbers in ngOnInit and not inside the constructor. The reason for this is that we can only access component inputs once the component has been completely initialized.

Finally, let's render out the sum in the component's HTML.

.

Now that we have a component, we can re-use this component as many times as we want.

That's all about component inputs for now.

Component Outputs

We just saw how Inputs allows components to receive information from the parent component and use in any way it wants. In a similar fashion, there may be cases where we want the parent component to know of an event inside the child component. In such a situation, a child component can emit events and the parent component can bind to such events using Event Binding.

Let's have a look at an example. We start with the same SumComponent component that we created in the last section and added Inputs to. We will define an output event within the same component. For simplicity, let's emit an event whenever a user clicks on the SumComponent. You may argue that we can simply the click event on the SumComponent but we will be creating our own event.

Let's import the Output decorator and EventEmitter class from the @angular/core package.

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

Next, we declare an output event using the Output decorator like

In the above code snippet, we have created a custom event by the name onClicked. Now we can emit this event from anywhere in our code. As of now, we will emit this event whenever the user clicks on the <p> element that contains the text in the SumComponent's HTML file. We use event binding to bind the click event to a method called clicked().

Next, we create this method in the component's class file.

Finally, we bind the `onClicked` event (of the `SumComponent`) to a method in the `AppComponent`.

```
<app-sum [number1]="12" [number2]="76" (onClicked)="onClicked()"
"></app-sum>
```

The `onClicked` event now exists on the `SumComponent` and we can bind methods to it. Here we are binding it to a method with the same name, `onClicked`. Let's create the method in `app.component.ts`.

In this method, we are logging out a simple message with the current date and time to the console.

If we now test the app, click on the component in the page, you should see a message logged to the console, everytime. This way, we are executing a method in the `AppComponent` when something happens (a click) in the `SumComponent`.

All this can be a little confusing but with a little practice and getting used to will help. Moreover, do not use `Outputs` and `Inputs` unless you feel the need to do it.

`Outputs` are used to notify other components about the events happening within a component. We can even pass in data to emit method that can be retrieved in the method bound to the event.

