Contents

Day01 Angular Components and Templates

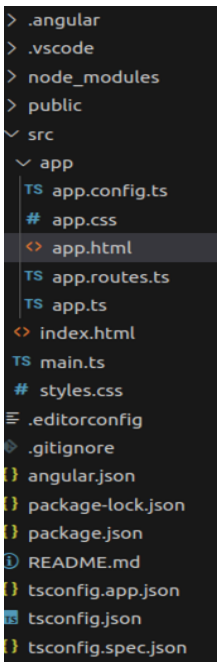
PART 01 – NG20 SETUP

This section assumes that you have already installed the latest Angular CLI. If you did not, please run the command **`npm install -g @angular/cli`** before proceeding. For this particular boot camp, I will use *skills20* as the app folder, but you may use just *skills* or any other name you wish. The image below is the current version on my OS.

```
Angular CLI: 20.3.8
Node: 20.19.5
Package Manager: npm 11.6.2
OS: linux x64
```

1. From your root folder (Documents in my case), open a terminal window (or tab) to and type the command **`ng new skills20.`** If you do not want to work with git or write tests, just add this flag after your **`ng`** command:
   `--skip-tests --skip-git`

2. If you are using Angular 20 you will be asked a series of questions.
   Choose plain **CSS** for styles. To choose CSS use the arrow keys on the keyboard, however CSS should be auto selected, just hit **Enter**.
   Choose **N** for Server-Side Rendering AND zoneless.
   Choose "None" when asked about AI tools (a future boot camp will include Claude)



3. CD into the skills20 directory and open a terminal window pointing to that folder. After typing **`ng serve`** in the terminal window, notice the word 'compiling', the TypeScript (ts) code is compiled in order to run successfully. (**`or ng serve -o`**)

4. Open the application in VS Code (or another editor) and most of our work will be in the **app** folder, which acts like the parent folder.

View the code of app.html, this is the file that feeds the default page that shows up on the browser at port 4200. Remove everything except the **** tag which has the code **<h1>Hello, {{ title() }}</h1>**And the **<router-outlet>** tags. (around line 233)

```
<h1>Hello, {{ title() }}</h1>
<p>Congratulations! Your app is running.</p>
<router-outlet />
```

Leave these three lines in app. html.

PART 02 – HTML ELEMENT OBJECTS

1. Just to orient ourselves, lets try to change the text at the top of the page, right now it says **Hello, skills20**. In app.ts file in the app folder, make the change highlighted in yellow:

```
export class App {
  protected readonly title = signal('Skills2025');
}
```

The browser should refresh and you will see the updated content. To stop the app, on the keyboard type **CTRL-C**

5. Angular like other JS frameworks, use components as a way to organize code. Lets begin by creating a **home** component. This must be done using the CLI, so in a terminal window:

```
ng g c components/home
```

This will also create a components folder

2. When creating a new component and when using the Angular CLI, it does provide a single line of content in the template (home.html), lets remove that line and replace it with the following lines:

```
<h2>Welcome Home</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ...</p>
<p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.</p>
```

Do this in the home.html file, usually referred to as the template. Feel free to shorten these paragraphs.

3. We now need to import that **Home** component we created in #2, we do this in the app.ts file:

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Home } from './components/home/home';
@Component({
```

1. Usually if we *import* a component or any other Angular class based file, there is usually always second action. That component (or service, or pipe or directive) needs to be inserted into the **imports[]** array:

```
@Component({
  selector: 'app-root',
  imports: [RouterOutlet, Home],
  templateUrl: './app.html',
```

2. Now we want to show this component on the browser, so in app.html add this line:

```
<h1>Hello, {{ title() }}</h1>
<p>Congratulations! Your app is running.</p>
<app-home></app-home>
<router-outlet />
```

The app.html file is in the root of your Angular app, inside the src folder. Now run the app and you should see the **Home** component as well as the initial lines we had when we started this app. Angular recently went to stand-alone components, so any components used must be imported using the imports array of the consumer component, App in this case.

3. Before leaving the app.ts file, add in the standalone key/value configuration in the @Component decorator function. This is for bootstrapping the application:

```
@Component({
  selector: 'app-home',
  standalone: true,
  imports: [],
  templateUrl: './home.html'
```

This requirement was added since NG15, but may change in the future.

4. Let's setup basic routing, so in the app.routes.ts file, add these lines:

```
import { Routes } from '@angular/router';
import { Home } from './components/home/home';
export const routes: Routes = [
  {path:'', component:Home},
  {path:'home', redirectTo: '', pathMatch: 'full'}
];
```

Here I am using the **redirectTo** *key* with a *value* of an empty path, so that the **home** view shows on the browser. The second key:value pair is for URL path verification. It ensures that the path is typed properly otherwise a 404 page shows or we can simply redirect all mistakes to the home page.

5. Lets add one more catch all route, this will re-route misplaced routes to /home:

```
export const routes: Routes = [
  {
    path: '',component: Home, title: 'Home - Skills20'
  },
  {
    path: 'home', redirectTo: '', pathMatch: 'full'
  },
  {
    path: '**', redirectTo: '', pathMatch: 'full'
  }
];
```

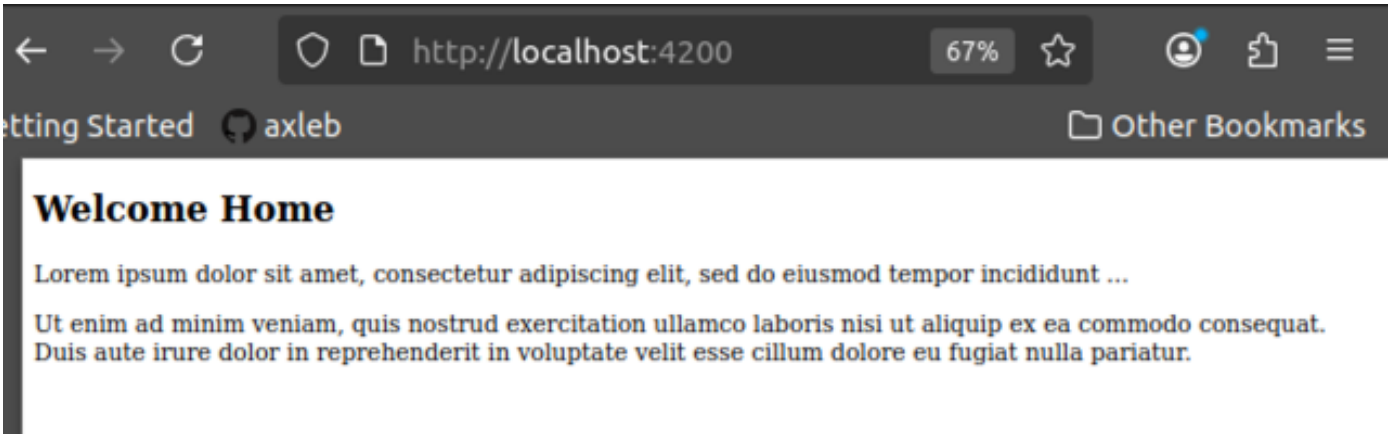
When the page displays in the browser, the title will be shown

Back in the app.html file, remove the first three lines of text from that template and just leave the **router** custom tags:

```
      <h1>Hello, {{ title() }}</h1>
      <p>Congratulations! Your app is running.</p>
      <app-home></app-home>
      <router-outlet />
```

Also remember to remove the **Home** component from the app.ts file. We imported it in #5 above. If using routing, we do not need this code.

At the end of Part 02, you should now be seeing something similar to this image:



If you got to this point, you will need to remove the Home component from the App component. So in app.ts remove the import and remove Home from the imports array.

PART 03 – INTERPOLATION

Almost all technologies that exist in this web development space has some form of interpolation. It is how the JS code computes a value that just happens to be embedded in the HTML. You have already been exposed to interpolation. The app's title was displayed in the browser in Part 01 using this feature.

1. In home.ts add a new property to the class:

```
export class Home {
  empName = "Axle";
}
```

Note, you could also use strict typing like this – `empName : string = "Axle";`

2. Now in the template, lets display this name:

```
<h2>Welcome Home {{empName}} </h2>
```

Notice that in Angular two pairs of curly braces are used for interpolation. If you spin the app now, the value in #1 will be show on the home page.

3. Like other JS frameworks, these curly braces can be used to display literal values:

```
<h2>Welcome Home {{ "Axle" }}</h2>
```

4. Even small expressions can be interpreted and displayed:

```
<h2>Welcome Home {{ 4+5 }}</h2>
```

5. Functions can be called:

```
<h2>Welcome Home {{ getName() }}</h2>
```

6. This function must exist in the TS file, so in home.ts:

```
export class Home {
  empName = "Axle";
  getName(){
    return this.empName;
  }
```

7. Of course a signal can be used also, here is the change in the TS file:

```
export class Home {
  empName = signal("Axle");
}
```

Remember to import the signal function from @angular/core

8. Then in the template:

```
<h2>Welcome Home {{ empName() }}</h2>
```

PART 04 – PROPERTY BINDING

It is possible to control an HTML element's property via TS code. Any property can be bound, and in this section, we will bind the *checked* property of a checkbox. Property binding is a one-way data flow, from the Component to the HTML Element's Property. Binding in Angular 20 is related to **Model Inputs**.

1. Add this simple CSS class in the home.css file. It is not necessary but will make the *checkbox* a little larger:

```
.checkbox-lg {
  transform: scale(3.5);
  margin: 20px;
}
```

2. Back in the HTML file of the **home** component add the following code which will add a *checkbox* on the browser window:

```
<div class="form-check">
  <input
  class="checkbox-lg"
  type="checkbox"
  id="cb1"
  checked
/>
```

I used separate lines within the ** ** to make the code easier to read. You can add this code below the last <p> tag.

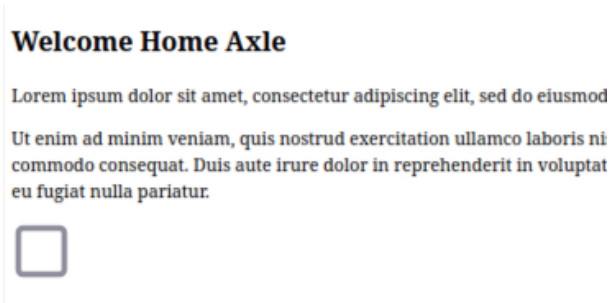
3. Binding is possible because of a special syntax in Angular. Just add square brackets around the attribute you wish to control. In our case it is the *checked* property of the checkbox:

```
  id="cb1"
  [checked] = "isCbChecked()"
/>
</div>
```

The square brackets [] signify property binding. This is a one-way data flow from the component to the checkbox. The checked property of the checkbox is now bound to a function (signal) in the component. We can now update the "tick" in the checkbox from the component.

4. Now we can complete the class part of this section. First create the signal in the component:

```
export class Home {
  empName = signal("Axle");
  isCbChecked = signal(true);
```



Just to recap. The state of the checkbox on the template is controlled by the state of a property on the class. In this case that property is **`isCbChecked`**. That that property is a signal. It starts out with a value of true, so the checkbox is initially checked when the view is loaded in the browser.

If you want to see this in action, just change the value of the signal to false in the class and the checkbox will not have the "tick" in it when the view reloads in the browser.

This, by itself is not interesting, but when paired with **event binding**, then we can have more sophisticated activity on the browser. If we can control the signal, then we can control the state of the checkbox.

PART 05 – EVENT AND TWO-WAY BINDING

Data can be generated from several sources, but typically in this boot camp, data will usually refer to something that the user wants us to know. The user will communicate with our application via the HTML document we provide.

1. Create a function in the component class to change the state of the signal:

```
name = signal("Axle");
isCbChecked = signal(true);
toggle_CB_State() {
  this.isCbChecked.update(currentValue => !currentValue);
}
```

Remember if the signal changes, then the value on the template, listening for that signal, picks up that new value. With signals, you don't directly change it's value, you *update* the old value to a new one, using the old value.

2. Add a button to the template that will be used to toggle the signal, which will then toggle the checkbox value on the actual checkbox:

```
  [checked] = "isCbChecked()"
/>
<div>
  <button (click)="toggle_CB_State()">
    Toggle Checkbox State
  </button>
</div>
```

Notice the parenthesis around the click event, this is **event binding**. By clicking on the button "Toggle Checkbox State" it then calls into the component and fires the toggle_CB_State() function, from #1.
This happens because I bounded that class function to the click event on the template button. That function then updates the signal, **`isCbChecked`**, which eventually toggles the value of the checkbox.
Note, checking the checkbox form control itself, **does not** update the signal.

3. Now that we have covered both property and event binding, can we combine the two features on one element? This is known as **two-way binding**. This feature is available via the *forms* module in Angular, so let's import that first:

```
import { Component, signal } from '@angular/core';
import { FormsModule } from '@angular/forms';
@Component({
```

Also add the **`formsModule`** to your imports array in the **`@Component()`** section.
Later you will see a non-formsmodule method to achieve two-way binding.

4. Now that we have the forms module imported, we have access to a special directive called **ngModel**. This is a class that is used to add new behavior or modify the appearance of elements in the template, so DOM based. Two way binding is typically used with the input tag, add this line in the home template:

```
    … voluptate velit esse cillum dolore eu fugiat nulla pariatur.</p>
    <input type="text" [(ngModel)] = "empName" placeholder="Enter your name for a
    special welcome message" />
<div>
```

   The directive **ngModel** is now 2-way bound to the class property empName. This means that we can update the value in input form control and if that updates, we can get the value in the component class.

5. In the class create a regular **fName** property, not a signal:

```
    export class Home {
      empName = "Axle";
      fName = "Jane";
  getName(){
```

6. In the template replace the function call with a class property in the welcome message:

```
      <h2>Welcome Home {{ fName }}</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
      tempor incididunt ...</p>
```

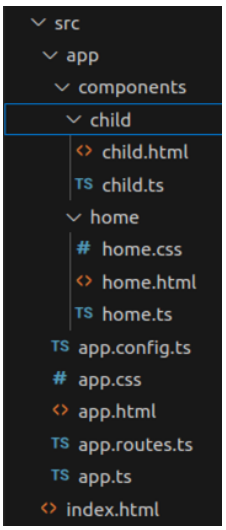
7. In the input form control, change the 2-way binding directive to point to this new fName property:

```
      <input
        type="text"
        [(ngModel)] = "fName"
        placeholder="Enter your name for a special welcome message"
      />
```

You will now see Jane as the name in the welcome message but if you change it in the input box, the welcome message name will also change. So initially, **fName** got it's value from the class but it is changeable from the template.

PART 06 –CHILD/STANDALONE COMPONENT

Since Angular has adopted **standalone** components, now a child component is any component whose selector (child) is used inside the template of another component (the parent).



Before continuing, use the CLI to create a new component:

```
ng g c components/child --standalone --skip-tests -s
```

Note: the `--standalone` flag is NOT necessary, I added it here for clarity. We will not use tests or CSS for this component. Creating a component means creating a folder that will contain up to four files, in this case it will be just two, due to the flags I added.

Once we create the **`Child`** component, we will attempt to communicate from **`Child`** to parent and back.

1. Whenever you create a new component in Angular, you always get some dummy text in the template. This is so you know you can access the component and that it is working. Now that we have a new component, we can show the contents of that component's template via any other template. Here we will use the **`Home`** component to show the **`Child`** component. So, first in home.ts, import the new component and add it to the *imports* array:

```
import { FormsModule } from '@angular/forms';
import { Child } from '../child/child';
@Component({
  selector: 'app-home',
  imports: [FormsModule, Child],
  templateUrl: './home.html',
```

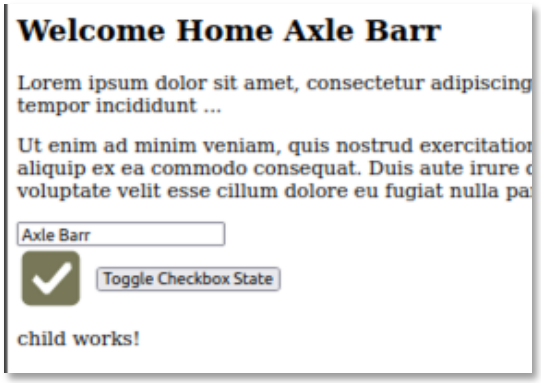
You will see a yellow warning squiggly line under the Child component in the *imports* array, this is normal, we will fix it once we actually use the imported item.

2. Now that we have imported the component, we can simply find a place in the **`Home`** component to show the Child component:

```
      Toggle Checkbox State
    </button>
  </div>
  <div>
    <app-child></app-child>
  </div>
```

In the **`Home`** component use the child **`selector`** to show the child component.

3. This is what your browser should look like after #2. The content of the **Child** component appears below the button since I placed it there in the parent's template, **Home** in this case.



4. For now, we will delete the checkbox and the button from the **Home** template and move the input text box to the child, first remove the checkbox and button:

```
  placeholder="Enter your name for a special welcome message"
/>
<div class="form-check">
  <input
  class="checkbox-lg"
  type="checkbox"
  id="cb1"
  [checked] = "isCbChecked()"
/>
<div>
  <button (click)="toggle_CB_State()">
    Toggle Checkbox State
  </button>
</div>
<div>
  <app-child></app-child>
```

This shows which lines to remove

5. This shows the home.html template file after the lines from #4 were removed:

```
pariatur.</p>
<input
  type="text"
  [(ngModel)] = "fName"
  placeholder="Enter your name for a special welcome message"
/>
<div>
  <app-child></app-child>
</div>
```

6. Move the input tag from **Home** to **Child** template. Replace the dummy line from the Child template. This is the only line that should be there now:

```
<input type="text" [(ngModel)]="empName" placeholder="Enter your name for a special welcome message" />
```

7. Previously we had to import the forms module so that we can achieve two-way binding. Well since Angular 17 we have a new mechanism to handle two-way binding, the **model** input, so import that in the Child component:

```
import { Component, model } from '@angular/core';
@Component({
```

8. Now in the **Child** class itself, use that **model input function** to create a signal that can be read from the parent, Home in this case:

```
export class Child {
  empNameC = model("");
}
```

Since I will be using the same variable name, **empName**, I add a 'C' at the end of the version on the **Child**. The parent, so **Home**, will have an 'H' at the end of it. Remember **model()** is a function, a signal function.

9. Since the **Child** component will handle its own input, we need a function that will accept the **Event** object:

```
 empNameC = model("");
  onInput(event : Event){

  }
```

If the event occurs on the Child, we need to listen to that event there.

10. First get the value from the form field, then update **empName**, which is a signal:

```
  empNameC = model("");
  onInput(event : Event){
    const empNameValue = (event.target as HTMLInputElement).value;
    this.empNameC.set(empNameValue);
  }
```

The **const** line also includes Type Assertion, see Appendix A for more information on why we do this in Angular and other Node based platforms. Remember that **empNameC** is a signal, so we have to use **set()** when changing the value.

11. On the Child template, if you have two-way binding from the previous example, remove it for now:

```
<input
  type="text"
  [(ngModel)]="empName"
  placeholder="Enter your name for a special welcome message"
/>
```

This line would have come over when we copied the input from the Home component to Child component.

12. In the HTML of the **Child** component, lets re-wire the form field to create two way binding. First just bind to the **emNameC** signal on the class of this file. Since this is a signal, we need the parenthesis. We listen for any change in the value of the field, once the user types in a new value, the (input) part:

```
<input type="text" [value]="empNameC()" (input)="onInput($event)"
placeholder="Enter your name for a special welcome message" />
```

To listen for changes on this field, we bind the HTML input event to our TS function called **onInput()** see #9. The **$event** is a special variable in Angular that stores data about some HTML element. All JS platforms have some form of this object.

13. Back on the parent, so home.ts, we need to declare a signal, we already have **empName** as a string, so use that same name as a signal, just add an "H":

```
export class Home {
  empNameH = signal("Axle");
  fName = "Jane";
  isCbChecked = signal(true);
```

Remember to import signal if it is not already there in this class. You can remove the forms module if you have it. Remove any code that worked with the checkbox we had earlier. You should have an **isCbChecked** property and a **toggle_CB_State()** function. Remove both of these.

14. Now to the template of the **Home** component, we must add the child component if we want to show it:

```
<p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur.</p>
<app-child></app-child>
```

You may have already done this part in #2 above.

15. But we are not done, we still have not established two-way binding. That is done by binding to the child's **empNameC** property value, at the point where we display the child component.

```
<h2>Welcome Home {{ empNameH() }}</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ...</p>
<p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur.</p>
<app-child [empNameC]="empNameH()"></app-child>
```

Since **empNameH** is now a signal, we add the parenthesis after the name itself in both places, but see #13.

16. The above code will create only one-way binding between parent and child. To achieve two-binding remember we must add the parenthesis around the bounded child property as well:

```
voluptate velit esse cillum dolore eu fugiat nulla pariatur.</p>
<app-child [( empNameC )]="empName"></app-child>
```

Binding is always done on the element, in this case the element was created by us. So, we are binding the **empNameC** property on the child to the **empNameH** property on the parent, **Home** in this case.

Notice that I removed the parenthesis from **empNameH**. This is because Angular does not allow function calls in two-way binding expressions. You must use a property that Angular can read/write to.

With these changes, a property on the child, empNameC is now bound to a property on the parent, empNameH. This means that when empNameH changes, it updates the value that empNameC is point to. This also happens in the reverse, hence 2-way binding.

17. Now this works normally but there is some inconsistency here. In the parent component we use the signal() type but model() type in the child. Since two-way binding now requires model(), both components should use model() rather than signal(). So, in the Home component, import model:

```
import { Component, model } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Child } from '../child/child';
```

18. Then empNameH should now point this way:

```
export class Home {
  empNameH = model("Axle");
  fName : string = "Axle Barr";
};
```

There is no need now for the isCbChecked signal or the toggle_CB_State() function.

PART 07 – TEMPLATE REFERENCE VARIABLES

Template variables can target various parts of an HTML template such as DOM elements, directives or components. It provides the .ts file with a handle on to the element it targets. This feature is mainly used to respond to user input. A template variable also gives access to one part of the template from another part! This mechanism is used mainly on forms, but it can be used anywhere on the DOM.

1. In the template of the child component, add this **** tag below the **<input>** tag, wrapped by a pair of ** ** tags:

```
<div>
  <span #spanMessage>Typical Span Message</span>
</div>
```

To make this **** tag have a reference to it, just add any name along with the hash tag. So this name #**spanMessage** is the reference variable.

2. We can now access that span-tag and all its properties and events just by using the handle, **spanMessage**. Next create a button in the same child component and we will apply event binding to call a function soon:

```
<div>
  <button>Log Message</button>
</div>
```

3. Use event binding to listen to the *click* event on this button. When this event happens, meaning the click event, we will call into a function called **logMessage()** on the class. At the same time we can pass to that class function, the textual content of the span tag on this template:

```
  <span #spanMessage>
    Typical Span Message
  </span>
</div>
<div>
  <button
    (click)="logMessage(spanMessage.textContent)"
  >
    Show Message
  </button>
</div>>
```

We can do this because we can now access the **textContent** property of the span tag. We can also get other values like innerHTML and even it's X and Y position on the browser.

4. Over in the component (class) we can code the function referred to in #3. This function can go anywhere in the class. We expect a string but we can also cater for null values also. The function will not return anything at the moment:

```
  logMessage(content: string | null): void {

  };
```

5. Lets first check for a content object and if we get one, console log out the message passed in by the button on the template:

```
  logMessage(content: string | null): void {
    if (content) {
      console.log('Content from span:', content.trim());
    }
  };
```

I just added the **`trim()`** method as a good programming practice.

If you run the app now and click the **`Log Message button`**, you will be able to see in the **console window**, the textual content between the span tags we created in #1 at the start of this part. This happens because when the button is clicked, it passes to the class function, the textual part of **`spanMessage`**. Remember that the span-tag has a template reference variable called **`spanMessage`**. This exposes all the properties and functions of that HTML element.

6. In fact you don't even need a component function to communicate with the span tag now that it has this reference:

```
<div>
  <span #spanMessage></span>
</div>
<div>
  <button (click)="spanMessage.textContent='Hello'">Show Message</button>
</div>
```

Now when the button is clicked, the span tag message "Hello" appears. In this way we can completely bypass any associated function on the TS file.

Of course you could perform two actions on a button click event. It does not make sense in this case but here is what that will look like:

```
<button(click)=
"logMessage(spanMessage.textContent);
spanMessage.textContent='hello'">
```

APPENDIX A – TYPE ASSERTION IN ANGULAR EVENT HANDLING

In Part 06 we introduced type assertion in Angular. This is the line we are looking closely at:

```
const empNameValue = (event.target as HTMLInputElement).value;
```

This code shows a type assertion sometimes referred to as type casting. This is TypeScript's way of converting the generic event.target from HTML to a more specific HTMLInputElement type for more robust code development.

The Problem with DOM events in Angular

The **event.target** property translates to a broad generic type of **EventTarget | null**. TypeScript will work better if it knows the specific type of HTML element that triggered the event. This detail will mean that input-specific properties like value, checked, or disabled can be handled by TypeScript without throwing compilation errors. If we can guarantee that a particular property is available on an HTML element, then we have better code.

The Type Assertion Solution

The **as HTMLInputElement** part is a special *type* assertion that tells TypeScript that this event target is specifically an HTML input element. This assertion gives you access to all the properties and methods that belong to input elements, such as:

value - the current input value
checked - for checkboxes and radio buttons
disabled - the disabled state
focus() - method to focus the input