



2

# Angular Components and Services

# Angular Components and Services

الكتاب الثاني من

سلسلة تعلم Angular بالعربي

تم انتاج هذا العمل في عام

٢٠٢٠

المؤلف

فيصل الفهد

وادي التقنية © النسخة الأولى 2020

هذا العمل مرخص بموجب رخصة المشاع الإبداعي: نسب المصنف –  
غير تجاري – الترخيص بالمثل 4.0 الدولي



### إهداء

إلى أظهر قلوبين في حياتي... والديّ العزيزين.  
إلى من شاركني السراء والضراء، ولم أرها عابسة يوماً..... زوجتي  
المخلصة.  
إلى من أتشوّق لأن أرى مستقبلهما المشرق بإذن الله..... ابنائي  
وبناتي.  
إلى جميع متلهف للعلم ويتوق للمعرفة  
أهديكم هذا الكتاب المتواضع، وأدعو الله أن يحوز إعجابكم.

المؤلف



## مقدمة السلسلة

اللهم علمنا ما ينفعنا وانفعنا بما علمتنا إنك انت العليم الحكيم..

أضع بين إيديكم أحبتي وأخوتي مطوري الويب وبالتحديد مطوري Front End أول سلسلة عربية تتكلم عن إطار عمل Angular، حيث تقوم فكرة هذه السلسلة على حصر جميع الميزات التي يُقدمها Angular ومن ثم في كل كتاب يتم أخذ ميزة او ميزتين والإبحار فيها بشكل مستقل محاولاً بذلك تغطية أغلب وأهم جوانبها مع الشرح المفصل والأمثلة المتعددة.

ومن هذا المنطلق يجب التنويه أن هذه السلسلة ليست للمبتدئين في البرمجة، وانما لابد ان تمتلك عزيزي المتعلم على الأقل أساسيات أركان تطوير الويب الثلاثة HTML-CSS-JavaScript ومن ثم لابد ان تمتلك اساسيات Typescript لأن إطار عمل Angular يعتمد على هذه التقنية، ولا يخفى على كل مطور واجهات أمامية أهمية Typescript والتي من وجهة نظري أرى انه يجب على كل مطور ويب تعلمها واتقانها، ونكتفي ان نقول ان Deno.js وهي اللغة الجديدة من Node.js أصبحت تدعم هذه التقنية بشكل كامل، لذلك انصح كل مطور عربي بتعلم هذه التقنية ومحاولة الإلمام بجوانبها وكيفية الاستفادة منها في إطار عمل Angular بشكل خاص ولتطوير الويب بشكل عام.

ولا يخفى عليك عزيزي المتعلم أن الكمال لله ولا يخلوا عمل من الأخطاء فالخطأ وارد والتقصير موجود، لذلك في حال وجدت أي تقصير او أي خطأ في هذه السلسلة او أردت تقديم أي اقتراح لتطوير هذه السلسلة فلا تتردد بمراسلتي على البريد الإلكتروني الذي سوف اضعه بأسفل هذه الصفحة.

وأخيراً أسأل الله العلي العظيم أن ينفع به وان يتقبله عملاً خالصاً لوجهه سبحانه،

ولا تنسوني أخوتي من صالح دعائكم.

المؤلف

فيصل الفهد

Faisal.alfahd.ksa@gmail.com

# جدول المحتويات

١	مقدمة الكتاب
٢	الفصل الأول مدخل إلى Angular Components
٣	1.1 المقدمة
٤	2.1 Root Component
٧	3.1 Selector
٩	4.1 آلية عمل Components
١٠	5.1 كيف يقوم Angular بإنشاء Components
١٢	6.1 Lifecycle hooks
١٣	1.6.1 constructor
١٦	2.6.1 ngOnChanges
١٧	3.6.1 ngOnInit
٢٢	4.6.1 ngDoCheck
٢٣	5.6.1 ngAfterContentInit
٢٣	6.6.1 ngAfterContentChecked
٢٣	7.6.1 ngAfterViewInit
٢٣	8.6.1 ngAfterViewChecked
٢٣	9.6.1 ngOnDestroy
٣٥	7.1 Template Reference
٣٧	الفصل الثاني Angular Component Communication
٣٨	1.2 المقدمة
٤٠	2.2 التواصل بين ملفات Component الواحد
٤٠	1.2.2 Interpolation Data Binding
٤٣	2.2.2 Property Data Binding
٤٥	3.2.2 Attribute Data Binding
٤٦	4.2.2 Class Data Binding
٤٩	5.2.2 Style Data Binding
٥٢	6.2.2 Event Data Binding
٥٨	7.2.2 Tow Way Data Binding
٦٦	8.2.2 ViewChild/ViewChildren
٨٨	3.2 التواصل بين Components المختلفة التي تربطهم علاقة (أب - ابن)
٩٠	1.3.2 الاتصال من component الأب إلى الابن عن طريق @Input()

١٠٢	2.3.2. الاتصال من component الأب إلى الابن عن طريق (@Output)
١١٠	3.3.2. الوصول إلى خصائص اودوال Component الأب عن طريق Component الأب
١٢٤	4.2. التواصل بين Components المختلفة التي تربطهم علاقة (أخ - أخ)
١٣٢	الفصل الثالث Services
١٣٣	1.3. مقدمة
١٣٣	2.3. الفرق بين services و components
١٣٥	3.3. تنظيم ملفات Services من خلال Core Module
١٣٧	4.3. إضافة Services وشرح أجزاء الملف
١٣٨	5.3. بناء Service تحتوي على Functionality ممكن يُعاد استخدامها في أكثر من مكان في التطبيق
١٥٦	6.3. استخدام Services لتواصل مع component ونفسه
١٧٤	7.3. استخدام Services لتواصل بين Components
١٧٤	1.7.3. الطريقة البسيطة لتواصل بين components عن طريق service
١٨٣	2.7.3. التواصل بين components عن طريق service بواسطة Subjects
١٨٨	8.3. Services Scope
١٩٢	الفصل الرابع مواضيع متقدمة في Components
١٩٣	1.4. المقدمة
١٩٣	2.4. Change Detection
١٩٧	1.2.4. Input Reference Changes
٢٠٨	2.2.4. Event Handler Is Triggered
٢٠٨	3.2.4. Async Pipe
٢١٠	4.2.4. Manual check and change detection
٢١٥	5.2.4. NgZone
٢٢٧	3.4. Smart & Dumb Components
٢٣٣	4.4. Content Projection
٢٣٥	1.4.4. Multi ng-content
٢٣٩	2.4.4. Styling Projected Content
٢٥٠	5.4. ng-container & ng-template
٢٥٤	6.4. NgTemplateOutlet
٢٦١	7.4. Component Styles
٢٦١	1.7.4. Web Components
٢٦٣	2.7.4. View Encapsulation
٢٧٥	3.7.4. طرق إضافة Styles إلى Component
٢٧٧	4.7.4. Global Styles
٢٧٨	5.7.4. Component Styles Special Selectors
٢٨٧	8.4. Renderer2 Service

٢٨٧	createElement() & createText() & appendChild() .1.8.4
٢٨٩	insertBefore() .2.8.4
٢٩٣	removeChild() .3.8.4
٣٠٠	parentNode() .4.8.4
٣٠٠	nextSibling() .5.8.4
٣٠٢	addClass() .6.8.4
٣٠٥	removeClass() .7.8.4
٣٠٨	setStyle() / removeStyle() .8.8.4
٣١١	setAttribute() / removeAttribute() .9.8.4
٣١٩	setProperty() .10.8.4
٣٢٩	selectRootElement() .11.8.4
٣٣٦	listen() .12.8.4
٣٤٢	ViewChild Token .9.4
٣٤٧	NgComponentOutlet .10.4
٣٦٨	Inheritance Components .12.4
٣٨١	Dynamic Components .13.4
٣٨١	المثال الأول .1.13.4
٣٨٧	المثال الثاني .2.13.4
٣٩٠	المثال الثالث .3.13.4
٣٩٤	المثال الرابع .4.13.4
٤٠٤	المثال الخامس .5.13.4
٤١٢	المثال السادس .6.13.4
٤٢٨	References:

## مقدمة الكتاب

في هذا الكتاب نبدأ معك عزيزي المتعلم في تعلم كيفية التعامل مع Components وإدارتها والتحكم بها وطريقة تبادل البيانات بين هذه components المختلفة، بالإضافة إلى بعض المواضيع المتقدمة في Components.

وأستطيع أنؤكد لك عزيزي المتعلم ان فهمك للComponent هو فهمك لنصف إطار عمل Angular لأن أي SPA سواء كانت Angular او غيره من مكتبات وأطر العلم الأخرى تعتمد بالأساس على مفاهيم Components وكيفية إدارتها والتعامل معها.

وهذا الكتاب قمت بتقسيمه إلى أربع فصول الفصل الأول جعلته مدخل لك لفهم Components في إطار عمل Angular اما الثاني فقامت بتخصيصه لطرق تبادل البيانات بين هذه Components والثالث هو Services وهي تقنية نستخدمها في Angular لفصل Logic معين عن Component او لمشاركة بيانات معينة بين مجموعة من Components او بين Component ونفسه، وأخيراً خصصت الفصل الأخير للكلام عن بعض المواضيع المتقدمة في Component.

المؤلف

فيصل الفهد

Faisal.alfahd.ksa@gmail.com

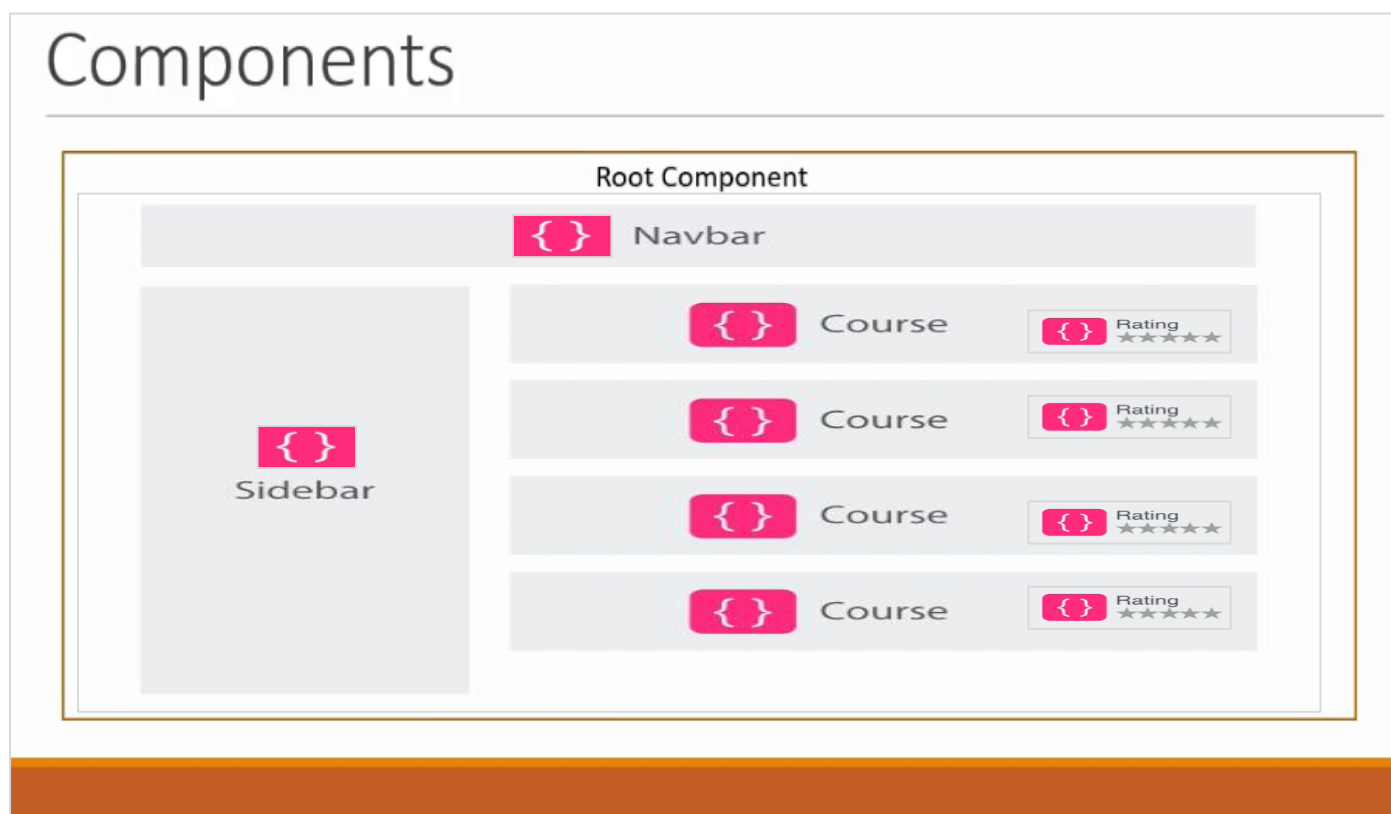
# الفصل الأول

مدخل إلى

Angular Components

## 1.1.1 المقدمة:

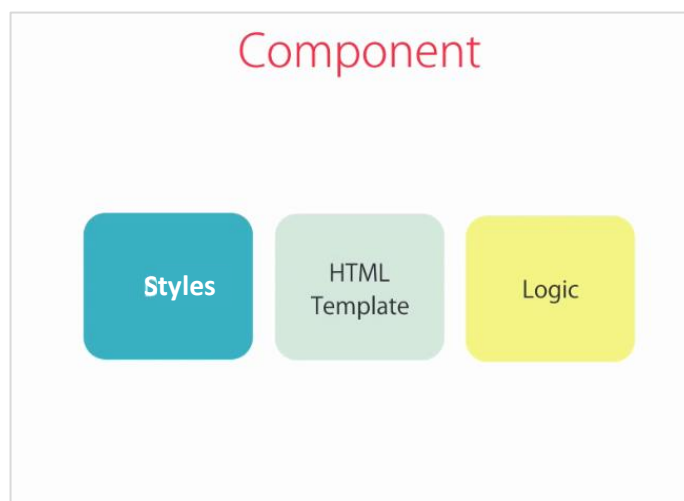
من أهم الأجزاء التي يجب فهمها ومعرفة كيفية التعامل معها هي Components في إطار عمل Angular بشكل خاص وبجميع Frameworks التي مبنية على فكرة SPA (Single page Application) بشكل عام ومعرفة كيفية تفاعل هذه Components مع بعضها البعض ومعرفة كيفية تمرير البيانات والتواصل فيما بينها، وتقوم فكرة Components على تقسيم الصفحة إلى أجزاء صغيرة بحيث كل جزء من هذه الأجزاء نسميه Component ويقوم بعرض مجموعة معينة من البيانات، وعند تحديث هذه البيانات نقوم بتحديث component الذي يحتوي على هذه البيانات فقط بدون تحديث أو إعادة تحميل كامل الصفحة كما في الطريقة التقليدية، وفي إطار عمل Angular يوجد Component رئيسي يتم انشاءه عند انشاء أي مشروع Angular جديد ويسمى هذا Component بـ Root Component او App Component، ويمكن توضيح ما قيل سابقاً في الشكل التالي:



نلاحظ في الشكل السابق انه هنالك Root Component وهو الرئيسي ويطلق عليه ايضاً الأب وتم تقسيم الصفحة إلى مجموعة من الأجزاء Components بحيث أن Navbar يعتبر Component مستقل وايضاً Sidebar وقائمة الكورسات بحيث كل كورس هو Component ولكن تم تكراره بطريقة معينة وهذا Component يحتوي بداخله ايضاً على Component فرعي آخر هو Rating بحيث يعتبر أب له، وهكذا نستطيع تقسيم الصفحة إلى مجموعة كبيرة من Components المتداخلة مع بعضها البعض والتي تنتج بالنهاية شكل تطبيق الويب.

وبنفس الوقت يجب الاشارة أن لكل Component في Angular أربعة ملفات:

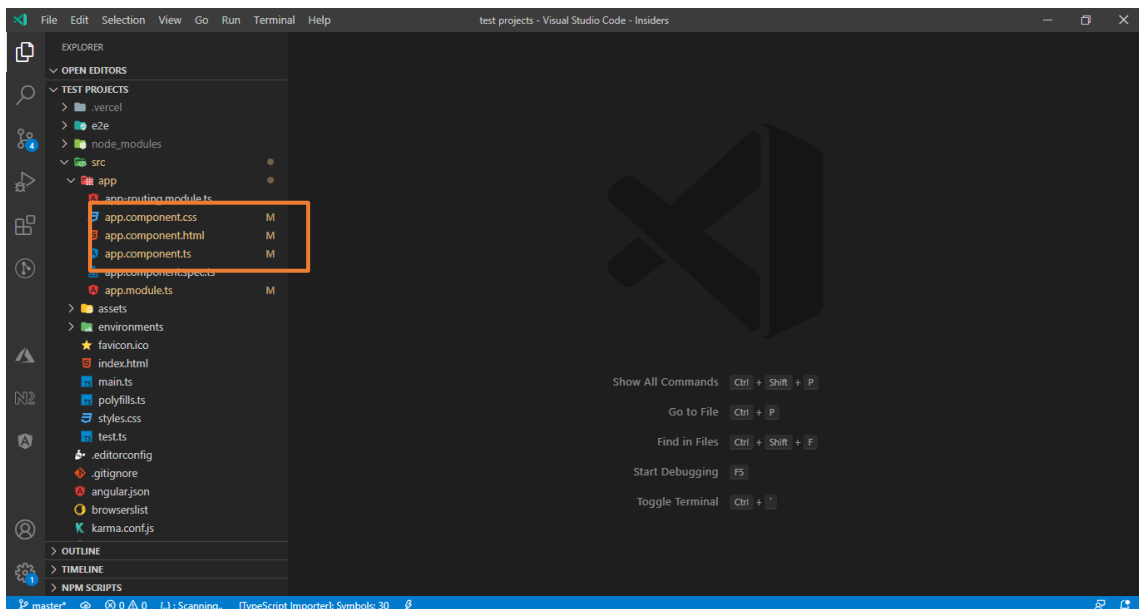
- (١) ملف ينتهي باللاحقة html ويُعرف بملف Template ويحتوي هذا الملف على Markup الخاص بالعرض view لهذا component مثل تافات HTML وما تحتيه من خصائص او التافات التي تقدمها لنا Angular او بعض المتغيرات التي نمررها (نعمل لها Binding) بين ملفات component الواحد.
- (٢) ملف ينتهي باللاحقة css او scss او sass، ويسمى ملف style ونضع فيه جميع الأكواد التي تهتم بتنسيق ملف .template.
- (٣) ملف ينتهي باللاحقة ts ويسمى ملف class وفيه يتم كتابة Logic البرمجي او أكواد Typescript، وحقيقة لو نظرنا إلى أي Component من جهة برمجية لوجدنا انه عبارة عن كلاس Class، وعندما يقوم Angular بتنفيذ أي Component فإنه يأخذ instance منه ثم يقوم بتنفيذه.
- (٤) ملف ينتهي باللاحقة spec.ts وهذا خاص بعمل test لكل component.
- ويمكن تمثيل ما قيل سابقاً بالشكل التالي:



## 2.1. Root Component:

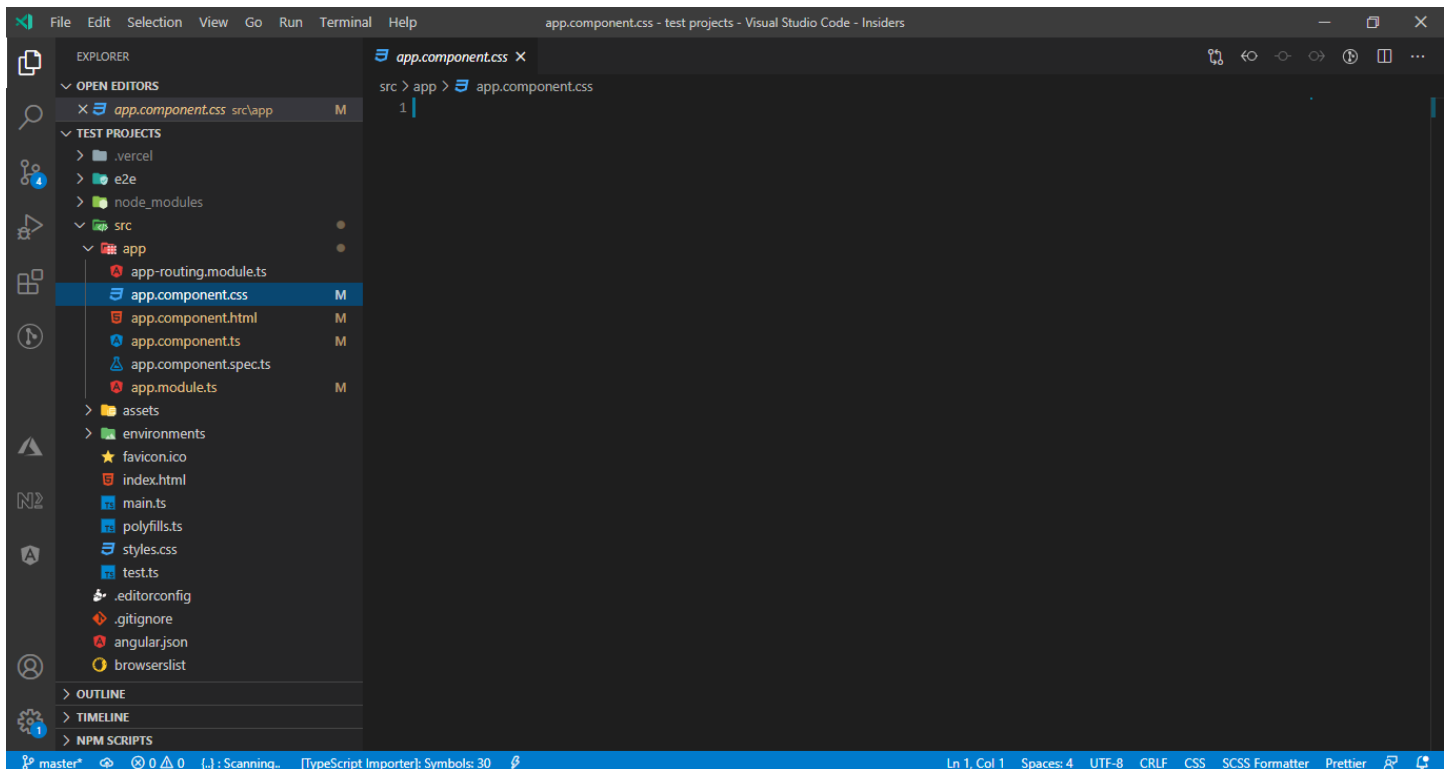
لقد أشرنا إلى Root Component في المقدمة وهنا من خلال مشروع Angular جديد سوف نتعرف عليه بشكل مفصل أكثر، لذلك لنقم بإنشاء مشروع جديد (لمعرفة طريقة إنشاء مشروع Angular جديد الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، وسوف استخدم محرر الاكواد VS Code (لمعرفة طريقة التعامل مع VS Code الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كالتالي:





كما نلاحظ يوجد في قائمة المشروع مجموعة كبيرة من الملفات التي يتم انشاءها افتراضياً (لمعرفة بنية مشروع Angular الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، وسوف نجد تحت المجلد app مجموعة من الملفات منها ثلاثة ملفات تمثل Root Component هم `app.component.css` ويمثل ملف `Style` وملف `app.component.html` ويمثل ملف `template` وملف `app.component.ts` ويمثل ملف `class`، وهذه الملفات كما اشرت قبل قليل هي عبارة عن `Root Component`، ولو استعرضنا هذه الملفات لوجدنا التالي:

ملف `app.component.css`: ملف فارغ لا يوجد به أي اكواد افتراضية.



ملف `app.component.html`: وبشكل افتراضي يوجد به بعض Markup وسوف نقوم بحذفها لكي نقوم بكتابة اكواد HTML التي تناسب احتياجنا، مع ملاحظة انه في حال انشاء مشروع Angular جديد وتضمنين Routing معه فإنك سوف تشاهد تاغ مع هذ Markup اسمه `<router-outlet></router-outlet>` لذلك يجب الانتباه عند حذف أي اكواد افتراضية الإبقاء على هذا



```

You, 3 minutes ago | 1 author (You)
1  import { Component } from '@angular/core'; 1
2
You, 3 minutes ago | 1 author (You)
3  @Component({
4      selector: 'app-root',
5      templateUrl: './app.component.html', 2
6      styleUrls: ['./app.component.css']
7  })
8
9  export class AppComponent {
10     title = 'app'; 3
11 }
12

```

الجزء الأول: خاص بالاستدعاءات بحيث أي Module نريد استخدامه في هذا component سواء كان مبني ضمناً مع Angular او حتى من خلال مكتبة خارجية يتم استدعاءه في هذا الجزء عن طريق الامر import.

الجزء الثاني: ويحتوي على توصيف لهذا component وهو على شكل كائن Object يحتوي على ثلاث مفاتيح Keys وهي كالتالي:

- selector: وهو مهم جداً حيث يُعبر كأنه العنوان الذي يشير إلى هذا component عند استخدامه مع components الأخرى، مع العلم انه يتم استخدامه على شكل تاغ <app-root>.
- templateUrl: وهو عبارة عن مسار ملف template لهذا component.
- styleUrls: وهو عبارة عن مسار ملف styles لهذا component ونلاحظ انه موجود بين اقواس المصفوفة والتي تدل على انه يقبل اكثر من ملف Styles وليس ملف واحد فقط ولكن حقيقة جرت العادة ان يُكتفى بملف واحد فقط.

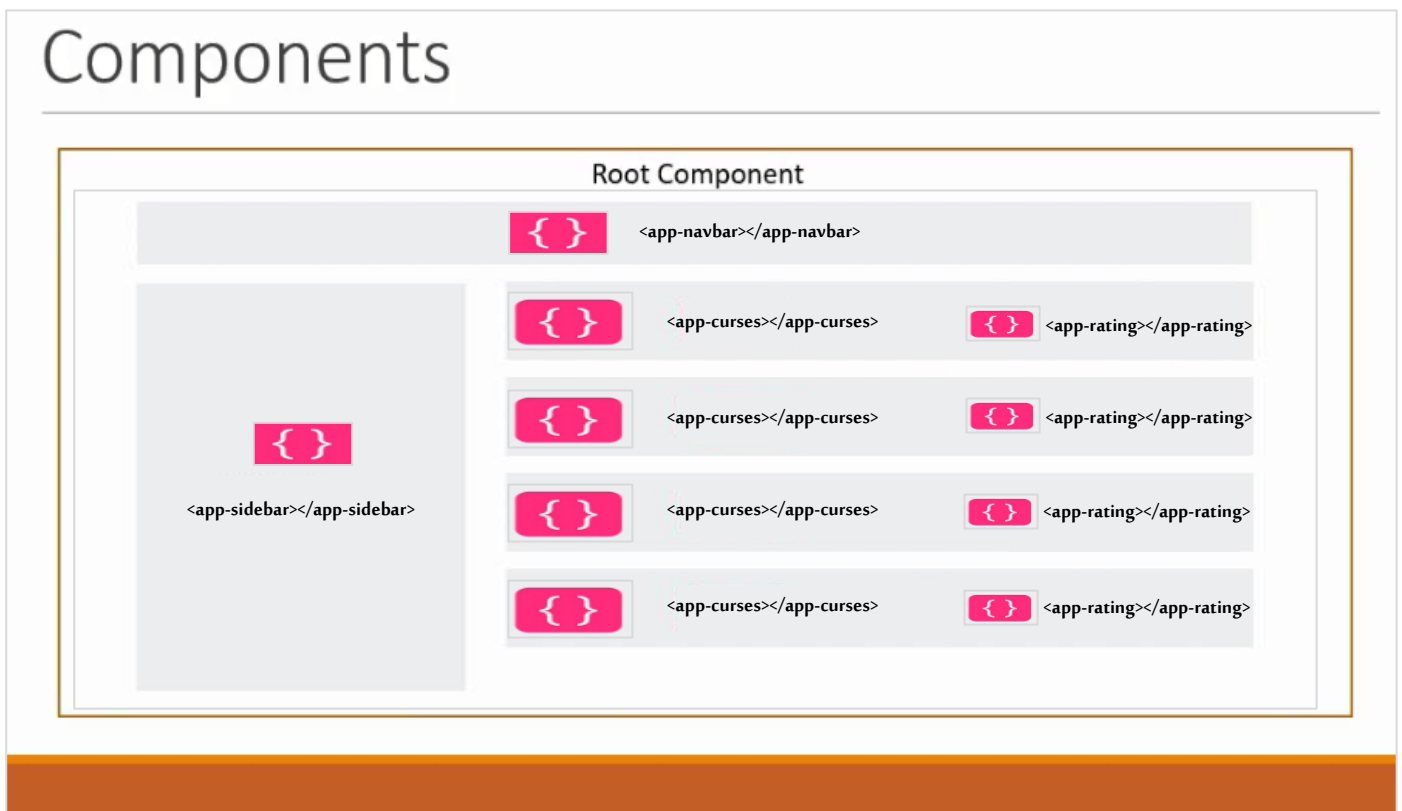
الجزء الثالث: وهو عبارة عن class الذي يمثل هذا component ويتم داخله كتابة logic البرمجي من متغيرات ودوال وغيره، ونلاحظ ان هذا class يحتوي على اسم وهذا الاسم مهم في تعريف هذا class في ملف app.module.ts، وفي استخدامه برمجياً في أجزاء أخرى في مشروعك.

ومما ذكر سابقاً نلاحظ أهمية هذا الملف وانه حلقة الوصل بين ملفات أي component، وهذا المقصد الذي ذكرناه سابقاً عندما قلنا ان Angular يأخذ instance من هذا الكلاس عندما يقوم ببناء أي component لأن عن طريقه يعرف مسار الملفات الأخرى الخاصة بهذا component وجميع الاستدعاءات التي تمت على هذا component لكي يقوم بقراءتها جميعاً وبالأخير ينتج لنا component بشكله النهائي للمستخدم جنباً إلى جنب مع components الأخرى.

### 3.1 Selector:

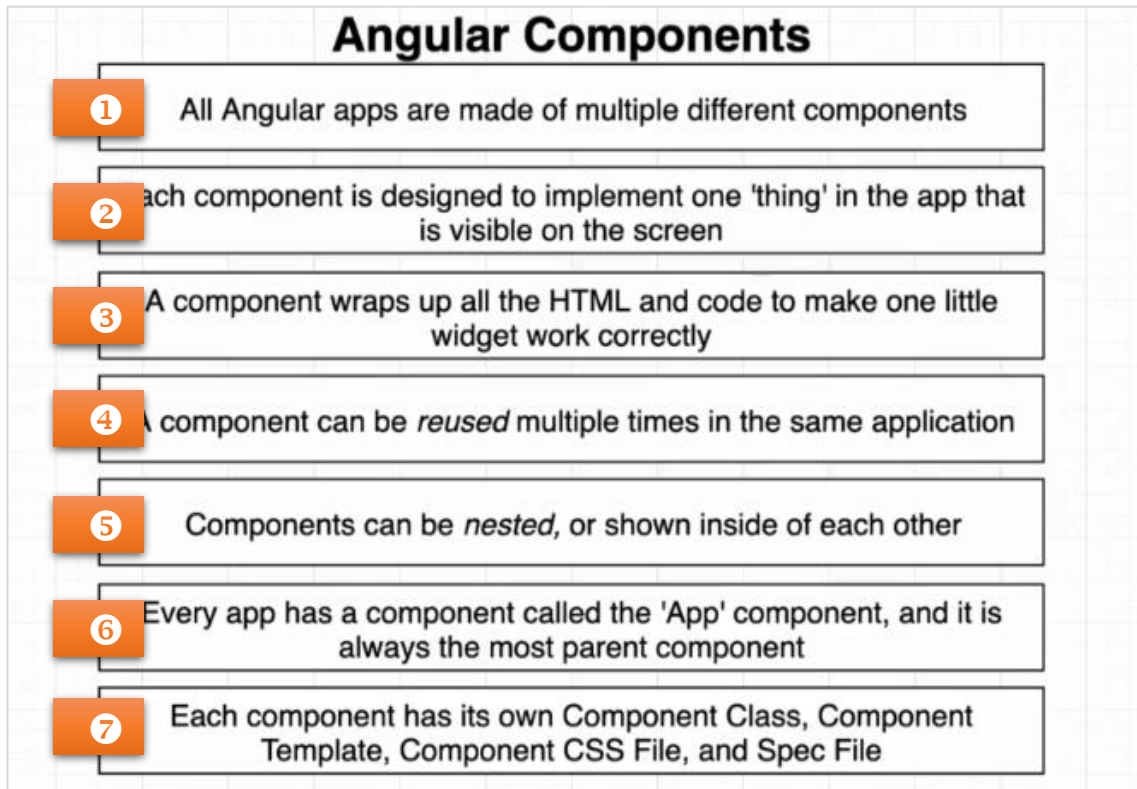
لأهمية selector لا بد من شرحها بشكل عملي أكثر، يوجد في ملفات المشروع ملف مهم اسمه index.html وهو الملف الذي يقرأه server في العادة لأي تطبيق او موقع web، ويوجد في هذا الملف selector الخاص بي Root Component، على شكل تاغ html كالتالي:

وعندما يقوم Angular بقراءة ملف index.html وهو الانطلاقة التي يبدأ بها Angular سوف يجد تاغ <app-root> وسوف يعرف أن هذا هو selector الخاص بي Root Component وبشكل تلقائي سوف يأخذ instance من الكلاس لهذا component ومعرفة أي ملفات أخرى متعلقة به او استدعاءات ومن ثم يقوم بقراءتها جميعاً، وكما قلنا سابقاً أن هذا Root Component يعتبر بمثابة الاب لذلك هو الآخر قد يحتوي على selector لي component آخر، ويمكن إعادة تمثيل أول شكل استعرضناه بفرض ان أسماء selector مشابهه للأسماء الافتراضية التي قمنا بتضمينها في هذا الشكل (navbar-sidebar-..etc.)، كالتالي:



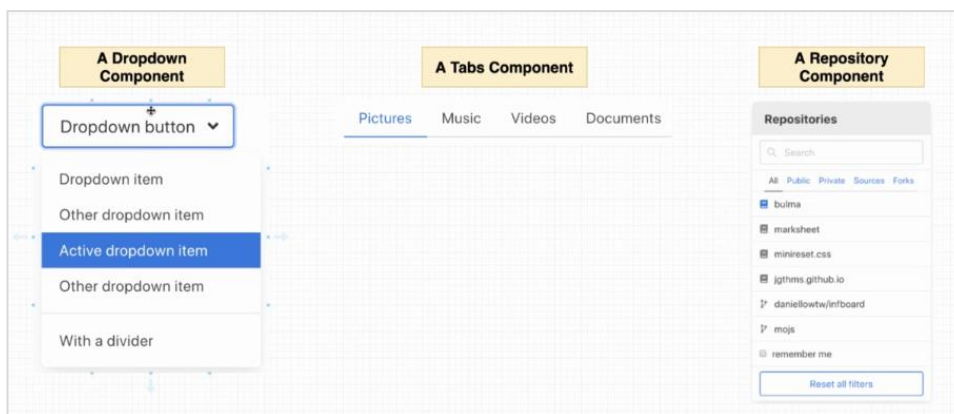
## 4.1. آلية عمل Components:

نستطيع فهم آلية عمل أي component من خلال الخطوات التالية، كالتالي:



(١) النقطة الأولى عبارة عن مفهوم عام تم فيه ذكر ان جميع تطبيقات angular هي في الحقيقة مكونة من مجموعة مختلفة من components، او بصيغة أخرى تم بناء هذه التطبيقات بناءً على مجموعة متعددة من components المختلفة.

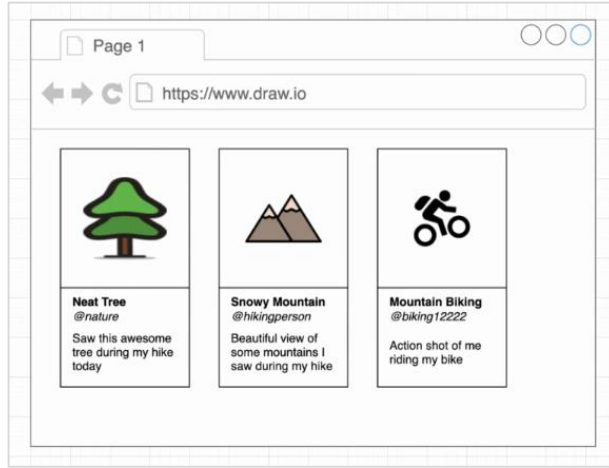
(٢) اما هذه النقطة فتبين أن لكل component مهمة معينة يقوم فيها بعرض جزء ما على شاشة المستخدم، ويمكن توضيحها بالشكل التالي:



نلاحظ في الشكل السابق لدينا ثلاث أجزاء جزء عبارة عن Dropdown Button لعرض مجموعة من البيانات وجزء آخر عبارة عن مجموعة من tabs لعرض مجموعة من الأقسام وجزء آخر عبارة عن List، لذلك نستطيع وضع كل جزء من هذه الأجزاء في component خاص فيه. بحيث كل component يقوم بعرض جزء معين او شيء ما، وهذا يوضح ايضاً ما تم ذكره في النقطة الأولى.

(٣) اما هذه النقطة تشير إلى انه يتم كتابة جميع تاغات HTML في ملف Template والكود البرمجي Logic في ملف class واخيراً ملف css لتنسيق المظهر، وجميع هذه الأمور يتم كتابتها والتفاعل بينها لكي يعمل هذا الجزء (الشيء ما) بشكل صحيح.

(٤) وهذه النقطة تُشير إلى انه من فائدة تقسيم التطبيق إلى مجموعة من components يمكن إعادة استخدام هذه components في مواضع مختلفة في نفس التطبيق، فمثلاً لو كان لدينا مجموعة من cards، كما في الشكل التالي:



وممكن نحتاج أن نستخدم هذه cards بمواقع أخرى في التطبيق ولكن ببيانات مختلفة، فنستطيع انشاء component واحد يحتوي على card واحدة ونجعله يستقبل البيانات بشكل ديناميكي ويقوم بعرضها ولكن ببيانات مختلفة.

(٥) في هذه النقطة يتم توضيح كيف يمكن استخدام component معين داخل component آخر بحيث يكون هذا الأخير بمثابة الحاوية أو الأب للـ component الأول. وبنفس الوقت هذا component الأب هو الآخر ممكن ان يصبح حاوية أو أب لـ component آخر، وهكذا نستطيع انشاء شجرة متداخلة من components بحسب الاحتياج.

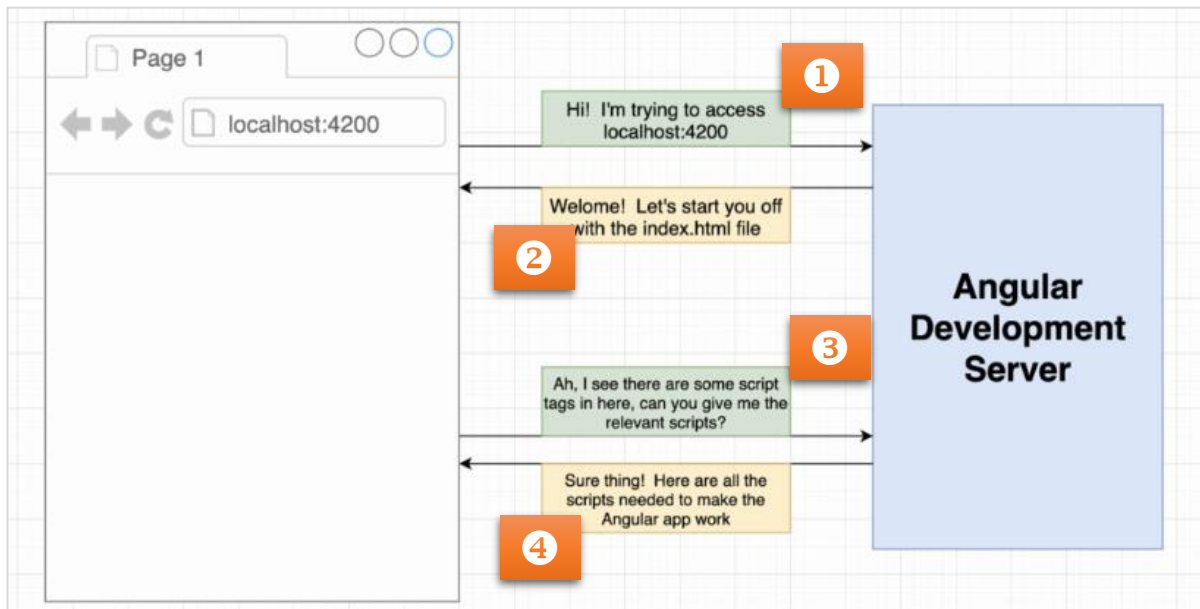
(٦) وهذه النقطة أشرنا لها سابقاً وتعني ان لكل تطبيق Angular هنالك component يعتبر بمثابة الأب لجميع الـ components الأخرى.

(٧) وهذه النقطة ايضاً أشرنا لها سابقاً وهو ان لكل component أربعة ملفات مختلفة يقوم كل ملف منها بالقيام بمهمة محددة سواء كانت الاهتمام بعرض أكواد HTML ويسمى هذه الملف Template او التنسيق الخاصة بهذا component وفي هذه الحالة يسمى ملف css او كتابة Logic البرمجي وفي هذه الحالة يسمى ملف class وأخيراً اجراء الاختبارات على هذا component وفي هذه الحالة يسمى spec.

## 5.1. كيف يقوم Angular بإنشاء الـ Components :

نستطيع توضيح هذه الجزئية بشكل مجمل من خلال هذا الشكل:





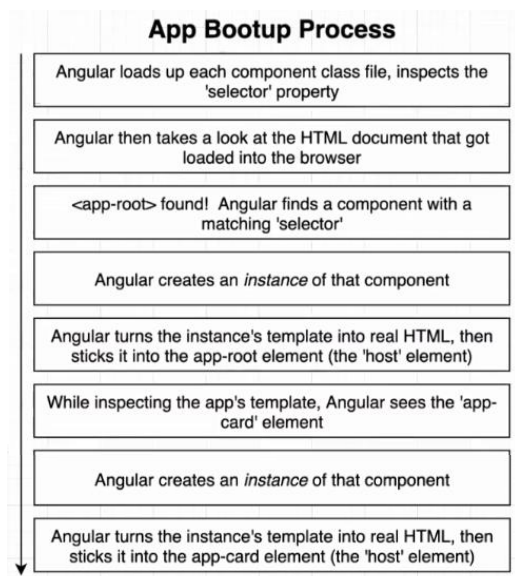
(١) حيث يقوم المتصفح في البداية بمحاولة الاتصال بالـ Angular Server والذي هو في حالتنا هذه localhost،  
 (٢) ومن ثم يبدأ يقرأ ملف index.html. وسوف يجد مجموعة من ملفات scripts والتي يقوم Angular بإنشائها ديناميكياً أثناء التشغيل في DOM وليست موجودة بشكل static كما هي العادة عند إضافة هذه الملفات إلى ملف index.html في pure JavaScript، ولو استعرضنا DOM عن طريق المتصفح لوجدنا هذه الملفات (لمعرفة كيفية التعامل مع أدوات المطور في المتصفح الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كما في الشكل التالي:

```

<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <script src="runtime.js" type="module"></script>
    <script src="polyfills.js" type="module"></script>
    <script src="styles.js" type="module"></script>
    <script src="vendor.js" type="module"></script>
    <script src="main.js" type="module"></script>
  </body>
</html>
  
```

(٣) لذلك سوف يطلب المتصفح من Angular هذه الملفات.  
 (٤) وأخيراً يستقبل المتصفح هذه الملفات ويقوم بقراءتها، لكي يعمل Angular بشكل صحيح.  
 وبنفس الوقت لو رجعنا إلى ملف index.html لوجدنا التاغ (selector) ذو الاسم app-root، الذي يشير إلى component الاب (كما ذكرنا سابقاً) وسوف يقوم بعمل instance من الكلاس الخاص بهذا component ويقوم بتحويل جميع هذه الملفات

إلى اكواد HTML وJavaScript ويعرضها في المتصفح، وبنفس الوقت إذا وجد selector آخر موجود في component الال يقوم بعمل بنفس الامر معه، وهكذا، كما في الشكل التالي:



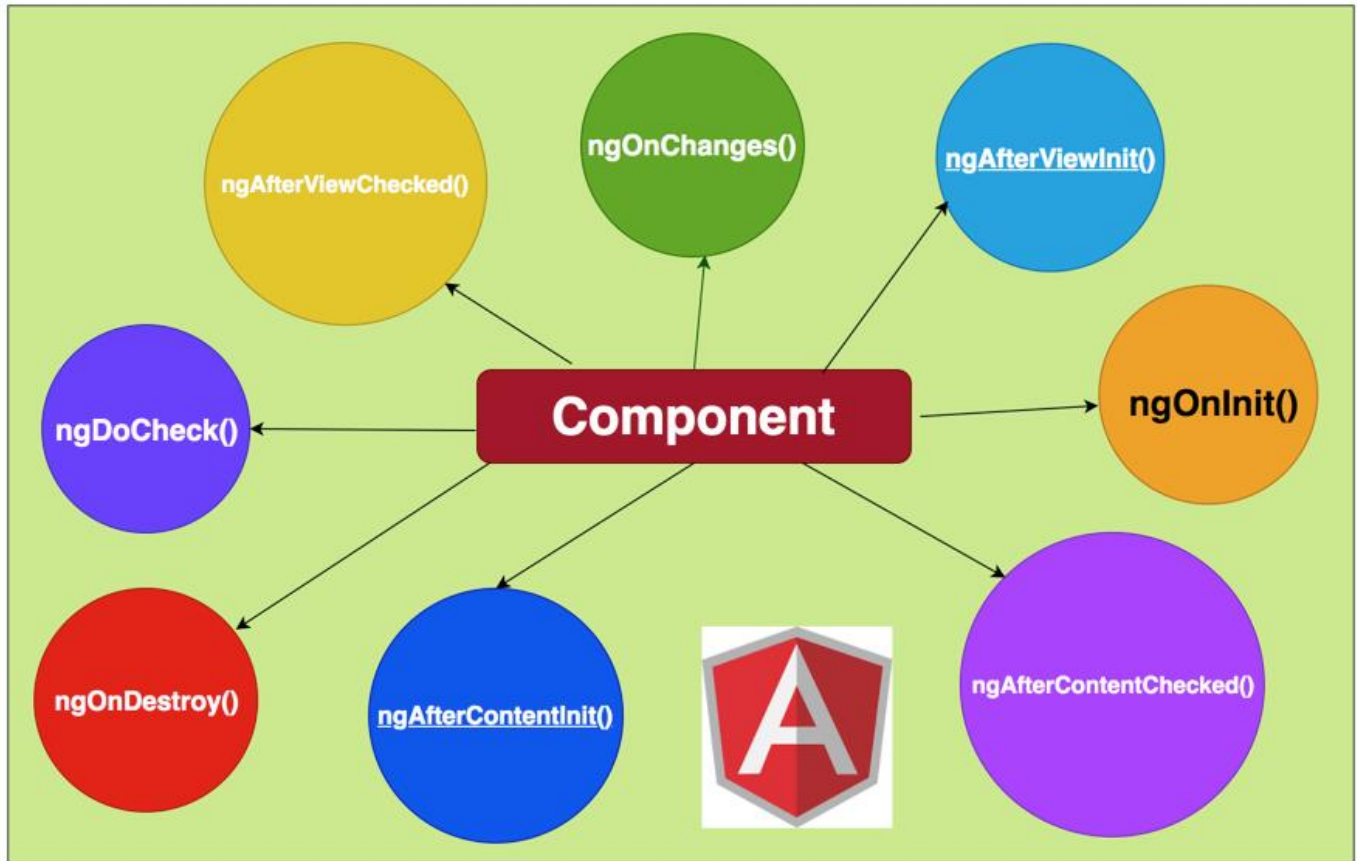
## 6.1. Lifecycle hooks

يمر أي Component في مجموعة من المراحل وتسمى هذه المراحل دورة حياة Component سواء من قبل إنشاء هذا الComponent إلى تدميره، وقدمت لنا Angular قدرة على التحكم بجميع مراحل حياة أي Component على شكل دوال، وعلى سبيل المثال لو أردنا ان ننفذ كود معين قبل لا يتم بناء template الخاص بهذا component فنستخدم في هذه الحالة الدالة constructor ونكتب الكود الذي نريده بداخلها، أما في حال أردنا ان ننفذ كود معين في بداية إنشاء هذا Component فنستخدم ngOnInit، أما في حال أردنا تنفيذ كود ما في حال تدمير هذا Component فنستخدم ngOnDestroy، ونستطيع أن نقول بعبارة أخرى ان هذه الدوال بمثابة أحداث Event، نستطيع الاستفادة منها لتنفيذ عمل معين.

وفيما يلي قائمة بجميع هذه الأحداث، ومرتبة من بداية دورة حياة الComponent إلى تدمير هذا Component:

- 1 constructor
- 2 ngOnChanges
- 3 ngOnInit
- 4 ngDoCheck
- 5 ngAfterContentInit
- 6 ngAfterContentChecked
- 7 ngAfterViewInit
- 8 ngAfterViewChecked





وجميع هذه الدوال موجودة في interface تحمل نفس اسم الدالة بدون ng ما عدا constructor يتم استدعائها بشكل مباشر، بمعنى لو أردنا استدعاء الدالة ngOnInit فيفضل في البداية ان نعمل implements لي interface الذي يحمل الاسم OnInit ومن ثم نستدعي الدالة ngOnInit، وهكذا بقية الدوال ما عدا constructor.

والآن بعد السرد المجلل لنأتي إلى التفصيل بشرح كل دالة على حدا.

### 1.6.1 :constructor

حقيقة ميزة constructor ليست خاصة بي angular وانما هي ميزة من ميزات الـ Classes في JavaScript بشكل عام وفي Typescript بوجه خاص، حيث من أسس أي class ان يكون له دالة constructor ويتم تنفيذ هذه الدالة عند عمل instance من هذا class، فلو افترضنا انه لدينا class يحمل الاسم car ويحمل بعض Property بداخله ودالة constructor بطبيعة الحال، كالتالي:

```

class Car {
  constructor () {
    console.log('Hello world from Constructor!');
  }
}

```

فإنه عندما نعمل instance من هذا الكلاس في كائن object فإنه أول ما يقوم به هو تنفيذ محتويات الدالة constructor، كالتالي:

```
Const car = new Car()
```

وسوف تكون النتيجة طباعة الرسالة ('Hello world from Constructor!')

ومن الفوائد المهمة لهذه الدالة هي تمرير مدخلات لهذا class عند عمل instance منه، كالتالي:

```
class Car {
  private color: string;
  private name: string;
  private model: number;
  constructor (color: string, name: string, model: number) {
    this.color = color;
    this.name = name;
    this.model = model
  }
  printCarInfo {
    console.log('Car Name: ' + this.name);
    console.log('Car Color: ' + this.color);
    console.log('Car Model: ' + this.Model);
  }
}
```

واختصارا نستطيع كتابة class السابق بالطريقة التالية:

```
class Car {

  constructor (color: string, name: string, model: number) { }

  printCarInfo {
    console.log('Car Name: ' + this.name);
    console.log('Car Color: ' + this.color);
    console.log('Car Model: ' + this.Model);
  }
}
```

نلاحظ أننا اكتفينا بتمرير المتغيرات عن طريق دالة constructor بحيث أصبحت متاحة لجميع الدوال والخصائص داخل هذا class، ولذلك استخدمناها بشكل مباشر في الدالة printCarInfo،

وعندما نعمل instance من هذا class فإننا نمرر القيم بشكل مباشر له عن طريق دالة constructor الخاصة به، كالتالي:

```
const car = Car('avalon', 'white', 2019);
car.printCarInfo();
```

وسوف تكون النتيجة:

```
Car Name: Avalon
Car Color: White
Car Model: 2019
```

وبعدما استعرضنا بشكل سريع ماهية هذه الدالة في Typescript، نريد ان نسقط هذه المعلومات على عمل constructor في Angular.

ولو رجعنا إلى بداية استعراضنا لمقدمة كلامنا عن components حيث قلنا ان أي component هو في حقيقة الأمر برمجياً عبارة عن كلاس class، وعند بداية تشغيل أي component فإن Angular سوف يقوم بعمل instance من هذا class وهنا يبدأ عمل دالة constructor حيث يتم تنفيذها عند عمل أي instance من أي component.

وفي الغالب يُستفاد منها بعمل inject لأي service كما فعلنا في المثال السابق عندما عرفنا المدخلات (المتغيرات) color و name و model في دالة constructor، وهو ما يسمى بمفهوم Dependency Injection، وسوف نتكلم عن هذا الامر في فصل Services.

فعلى سبيل المثال لو رجعنا واستعرضنا محتويات app.component.ts، التي قمنا باستعراضها سابقاً في المقدمة، واضفنا دالة constructor، كالتالي:

```
app.component.ts ملف
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {

  constructor() {

  }

  title = 'app';
}
```

واردنا ان نستفيد من service ذات الاسم FormBuilder التي تقدمها لنا angular لتعامل مع النماذج، فنستطيع ان نعمل لهذه service مايسمى inject في متغير عن طريق دالة constructor، وبذلك يصبح هذا المتغير يشير إلى هذه service ويحمل جميع الخصائص والدوال الموجودة فيها، وبنفس الوقت نستطيع الاستفادة منه داخل هذا class كما فعلنا سابقاً في الكلاس Car، كالتالي:

```
app.component.ts ملف
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```

}))

export class AppComponent {

  constructor(private fb: FormBuilder) {

  }

  title = 'app';
}

```

وبذلك أصبح المتغير fb متاح داخل الكلاس AppComponent فقط، ولكن ماذا لو اردنا الاستفادة من هذا المتغير خارج هذا class، في ملف tamplate الخاص بهذا الملف في هذه الحال نستطيع بكل بساطة تغيير التعريف من private إلى public، او نعرف متغير آخر من النوع public ونجعله يأخذ القيمة في دالة constructor، كالتالي:

ملف app.component.ts

```

import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {

  public fb2: FormBuilder;

  constructor(private fb: FormBuilder) {
    this.fb2 = fb;
  }

  title = 'app';
}

```

وبالتالي أصبح fb2 نستطيع استخدامه خارج هذا class.

## 2.6.1. ngOnChanges:

هذه اول دوال lifecycle hooks، وبما انها من دوال lifecycle hooks لذلك لابد من استخدامها ان نعمل implements ل interface الخاص بكل دالة من هذه الدوال، اما ترتيب تنفيذها فتأتي الثانية بعد constructor، وبنفس الوقت يتم استدعاءها وتنفيذ محتواها في حال تغير القيمة التي تم تمريرها إلى Component الأب من Component الأب عن طريق Decorator ذو الاسم (@Input)، وبما اننا لم نتطرق إلى طرق تبادل البيانات بين أنواع Components المختلفة لذلك سوف نؤجل الكلام فيها لاحقاً في فصل Angular Component Interaction.

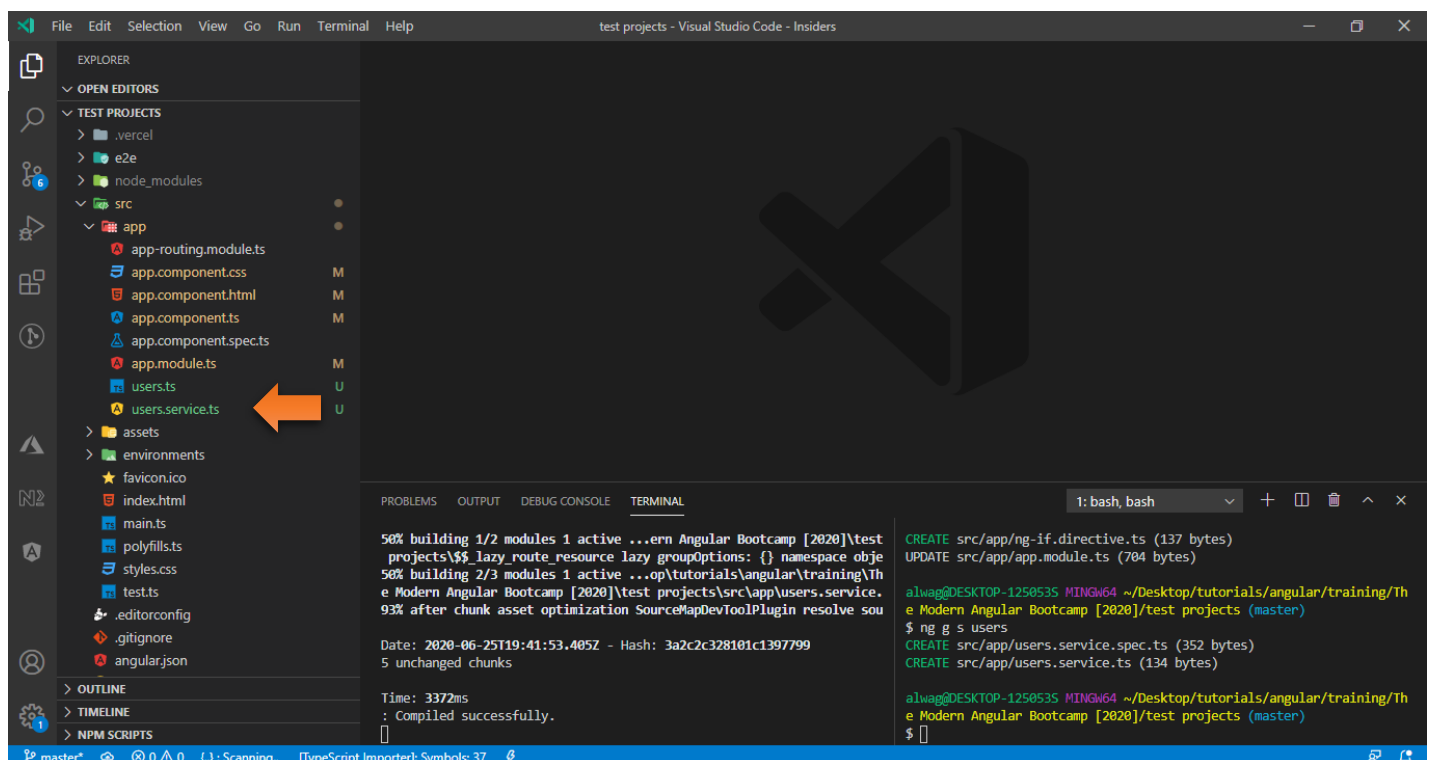
### :ngOnInit .3.6.1

وهذه الدالة من دوال lifecycle hooks ويأتي ترتيب تنفيذها الثالث بعد ngOnChanges وتنفذ عند بداية عمل Initialize لي component فقط ولا تُستدعى مرة أخرى، وهي مشابهة لعمل constructor، ولعل من الفروق بينها وبين constructor انها خاصة بي angular ويتم تنفيذها قبل انشاء ملف template، وغالباً نضع فيها الأكواد اللازمة لقراءة البيانات وتخزينها في متغير قبل تمرير هذا المتغير إلى ملف template لكي يتم بناء هذا الملف بناءً على هذه البيانات.

ولتوضيح الصورة لنفرض أنه لدينا service معينة تتصل بقاعدة البيانات عن طريق api معينة لجلب بيانات عدد من المستخدمين وعرضها في ملف template على شكل list، بمعنى أن هذا template يعتمد ببناؤه على هذه البيانات، في هذه الحالة نستطيع ان نقوم بحقن inject هذه service في constructor كما أشرنا سابقاً ومن ثم نستدعي الدالة الخاصة بجلب هذه البيانات ونخزنها في متغير ونمرر هذا المتغير إلى ملف template لهذا component لكي يقوم ببناء هذا الملف بناءً على هذه البيانات.

ولنقوم بتطبيق ما قيل سابقاً من خلال مثال عملي، ويجب الانتباه أننا إلى الآن لم نشرح مفهوم service لذلك لا يتشتت انتباهك عزيزي المتعلم بماهية service واجعل تركيزك فقط على دالة ngOnInit ومعرفة عملها فقط.

الآن لنفرض انه لدينا هذه service وليكن اسمها users.service.ts، كالتالي:



حيث تقوم بالاتصال بقاعدة البيانات عن طريق رابط api معين لجلب بيانات المستخدمين وقد أنشأت دالة اسميتها getAllUsers لتعامل مع هذه الجزئية، كالتالي:

ملف users.service.ts

```
import { Injectable } from '@angular/core';
import { Users } from './users';
import { Observable } from 'rxjs';
```

```
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})

export class UsersService {
  constructor(private http: HttpClient) { }

  getAllUsers(): Observable<Users[]> {
    return this.http.get<Users[]>('https://jsonplaceholder.typicode.com/users');
  }
}
```

وفي ملف app.component.ts نقوم بعمل inject لهذه service في متغير وليكن اسمه userService، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {

  constructor(private userService: UsersService) {}

}
```

الآن نأتي إلى ما يهمنا هنا وهو الاستفادة من الدالة ngOnInit فكما هو واضح نريد عرض بيانات المستخدمين في ملف template لهذا component، ولكن نحتاج إلى هذه البيانات في البداية لكي نستطيع بناء هذا template بناءً على هذه البيانات، وهنا يأتي دور ومهمة هذه الدالة، حيث يتم تنفيذها قبل بناء ملف template وهذا الذي نريده بالتحديد لذلك لنقم بعمل implements لي interface الخاص بهذا الدالة، كالتالي:

ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  constructor(private users: UsersService) {}

}
```

```
}
```

بعدها نقوم بكتابة الدالة `ngOnInit`، ونلاحظ أن `interface` لهذه الدالة هو نفس الاسم ولكن بدون `ng` فقط `OnInit`.

ملف `app.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  constructor(private users: UsersService) {}

  ngOnInit(): void {
  }
}
```



وأخر خطوة هي كتابة `logic` البرمجي الخاص باستقبال البيانات في هذه الدالة، لأن هذا هو المكان الذي يتم فيه قراءة أي `logic` برمجي قبل بناء ملف `template`، وايضاً لا بد أن اشير مرة أخرى أن لا تنشئت عزيزي المتعلم فليس الهدف هنا هو شرح مفاهيم `rxjs`، وانما الهدف الأساسي هو دالة `ngOnInit` وكيف استفيد منها، كالتالي:

ملف `app.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Users } from './users';
import { UsersService } from './users.service';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  users$: Observable<Users[]>;

  constructor(private users: UsersService) {}

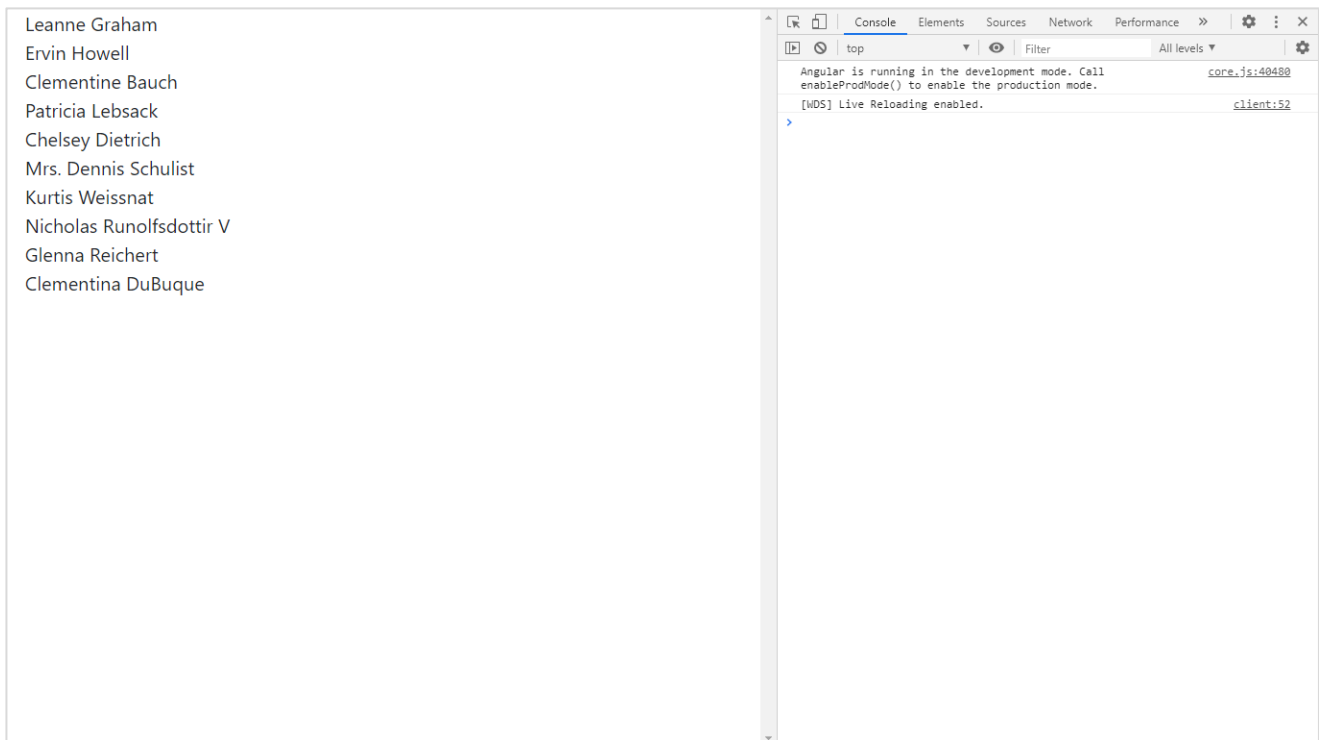
  ngOnInit(): void {
    this.users$ = this.users.getAllUsers();
  }
}
```

استقبلنا بيانات المستخدمين في متغير اسميناها `this.users$`، والآن لنمرر هذا المتغير إلى ملف `template` لعرض هذه البيانات، كالتالي:

ملف `app.component.html`

```
<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }}
  </li>
</ul>
```

أما النتيجة (لأبداً ان تعمل للمشروع `serve` عن طريق الأمر `ng serve --open`، ولمعرفة كيفية التعامل مع أوامر Angular CLI الرجاء مراجعة الكتاب الأول من هذه السلسلة (Angular Environment Setup)، كالتالي:



ولمزيد من التمرين لنعمل styling عن طريق ملف `style` الخاص بهذا `component`، كالتالي:

ملف `app.component.css`

```
* {
  margin: 0;
  padding: 0;
}

div {
  margin: 20px;
}

ul {
  list-style-type: none;
  width: 500px;
}
```



```

h3 {
  font: bold 20px/1.5 Helvetica, Verdana, sans-serif;
}

li img {
  float: left;
  margin: 0 15px 0 0;
}

li p {
  font: 200 12px/1.5 Georgia, Times New Roman, serif;
}

li {
  padding: 10px;
  overflow: auto;
  border: .1px solid #e6e6e6;
}

li:hover {
  background: #eee;
  cursor: pointer;
}

```

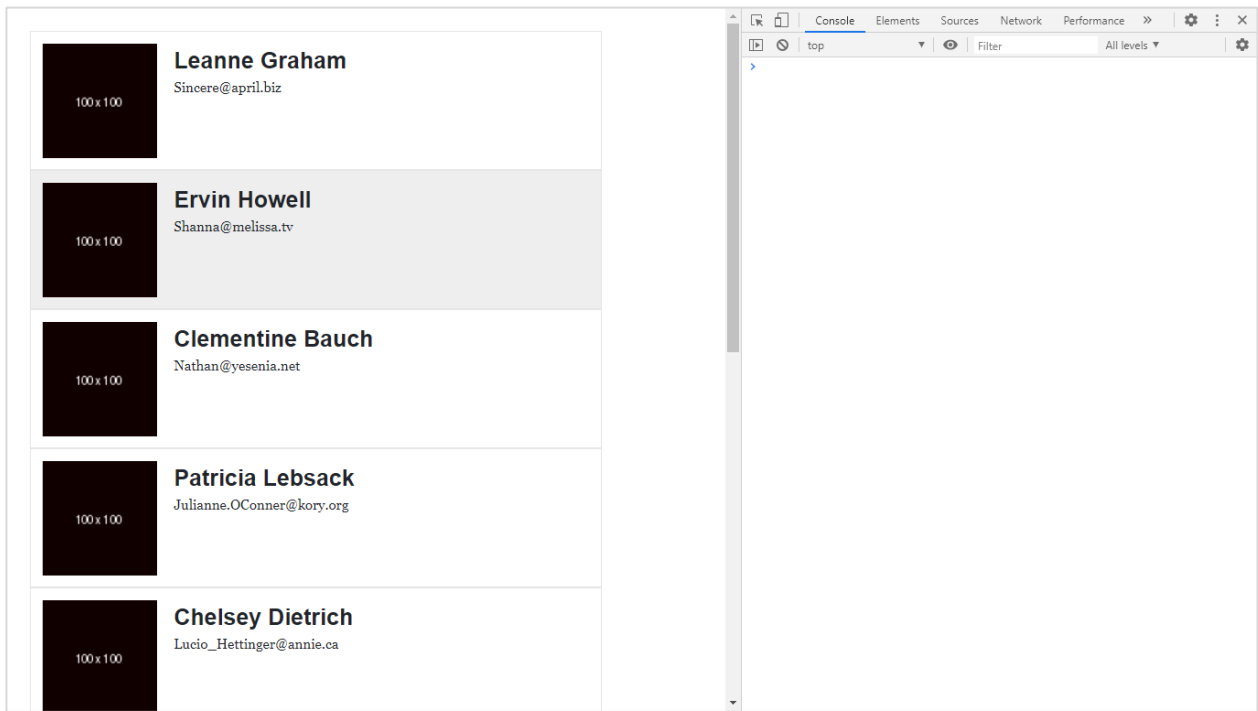
ملف app.component.html

```

<div>
  <ul>
    <li *ngFor="let user of users$ | async">
      
      <h3>{{ user.name }}</h3>
      <p>{{ user.email }}</p>
    </li>
  </ul>
</div>

```

والنتيجة في المتصفح (لا بد ان تعمل للمشروع serve عن طريق الأمر ng serve –open، ولمعرفة كيفية التعامل مع أوامر Angular CLI الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كالتالي:



ومراجعة سريعة على هذه الدالة نستطيع ان نقول ان هذه الدالة من دوال lifecycle hooks ويفضل ان نعمل impalements لي interface الخاص بها، ونكتب فيها Logic البرمجي الذي نريده ان يعمل قبل بناء template للComponent كأن يكون لدينا مجموعة بيانات ونريد استقبال وتخزين هذه البيانات في متغير ما، فهذه الحالة نكتب هذا Logic البرمجي في الدالة ngOnInit.

#### 4.6.1. ngDoCheck:

وهي أيضاً من دوال lifecycle hooks، وتُستدعى أول مرة بعد ngOnInit ومن ثم يتم استدعاءها في كل مرة يتم أي تغيير على component، أو أي component مرتبط فيه ونستطيع ان نقول ان هذه الدالة من أكثر الدوال التي يتم استدعاءها في مشاريع Angular، وبنفس الوقت يُنصح بعدم استخدامها مع الدالة ngOnChanges بنفس components، أما من الصور التي يتم استدعاءها على سبيل المثال أي تغيير يحدث من المستخدم (ضغط على زر معين، قام بكتابة بيانات ما، الخ) سواء كان هذا الحدث الذي اجراه المستخدم قام بتغيير شيء ما في التطبيق او لم يقم، فعند ضغطه فقط على زر ولو لم يحدث شيء فإن angular يقوم بتفعيل هذه الدالة وعمل check لجميع أجزاء component وأي component مرتبط به لتأكد من موجود أي تعديلات حدثت من عدمه وعند وجود أي تعديلات يتم تحديث template بناءً على هذه التغييرات الخاصة، ومن صورها أيضاً في حال استقبال بيانات من قاعدة بيانات معينة وفي كل تحديث ديناميكي يتم على هذه البيانات أيضاً يتم استدعاء هذه الدالة، لذلك نستطيع ان نقول ان أي تعديل او حدث يتم على هذا component مهما كان وسواء تم تعديل او لم يتم تعديل فإن هذه الدالة سوف تُستدعى، وليس هذا فحسب فلو كان لدينا مجموعة من components المتداخلة والتي تربطها علاقة ما فإن أي تعديل على احد هذه component فإن هذه الدالة سوف يتم تشغيلها وعمل check بجميع هذه components، وهذا يرجع إلى آلية يتبعها Angular بشكل افتراضي تسمى Change Detection، لذلك عزيزي المتعلم يجب الانتباه وعدم استخدام هذه الدالة إلا في الحاجة الماسة لها لأن أي كود يتم كتابته يتم تكراره بشكل كبير وهو ما قد يستنزف الذاكرة RAM بالإضافة إلى تقليل من أداء التطبيق لديك بسبب تكرار العمليات الغير ضرورية.

أما لماذا Angular قدمت لنا هذه الدالة ونصح الفريق المختص بالتعامل معها بحذر، لأن هنالك بعض الحالات التي لا يستطيع Angular الكشف عن هذه التعديلات بشكل افتراضي ولذلك لابد من القيام بها يدوياً، منها على سبيل المثال في حال كان لدينا بيانات تم تمريرها من component الأب إلى component الأبن وكانت هذه البيانات على شكل كائن object لذلك في حال تعديل قيمة من قيم هذا الكائن في component الأب ولم يتم التعديل على Reference الخاص بالكائن، فإن Angular لن يلاحظ هذا التعديل ولابد من القيام بهذا بشكل يدوي، والسبب ان Angular يُراقب Object Reference وفي حال أي تعديل عليه سوف يلاحظه، وبإذن الله سوف نتطرق إلى هذا الأمر بشيء من التفصيل عند كلامنا في فصل مواضيع متقدمة في components ومنها Angular Change Detection Mechanism. اما ما يجب منك معرفته هنا ان هنالك دالة تسمى ngDoCheck وتستخدم لعمل check في أي تحديث او تعديل يتم على أي جزء من أجزاء تطبيق Angular.

### 5.6.1. ngAfterContentInit:

يتم تشغيل هذه الدالة مرة واحدة في حال اكتمال بناء content الخاص بالcomponent الأبن داخل component الأب، ويتم تشغيلها بعد الدالة ngDoCheck.

وهي مرتبطة بمجموعة من المفاهيم مثل ng-content وviewContentChild، لذلك سوف نؤجل إعطاء مثال عليها إلى ان نشرح هذه المفاهيم لاحقاً في الجزء الخاص بمواضيع متقدمة في components.

### 6.6.1. ngAfterContentChecked:

وهذه الدالة مشابهة في عملها ngDoChecked ولكن تختص في عمل check على content المعروض في component الأب والخاص بcomponent الأبن، وهي تأتي اول مرة بعد ngAfterContentInit ومن ثم في حالة وجود أي تعديل على هذا content فإن Angular يقوم بتشغيل هذه الدالة بعد الدالة ngDoCheck.

### 7.6.1. ngAfterViewInit:

ويتم تشغيل هذه الدالة مرة واحدة في حال اكتمال بناء content بالإضافة إلى template او ما يسمى view الخاص بالcomponent، وجميع template الخاص بالأبناء إن وجد، ويتم استدعاءها مرة واحدة في البداية بعد دوال ngAfterContentInit وngAfterContentChecked، وهي ايضاً مرتبطة ببعض المفاهيم وهي ViewChild وViewChildren، لذلك سوف نؤجل إعطاء مثال عليها إلى ان نشرح هذه المفاهيم.

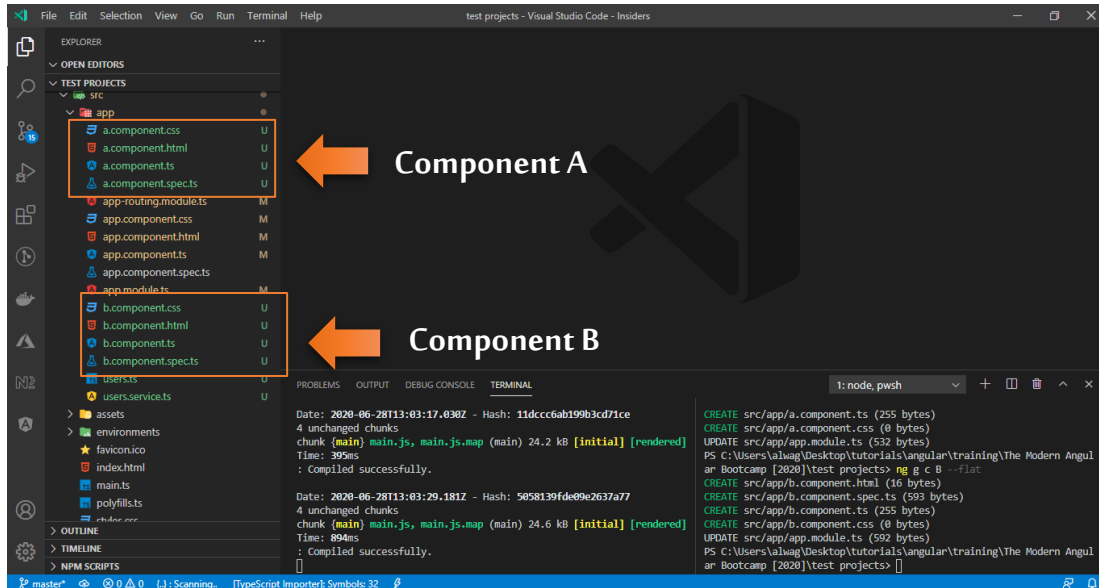
### 8.6.1. ngAfterViewChecked:

ونفس ما قيل في ngAfterContentChecked يُقال هنا، والفرق في الترتيب حيث انها تأتي في اول بعد ngAfterViewInit ومن ثم تُستدعى في كل مرة يتم فيها عمل أي اجراء على view الخاص بtemplate الأب او الأبن.

### 9.6.1. ngOnDestroy:

يتم استدعاء هذه الدالة عند الغاء وحذف component من DOM، ولفهمها بشكل أوضح لنقوم بإعطاء مثال، ووجب ان أوضح هنا ان هنالك بعض التقنيات التي لا يجب عليك فهمها لأنه ليس هنا المقام لشرحها وهي تقنيات Angular Routing، وقد تكلمت عن هذه التقنية بالتفصيل في الكتاب الرابع من هذه السلسلة.

الآن لنقوم بإنشاء اثنين components، وليكن الأول اسمه A والثاني اسمه B، (لمعرفة كيفية امشاء components والتعامل مع Angular CLI الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كالتالي:



ولنقم بتهيئة Routing (ولمعرفة كيفية التعامل مع Routing الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular Routing and Modules)، كالتالي:

```

app-routing.module.ts ملف
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AComponent } from './a.component';
import { BComponent } from './b.component';

const routes: Routes = [
  {path: 'componentA', component: AComponent},
  {path: 'componentB', component: BComponent},
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule { }

```

بعد تهيئة Routing لنقم بتهيئة Navigation لكل component، كالتالي:

```

app.component.html ملف
<a [routerLink]="['./componentA']">Show Component A</a>
<a [routerLink]="['./componentB']">Show Component B</a>

<div class="outlet">
  <router-outlet></router-outlet>
</div>

```

```

</div>

<div>
  <ul>
    <li *ngFor="let user of users$ | async">
      
      <h3>{{ user.name }}</h3>
      <p>{{ user.email }}</p>
    </li>
  </ul>
</div>

```

وأخيراً لنقوم ببعض اللمسات الجمالية في ملف style، كالتالي:

ملف app.component.css

```

* {
  margin: 0;
  padding: 0;
}

a {
  margin-left: 10%;
}

.outlet {
  text-align: center;
  border: 1px solid black;
}

div {
  margin: 20px;
}

ul {
  list-style-type: none;
  width: 500px;
}

h3 {
  font: bold 20px/1.5 Helvetica, Verdana, sans-serif;
}

li img {
  float: left;
  margin: 0 15px 0 0;
}

li p {
  font: 200 12px/1.5 Georgia, Times New Roman, serif;
}

li {
  padding: 10px;
}

```

```

overflow: auto;
border: .1px solid #e6e6e6;
}

li:hover {
  background: #eee;
  cursor: pointer;
}

```

جميع ما تم ذكره سابقاً ليس هنا المقام لشرحه، وما يهمنا هنا هو ما سوف يُقال الآن، حيث سوف نقوم بعمل implements لي interface ذو الاسم OnDestroy في ملف class الخاص بكل component سواء كان A و B، ونستدعي الدالة ngOnDestroy ونمظهر سالة في console في حال تم تدمير هذه component وحذفه من DOM، كالتالي:

ملف a.component.ts

```

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

@Component({
  selector: 'app-a',
  templateUrl: './a.component.html',
  styleUrls: ['./a.component.css']
})
export class AComponent implements OnInit, OnDestroy {

  constructor() { }

  ngOnInit(): void {
  }

  ngOnDestroy() {
    console.log('Component A ==> Destroyed');
  }
}

```



ملف b.component.ts

```

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

@Component({
  selector: 'app-b',
  templateUrl: './b.component.html',
  styleUrls: ['./b.component.css']
})
export class BComponent implements OnInit, OnDestroy {

  constructor() { }

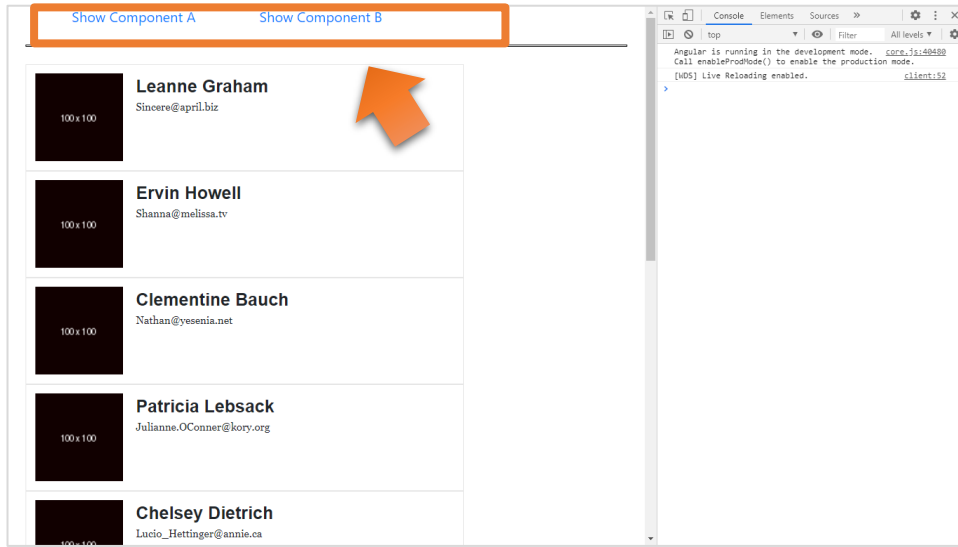
  ngOnInit(): void {
  }

  ngOnDestroy() {
    console.log('Component B ==> Destroyed');
  }
}

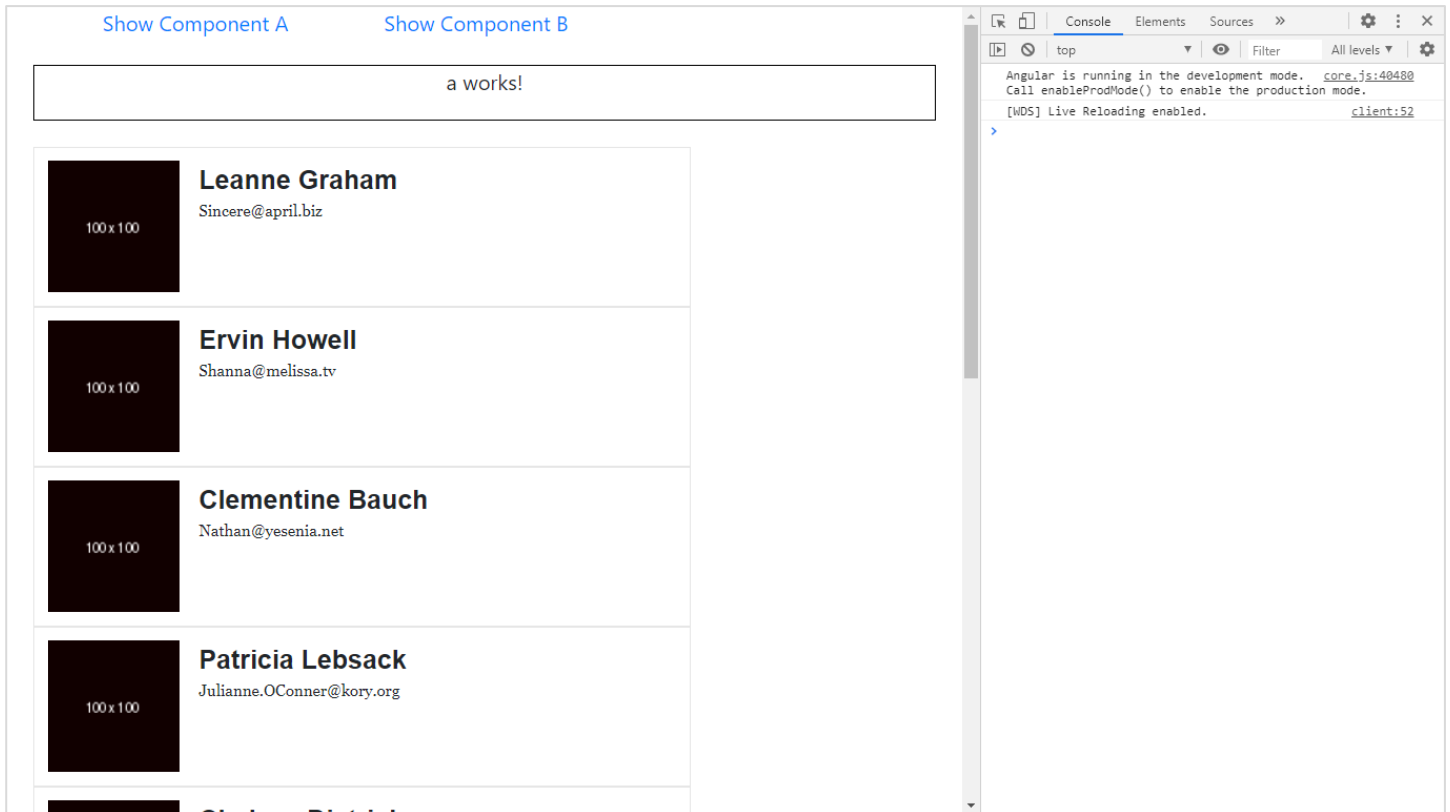
```



ولنرى النتيجة في المتصفح، كالتالي:

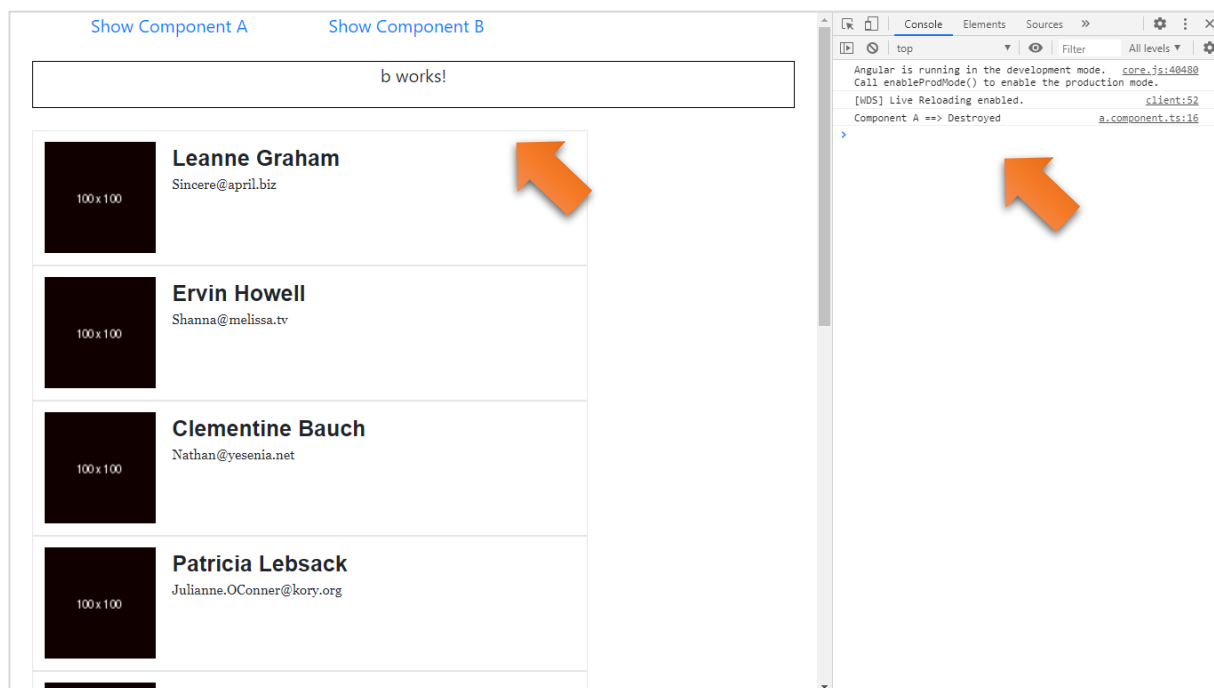


في بداية تشغيل التطبيق نلاحظ وجود الروابط Show Component A لعرض محتويات هذا component والرابط الثاني لعرض محتويات Component ذو الاسم B، الآن لنضغط على Show Component A، ونرى النتيجة:

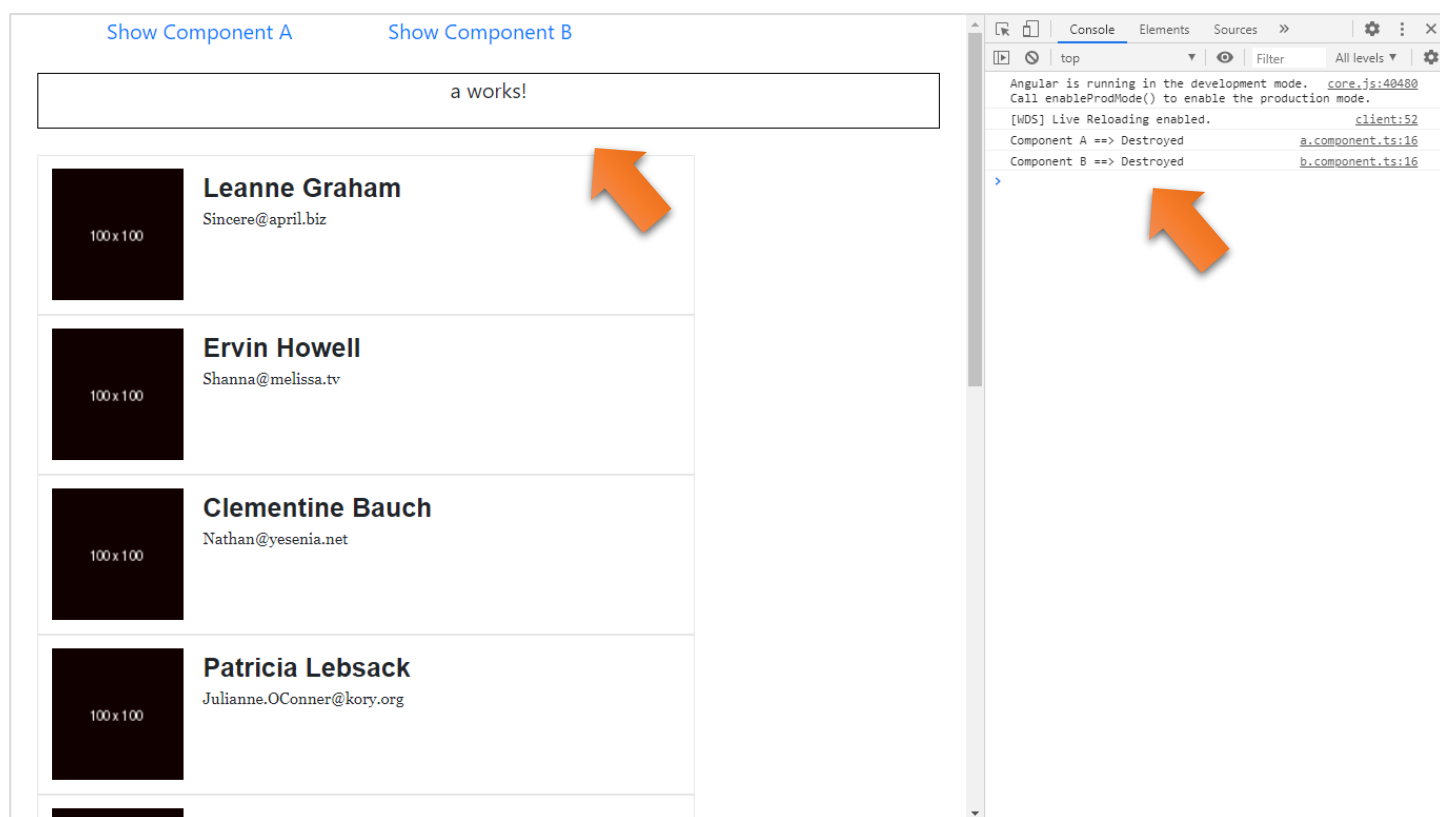


نلاحظ ظهر لنا محتويات هذا Component وهو الجملة الموجودة في ملف a.component.html، وهذا معناه ان Angular قام ببناء هذا component بدأ من constructor وانتهاءً بي ngAfterViewChecked، ولكن لم يستدعي دالة ngOnDestroy إلى الآن لأن هذا component موجود في DOM، الآن لنقوم باختيار الرابط الثاني Show Component B، وفي

هذه الحالة سوف يقوم Angular بحذف Component A من DOM وبناء Component B مكانه، وفي هذه الحالة سوف يتم استدعاء الدالة ngOnDestroy وإظهار الرسالة Component A ==> Destroyed في console، كالتالي:



الآن قام Angular ببناء Component B وتدمير Component A لذلك اظهر هذه الرسالة، الآن لنقم باختيار Component A ونرى النتيجة:



لذلك نستخدم هذه الدالة لتنظيف أي component قبل حذفه من DOM مثل عمل unsubscribe لبيانات معينة.



وبعد عرضنا لجميع هذه lifecycle hooks، لنقم بإعطاء مثال شامل نبين فيه بشكل (مجمل) ترتيب هذه الدوال وكم مرة يتم استدعاء هذه الدوال، وسوف نقوم بالتطبيق على المثال السابق، حيث انه كما قلنا سابقاً أن Root Component يعتبر هو الأب لجميع Components الأخرى لذلك A Component و B Component يعتبرون أبناء لي Root، وسوف نستفيد من هذا الأمر بتطبيق مفاهيم lifecycle hooks بشكل مختصر ومجمل، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy
} from '@angular/core';
import { Users } from './users';
import { UsersService } from './users.service';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy {

  users$: Observable<Users[]>;

  constructor(private users: UsersService) {
    console.log('this Message From Root Component Constructor');
  }

  ngOnInit(): void {
    this.users$ = this.users.getAllUsers();
    console.log('this Message From Root Component ngOnInit');
  }

  ngDoCheck(): void {
    console.log('this Message From Root Component ngDoCheck');
  }
}
```

```

ngAfterContentInit(): void {
  console.log('this Message From Root Component ngAfterContentInit');
}

ngAfterContentChecked(): void {
  console.log('this Message From Root Component ngAfterContentChecked');
}

ngAfterViewInit(): void {
  console.log('this Message From Root Component AfterViewInit');
}

ngAfterViewChecked(): void {
  console.log('this Message From Root Component AfterViewChecked');
}

ngOnDestroy(): void {
  console.log('this Message From Root Component OnDestroy');
}
}

```

نلاحظ اننا لم نستدعي ngOnChanges لأنه كما قلنا سابقاً لا يُفضل جمع كلاً من ngOnChanges و ngDoCheck في نفس Component، الآن لنحفظ التعديلات ونرى النتيجة في المتصفح:

Show Component A

Show Component B

100x100

Leanne Graham

Sincere@april.biz

100x100

Ervin Howell

Shanna@melissa.tv

100x100

Clementine Bauch

Nathan@yesenia.net

100x100

Patricia Lebsack

Julianne.OConner@kory.org

100x100

Chelsey Dietrich

Lucio\_Hettinger@annie.ca

top

Filter

All levels

this Message From Root Component Constructor

app.component.ts:33

this Message From Root Component ngOnInit

app.component.ts:38

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentInit

app.component.ts:46

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component AfterViewInit

app.component.ts:54

this Message From Root Component AfterViewChecked

app.component.ts:58

Angular is running in the development mode. core.js:40480

Call enableProdMode() to enable the production mode.

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component AfterViewChecked

app.component.ts:58

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component AfterViewChecked

app.component.ts:58

[WDS] Live Reloading enabled.

client:52

نلاحظ كيف كان الترتيب حيث في الشكل (1) قام بعمل استدعاء لجميع دوال lifecycle hooks الخاصة بي Root Component الأب، ومن ثم ايضاً في الشكل (2) والشكل (3) استدعى دوال lifecycle hooks الخاصة بعمل checked وهم

بالترتيب ngDoCheck و ngAfterContentChecked و ngAfterViewChecked، مرتين لكل دالة، والسبب في ذلك انه هنالك ngFor وتكرار للبيانات لذلك عن أي تعديل للبيانات يتم استدعاء هذه الدوال، ومن هذا المنطلق يجب الحرص عند استخدامها، ونفس الآلية لنقوم بتطبيقها على الـ components الأبناء، كالتالي:

ملف a.component.ts

```
import {
  Component,
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy
} from '@angular/core';

@Component({
  selector: 'app-a',
  templateUrl: './a.component.html',
  styleUrls: ['./a.component.css']
})

export class AComponent implements
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy {

  constructor() {
    console.log('this Message From (A) Component Constructor');
  }

  ngOnInit(): void {
    console.log('this Message From (A) Component ngOnInit');
  }

  ngDoCheck(): void {
    console.log('this Message From (A) Component ngDoCheck');
  }

  ngAfterContentInit(): void {
    console.log('this Message From (A) Component ngAfterContentInit');
  }

  ngAfterContentChecked(): void {
    console.log('this Message From (A) Component ngAfterContentChecked');
  }
}
```

```

ngAfterViewInit(): void {
  console.log('this Message From (A) Component ngAfterViewInit');
}

ngAfterViewChecked(): void {
  console.log('this Message From (A) Component ngAfterViewChecked');
}

ngOnDestroy(): void {
  console.log('this Message From (A) Component ngOnDestroy');
}
}

```

#### ملف b.component.ts

```

import {
  Component,
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy
} from '@angular/core';

@Component({
  selector: 'app-b',
  templateUrl: './b.component.html',
  styleUrls: ['./b.component.css']
})

export class BComponent implements
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy {

  constructor() {
    console.log('this Message From (B) Component Constructor');
  }

  ngOnInit(): void {
    console.log('this Message From (B) Component ngOnInit');
  }

  ngDoCheck(): void {
    console.log('this Message From (B) Component ngDoCheck');
  }
}

```

```

ngAfterContentInit(): void {
  console.log('this Message From (B) Component ngAfterContentInit');
}

ngAfterContentChecked(): void {
  console.log('this Message From (B) Component ngAfterContentChecked');
}

ngAfterViewInit(): void {
  console.log('this Message From (B) Component ngAfterViewInit');
}

ngAfterViewChecked(): void {
  console.log('this Message From (B) Component ngAfterViewChecked');
}

ngOnDestroy(): void {
  console.log('this Message From (B) Component ngOnDestroy');
}
}

```

الآن لنذهب إلى المتصفح ونرى النتيجة:

Show Component A

Show Component B

100x100

**Leanne Graham**  
 Sincere@april.biz

100x100

**Ervin Howell**  
 Shanna@melissa.tv

100x100

**Clementine Bauch**  
 Nathan@yesenia.net

100x100

**Patricia Lebsack**  
 Julianne.OConner@kory.org

100x100

**Chelsey Dietrich**  
 Lucio\_Hettinger@annie.ca

Root Component

الرابط localhost:4200

clear console

top

this Message From Root Component Constructor

app.component.ts:33

this Message From Root Component ngOnInit

app.component.ts:38

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentInit

app.component.ts:46

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component ngAfterViewInit

app.component.ts:54

this Message From Root Component ngAfterViewChecked

app.component.ts:58

Angular is running in the development mode. core.js:40480

Call enableProdMode() to enable the production mode.

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component ngAfterViewChecked

app.component.ts:58

this Message From Root Component ngDoCheck

app.component.ts:42

this Message From Root Component ngAfterContentChecked

app.component.ts:50

this Message From Root Component ngAfterViewChecked

app.component.ts:58

[WDS] Live Reloading enabled.

client:52

الآن لنضغط على زر clear console ومن ثم نضغط على الرابط Show Component A، كالتالي:

The screenshot shows a web application with a header containing two buttons: "Show Component A" and "Show Component B". Below the header is a list of people, each with a placeholder image (100x100), a name, and an email address. The list includes Leanne Graham, Ervin Howell, Clementine Bauch, and Patricia Lebsack. To the right of the list, a console log is visible, showing messages from the Angular component, including "this Message From (A) Component Constructor" and "this Message From Root Component ngDoCheck".

## 7.1 .Template Reference

من الأمور المهمة والتي قدمها لنا Angular هي Template Reference وتسمى أيضاً Template Variable، وهي إعطاء متغير لأي template او بمعنى آخر اعطاء متغير لأي عنصر داخل template بحيث يصبح هذا المتغير يشير إلى هذا العنصر ويمتلك حق الوصول إلى جميع خصائصه، ونستطيع مناداة هذا العنصر سواء بنفس template الذي تم تعريف هذا المتغير فيه او مناداته بنفس ملف class لهذا component الموجود فيه ملف template الذي يحتوي هذا العنصر الذي تم تعريف المتغير له، ويتم تعريف المتغير بوضع علامة # قبله.

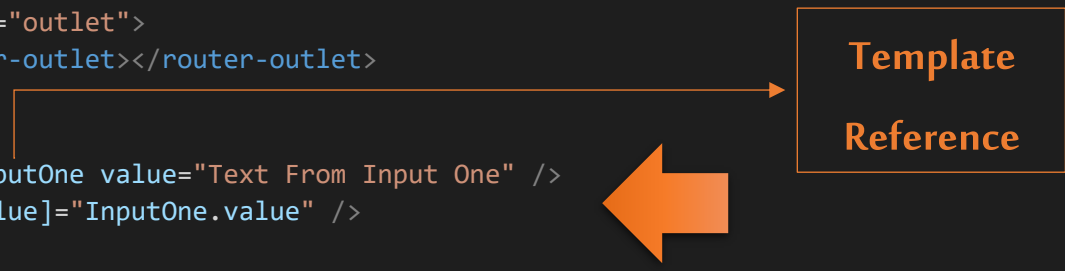
ولتوضيح ما قيل سابقاً لنعطي مثال، وذلك عن طريق ادراج عنصرين input الأول نعطيهِ متغير (Template Reference) اما الثاني فنجعل قيمة value له هي نفسها قيمة value للـ Input الأول، كالتالي:

```
app.component.html
<a [routerLink]="['./componentA']">Show Component A</a>
<a [routerLink]="['./componentB']">Show Component B</a>

<div class="outlet">
  <router-outlet></router-outlet>
</div>

<input #InputOne value="Text From Input One" />
<input [value]="InputOne.value" />

<div>
  <ul>
    <li *ngFor="let user of users$ | async">
      
      <h3>{{ user.name }}</h3>
      <p>{{ user.email }}</p>
    </li>
  </ul>
</div>
```



نلاحظ اننا اعطينا الـ input الأول متغير واسميناه #InputOne وبذلك اصبح هذا المتغير يشير إلى هذا input ويمتلك حق الوصول إلى جميع خصائصه، ومن هذه الخصائص خاصية value لذلك جعلنا قيمة input الثاني تساوي القيمة الموجودة في المتغير InputOne. اما النتيجة فسوف تكون كالتالي:

Show Component A

Show Component B

b works!

1

Text From Input One

2

Text From Input One

100 x 100

**Leanne Graham**  
Sincere@april.biz

100 x 100

**Ervin Howell**  
Shanna@melissa.tv

100 x 100

**Clementine Bauch**  
Nathan@yesenia.net

100 x 100

**Patricia Lebsack**  
Julianne.OConner@kory.org

Console

top

Filter

All levels

this Message From Root Component

app.component.ts:54

ngAfterViewInit

this Message From Root Component

app.component.ts:58

ngAfterViewChecked

Angular is running in the development

core.js:40480

mode. Call enableProdMode() to enable the production

mode.

this Message From (B) Component

b.component.ts:28

Constructor

this Message From Root Component

app.component.ts:42

ngDoCheck

this Message From Root Component

app.component.ts:50

ngAfterContentChecked

this Message From (B) Component

b.component.ts:32

ngOnInit

this Message From (B) Component

b.component.ts:36

ngDoCheck

this Message From (B) Component

b.component.ts:40

ngAfterContentInit

this Message From (B) Component

b.component.ts:44

ngAfterContentChecked

this Message From (B) Component

b.component.ts:48

ngAfterViewInit

this Message From (B) Component

b.component.ts:52

ngAfterViewChecked

this Message From Root Component

app.component.ts:58

ngAfterViewChecked

this Message From Root Component

app.component.ts:42

ngDoCheck

this Message From Root Component

app.component.ts:50

ngAfterContentChecked

this Message From (B) Component

b.component.ts:36

ngDoCheck

this Message From (B) Component

b.component.ts:44

ngAfterContentChecked

this Message From (B) Component

b.component.ts:52

ngAfterViewChecked

this Message From Root Component

app.component.ts:58

ngAfterViewChecked

[WDS] Live Reloading enabled.

client:52

رقم واحد هو input الذي اضعنا له Template Reference (#InputOne)، اما رقم اثنين هو input الذي اسندنا له القيمة value الخاصة بي input الأول.

أما الصيغة الموجودة وهي وضع خاصية value بين مربعين [ ] فهذا يسمى Property Binding، وهو ما سوف نتكلم عنه في الفصل القادم بإذن الله.



# الفصل الثاني

## Components Communication

## 1.2. المقدمة:

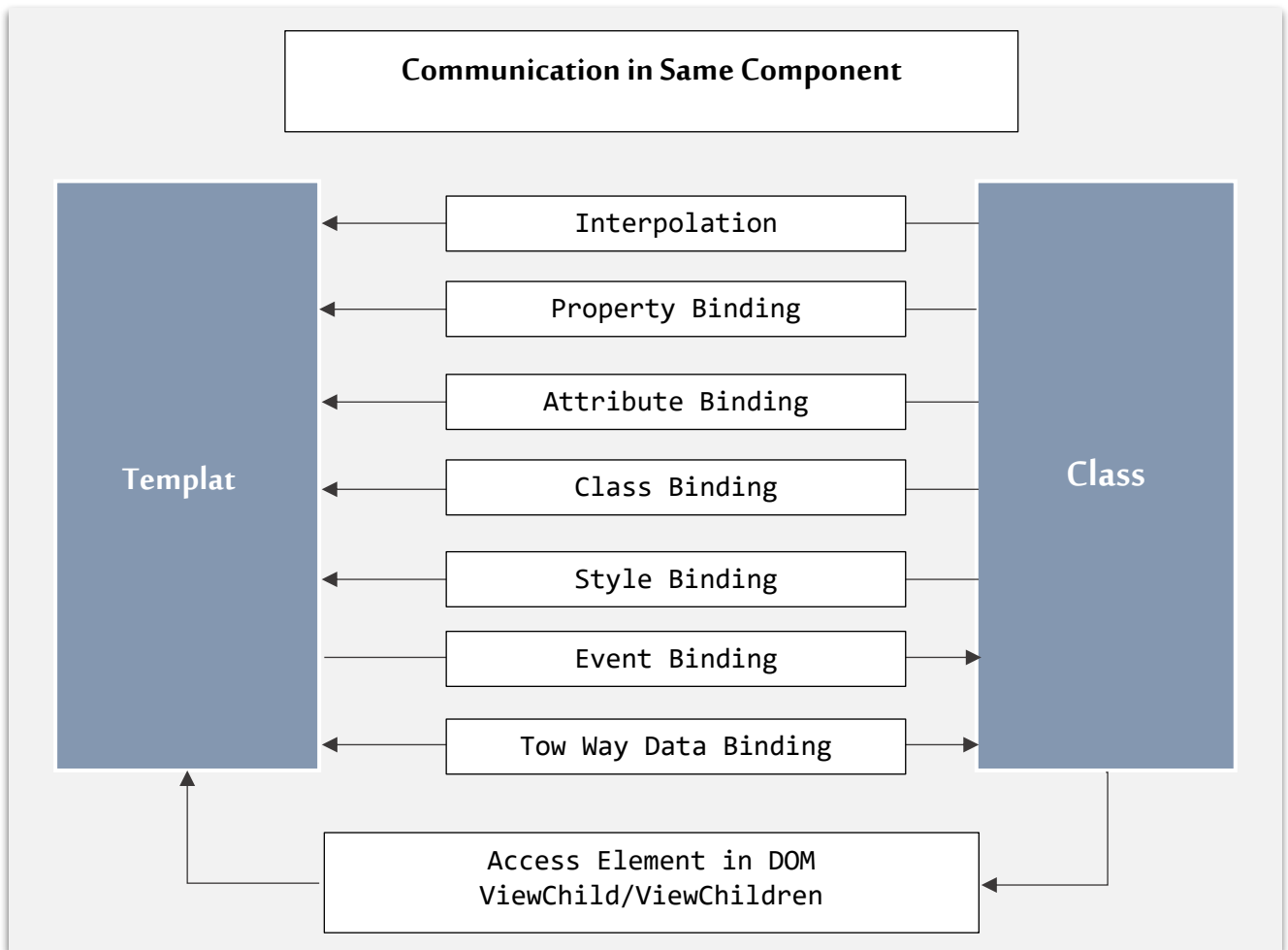
بعدما قدمنا مدخل إلى components في الفصل السابق، ففي هذا الفصل سوف نتكلم عن طريقة الاتصال والتواصل وتبادل البيانات بين ملفات component الواحد من جهة، وبين Components المختلفة من جهة أخرى، ومن مكامن القوة لدى Angular تقديمها للمطورين طرق متعددة ومبسطة وميسرة لتبادل البيانات بين components، ونستطيع تقسيم هذا التواصل بشكل عام إلى قسمين رئيسيين:

القسم الأول: التواصل بين ملفات component الواحد.

القسم الثاني: التواصل بين components المختلفة، سواء كانت هذه Components التي ترتبط بينهم علاقة كعلاقة Component الأب مع الأبن والعكس الأب مع الأب أو مع الأخ والأخ أو التي لا تربطها أي علاقة.

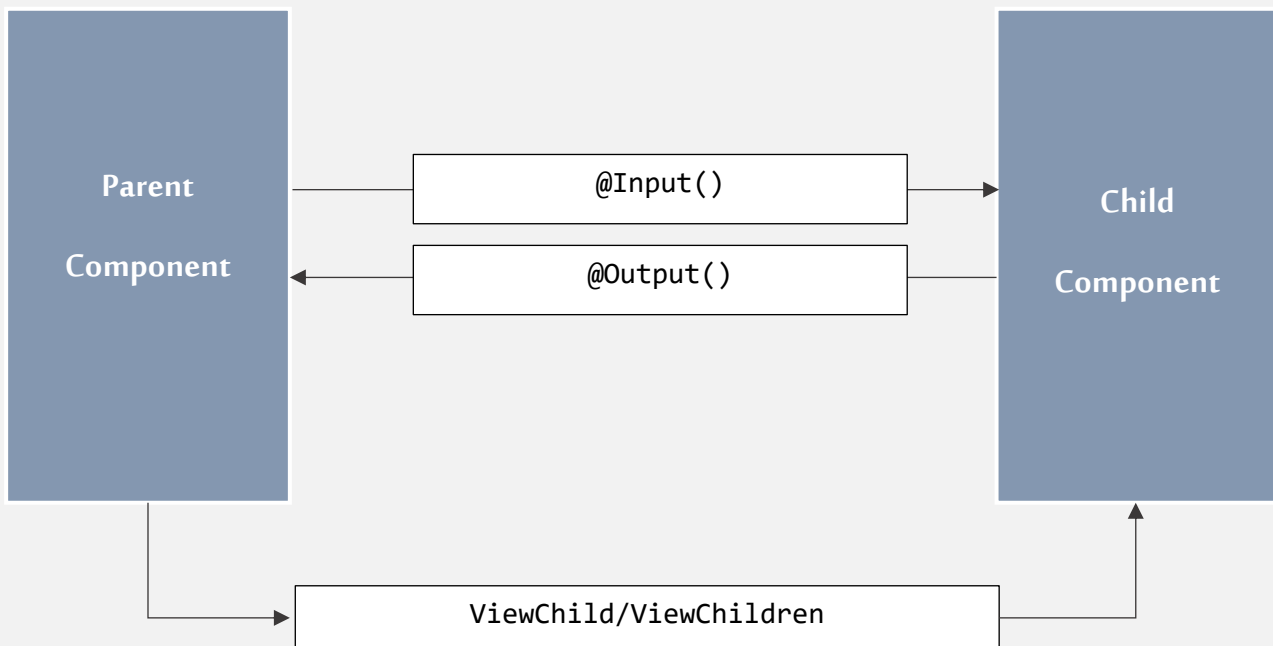
وسوف نتطرق بإذن الله لجميع هذه الحالات السابقة وكيفية تبادل البيانات بينها وطرق التواصل بين هذه components.

ويمكن تمثيل طرق تبادل البيانات والتواصل بين ملفات Component الواحد بالشكل التالي:



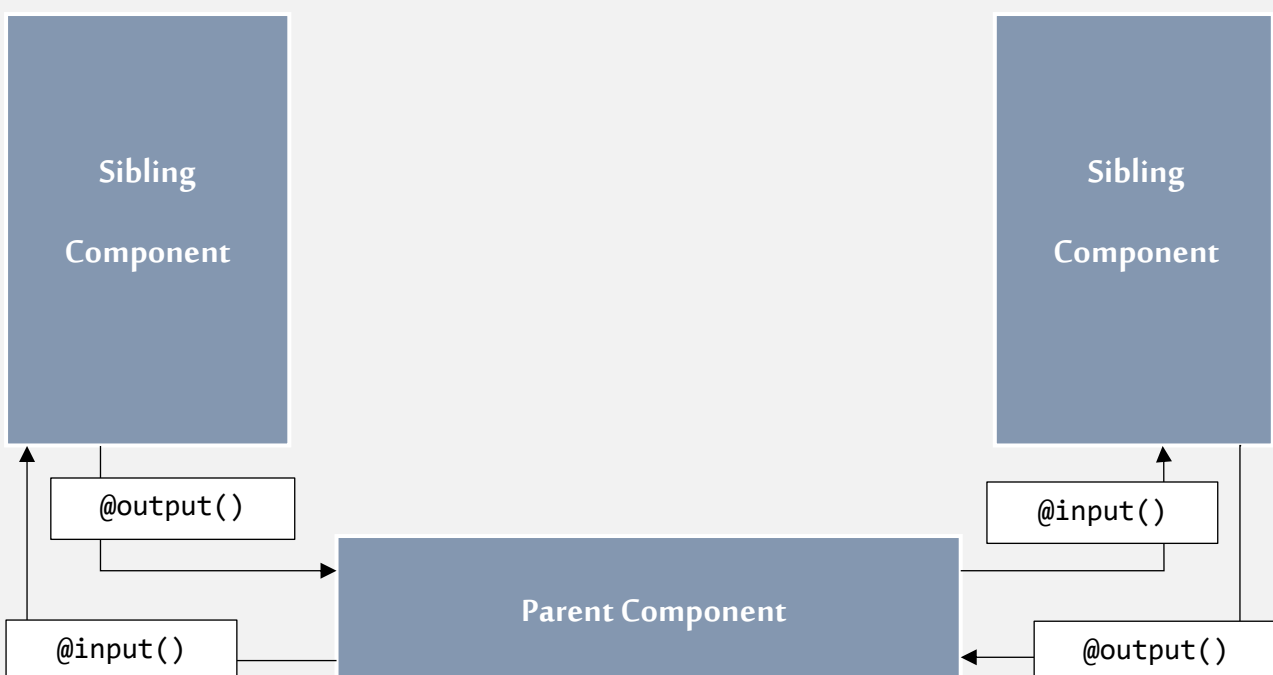
وايضاً يمكن تمثيل تبادل البيانات بين مجموعة من Components المختلفة التي ترتبط بينهما علاقة بالشكل التالي:

### Communication Between (Parent <==> Child) Components

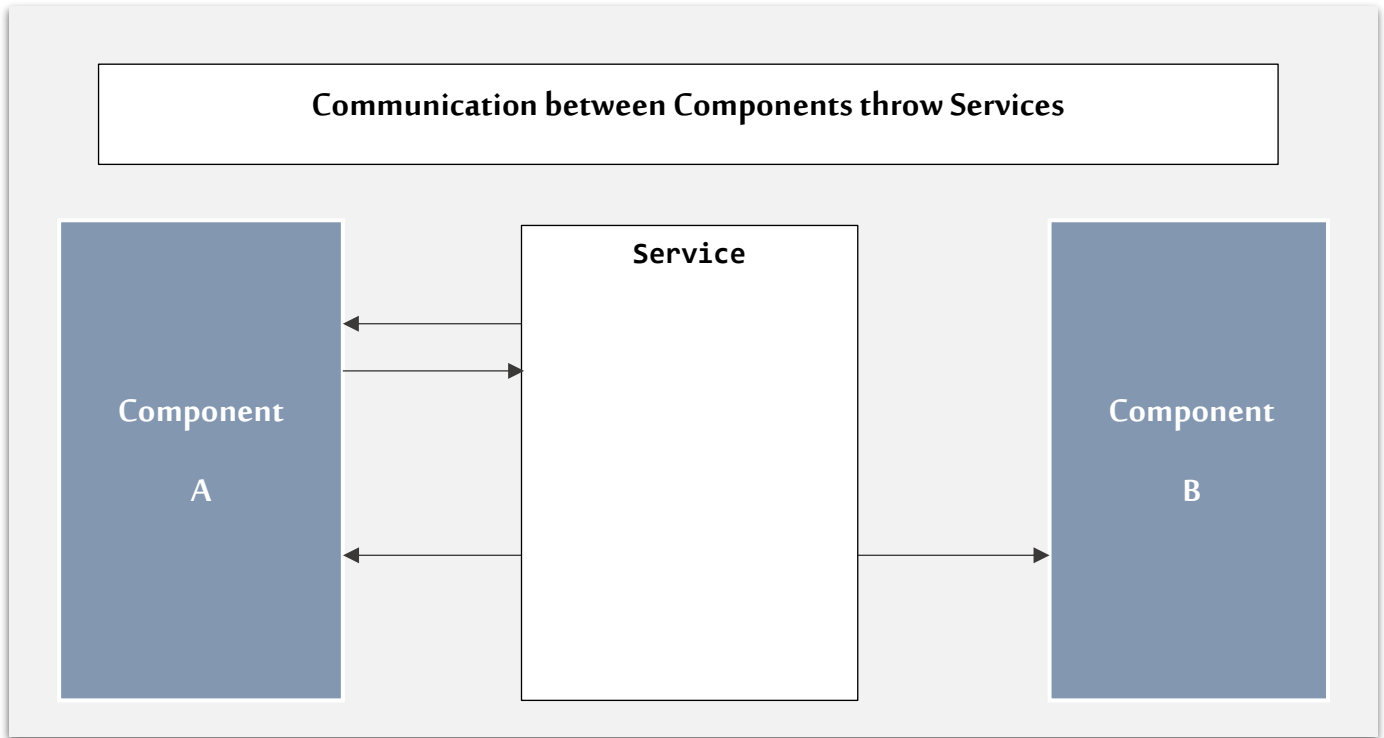


اما الشكل التالي فيمثل اشكال التواصل بين Components التي ترتبط بينهما علاقة أخوة، كالتالي:

### Communication between Sibling Components



اما الشكل الأخير فندستطيع تمثيل طرق التواصل بين components التي لا ترتبط بينها أي علاقة.



وبعد استعراضنا لجميع أنواع التواصل مع الأشكال التوضيحية لكل نوع بشكل مجمل، سوف ننتقل الآن إلى الشرح المفصل لكل نوع لكي تتضح الصورة لك عزيزي المتعلم وتكون بنية معرفتك على أرض صلبة.

## 2.2. التواصل بين ملفات Component الواحد:

هنالك أنواع متعددة لتواصل بين ملفات نفس component، منها ما هو نقل البيانات باتجاه واحد من ملف class إلى ملف template ومنها ما هو موجه من ملف template إلى ملف class وفي هذه الحالة يسمى One Way Data Binding، ومنها ما هو يعمل في كلا الاتجاهين وفي هذه الحالة يسمى Two Way Data Binding، وسوف نستعرض جميع هذه الأنواع، ولكن قبلها لنقم بإنشاء مشروع Angular جديد وليكن اسمه ComponentCommunication (لمعرفة طريقة انشاء مشروع جديد الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كالتالي:

### 1.2.2. Interpolation Data Binding:

ويعتبر من أنواع One Way Data Binding حيث يتم التحكم في البيانات وارسالها في اتجاه واحد من ملف class وعرضها في ملف template على شكل متغيرات (خصائص) ويتم عرضها بين اقواس المجموعة الزوجية {{ } }، ولا يمكن تعديل القيم إلا في ملف class فقط، فملف template يتم فيه عرض القيم المخزنة في هذه الخصائص، ولتوضيح لنعطي مثال، لنفرض ان لدينا متغير ولنسند له قيمة، ولنستخدم طريق Interpolation لعرض البيانات في ملف template، كالتالي:

```
app.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```

styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  name = 'Hello Faisal';

  constructor() {}

  ngOnInit(): void {}
}

```

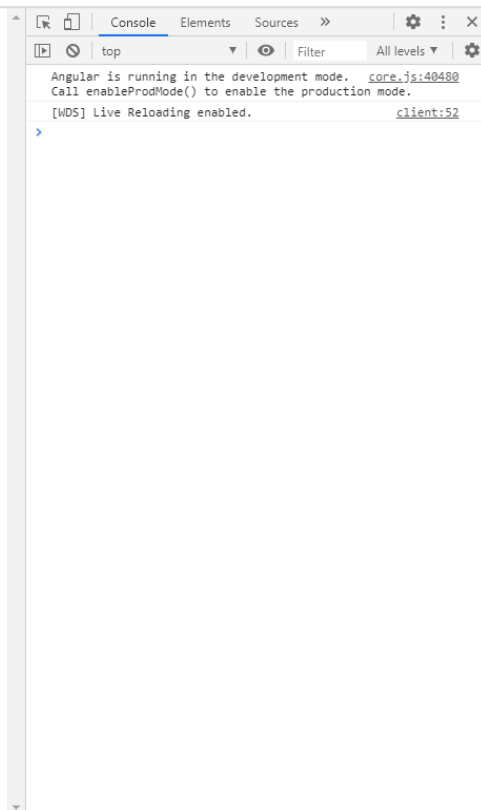
نلاحظ اننا قمنا بتعريف متغير (خاصية) باسم name وأسندنا لها قيمة معينة، ولعرض هذه القيمة باستخدام Interpolation، نقوم بعمل التالي:

ملف app.component.html

```
<h3>{{name}}</h3>
```

نلاحظ اننا استخدمنا اقواس المجموعة الزوجية لعرض الخاصية name في ملف template، اما النتيجة فتكون كالتالي (لابد أن نعمل serve للمشروع لكي نشاهد النتيجة في المتصفح وذلك عن طريق كتابة الأمر ng s -o ولمعرفة طرق التعامل مع أوامر Angular CLI الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup):

Hello Faisal



ظهرت لنا النتيجة كما هو متوقع وهي قيمة الخاصية name وذلك لأننا عملنا لها Bind باتجاه واحد One Way من ملف class إلى ملف template، ولا يمكن تعديل قيمها إلا عن طريق ملف class، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

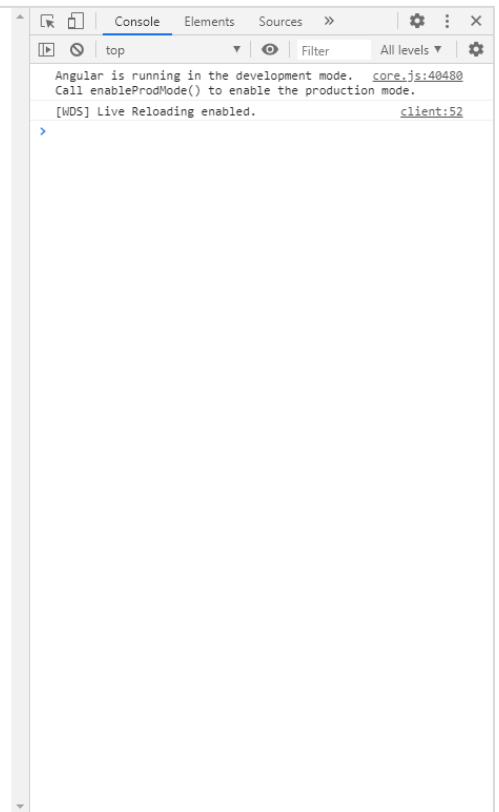
export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  constructor() {}

  ngOnInit(): void {}
}
```

اما النتيجة فتكون (يجب حفظ التعديلات لكي يقوم Angular بتحديث المتصفح وإظهار التعديلات الجديدة):

Welcome Faisal



مع العلم انه لا يلزم وضع اقواس المجموعة داخل تاغات HTML، فنستطيع وضعها بأي مكان في ملف template وسوف تعمل بشكل صحيح، كالتالي:

ملف app.component.html

```
{{name}}
```

ويجب التنويه ان Angular سوف يعامل أي شيء مكتوب بداخل هذه الاقواس على انه كود برمجي، بحيث لو كتبنا أي كلمة فانه سوف يعاملها على انها خاصية وسوف يبحث عنها في ملف class وفي حال عدم وجودها سوف تظهر لنا رسالة خطأ.

وفي حال اردنا كتابة قطعة نصية بشكل مباشر بين هذه الاقواس نستطيع اضافتها عن طريق إضافة علامتي التنصيص سواء المفردة او الزوجية ومن ثم كتابة النص الذي نريده، وبنفس الوقت نستطيع إضافة ارقام بشكل مباشر، او نجري عملية حسابية بشكل مباشر لأنه كما قلت قبل قليل أي شيء يُكتب بداخل هذه الاقواس سوف يُعتبر أكواد برمجية:

ملف app.component.html

```
{{ 'Message: ' + name }}
```

```
<br><br>
```

```
{{1234 + name}}
```

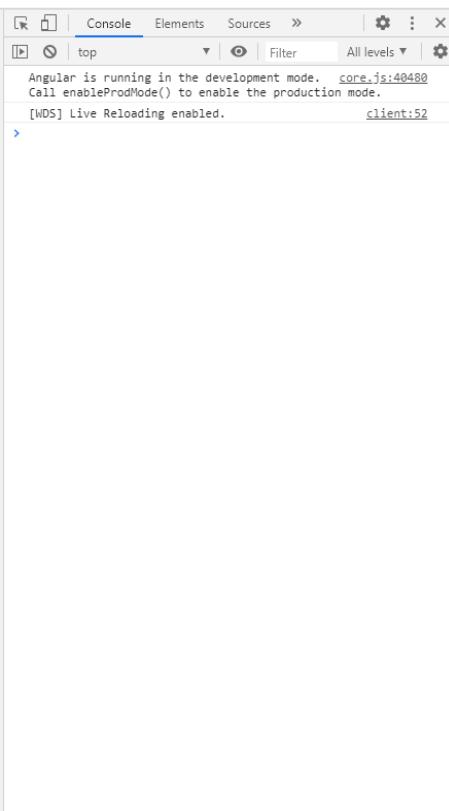
```
<br><br>
```

```
{{5 + 10}}
```

Message: Welcome Faisal

1234Welcome Faisal

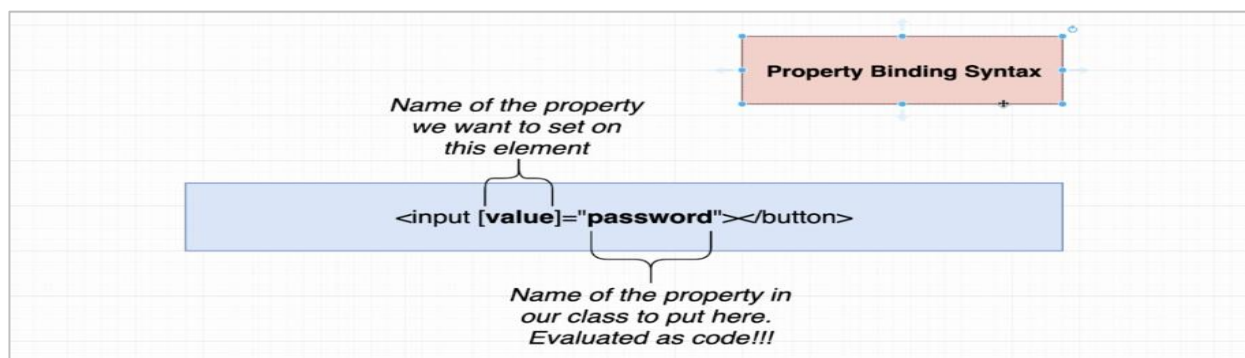
15



## 2.2.2. Property Data Binding

وهي ايضاً من One Way Data Binding حيث تكون باتجاه واحد من ملف class إلى ملف template، وكما هو معروف ان جميع تاغات HTML لها خصائص Property مثل خاصية src الخاصة بالتاغ img والتي نضيف فيها مسار الصورة التي نريد عرضها، او الخاصية disabled لتاغ button لتعطيله او تفعيله، او حتى خاصية value لتاغ input التي استعرضناها من قبل والقائمة تطول وليس هنا المكان لاستعراضها والتي يُفترض بك عزيزي المتعلم ان تكون مُلم بها، ولكن ما يهمنا هنا هو كيف يمكننا تمرير بيانات او قيم ديناميكية لهذه الخصائص، وحقيقة Angular قدمت لنا طريقة سهلة وبسيطة لعمل bind لأي

قيمة نريدها إلى أي خاصية، وذلك بكتابة اسم الخاصية بين اقواس المصفوفة []، ومن ثم اسناد أي متغير (خاصية) نريدها إلى هذه Property، ويمكن تمثيلها بالشكل التالي:



وكل ما قيل في Interpolation يُقال هنا من حيث أن أي شيء مكتوب بين علامتي التنصيص سوف يعتبره Angular أكواد برمجية وفي حال أردنا ان نمرر قطعة نصية وذلك بكتابة القيمة بين علامتي التنصيص المفردة ' ' فقط وليست المزدوجة، وايضاً نستطيع تمرير دالة وهذه الدالة لابد ان تُعيد return قيمة مناسبة للقيمة الخاصة بالخاصية، مثل الخاصية src في التاغ img متوقع ان يسند لها قيمة نصية تمثل مسار او رابط صورة معينة، وهكذا بقية الخصائص، ليس هذا فحسب بل نستطيع ايضاً تمرير Logic برمجي، مثلاً على شكل شرط برمجي يُعيد true او false للخصائص التي تكون قيمتها قيمة منطقية مثل الخاصية disabled.

ولتوضيح ما قيل لنعطي مثال، كالتالي:

```

ملف app.component.html
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  constructor() {}

  ngOnInit(): void {}

  inputValue() {
    return 'Value From Function';
  }
}

```



نلاحظ عرفنا متغير (خاصية) باسم val ودالة باسم inputValue تُعيد قطعة نصية، اما في ملف `templat` فسيكون كالتالي:

ملف `app.component.html`

```
{{ 'Message: ' + name }}  
<br><br> {{1234 + name}}  
<br><br> {{5 + 10}}  
  
<br><br> <input type="text" [value]="val" />  
<br><br> <input type="text" [value]=" 'val: ' + val " />  
<br><br> <input type="text" [value]="5 + 10" />  
<br><br> <input type="text" [value]="inputValue()" />  
<br><br> <button [disabled]="num === 5">Click Me!</button>
```

Message: Welcome Faisal

1234Welcome Faisal

15

My Nice Input

val: My Nice Input

15

Value From Function

Click Me!

Console Elements Sources  
top Filter All levels  
Angular is running in the development mode. `core.js:40480`  
Call `enableProdMode()` to enable the production mode.  
[WDS] Live Reloading enabled. `client:52`

### 3.2.2. Attribute Data Binding

تكلّمنا سابقاً عن Property Binding وهي عبارة عن خصائص Standard لأي عنصر من عناصر HTML، بحيث أن المتصفح عندما يقوم بقراءة جميع عناصر HTML يقوم بتحويلها إلى شجرة من الكائنات Objects في DOM وبنفس الوقت يقوم بقراءة جميع الخصائص الموجودة في العناصر ومقارنتها بالخصائص Standard الموجودة لديه فإذا كانت موجودة يقوم بتحويلها إلى خصائص one-to-one إلى هذه الكائنات في DOM، ولكن هنالك بعض الخصائص ليست Standard لعنصر HTML، ففي هذه الحالة نسميه مجازاً Attributes، ولكي نعمل Binding لهذا Attribute نكتب قبله فقط `attr` ومن ثم نقطة ومن ثم اسم Attribute.

مثال / `colspan` يعتبر Attributes بالنسبة للعنصر `<td>` وفي حال أردنا ان نعمل Biding له نكتب الامر التالي:

```
<td [attr.colspan]="count"></td>
```

بحيث ان count هو المتغير الذي يحمل القيمة التي نريد ان نعمل لها Binding لهذا Attribute الذي هو colspan.

مع العلم ان كتابة attr قبل Attribute ليس حكراً عليها فيمكن كتابته ايضاً قبل Property، لذلك في حال اردت ان لا تحدث مشكلة لديك ولا تعرف هل هذه Property ام Attribute فقم بكتابة attr معها جميعاً.

## 4.2.2. Class Data Binding:

وهي ايضاً من One Way Data Binding ومشابهه لحدّ ما مع Property Binding والفرق هنا انها تتعامل مع كلاسات css، حيث تتيح لنا ان نعمل bind لكلاس معين بشكل ديناميكي من ملف styles إلى ملف template وفق شرط معين ولا بد ان يُعيد هذا الشرط قيمة منطقية true أو false بحيث إذا كان true يضيف هذا الكلاس وإذا كان false يحذف هذا الكلاس، وهذا الشرط نستطيع كتابته بشكل مباشر او ننشأ دالة او متغير (كما فعلنا في السابق مع المتغير num والخاصية disabled)، لذلك نستطيع ان نقول ان Class Binding تعمل بالاتصال باتجاه واحد من ملف styles إلى ملف template وبنفس الوقت تحقق الاتصال من ملف class إلى ملف template، اما الصيغة العامة لها، هي:

"الشرط"=[اسم الكلاس.class]

ولتوضيح لنعطي المثال التالي، الذي نريد منه ان يقوم بتغيير خلفية الزر مع كل ضغطة بشكل ديناميكي، كالتالي:

اولاً ننشأ كلاسات css في ملف styles التي نريد ان نضيفها او نحذفها بشكل ديناميكي:

ملف app.component.css

```
.btn {
  width: 100px;
  left: calc(50% - 55px);
  top: 100px;
  color: white;
}

.redClass {
  background: red;
}

.greenClass {
  background: green;
}
```

وفي ملف class نعرف المتغيرات (الخصائص)، وبنفس الوقت نعرف دالة تقوم بتغيير قيمة هذه المتغيرات من true إلى false ومن false إلى true مع كل ضغطة، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```

}))

export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  constructor() {}

  ngOnInit(): void {}

  inputValue() {
    return 'Value From Function';
  }

  changeColor() {
    this.red = !this.red;
    this.green = !this.green;
  }
}

```

نلاحظ عرفنا متغيرين منطقيين واحد باسم red والثاني باسم green، وبنفس الوقت عرفنا دالة تقوم بتغيير قيم هذه المتغيرات مع كل ضغطة.

الآن لنذهب إلى ملف template ولننشئ زر ونضيف له الكلاس btn بشكل ثابت أما الكلاسات redClass، greenClass، فنريد أن نضيفها ديناميكياً إذا تحقق شرط معين، لذلك سوف نستخدم Class Binding، كالتالي:

ملف app.component.html

```

{{ 'Message: ' + name }}

<br><br> {{1234 + name}}
<br><br> {{5 + 10}}

<br><br>
<input type="text" [attr.value]="val" />

<br><br>
<input type="text" [attr.value]=" 'val: ' + val " />

<br><br>
<input type="text" [attr.value]="5 + 10" />

<br><br>
<input type="text" [attr.value]="inputValue()" />

```

```

<br><br>
<button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

```

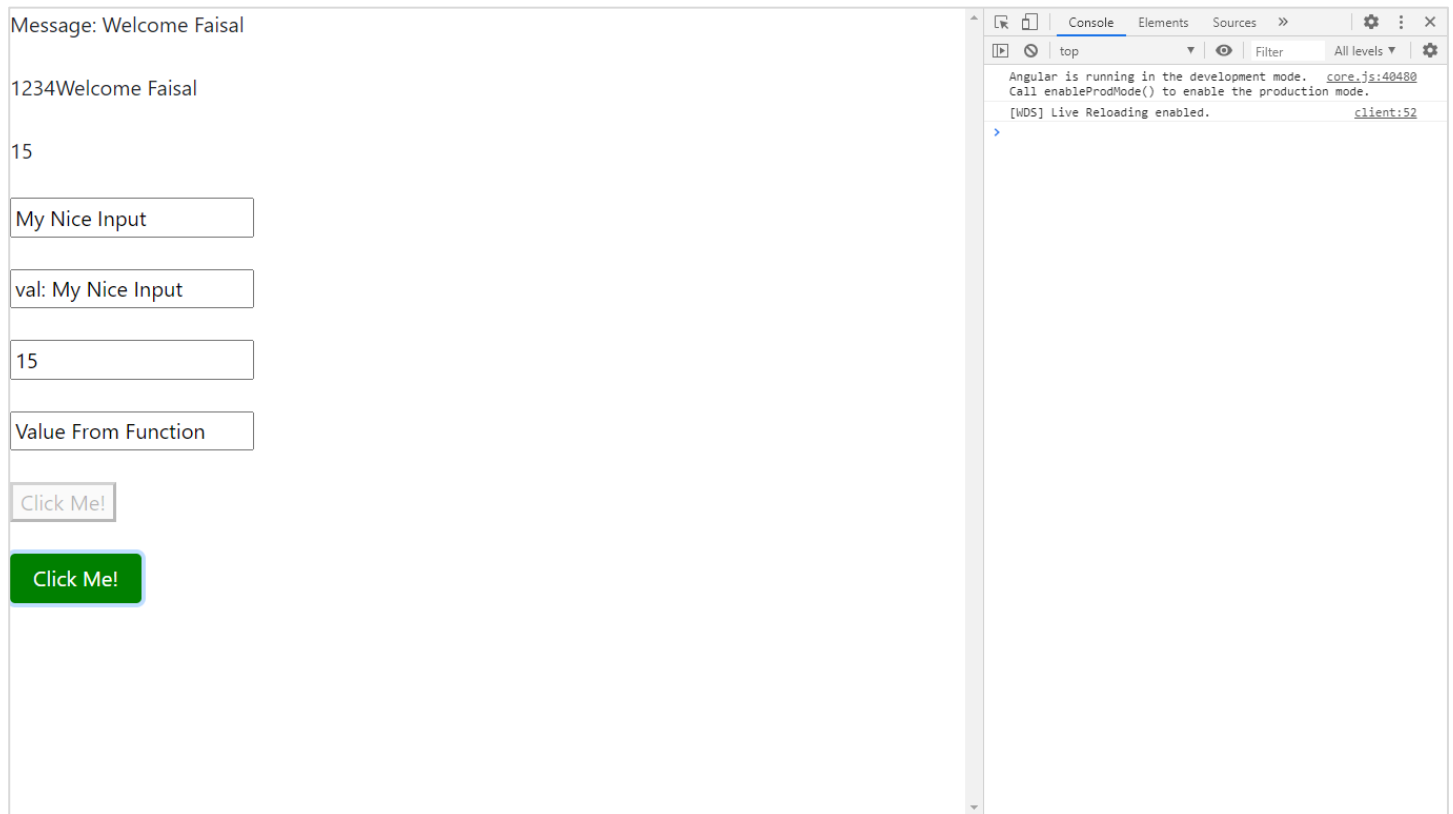
لا تهتم إلى (click) فهذه تسمى event binding وسوف نشرحها لاحقاً (تنفيذ الدالة عند الضغط على هذا الزر)، وليكن تركيزك على class binding، وكيف أننا قمنا بعمل binding لكل من redClass والكلاس greenClass بحيث يضيف هذين الكلاسين في حال كان قيمة المتغيرين red تساوي true وgreen تساوي true (ملاحظة مهمة لقد كتبنا اسم المتغير فقط اختصاراً red===false وgreen===true، وفي حال اردنا ان يكونا false نكتب !red او !green! اختصاراً). (green===false و).

أما النتيجة في المتصفح فتكون كالتالي:

The screenshot shows a web application interface with the following elements:

- A message: "Welcome Faisal"
- A counter: "1234"
- An input field with the value "15" and a label "My Nice Input".
- A button labeled "Click Me!".
- The browser's developer console is open, showing messages from Angular and WDS.

وعند الضغط على الزر:



كما يجب الإشارة أنه يمكننا أن نضيف أي شرط يعيد قيمة منطقية true او false كأن نضع شرط "red"="red" هنا أسندنا قطعة نصية او ممكن ان تكون هذه القطعة النصية محفوظة في متغير ما او من ادخالات المستخدم، بحيث إذا ساوى هذا المتغير هذه القطعة النصية فأن الجافا سكربت سوف يعيد القيمة true، كما يمكن ايضاً ممكن ان يكون دالة Function تُعيد قيمة منطقية.

## 5.2.2. Style Data Binding:

وهي مشابهة لـ Class Binding ولعل الفرق الاهم أننا نعمل Bind لخاصية من خصائص Style وفق شرط معين وليس كلاس يحتوي على مجموعة خصائص Style.

وممكن معرفة جميع هذه الخصائص عن طريق هذا الرابط [https://www.w3schools.com/jsref/dom\\_obj\\_style.asp](https://www.w3schools.com/jsref/dom_obj_style.asp)

ولنعطي مثال لتغيير لون خلفية زر معين، كالتالي:

```

app.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  val = 'My Nice Input';

```

```

num = 5;

red = true;
green = false;

bgColor = false;

constructor() {}

ngOnInit(): void {}

inputValue() {
  return 'Value From Function';
}

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

changeBackgroundndColor() {
  this.bgColor = !this.bgColor;
}
}

```

نلاحظ اننا قمنا بتعريف متغير bgColor واعطيناه القيمة false، وبنفس الوقت عرفنا دالة تقوم بتغيير قيمة هذا المتغير.

ملف app.component.html

```

{{'Message: ' + name}}
<br><br> {{1234 + name}}
<br><br> {{5 + 10}}
<br><br>
<input type="text" [attr.value]="val" />
<br><br>
<input type="text" [attr.value]=" 'val: ' + val " />
<br><br>
<input type="text" [attr.value]="5 + 10" />
<br><br>
<input type="text" [attr.value]="inputValue()" />
<br><br>
<button [disabled]="num === 5">Click Me!</button>
<br><br>
<button class="btn" [class.redClass]="red" [class.greenClass]="green" (click)="changeColor()">
  Click Me!
</button>
<button class="button" [style.backgroundColor]="bgColor ? 'blue' : 'black'" (click)="changeBackgroundndColor()">
  click Me!
</button>

```



نلاحظ استخدمنا الكلمة style للوصول إلى الخاصية backgroundColor، ووضعنا شرط بحيث إذا كانت قيمة المتغير bgColor تساوي true اسند القيمة blue للخاصية backgroundColor وإذا كانت false اسند القيمة black، ونلاحظ اننا جعلنا القيمة blue وblack بين علامتي تنصيص مفردة وذلك معناها اننا نريد ان يتعامل معها على انها قطعة نصية string وليست متغير اسمه blue او black، ونستطيع استبدالها بمتغير يحمل قيمة ديناميكية وفق شروط معينة وفي هذه الحالة نكتب اسم هذا المتغير بدون علامتي التنصيص المفردة.

Message: Welcome Faisal

1234Welcome Faisal

15

My Nice Input

val: My Nice Input

15

Value From Function

Click Me!

Click Me! click Me!

Console Elements Sources

top Filter All levels

Angular is running in the development mode. core.js:40480  
Call enableProdMode() to enable the production mode.  
[WDS] Live Reloading enabled. client:52

Message: Welcome Faisal

1234Welcome Faisal

15

My Nice Input

val: My Nice Input

15

Value From Function

Click Me!

Click Me! click Me!

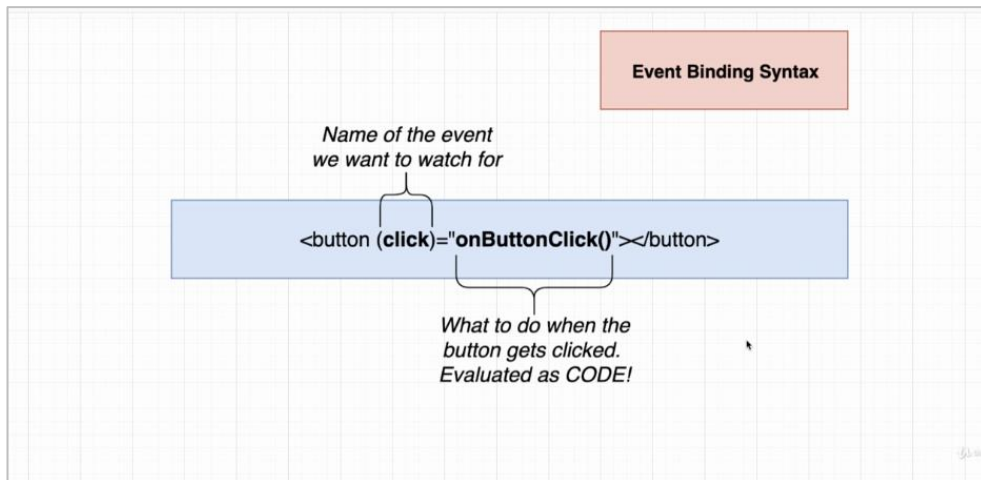
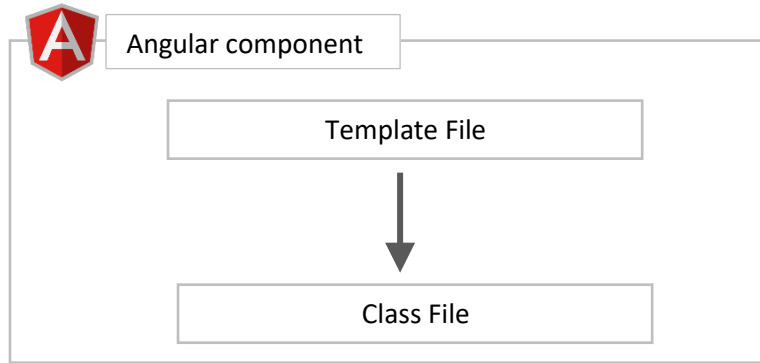
Console Elements Sources

top Filter All levels

Angular is running in the development mode. core.js:40480  
Call enableProdMode() to enable the production mode.  
[WDS] Live Reloading enabled. client:52

## 6.2.2 Event Data Binding

وهذه أيضاً تعتبر من One Way Data Binding بحيث يتم الاتصال في اتجاه واحد ولكن هنا يتم الاتصال من ملف template إلى ملف class بعكس الأنواع الأخرى، ويمكن تمثيلها بالأشكال التالية:



كما هو واضح من الأشكال السابقة أن Event Binding عن طريقها نقوم بعمل Handle للأحداث Events التي يقوم بها المستخدم عن طريق ملف Template ونمررها إلى ملف class على شكل دالة Function، كالضغط على زر معين أو مرور مؤشر الفأرة على عنصر معين...الخ.

وهذه الأحداث هي مشابهة للأحداث الموجودة بالأساس في JavaScript ولكن نحذف on ونضعها بين اقواس ( )، فمثلاً الحدث onclick يصبح (click) والحدث onmouseover يصبح (mouseover) وهكذا بقية الأحداث.

وما قيل سابقاً يُقال هنا، حيث يتم كتابة Logic برمجي بين علامتي التنصيص بشكل مباشر أو يمكن كتابة دالة وتنفذ هذه الدالة عند وقوع هذا الحدث وهذا هو الأكثر استخداماً.

كما نستطيع تمرير Template Reference لهذا الدالة كباراميتير للوصول إلى خصائص هذا التاغ، كالتالي:

```
app.component.html ملف
{{ 'Message: ' + name }}
<br><br> {{1234 + name}}
<br><br> {{5 + 10}}
<br><br>
```



```

<input type="text" [attr.value]="val" />

<br><br>
<input type="text" [attr.value]=" 'val: ' + val " />

<br><br>
<input type="text" [attr.value]="5 + 10" />

<br><br>
<input type="text" [attr.value]="inputValue()" />

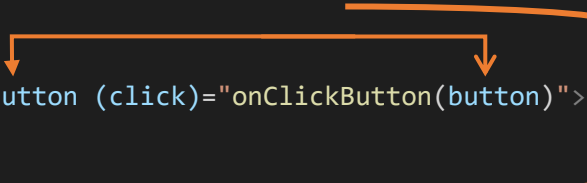
<br><br>
<button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgrondColor()"
>
click Me!
</button>

<br><br>
<button class="button" #button (click)="onClickButton(button)">
  Event Binding Example
</button>

```



ملف app.component.ts

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

```

```

red = true;
green = false;

bgColor = false;

constructor() {}

ngOnInit(): void {}

inputValue() {
  return 'Value From Function';
}

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

changeBackgrondColor() {
  this.bgColor = !this.bgColor;
}

onClickButton(e1) {
  console.log(e1);
}
}

```

نلاحظ اننا انشأنا الدالة واستقبلنا البارامتر واسميته e1 وعملنا console.log (لمعرفة كيفية التعامل مع console وبقية أدوات المطور في المتصفح الرجاء مراجعة الكتاب الأول (Angular Environment Setup)، والنتيجة كالتالي:

نلاحظ عند الضغط على هذا الزر ظهر لنا في console الزر نفسه، وذلك بسبب اننا مررنا Template Reference والذي يشير إلى العنصر الموجود فيه، كما أن angular وفرت لنا كائن Object يعطي معلومات كثيرة عن هذا الحدث وكيفية التعامل معه على العنصر الذي وقع عليه هذا الحدث، واسم هذا الكائن \$event ويمرر كبارامتر لدالة المنشأة بدلاً من Template

Reference، مع العلم ان `$event.target` تساوي `button` (template reference) كالتالي: ولو عملنا `console` لهذا الكائن عن الضغط على هذا الزر فسوف تظهر لنا البيانات التالية:

ملف `app.component.html`

```
{{ 'Message: ' + name }}

<br><br> {{1234 + name}}

<br><br> {{5 + 10}}

<br><br>
<input type="text" [attr.value]="val" />

<br><br>
<input type="text" [attr.value]=" 'val: ' + val " />

<br><br>
<input type="text" [attr.value]="5 + 10" />


<br><br>
<input type="text" [attr.value]="inputValue()" />

<br><br>
<button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<br><br>
<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgroundndColor()"
>
click Me!
</button>

<br><br>
<button class="button" (click)="onClickButton($event.target)">
  Event Binding Example
</button>
```



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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  bgColor = false;

  constructor() {}


  ngOnInit(): void {}

  inputValue() {
    return 'Value From Function';
  }

  changeColor() {
    this.red = !this.red;
    this.green = !this.green;
  }

  changeBackgrondColor() {
    this.bgColor = !this.bgColor;
  }

  onClickButton(e1) {
    console.log(e1);
  }
}
```



Message: Welcome Faisal

1234Welcome Faisal

15

My Nice Input

val: My Nice Input

15

Value From Function

Click Me!

Click Me! click Me!

Event Binding Example

Console

Angular is running in the development mode. [core.js:40480](#)  
Call enableProdMode() to enable the production mode.

[WDS] Live Reloading enabled. [client:52](#)

[app.component.ts:39](#)

<button \_ngcontent-uso-c18 class="button">  
Event Binding Example  
</button>

نلاحظ النتيجة هي نفسها.

كما يجب الإشارة انه في حال كان لدينا عنصرين متداخلين وكل عنصر له Event Binding فإنه سوف يتم تنفيذ كلا الحدين:

```
<div (click)="method1( )">
  <div (click)="method2( )"></div>
</div>
```

```
method1( ) {
  console.log("Event 1");
}
method2( ) {
  console.log("Event 2");
}
```

سوف نلاحظ في حال قام المستخدم بعمل حدث click الضغط على احدى هذين العنصرين div، فإنه سوف يتم تنفيذ كلا الدالتين، وفي حال الرغبة بتنفيذ كل حدث على حدى في حال الضغط عليه نستخدم دالة من جافا سكريبت أسمها \$event.stopPropagation

```
method1( ) {
  console.log("Event 1");
}
method2( ) {
  $event.stopPropagation()
  console.log("Event 2");
}
```

## :Tow Way Data Binding .7.2.2

ومن اسمها هي تعمل في كلا الاتجاهين من ملف class إلى ملف template ومن ملف template إلى ملف class، بمعنى آخر اننا نستطيع تغيير القيمة والتحكم بها في كلا الملفين class و template، ولتوضيح أكثر لنعطي مثال، كالتالي:

ملف app.component.ts

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

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```
export class AppComponent implements OnInit {
```

```
  name = 'Welcome Faisal';
```

```
  val = 'My Nice Input';
```

```
  num = 5;
```

```
  red = true;
```

```
  green = false;
```

```
  bgColor = false;
```

```
  textValue = '';
```



```
  constructor() {}
```

```
  ngOnInit(): void {}
```

```
  inputValue() {
    return 'Value From Function';
  }
```

```
  changeColor() {
    this.red = !this.red;
    this.green = !this.green;
  }
```

```
  changeBackgrondColor() {
    this.bgColor = !this.bgColor;
  }
```

```
  onClickButton(e1) {
    console.log(e1);
  }
```

```
}
```

نلاحظ عرفنا متغير نصي باسم textValue وأسندنا إليه قيمة فارغة.

الآن سوف نقوم بربط هذا المتغير مع عنصر input عن طريق ميزة قدمتها لنا Angular اسمها ngModel بحيث تتيح لنا القيام بالربط بالاتجاهين، ومن ثم نعمل bind لهذا المتغير عن طريق interpolation في عنصر آخر، ولكي نستطيع الاستفادة من هذه الميزة لابد من استدعاء (إضافة) FormsModule في ملف app.module.ts ، (ولمعرفة كيفية التعامل مع ملف app.module.ts بشكل عام وطرق إضافة Modules الأخرى له الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup اما في حال أردت الدخول بتفاصيل Modules الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular (Routing and Modules)، كالتالي:

#### ملف app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

بعدما اضعنا هذا Module لنقوم الآن بالاستفادة من ميزة [(ngModel)] وتسمى مجازاً banana in the box ، كالتالي:

#### ملف app.component.html

```
{{ 'Message: ' + name }}

<br><br> {{1234 + name}}

<br><br> {{5 + 10}}

<br><br><input type="text" [attr.value]="val" />

<br><br><input type="text" [attr.value]=" 'val: ' + val " />

<br><br><input type="text" [attr.value]="5 + 10" />

<br><br><input type="text" [attr.value]="inputValue()" />

<br><br><button [disabled]="num === 5">Click Me!</button>

<br><br>
```

```

<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgrondColor()"
>
click Me!
</button>

<br><br>
<button class="button" (click)="onClickButton($event.target)">
  Event Binding Example
</button>

<br><br><input type="text" [(ngModel)]="textValue" />
<h3>{{textValue}}</h3>

```



أما النتيجة فتكون كالتالي:

Message: Welcome Faisal

1234Welcome Faisal

15

val: My Nice Input

Value From Function

Click Me!

Click Me! click Me!

Event Binding Example

Welcome

Console

Angular is running in the development mode. Call enableProdMode() to enable the production mode.

[WDS] Live Reloading enabled.

نلاحظ اننا عندما قمنا بكتابة الكلمة Welcome فأن السيناريو الذي حدث كالتالي:



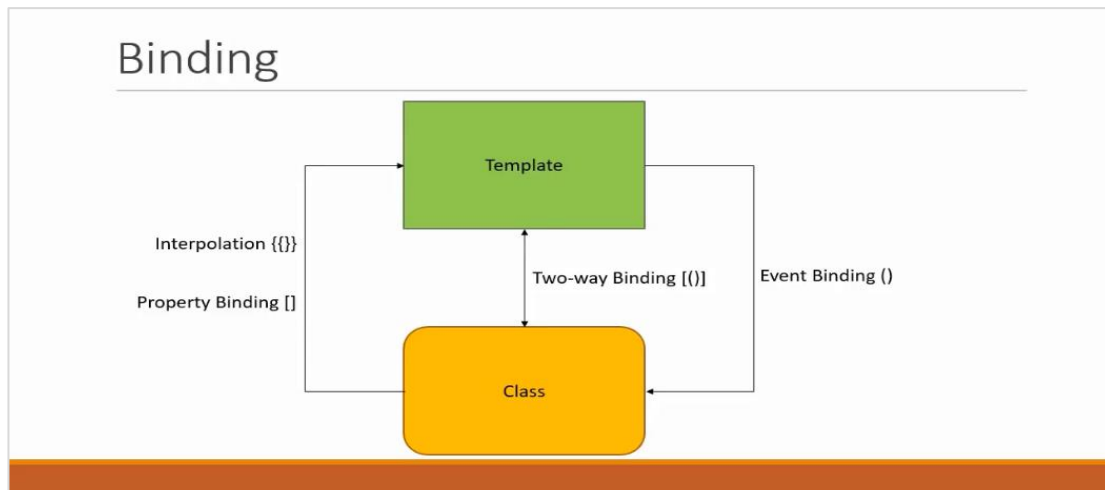
١. قام Angular بتحديث المتغير `textValue` وأضاف له القيمة الموجودة في العنصر وهي كلمة `Welcome`.
٢. وبما اننا قمنا بعمل `bind` لهذه الخاصية عن طريق `Interpolation` فإن أي تحديث على قيمة هذا المتغير سوف تظهر بشكل مباشر في العنصر `h3`.

وفي حقيقة ان الامر `[(ngModel)]` هو مزج بين `Property Binding` و `Event Binding`، بحيث يكون كالتالي:

جزء من ملف `app.component.html`

```
<br><br>
<input
  type="text"
  [(ngModel)]="textValue"
  (ngModelChange)="textValue = $event"
/>
<h3>{{textValue}}</h3>
```

ويمكن تمثيل ما قيل سابقاً بالشكل التالي:



### 1.7.2.2 تجزئة ngModel إلى Property Binding and Event Binding:

أشرت سابقاً ان `ngModel` هي في الحقيقة مزج بين `Property Binding` و `Event Binding`، وفي بعض الأحيان نحتاج إلى تجزئة هذه الصيغة المختصرة `[(ngModel)]` ومن الاحتياجات أو الأسباب التي تستدعينا إلى تجزئة هذه الصيغة المختصرة هي في حالة اردنا أن نضيف أكواد برمجية اضافية غير ربط المتغير في `class` في `template` والعكس، ففي هذه الحالة لابد ان نقسم الصيغة المختصرة إلى `Property Binding` و `Event Binding`، ولتوضيح لنعطي المثال التالي لنفرض أننا نريد ان نراقب القيمة التي يُدخلها المستخدم في `input` وفي حال كان الاسم مثلاً `faisal` فليقم بإظهار رسالة ترحيب، ففي هذه الحالة لا ينفع ان نستخدم الطريقة المختصرة وانما يجب ان نستخدم `Event Binding` ونمرر لها دالة وهذه الدالة تأخذ البارامتر `$event`، لأن الرسالة التي تظهر هي الكود البرمجي الإضافي الذي نريد اضافته إلى هذا النوع من الربط، كالتالي:

ملف `app.component.html`

```
{{ 'Message: ' + name }}
```

```

<br><br> {{1234 + name}}

<br><br> {{5 + 10}}

<br><br><input type="text" [attr.value]="val" />

<br><br><input type="text" [attr.value]=" 'val: ' + val " />

<br><br><input type="text" [attr.value]="5 + 10" />

<br><br><input type="text" [attr.value]="inputValue()" />

<br><br><button [disabled]="num === 5">Click Me!</button>

<br><br><button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgroundndColor()"
>
  click Me!
</button>

<br><br>
<button
  class="button"
  (click)="onClickButton($event.target)"
>
  Event Binding Example
</button>

<br><br>
<input
  type="text"
  [ngModel]="textValue"
  (ngModelChange)="showMsg($event)"
/>
<h3>{{textValue}}</h3>

```



كما تلاحظ عزيزي المتعلم قمنا بعمل Property Binding لي الخاصية textValue وبنفس الوقت استخدمنا Event Binding عن طريق الحدث ngModelChange وهو حدث خاص بي Angular فقط لتنفيذ دالة معينة، ومحتوى هذه الدالة نكتب الكود الإضافي الذي نريده تنفيذه عند تغير القيمة في مربع الإدخال، وفي ملف class نكتب محتوى الدالة، كالتالي:

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

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

```
export class AppComponent implements OnInit {  
  name = 'Welcome Faisal';
```

```
  val = 'My Nice Input';
```

```
  num = 5;
```

```
  red = true;
```

```
  green = false;
```

```
  bgColor = false;
```

```
  textValue = '';
```

```
  constructor() {}
```

```
  ngOnInit(): void {}
```

```
  inputValue() {  
    return 'Value From Function';  
  }
```

```
  changeColor() {  
    this.red = !this.red;  
    this.green = !this.green;  
  }
```

```
  changeBackgrondColor() {  
    this.bgColor = !this.bgColor;  
  }
```

```
  onClickButton(e1) {  
    console.log(e1);  
  }
```

```
  showMsg(event) {  
    this.textValue = event;  
    if (this.textValue === 'faisal') {  
      alert('Welcome ' + this.textValue);  
    }  
  }
```

```
}
```

هذه الطريقة، وهنالك طريقة أخرى وهي باستخدام getter وsetter، وهو ما سوف نتكلم عنه في النقطة القادمة.

## 2.7.2.2. استخدام Getter/Setter مع Two Way Data Binding:

من أسس OOP لتعريف متغيرات (خصائص) في أي كلاس هنالك طريقتين:

١. الطريقة الأولى عن طريق (Access modifiers (public – private – protected) وهو ما قمنا به عندما عرفنا الخصائص textValue او name او...الخ، مع العلم انه في Typescript في حال كتابة اسم الخاصية فقط فإنها تُعتبر public، بمعنى " " name: string = " " هي نفسها " " public name: string = " ".
٢. الطريقة الثانية: عن طريق تعريف الخاصية private ومن ثم عن طريق getter/setter وهي عبارة عن دوال نستطيع عن طريقها وضع قيمة للمتغير وارجاع قيمة هذا المتغير، حيث ان دالة setter لا تُرجع أي قيمة وانما تسند القيمة للمتغير private اما getter فهي تُرجع قيمة مساوية للقيمة التي تم تعريف المتغير private بها، واخيراً نقوم بعمل bind لدالة getter/setter لي [(ngModel)]، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
```

```

name = 'Welcome Faisal';

val = 'My Nice Input';

num = 5;

red = true;
green = false;

bgColor = false;

private _textValue = '';
constructor() {}

ngOnInit(): void {}

inputValue() {
  return 'Value From Function';
}

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

changeBackgrondColor() {
  this.bgColor = !this.bgColor;
}

onClickButton(e1) {
  console.log(e1);
}

get textValue(): string {
  return this._textValue;
}

set textValue(value: string) {
  this._textValue = value;
  if (value === 'faisal') {
    alert('Welcome ' + value);
  }
}

```

وفي ملف template نمرر دالة getter/setter والتي اسميناها textValue إلى ngModel، كالتالي:

جزء من ملف app.component.html

```

<input type="text" [(ngModel)]="textValue" />
<h3>{{textValue}}</h3>

```

والنتيجة هي نفسها، كالتالي:

Message: Welcome Faisal

1234Welcome Faisal

15

My Nice Input

val: My Nice Input

15

Value From Function

Click Me!

Click Me!click Me!

Event Binding Example

faisa

faisa

يعرض موقع localhost:4200

Welcome faisal

حسناً

Console

Angular is running in the development mode. Call enableProdMode() to enable the production mode.

[Violation] 'setTimeout' handler took 123ms

[WDS] Live Reloading enabled.

[Violation] 'input' handler took 5093ms

[Violation] 'input' handler took 13664ms

## 8.2.2. ViewChild/ViewChildren

من الميزات المهمة التي يقدمها Angular هي تقديمها لطرق سهلة ومبسطة للوصول إلى العناصر (التاغات) في ملف template عن طريق ملف class والتحكم بجميع خواصها برمجياً، وذلك عن طريق Decorator ذو الاسم @ViewChild للوصول إلى عنصر واحد أو @ViewChildren للوصول إلى قائمة من العناصر دفعة واحدة على شكل Array.

### 1.8.2.2. ViewChild

كما قلنا سابقاً هذا النوع نستطيع عن طريقه الوصول إلى عنصر واحد وللتوضيح لنعطي مثال، وليكن انه في بداية تشغيل التطبيق نريده ان يعمل focus على Input معين، وللقيام بذلك لنعطي العنصر (input) المستهدف Template Reference كما تعلمنا سابقاً وليكن اسمه على سبيل المثال #focusInput، كالتالي:

```
app.component.html ملف
{{ 'Message: ' + name }}
<br><br> {{1234 + name}}
<br><br> {{5 + 10}}
<br><br><input #focusInput type="text" [attr.value]="val" />
<br><br><input type="text" [attr.value]=" 'val: ' + val " />
<br><br><input type="text" [attr.value]="5 + 10" />
<br><br><input type="text" [attr.value]="inputValue()" />
```

```

<br><br><button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgroundndColor()"
>
  click Me!
</button>

<br><br>
<button
  class="button"
  (click)="onClickButton($event.target)"
>
  Event Binding Example
</button>

<br><br>
<input
  type="text"
  [(ngModel)]="textValue"
/>
<h3>{{textValue}}</h3>

```

نلاحظ أننا أضفنا لأول input اسم #focusInput وسوف نستخدم هذا الاسم في ملف class للوصول إلى هذا العنصر والتحكم به برمجياً، وللقيام بذلك نستخدم ViewChild ونمرر له الاسم الذي اخترناه focusInput، ومن ثم نقوم بتعريف متغير (خاصية) تكون كأنها صورة لهذا العنصر أو مؤشر يشير إلى هذا العنصر ويكون من النوع ElementRef، وهذا المتغير هو الذي نتعامل معه برمجياً للوصول إلى هذا العنصر والتحكم بخصائصه، كالتالي:

ملف app.component.ts

```

import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild('focusInput') fInput: ElementRef;

```

```

name = 'Welcome Faisal';

val = 'My Nice Input';

num = 5;

red = true;
green = false;

bgColor = false;

private _textValue = '';

constructor() {}

ngOnInit(): void {
}

inputValue() {
    return 'Value From Function';
}

changeColor() {
    this.red = !this.red;
    this.green = !this.green;
}

changeBackgrondColor() {
    this.bgColor = !this.bgColor;
}

onClickButton(el) {
    console.log(el);
}

get textValue(): string {
    return this._textValue;
}

set textValue(value: string) {
    this._textValue = value;
    if (value === 'faisal') {
        alert('Welcome ' + value);
    }
}
}

```

نلاحظ اننا عرفنا متغير باسم flnput ويجب التنويه انه لك عزيزي المتعلم حرية اختيار الاسم الذي تُريده مع العلم انه يُسمح باختيار اسم متغير مشابهه لاسم Template Reference المُعطى للعنصر في ملف template.



وأفضل مكان لكتابة الكود الخاص بالتحكم بهذا العنصر هو في الدالة `ngAfterViewInit` (من دوال lifecycle hooks التي شرحناها في الفصل السابق) وكما قلنا سابقاً ان هذه الدالة يتم استدعائها بعد اكتمال View الخاص بي template وآخر جزء يتم بناءه في ملف template هو العناصر التي عرفناها على انها `view child` او `view children` لذلك أي كود تريد ان تقوم بكتابته يقوم بعمل شيء ما على أحد العناصر بعد بنائه بشكل كامل، يفضل كتابته في هذه الدالة، لذلك لنقوم بكتابة الكود الخاص بالوصول إلى الخاصية `focus` لهذا العنصر، كالتالي:

ملف `app.component.ts`

```
import {
  Component,
  OnInit,
  AfterViewInit,
  ViewChild,
  ElementRef
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild('focusInput') fInput: ElementRef;

  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  bgColor = false;

  private _textValue = '';

  constructor() {}

  ngOnInit(): void {}

  ngAfterViewInit(): void {
    this.fInput.nativeElement.focus();
  }

  inputValue() {
    return 'Value From Function';
  }

  changeColor() {
```



```

    this.red = !this.red;
    this.green = !this.green;
  }

  changeBackgrondColor() {
    this.bgColor = !this.bgColor;
  }

  onClickButton(e1) {
    console.log(e1);
  }

  get textValue(): string {
    return this._textValue;
  }

  set textValue(value: string) {
    this._textValue = value;
    if (value === 'faisal') {
      alert('Welcome ' + value);
    }
  }
}

```

اما النتيجة فتكون كالتالي:

The screenshot shows a web application with the following elements:

- A message: "Message: Welcome Faisal"
- A text input field containing "1234Welcome Faisal"
- A text input field containing "15"
- A text input field containing "My Nice Input" (highlighted by an orange arrow)
- A text input field containing "val: My Nice Input"
- A text input field containing "15"
- A text input field containing "Value From Function"
- A button labeled "Click Me!"
- A red button labeled "Click Me!" and a black button labeled "click Me!"
- A button labeled "Event Binding Example"
- A text input field

The browser's console shows the following messages:

- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.

نلاحظ أن أول عنصر (input) تم وضع focus عليه عند بداية تشغيل التطبيق وبذلك نكون تعلمنا كيف نصل إلى أي عنصر في ملف template عن طريق ملف class ونتحكم بجميع خصائصه برمجياً. ولكن هنالك شيء غير محبب في هذا الكود، وهو الوصول المباشر لهذا العنصر في DOM وتغيير خصائصه، وهذه الطريقة غير محببة في إطار عمل Angular لعدة أسباب، منها:

١. أكواد Angular تعمل في بيئات غير بيئة DOM فمثلاً نستطيع عن طريق Angular عمل تطبيقات موبايل او تطبيقات مكتبية.

٢. دواعي امنية فالوصول بشكل مباشر إلى DOM يكون عرضة لهجمات XSS.

لذلك وفرت لنا Angular مكتبة أخرى service نستطيع عن طريقها الوصول إلى DOM والتحكم بخصائصه، وهذه service اسمها Renderer2، وللاستفادة منها لابد ان نعمل لها inject في constructor الخاص بملف class لهاذا component، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  Renderer2,
  AfterViewInit,
  ViewChild,
  ElementRef
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild('focusInput') fInput: ElementRef;

  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  bgColor = false;

  private _textValue = '';

  constructor(private renderer: Renderer2) {}

  ngOnInit(): void {
  }

  ngAfterViewInit(): void {
    this.renderer.selectRootElement(this.fInput.nativeElement).focus();
  }

  inputValue() {
    return 'Value From Function';
  }
}
```

```

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

changeBackground() {
  this.bgColor = !this.bgColor;
}

onClickButton(e1) {
  console.log(e1);
}

get textValue(): string {
  return this._textValue;
}

set textValue(value: string) {
  this._textValue = value;
  if (value === 'faisal') {
    alert('Welcome ' + value);
  }
}
}

```

نلاحظ أننا وصلنا إلى العنصر في DOM عن طريق الطريقة الأولى `this.flInput.nativeElement` ولكن التحكم بخصائص هذا العنصر استخدمنا `Renderer2`، وسوف نشرح `Renderer2` بشيء من التفصيل في فصل مواضيع متقدمة في `Components` بإذن الله.

## 2.8.2.2:ViewChildren

وهي ميزة أخرى تقدمها لنا `Angular` لإحصاء عدد من العناصر وتجميعها فيما يسمى `QueryList` وهذا النوع ما هو إلا كائن يتم فيه تخزين قائمة من العناصر، والمميز فيه ان `Angular` يراقب أي تعديلات تحدث على هذه العناصر وعند حدوث أي تعديل فإنه يقوم بشكل تلقائي بتحديث قائمة عناصر `QueryList`. كما ان هذا النوع له مجموعة من `api`، يمكن حصرها في الجدول التالي:

Properties	first	الحصول على اول عنصر في القائمة
	last	الحصول على آخر عنصر في القائمة
	length	الحصول على عدد العناصر في القائمة
Methods	map()	وهي مشابهة لدوال الموجودة في الجافا سكريبت
	filter()	
	find()	
	reduce()	

	forEach()  some()  toArray()  changes()	لتحويل القائمة إلى مصفوفة جافا سكربت، وبذلك لا نستطيع الاستفادة من الميزات الموجودة في QueryList وهو من النوع observable حيث نعمل له subscribe ومراقبة أي تعديلات وفي حال حدوث أي تعديل على القائمة ننفذ كود معين
--	---	---

اما من ناحية طريقة كتابتها او صيغتها العامة فهي تقريبا مشابهة لـ ViewChild بما فيها كتابة الكود في الدالة ngAfterViewInit، ولكن تمتاز عنها، بالآتي:

- نستطيع التعامل مع مجموعة من العناصر وليس عنصر واحد مع إمكانية استخدامها لتعامل مع عنصر واحد
  - تُعيد النوع QueryList وهذا النوع هو generic، لذلك نستطيع تمرير له ElementRef في حال أردنا التعامل مع عناصر بصورة template references سواء نمرر single template reference او multi template references كما نستطيع تمرير اسم component في حال أردنا الوصول إلى component ابن او حتى نمرر له ngModel في حال كنا نتعامل مع Forms ولدينا مجموعة من inputs مضاف لها هذا Directive فنستطيع تجميعها عن طريقه.
  - نستطيع عمل subscribe ومراقبة التعديلات لتنفيذ كود معين.
- ونستطيع تمثيل الحالات التي عن طريقها عمل Grouping لمجموعة من العناصر من خلال الشكل التالي:

ViewChildren	
Angular Directive	@ViewChildren(NgModel) inputs: QueryList<NgModel>;
Custom Directive / Child Component	@ViewChildren(StarComponent) stars: QueryList<StarComponent>;
Template Reference Variable	@ViewChildren('divElementVar') divElementRefs: QueryList<ElementRef>;
Template Reference Variables	@ViewChildren('filterElement, nameElement') divElementRefs: QueryList<ElementRef>;

ولتوضيح لنعطي مثال، لنفرض انه لدينا مجموعة من عناصر لا ونريد ان نتحكم فيها عن طريق ملف class، وللقيام بذلك سوف نقوم ببناء مجموعة من لا وكل لا نعطيها نفس template reference، كالتالي:

```

{{ 'Message: ' + name }}

<br><br> {{1234 + name}}

<br><br> {{5 + 10}}

<br><br><input #focusInput type="text" [attr.value]="val" />

<br><br><input #focusInput1 type="text" [attr.value]=" 'val: ' + val " />

<br><br><input type="text" [attr.value]="5 + 10" />

<br><br><input type="text" [attr.value]="inputValue()" />

<br><br><button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgrondColor()"
>
  click Me!
</button>

<br><br>
<button
  class="button"
  (click)="onClickButton($event.target)"
>
  Event Binding Example
</button>

<br><br>
<input
  type="text"
  [(ngModel)]="textValue"
/>
<h3>{{textValue}}</h3>

<ul>
  <li #testLi>Item 1</li>
  <li #testLi>Item 2</li>

```

```
<li #testLi>Item 3</li>
<li #testLi>Item 4</li>
<li #testLi>Item 5</li>
</ul>
```

وسوف نقوم بتجميع جميع هذه العناصر في متغير من النوع QueryList ونمرر له ElementRef، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  Renderer2,
  AfterViewInit,
  ViewChild,
  ElementRef,
  ViewChildren,
  QueryList
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild('focusInput') fInput: ElementRef;

  @ViewChildren('testLi') testLi: QueryList<ElementRef>;

  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  bgColor = false;

  private _textValue = '';

  constructor(private renderer: Renderer2) {}

  ngOnInit(): void {}

  ngAfterViewInit(): void {
    this.renderer.selectRootElement(this.fInput.nativeElement).focus();
  }

  inputValue() {
```



```

    return 'Value From Function';
  }

  changeColor() {
    this.red = !this.red;
    this.green = !this.green;
  }

  changeBackgrondColor() {
    this.bgColor = !this.bgColor;
  }

  onClickButton(e1) {
    console.log(e1);
  }

  get textValue(): string {
    return this._textValue;
  }

  set textValue(value: string) {
    this._textValue = value;
    if (value === 'faisal') {
      alert('Welcome ' + value);
    }
  }
}

```

نلاحظ اننا قمنا بتعريف متغير واسميناه بنفس اسم template reference وهو testList، وبذلك أصبح لدينا متغير من النوع QueryList لذلك سوف نستخدم الدالة forEach لعمل Loop على جميع هذه العناصر ومن ثم تغيير لون النص لكل عنصر، كالتالي:

جزء من ملف app.component.ts

```

ngAfterViewInit(): void {
  this.renderer.selectRootElement(this.fInput.nativeElement).focus();
  this.testLi.forEach(element => {
    this.renderer.setStyle(element.nativeElement, 'color', 'red');
  });
}

```

والنتيجة في المتصفح، تكون كالتالي:



كما نستطيع مراقبة أي تغيرات تحدث في القائمة وتنفيذ كود معين، وسوف نقوم بجعل القائمة ديناميكية بناءً على عدد عناصر مصفوفة نقوم بإنشائها ونسند لهذه المصفوفة قيمة مبدئية، ومن ثم في template نعمل Loop على هذه القائمة عن طريق \*ngFor لكي نقوم ببنائها ديناميكياً في DOM (لمعلومات أكثر عن Directive ذو الاسم \*ngFor الرجاء مراجعة الكتاب الثالث من هذه السلسلة Angular Pipes and Directives)، وكنوع من التدريب على بعض ما قيل سابقاً لنقم بإنشاء مربع نص input وزر، ونضيف لمربع النص template reference ونعمل لزر Event Binding لتنفيذ Logic برمجي مهمته إضافة القيمة الموجودة في مربع النص إلى المصفوفة وبشكل تلقائي لكي يتم تحديث القائمة وإظهارها للمستخدم، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  Renderer2,
  AfterViewInit,
  ViewChild,
  ElementRef,
  ViewChildren,
  QueryList
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild('focusInput') fInput: ElementRef;
  @ViewChildren('testLi') testLi: QueryList<ElementRef>;
```

```
Items: string[] = ['Angular'];  
  
name = 'Welcome Faisal';  
  
val = 'My Nice Input';  
  
num = 5;  
  
red = true;  
green = false;  
  
bgColor = false;  
  
private _textValue = '';  
  
constructor(private renderer: Renderer2) {}  
  
ngOnInit(): void {  
}  
  
ngAfterViewInit(): void {  
    this.renderer.selectRootElement(this.fInput.nativeElement).focus();  
}  
  
inputValue() {  
    return 'Value From Function';  
}  
  
changeColor() {  
    this.red = !this.red;  
    this.green = !this.green;  
}  
  
changeBackgrondColor() {  
    this.bgColor = !this.bgColor;  
}  
  
onClickButton(e1) {  
    console.log(e1);  
}  
  
get textValue(): string {  
    return this._textValue;  
}  
  
set textValue(value: string) {  
    this._textValue = value;  
    if (value === 'faisal') {  
        alert('Welcome ' + value);  
    }  
}  
}
```

ملف app.component.html

```
{{ 'Message: ' + name }}
<br><br> {{1234 + name}}
<br><br> {{5 + 10}}

<br><br><input #focusInput type="text" [attr.value]="val" />
<br><br><input #focusInput1 type="text" [attr.value]=" 'val: ' + val " />
<br><br><input type="text" [attr.value]="5 + 10" />
<br><br><input type="text" [attr.value]="inputValue()" />

<br><br><button [disabled]="num === 5">Click Me!</button>

<br><br>
<button
  class="btn"
  [class.redClass]="red"
  [class.greenClass]="green"
  (click)="changeColor()"
>
  Click Me!
</button>

<button
  class="button"
  [style.backgroundColor]="bgColor ? 'blue' : 'black'"
  (click)="changeBackgrondColor()"
>
  click Me!
</button>

<br><br>
<button
  class="button"
  (click)="onClickButton($event.target)"
>
  Event Binding Example
</button>

<br><br>
<input type="text" [(ngModel)]="textValue" />
<h3>{{textValue}}</h3>

<input type="text" #InputVal/>
<button (click)="Items.push(InputVal.value)">Add Items</button>

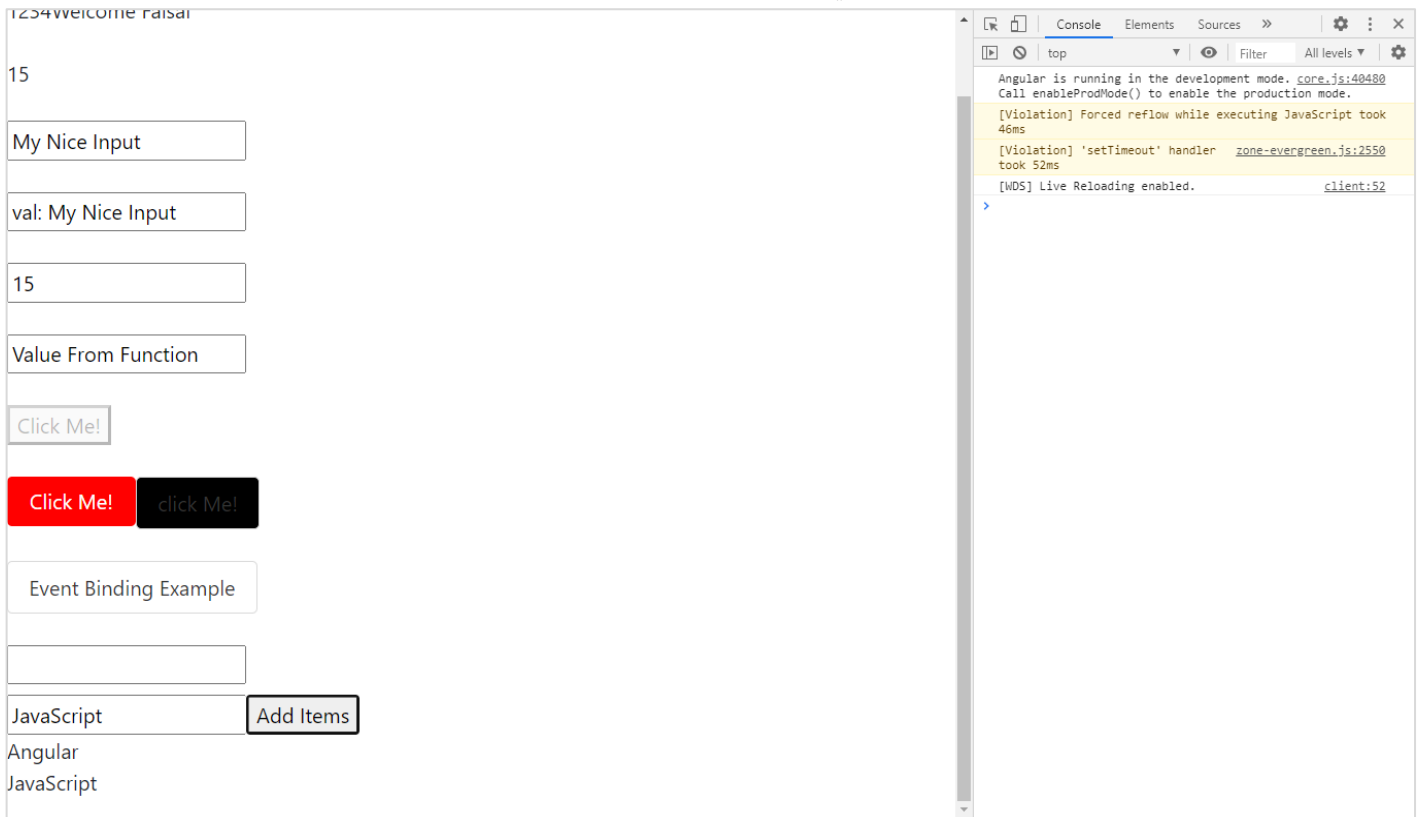
<ul>
  <li #testLi *ngFor="let item of Items">{{item}}</li>
</ul>
```



نلاحظ وجود العنصر input و template reference الخاص به وهو #InputVal، وايضاً نلاحظ الزر button حيث عملنا له Event Binding اما Logic فهو سطر برمجي قمنا فيه باستدعاء المصفوفة Items وعملنا push للقيمة الموجودة بداخل input داخل المصفوفة، وأخيراً عملنا Loop على جميع عناصر المصفوفة وقمنا ببناء عناصر li ديناميكياً بناءً على عناصر المصفوفة مع الحفاظ على نفس template reference لجميع عناصر li بحيث يكون #testLi لكي نعمل لها جميعاً ViewChildren لاحقاً، وبطبيعة الحال في البداية المصفوفة لا يوجد بها إلى عنصر واحد وهو Angular لذلك سوف يقوم ببناء li واحد، كالتالي:

الآن لنقم بكتابة قيمة أخرى داخل input ونضغط على الزر Add Items، كالتالي:

وعند الضغط على الزر سوف تكون النتيجة، كالتالي:



The screenshot shows a web application with the following elements:

- A text input field containing "1234welcome raisai".
- A text input field containing "15".
- A text input field containing "My Nice Input".
- A text input field containing "val: My Nice Input".
- A text input field containing "15".
- A text input field containing "Value From Function".
- A button labeled "Click Me!".
- A red button labeled "Click Me!" and a black button labeled "click Me!".
- A button labeled "Event Binding Example".
- A text input field.
- A text input field containing "JavaScript" and a button labeled "Add Items".
- A text input field containing "Angular".
- A text input field containing "JavaScript".

The console on the right shows the following messages:

- Angular is running in the development mode. core.js:40480
- Call enableProdMode() to enable the production mode.
- [Violation] Forced reflow while executing JavaScript took 46ms
- [Violation] 'setTimeout' handler zone-evergreen.js:2550 took 52ms
- [WDS] Live Reloading enabled. client:52

نلاحظ أضاف لنا القيمة في القائمة، الآن نريد عمل مراقبة لتغييرات التي تحدث للعنصر `li` لأننا عملنا `ViewChildren` عن طريق `Template Reference` ذو الاسم `#testLi` والمشارك لجميع عناصر `li`، وهنالك طريقتين لمراقبة أي تغييرات الأولى عن طريق الدالة `ngAfterViewChecked` (من دوال `lifecycle hooks`) بحيث نكتب الكود الذي نريده هناك وفي حال وقوع أي تعديل ننفذ كود معين، اما الطريقة الثانية فتكون باستخدام الدالة `changes` الموجودة ضمناً في المتغيرات التي من النوع `QueryList`، وقد اشرنا لهذا سابقاً، وسوف نقوم بتطبيق كلا الطريقتين، ولنعطي في البداية مثال على دالة `ngAfterViewChecked`، كالتالي:

```
app.component.ts ملف
import {
  Component,
  OnInit,
  Renderer2,
  AfterViewInit,
  ViewChild,
  ElementRef,
  ViewChildren,
  QueryList,
  AfterViewChecked
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit, AfterViewInit, AfterViewChecked {
```

```

@ViewChild('focusInput') fInput: ElementRef;
@ViewChildren('testLi') testLi: QueryList<ElementRef>;

Items: string[] = ['Angular'];

name = 'Welcome Faisal';

val = 'My Nice Input';

num = 5;

red = true;
green = false;

bgColor = false;

private _textValue = '';

constructor(private renderer: Renderer2) {}

ngOnInit(): void {
}

ngAfterViewInit(): void {
  this.renderer.selectRootElement(this.fInput.nativeElement).focus();
}

```



```

ngAfterViewChecked(): void {
  console.log(this.testLi);
  console.log('Test Li Length: ' + this.testLi.length);
  console.log('Test Li First Item : ' +
    this.renderer.selectRootElement(
      this.testLi.first.nativeElement, true).textContent
    );
  console.log('Test Li Last Item : ' +
    this.renderer.selectRootElement(
      this.testLi.last.nativeElement, true).textContent
    );
}

```

```

inputValue() {
  return 'Value From Function';
}

```

```

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

```

```

changeBackgroundndColor() {
  this.bgColor = !this.bgColor;
}

```

```

onClickButton(e1) {
  console.log(e1);
}

get textValue(): string {
  return this._textValue;
}

set textValue(value: string) {
  this._textValue = value;
  if (value === 'faisal') {
    alert('Welcome ' + value);
  }
}
}

```

نلاحظ اننا قمنا بطباعة مجموعة من القيم في console:

طباعة الكائن نفسه الذي مررناه عن طريق ViewChildren وهو من النوع QueryList

```
console.log(this.testLi);
```

طباعة عدد العناصر (li) في الكائن testLi

```
console.log('Test Li Length: ' + this.testLi.length);
```

طباعة اول عنصر من عناصر الكائن

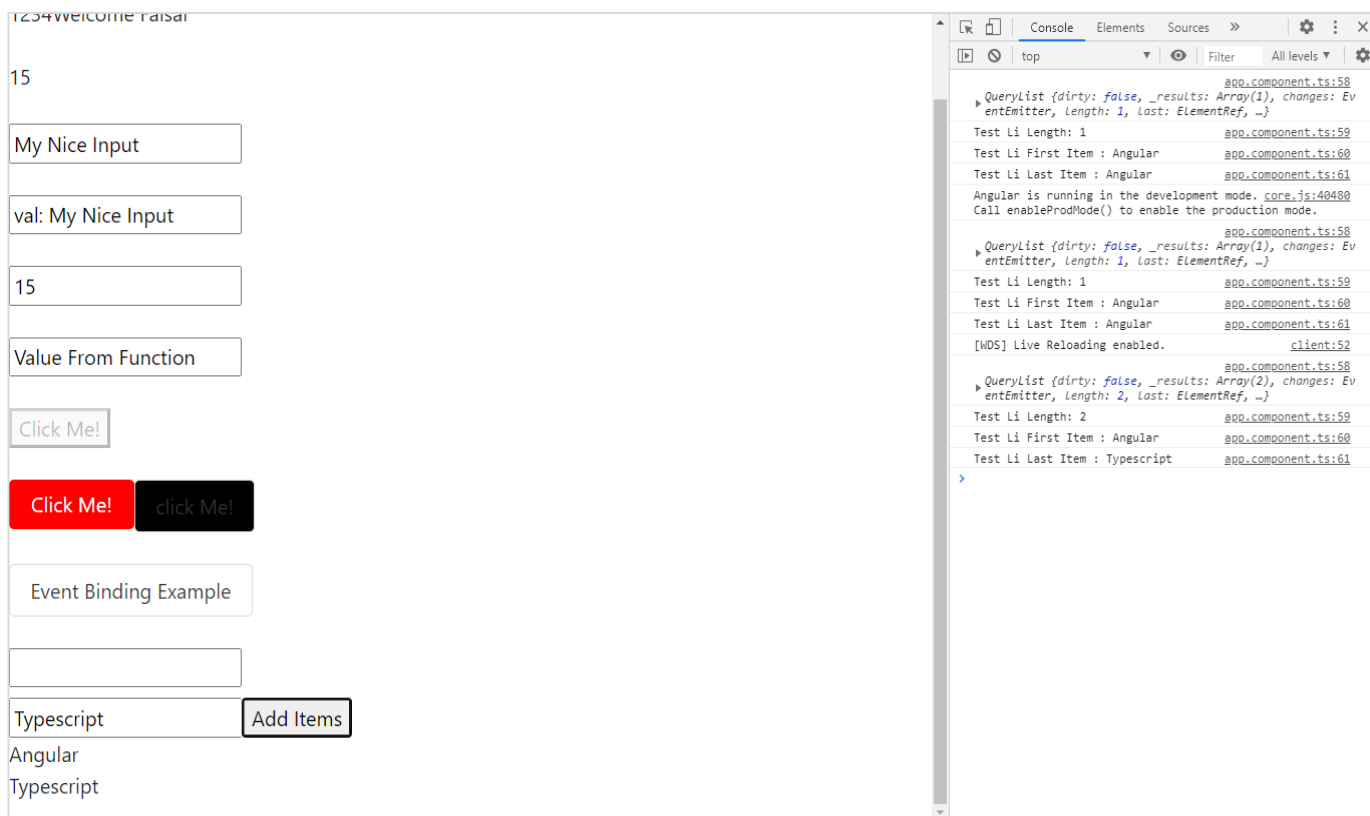
```
console.log('Test Li First Item: ' + this.renderer.selectRootElement(this.testLi.first.nativeElement, true).textContent);
```

طباعة آخر عنصر من عناصر الكائن

```
console.log('Test Li Last Item: ' + this.renderer.selectRootElement(this.testLi.last.nativeElement, true).textContent);
```

والنتيجة كالتالي:

نلاحظ في console تمت طباعة القيم مرتين وذلك بسبب الآلية التي قللناها سابقاً حيث يقوم باستدعاء الدالة ngDoDcheck ومن ثم ngAfterViewChecked لذلك نفذ الكود مرتين، والآن لنقوم بكتابة أي قيمة في مربع الادخال ونضغط على الزر Add Item، كالتالي:



نلاحظ انه قام بتنفيذ الكود وذلك من خلال مراقبة التغيرات وعند أي تغيير يقوم بتنفيذ الكود.

ولكن هذه الطريقة غير محببة، لأنه أي تعديل على template ولو لم يكن يختص الجزء الذي نريد ان نراقب التغيرات التي حدثت عليه (إضافة عناصر إلى testLi) فإنه سوف يقوم بتنفيذ واستدعاء الدالة ngAfterViewChecked وهذا ليس المتوقع، فالذي نريده ان يقوم بمراقبة التغيرات فقط على الكائن testLi وفي حال وقوع أي تعديل نقوم بتنفيذ الأكواد، لذلك سوف نقوم باستخدام الطريقة الثانية وهي عن طريق الدالة changes ونعمل لها subscribe، داخل الدالة ngAfterViewInit كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  Renderer2,
  AfterViewInit,
  ViewChild,
  ElementRef,
  ViewChildren,
  QueryList,
  AfterViewChecked
} from '@angular/core';

@Component({
  selector: 'app-root',
```



```

templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
}))

export class AppComponent implements OnInit, AfterViewInit, AfterViewChecked {
  @ViewChild('focusInput') fInput: ElementRef;
  @QueryList<ElementRef> testLi: QueryList<ElementRef>;

  Items: string[] = ['Angular'];

  name = 'Welcome Faisal';

  val = 'My Nice Input';

  num = 5;

  red = true;
  green = false;

  bgColor = false;

  private _textValue = '';

  constructor(private renderer: Renderer2) {}

  ngOnInit(): void {
  }

  ngAfterViewInit(): void {
    this.renderer.selectRootElement(this.fInput.nativeElement).focus();

    this.testLi.changes.subscribe(QList => {
      console.log(QList);
      console.log('Test Li Length: ' + QList.length);
      console.log(
        'Test Li First Item : ' + this.renderer.selectRootElement(
          QList.first.nativeElement, true).textContent
      );
      console.log(
        'Test Li Last Item : ' + this.renderer.selectRootElement(
          QList.last.nativeElement, true).textContent
      );
    });
  }

  ngAfterViewChecked(): void {
    // console.log(this.testLi);
    // console.log('Test Li Length: ' + this.testLi.length);
    // console.log('Test Li First Item : ' + this.renderer.selectRootElement(this.testLi.first.nativeElement, true).textContent);
    // console.log('Test Li Last Item : ' + this.renderer.selectRootElement(this.testLi.last.nativeElement, true).textContent);
  }
}

```



```

}

inputValue() {
  return 'Value From Function';
}

changeColor() {
  this.red = !this.red;
  this.green = !this.green;
}

changeBackgrondColor() {
  this.bgColor = !this.bgColor;
}

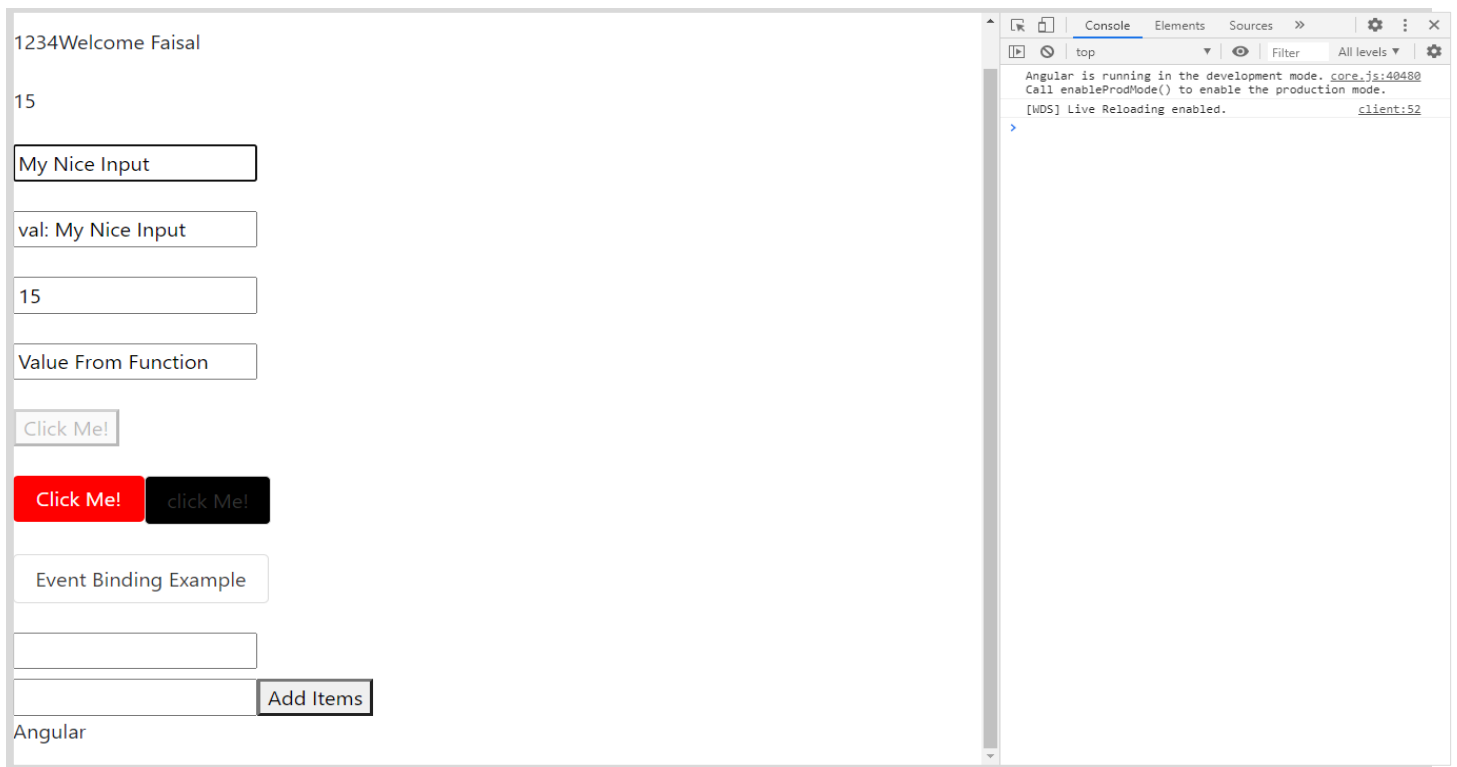
onClickButton(el) {
  console.log(el);
}

get textValue(): string {
  return this._textValue;
}

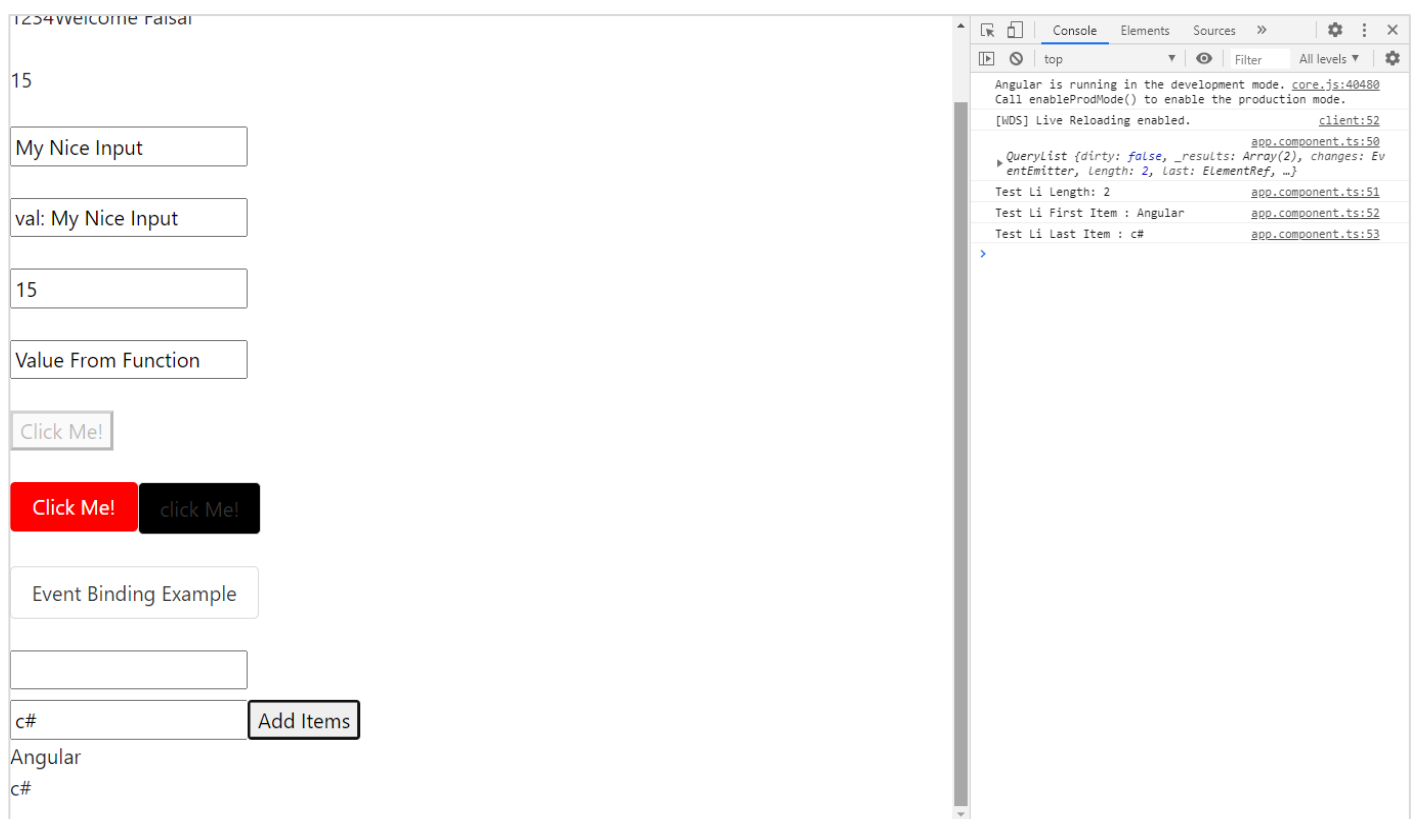
set textValue(value: string) {
  this._textValue = value;
  if (value === 'faisal') {
    alert('Welcome ' + value);
  }
}
}

```

الآن لنرى النتيجة في المتصفح:



اول شيء نلاحظه انه لم يطبع أي شيء في console وهذا هو المتوقع لأننا لم نقوم بالتعديل على الكائن testLi، الآن لنقم بإضافة عنصر، ونرى النتيجة:

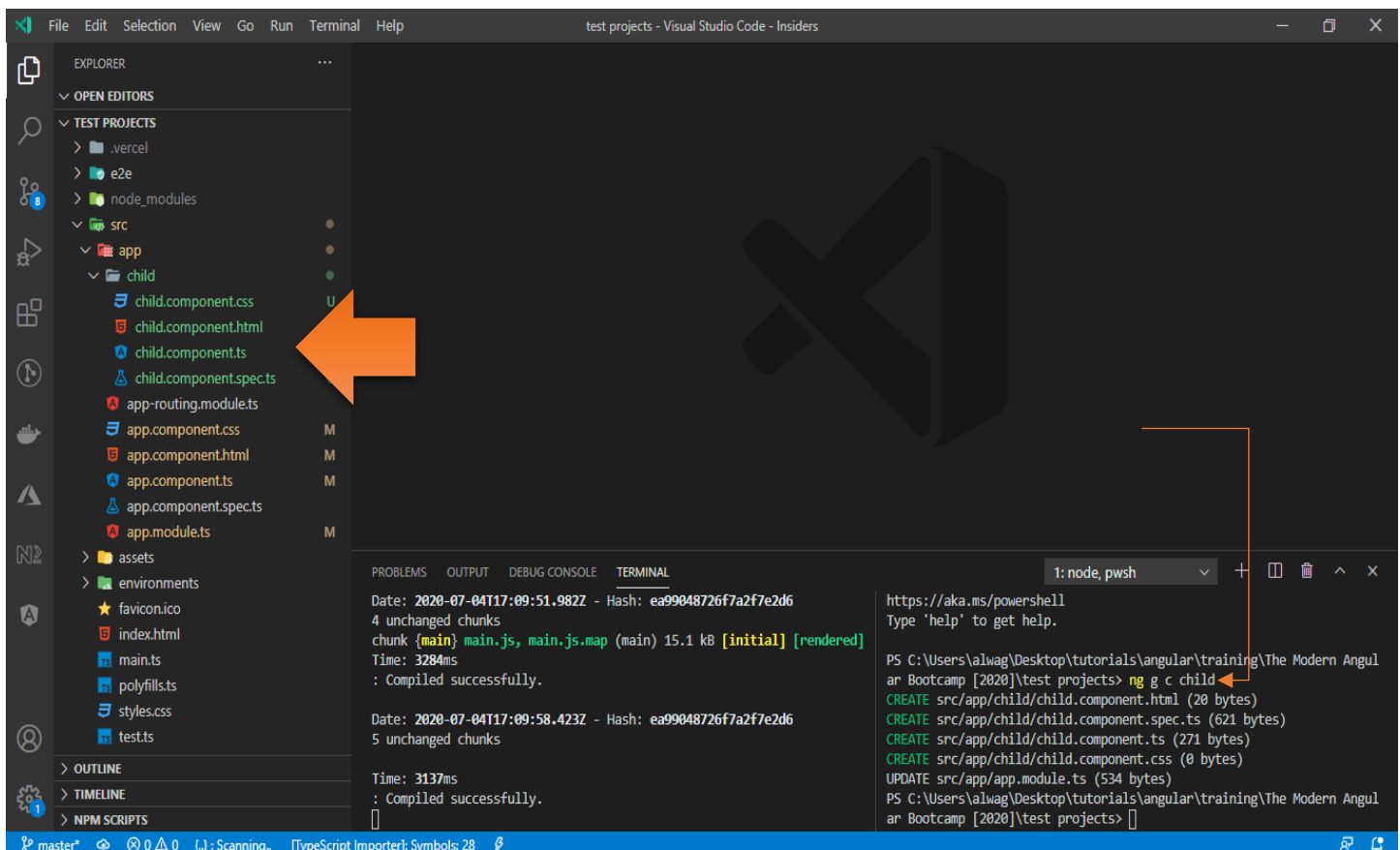


عندما قمنا بالتعديل على الكائن تلقائياً قام بتفعيل الدالة changes ونفذ الكود الموجود بداخلها وطبع القيم في console، وايضاً هذا هو المتوقع وما نريد منه ان يقوم به، وبنفس الوقت مهما قمت بالتعديل على View او باسم آخر template فإنه لن يفعل الدالة changes لأن هذه الدالة لا تنفذ إلا في حال التعديل على الكائن testLi.

وبذلك نكون قمنا بشرح اغلب المفاهيم الخاصة بالتواصل بين ملفات component نفسه وكيفية التعامل معها مع ضرب الأمثلة عليها، وفي الجزء القادم سوف نتقل إلى جزء مهم آخر وهو التواصل بين components المختلفة التي تربط بينهما علاقة أب وأبن.

### 3.2. التواصل بين Components المختلفة التي تربطهم علاقة (أب – أبن):

من أهم الأمور التي تواجهك عزيزي المتعلم اثناء بناء تطبيقات عن طريق إطار عمل Angular هو احتياجك لوجود Components داخل Components أخرى، ومعرفتك لتنظيم هذا الأمر سوف يساعدك بشكل كبير في سرعة بناء التطبيق الخاص بك بالإضافة إلى سهولة الفهم في حال مشاركتك لتطبيقك مع فريق مستقبلاً، أو حتى في مرحلة التطوير، ويفضل بناء مخطط Diagram لجميع أجزاء التطبيق ويتم فيها توضيح هذه Components والعلاقات التي تربطها، وحقيقة ليس هنا المقام لشرح هذه الجزئية، وما يهمنا هو كيفية التواصل بين Components التي تربطها علاقة أب – أبن (Parent-Child)، وقبل البدء بشرح أنواع الاتصال والتواصل بين هذا النوع من العلاقات، لنقم بإنشاء مشروع Angular جديد، وبعد إنشاء هذا المشروع لنقم بإنشاء component جديد وليكن اسمه child، عن طريق الامر (ng g c child)، كالتالي:



لورجعنا إلى الفصل السابق والمعنون بعنوان مدخل إلى Components لوجدت عزيزي المتعلم أننا ذكرنا أن في ملف class لكل component يوجد ما يسمى selector، كالتالي:

```
app.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
```

```

templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  constructor() {}
  ngOnInit(): void {}
}

```

ملف child.component.ts

```

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}

```

وهذا selector لو أضفناه إلى ملف template في component آخر على شكل تاغ HTML فإنه يصبح أبن لهذا component، لذلك سوف نضيف selector ذو الاسم app-child إلى ملف template في Root Component، وبذلك نكون انشأنا علاقة أب وأبن بين هذين Components بحيث الأب هو Root Component الذي يحوي بداخله selector الخاص بالأبن app-child، مع العلم ان أسماء selector ليست ثابتة فتستطيع تغيير الاسم الذي تريده ولكن يجب الالتزام به في جميع الاستخدامات لهذه selector، وايضاً نلاحظ أن أي component يتم اضافته فإنه بشكل تلقائي سوف يتم إضافة الكلمة app لـ Selector الخاص به، مع العلم انك تستطيع تغيير هذا الوضع الافتراضي وتغيير الاعدادات كما تشاء ولمعلومات أكثر عن كيفية التحكم بأسماء Selectors للـ components الرجاء مراجعة الكتاب الأول Angular Environment Setup، والآن لنقوم بإنشاء هذه العلاقة عملياً، كالتالي:

ملف app.component.html

```
<app-child></app-child>
```

نلاحظ اضفنا selector على شكل تاغ HTML وبذلك أصبح هذا component أبن في Root Component.

وكما قلنا سابقاً أن هنالك طرق متعددة للتواصل والاتصال بين components التي تربطها علاقة من هذا النوع، كالتالي:

### 1.3.2. الاتصال من component الأب إلى الأبن عن طريق @Input():

اتاحة لنا Angular طريقة مبسطة لتمير البيانات من الأب إلى الأبن عن طريق Decorator ذو الاسم @Input() حيث تقوم فكرته بكل بساطة، بما أن selector هو تاغ HTML في ملف template داخل component الأب فلنقوم بعمل property binding من ملف class إلى ملف template لهذا التاغ، ومن ثم نستقبل هذه الخاصية property في component الأبن عن طريق @Input فقط لا غير، والآن لنقم بتطبيق ما قيل عملياً، كالتالي:

```
ملف app.component.ts
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  name = 'Faisal';

  constructor() {}
  ngOnInit(): void {}

}
```

نلاحظ اننا قمنا بتعريف متغير باسم name وأسندنا له القيمة Faisal، ونريد ان نمرر هذا المتغير من component الاب إلى الأبن، لذلك أو خطوة نقوم بها هي عمل Property Binding لهذه الخاصية لتاغ <app-child></app-child> الذي يمثل الأبن لهذا Component، كالتالي:

```
ملف app.component.html
<h3>Parent Component</h3>
<app-child [FirstName]="name"></app-child>
```

نلاحظ عملنا Property Binding للمتغير (الخاصية) name في المتغير (الخاصية) FirstName، الآن لنقوم باستقبال الخاصية FirstName في component الأبن عن طريق Input كالتالي:

```
ملف child.component.ts
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})

export class ChildComponent implements OnInit {

  @Input('FirstName') FName: string;

}
```

```

constructor() { }

ngOnInit(): void {
}
}

```

نلاحظ استقبلنا الخاصية FirstName التي تم تمريرها من component الأب إلى الابن وخرنا هذه القيمة في المتغير (الخاصية) FName، الآن أصبحت القيمة متاحة لهذا component ونستطيع ان نعمل بها ما نريد، ولتسهيل سوف أقوم بعرض هذه القيمة في ملف template عن طريق Interpolation، كالتالي:

ملف app.component.html

```
<h5>Welcome {{FName}} from Child Component</h5>
```

ملاحظة مهمة انا قمت بتغيير أسماء المتغيرات لتوضيح مسار القيمة من بداية انشاءها إلى وصولها إلى component الابن، وتستطيع توحيد جميع هذه الأسماء باسم واحد، فبدلاً من name و FirstName، FName، تستطيع توحيدها جميعاً باسم واحد كأن يكون مثلاً name، كالتالي:

```
name='Faisal'
```

```
<app-child [name]="name"></app-child>
```

```
@Input('name') name: string – {{name}} ]
```

ملاحظة أخرى مهمة: نستطيع الاستغناء عن الاسم الموجود بين اقواس @Input() والاكتفاء باسم المتغير بشرط ان يكون اسم المتغير نفس الاسم في property binding، كالتالي:

```
[ n='Faisal'
```

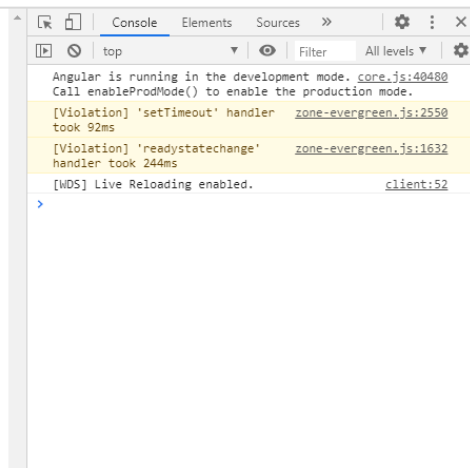
```
<app-child [name]="n"></app-child>
```

```
@Input() name: string – {{name}} ]
```

الآن لنذهب إلى المتصفح ونرى النتيجة، كالتالي:

## Parent Component

Welcome Faisal from Child Component

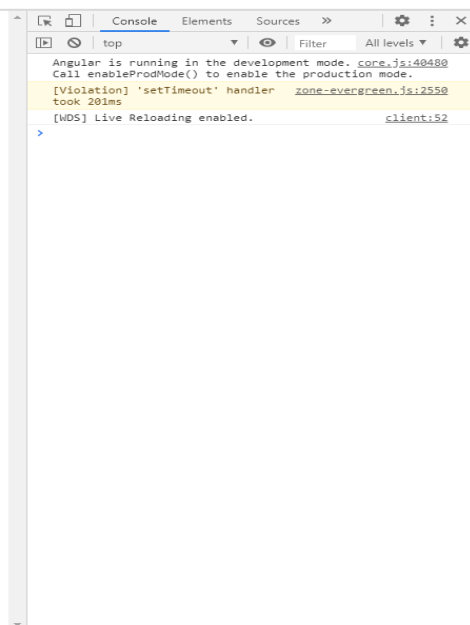


ولو قمنا بتغيير القيمة إلى قيمة أخرى فإن القيمة التي تم تمريرها إلى الأب سوف يتم تحديثها بشكل تلقائي، كالتالي:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  name = 'Fahd';
  constructor() {}
  ngOnInit(): void {}
}
```

## Parent Component

Welcome Fahd from Child Component



بعض الأحيان نحتاج إلى مراقبة البيانات الداخلة إلى component الابن بواسطة `@Input()` وفي حال تغير هذه البيانات نقوم بتنفيذ كود معين.

ونستطيع القيام بهذا الامر بطريقتين الأولى باستخدام Setter/Getter والثانية باستخدام دالة lifecycle hooks ذات الاسم `ngOnChanges`، اما إجابة السؤال الذي قد يترأود إلى ذهنك عزيزي المتعلم، وهو متى استخدم Setter/Getter ومتى استخدم



الدالة ngOnChanges؟ والإجابة ترجع إلى عدد البيانات الداخلة والتي نريد مراقبتها، فإذا كان لدينا خمس متغيرات (خصائص) قمنا بتمريرها إلى component الأب، ولا نريد مراقبة إلا متغير واحد فعندئذ يُفضل استخدام Setter/Getter، أما إذا كان هنالك أكثر من متغير نريد مراقبة التغيرات التي تحدث على القيم الخاصة بهم، ففي هذه الحالة يُفضل استخدام الدالة ngOnChanges، كالتالي:

### 1.1.3.2. مراقبة التغيرات التي تحدث على القيم باستخدام Getter/Setter:

لنقوم بتمرير متغير من النوع المنطقي boolean بحيث إذا كانت القيمة true نطبع له رسالة ترحيب وإذا كانت القيمة false نطبع له رسالة نطالبه فيها بتسجيل الدخول، لذلك لنقوم في البداية بتعريف هذا المتغير في ملف class للـ component الأب وليكن اسمه loggedIn ونعطيه قيمة مبدئية true، كالتالي:

```
app.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  name = 'Faisal';
  loggedIn = true;

  constructor() {}
  ngOnInit(): void {}
}
```

ومن ثم نعمل Property Binding لهذه الخاصية في ملف template بداخل التاغ <app-child> كالتالي:

```
app.component.html ملف
<h3>Parent Component</h3>
<app-child [name]="name" [loggedIn]="loggedIn"></app-child>
```

ونستقبل هذا المتغير عن طريق @Input() في ملف class للـ component الأب، ولكن باختلاف بسيط عن الطريقة الأولى ففي هذه المرة سوف نستقبل القيمة عن طريق Input ولكن باستخدام الدالة Setter ونمرر لهذه الدالة باراميتر بحيث يحمل هذا الباراميتر القيمة التي تم إرسالها من component الأب، وبداخل هذه الدالة نراقب التعديلات التي تتم على هذه القيمة وفي حال أي تغيير يحدث عليها ننفذ كود معين وفي حالتنا هنا هي اظهار رسالة للمستخدم، كالتالي:

```
child.component.ts ملف
import { Component, OnInit, Input } from '@angular/core';
```

```

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit {

  @Input('name') name: string;

  private _loggedIn: boolean;
  message: string;

  get loggedIn(): boolean {
    return this._loggedIn;
  }

  @Input() set loggedIn(value: boolean) {
    this._loggedIn = value;
    this.message = value? `Welcome Back ${this.name} from Child Component`: 'please Login..';
  }

  constructor() { }

  ngOnInit(): void {
  }

}

```

نلاحظ اننا قمنا بتخزين الرسالة في المتغير message، الآن لنعمل Interpolation لهذا المتغير في ملف template لكي نظهر الرسالة للمستخدم، كالتالي:

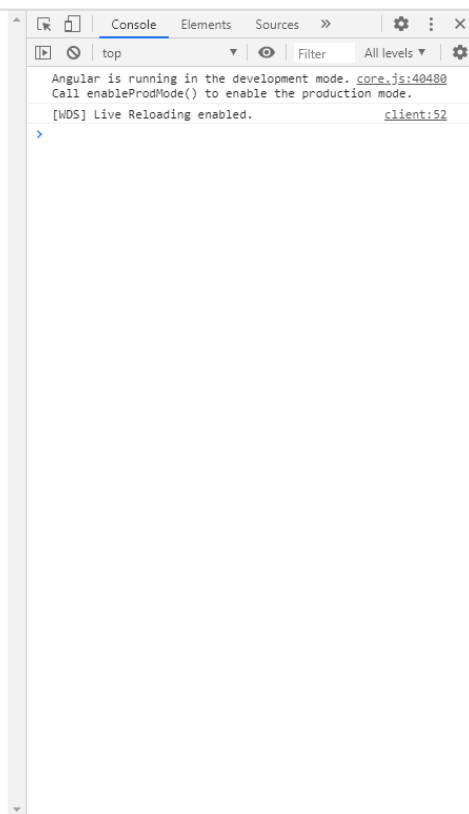
ملف child.component.html

```
<h5>{{message}}</h5>
```

اما النتيجة في المتصفح، كالتالي:

## Parent Component

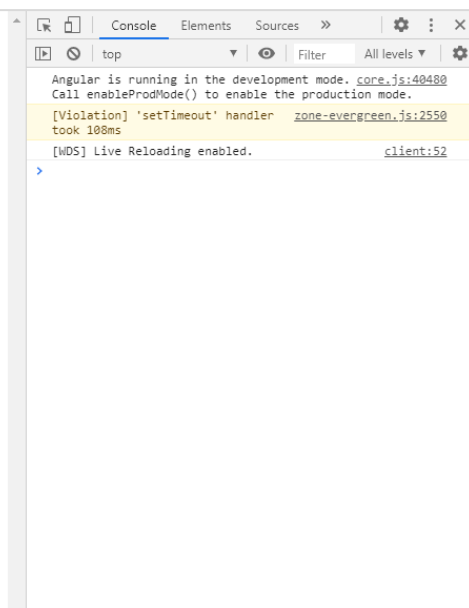
Welcome Back Faisal from Child Component



في حال كانت قيمة المتغير true، تظهر الرسالة السابقة، والآن لنقوم بتغيير قيمة المتغير loggedIn في ملف class component إلى القيمة false، ولنرى النتيجة:

## Parent Component

please Login..



هذا في حال كنا نريد مراقبة Input واحد ولكن ماذا لو أردنا مراقبة أكثر من Input، ففي هذه الحالة يُفضل استخدام الدالة ngOnChanges، وهذا ما سوف نتكلم عنه في الجزء التالي.

### 2.1.3.2. مراقبة التغيرات التي تحدث على القيم باستخدام ngOnChanges:

لقد تطرقنا بالسابق إلى هذه الدالة ذكراً فقط في الفصل السابق، حيث قلنا انها مرتبطة بي (@Input)، لأنها تقوم بمراقبة التغيرات التي تحدث للقيم الداخلة عن طريق هذه Decorators وفي حال وقوع أي تعديل فإن Angular يقوم باستدعاء هذه الدالة، وبما انه لدينا اثنين Inputs واحد name والآخر loggedIn فسوف نكتفي بهذه المدخلات ونراقب التعديلات الحاصلة

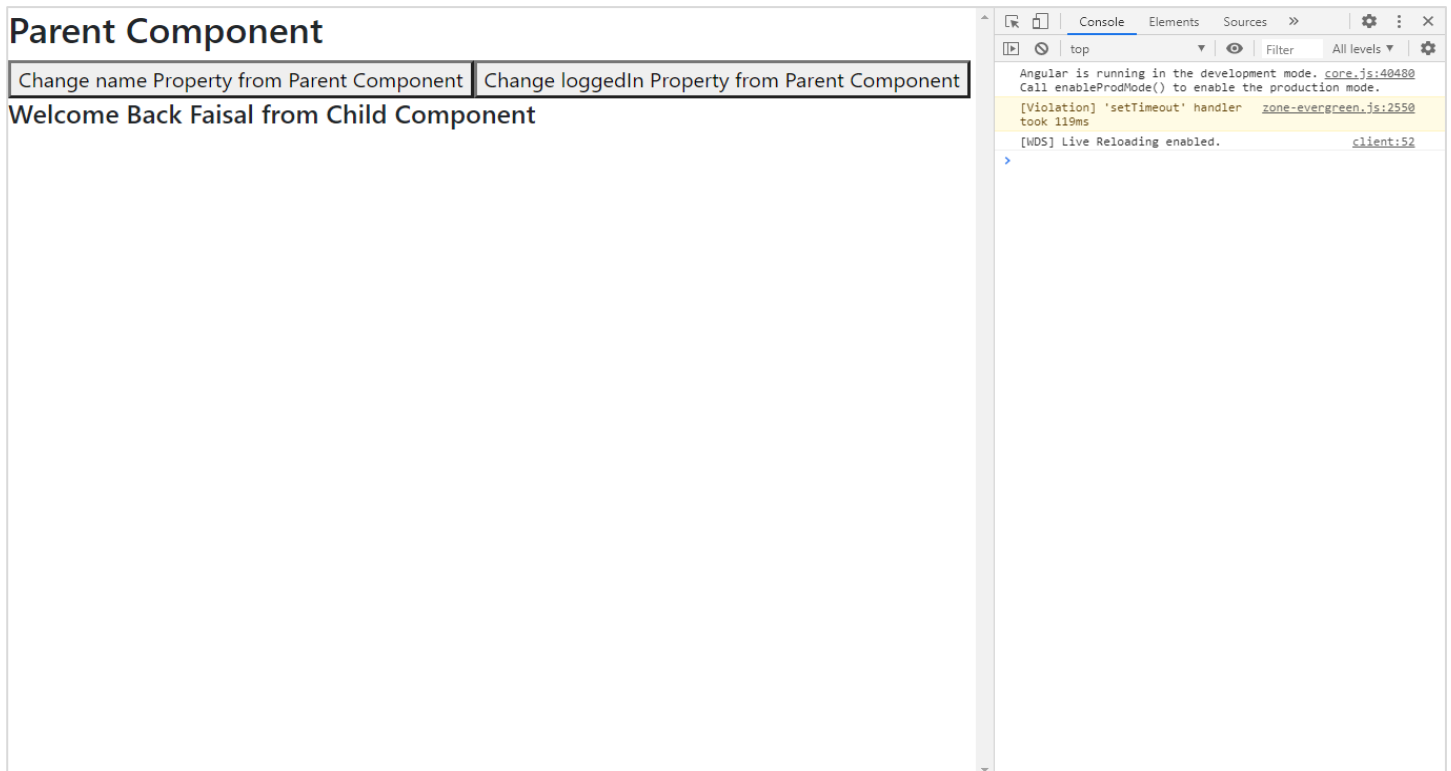
عليها، لذلك سوف نقوم بإنشاء زر في component الأب لكي نقوم بتغيير القيم من خلالهما ومن ثم مراقبة التغييرات التي تحدث على هذه القيم في component الابن، كالتالي:

ملف app.component.html

```
<h3>Parent Component</h3>
<button (click)="name='Fahd'">Change name Property from Parent Component</button>
<button (click)="loggedIn=!loggedIn">Change loggedIn Property from Parent Component</button>

<app-child [name]="name" [loggedIn]="loggedIn"></app-child>
```

والنتيجة في المتصفح، كالتالي:



ولو ضغطنا على هذه الأزرار سوف نلاحظ ان هذه القيم تتغير وتظهر التغييرات في component الابن، والآن نريد ان نراقب هذه التغييرات عن طريق الدالة ngOnChanges، لذلك في البداية لنستدعي الـ Interface الخاص بهذه الدالة ونعمل له implements في ملف class الخاص بالـ component الابن، مع العلم ان هذه الدالة هي الدالة الوحيدة من دوال lifecycle hooks التي نمرر لها باراميتر وهو من النوع SimpleChanges وهو عبارة عن كائن يحتوي الخصائص (المتغيرات) التي تم تمريرها للـ component الابن بالإضافة إلى احتواءها على بعض الخصائص الأخرى التي تصف القيم الداخلة، وسوف نقوم بطباعة هذا الباراميتر في console، لمشاهدة ما يحتويه، كالتالي:

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
```

```

})
export class ChildComponent implements OnInit, OnChanges {

  @Input('name') name: string;

  private _loggedIn: boolean;
  message: string;

  get loggedIn(): boolean {
    return this._loggedIn;
  }

  @Input() set loggedIn(value: boolean) {
    this._loggedIn = value;
    this.message = value ? `Welcome Back ${this.name} from Child Component` : 'please Login..';
  }

  constructor() { }

  ngOnInit(): void { }

  ngOnChanges(changes: SimpleChanges): void {
    console.log(changes);
  }
}

```

ولنشاهد النتيجة في المتصفح:

## Parent Component

Welcome Back Faisal from Child Component

child.component.ts:32

▼ {name: SimpleChange, loggedIn: SimpleChange}

▼ loggedIn: SimpleChange

currentValue: true

firstChange: true

previousValue: undefined

\_\_proto\_\_: Object

▼ name: SimpleChange

currentValue: "Faisal"

firstChange: true

previousValue: undefined

\_\_proto\_\_: Object

Angular is running in the development mode. core.js:40480

Call enableProdMode() to enable the production mode.

[WDS] Live Reloading enabled. client:52

نلاحظ ان من بداية تشغيل التطبيق في المتصفح قام Angular باستدعاء هذه الدالة وتنفيذ ما يحتويها وهو طباعة محتويات هذا الكائن في console.

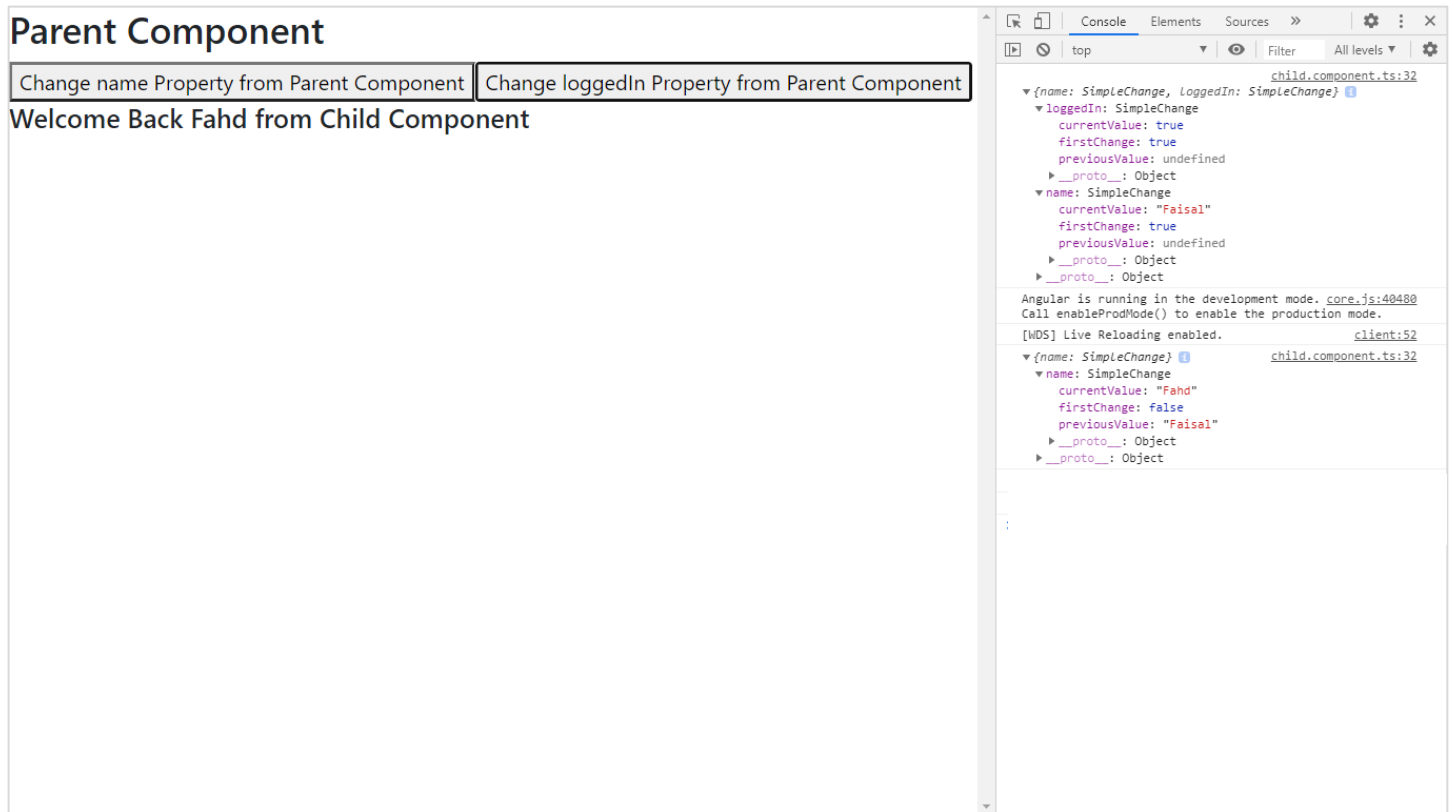
كما نلاحظ انه يحتوي على الخاصيتين name و loggedIn وكل خاصية من هذين الخاصيتين تحتوي بداخلها على ثلاث خصائص:

currentValue: وتحمل القيمة الحالية للخاصية.

**firstChange**: وتحمل قيمة منطقية true او false، بحيث تكون قيمتها true في حال اول قيمة تم تمريرها لي component الأبن، وفي حال تغيير هذه القيمة سوف تحمل القيمة false.

**previousValue**: وتحمل القيمة السابقة للمتغير وفي حال كان اول مرة تم تشغيل التطبيق وتم تمرير اول قيمة لي component الأبن فان هذه الخاصية تحمل القيمة undefined اما في حال تغيير القيمة لاحقاً فإنها تحمل القيمة الموجودة في **currentValue** والخاصية **currentValue** تحمل القيمة الجديدة، وهكذا في كل تغيير.

لنقم بالضغط على أحد الأزرار ونرى النتيجة:



The screenshot shows a web application with a 'Parent Component' containing two buttons: 'Change name Property from Parent Component' and 'Change loggedIn Property from Parent Component'. Below the buttons, it says 'Welcome Back Fahd from Child Component'. The browser's developer console is open, showing the state of the components. The first log shows the 'loggedIn' component with 'currentValue: true', 'firstChange: true', and 'previousValue: undefined'. The second log shows the 'name' component with 'currentValue: "Faisal"', 'firstChange: true', and 'previousValue: undefined'. The third log shows the 'name' component with 'currentValue: "Fahd"', 'firstChange: false', and 'previousValue: "Faisal"'. The fourth log shows the 'loggedIn' component with 'currentValue: "Fahd"', 'firstChange: false', and 'previousValue: "Faisal"'. The console also shows messages from Angular and Webpack Dev Server (WDS).

نلاحظ عند تغيير الخاصية name من Faisal إلى Fahd التقطت الدالة ngOnChanges هذه التعديلات ونفذت الكود الموجود بداخلها، وهو طباعة الكائن في console، ونلاحظ انه لم يطبع إلا المتغير (الخاصية) الذي تم تغيير قيمته، وبنفس الوقت تغيرت الخصائص **currentValue** لتصبح Fahd و **firstChange** أصبحت قيمتها false بمعنى انه تم تغيير قيمة هذا المتغير مرة أخرى وليست المرة الأولى وأخيراً القيمة **previousValue** أصبحت قيمتها Faisal وهي القيمة السابقة للخاصية **currentValue**. الآن لنقوم بكتابة الكود الذي يقوم بتنفيذ مهمة معينة عند تغيير القيم لهذه المدخلات، وسوف اكتب الكود ومن ثم أقوم بالتعليق عليه، كالتالي:

```
child.component.ts ملف
import { Component, OnInit, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
```

```

export class ChildComponent implements OnInit, OnChanges {

  @Input('name') name: string;

  private _loggedIn: boolean;
  message: string;

  get loggedIn(): boolean {
    return this._loggedIn;
  }

  @Input() set loggedIn(value: boolean) {
    this._loggedIn = value;
    this.message = value ? `Welcome Back ${this.name} from Child Component` : 'please Login..';
  };

  constructor() { }

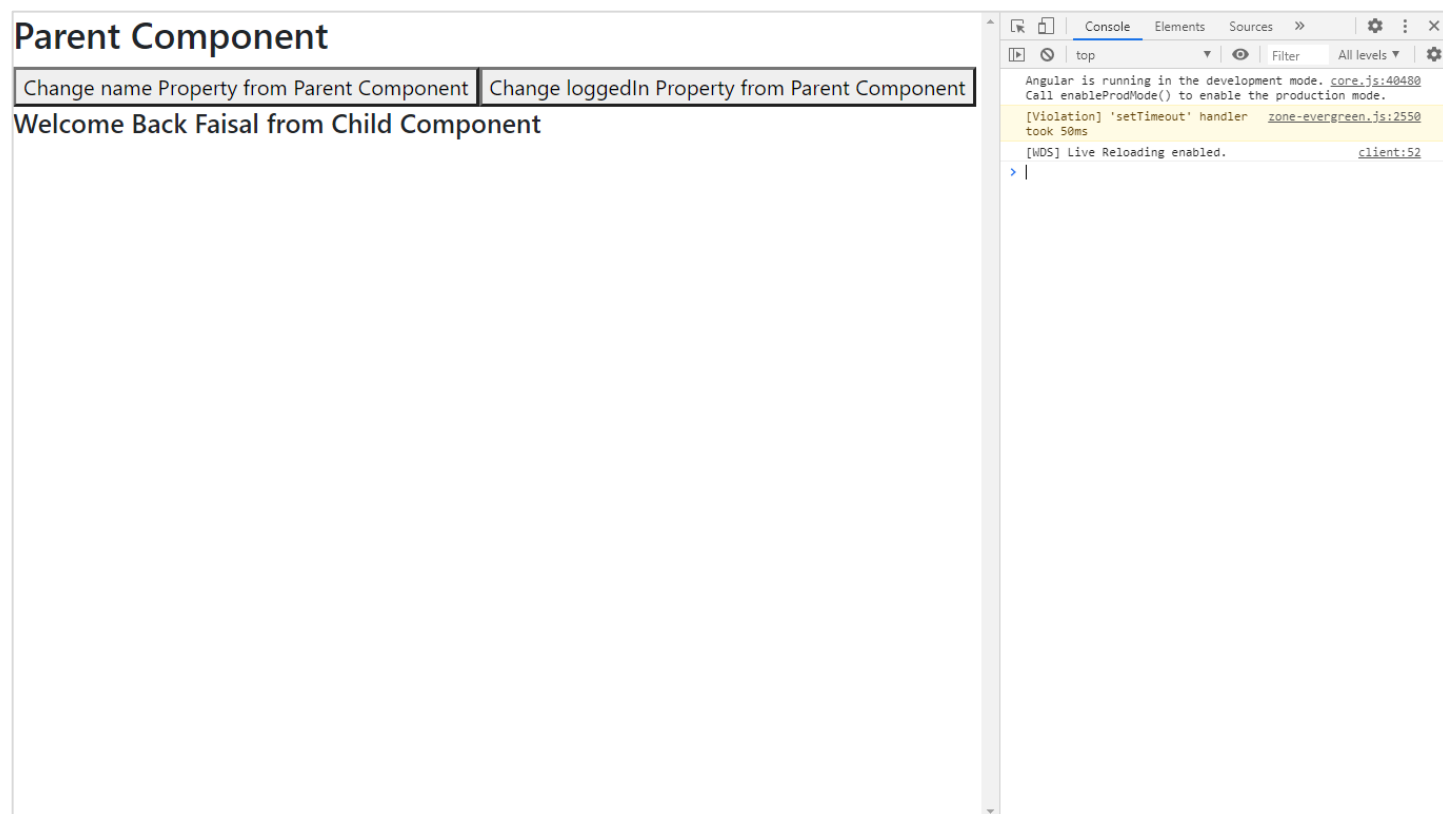
  ngOnInit(): void { }

  ngOnChanges(changes: SimpleChanges): void {
    for (const propertyName in changes) {
      if (changes.hasOwnProperty(propertyName) && !changes[propertyName].firstChange) {
        switch (propertyName) {
          case 'name': {
            console.log('name property changed');
            break;
          }
          case 'loggedIn': {
            console.log('loggedIn property changed');
            break;
          }
        }
      }
    }
  }
}

```

قمنا بعمل Loop لأنه لدينا أكثر من مدخل متمثلة في الكائن changes والذي يحتوي على المتغيرات (الخواص) الداخلة إلى component الأب عن طريق @Input() على شكل keys، لذلك استخدمنا for in بحيث ان المتغير propertyName يكون في البداية قيمته name وفي المرة الثانية يكون قيمته loggedIn، ومن ثم وضعنا شرط للتأكد ان هنالك قيمة في هذا المتغير وشرط آخر في حال كان هذا اول تغيير له ام لا (الفائدة من الشرط الأخير هو انه لا نريد أن نقوم بتنفيذ محتويات الدالة ngOnChanges عند بداية تشغيل التطبيق في البداية وانما يتم قراءة وتنفيذ محتويات هذه الدالة عند اول تغيير يحدث بعد تشغيل التطبيق، وفي حال أردت ان تجعل الوضع الافتراضي على ما هو عليه وهو قراءة محتويات هذه الدالة في البداية ومن ثم عند أي تغيير فنقوم فقط بإلغاء هذا الشرط (!changes[propertyName].firstChange)، وفي حال تحقق الشرطين نعمل switch-case لمعرفة حالات القيم في المتغير propertyName، فإذا كانت القيمة الداخلة التي حدث عليها تغيير هي name فقم

بتنفيذ كود معين (وفي حالتنا هنا نقوم بطباعة رسالة في console)، ونفس الوضع مع المتغير الثاني `loggedIn`، وبذلك كل ما أضفنا مدخلات جديدة إلى `component` ونريد مراقبتها نضيفها إلى `switch case`.



نلاحظ عند بداية التطبيق لم يطبع أي شيء في console وذلك بسبب الشرط السابق الذي تكلمت عنه قبل قليل لذلك لم يقوم بقراءة وتنفيذ محتويات الدالة عند بداية تشغيل التطبيق.

الآن لنقم بالضغط على الزر الي يقوم بتغيير الخاصية `loggedIn`، ونرى النتيجة، كالتالي:



## Parent Component

Change name Property from Parent Component

Change loggedIn Property from Parent Component

please Login..

Console

top

Filter

All levels

Angular is running in the development mode. core.js:40480  
Call enableProdMode() to enable the production mode.

[Violation] 'setTimeout' handler zone-evergreen.js:2550  
took 50ms

[WDS] Live Reloading enabled. client:52

loggedIn property changed child.component.ts:40

نلاحظ نفذ الكود الخاص عند تغير القيمة في الخاصية loggedIn فقط والذي هو طباعة رسالة في console، الآن لنقم بالضغط على زر تغيير الخاصية name، كالتالي:

## Parent Component

Change name Property from Parent Component

Change loggedIn Property from Parent Component

please Login..

Console

top

Filter

All levels

Angular is running in the development mode. core.js:40480  
Call enableProdMode() to enable the production mode.

[Violation] 'setTimeout' handler zone-evergreen.js:2550  
took 50ms

[WDS] Live Reloading enabled. client:52

loggedIn property changed child.component.ts:40

name property changed child.component.ts:36

نلاحظ نفذ الكود عن تغير هذه الخاصية.

وبذلك نكون قمنا بتغطية اهم المفاهيم للاتصال وارسال البيانات من الابن إلى الأب عن طريق @Input وسوف نتكلم بإذن الله في الجزء التالي عن كيفية الاتصال من الأب إلى الأب عن طريق @Output.

### 2.3.2. الاتصال من component الأب إلى الأب عن طريق @Output():

كما اننا نستطيع ان نتواصل ونرسل البيانات من component الأب إلى الأب عن طريق Decorator @Input()، فإننا ايضاً نستطيع ان نرسل البيانات بطريقة عكسية من الأب إلى الأب عن طريق @Output()، ولكن هنا مختلف قليلاً حيث ان الأب يستمع إلى الأب من خلال حدث event معين وعند وقوع هذا الحدث في component الأب يتم تمرير البيانات عن طريق هذا الحدث، إلى ملف template في component الأب ومن ثم عن طريق Event Binding يتم تمرير هذه البيانات من ملف template إلى ملف class في component الأب.

ولتوضيح لنعطي مثال، لو رجعنا إلى المثال السابق لوجدنا اننا قمنا بتمرير قيمتين إلى component الأب وهما name و loggedIn ونقوم بتغيير قيمهما عن طريق component الأب، ولكن ماذا لو اردنا لأي سبب من الأسباب ان نقوم بتغيير هذه القيم في component الأب ونجعل الأب يستمع إلى هذه القيم وفي حال حدوث أي تغيير في قيمهما يقوم بتحديث القيمة المتغير الخاصة الموجودة عنده ومن ثم يقوم بإعادة تمريرها إلى الأب عن طريق Input مرة أخرى، وقد يتراود إلى ذهنك عزيز المتعلم وما الفائدة من ذلك لنجعل الوضع على ما هو عليه، ولتوضيح هذه المسألة تخيل معي عزيزي المتعلم لو انه لدينا اكثر من أب لنفس الأب وجميع هذه الأبناء مشتركة بمتغير (خاصية) واحدة تم تمريرها إلى جميع الأبناء ونريد ان يستمع الأب إلى المتغير (الخاصية) لدى جميع الأبناء عن طريق حدث معين، وفي حال وقوع أي تعديل على قيمة هذا المتغير سيقوم الأب بتحدث قيمة المتغير لديه ومن ثم يقوم بإعادة ارسالها إلى جميع الأبناء، ففي هذه الحالة وغيرها من الحلول المطروحة يُفضل استخدام @Output().

الآن لنقم بتنفيذ المثال وأول خطوة نقوم بها هي حذف الأزرار من ملفي template و class في component الأب، كالتالي:

ملف app.component.html

```
<h3>Parent Component</h3>
<app-child [name]="name" [loggedIn]="loggedIn"></app-child>
```

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  name = 'Faisal';
  loggedIn = true;
```

```

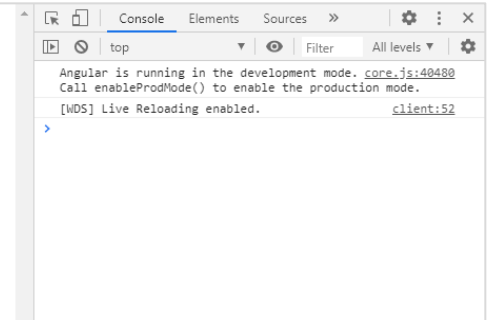
constructor() {}
ngOnInit(): void {}
}

```

والنتيجة:

## Parent Component

Welcome Back Faisal from Child Component



وكما قلنا سابقاً اننا سوف نرسل البيانات عند وقوع حدث معين لذلك سوف ننشأ زر في component الأب وعند وقوع حدث click عليه ننفذ دالة ولنسمي هذه الدالة changeStatus(), كالتالي:

ملف child.component.html

```

<h5>{{message}}</h5>
<button (click)="changeStatus()">Change loggedIn Property from Child Component</button>

```

ومهمة هذه الدالة هو في حال الضغط على هذا الزر تقوم بتغيير قيمة المتغير loggedIn الداخلة إلى هذا component عن طريق الأب بواسطة @Input(), ولكن قبل أن نغير قيمة هذا المتغير لابد أن نرئي @Output() وذلك من خلال متغير جديد نقوم بتعريفه ولابد أن يكون من النوع <EventEmitter> وهو generic وهذا المتغير هو الذي يستمع إليه component الأب، كالتالي:

ملف child.component.ts

```

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit, OnChanges {

  @Output() newStatus: EventEmitter<boolean> = new EventEmitter<boolean>();
}

```



```

@Input('name') name: string;

private _loggedIn: boolean;
message: string;

get loggedIn(): boolean {
  return this._loggedIn;
}

@Input() set loggedIn(value: boolean) {
  this._loggedIn = value;
  this.message = value ? `Welcome Back ${this.name} from Child Component` : 'please Login..';
}

constructor() { }

ngOnInit(): void { }

ngOnChanges(changes: SimpleChanges): void {
  for (const propertyName in changes) {
    if (changes.hasOwnProperty(propertyName) && !changes[propertyName].firstChange) {
      switch (propertyName) {
        case 'name': {
          console.log('name property changed');
          break;
        }
        case 'loggedIn': {
          console.log('loggedIn property changed');
          break;
        }
      }
    }
  }
}
}

```

نلاحظ عرفنا متغير جديد باسم `newStatus` وهو من النوع `EventEmitter` وبما انه `generic` مررنا له النوع `boolean` لأن قيمة المتغير `loggedIn` الداخل إلى `component` الأبن هو من النوع `boolean`، الآن بعدما هيئنا هذا المتغير على انه `Output` لنقوم بعمل تغيير القيمة ونعمل لها `emit` عن طريق المتغير الجديد `newStatus`، كالتالي:

ملف `child.component.ts`

```

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

```

```

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit, OnChanges {

  @Output() newStatus: EventEmitter<boolean> = new EventEmitter<boolean>();

  @Input('name') name: string;

  private _loggedIn: boolean;
  message: string;

  get loggedIn(): boolean {
    return this._loggedIn;
  }

  @Input() set loggedIn(value: boolean) {
    this._loggedIn = value;
    this.message = value ? `Welcome Back ${this.name} from Child Component` : 'please Login..';
  };


  constructor() { }

  ngOnInit(): void { }

  ngOnChanges(changes: SimpleChanges): void {
    for (const propertyName in changes) {
      if (changes.hasOwnProperty(propertyName) && !changes[propertyName].firstChange) {
        switch (propertyName) {
          case 'name': {
            console.log('name property changed');
            break;
          }
          case 'loggedIn': {
            console.log('loggedIn property changed');
            break;
          }
        }
      }
    }
  }

  changeStatus() {
    this.newStatus.emit(!this._loggedIn);
  }
}

```




نلاحظ عند كل ضغطة على الزر تُنفذ هذه الدالة وهو تمرير القيمة العكسية للمتغير بحيث إذا كانت false تصبح true وإذا كانت true تصبح false.

الآن لنقوم بتهيئة component الأب للاستماع إلى المتغير الخاصية (newStatus) وفي حال تغيير قيمتها ينفذ دالة معينة وهذه الدالة تقوم بوضع القيمة الجديدة للمتغير loggedIn، كالتالي:

ملف app.component.html

```
<h3>Parent Component</h3>
<app-child [name]="name" [loggedIn]="loggedIn" (newStatus)="changeStatus($event)"></app-child>
```



نلاحظ اننا عملنا Event Binding للمتغير newStatus (لأبدا ان يكون نفس الاسم في component الأب) ونفذنا دالة اسميتها changeStatus (لك حرية اختيار الاسم الذي تريده) ومررنا له باراميترو وهو عبارة عن الكائن \$event (وهنا هذا الكائن يشير إلى القيمة التي تم تمريرها من component الأب إلى الأب)، الآن لنقم بتنفيذ محتوى هذه الدالة، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  name = 'Faisal';
  loggedIn = true;

  constructor() {}
  ngOnInit(): void {}

  changeStatus(status: boolean) {
    this.loggedIn = status;
  }
}
```

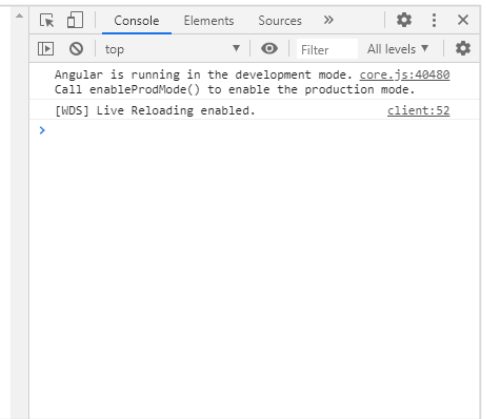


ولنشاهد النتيجة في المتصفح، كالتالي:

## Parent Component

Welcome Back Faisal from Child Component


Change loggedIn Property from Child Component



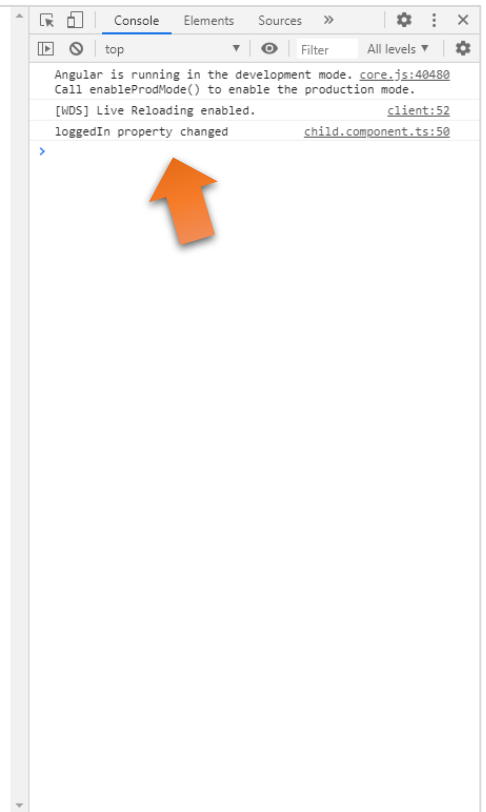
الآن لنقوم بالضغط على الزر لكي نغير القيمة ونرى النتيجة، كالتالي:

ويجب ملاحظة أن هذا الزر موجود في component الأب، والأب يستمع إلى التغييرات وفي حال حدوثها يقوم بتحديث قيمة المتغير loggedIn بحسب القيمة الجديدة، وبشكل تلقائي تتحدث قيمة المتغير loggedIn الداخلة إلى component الأب عن طريق @Input().

## Parent Component

please Login.. 

Change loggedIn Property from Child Component



نلاحظ تم التغيير، وبذلك نكون مررنا القيمة من الأب إلى الابن عن طريق event معين (الضغط على الزر) وبنفس الوقت الأب يستمع إلى هذا الحدث وفي حال وقوعه ينفذ هو الآخر مهمة معينة.

الآن لنقوم بتنفيذ نفس الخطوات ولكن هذه المرة على المدخل name وذلك من خلال إضافة زر ومربع إدخال في component الأب وعند كتابة قيمة داخل مربع الإدخال والضغط على الزر ننفذ دالة تقوم بقراءة القيمة ونعمل لها emit إلى component الأب، وبنفس الوقت الأب يستمع إلى التغييرات وعند حدوثها يحدث قيمة الخاصية الموجودة لديه name بالقيمة الجديدة، كالتالي:

```

<h5>{{message}}</h5>
<input type="text" #inputValue>
<br><br>
<button
  (click)="changeName(inputValue.value)"
>
  Change name Property from Child Component
</button>
<button (click)="changeStatus()">Change loggedIn Property from Child Component</button>

```

```

import {
  Component,
  OnInit,
  Input,
  OnChanges,
  SimpleChanges,
  EventEmitter,
  Output
} from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit, OnChanges {
  @Output() newName: EventEmitter<string> = new EventEmitter<string>();
  @Output() newStatus: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Input() name: string;
  private _loggedIn: boolean;
  message: string;
  get loggedIn(): boolean {
    return this._loggedIn;
  }
  @Input() set loggedIn(value: boolean) {
    this._loggedIn = value;
    this.message = value ? `Welcome Back ${this.name} from Child Component` : 'please Login.';
  }
  constructor() { }
  ngOnChanges(changes: SimpleChanges): void {
    for (const propertyName in changes) {
      if (changes.hasOwnProperty(propertyName) && !changes[propertyName].firstChange) {
        switch (propertyName) {
          case 'name': {
            this.message = this._loggedIn ?
              `Welcome Back ${this.name} from Child Component` : 'please Login..';
            break;
          }
          case 'loggedIn': {
            console.log('loggedIn property changed');
            break;
          }
        }
      }
    }
  }
}

```



```

    }
  }
}
}
changeName(value: string) {
  this.newName.emit(value);
}
changeStatus() {
  this.newStatus.emit(!this._loggedIn);
}
}

```

ملف app.component.html

```

<h3>Parent Component</h3>
<app-child
  [name]="name"
  [loggedIn]="loggedIn"
  (newName)="changeName($event)"
  (newStatus)="changeStatus($event)"
>
</app-child>

```

ملف app.component.ts

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

  name = 'Faisal';
  loggedIn = true;

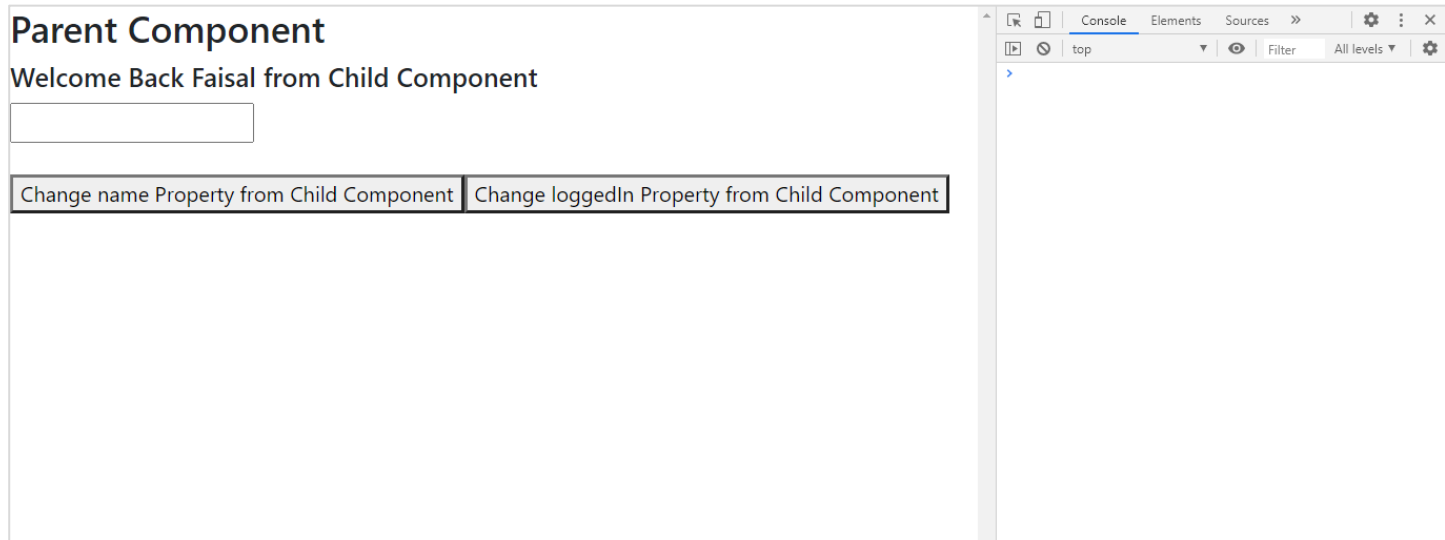
  constructor() {}
  ngOnInit(): void {}

  changeName(value: string) {
    this.name = value;
  }

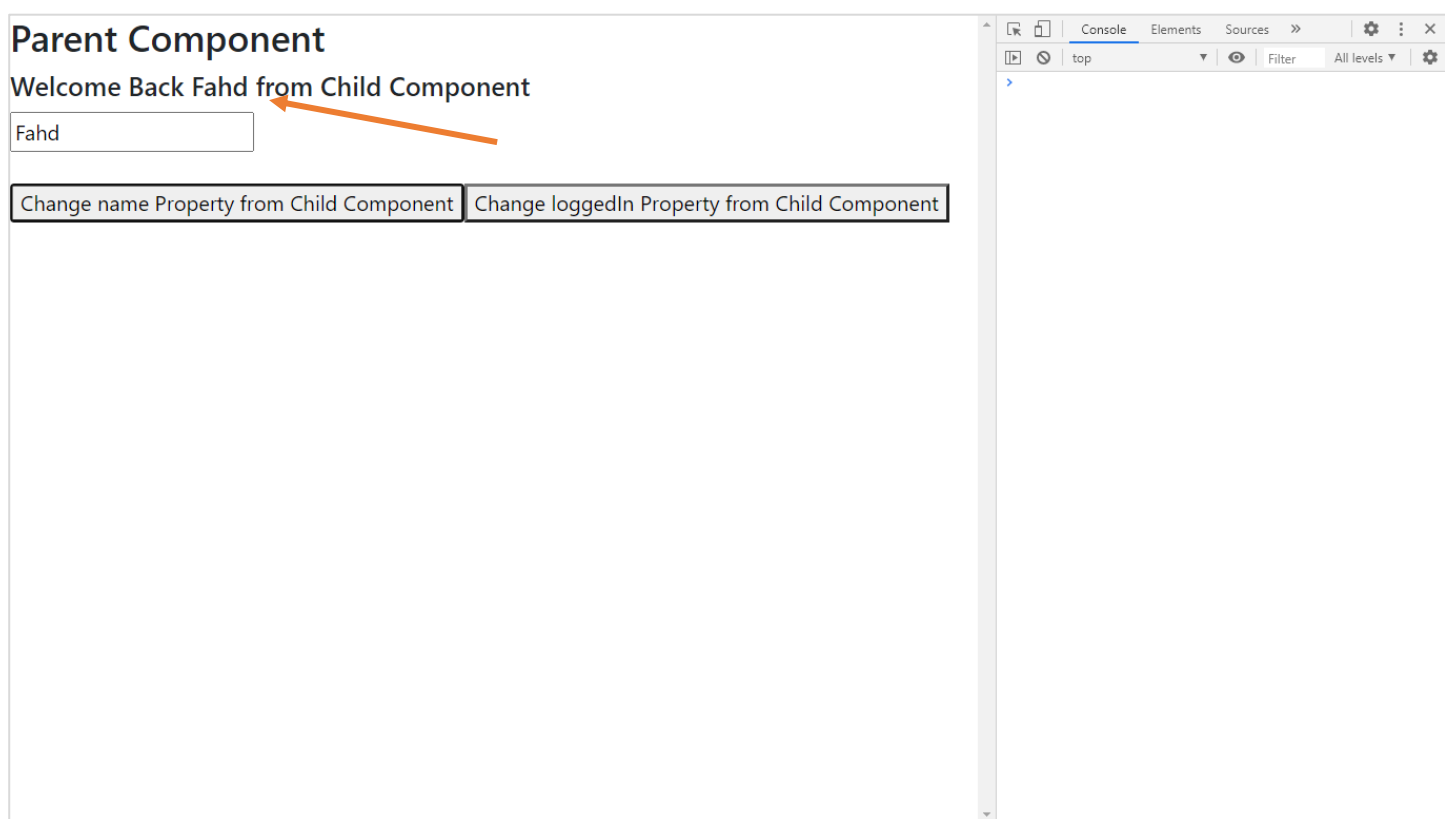
  changeStatus(status: boolean) {
    this.loggedIn = status;
  }
}

```

ولنذهب إلى المتصفح لنرى النتيجة، كالتالي:



لنقم بكتابة أي اسم في مربع الدخال ونضغط على زر تغيير الخاصية name، كالتالي:



وبذلك نكون غطينا اهم المفاهيم المتعلقة بالاتصال وتمرير البيانات من component الأب إلى component الابن، وسوف نتكلم في الجزء التالي عن كيفية الوصول إلى خاصية أو عنصر محدد من component الأب إلى الابن سواء عن طريق ملف template أو ملف class.

### 3.3.2. الوصول إلى خصائص اودوال Component الأب عن طريق Component الأب:

تتيح لنا Angular طرق متعددة للوصول إلى جميع الخصائص Properties والدوال Methods الموجودة في ملف class لي component الأب عن طريق ملفي class و template لي component الأب، وسوف نتكلم عن كلا الطريقتين، كالتالي:

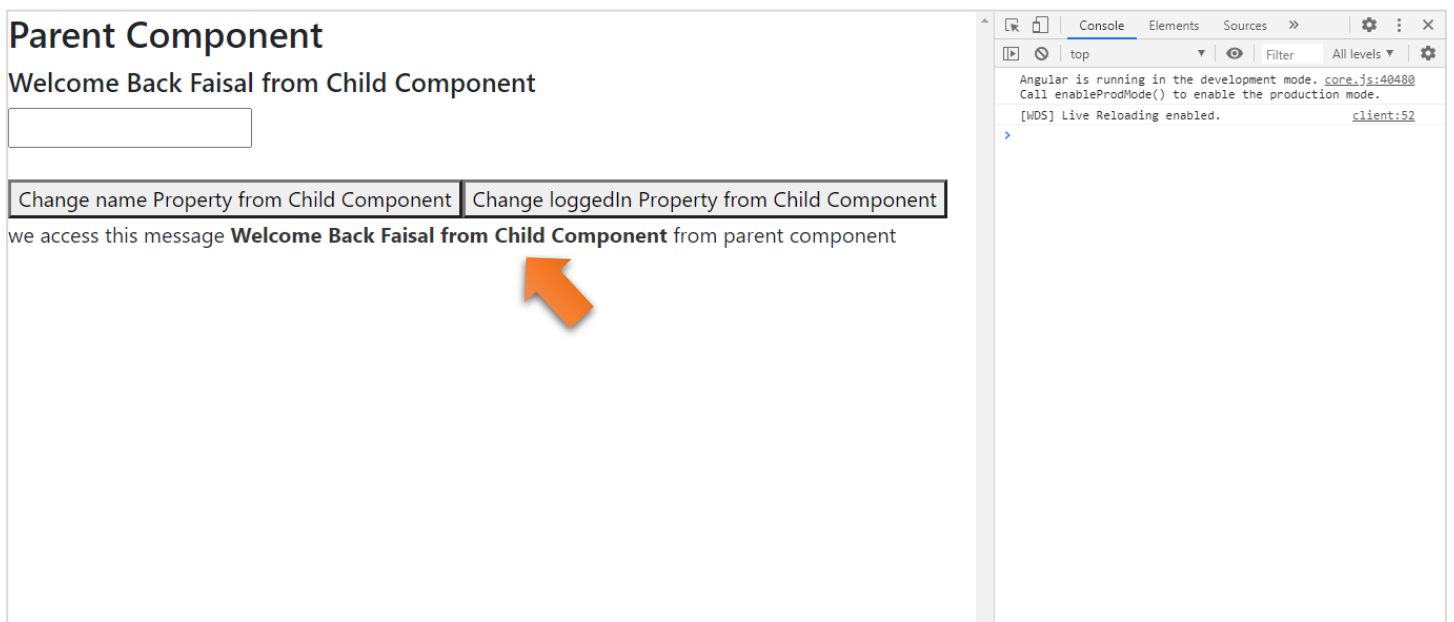
### 1.3.3.2. الوصول للخصائص والدوال عن طريق ملف template لـ component الأب:

قلنا سابقاً في المقدمة ان component في الأساس هو عبارة عن Class وكما هو معروف ان أي Class يحتوي على خصائص ودوال ونستطيع الوصول إلى جميع هذه الخصائص والدوال إلا private او protected (وهذا لا علاقة لي Angular به فهو يختص بمفاهيم OOP)، ذلك قدمت لنا Angular طريقة للوصول إلى هذه الخصائص والدوال الموجودة في component الأب عن طريق ملف template في component الأب، ونستطيع ذلك بكل بساطة عن طريق إضافة Template Reference إلى التاغ <app-child></app-child>، ومن ثم عن طريق هذا Reference نستطيع الوصول للذي نريده.

ولتوضيح لنعطي مثال بحيث نصل إلى الخاصية message التي عرفناها في component الأب عن طريق ملف template في component الأب، كالتالي:

```
ملف app.component.html
<h3>Parent Component</h3>
<app-child
  #child
  [name]="name"
  [loggedIn]="loggedIn"
  (newName)="changeName($event)"
  (newStatus)="changeStatus($event)"
>
</app-child>
<p>
  we access this message <strong>{{ child.message }}</strong> from parent component
</p>
```

والنتيجة:



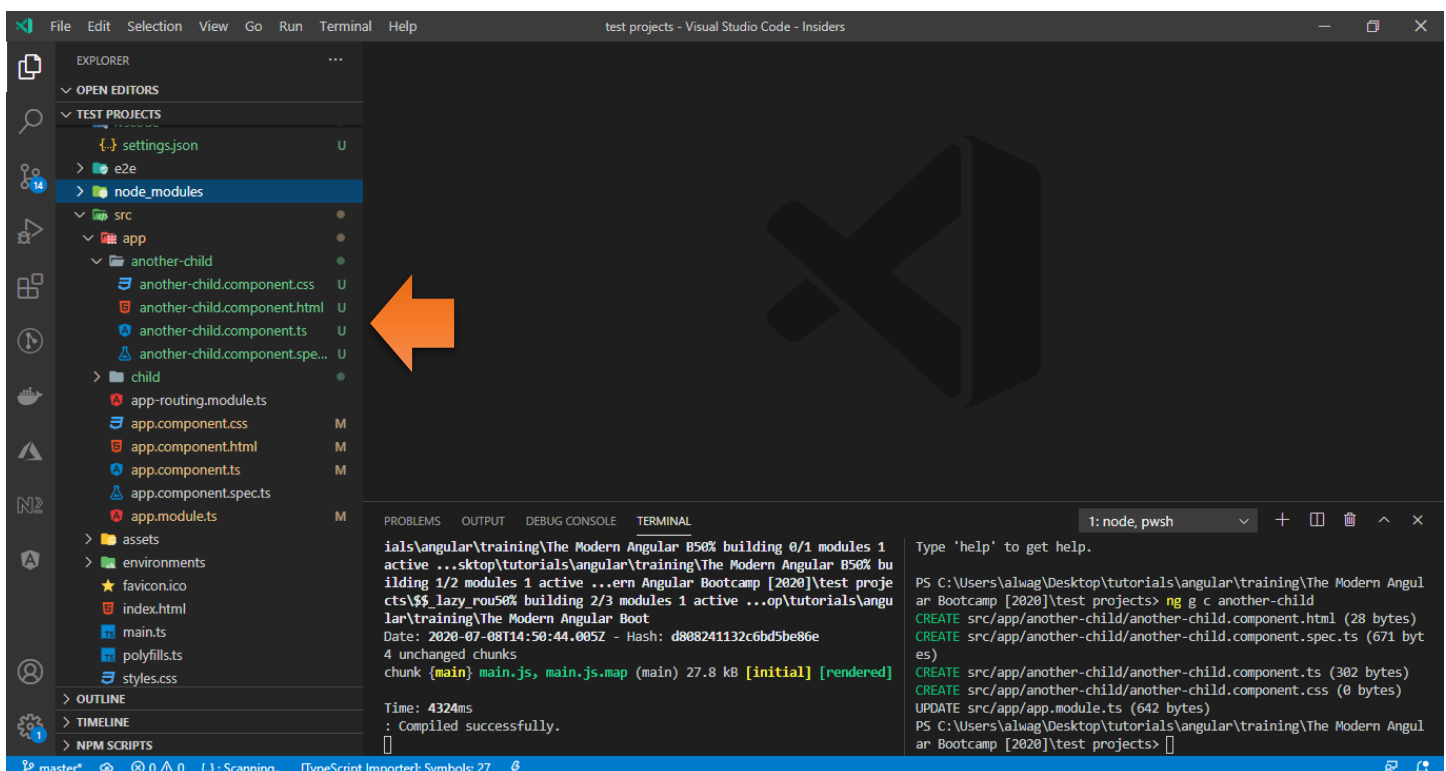
### 2.3.3.2. الوصول للخصائص والدوال عن طريق ملف class في component الأب:

كما أنه يمكننا أن نصل إلى جميع الخصائص والدوال عن طريق ملف template لملف class في component الأب فمن باب أولى ان يكون هنالك طريقة للوصول لملف class في component الأب عن طريق ملف class في component الأب، ونستطيع

القيام بذلك عن طريق `@ViewChild` و `@ViewChildren` حيث بدلاً من أن نحقق هذين Decorators بي Template Reference لعناصر HTML الموجودة في ملف template كما كنا نفعل سابقاً، نقوم بحققها هذه المرة بي Template Reference لي Selector الخاص بي component الأب، أو نقوم بحققه باسم الكلاس لي component الأب مباشرة، وسوف نتكلم عن كلا الطريقتين مع الأمثلة.

### 1.2.3.3.2. الوصول للخصائص والدوال عن طريق ملف class في component الأب باستخدام `@ViewChild()`:

لكيلا نُعيد تكرار ما قلناه سابقاً في هذا Decorator سوف نقوم بإعطاء المثال بشكل مباشر، لنفرض انه لدينا component أب آخر ويقوم بعرض قائمة معينة للمستخدم، ونريد لأي سبب من الأسباب أن نقوم بإظهار وإخفاء هذه القائمة عن طريق Component الأب، طبعاً هنالك طرق متعددة للقيام بهذا الأمر منها (وليس أفضلها ولكن يخدمنا لتوضيح ما نريد توضيحه هنا) ان نقوم بتعريف خاصية في component الأب الآخر واسمها مثلاً `showItems` ونسند لها قيمة منطقية ولتكن `true` ونعمل لها `Bind` في ملف template لهذا component ونضع شرط إذا كانت `true` اظهر هذه القائمة وإذا كانت `false` احذف هذه القائمة من DOM، وفي component الأب نضع زر وهذا الزر يُنفذ دالة عن الضغط عليه ومهمة هذه الدالة الوصول إلى ملف class لي component الأب وبالتحديد للخاصية `showItems` والتحكم بقيمتها، لذلك في البداية لنقم بإنشاء هذا component الأب الجديد، كالتالي:



وفي ملف class لهذا الأب نقوم بتعريف المتغير `showItems` ونسند له القيمة `true`، كالتالي:

```
another-child.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-another-child',
```

```

    templateUrl: './another-child.component.html',
    styleUrls: ['./another-child.component.css'],
  })
  export class AnotherChildComponent implements OnInit {

    showItems = true;

    constructor() {}

    ngOnInit(): void { }

  }

```

وفي ملف template نقوم ببناء القائمة ووضع الشرط، كالتالي:

ملف another-child.component.html

```

<div *ngIf="showItems">
  <ul *ngFor="let item of [1, 2, 3, 4, 5, 6]">
    <li>Note NO.{{ item }}</li>
  </ul>
</div>

```

ولنضع Selector في ملف template لي Root Component لكي يصبح اب لهذا component الأبن الجديد، وبنفس الوقت لنقم بإنشاء زر عن الضغط عليه يُنفذ دالة معينة وليكن أسم هذه الدالة على سبيل المثال showHideItems كالتالي:

ملف app.component.html

```

<strong>
  Parent Component
  <button (click)="showHideItems()">Show/Hide Items</button>
</strong>
<br /><br />
<app-child
  #child
  [name]="name"
  [loggedIn]="loggedIn"
  (newName)="changeName($event)"
  (newStatus)="changeStatus($event)"
>
</app-child>
<p>
  we access this message <strong>{{ child.message }}</strong> from parent component
</p>
<app-another-child></app-another-child>

```

والآن لنقم بحفظ العمل ولنذهب إلى المتصفح لنرى النتيجة:

Parent Component

Show/Hide Items

Welcome Back Faisal from Child Component

Change name Property from Child Component

Change loggedIn Property from Child Component

we access this message **Welcome Back Faisal from Child Component** from parent component

Note NO.1

Note NO.2

Note NO.3

Note NO.4

Note NO.5

Note NO.6

Console

Elements

Sources

top

Filter

All levels

Angular is running in the development mode. core.js:40480

Call enableProdMode() to enable the production mode.

[WDS] Live Reloading enabled. client:52

الآن لنقم بتطبيق مفهوم @ViewChild والتي تتيح لنا الوصول إلى جميع الخصائص والدوال في component الأب عن طريق ملف class في component الأب، كالتالي:

ملف app.component.ts

```

import {
  Component,
  OnInit,
  ViewChild,
} from '@angular/core';
import { AnotherChildComponent } from '../another-child/another-child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit, AfterViewChecked {
  @ViewChild(AnotherChildComponent) anotherChild: AnotherChildComponent;

  name = 'Faisal';
  loggedIn = true;

  constructor() {}
  ngOnInit(): void {}

  showHideItems() {
    this.anotherChild.showItems = !this.anotherChild.showItems;
  }

  changeName(value: string) {
    this.name = value;
  }

```

```

}

changeStatus(status: boolean) {
  this.loggedIn = status;
}
}

```

نلاحظ أننا اخذنا نسخة instance من class لي component الأب عن طريق ViewChild ووضعناها في كائن اسميناه anotherChild وهو أيضاً من النوع class لي component وهو AnotherChildComponent، بحيث أصبح هذا الكائن يمتلك حق الوصول لجميع خواص ودوال هذا class، ولذلك قمنا بالوصول إلى الخاصية showItems في الدالة showHideItems، اما النتيجة في المتصفح، كالتالي:

Parent Component

Show/Hide Items

Welcome Back Faisal from Child Component

Change name Property from Child Component

Change loggedIn Property from Child Component

we access this message **Welcome Back Faisal from Child Component** from parent component

Note NO.1

Note NO.2

Note NO.3

Note NO.4

Note NO.5

Note NO.6

Console

Elements

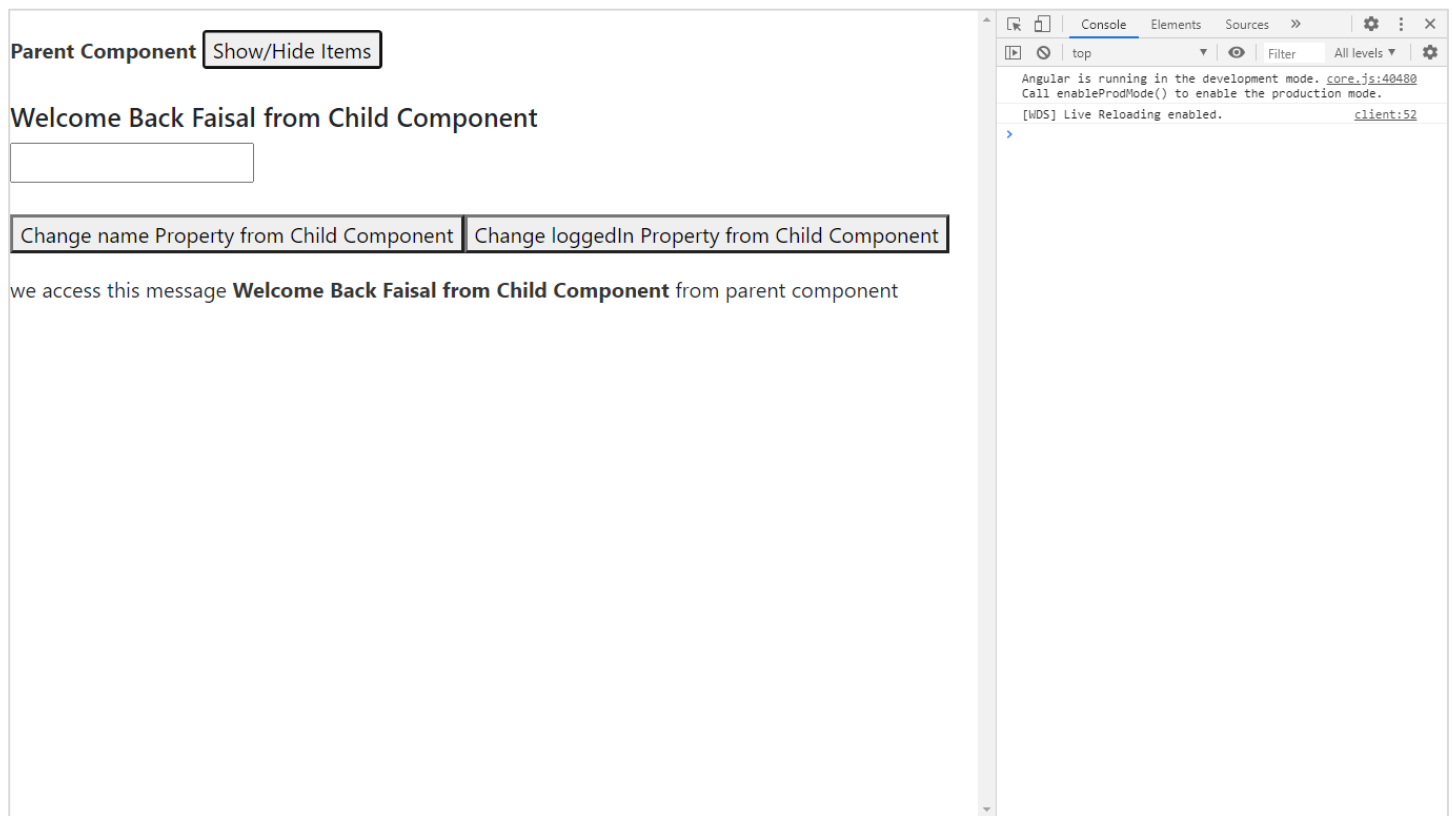
Sources

Angular is running in the development mode. core.js:40480

Call enableProdMode() to enable the production mode.

[WDS] Live Reloading enabled. client:52

وعند الضغط على الزر، تكون النتيجة:



نلاحظ تم إخفاء القائمة، مع العلم أنه نستطيع ان نضع template reference لي `<app-another-child></app-another-child>` كالتالي:

جزء من ملف `app.component.html`

```
<app-another-child #AnotherChild ></app-another-child>
```

ومن ثم نمرر هذا reference إلى `@ViewChild` بين علامتي تنصيص بدلاً من اسم class لي component الأب، كالتالي:

جزء من ملف `app.component.ts`

```
@ViewChild('AnotherChild') anotherChild: AnotherChildComponent;
```

ولو جربنا لوجدنا انها تعمل بدون مشاكل.

## 2.2.3.3.2. الوصول للخصائص والدوال عن طريق ملف class باستخدام `@ViewChildren()`:

وهي مشابهة لاستعمالها عندما نريد الوصول لعنصر من عناصر DOM عن طريق ملف class والتي شرحناها سابقاً، ولكن هنا على مستوى component، لذلك لنقوم بتغيير بسيط في الكود ليصبح بالشكل التالي:

```
<strong>Parent Component
  <button (click)="showHideItems()">Show/Hide Items</button>
</strong>
<br /><br />
<app-child
  #child [name]="name"
  [loggedIn]="loggedIn"
```




```

        (newName)="changeName($event)"
        (newStatus)="changeStatus($event)"
    >
</app-child>
<p>
    we access this message <strong>{{ child.message }}</strong> from parent component
</p>

<app-another-child
    #AnotherChild
    *ngFor="let item of [1, 2, 3, 4, 5, 6]"
    [item]="item"
>
</app-another-child>

```



#### another-child.component.html ملف

```

<div *ngIf="showItems">
    <ul>
        <li>Note NO.{{ item }}</li>
    </ul>
</div>

```

#### another-child.component.ts ملف

```

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

@Component({
    selector: 'app-another-child',
    templateUrl: './another-child.component.html',
    styleUrls: ['./another-child.component.css'],
})
export class AnotherChildComponent implements OnInit {
    showItems = true;
    @Input() item: number;

    constructor() {}

    ngOnInit(): void {}
}

```

#### app.component.ts ملف

```

import {
    Component,
    OnInit,
    ViewChild,
    ViewChildren,
    QueryList,
} from '@angular/core';

import { AnotherChildComponent } from './another-child/another-child.component';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',

```

```

styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {
  @ViewChild(AnotherChildComponent) anotherChild: AnotherChildComponent;
  @ViewChildren('AnotherChildren') anotherChildren: QueryList<AnotherChildComponent>;
  name = 'Faisal';
  loggedIn = true;


  constructor() {}
  ngOnInit(): void {}

  showHideItems() {
    this.anotherChildren.forEach((item) => {
      item.showItems = !item.showItems;
    });
    // this.anotherChild.showItems = !this.anotherChild.showItems;
  }

  changeName(value: string) {
    this.name = value;
  }

  changeStatus(status: boolean) {
    this.loggedIn = status;
  }
}

```



نلاحظ اننا استبدلنا السطر البرمجي السابق بالسطر الحالي لأن في الأول كنا نتعامل مع عنصر واحد ولكن الآن نتعامل مع اكثر من عنصر.

وبذلك أنهينا طرق التعامل مع الخصائص والدوال ولكن ماذا لو أردنا الوصول إلى عنصر محدد (عنصر HTML) موجود في component الأب عن طريق ملف class لي component الأب، وهذا ما سوف نتكلم عنه في الجزء التالي.

### 3.3.3.2. الوصول للعناصر في ملف Template الخاص بال component الأب عن طريق ملف class لل component الأب:

نستطيع الوصول لجميع العناصر في ملف template الخاص بي component الأب عن طريق عمل ViewChild او ViewChildren لهذه العناصر من ملف template لملف class لنفس component الأب ومن ثم نعمل ViewChild او ViewChildren من ملف class الخاص بال component الأب إلى ملف class في component الأب، كما فعلنا قبل قليل، وسوف نتكلم عن كلا الطريقتين بإذن الله.

### 1.3.3.3.2. الوصول للعناصر في ملف Template الخاص بي component الأب عن طريق ملف class لي

#### الcomponent الأب عن طريق @ViewChild:

لتوضيح هذه الجزئية لنعطي مثال، لو نظرنا إلى ملف template الخاص بالcomponent الأب الأول child.component.html لوجدنا انه لدينا input وله template reference، لذلك نريد أن نصل إلى هذا العنصر input من ملف class الخاص بالcomponent الأب، وللقيام بذلك كما ذكرنا قبل قليل في البداية نقوم بعمل ViewChild لهذا input من ملف template إلى ملف class، كالتالي:

ملف child.component.html

```
<h5>{{ message }}</h5>
<input type="text" #inputValue />
<br /><br />
<button (click)="changeName(inputValue.value)">
  Change name Property from Child Component
</button>
<button (click)="changeStatus()">
  Change loggedIn Property from Child Component
</button>
```

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})
export class ChildComponent implements OnInit, OnChanges {
  private _loggedIn: boolean;
  message: string;

  @ViewChild('inputValue') inputValue: ElementRef;

  @Output() newName: EventEmitter<string> = new EventEmitter<string>();
  @Output() newStatus: EventEmitter<boolean> = new EventEmitter<boolean>();

  @Input() name: string;

  get loggedIn(): boolean {
    return this._loggedIn;
  }
```

```

}

@Input() set loggedIn(value: boolean) {
  this._loggedIn = value;
  this.message = value
    ? `Welcome Back ${this.name} from Child Component`
    : 'please Login..';
}

constructor() {}

ngOnInit(): void {}

ngOnChanges(changes: SimpleChanges): void {
  for (const propertyName in changes) {
    if (
      changes.hasOwnProperty(propertyName) &&
      !changes[propertyName].firstChange
    ) {
      switch (propertyName) {
        case 'name': {
          this.message = this._loggedIn
            ? `Welcome Back ${this.name} from Child Component`
            : 'please Login..';
          break;
        }
        case 'loggedIn': {
          console.log('loggedIn property changed');
          break;
        }
      }
    }
  }
}

changeName(value: string) {
  this.newName.emit(value);
}

changeStatus() {
  this.newStatus.emit(!this._loggedIn);
}
}

```

وبما انه أصبح الآن في ملف class، نستطيع ان نعمل له ViewChild كما كنا نعمل بالسابق، كالتالي:

ملف app.component.ts

```

import {
  Component,
  OnInit,
  ViewChild,
  ViewChildren,
  QueryList,

```

```

    AfterViewInit,
    Renderer2,
  } from '@angular/core';

import { AnotherChildComponent } from '../another-child/another-child.component';
import { ChildComponent } from '../child/child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild(ChildComponent) child: ChildComponent;
  @ViewChild(AnotherChildComponent) anotherChild: AnotherChildComponent;
  @ViewChildren('AnotherChildren') anotherChildren: QueryList<AnotherChildComponent>;

  name = 'Faisal';
  loggedIn = true;

  constructor(private renderere: Renderer2) {}
  ngOnInit(): void {}

  ngAfterViewInit(): void {
    this.renderere.selectRootElement(this.child.inputValue.nativeElement).focus();
  }

  showHideItems() {
    this.anotherChildren.forEach((item) => {
      item.showItems = !item.showItems;
    });
    // this.anotherChild.showItems = !this.anotherChild.showItems;
  }

  changeName(value: string) {
    this.name = value;
  }

  changeStatus(status: boolean) {
    this.loggedIn = status;
  }
}

```

ولو قمنا بتجربة التطبيق لوجدنا اننا استطعنا الوصول إلى عنصر input والموجود في component الأب عن طريق ملف class في component الأب، والتحكم به وهنا قمنا فقط بإضافة focus للعنصر عند بداية التطبيق، أي بمعنى وضع مؤشر الكتابة بشكل تلقائي داخل هذا input عند بداية تشغيل التطبيق.


## 2.3.3.3.2. الوصول للعناصر في ملف Template الخاص بي component الأب عن طريق ملف class لي

### الcomponent الأب عن طريق @ViewChildren:

ولتوضيح هذه الجزئية سوف نقوم بإعطاء مثال، لنفرض أننا نريد الوصول لجميع عناصر li الموجودة في ملف template في component الأب الثاني another-child.component.html، ونقوم بتغيير لون الخط لهذه العناصر جميعاً، وكما فعلنا في السابق نكرره الآن ولكن مع ViewChildren، كالتالي:

ملف another-child.component.html

```
<div *ngIf="showItems">
  <ul>
    <li #Li>Note NO.{{ item }}</li>
  </ul>
</div>
```



ملف another-child.component.ts

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

@Component({
  selector: 'app-another-child',
  templateUrl: './another-child.component.html',
  styleUrls: ['./another-child.component.css'],
})
export class AnotherChildComponent implements OnInit {
  showItems = true;
  @ViewChildren('Li') li: QueryList<ElementRef>;
  @Input() item: number;
  constructor() {}
  ngOnInit(): void {}
}
```



وبما أنه قمنا بعمل ViewChildren لهذا component من قبل في component الأب، فسوف نستفيد منها للوصول إلى هذه العناصر وتغيير لون النص، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  ViewChild,
  ViewChildren,
  QueryList,
  AfterViewInit,
  Renderer2,
```

```

} from '@angular/core';
import { AnotherChildComponent } from '../another-child/another-child.component';
import { ChildComponent } from '../child/child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild(ChildComponent) child: ChildComponent;
  @ViewChild(AnotherChildComponent) anotherChild: AnotherChildComponent;
  @ViewChildren('AnotherChildren') anotherChildren: QueryList<AnotherChildComponent>;

  name = 'Faisal';
  loggedIn = true;

  constructor(private renderer: Renderer2) {}
  ngOnInit(): void {}

  ngAfterViewInit(): void {
    this.renderer
      .selectRootElement(this.child.inputValue.nativeElement)
      .focus();
    this.anotherChildren.forEach((item) => {
      const el = item.li.first.nativeElement;
      this.renderer.setStyle(el, 'color', 'red');
    });
  }

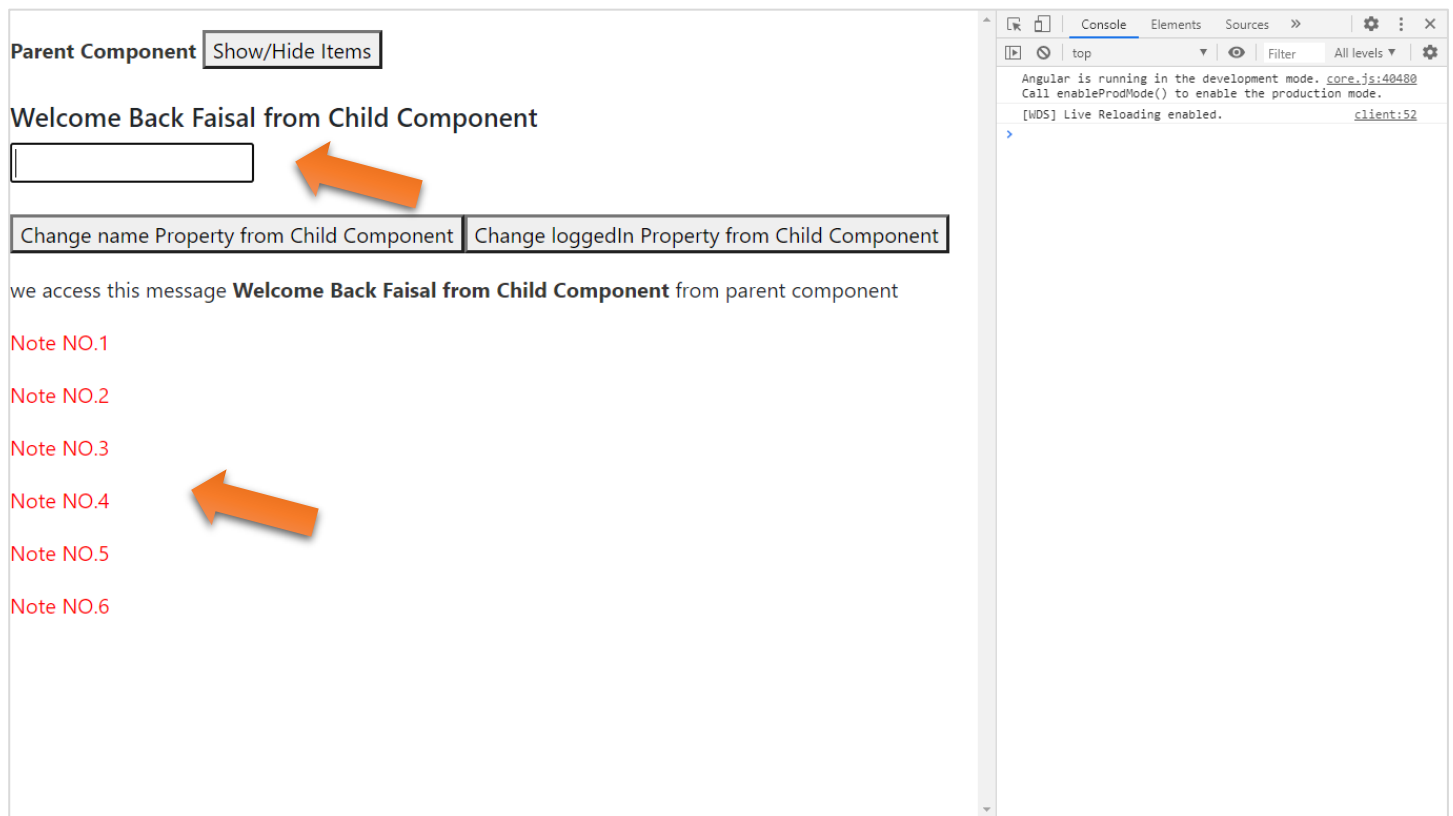
  showHideItems() {
    this.anotherChildren.forEach((item) => {
      item.showItems = !item.showItems;
    });
    // this.anotherChild.showItems = !this.anotherChild.showItems;
  }

  changeName(value: string) {
    this.name = value;
  }

  changeStatus(status: boolean) {
    this.loggedIn = status;
  }
}

```

ولنقم بحفظ العمل ونرى النتيجة، كالتالي:



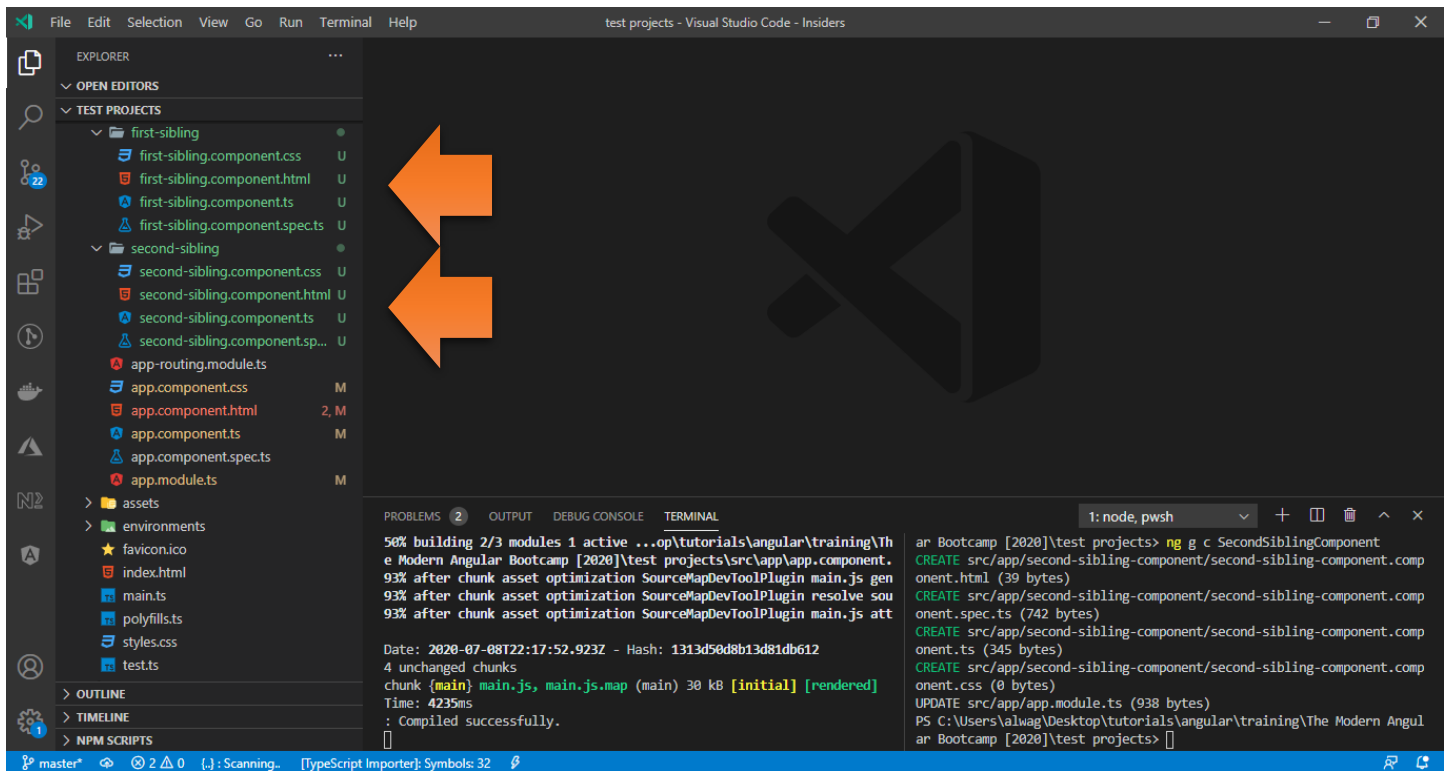
نلاحظ المؤشر موجود في المربع الادخال عند بداية التطبيق وبنفس الوقت استطعنا الوصول وتغيير لون الخط لجميع هذه العناصر.

وبذلك نكون قمنا بتغطية أغلب طرق التواصل التي تختص بالcomponent التي بينها علاقة أب وأبن، اما باقي العلاقات فهي عامة لتواصل بين أي components سواء بينهما علاقة ام لا، لذلك سوف نركز هنا فقط بأنواع الاتصال التي تختص كل علاقة على حدا، وفي الفصل التالي Services سوف نتكلم عن هذه الأنواع بإذن الله.

## 4.2. التواصل بين Components المختلفة التي تربطهم علاقة (أخ – أخ):

حقيقة هذا النوع ليس ميزة جديدة تحتاج منا لشرحها وإنما هي مزج بين @Input() و @Output() بطريقة معينة لكي نحقق الاتصال بين components التي تربطهم علاقة أخوة sibling، وتقوم الفكرة بأن نقوم بإرسال البيانات من component الأول عن طريق Output إلى الأب ومن ثم نقوم بتمريرها إلى component الثاني عن طريق الأب باستخدام Input، ولتوضيح لنضرب مثال، ولنقم في البداية بإنشاء مشروع جديد وإضافة اثنين components ولنسمي الأول باسم FirstSibling والثاني ليكن باسم SecondSibling والأب المشترك بينهم هو Root Component، كالتالي:





ولنضع Selector الخاص بهذه components الأبناء في Root Component كما تعلمنا سابقاً، مع إضافة بعض كلاسات مكتبة bootstrap (لمعرفة كيفية إضافة مكتبات خارجية إلى المشروع الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup) لتحسين المظهر فقط لا غير، كالتالي:

```

app.component.html ملف
<div class="container">
  <h3>Employees</h3>
  <div class="row">
    <div class="col-lg-6">
      <app-first-sibling></app-first-sibling>
    </div>
    <div class="col-lg-6">
      <app-second-sibling></app-second-sibling>
    </div>
  </div>
</div>

```

والمطلوب في هذا المثال نريد ان ننشأ قائمة في الأخ الأول تحتوي على قائمة بأسماء الموظفين وعند الضغط على زر بجانب كل موظف نريد أن يظهر لنا بيانات هذا الموظف في component الأخ الآخر، لذلك لنقوم بتجهيز الأخ الأول من ناحية ملف template وملف class، كالتالي:

```

first-sibling.component.ts ملف
import { Component, OnInit } from '@angular/core';
import { Employee } from '../employee';

@Component({
  selector: 'app-first-sibling',
  templateUrl: './first-sibling.component.html',

```

```

    styleUrls: ['./first-sibling.component.css'],
  })
export class FirstSiblingComponent implements OnInit {

  employees: Employee[] = [
    {
      id: 1,
      name: 'Faisal',
      city: 'Riyadh',
    },
    {
      id: 2,
      name: 'Fahd',
      city: 'Dammam',
    },
    {
      id: 3,
      name: 'Saad',
      city: 'Makkah',
    },
    {
      id: 4,
      name: 'Bader',
      city: 'Jeddah',
    },
  ];

  constructor() {}

  ngOnInit(): void {}
}

```

للمعلومية النوع Employee هو عبارة عن custom type ومهمته فقط وصف البيانات حيث قمت بإنشاء ملف باسم employee.ts وهو عبارة عن class، ومحتوياته كالتالي:

```

export class Employee {
  id: number;
  name: string;
  city: string;
}

```

ومن ثم استدعيت هذا الملف في first-sibling.component.ts، اما الآن لنكمل مع ملف template للأخ الأول، كالتالي:

ملف first-sibling.component.html

```

<p>Number of Employees: {{ employees.length }}</p>
<ul class="list-group">
  <li class="list-group-item" *ngFor="let employee of employees">
    {{ employee.name }}
    <button class="btn btn-dark float-right">View</button>
  </li>
</ul>

```

اما الأخ الآخر فمهمته عرض البيانات، وسوف نقوم بتجهيز ملف template الخاص به، كالتالي:

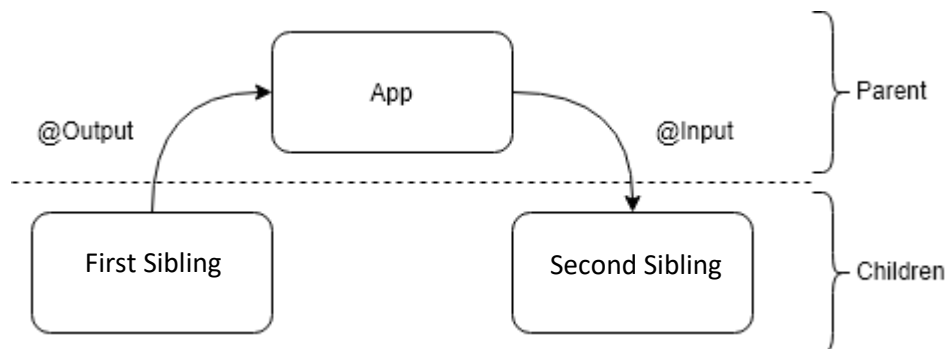
ملف second-sibling.component.html

```
<div class="card">
  <div class="card-header">

</div>
<div class="card-body">
  <h5 class="card-title"></h5>
</div>
</div>
```

اما ملف class لهذا component فلا يحتوي على شيء إلا الآن، ولنقوم بحفظ هذه التعديلات ونرى النتيجة، كالتالي:


الآن لنقوم بعرض بيانات، وكما قلنا سابقاً نقوم الفكرة على إرسال البيانات من الأخ الأول إلى الأب عن طريق @Output ومن ثم يقوم الأب بتمريرها إلى الأبن عن طريق @Input، كما في الشكل التالي:



لذلك لنقم في البداية بعمل @Output للبيانات عند الضغط على الزر بجانب اسم كل موظف، وذلك من خلال انشاء دالة وليكن اسمها employeeDetails، ونمرر لها الكائن الذي يشير إلى الموظف الحالي، كالتالي:

ملف first-sibling.component.html

```
<p>Number of Employees: {{ employees.length }}</p>
<ul class="list-group">
  <li class="list-group-item" *ngFor="let employee of employees">
    {{ employee.name }}
    <button
      class="btn btn-dark float-right"
      (click)="employeeDetails(employee)"
    >
      View
    </button>
  </li>
</ul>
```




ملف first-sibling.component.ts

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

@Component({
  selector: 'app-first-sibling',
  templateUrl: './first-sibling.component.html',
  styleUrls: ['./first-sibling.component.css'],
})
export class FirstSiblingComponent implements OnInit {
  @Output() employee: EventEmitter<Employee> = new EventEmitter<Employee>();

  employees: Employee[] = [
    {
      id: 1,
      name: 'Faisal',
      city: 'Riyadh',
    },
    {
      id: 2,
      name: 'Fahd',
      city: 'Dammam',
    },
    {
      id: 3,
      name: 'Saad',
      city: 'Makkah',
    },
    {
      id: 4,
      name: 'Bader',
      city: 'Jeddah',
    },
  ];
};
```




```

constructor() {}

ngOnInit(): void {}

employeeDetails(employee: Employee): void {
  this.employeeData.emit(employee);
}
}

```



وفي component الأب نستقبل هذه القيمة متمثلة في employee، ومن ثم ننفذ دالة أخرى وليكن اسمها ()getData وتستقبل باراميترو وهو بيانات الموظف الذي مررناه للcomponent الأب قبل قليل، كالتالي:

ملف app.component.html

```

<div class="container">
  <h3>Employees</h3>
  <div class="row">
    <div class="col-lg-6">
      <app-first-sibling (employee)="getData($event)"></app-first-sibling>
    </div>
    <div class="col-lg-6">
      <app-second-sibling></app-second-sibling>
    </div>
  </div>
</div>

```

ومهمة هذه الدالة بكل بساطة هي تخزين القيمة في متغير جديد وهذا المتغير هو الذي نممره للأخ الآخر، كالتالي:

ملف app.component.ts

```

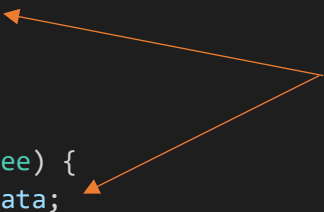
import { Component, OnInit } from '@angular/core';
import { Employee } from './employee';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {
  employee: Employee;
  constructor() {}
  ngOnInit(): void {}

  getData(data: Employee) {
    this.employee = data;
  }
}

```



والآن لنمرر هذا المتغير الجديد employee إلى component الأخ الآخر، كالتالي:

ملف app.component.html

```

<div class="container">
  <h3>Employees</h3>

```

```

<div class="row">
  <div class="col-lg-6">
    <app-first-sibling (employee)="getData($event)"></app-first-sibling>
  </div>
  <div class="col-lg-6">
    <app-second-sibling [employee]="employee"></app-second-sibling>
  </div>
</div>
</div>

```

الآن لنستقبل هذه القيمة في component الأخ الآخر عن طريق `@Input`، كالتالي:

ملف `second-sibling.component.html`

```

import { Component, OnInit, Input } from '@angular/core';
import { Employee } from '../employee';

@Component({
  selector: 'app-second-sibling',
  templateUrl: './second-sibling.component.html',
  styleUrls: ['./second-sibling.component.css'],
})
export class SecondSiblingComponent implements OnInit {
  @Input() employee: Employee;

  constructor() {}

  ngOnInit(): void {}
}

```

وأخيراً نقوم بعمل `bind` لهذه القيم إلى ملف `template` لعرضها، كالتالي:

ملف `second-sibling.component.html`

```

<div class="card" *ngIf="employee">
  <div class="card-header">{{ employee.name }}</div>
  <div class="card-body">{{ employee.city }}</div>
</div>

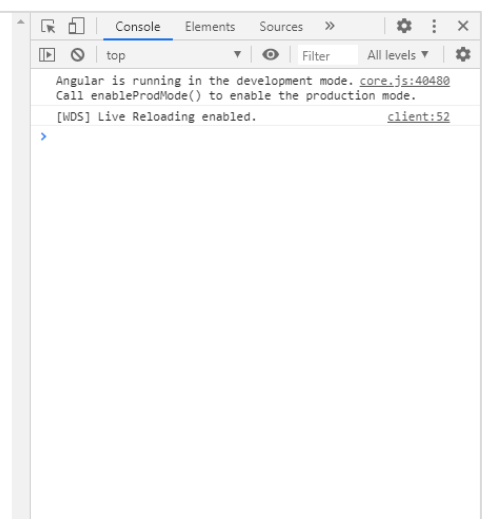
```

ولنشاهد النتيجة في المتصفح، كالتالي:

## Employees

Number of Employees: 4

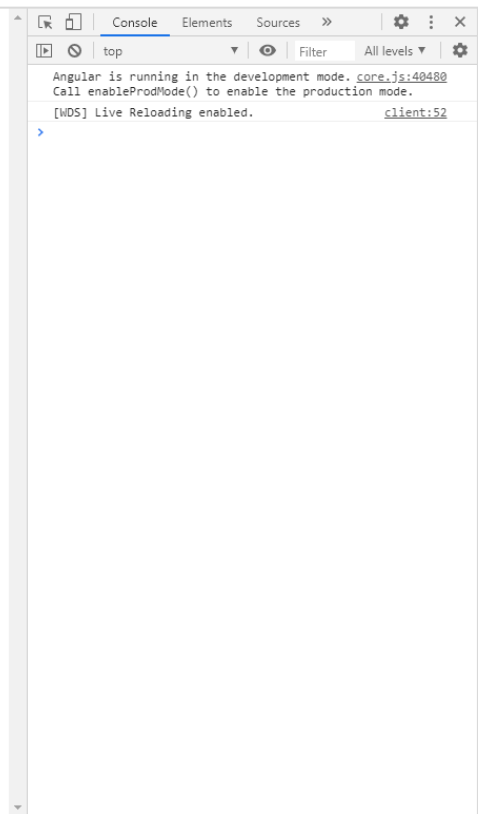
Faisal	View
Fahd	View
Saad	View
Bader	View



## Employees

Number of Employees: 4

Faisal	View
Fahd	View
Saad	View
Bader	View
Fahd	
Dammam	



وبذلك نكون أنهينا اغلب طرق التواصل بين components سواء كانت الملفات بنفس component أو التواصل بين Components المختلفة التي تربطهم علاقة معينة، وبقي أخيراً أن نتكلم عن طرق التواصل بين components التي لا تربطهم أي علاقة، عن طريق Services وفي الحقيقة هذه الطرق تنفع لتواصل بين أي components سواء كانت بينهم علاقة ما، أو لم يكن، وبما أن هذه الطرق جميعها تستخدم Services، لذلك سوف نفرد فصل كامل للكلام عن Services ومفهومها وكيف يتم الاستفادة منها لتواصل مع Components.

# الفصل الثالث

## Services



### 1.3. مقدمة:

وهي عبارة عن ملفات Typescript تحتوي على class وهذا class تم عمل له export مثلها مثل ملفات component ولكن هنا مُعرفة Decorator ذو الاسم (@Injectable) بعكس component المعرفة بي (@Component)، ويحتوي هذا Decorator على خاصية واحدة وهي provider ويسند له القيمة root، اما فائدة هذا Decorator هو إعلام Angular ان هذا الملف هو service وتطبق عليه آلية Dependency Injection (أحدى تقنيات Design Patterns التي تعتبر من مفاهيم OOP وهي ليست مقتصرة على Angular وانما يتم استخدامها بأغلب لغات البرمجة التي تتبع OOP) وهذه التقنية بشكل مبسط وبعيداً عن تعقيداتها تتيح لنا أن نعمل نسخة (كائن) instance من هذه service في أي component او service آخر او حتى أي ملف Typescript يحتوي على كلاس في دالة constructor الخاصة به، بحيث تمتلك هذه النسخة جميع الصلاحيات وتستطيع الوصول إلى جميع خصائص ودوال هذه service، وقد تعاملنا سابقاً في هذا الكتاب مع هذه التقنية عندما قمنا بعمل inject لي service ذات الاسم Renderer2 في constructor، طبعاً هذه التقنية لها تعقيداتها وقواعدها وليس هنا المقام لشرحها ولكن الذي اريدك ان تعرفه اننا إذا اردنا ان نبني أي Service لابد من وجود هذا Decorator فيها، لكي نستطيع الاستفادة من Dependency Injection المبني ضمناً في إطار عمل Angular والذي يتيح ان نعمل inject لهذه service في أي ملف Typescript يحتوي على كلاس عن طريق دالة constructor الخاصة به، اما الخاصية Provider وقيمتها root الموجودة ضمن هذا Decorator فهي الطريقة الحديثة لتعريف وتسجيل هذا الكلاس في مشاريع Angular حيث كان سابقاً نعرف أي service ضمن ملف app.module.ts ونضع أسماء هذه Services في الخاصية Providers على شكل مصفوفة، اما الآن فلا نحتاج ان نقوم بهذا الأمر فالخاصية provider والقيمة root تفي بهذا الغرض، اما مهمة هذه الخاصية فهي لإعلام Angular ان يُعرف او بصيغة أخرى ان يقوم بتسجيل هذه Services على انها Singleton (وهو ايضاً Design Patterns تبناه إطار عمل Angular ويستخدمه ضمناً في مشاريعه ومهمته بشكل مبسط ان يقوم بتنفيذ هذه الملفات مرة واحدة عن بداية تشغيل التطبيق وتكون موجودة في الذاكرة RAM مادام التطبيق يعمل، بحيث تكون الأكواد موجودة في هذه الملفات Services متاحة لجميع النسخ (الكائنات) للوصول إليها، وهذا Design Patterns لا يختص به Angular وانما هو عام لجميع لغات البرمجة التي تتبع أسلوب وتقنيات (OOP)، ومن هذا المنطلق ممكن يتراود إلى ذهنك هل نستطيع ان نجعل هذه services لا تعمل تحت لواء Singleton والاجابة نعم صحيح هنالك بعض الحالات التي لا تُريد ان تكون هذه Services تعمل طوال الوقت الذي يعمل به التطبيق وتستنزف مساحة من الذاكرة ونحن لسنا بحاجة اليها وانما لها مهمة محددة تنتهي بانتهاء هذه المهمة ويتم حذفها من الذاكرة RAM ويستمر التطبيق في العمل وعند الحاجة اليها يتم استدعائها مرة أخرى، وسوف نتطرق إلى هذه الحالات لاحقاً بإذن الله.

والسؤال الآخر الذي قد يتراود إلى ذهنك عزيزي المتعلم هو ما هو الفرق بين ملف class في component وملف class في services، وهذا ما سوف نتكلم عنه في النقطة التالية، بإذن الله.

### 2.3. الفرق بين services وcomponents:

لهم الفرق لناخذ هذا الاقتباس من الموقع الرسمي لإطار العمل Angular الذي يصف المهمة الأساسية لـ Components، كالتالي:

“Do limit logic in a component to only that required for the view. All other logic should be delegated to services.”

Angular Style Guide

Style 05-15 (July 2017)

<https://angular.io/guide/styleguide#delegate-complex-component-logic-to-services>

ومعنى هذا الاقتباس بشكل مبسط أن ملف class الخاص بالcomponent يُكتب فيه جميع Logic الخاص بالتحكم بآلية وطرق عرض البيانات في ملف template او ما يسمى ملف view وأي logic آخر يتم نقله إلى ملف service منفصل، ومن الأمثلة لو كان لدينا Logic يتعامل مع api لجلب بيانات من قاعدة البيانات او حذفها او الإضافة عليها او تعديلها فجميع هذه functionality يتم كتابتها في ملف service منفصل وأي component يُريد الاستفادة من هذه functionality نقوم بعمل inject لهذه service في ملف class لهذا component عن طريق Dependency Injection، ومن الأمثلة الأخرى لو اردنا ان نعمل مشاركة لمجموعة من البيانات بين مجموعة من components، وبهذه الحالة نستطيع ان نقول ان هذا النوع من Services يتبع Singleton Design Patterns، ومن الأمثلة ايضاً أنه في حال كان Logic لا يؤثر بشكل مباشر على عرض البيانات او التحكم في ملف template ولو كان يتم استخدامه مره واحده لي component واحد، ونستطيع ان نقول في هذه الحالة نستطيع ان نجعل هذا component لا يعمل تحت لواء Singleton Design Patterns، لأنه يؤدي مهمة لي component واحد ونستطيع ان نستدعيه في حال احتيجنا له من خلال هذا component فقط.

ومما قيل سابقاً نستطيع ان نلخص الهدف من انشاء أي Service، من خلال النقاط التالية:

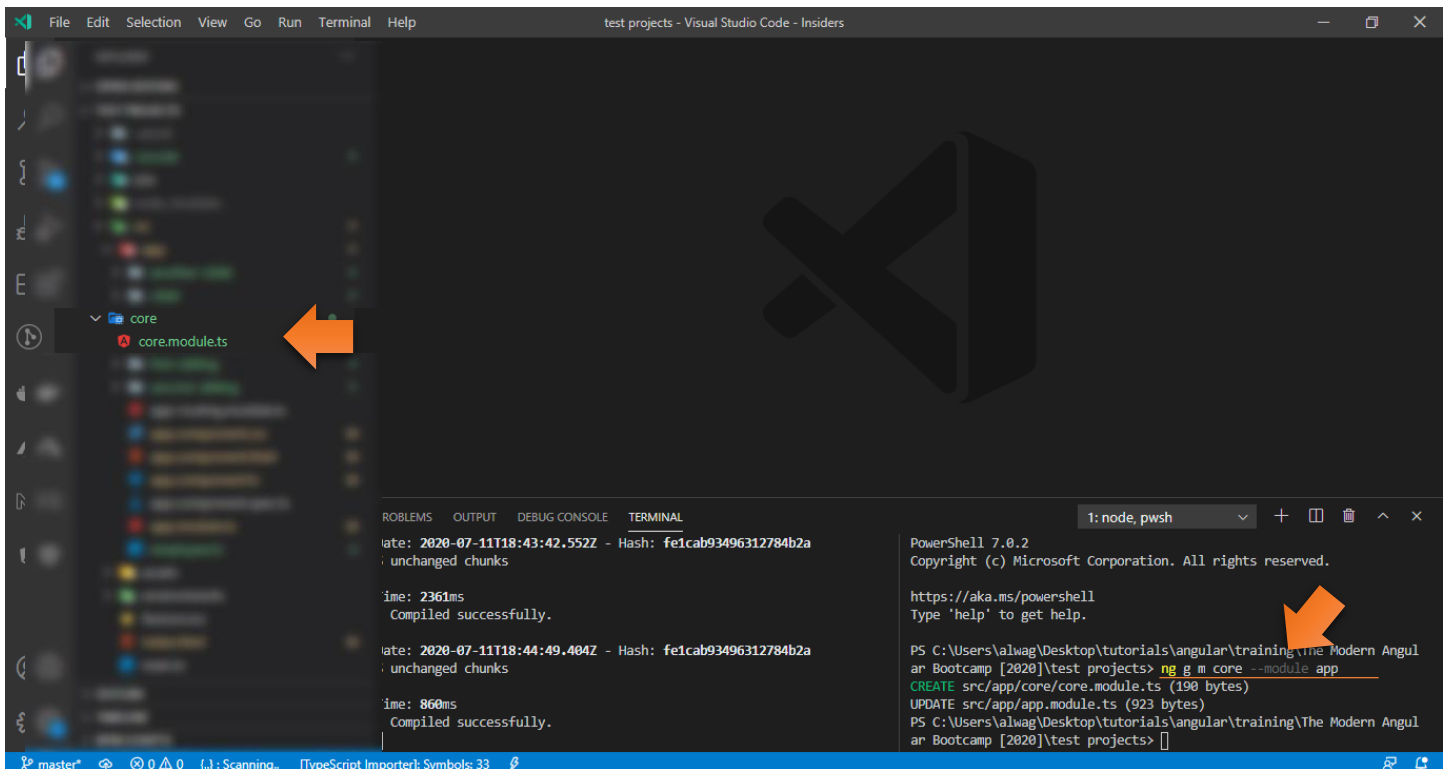
- في حال كان Logic او Functionality غير مطلوبة بشكل مباشر لتعامل مع ملف template.
- في حال كان Logic او Functionality ممكن استخدامها في أماكن متفرقة في التطبيق كدوال التعامل مع قاعدة البيانات، او كان هذا Logic ممكن نُعيد استخدامه في أكثر من component.
- في حال كان هنالك بيانات معينة مخزنة في متغيرات (خصائص) وأردنا ان نشارك هذه البيانات في مجموعة من components.

بعد هذا الشرح الموجز عن الفرق بين component و service، يجب ان نبين كيف ممكن ان ننظم ملفات services وخصوصاً إذا كان التطبيق من متوسط إلى كبير ولذلك ينصح الفريق المشرف على إطار عمل Angular أن نقوم بتنظيم ملفات Services من خلال Core Module، وهو ما سوف نتكلم عنه في الجزء التالي.

### 3.3. تنظيم ملفات Services من خلال Core Module:

ليس هنا المقام لشرح Modules وأنواعها، ولفهم أعمق عن Modules الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular Routing and Modules، لذلك سوف اختصر على الجزء الخاص بكيفية استخدامها لتنظيم ملفات Services، مع العلم ان Services التي يتم تنظيمها في Core Module هي التي تتبع مفاهيم Singleton Design Patterns، اما Services التي لا تتبع هذه المفاهيم فستطيع وضعها في Shared Modules إذا كنا نستخدم Lazy Loading او نضعها في نفس component إذا كنا لا نستخدمها إلا في هذا component فقط، وهنا سوف اركز فقط على طريقة تنظيم ملفات services في Core Modules لأنها الأكثر استخداماً وتقريباً ٩٠% من هذه services تتبع هذا النهج.

وبعد هذه المقدمة المطولة لنبدأ بالتطبيق العملي ولننشأ مشروع جديد وليكن باسم services باستخدام Angular CLI، وبعد إنشاء هذا المشروع للنشأ Module جديد باسم core، عن طريق كتابة هذا الأمر `ng g m core --module app` (لمعرفة كيفية التعامل مع Angular CLI وإنشاء Modules وغيرها من Features الأخرى الرجاء مراجعة الكتاب الأول من هذه السلسلة Angular Environment Setup)، كالتالي:



ولو استعرضنا محتويات الملف `app.module.ts` لوجدنا انه تم تعريف هذا module في قائمة imports، كالتالي:

#### ملف `app.module.ts`


```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CoreModule } from './core/core.module';

@NgModule({
  declarations: [
    AppComponent,
```

```

],
imports: [BrowserModule, AppRoutingModule, CoreModule],
providers: [],
bootstrap: [AppComponent],
})
export class AppModule {}

```



اما لو استعرضنا محتويات ملف core.module.ts، فسوف نجد التالي:

ملف core.module.ts

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class CoreModule { }

```

وينصح الفريق المشرف على إطار عمل Angular كتابة Logic معين في constructor الخاص بهذا Module لكي نتأكد انه لن يتم استدعاء هذا Module إلا في ملف app.module.ts فقط، لأن تكرار استدعاءها سوف يستهلك موارد الذاكرة مما يقلل من أداء التطبيق الخاص بك وخصوصاً في التطبيقات من المتوسطة إلى الكبيرة.

ملف core.module.ts

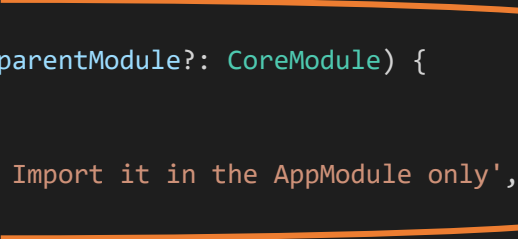
```

import { NgModule, SkipSelf, Optional } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [CommonModule],
})

export class CoreModule {
  constructor(@Optional() @SkipSelf() parentModule?: CoreModule) {
    if (parentModule) {
      throw new Error(
        'CoreModule is already loaded. Import it in the AppModule only',
      );
    }
  }
}

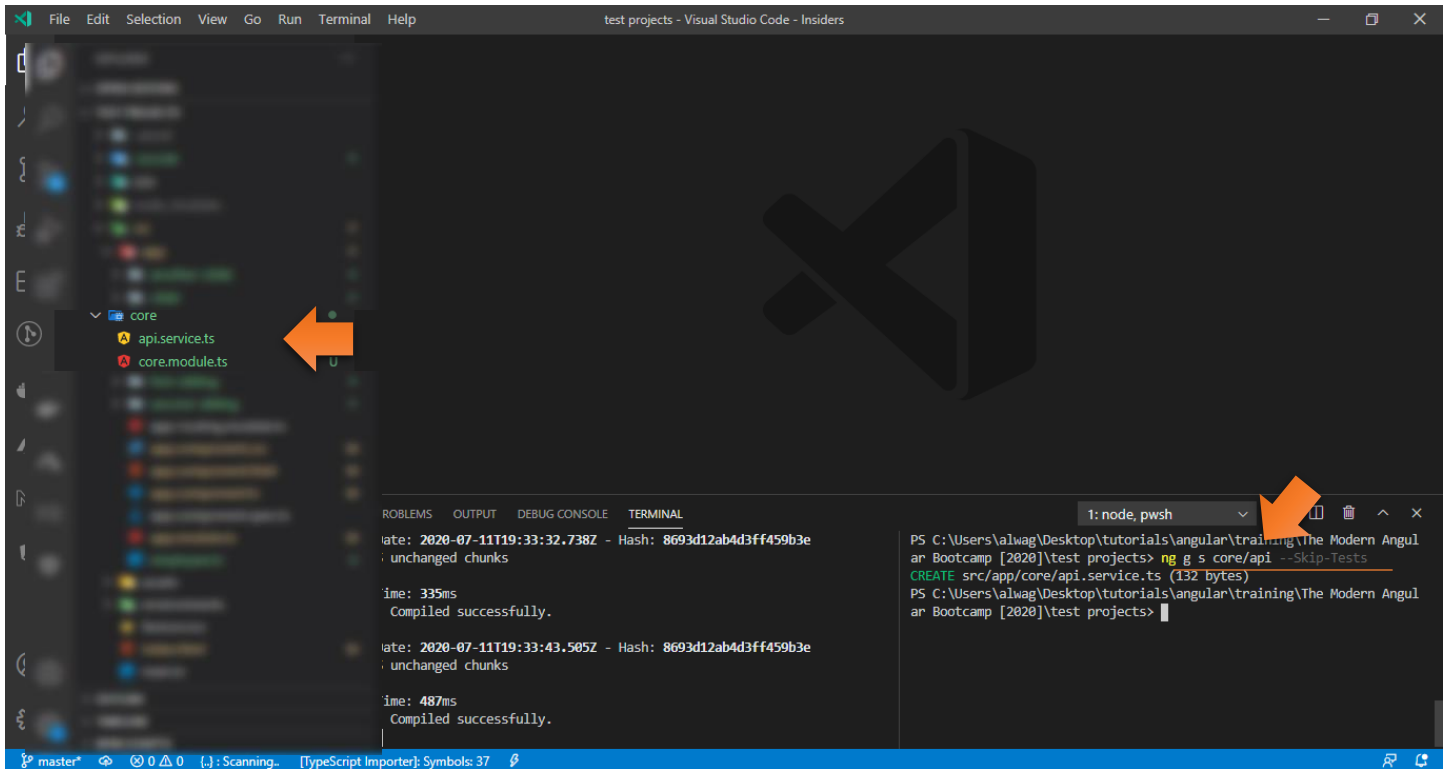
```



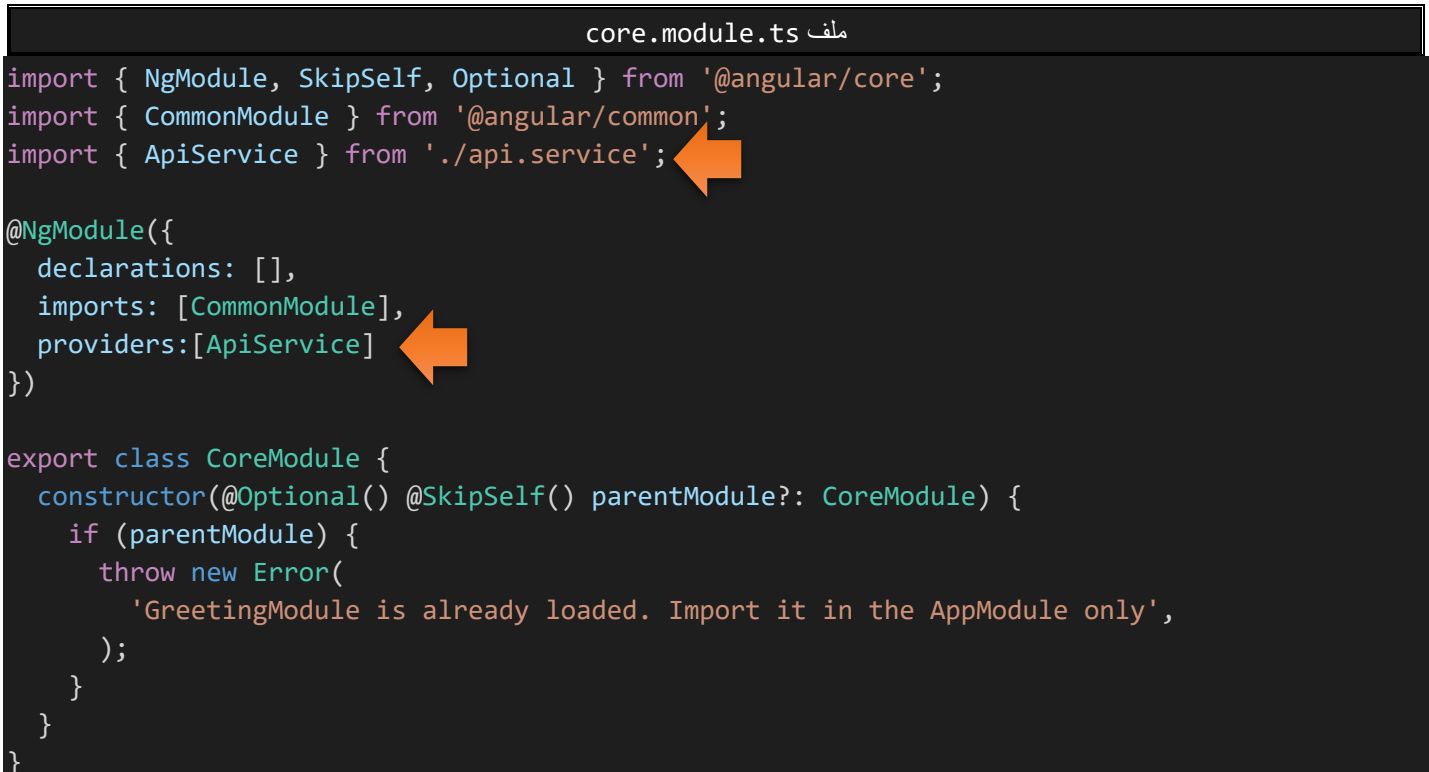
الآن أصبح هذا Module جاهز لاستقبال أي service جديدة. وهذا ما سوف نتكلم عنه في الجزئية التالية.

### 4.3. إضافة Services وشرح أجزاء الملف:

نستطيع إضافة service جديدة بكل سهولة عن طريق Angular CLI من خلال كتابة الأمر `ng g s` ومن ثم اسم الملف، وسوف نضيف هذا الملف في مجلد `core` الذي انشأناه سابقاً، وليكن اسمها `api`، كالتالي:



ولابد ان نعرف هذه service في ملف `core.module.ts`، عن طريق إضافة الخاصية `providers` ثم إضافة الكلاس الخاص بملف `service` إلى هذه الخاصية على شكل مصفوفة، كالتالي:



اما إذا استعرضنا محتويات ملف `service` فسوف نجد التالي:

```
import { Injectable } from '@angular/core'; ❶

@Injectable({
  providedIn: 'root' ❷
})

export class ApiService {
  constructor() { } ❸
}
```

رقم (1) استدعينا Decorator ذو الاسم Injectable.

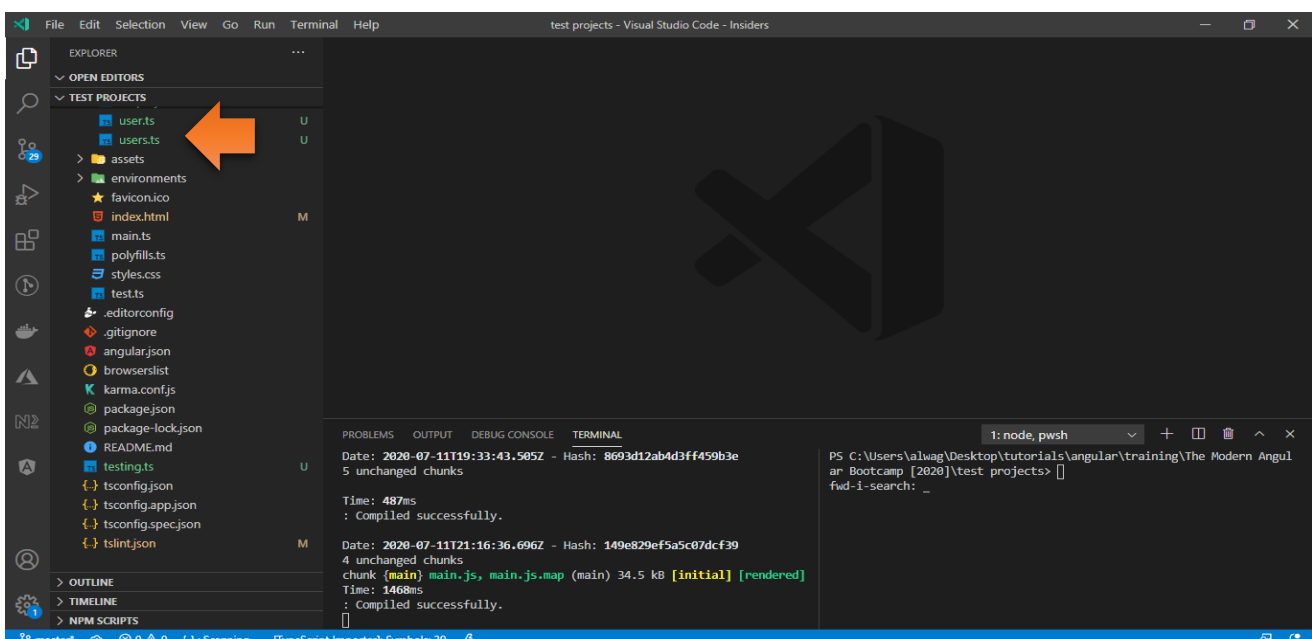
رقم (2) كتبنا Decorator على شكل كائن Object ويحتوي على الخاصية providedIn والقيمة المسندة له root وبذلك نستنتج أن أي service يتم انشاءها عن طريق Angular CLI بشكل افتراضي هي Singleton.

رقم (3) وهو عبارة عن class الخاص بهذا service ويحتوي على اسم class الذي عرفناه في ملف core.module.ts، وبداخل هذا class نكتب جميع Logic و Functionality.

وبعد استعراضنا لأغلب مفاهيم واساسيات service لنقم بإعطاء امثلة لبعض استخدامات هذه Services.

### 5.3. بناء Service تحتوي على Functionality ممكن يُعاد استخدامها في أكثر من مكان في التطبيق:

ولعل أشهر استخدامات هذه Functionality هي التعامل مع قواعد البيانات من جلب لهذه البيانات وإضافة وحذف وتعديل، ولكن بما انه ليس لدينا قاعدة بيانات و Angular لا يتعامل مع قاعدة البيانات بشكل مباشر من قراءة وإضافة وتعديل وحذف لهذه البيانات، لذلك سوف نقوم بإنشاء ملف ts يحتوي على بيانات غير حقيقة او ما تسمى Dummy Data، وبنفس الوقت ننشأ ملف آخر عبارة عن class او interface لكي يكون custom type (كما فعلنا سابقاً) لهذه البيانات لوصف أنواع هذه البيانات، كالتالي:



## ملف user.ts

```
export class User {
  name: string;
  city: string;
  phone: number;
  userName: string;
  password: number;
}
```

## ملف users.ts

```
import { User } from './user';

export const USERS: User[] = [
  {
    name: 'faisal',
    city: 'riaydh',
    phone: 555555555,
    userName: 'DevFaisal',
    password: 12345
  },
  {
    name: 'fahd',
    city: 'riaydh',
    phone: 444444444,
    userName: 'DevFahd',
    password: 8769876
  },
  {
    name: 'saad',
    city: 'Jeddah',
    phone: 333333333,
    userName: 'DevSaad',
    password: 765557
  },
  {
    name: 'bader',
    city: 'Dammam',
    phone: 22222222,
    userName: 'DevBader',
    password: 12345
  },
];
```

الآن لنقم بكتابة جميع Logic الخاص بالتعامل مع هذه البيانات CRUD (Create-Read-Update-Delete)، في ملف service  
 ذو الاسم api.service.ts، كالتالي:

## ملف api.service.ts

```
import { Injectable } from '@angular/core';
import { User } from '../user';
import { USERS } from '../users';
```

```

@Injectable({
  providedIn: 'root'
})

export class ApiService {

  constructor() { }

  createUser(user: User) {
    USERS.push(user);
  }

  getAllUsers(): User[] {
    return USERS;
  }

  getUserById(id: number): User {
    return USERS.find(user => user.id === id);
  }

  updateUser(user: User) {
    const userIndex = USERS.findIndex((obj => obj.id === user.id));
    USERS[userIndex].name = user.name;
    USERS[userIndex].city = user.city;
    USERS[userIndex].phone = user.phone;
    USERS[userIndex].userName = user.userName;
    USERS[userIndex].password = user.password;
  }

  deleteUser(id: number) {
    const userIndex = USERS.findIndex((user => user.id === id));
    USERS.splice(userIndex, 1);
  }

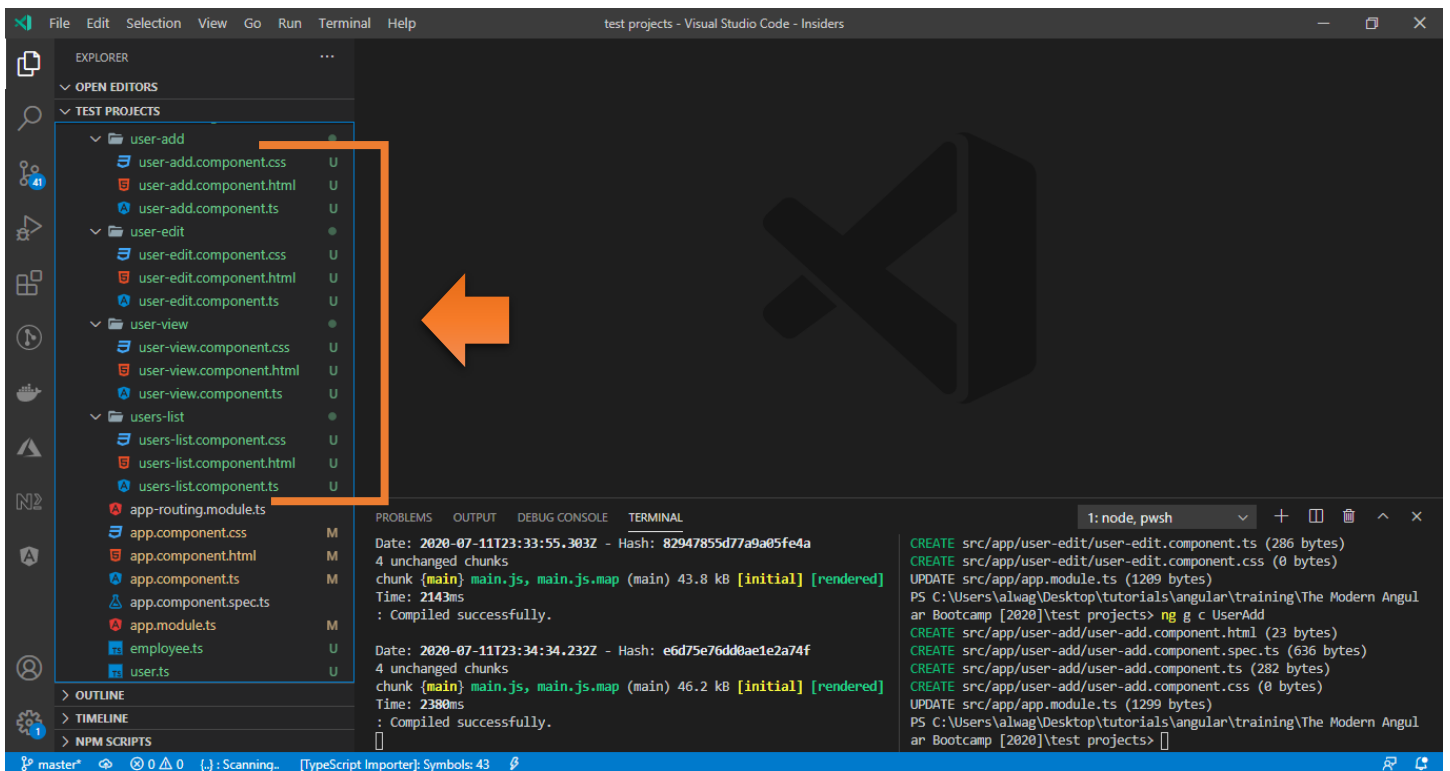
  usersCount(): number {
    return USERS.length;
  }
}

```

بعدما كتبنا جميع Functionality الخاصة بالتعامل مع البيانات، يجب ملاحظة أننا هنا لسنا بصدد شرح كيفية بناء Logic لتواصل مع قاعدة البيانات، وإنما كيف نضع جميع هذا Logic (مهما كان) في Service وكيف نستفيد من هذه Service في Components المختلفة.

لنقم الآن بإنشاء مجموعة من components واحد لعرض قائمة بالموظفين، وواحد لعرض بيانات موظف محدد وواحد لتعديل بيانات الموظفين، ومن ثم نقوم بتهيئة Routing للانتقال بين هذه components (لفهم أعمق لكيفية التعامل مع Routing الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular Routing and Modules)، كالتالي:





الآن لنقم بتهيئة Routing، كالتالي:

```

app-routing.module.ts ملف
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { UsersListComponent } from './users-list/users-list.component';
import { UserEditComponent } from './user-edit/user-edit.component';
import { UserAddComponent } from './user-add/user-add.component';
import { UserViewComponent } from './user-view/user-view.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [
  { path: 'list', component: UsersListComponent },
  { path: 'edit/:id', component: UserEditComponent },
  { path: 'add/:id', component: UserAddComponent },
  { path: 'view/:id', component: UserViewComponent },
  { path: '', redirectTo: 'list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

اما الآن لنذهب إلى ملف app.component.html، ونتأكد أن التاغ router-outlet موجود، كالتالي:

```

app.component.html ملف
<router-outlet></router-outlet>

```

ملاحظة مهمة: المفاهيم السابقة الخاصة بالتواصل بين الأب والأبن والعكس أو الاخوة، لا نستطيع تطبيقها هنا لأننا استخدمنا router-outlet، مع العلم انه لايزال Root Component يعتبر بمثابة الأب أو الحاوية لباقي components الأخرى، لكن لو نلاحظ اننا لم نضع أي selector على شكل تاغ HTML لكي يصبح ابن ونستطيع تطبيق المفاهيم التي تعلمناها عليه، وهذا النوع <router-outlet></router-outlet> هو طريقة لعرض components بشكل ديناميكي، بحيث في كل مرة يتم فيها التوجيه إلى أحد هذه components فإن Angular سوف يقوم بتحميل أي components سابق ويضع مكانه component الجديد وهكذا في كل مرة.

الآن لنقم بتصميم أول component وهو الخاص بعرض بيانات المستخدمين على شكل قائمة، وأول خطوة نقوم بها هي ان نقوم بعمل inject لي service السابقة في ملف class وبالتحديد في constructor الخاص بهذا class، كالتالي:

```
ملف users-list.component.ts
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css']
})
export class UsersListComponent implements OnInit {

  constructor(private apiService: ApiService) { }

  ngOnInit(): void {
  }

}
```

نلاحظ اننا قمنا بعمل نسخة او instance من هذه service، ونظام Dependency Injection الخاص بالAngular هو الذي قام بأغلب الامر عنا، وقدم لنا نسخة جاهزة نستطيع عن طريقها الوصول إلى جميع Functionality الخاصة بهذه service. الآن لنقم بتعريف متغير لكي لنخزن فيه جميع بيانات المستخدمين، كالتالي:

```
ملف users-list.component.ts
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css']
})
export class UsersListComponent implements OnInit {
  users: User[] = [];

  constructor() { }
  ngOnInit(): void { }
}
```

بعدها قمنا بتهيئة المتغير (الخاصية) لنقم الآن بحقن هذه Service في هذا Component لكي نستطيع الوصول إلى جميع Functionality في هذه service، وبنفس الوقت لورجعنا إلى الجزء الخاص lifecycle hook قلنا إذا كان هنالك بيانات قادمة من خارج component و template الخاص بهذا component يعتمد على هذه البيانات فلا بد ان نضعها في دالة ngOnInit، كالتالي:

ملف users-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css']
})

export class UsersListComponent implements OnInit {

  users: User[] = [];

  constructor(private apiService: ApiService) { } ❶

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers(); ❷
  }

}
```

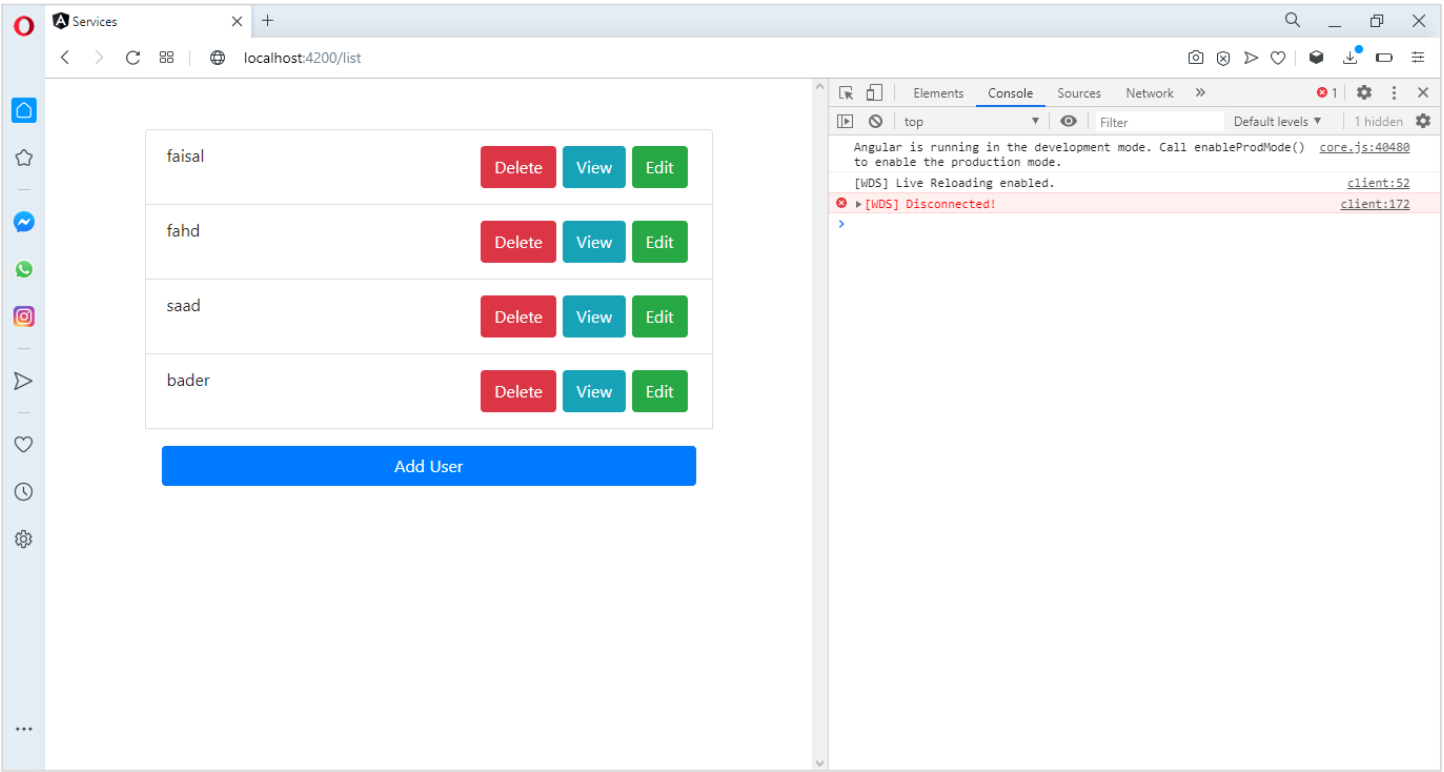
في البداية حقنا service في هذا component عن طريق constructor الخاص به، على شكل متغير اسميته apiService وبذلك أصبح هذا المتغير يمتلك حق الوصول لجميع Functionality في service ذات الاسم ApiService. اما ثاني خطوة فهي استدعينا الدالة getAllUsers والقيمة الراجعة من هذه الدالة قمنا بتخزينها في المتغير users، الآن سوف نعمل bind لهذا المتغير في ملف template ونعمل له Loop لكي نعرض جميع أسماء users، كالتالي:

ملف users-list.component.html

```
<div class="container">
  <div class="row justify-content-center pt-5">
    <ul class="list-group w-100">
      <li class="list-group-item" *ngFor="let user of users">
        {{ user.name }}
        <button
          class="btn btn-success float-right button"
          (click)="editUser(user.id)"
        >
          Edit
        </button>
        <button
          class="btn btn-info float-right button"
          (click)="viewUser(user.id)"
        >
```

```
>
View
</button>
<button
  class="btn btn-danger float-right button"
  (click)="deleteUser(user.id)"
>
Delete
</button>
</li>
<button class="btn btn-large btn-primary m-3" [routerLink]="['../add']">
  Add User
</button>
</ul>
</div>
</div>
```

استفدنا من مكتبة bootstrap لكي نعمل تجميع وتنسيق للواجهة في ملف template، ونلاحظ اننا عملنا Loop لي users وبنفس الوقت قمنا بعمل Interpolation Binding لأسم كل user، واضفنا مجموعة من الازرار وكل زر ينفذ دالة معينة فالزر الخاص بالتعديل يُنفذ الدالة editUser ومررنا له id للمستخدم والقادم لنا من المتغير (الكائن) users مع كل Loop، وهكذا بقية الأزرار، طبعاً نحن إلى الآن لم نكتب محتويات هذه الدوال إلى الآن وسوف نكتبها بعد قليل، وأخيراً استفدنا من تقنيات Angular Router للانتقال إلى component الخاص بإضافة مستخدم جديد في حال الضغط على الزر Add، الآن لنرى النتيجة في المتصفح، كالتالي:



نلاحظ ظهرت لنا القائمة وجميع الأزرار، ولكن هذه الأزرار إلى الآن لم تفعل لأننا لم نكتب محتويات الدوال الخاصة بها، ما عدا زر Add User يقوم بتحويلنا فقط إلى component الخاص بإضافة مستخدم جديد والذي هو أيضاً لم نقوم بتهيئته إلى الآن.

## ملف users-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css']
})
export class UsersListComponent implements OnInit {

  users: User[] = [];

  constructor(
    private apiService: ApiService,
    private router: Router
  ) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
  }

  editUser(id: number) {
    this.router.navigate(['./edit', id]);
  }

  viewUser(id: number) {
    this.router.navigate(['./view', id]);
  }

  deleteUser(id: number) {
    this.apiService.deleteUser(id);
  }
}
```

اعلم عزيزي المتعلم انه ليس عليك فهم ما كتبناه هنا لأن هذا يتعلق بي Angular Routing والتقنيات المرتبطة به وفي حال اردت ان تستزيد في هذه الجزئية الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular Routing and Modules، ولكن سوف أقوم بشرح مبسط لكي لا يحدث أي تشويش لديك، فالذي قمنا به هو اننا قمنا بعمل inject لي service جاهزة لكي نستفيد Functionality الموجودة بها، ومنها الدالة navigate التي تتيح لنا الانتقال إلى مسار محدد قمنا بتهيئته سابقاً في ملف app-routing.module.ts وربطناه هذه المسارات paths مع components المناسبة لها، اما id فهو عبارة عن باراميتر نرسله عن طريق الرابط لي component الذي نريد الانتقال إليه، وهو يحمل القيمة التي مررناها إلى الدالة في ملف template سابقاً .user.id

أما الدالة deleteUser فتقوم باستدعاء الدالة التي تحمل نفس الاسم والتي أنشأناها سابقاً في service ومررنا لها id لي user. الآن لنقم بتجهيز وتهيئة component الخاص بإضافة مستخدم جديد وكتابة جميع Functionality و Logic الخاص به، كالتالي:

ملف user-add.component.html

```
<div class="container">
  <div class="row justify-content-center pt-5">
    <div class="card w-100">
      <h5 class="card-header">Create New User</h5>
      <div class="card-body">
        <form [formGroup]="form">
          <div class="form-group">
            <label for="name">Name</label>
            <input
              type="text"
              class="form-control"
              formControlName="name"
              id="name"
            />
          </div>
          <div class="form-group">
            <label for="city">city</label>
            <input
              type="text"
              class="form-control"
              formControlName="city"
              id="city"
            />
          </div>
          <div class="form-group">
            <label for="phone">Phone</label>
            <input
              type="text"
              class="form-control"
              formControlName="phone"
              id="phone"
            />
          </div>
          <div class="form-group">
            <label for="userName">userName</label>
            <input
              type="text"
              class="form-control"
              formControlName="userName"
              id="userName"
            />
          </div>
          <div class="form-group">
            <label for="password">Password</label>
            <input
              type="password"
```

```

        class="form-control"
        formControlName="password"
        id="password"
    />
</div>
</form>
<div class="card-footer">
    <button class="btn btn-primary m-2" (click)="save()">
        Save
    </button>
    <a [routerLink]="['/']" class="btn btn-success m-2">Go Back</a>
</div>
</div>
</div>
</div>
</div>

```

#### ملف user-add.component.ts

```

import { Component, OnInit } from '@angular/core';
import { User } from '../user';
import { ApiService } from '../core/api.service';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-user-add',
  templateUrl: './user-add.component.html',
  styleUrls: ['./user-add.component.css']
})
export class UserAddComponent implements OnInit {
  form: FormGroup;
  constructor(
    private apiService: ApiService,
    private formBuilder: FormBuilder
  ) { }

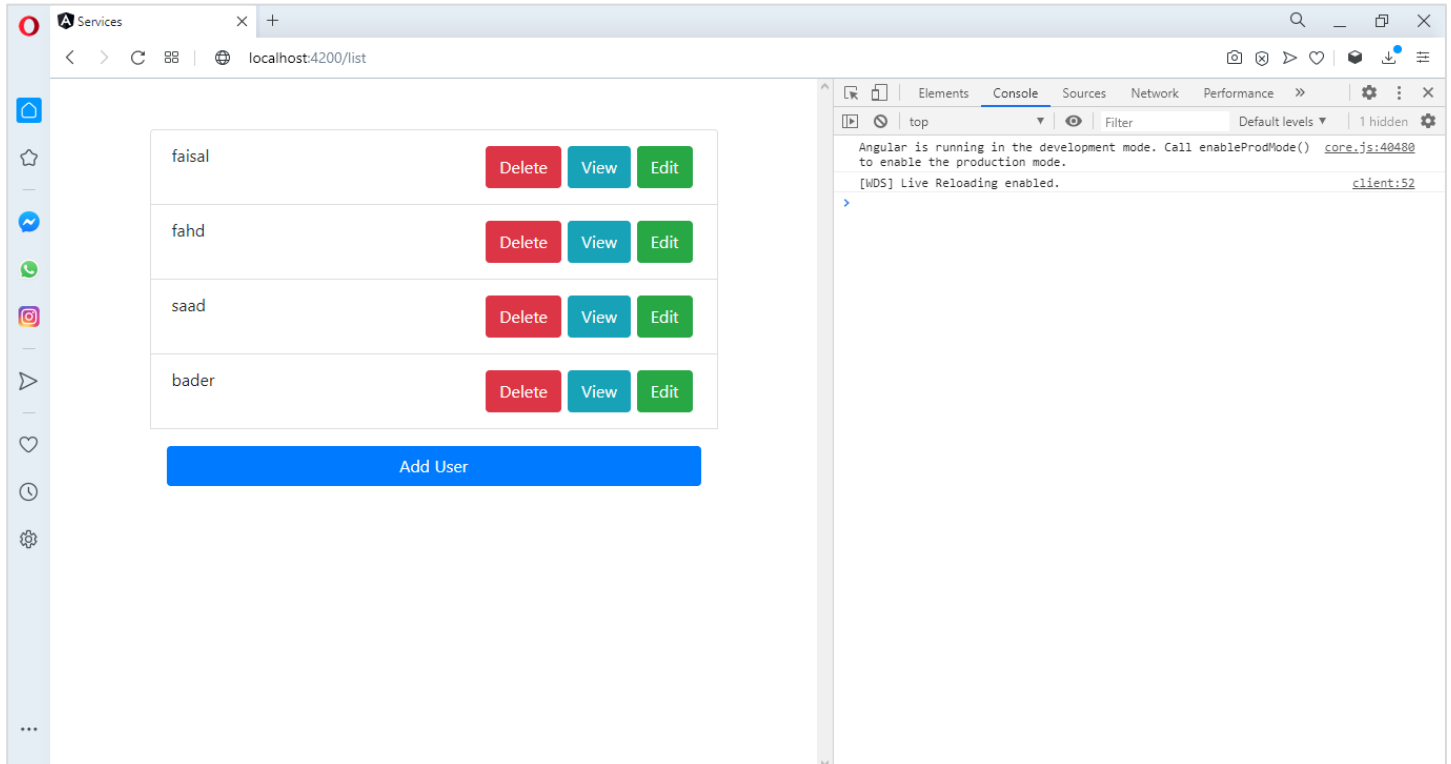
  ngOnInit(): void {
    this.form = this.formBuilder.group({
      name: [null],
      city: [null],
      phone: [null],
      userName: [null],
      password: [null]
    });
  }

  save() {
    const user: User = this.form.value;
    user.id = this.apiService.usersCount() + 1;
    this.apiService.createUser(user);
    alert('A user has been created Successfully');
  }
}

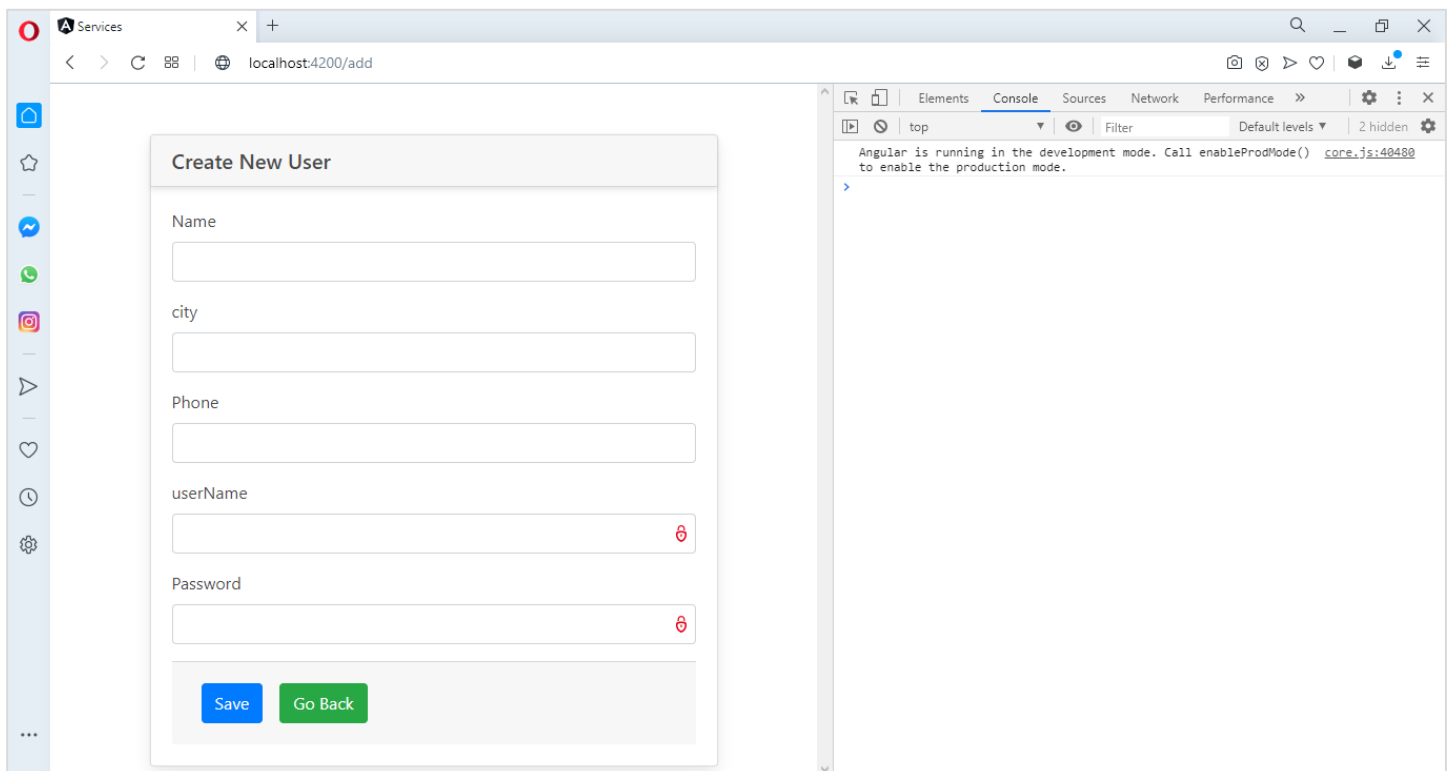
```



نلاحظ أننا أضفنا Angular Reactive Forms (ولفهم أعمق الرجاء مراجعة الكتاب الخامس من هذه السلسلة Angular Forms)، أما الآن لنذهب إلى المتصفح ونرى النتيجة، كالتالي:

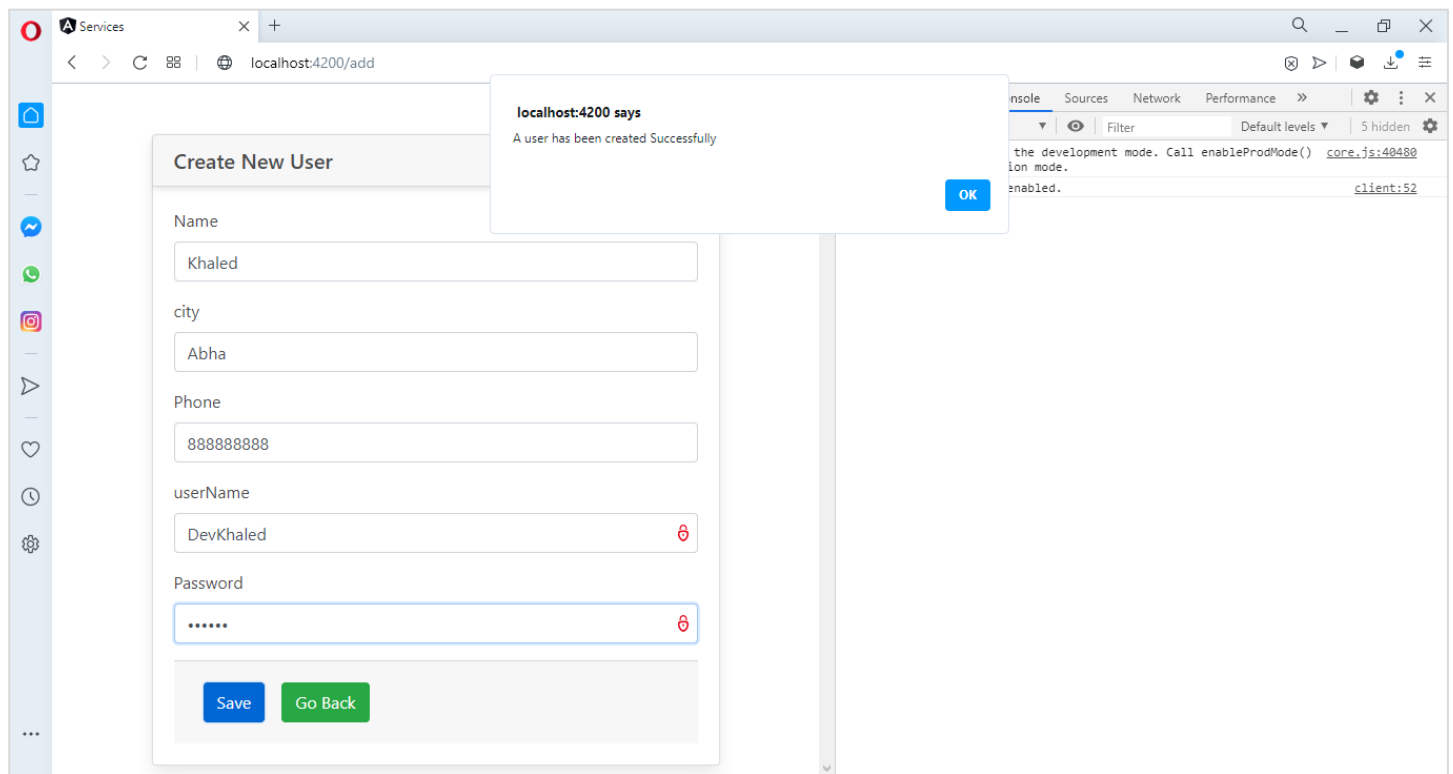


لنقم بالضغط على زر Add User، كالتالي:

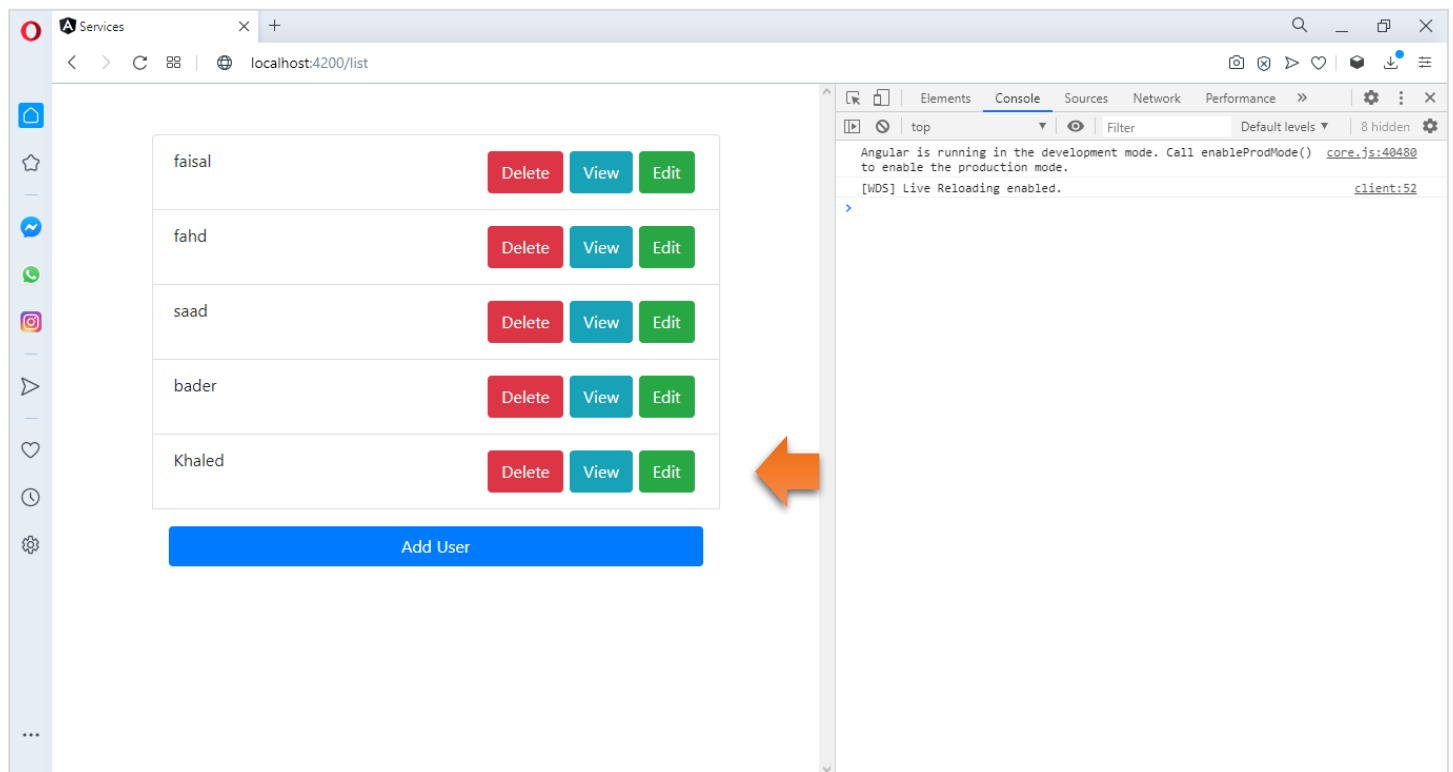


لنقم بتعبئة البيانات ونضغط على زر save، حيث ان هذا الزر هو الذي يحتوي على الدالة التي تتصل مع service، كالتالي:





بعدها لنضغط زر Go Back لرجوع إلى القائمة الرئيسية، كالتالي:



نلاحظ أضاف المستخدم الجديد إلى القائمة، الآن لنقوم بتهيئة component الخاص بعرض بيانات المستخدمين view، كالتالي:

#### ملف user-view.component.html

```
<div class="container">
  <div class="row justify-content-center pt-5">
    <div class="card">
      <h5 class="card-header">User Details</h5>
      
      <div class="card-body">
        <h5 class="card-title">{{ user.name }}</h5>
        <p class="card-text">city: {{ user.city }}</p>
        <p class="card-text">Phone: {{ user.phone }}</p>
        <p class="card-text">userName: {{ user.userName }}</p>
        <p class="card-text">Password: {{ user.password }}</p>
        <a [routerLink]="['/list']" class="btn btn-primary">Go Back</a>
      </div>
    </div>
  </div>
</div>
```

#### ملف user-view.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { ApiService } from '../core/api.service';
import { User } from '../user';

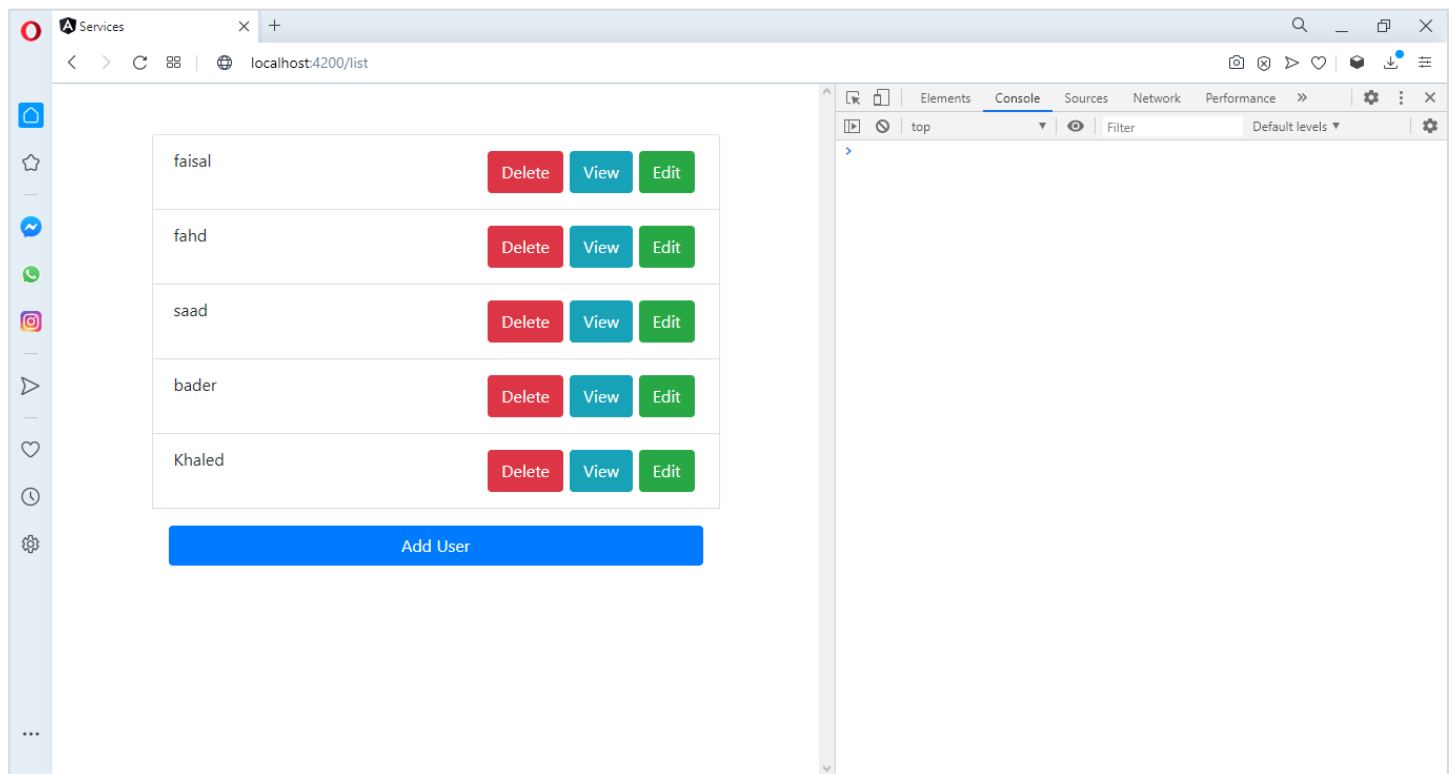
@Component({
  selector: 'app-user-view',
  templateUrl: './user-view.component.html',
  styleUrls: ['./user-view.component.css']
})
export class UserViewComponent implements OnInit {

  user: User;

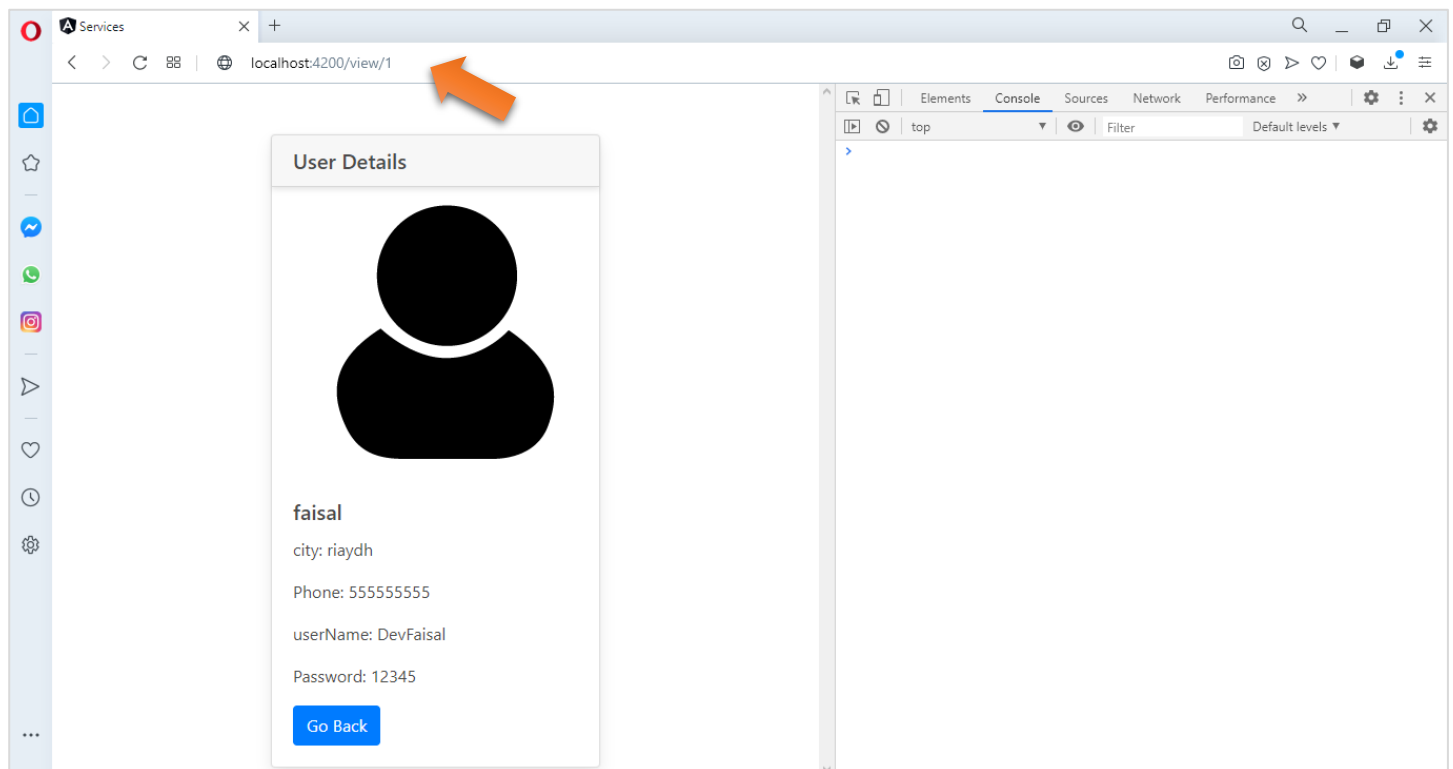
  constructor(
    private activatedRoute: ActivatedRoute,
    private apiService: ApiService
  ) { }

  ngOnInit(): void {
    const id = +this.activatedRoute.snapshot.params.id;
    this.user = this.apiService.getUserById(id);
  }
}
```

لعل الشيء الجديد هنا هو اننا قمنا باستقبال id الذي ارسلناه عن طريق الرابط سابقاً بعد تحويله إلى رقم عن طريق (+) ومن ثم مررنا هذا id إلى الدالة التي قمنا ببنائها سابقاً في service والتي تُرجع لنا user محدد بناءً على id الخاص به، الآن لنحفظ التعديلات ولنذهب إلى المتصفح ونرى النتيجة، كالتالي:



عند بداية التشغيل الآن لنختار أحد المستخدمين ونضغط على زر View الخاص به، كالتالي:



نلاحظ قام بقراءة id المرسل له في الرابط وبناءً عليه قام بالاتصال service وإظهار بيانات المستخدم المحدد.

وأخر جزء لنقم بتهيئة component الخاص بإجراء التعديلات على المستخدمين، كالتالي:

ملف user-edit.component.html

```
<div class="container">
  <div class="row justify-content-center pt-5">
    <div class="card w-100">
```

```

<h5 class="card-header">Edit User</h5>
<div class="card-body">
  <form #form="ngForm">
    <div class="form-group">
      <label for="name">Name</label>
      <input
        type="text"
        class="form-control"
        name="name"
        [(ngModel)]="user.name"
        id="name"
        [value]="user.name"
      />
    </div>
    <div class="form-group">
      <label for="city">city</label>
      <input
        type="text"
        class="form-control"
        name="city"
        [(ngModel)]="user.city"
        id="city"
        [value]="user.city"
      />
    </div>
    <div class="form-group">
      <label for="phone">Phone</label>
      <input
        type="text"
        class="form-control"
        name="phone"
        [(ngModel)]="user.phone"
        id="phone"
        [value]="user.phone"
      />
    </div>
    <div class="form-group">
      <label for="userName">userName</label>
      <input
        type="text"
        class="form-control"
        name="userName"
        [(ngModel)]="user.userName"
        id="userName"
        [value]="user.userName"
      />
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input
        type="password"
        class="form-control"
        name="password"

```

```

        [(ngModel)]="user.password"
        id="password"
        [value]="user.password"
    />
</div>
</form>
<div class="card-footer">
    <button class="btn btn-primary m-2" (click)="save(form)">Save</button>
    <a [routerLink]="['/list']" class="btn btn-success m-2">Go Back</a>
</div>
</div>
</div>
</div>
</div>

```

وهنا استخدمنا Template Driven Forms بعكس المرة السابقة التي استخدمنا بها Reactive Forms، وايضاً تم شرح هذه التقنية في كتاب Angular Forms بالتفصيل، أما ملف class فهو كالتالي:

#### ملف user-view.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';
import { ActivatedRoute } from '@angular/router';
import { User } from '../user';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-user-edit',
  templateUrl: './user-edit.component.html',
  styleUrls: ['./user-edit.component.css']
})
export class UserEditComponent implements OnInit {

  user: User;

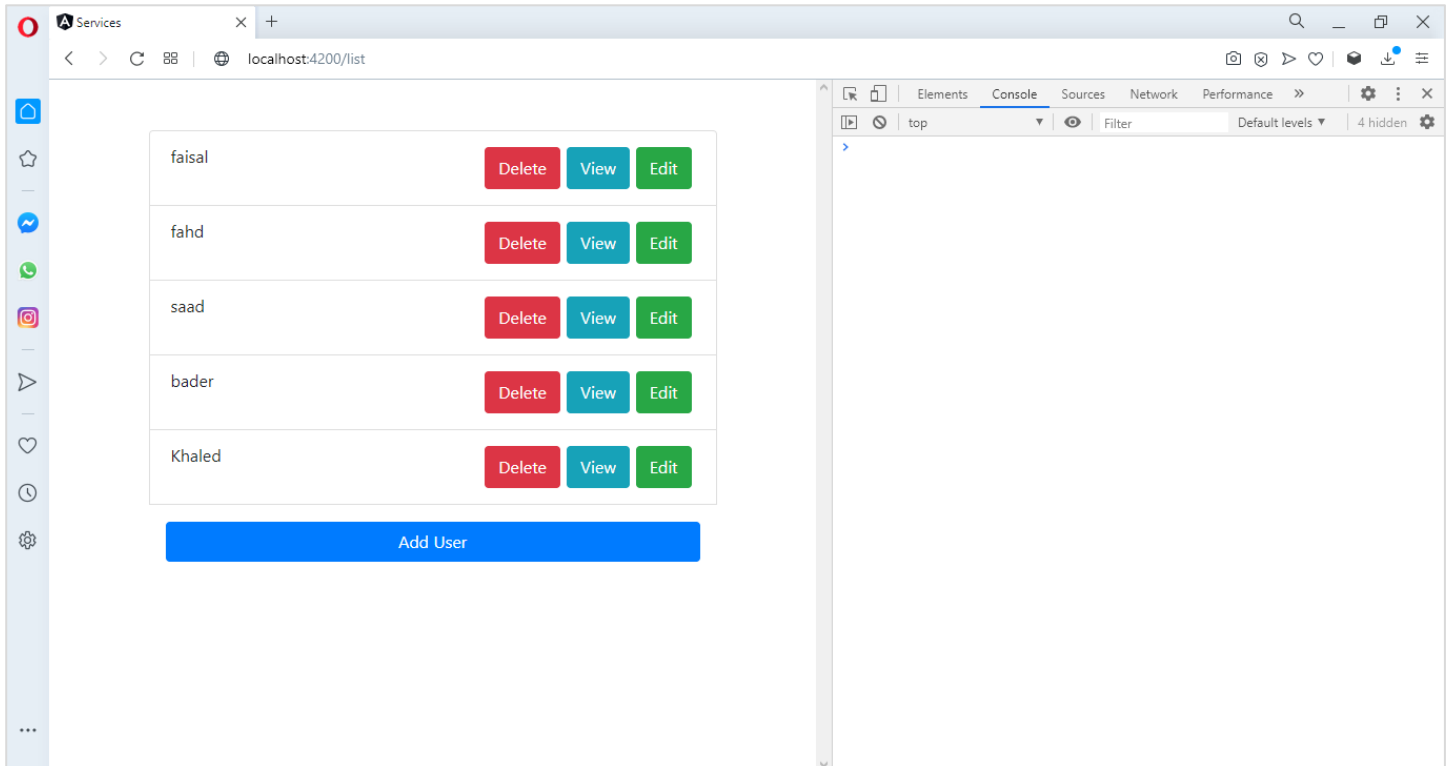
  constructor(
    private activatedRoute: ActivatedRoute,
    private apiService: ApiService
  ) { }

  ngOnInit(): void {
    const id = +this.activatedRoute.snapshot.params.id;
    this.user = this.apiService.getUserById(id);
  }

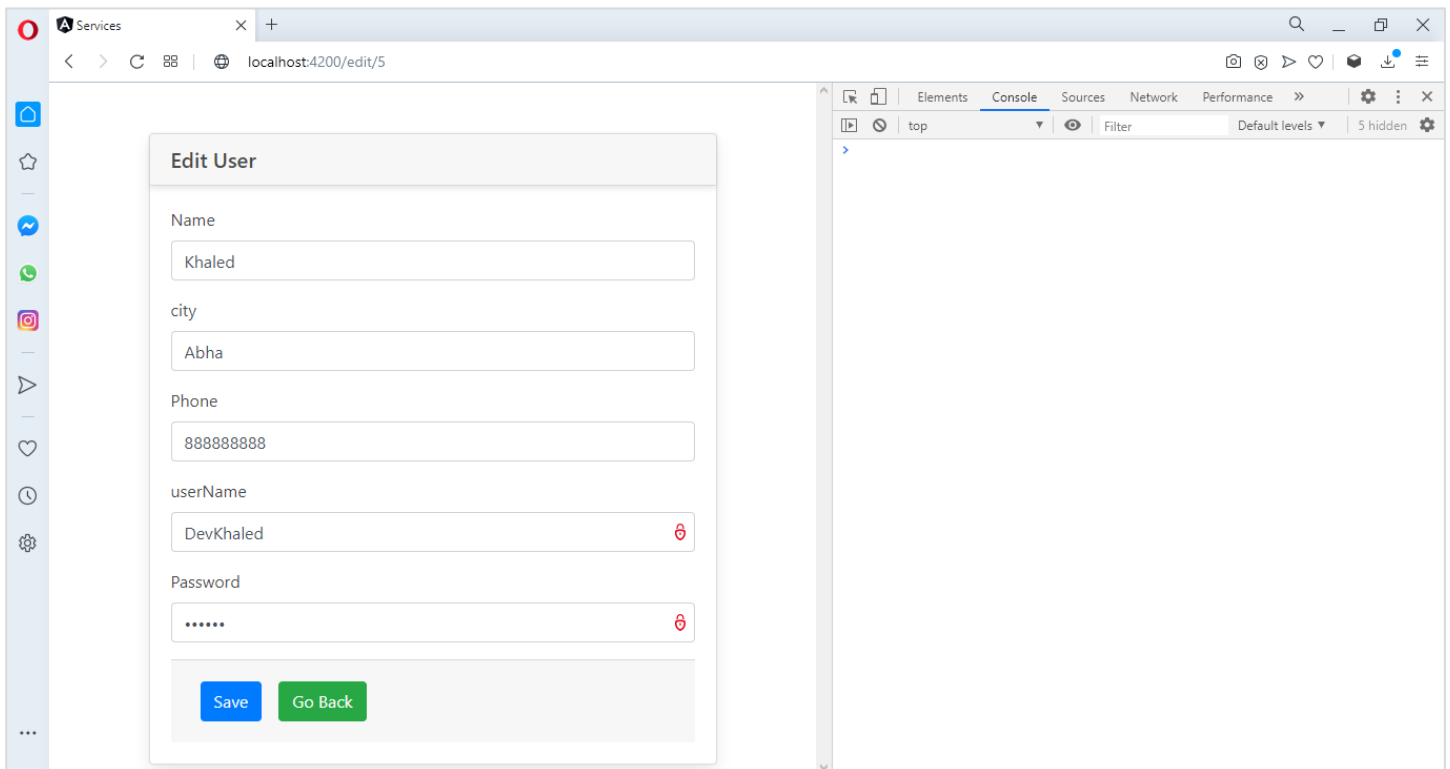
  save(form: NgForm) {
    form.value.id = this.user.id;
    this.apiService.editUser(form.value);
    alert('changes saved !!');
  }
}

```

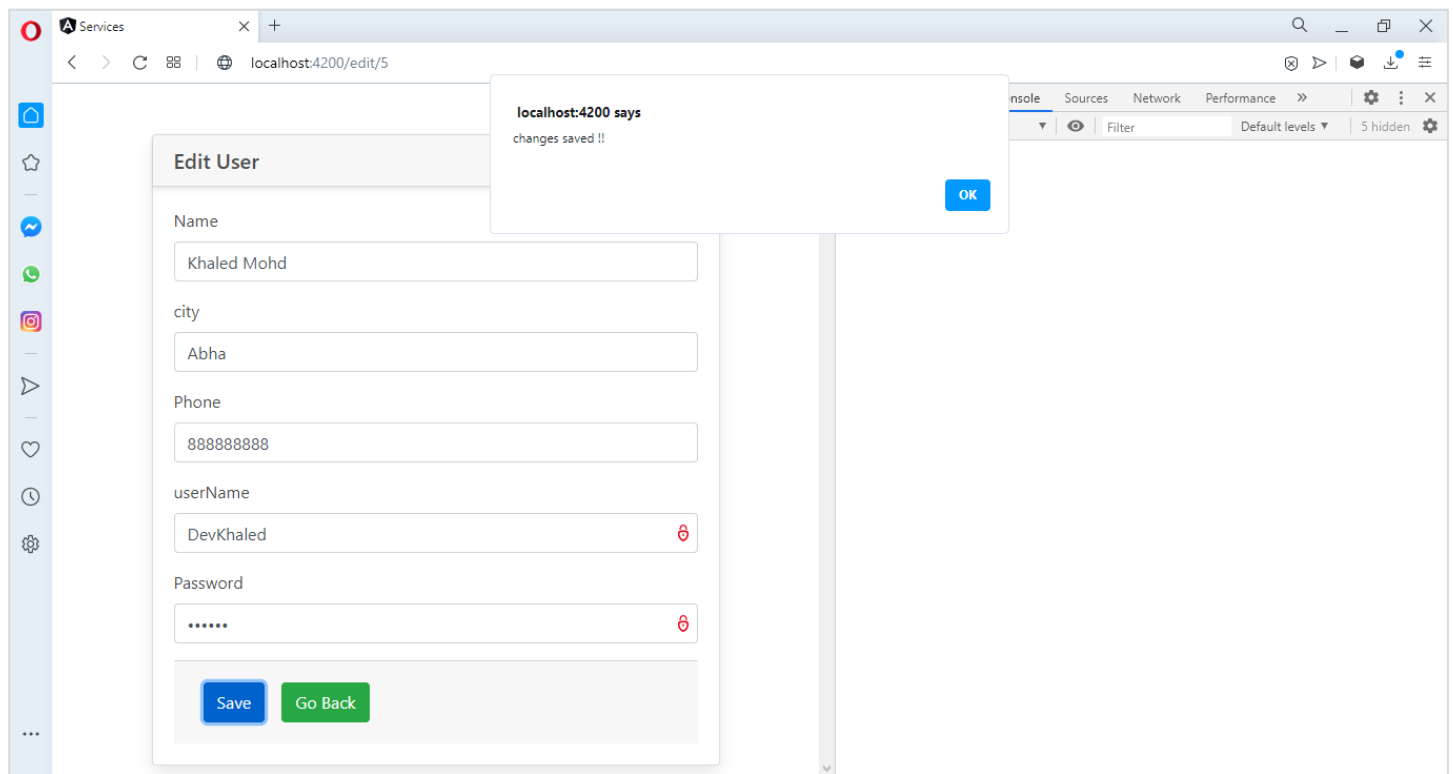
ولنشاهد النتيجة في المتصفح، كالتالي:



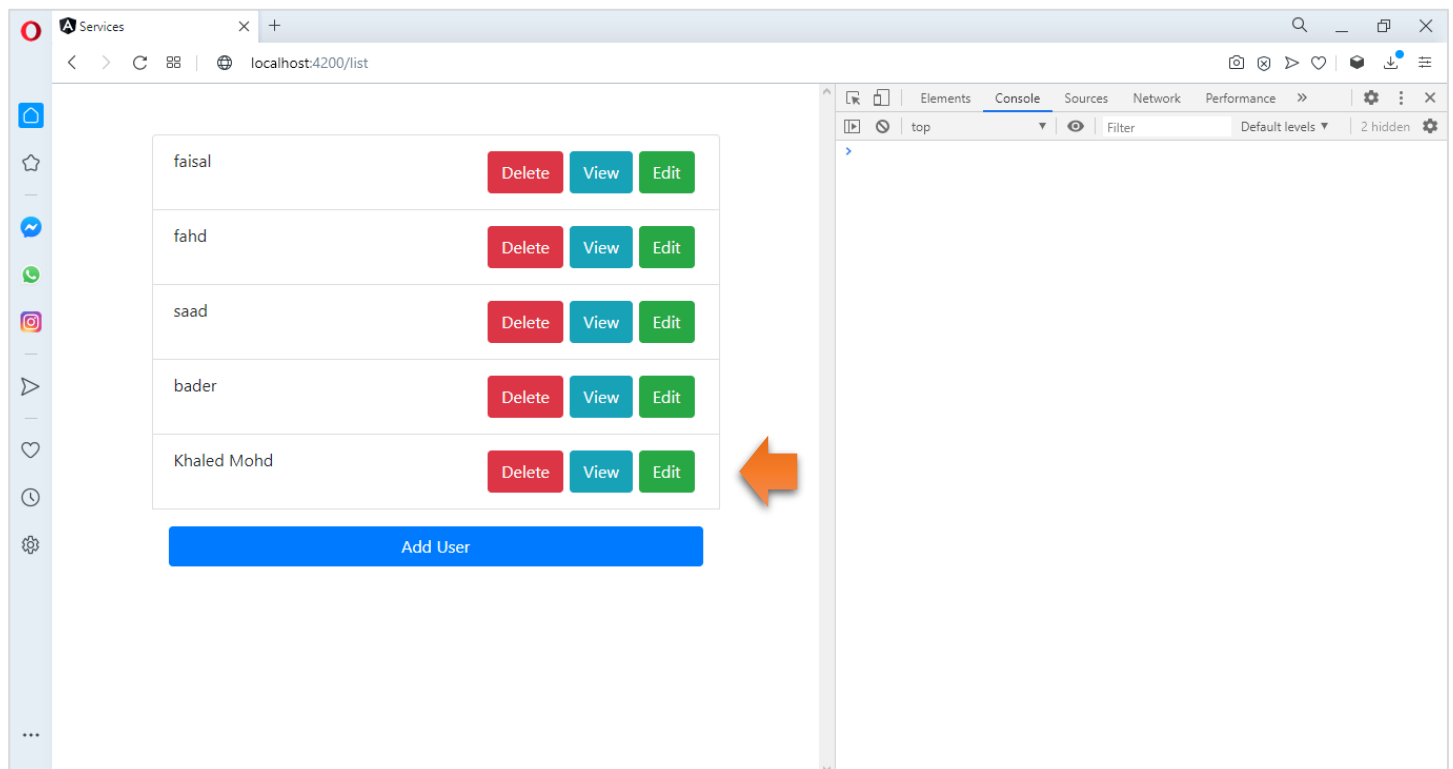
لنختار أحد المستخدمين، وليكن Khaled ونضغط على زر Edit الخاص به، كالتالي:



ظهرت لنا بيانات المستخدم خالد في وضع التعديل، لنقم بتغيير بياناته وليكن التغيير على اسمه، ونضغط بعدها زر save، كالتالي:



الآن لنضغط على زر Go Back، لنذهب إلى القائمة الرئيسية، كالتالي:



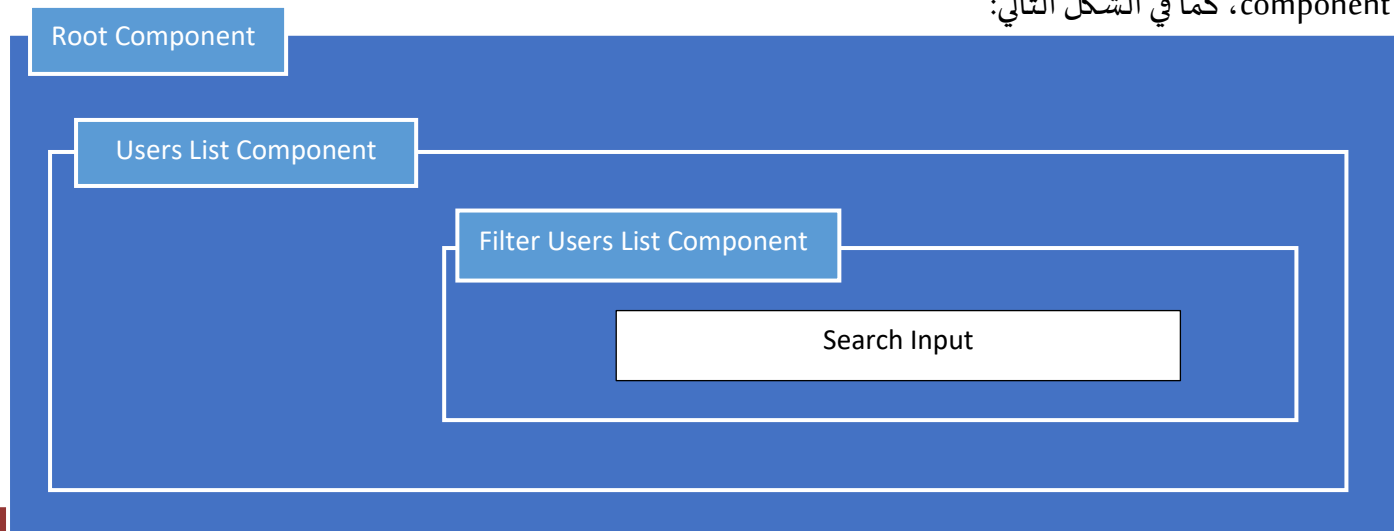
وبذلك تعلمنا كيف نقوم بفصل جميع Logic الذي لا يتعامل بشكل مباشر ولا يؤثر على template في service منفصل، ونستدعي هذه service في أي component يحتاج إلى هذا logic، وبنفس الوقت توضح لنا أهمية هذه service، فبدلاً من إعادة كتابة جميع Logic في كل component، فقط قمنا بكتابته في service ومن ثم استدعينا هذه service في أي component يحتاج إلى Functionality التي تحتويها هذه services.

وحقيقة وجدت في بعض المراجع من يعنون Asynchronous Services وهي نفس التي ذكرناها قبل قليل ولكن الدوال التي تحتويها لتعامل مع البيانات تكون Asynchronous أي غير تزامنية وتعتمد على تقنيات Observable ومكتبتها Rxjs أو Promises، وأنا لا انكر أهمية هذه التقنيات فهي بالغالب هي المستخدمة في التطبيقات الواقعية، ولكن ارتأيت عدم شرحها هنا لكي لا اشتت ذهن المتعلم في دهاليز هذه التقنيات وهي تحتاج إلى كتاب أو عدة كتب مستقلة لشرحها، وبذلك نكون ابتعدنا عن الهدف الأساس وهو services، وكما قلت انني في حال اردت شرح هذه الجزئية فلن اضيف شيء جديد من ناحية service فالاستخدام هو نفسه، وانما الفرق في طريقة بناء Functionality، لذلك سوف اشرح في النقطة التالية كيفية الاستفادة من services في التواصل بين component ونفسه، وبنفس الوقت سوف نطبق مجموعة من المفاهيم التي تعلمناها سابقاً.

### 6.3. استخدام Services لتواصل مع component ونفسه:

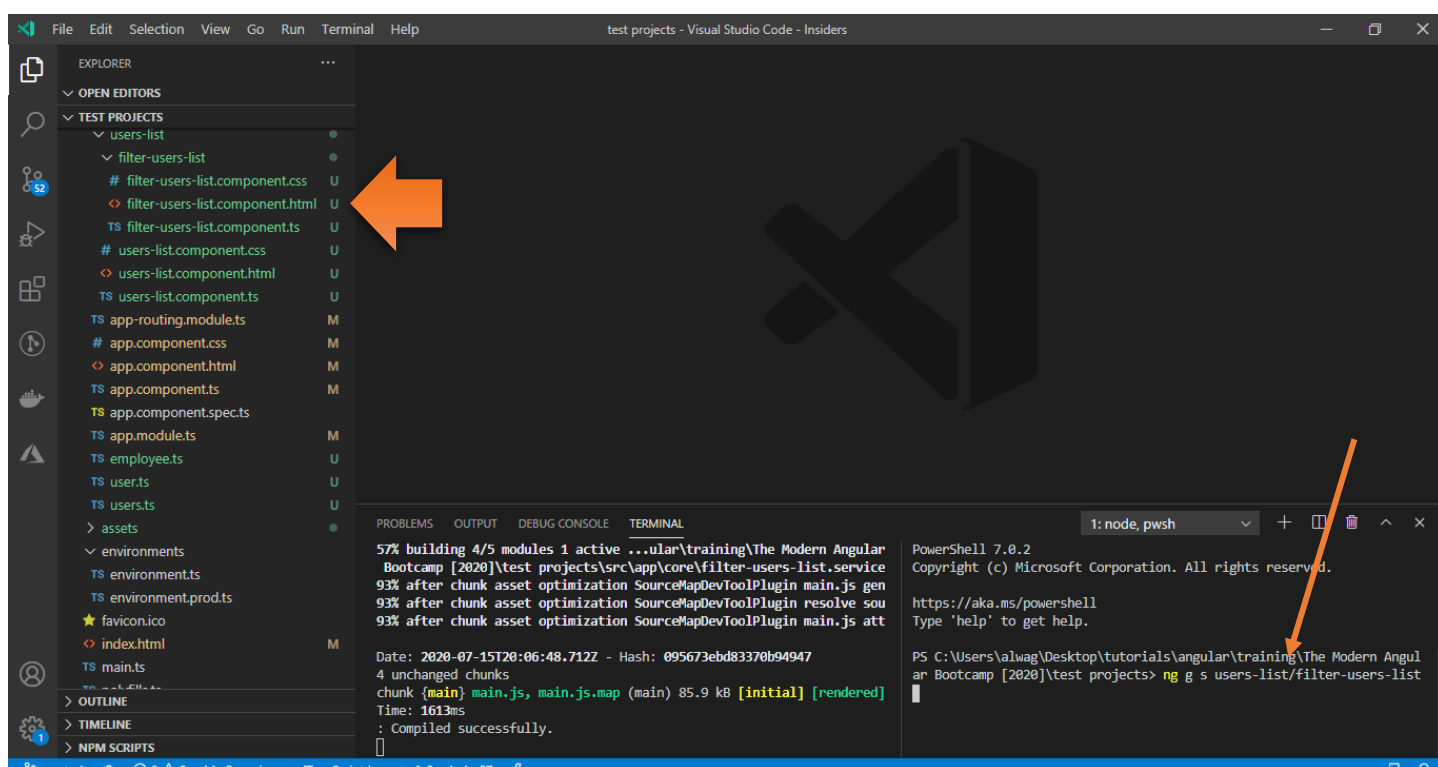
وهنا نقصد ان يكون لدينا Service يحتوي على متغير (خاصية) نخزن فيها قيمة خاصة بهذا component فقط بحيث عند الانتقال من هذا component والرجوع إليه مرة أخرى تبقى هذه القيمة ونستطيع الاستفادة منها في هذا component، ولتوضيح لنعطي مثال، لو افترضنا انه لدينا مربع نص للبحث عن أسماء المستخدمين، بحيث يقوم بعمل filter للقائمة بناءً على المدخل في مربع البحث، وعند اختيار أي اسم لعرضه فإن Angular سوف يقوم بعمل Destroy لي component الخاص بعرض قائمة المستخدمين ويقوم ببناء component الخاص بعرض بيانات المستخدم، وعند الرجوع فإنه يقوم بالعكس حيث يقوم بتحطيم هذا component الأخير وإعادة بناء component الذي يحتوي على قائمة المستخدمين، وفي هذه الحالة سوف تعود القائمة كما هي وجميع ما تم كتابته في مربع البحث سوف يختفي لأننا قمنا بتحطيم هذا component وإعادة بناءه، ومن هنا أتت فكرة ان component يحتاج ان يتواصل مع نفسه، بحيث نحتاج ان نقوم بتخزين القيمة الموجودة في مربع البحث عند تحطيم هذا component في مكان ما في التطبيق ومن ثم استدعاء هذه القيمة عند إعادة بناء هذا component وبناءً عليها نُعيد عمل filter لهذا القائمة بحسب القيمة الموجودة، وبطبيعة الحال ليس هنالك مكان افضل من services لتخزين هذه القيمة، مع العلم انه يمكن حلها باستخدام طرق أخرى منها على سبيل المثال استخدام Routing وتخزين القيمة في الرابط، ولكن بما اننا نتكلم هنا عن service فمن الأفضل ان نستخدم هذه الطريقة.

وبطبيعة الحال نستطيع ان نضع مربع البحث في component الخاص بعرض القائمة، ولكنني بالحقيقة اريد ان نتدرب ونراجع بعض ما تعلمناه سابقاً لذلك سوف أقوم بإنشاء component منفصل يحتوي على مربع النص الخاص بالبحث بحيث يكون هو الأبني لي component الخاص بعرض قائمة المستخدمين، وليكن اسم component الجديد filter users list component، كما في الشكل التالي:





الآن لنذهب إلى terminal ونقوم بإنشاء component جديد باسم filter users list وليكن داخل مجلد component الأب وهو users-list، ويتم ذلك عن طريق كتابة الأمر التالي ng g c users-list/filter-users-list، كالتالي:



الآن لنقوم بتصميم مربع البحث في ملف template لهذا component الجديد، كالتالي:

ملف filter-users-list.component.html

```
<div class="input-group mt-2 mb-2">
  <div class="input-group-prepend">
    <span class="input-group-text">Search</span>
  </div>
  <input type="text" class="form-control" />
</div>
```

الآن لنضيف selector الخاص بهذا component الأب إلى على شكل تاغ HTML في component الأب، كالتالي:

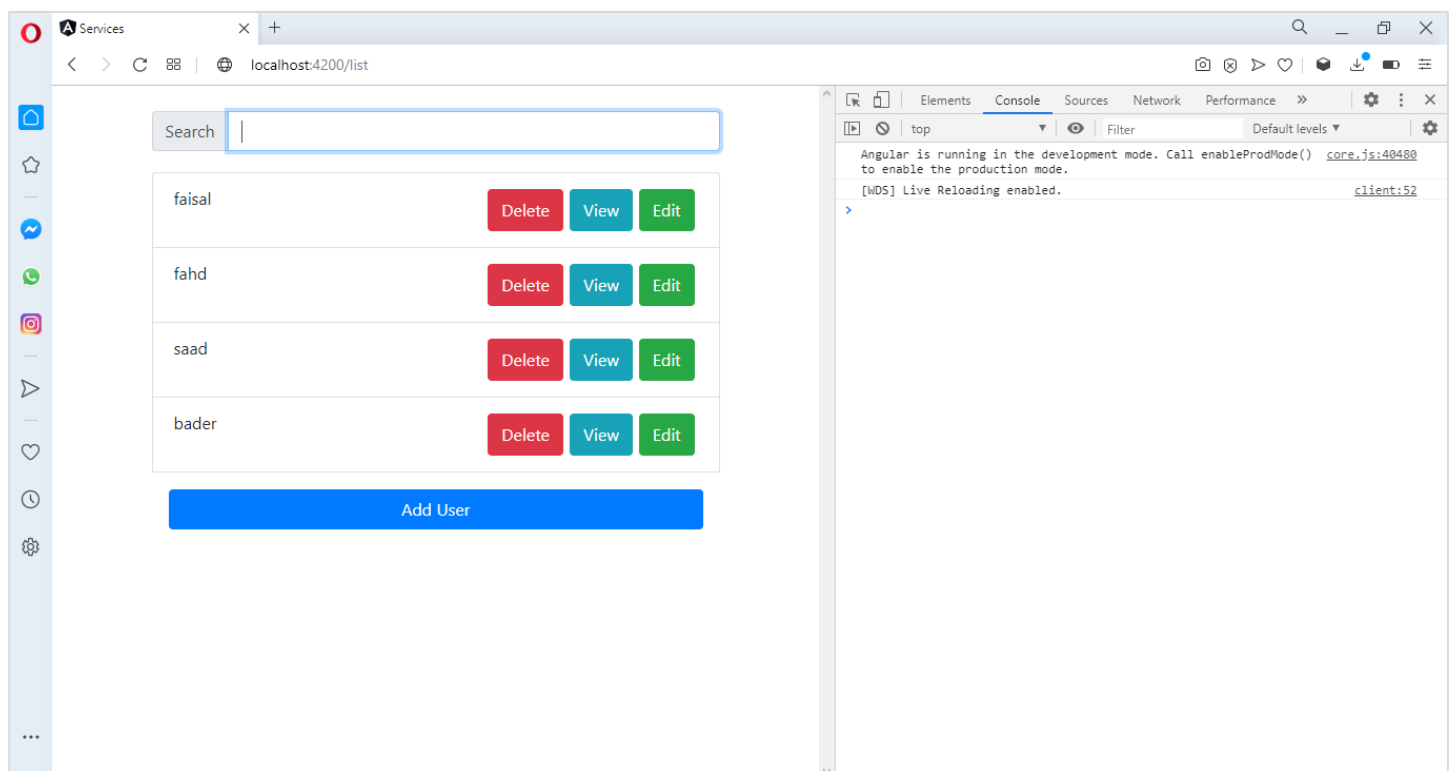
```
<div class="container">
  <div class="row justify-content-center pt-2">
    <app-filter-users-list></app-filter-users-list>
    <ul class="list-group w-100">
      <li class="list-group-item" *ngFor="let user of users">
        {{ user.name }}
        <button
          class="btn btn-success float-right button"
          (click)="editUser(user.id)"
        >
          Edit
        </button>
        <button
          class="btn btn-info float-right button"
          (click)="viewUser(user.id)"
```

```

    >
    View
  </button>
  <button
    class="btn btn-danger float-right button"
    (click)="deleteUser(user.id)"
  >
    Delete
  </button>
</li>
<button
  class="btn btn-large btn-primary m-3"
  [routerLink]="['../add']"
>
  Add User
</button>
</ul>
</div>
</div>

```

الآن لنحفظ التعديلات ونرى النتيجة في المتصفح، كالتالي:



نلاحظ ان النتيجة كما هو متوقع، والآن لنقوم بمراجعة بعض المفاهيم السابقة، ونقوم بتطبيقها هنا، فالذي نريده هو متغير يراقب أي تغييرات حدثت في Input ويقوم بتخزينها، وأفضل حل هو Two-Way-Data-Binding لذلك نقوم بتعريف متغير في ملف class لهذا component وليكن اسمه filterList، كالتالي:

```

ملف filter-users-list.component.ts
import { Component, OnInit } from '@angular/core';

@Component({

```

```

selector: 'app-filter-users-list',
templateUrl: './filter-users-list.component.html',
styleUrls: ['./filter-users-list.component.css']
})

export class FilterUsersListComponent implements OnInit {
  filterList: string;
  constructor() { }
  ngOnInit(): void {
  }
}

```

اما الخطوة الثانية هي ربط هذه المتغير عن طريق مفهوم Two-Way-Data-Binding في ملف template، كالتالي:

ملف filter-users-list.component.html

```

<div class="input-group mt-2 mb-2">
  <div class="input-group-prepend">
    <span class="input-group-text">Search</span>
  </div>
  <input type="text" class="form-control" [(ngModel)]="listFilter" />
</div>

```

المفهوم التالي الذي نريد مراجعته هو كيف أرسل هذه المتغير listFilter من هذا component الذي يعتبر الأب إلى الابن، ونستطيع عمل ذلك عن طريق استخدام @Output، وبنفس الوقت لو رجعنا إلى الجزء الذي شرحنا فيه Two-Way-Data-Binding لوجدنا اني ذكرت انه في حال اردنا ان ننفذ اسطر برمجية أخرى في حال تغير قيمة المتغير الذي عملنا له Two-Way-Data-Binding فإنه لدينا طريقتين الأولى بتقسيم ngModel إلى جزئين [ngModel] و (ngModelChange) اما الطريقة الثانية فهي عن طريق استخدام getter/setter، وايهما اخترت فلا ضير فكلاهما يؤديان نفي المهمة ولكن انا سوف استخدم الطريقة الثانية، والسبب لاستخدامي احدي هاتين الطريقتين اني اريد ان اراقب أي تغييرات على هذا المتغير وفي حال حدوث أي تغيير عليه ارسله مباشر إلى component الأب، اما طريقة الأرسال فعن طريق EventEmitter، كالتالي:

```

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

@Component({
  selector: 'app-filter-users-list',
  templateUrl: './filter-users-list.component.html',
  styleUrls: ['./filter-users-list.component.css']
})

```

```

}))

export class FilterUsersListComponent implements OnInit, OnChanges {
  @Output() valueChanges: EventEmitter<string> = new EventEmitter<string>();

  private _listFilter: string;

  get listFilter(): string {
    return this._listFilter;
  }

  set listFilter(value: string) {
    this._listFilter = value;
    this.valueChanges.emit(value);
  }

  constructor() { }

  ngOnInit(): void {}
}

```

الآن لنستقبل هذه القيمة في component الأب، كالتالي:

ملف users-list.component.html

```

<div class="container">
  <div class="row justify-content-center pt-2">
    <app-filter-users-list (valueChanges)="onChangeValue($event)">
    </app-filter-users-list>
    <ul class="list-group w-100">
      <li class="list-group-item" *ngFor="let user of users">
        {{ user.name }}
        <button
          class="btn btn-success float-right button"
          (click)="editUser(user.id)"
        >
          Edit
        </button>
        <button
          class="btn btn-info float-right button"
          (click)="viewUser(user.id)"
        >
          View
        </button>
        <button
          class="btn btn-danger float-right button"
          (click)="deleteUser(user.id)"
        >
          Delete
        </button>
      </li>
      <button
        class="btn btn-large btn-primary m-3"
        [routerLink]="['../add']"

```

```

    >
      Add User
    </button>
  </ul>
</div>
</div>

```

اما في ملف class فنقوم بكتابة محتوى الدالة onChangeValue وهو عبارة عن Logic يقوم بعمل filter للقائمة، كالتالي:

ملف users-list.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
})

export class UsersListComponent implements OnInit {
  users: User[];
  constructor(private apiService: ApiService, private router: Router) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
  }

  onChangeValue(value: string) {
    if (value) {
      const result = this.users.filter(user =>
        user.name.toLocaleLowerCase().indexOf(value.toLocaleLowerCase()) !== -1);
      this.users = result;
    } else {
      this.users = this.apiService.getAllUsers();
    }
  }

  editUser(id: number) {
    this.router.navigate(['./edit', id]);
  }

  viewUser(id: number) {
    this.router.navigate(['./view', id]);
  }

  deleteUser(id: number) {
    this.apiService.deleteUser(id);
  }
}

```



ولنقوم بمراجعة مفهوم آخر، وذلك من خلال إظهار عدد نتائج البحث ان وجد او إظهار رسالة للمستخدم في حال عدم وجود عدم تطابق، وللقيام بهذه الأمر نحتاج ان نمرر عدد عناصر المصفوفة this.users من الأب إلى component الأبن، لأن هذه المصفوفة هي التي نقوم بتخزين القيمة بعد الفلترة بها كما هو موضح في السطر this.users = result، اما طريقة الاتصال من الأب إلى الأبن فنستخدم @Input، كالتالي:

#### ملف users-list.component.html

```
<div class="container">
  <div class="row justify-content-center pt-2">
    <app-filter-users-list
      [listItemsNum]="users?.length"
      (valueChanges)="onChangeValue($event)"
    >
  </app-filter-users-list>
  <ul class="list-group w-100">
    <li class="list-group-item" *ngFor="let user of users">
      {{ user.name }}
      <button
        class="btn btn-success float-right button"
        (click)="editUser(user.id)"
      >
        Edit
      </button>
      <button
        class="btn btn-info float-right button"
        (click)="viewUser(user.id)"
      >
        View
      </button>
      <button
        class="btn btn-danger float-right button"
        (click)="deleteUser(user.id)"
      >
        Delete
      </button>
    </li>
    <button
      class="btn btn-large btn-primary m-3"
      [routerLink]="['../add']"
    >
      Add User
    </button>
  </ul>
</div>
</div>
```

والآن لنستقبل هذه القيمة في ملف class في component الأبن، كالتالي:

#### ملف filter-users-list.component.ts

```
import {
  Component,
```

```

    EventEmitter,
    Input,
    OnInit,
    Output
} from '@angular/core';

@Component({
  selector: 'app-filter-users-list',
  templateUrl: './filter-users-list.component.html',
  styleUrls: ['./filter-users-list.component.css']
})

export class FilterUsersListComponent implements OnInit, OnChanges {

  @Output() valueChanges: EventEmitter<string> = new EventEmitter<string>();

  @Input() listItemsNum: number;
  private _listFilter: string;

  get listFilter(): string {
    return this._listFilter;
  }

  set listFilter(value: string) {
    this._listFilter = value;
    this.valueChanges.emit(value);
  }

  constructor() { }

  ngOnInit(): void { }
}

```



والآن نريد ان نظهر رسالة للمستخدم بناءً على القيمة الداخلة إلى هذا component عن طريق هذا @Input، وهنا نريد ان نسترجع مفهوم آخر قمنا بشرحه سابقاً وهو انه في حال اردنا ان نراقب أي تغييرات على القيمة الداخلة عن طريق @Input، فلدينا طريقتين الأولى عن طريق setter/getter والثاني عن طريق الدالة ngOnChanges وهي من دوال lifecycle hooks، وكلا الطريقتين تؤديان نفس المهمة وانا هنا سوف استخدم الطريقة الثانية، كالتالي:

ملف filter-users-list.component.ts

```

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

```

```

@Component({
  selector: 'app-filter-users-list',
  templateUrl: './filter-users-list.component.html',
  styleUrls: ['./filter-users-list.component.css']
})

export class FilterUsersListComponent implements OnInit, OnChanges {

  @Output() valueChanges: EventEmitter<string> = new EventEmitter<string>();
  @Input() listItemsNum: number;

  message: string;
  private _listFilter: string;

  get listFilter(): string {
    return this._listFilter;
  }

  set listFilter(value: string) {
    this._listFilter = value;
    this.valueChanges.emit(value);
  }

  constructor() { }

  ngOnInit(): void {

  }

  ngOnChanges(changes: SimpleChanges): void {
    if (changes.listItemsNum && !changes.listItemsNum.currentValue) {
      this.message = 'No Matches found';
    } else {
      this.message = `Items Found: ${this.listItemsNum}`;
    }
  }
}

```

الآن لنعمل Bind للمتغير message باستخدام interpolation، وهنا نريد ان نراجع ايضاً مفهوم آخر وهو Template Reference بحيث نضعه لي input وليكن اسمه #inputValue ونستخدمه بحيث لا نظهر نتائج البحث إلى إذا كان هنالك قيمة في مربع النص، كالتالي:

ملف filter-users-list.component.html

```

<div class="input-group mt-2 mb-2">
  <div class="input-group-prepend">
    <span class="input-group-text">Search</span>
  </div>
  <input
    type="text"
    #inputValue

```




```

        class="form-control"
        [(ngModel)]="listFilter"
    />
</div>

<div class="row mt-2 mb-2" *ngIf="inputValue.value !== ''">
    {{ message }}
</div>

```



وهناك أيضاً مفهوم آخر نريد ان نراجعته من خلال وضع المؤشر في مربع البحث عند بداية تشغيل التطبيق، ونستطيع القيام بذلك عن طريق @ViewChild حيث ان العنصر المستهدف وضعنا له قبل قليل Template Reference وهو #inputValue لذلك سوف نستخدمه للوصول إلى هذا العنصر في ملف class وبنفس الوقت نسترجع الكلام الذي قلناه سابقاً وهو انه من غير المحبب الوصول إلى خصائص أي عنصر في DOM بشكل مباشر وانما يفضل استخدام الـ Service الجاهزة Renderer2، كالتالي:

ملف filter-users-list.component.ts

```

import { AfterViewInit } from '@angular/core';
import {
    Component,
    ElementRef,
    EventEmitter,
    Input,
    OnChanges,
    OnInit,
    Output,
    Renderer2,
    SimpleChanges,
    ViewChild
} from '@angular/core';

@Component({
    selector: 'app-filter-users-list',
    templateUrl: './filter-users-list.component.html',
    styleUrls: ['./filter-users-list.component.css']
})
export class FilterUsersListComponent implements OnInit, OnChanges, AfterViewInit {

    @ViewChild('inputValue') inputValue: ElementRef;
    @Output() valueChanges: EventEmitter<string> = new EventEmitter<string>();
    @Input() listItemsNum: number;

    message: string;

    private _listFilter: string;

    get listFilter(): string {
        return this._listFilter;
    }

    set listFilter(value: string) {

```




```

    this._listFilter = value;
    this.valueChanges.emit(value);
  }

  constructor(private renderer: Renderer2) { }

  ngOnInit(): void {
  }

  ngOnChanges(changes: SimpleChanges): void {
    if (changes.listItemsNum && !changes.listItemsNum.currentValue) {
      this.message = 'No Matches found';
    } else {
      this.message = `Items Found: ${this.listItemsNum}`;
    }
  }

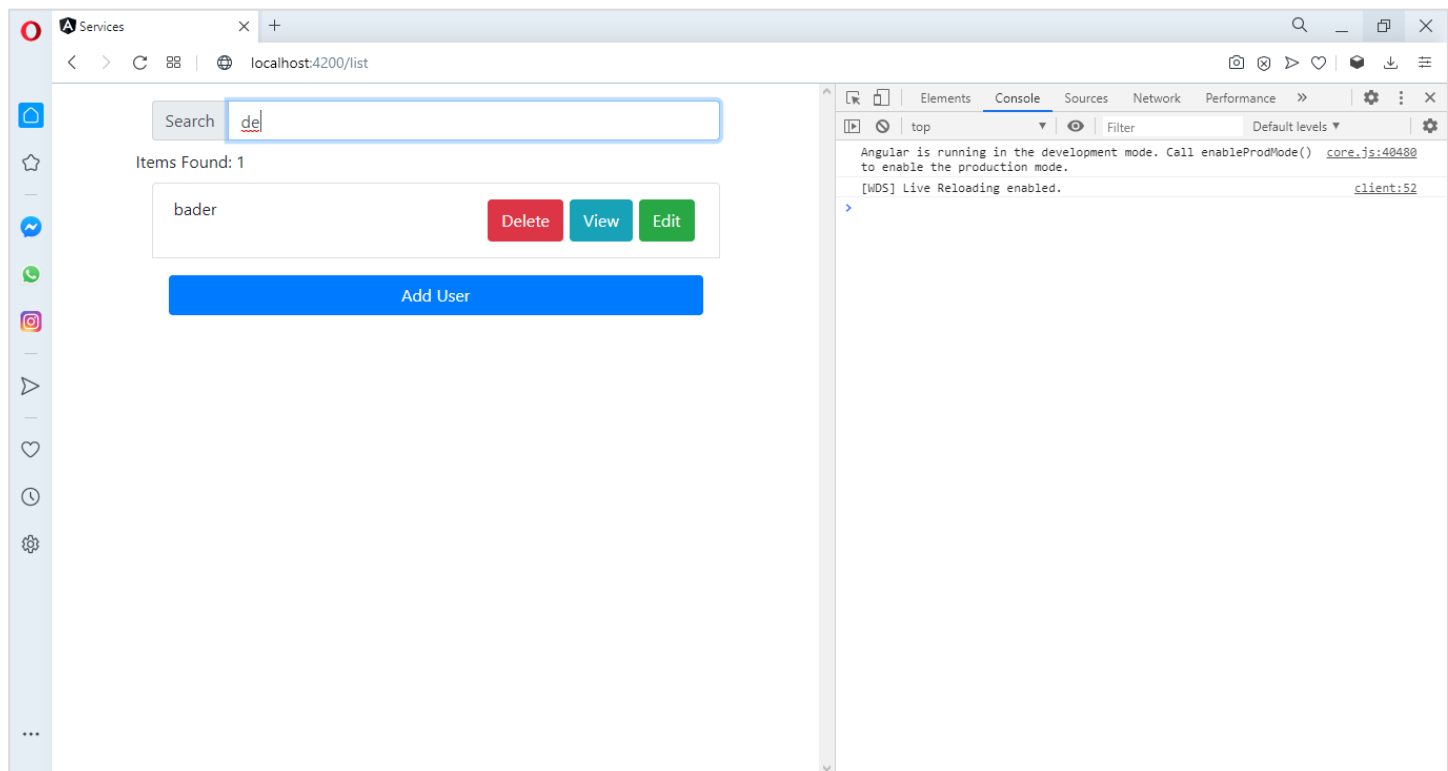
  ngAfterViewInit(): void {
    this.renderer.selectRootElement(this.inputValue.nativeElement).focus();
  }
}

```

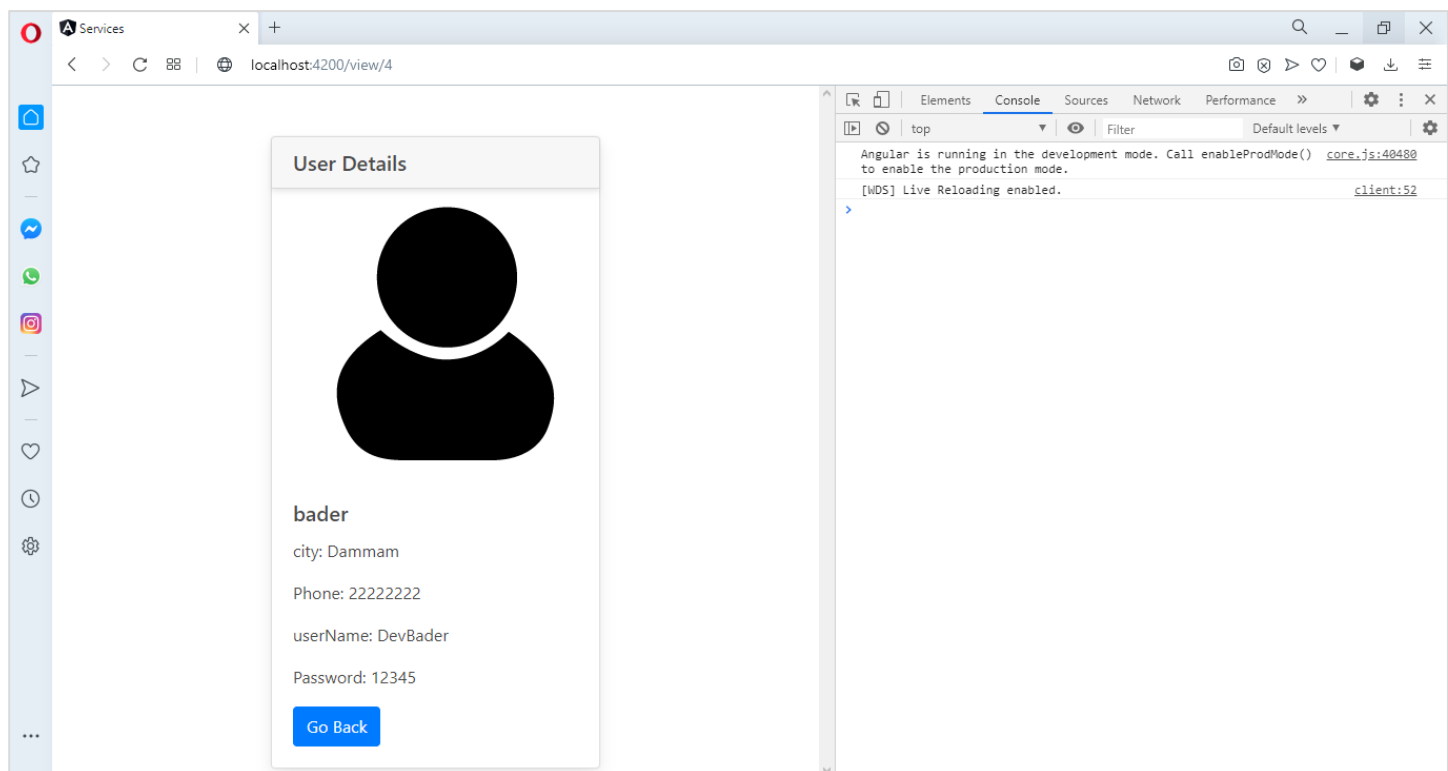


الآن بعد استعراض اغلب المفاهيم التي شرحناها سابقاً من خلال هذا المثال، لنقم بحفظ التعديلات ونذهب إلى المتصفح لنرى النتيجة، كالتالي:

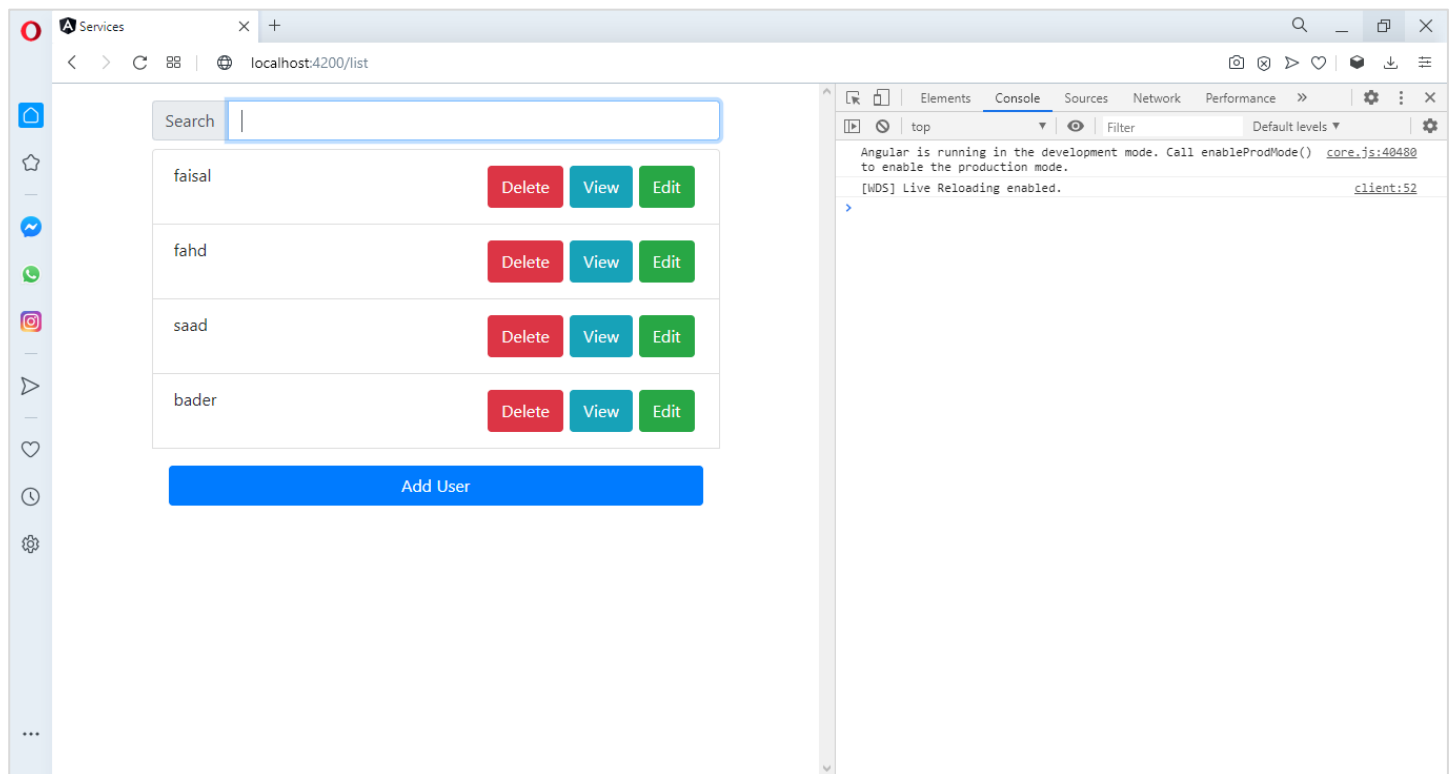
في بداية تشغيل التطبيق نلاحظ وجود المؤشر في مربع البحث، لنقم بعمل فلترة لهذه القائمة، كالتالي:



نلاحظ تمت فلتر القائمة بناءً على القيمة المدخلة في مربع البحث، وبنفس الوقت ظهرت لنا الرسالة التي تبين عدد نتائج البحث، الآن لنقم باختيار الزر View أو Edit، كالتالي:



ولنضغط على زر Go Back، كالتالي:



نلاحظ اختفى النص الموجود في مربع البحث ورجعت لنا القائمة كما كانت، لذلك نحتاج إلى service، بحيث نقوم بتخزين قيمة البحث داخل متغير في هذا component وعند إعادة بنائه من جديد نمرر القيمة ونعمل فلتر للقائمة من جديد، لذلك لنقوم بإضافة service جديدة وليكن اسمها Filter Users List Service، وبداخلها لنعرف متغير (خاصية) وليكن اسمها filteredList، كالتالي:

ملف filter-users-list.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})

export class FilterUsersListService {

  filterdList = '';

  constructor() { }
}
```

الآن نحتاج إلى خطوتين في ملف users-list.component.ts الأولى هي تخزين القيمة في هذا المتغير والثاني هي قراءة القيمة من هذا المتغير، أما لتخزين القيمة في المتغير الموجود في service فليس هنالك مكان افضل من onChangesValue والتي عن طريقها نستقبل القيمة التي مررناها من الابن إلى الأب، وهذه القيمة هي ما يكتبه المستخدم في مربع البحث، أما لقراءة القيمة، فنحتاج إلى حفظ القيمة المخزنة في المتغير filterList والموجودة في component الأب، حيث نقوم فقط بإسناد القيمة إلى هذا المتغير وAngular سوف يقوم بباقي، حيث انه يعلم اننا قمنا بربط هذا المتغير وارسال قيمته من الابن إلى الأب عن طريق

@Output، ومن ثم استقبال هذه القيمة في الدالة onChangesValue ومحتوى هذه الدالة هو القيام بالفلتر بناءً على هذه القيمة، كالتالي:

ملف users-list.component.ts

```
import { AfterViewInit, Component, OnInit, ViewChild } from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';
import { FilterUsersListComponent } from '../filter-users-list/filter-users-list.component';
import { FilterUsersListService } from '../core/filter-users-list.service';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
})
export class UsersListComponent implements OnInit, AfterViewInit {
  @ViewChild(FilterUsersListComponent) filterUsersList: FilterUsersListComponent;
  users: User[];
  constructor(
    private apiService: ApiService,
    private router: Router,
    private filterUsersListService: FilterUsersListService,
  ) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
  }

  ngAfterViewInit(): void {
    this.filterUsersList.listFilter = this.filterUsersListService.filteredList;
  }

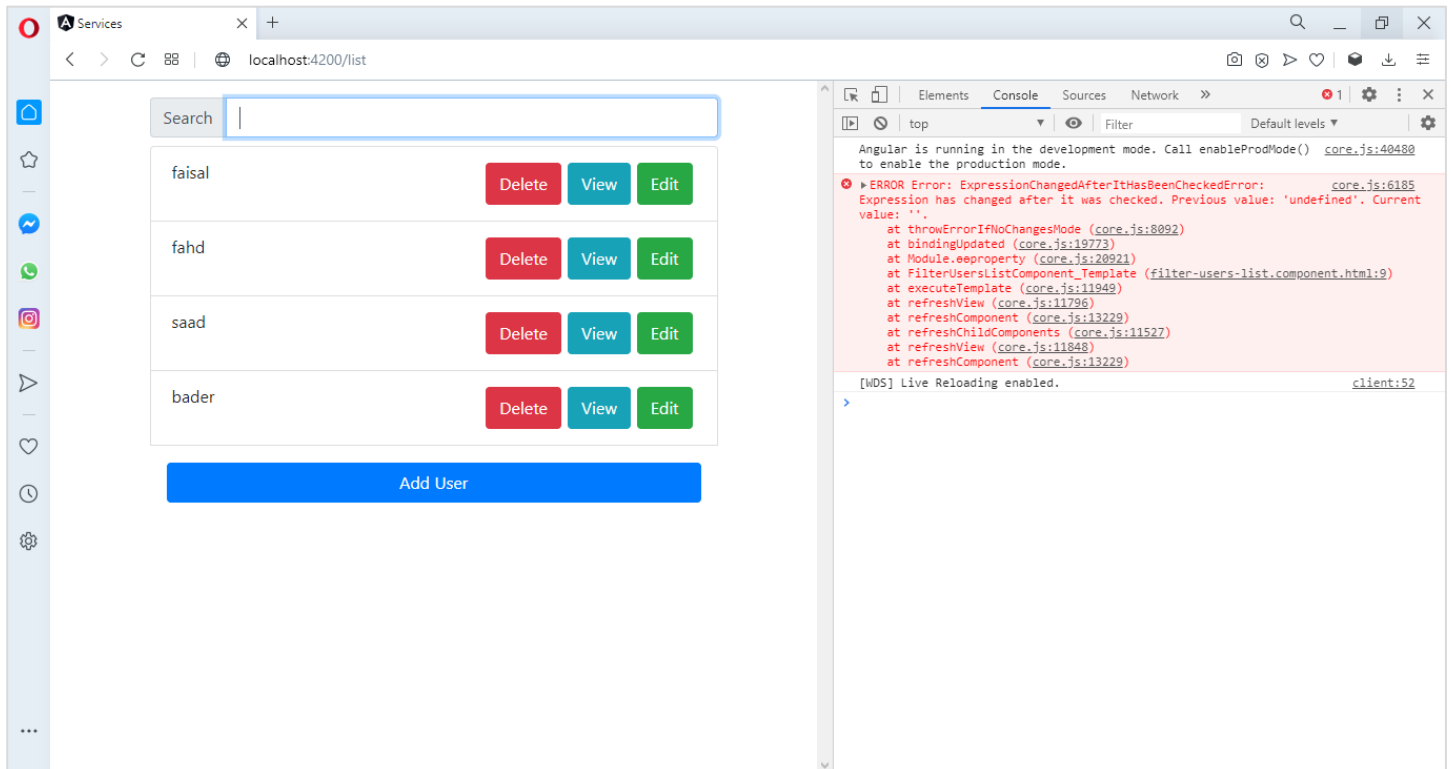
  onChangesValue(value: string) {
    this.filterUsersListService.filteredList = value;
    if (value) {
      const result = this.users.filter(user =>
        user.name.toLocaleLowerCase().indexOf(value.toLocaleLowerCase()) !== -1);
      this.users = result;
    } else {
      this.users = this.apiService.getAllUsers();
    }
  }

  editUser(id: number) {
    this.router.navigate(['./edit', id]);
  }

  viewUser(id: number) {
    this.router.navigate(['./view', id]);
  }
}
```

```
deleteUser(id: number) {
  this.apiService.deleteUser(id);
}
```

هنا أيضاً قمنا بمراجعة مفهوم سابق تكلمنا عنه وهو طريقة الوصول إلى جميع الخصائص والدوال الموجودة في component الأب عن طريق ملف class في component الأب، وقلنا نستطيع القيام بذلك عن طريق @ViewChild، وهذا ما نريده في الحقيقة فنريد الوصول إلى المتغير (الخاصية) filterList الموجودة في component الأب لكي نقوم بقراءة القيمة التي قمنا بتخزينها في service داخل المتغير (الخاصية) filteredList ونقوم بإسنادها إلى الخاصية filterList، أما تخزين القيمة فهو واضح حيث نقوم بتخزينها داخل الدالة onChangeValue، الآن لنقم بحفظ التعديلات ونذهب إلى المتصفح لنرى النتيجة:



نلاحظ ظهر لنا خطأ وهذا الخطأ هو في الحقيقة ناتج عن Angular Change Detection، وسوف نقوم بشرحه بإذن الله في فصل مواضيع متقدمة في components، أما الآن فقم فقط بإضافة هذه الاسطر البرمجية لكي نُزيل هذا الخطأ، كالتالي:

```
import {
  AfterViewInit,
  ChangeDetectorRef,
  Component,
  OnInit,
  ViewChild
} from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';
import { FilterUsersListComponent } from '../filter-users-list/filter-users-list.component';
import { FilterUsersListService } from '../core/filter-users-list.service';

@Component({
  selector: 'app-users-list',
```

```

templateUrl: './users-list.component.html',
styleUrls: ['./users-list.component.css'],
}))

export class UsersListComponent implements OnInit, AfterViewInit {

  @ViewChild(FilterUsersListComponent) filterUsersList: FilterUsersListComponent;

  users: User[];

  constructor(
    private apiService: ApiService,
    private router: Router,
    private filterUsersListService: FilterUsersListService,
    private cdr: ChangeDetectorRef
  ) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
  }

  ngAfterViewInit(): void {
    this.filterUsersList.listFilter = this.filterUsersListService.filteredList;
    this.cdr.detectChanges();
  }

  onChangesValue(value: string) {
    this.filterUsersListService.filteredList = value;
    if (value) {
      const result = this.users.filter(user =>
        user.name.toLocaleLowerCase().indexOf(value.toLocaleLowerCase()) !== -1);
      this.users = result;
    } else {
      this.users = this.apiService.getAllUsers();
    }
  }

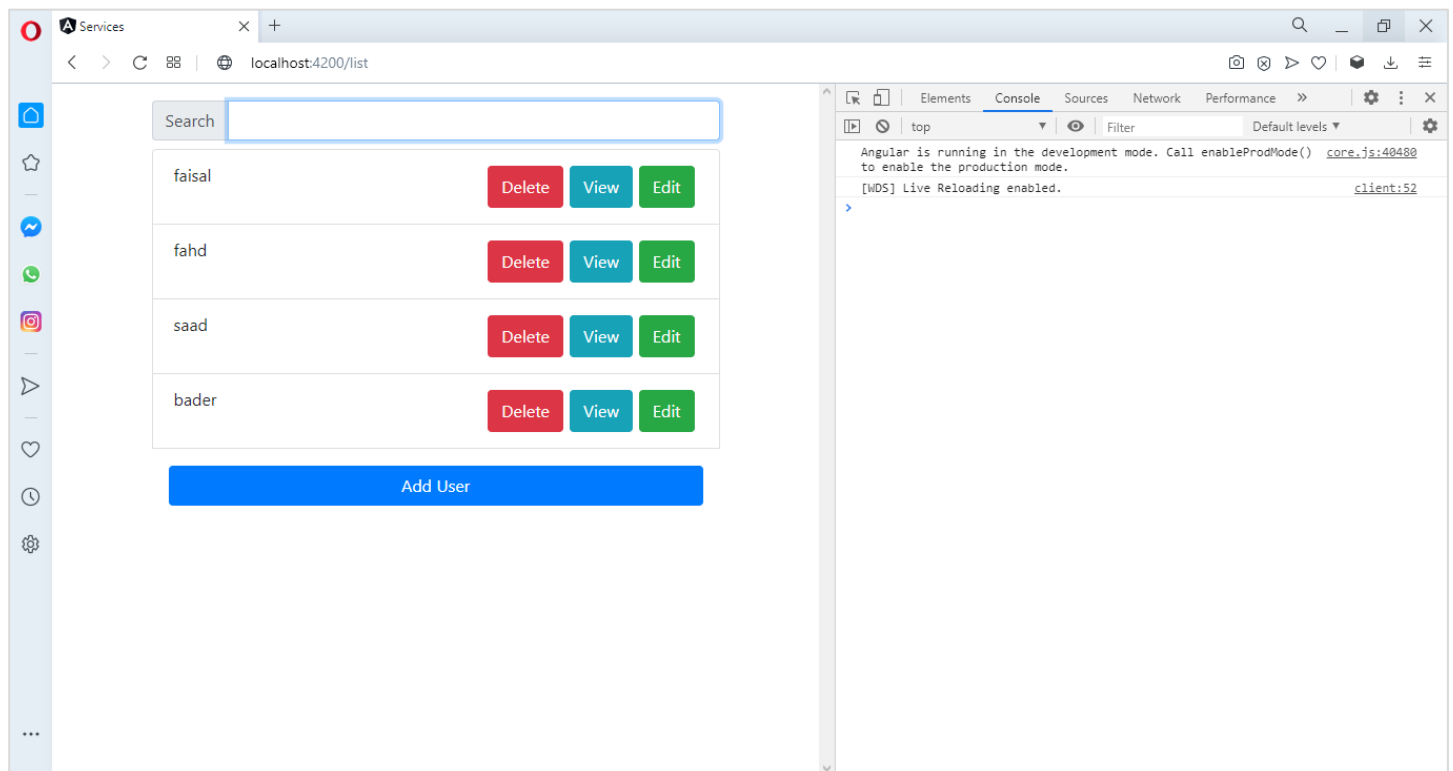
  editUser(id: number) {
    this.router.navigate(['./edit', id]);
  }

  viewUser(id: number) {
    this.router.navigate(['./view', id]);
  }

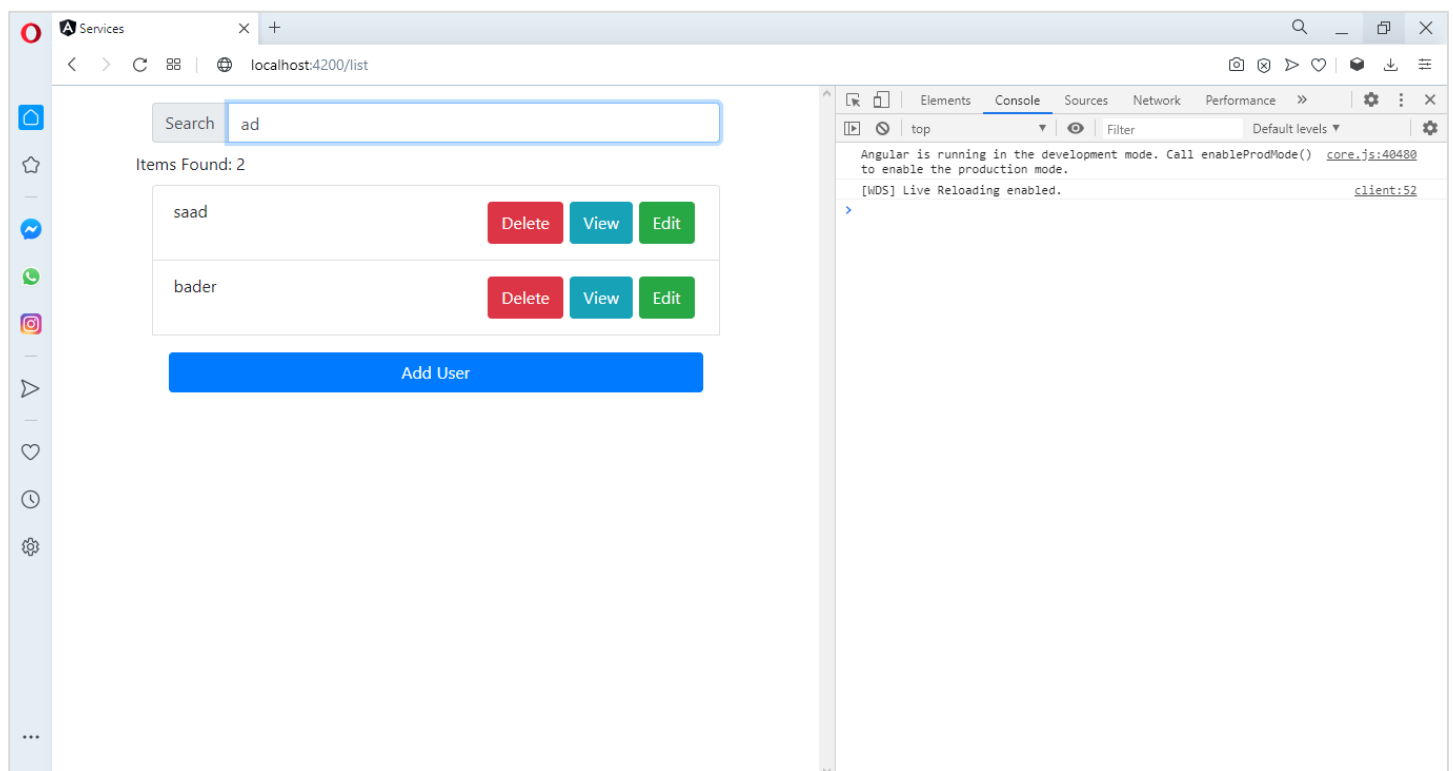
  deleteUser(id: number) {
    this.apiService.deleteUser(id);
  }
}

```

الآن لنحفظ التعديلات ونذهب إلى المتصفح ونرى النتيجة، كالتالي:

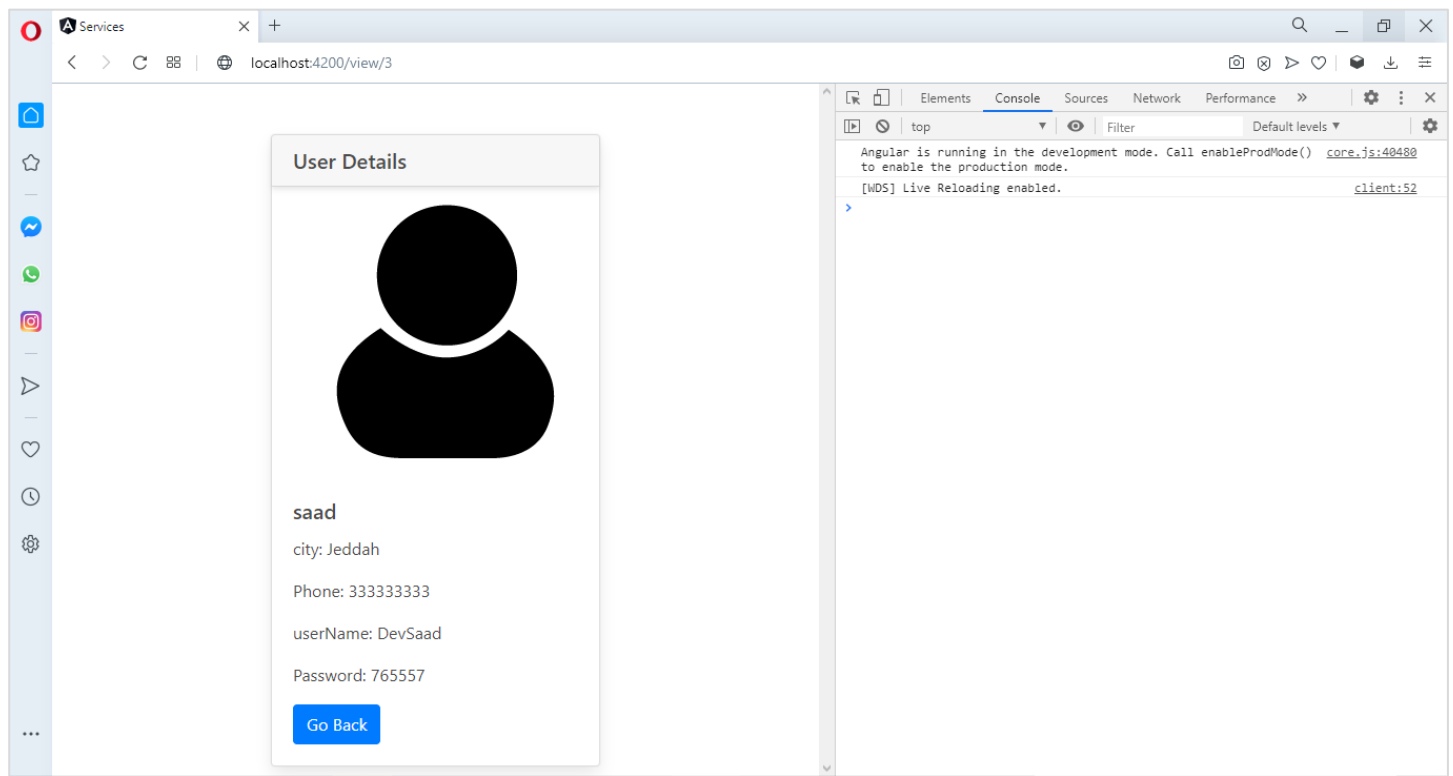


نلاحظ أن الخطأ اختفى، الآن لنقم بكتابة أي قيمة في مربع البحث، كالتالي:

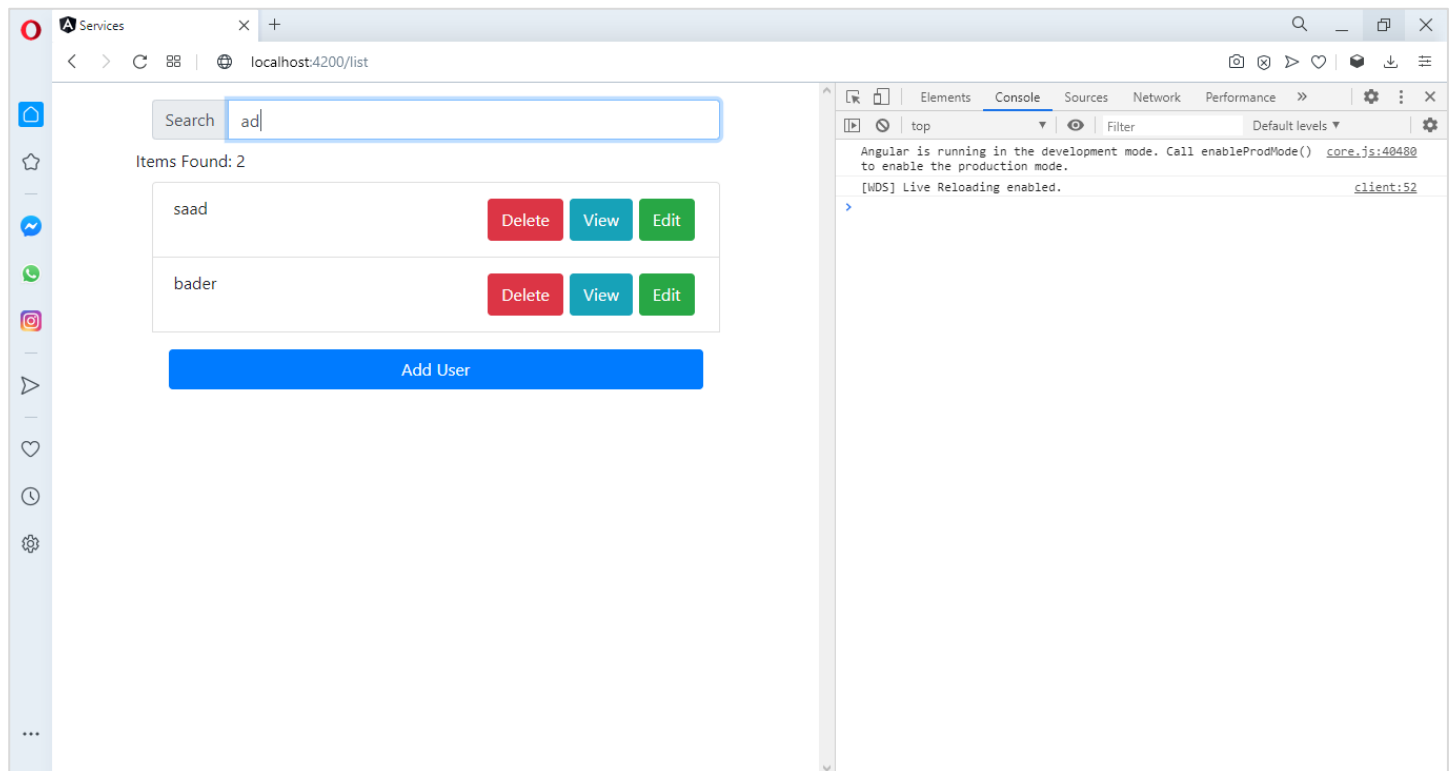


الآن لنختار أحد المستخدمين ونضغط على أحد الأزرار بجانبه سواء View أو Edit، كالتالي:





الآن لنضغط على زر Go Back، كالتالي:



نلاحظ ان نتائج البحث لازالت موجودة والقائمة على وضعها، وبذلك تعلمنا كيف نقوم بمخاطبة component ونفسه عن طريق استخدام service، اما الجزء القادم فسوف نشرح كيف يمكن ان نستخدم services لتواصل مع component المختلفة سواء كان بينهم علاقة ام لم يكن.

### 7.3. استخدام Services لتواصل بين Components:

وهنا لدينا استخدامين الأول الاستخدام البسيط وهو ان يكون لدينا مثلاً خاصية تحمل قيمة معينة وبنفس الوقت لدينا دالة تقوم بتغيير قيمة هذه الخاصية، بحيث يكون تنفيذ هذه الدالة من خلال component معين، لتغيير قيمة هذه الخاصية، وبذلك سوف يتم تعديلها في كافة components التي تستدعي هذه الخاصية، اما الطريقة الثانية وهي الأكثر احترافية والأكثر استخداماً في التطبيقات الواقعية وذلك عن طريق استخدام Subjects، وسوف نتكلم عن كلا الطريقتين.

#### 1.7.3. الطريقة البسيطة لتواصل بين components عن طريق service:

وهي مشابهة نوعاً ما، إلى طريقة اتصال component مع نفسه باستخدام service، ولكن هنا نعمل مشاركة بين مجموعة من components لقيمة معينة او مجموعة قيم، فمثلاً لو كان لدينا مجموعة قيم وفي حال تغير هذه القيمة يتم مشاركة هذه القيمة مع جميع components التي تستدعي هذه المتغيرات.

ولتوضيح لنعطي مثال، لنقوم ببناء نظام تسجيل دخول مبسط، بحيث يكون لدينا صفحة تسجيل الدخول وعندما يقوم المستخدم بتسجيل الدخول يقوم التطبيق بتوجيهه نحو صفحة قائمة المستخدمين، بحيث الازرار الخاصة بحذف المستخدمين و اضافتهم وتعديل بياناتهم لا تظهر إلا للمستخدم الذي يكون اسم المستخدم له DivFaisal، وسوف نقوم باستخدام مفاهيم التواصل بين components البسيط باستخدام service معينة، بحيث يكون لدينا service تحتوي على متغيرين (خاصيتين) من النوع المنطقي true او false، واحد منهما لتأكد من ان المستخدم قام بتسجيل الدخول من عدمه، والآخر لتأكد هل المستخدم هو DivFaisal ام لا، وسوف نقوم بمشاركة هذين المتغيرين مع جميع components، وبناءً على قيمهما نخفي ونظهر بعض أجزاء component او نقوم بتوجيهه إلى صفحة الدخول مرة أخرى، والآن لنقم في البداية بإنشاء service جديدة في مجلد core وليكن اسمها login، ومن ثم نقوم بتعريف متغيرين فيها عن طريق setter/getter، وليكن اسم أحدهما isLoggedIn وهو الذي سوف نستعمل معه getter/setter والآخر ليكن اسمه isAdmin، كالتالي:

ملف login.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})

export class LoginService {

  constructor() { }

  public isAdmin = false;

  private _isLoggedIn = false;
  public get isLoggedIn(): boolean {
    return this._isLoggedIn;
  }
  public set isLoggedIn(value: boolean) {
    this._isLoggedIn = value;
  }
}
```

والسبب في استخدامنا لـ getter/setter هو أننا نريد كتابة كود إضافي في حال تغير القيمة وهو توجيه المستخدم إلى صفحة قائمة المستخدمين في حال كانت القيمة true وإذا كانت false نعيد توجيهه إلى صفحة تسجيل الدخول، كالتالي:

#### ملف login.service.ts

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})

export class LoginService {

  constructor(private router: Router) { }

  isAdmin = false;

  private _isLoggedIn = false;
  public get isLoggedIn(): boolean {
    return this._isLoggedIn;
  }
  public set isLoggedIn(value: boolean) {
    this._isLoggedIn = value;
    value ? this.router.navigate(['/list']) : this.router.navigate(['/login']);
  }
}
```

الآن لنقوم بإنشاء component جديد وليكن اسمه login وفي ملف template الخاص به نضيف Markup التالي:

#### ملف login.component.html

```
<div class="container mt-5">
  <div class="card align-middle">
    <div class="card-header">
      LogIn
    </div>
    <div class="card-body">
      <div class="input-group mb-3">
        <div class="input-group-prepend">
          <span class="input-group-text" id="basic-addon1">Username</span>
        </div>
        <input
          #inputUserName
          type="text"
          class="form-control"
          aria-label="Username"
          aria-describedby="basic-addon1"
        />
      </div>
      <div class="input-group mb-3">
        <div class="input-group-prepend">
          <span class="input-group-text" id="basic-addon2">Password </span>

```

```

    </div>
    <input
      type="password"
      class="form-control"
      aria-label="Password"
      aria-describedby="basic-addon2"
    />
  </div>
</div>
<div class="card-footer">
  <a
    (click)="login(inputUserName.value)"
    class="btn btn-outline-dark btn-lg btn-block"
  >
    login
  </a>
</div>
</div>
</div>

```



نلاحظ لدينا دالة هذه الدالة اسمها login ومررنا لها قيمة مربع الإدخال الأول الخاص باسم المستخدم عن طريق template reference، الآن لنقوم بإنشاء هذه الدالة وكتابتها في ملف class، كالتالي:

ملف login.component.ts

```

import { Component, OnInit } from '@angular/core';
import { LoginService } from '../core/login.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})

export class LoginComponent implements OnInit {

  constructor(private loginService: LoginService) { }

  ngOnInit(): void { }

  login(value: string) {
    if (!value) { alert('No Value'); return; }
    value === 'DivFaisal' ? this.loginService.isAdmin = true :
this.loginService.isAdmin = false;
    this.loginService.isLoggedIn = true;
  }
}

```



نلاحظ اننا قمنا بعمل شرط بحيث إذا كانت قيمة value (وهي عبارة عن قيمة الباراميتر الذي مررناه لهذه الدالة في ملف template ويمثل القيمة التي يُدخلها المستخدم في مربع الإدخال الخاص باسم المستخدم)، تساوي DivFaisal فقم بتغيير القيمة الموجودة في الخاصية isAdmin إلى true، وإلا ضعها false، وبنفس الوقت قمنا بتغيير قيمة الخاصية isLoggedIn إلى

true وهي نفسها الخاصية التي عملنا لها getter/setter بحيث إذا كانت true قم بالانتقال إلى صفحة قائمة المستخدمين، ولو رجعنا إلى المقدمة التي تكلمنا فيها عن ان القيم الموجودة في service نقوم بتغييرها من خلال component واحد او مجموعة من components ومن ثم نقوم بمشاركة هذه التغييرات في components الأخرى، وفي الحقيقة هذا ما عملناه هنا حيث انه في هذه الدالة login() قمنا بتغيير قيم هذه الخصائص وفق شروط معينة وفي components الأخرى لنقوم بقراءة هذه الخصائص وبناءً على قيمهما ننفذ مهام معينة، لذلك سوف نذهب إلى ملف class لي component الخاص بعرض قائمة المستخدمين ونقوم بقراءة قيم الخصائص من ملف service، كالتالي:

ملف users-list.component.ts

```
import {
  AfterViewInit,
  ChangeDetectorRef,
  Component,
  OnInit,
  ViewChild
} from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';
import { FilterUsersListComponent } from '../filter-users-list/filter-users-list.component';
import { FilterUsersListService } from '../core/filter-users-list.service';
import { LoginService } from '../core/login.service';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
})

export class UsersListComponent implements OnInit, AfterViewInit {
  @ViewChild(FilterUsersListComponent) filterUsersList: FilterUsersListComponent;
  users: User[];
  isAdmin: boolean;
  constructor(
    private apiService: ApiService,
    private router: Router,
    private filterUsersListService: FilterUsersListService,
    private cdr: ChangeDetectorRef,
    private loginService: LoginService
  ) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
    this.isAdmin = this.loginService.isAdmin;
  }

  ngAfterViewInit(): void {
    this.filterUsersList.listFilter = this.filterUsersListService.filteredList;
    this.cdr.detectChanges();
  }
}
```

```

onChangesValue(value: string) {
  this.filterUsersListService.filteredList = value;
  if (value) {
    const result = this.users.filter(user =>
      user.name.toLocaleLowerCase().indexOf(value.toLocaleLowerCase()) !== -1);
    this.users = result;
  } else {
    this.users = this.apiService.getAllUsers();
  }
}

editUser(id: number) {
  this.router.navigate(['./edit', id]);
}

viewUser(id: number) {
  this.router.navigate(['./view', id]);
}

deleteUser(id: number) {
  this.apiService.deleteUser(id);
}

logout() {
  this.loginService.isLoggedIn = false;
  this.loginService.isAdmin = false;
}
}

```




نلاحظ اننا قمنا بتعريف متغير (خاصية) اسميته isAdmin لقراءة قيمة الخاصية التي تحمل نفس الاسم في ملف service، وقمنا ايضاً ببناء دالة اسميتها logout() لكي تقوم هي ايضاً بتغيير قيم الخصائص في service، الآن لنذهب إلى ملف template لهذا component ونقوم بإنشاء هذا الزر الذي ينفذ الدالة logout() وبنفس الوقت يُخفي او يُظهر الازرار View و Edit و Add User عن طريق \*ngIf، كالتالي:

ملف users-list.component.html

```

<div class="card h-25">
  <div class="card-footer text-center">
    <button class="btn btn-danger btn-small" (click)="logout()">
      Logout
    </button>
  </div>
</div>
<div class="container">
  <div class="row justify-content-center pt-2">
    <app-filter-users-list
      [listItemsNum]="users?.length"
      (valueChanges)="onChangesValue($event)"
    >
  </app-filter-users-list>

```



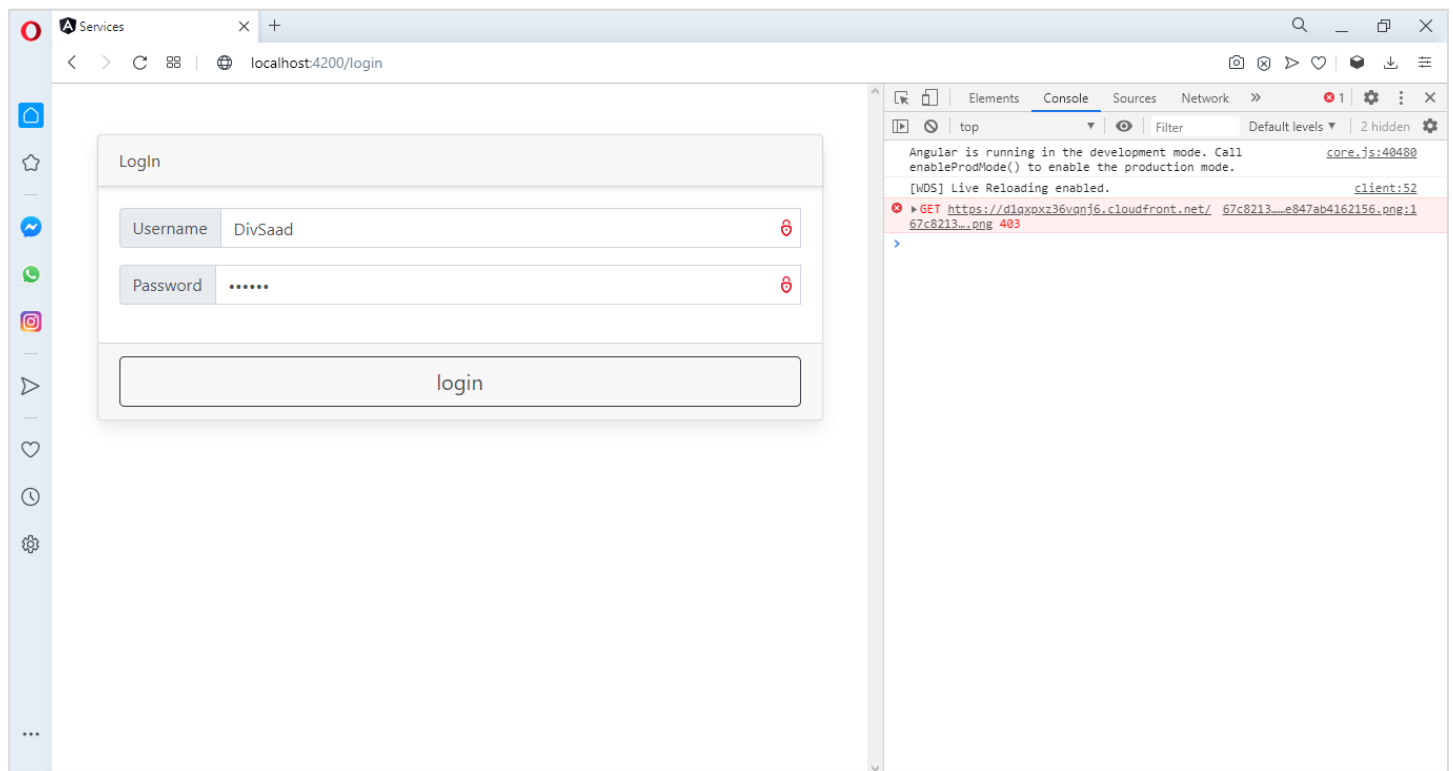
```

<ul class="list-group w-100">
  <li class="list-group-item" *ngFor="let user of users">
    {{ user.name }}
    <button
      *ngIf="isAdmin"
      class="btn btn-success float-right button"
      (click)="editUser(user.id)"
    >
      Edit
    </button>
    <button
      class="btn btn-info float-right button"
      (click)="viewUser(user.id)"
    >
      View
    </button>
    <button
      *ngIf="isAdmin"
      class="btn btn-danger float-right button"
      (click)="deleteUser(user.id)"
    >
      Delete
    </button>
  </li>
  <button
    *ngIf="isAdmin"
    class="btn btn-large btn-primary m-3"
    [routerLink]="['../add']"
  >
    Add User
  </button>
</ul>
</div>
</div>

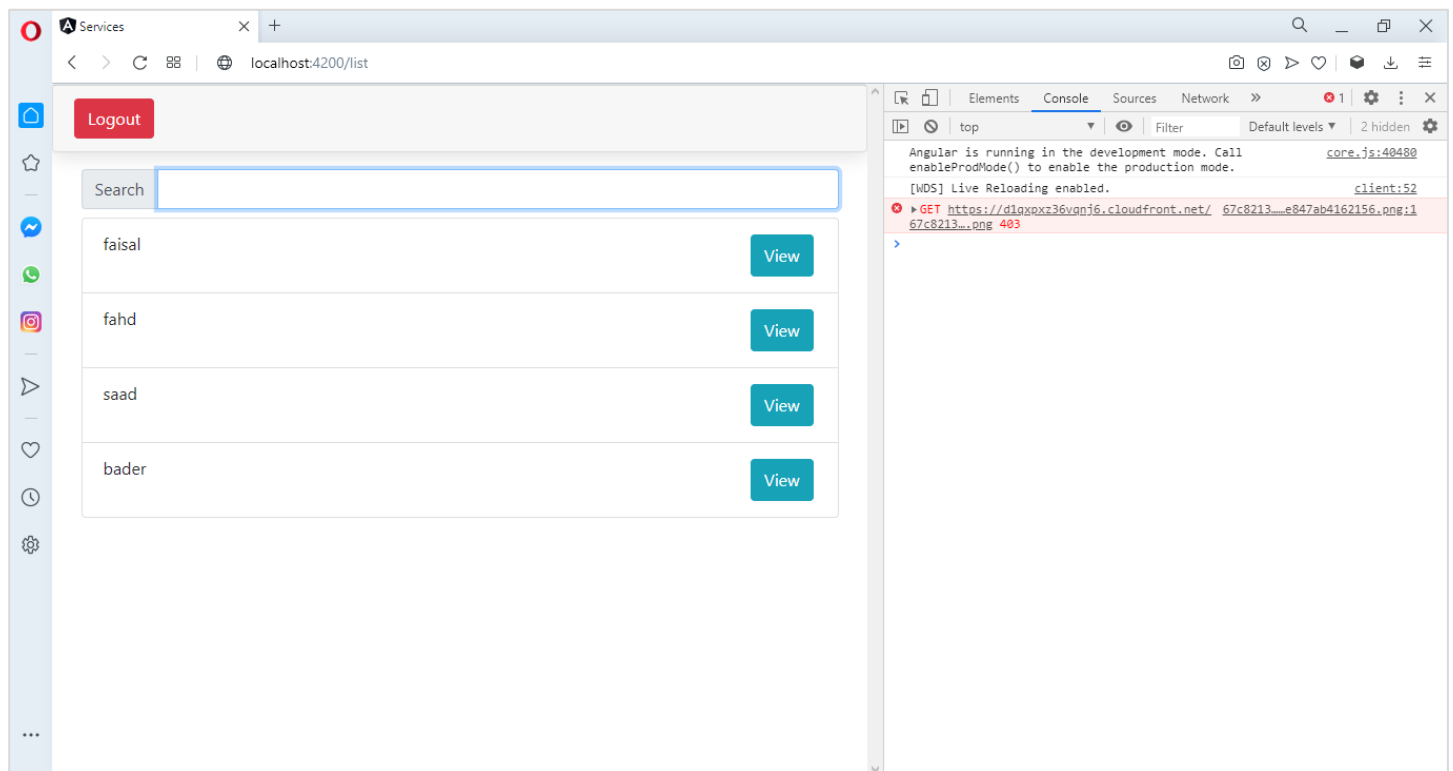
```

ونستطيع توسيع نطاق مشاركة هذه الخصائص مع components الأخرى كأن نقوم مثلاً تخزين الاسم بناءً على اسم المستخدم الذي اظهره ونشارك هذه القيمة مع components أخرى، أو كأن نظهر له رسالة ترحيب في أي component يزوره المستخدم، مع العلم ان الطريقة هي نفسها من حيث تغيير قيم الخصائص او قراءتها، لذلك منعاً لتكرار نكتفي بمشاركة الخصائص بين اثنين components صفحة تسجيل الدخول وصفحة لعرض قائمة المستخدمين.

الآن لنحفظ التعديلات ونذهب إلى المتصفح ونرى النتيجة، كالتالي:

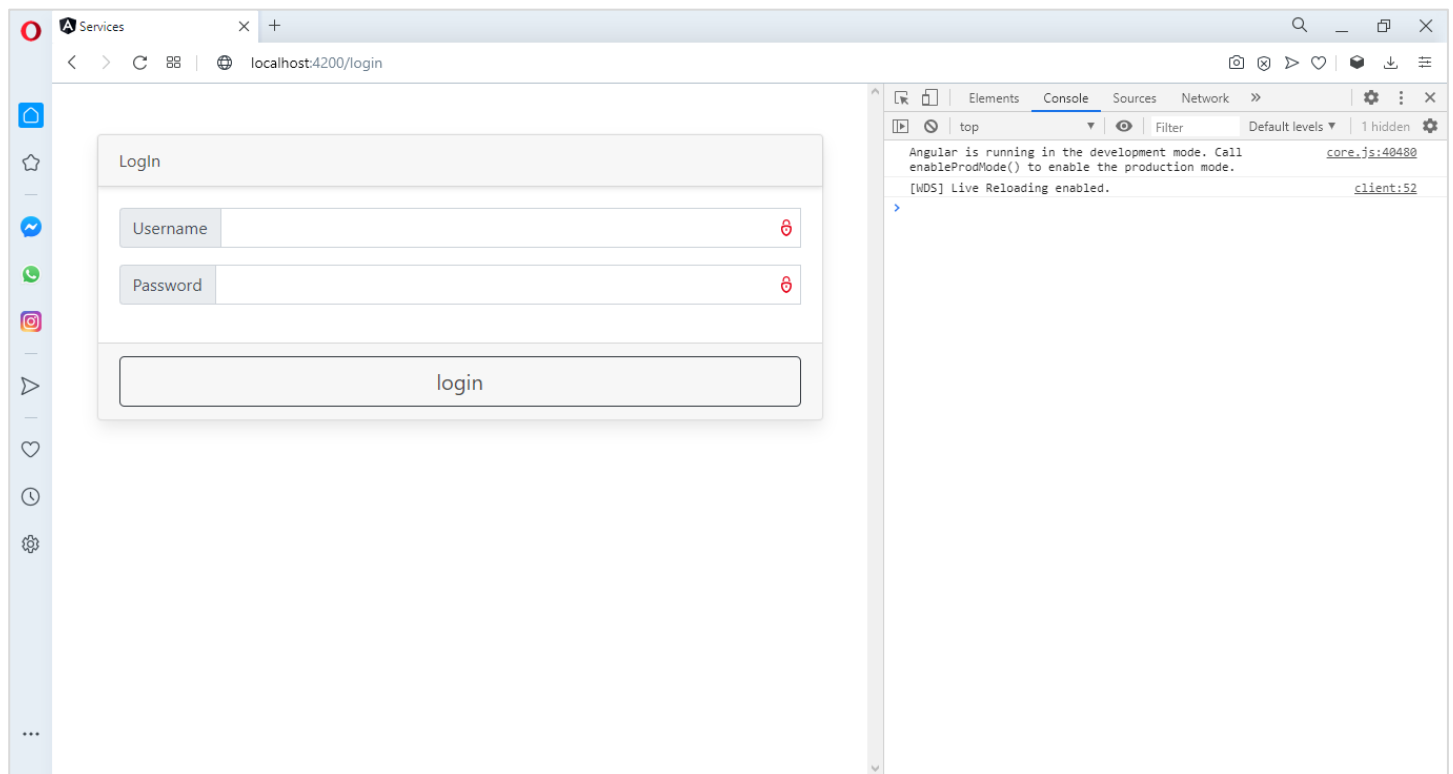


ما سبق هو صفحة تسجيل الدخول لنقم بإدخال اسم مستخدم DivSaad ونضغط على زر login، ونرى النتيجة:

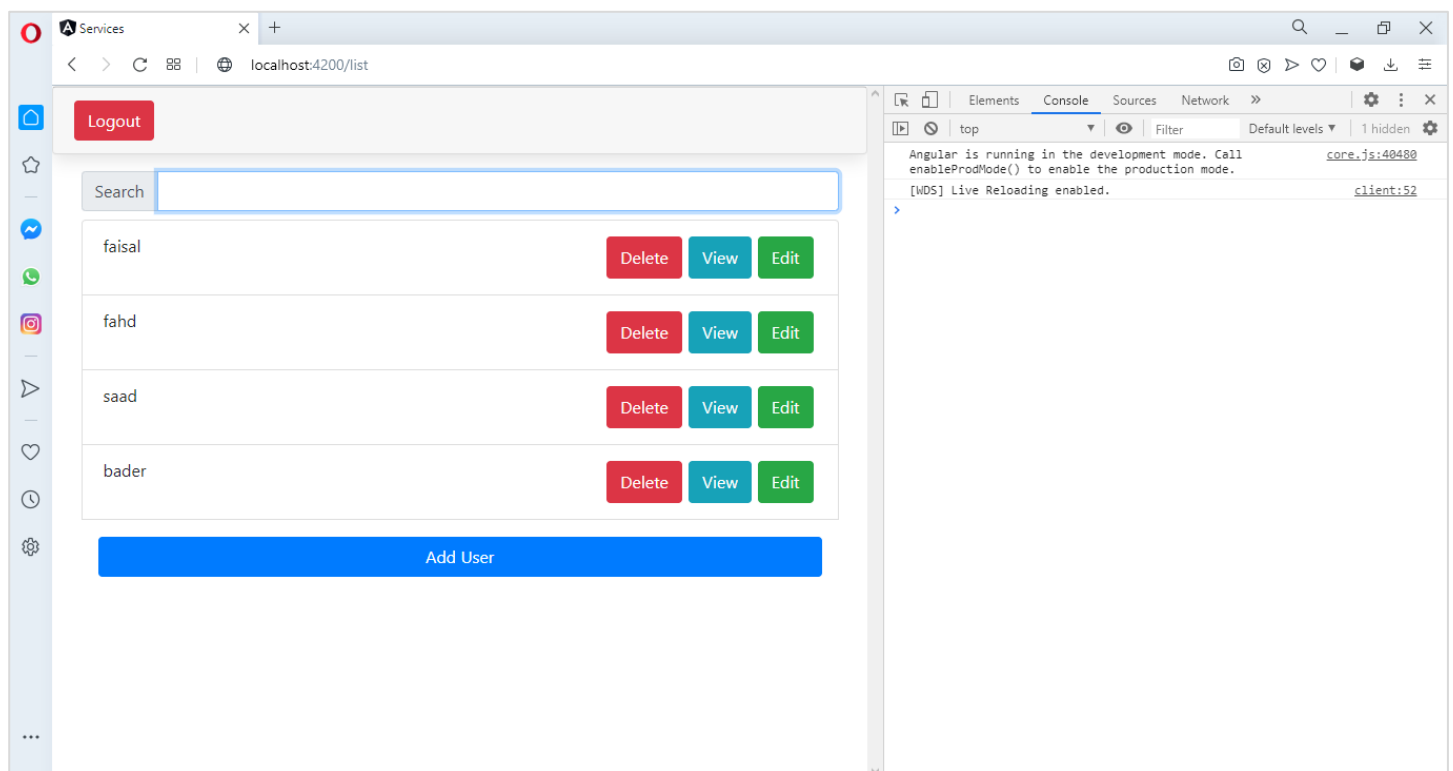


نلاحظ الإزرار الخاصة بإضافة مستخدم جديد او حذفه او التعديل عليه غير موجود لأن الخاصية isAdmin هي false، الآن لنضغط على زر logout ونرى النتيجة:



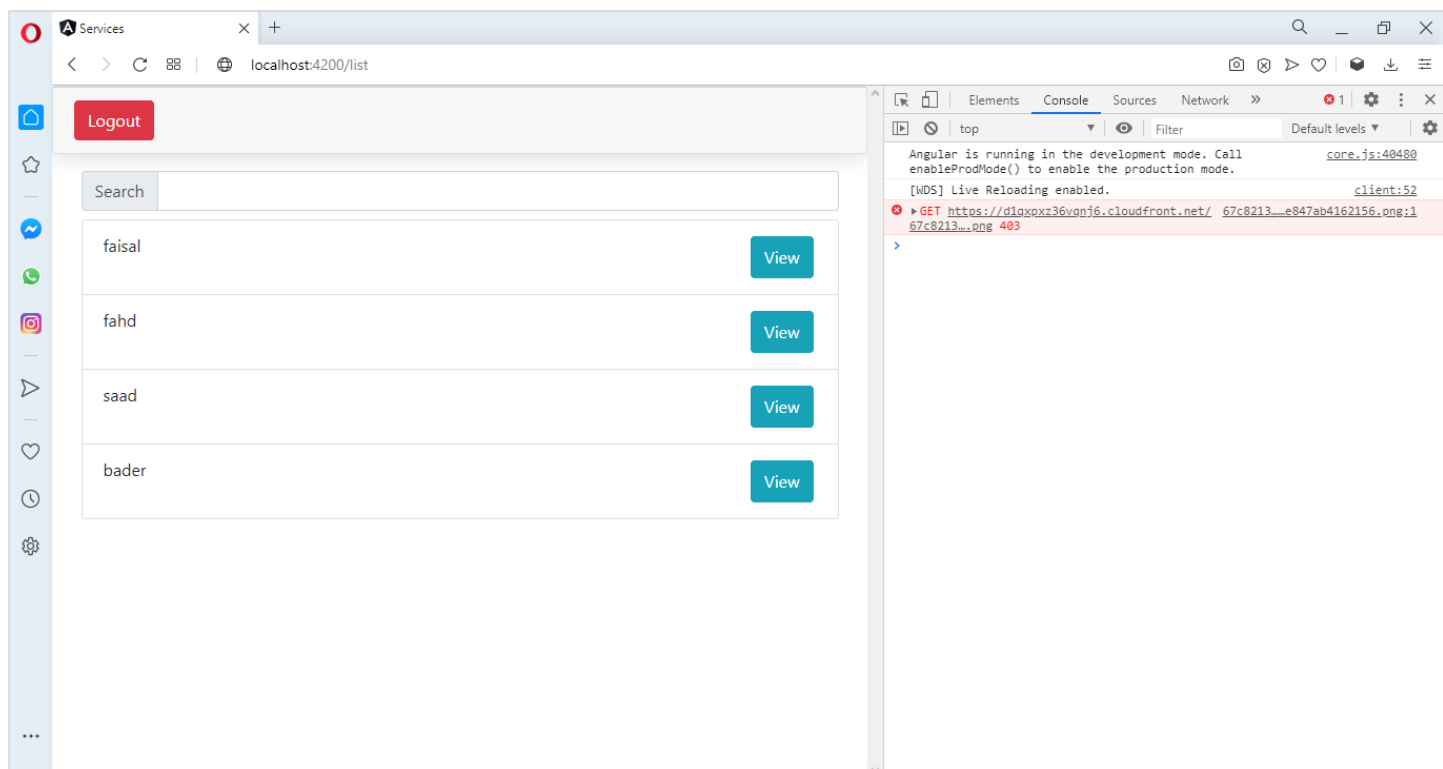
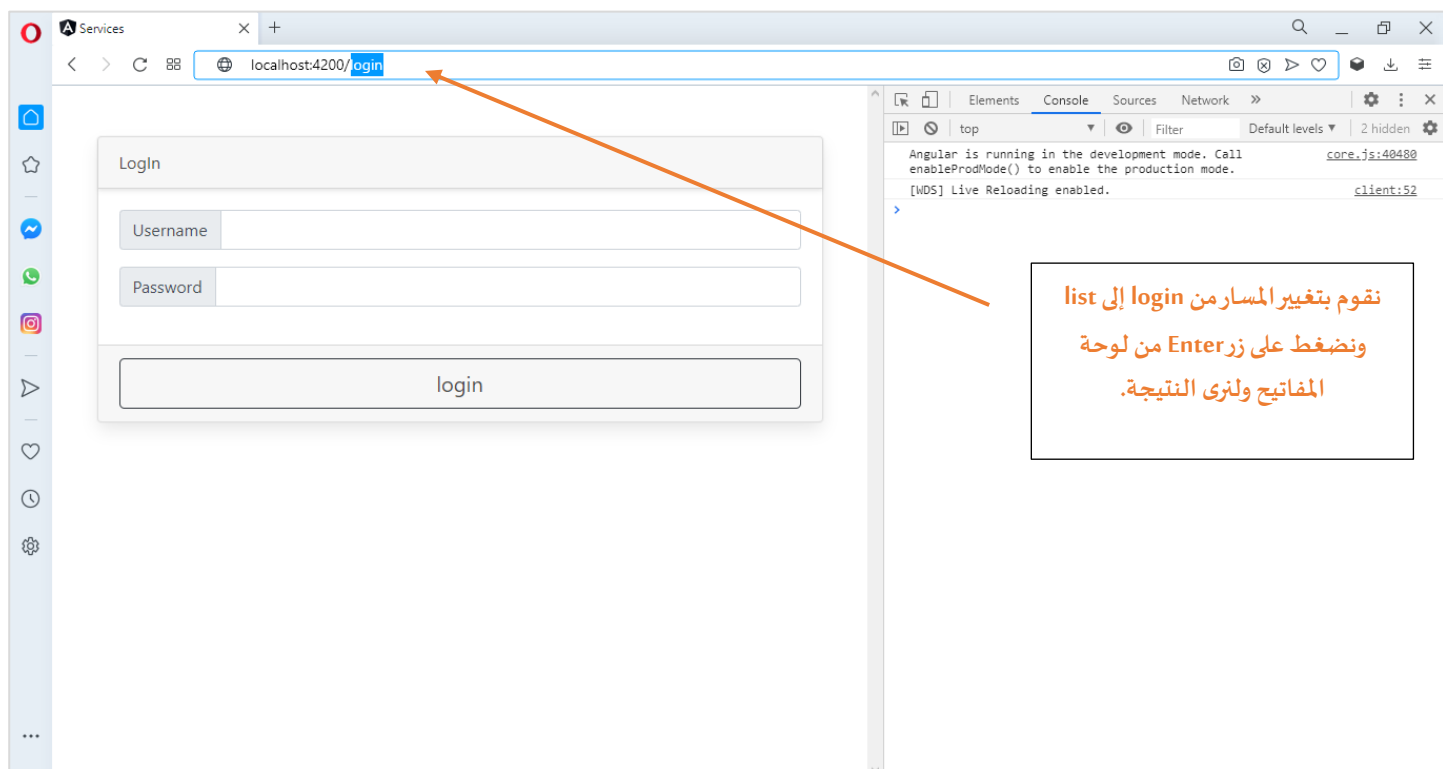


نلاحظ تم توجيهنا إلى صفحة تسجيل الدخول، الآن لنقم بتسجيل الدخول مرة أخرى باسم DivFaisal، ونرى النتيجة، كالتالي:

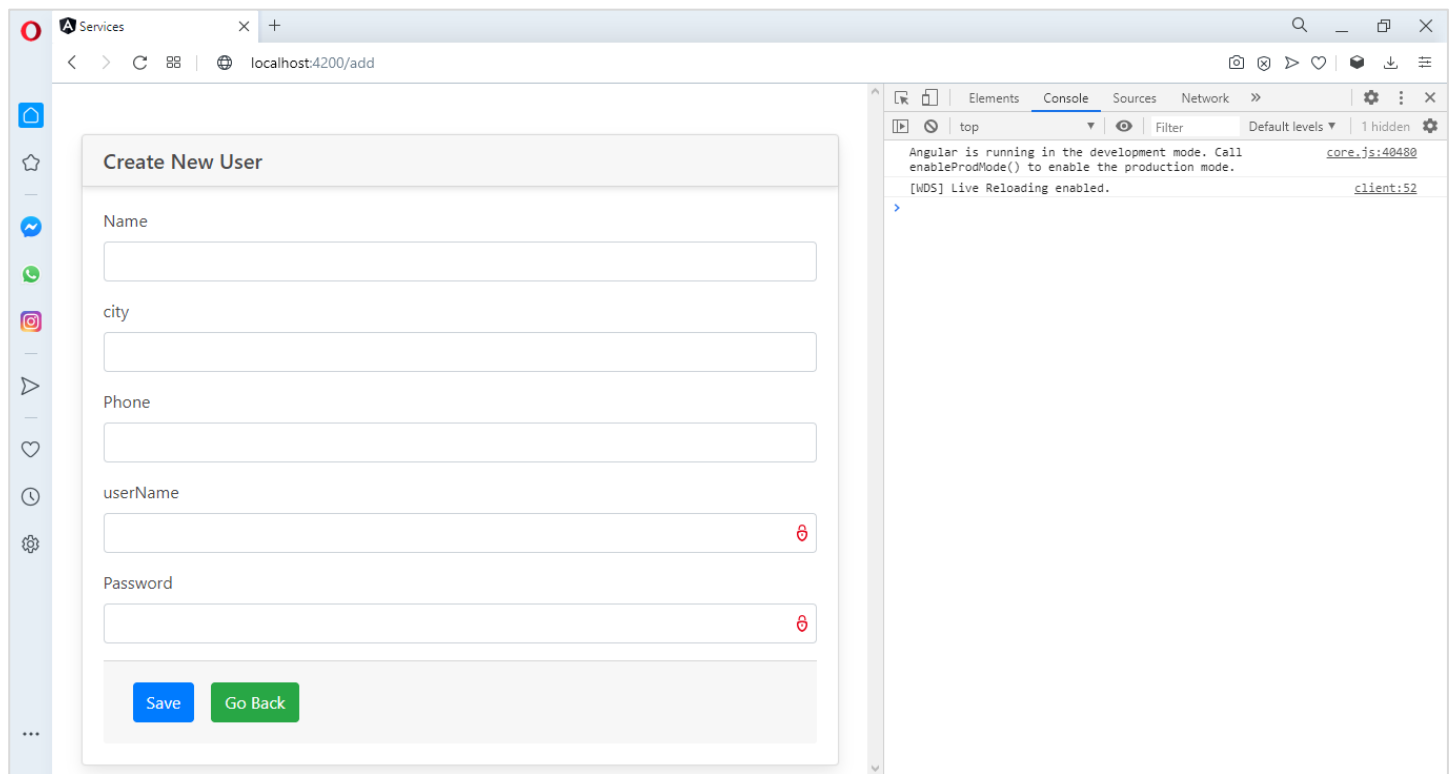


نلاحظ تم تسجيل الدخول وظهور جميع الأزرار وذلك بسبب ان قيمة الخاصية isAdmin أصبحت true، وبذلك اعطينا مثال لكيفية انشاء service تحتوي على مجموعة من الخصائص وكيفية التحكم بقيم هذه الخصائص ومشاركتها بين مجموعة من components، وتنفيذ مجموعة من المهام بناءً على هذه القيم، مع العلم انه يمكن مشاركة دوال Methods وليس فقط خصائص، وجميعها بنفس الطريقة وآليتها واحدة.

لكن في حقيقة الأمر هذا logic يحتوي على مجموعة من المشاكل، أولها لو قمنا بتغيير المسار عن طريق الرابط ولو لم نقوم بتسجيل الدخول فإنه سينقلنا إلى صفحة قائمة المستخدمين، كالتالي:



نلاحظ نقلنا إلى صفحة قائمة المستخدمين على الرغم من أن الخاصية isLoggedIn تساوي false وبنفس الوقت لو قمنا بتغيير المسار إلى add نستطيع الوصول إلى صفحة إضافة المستخدمين على الرغم من أن الخاصية isAdmin تساوي هي الأخرى false، وهذا بخلاف المتوقع، كالتالي:



ولدينا ايضاً مشكلة أخرى لو افترضنا أن المستخدم قام بتسجيل الدخول وبعد تسجيله لدخول لأي سبب من الأسباب تم تغيير قيمة الخاصية isLoggedIn من true إلى false، أو أصبحت الصفحة خاملة لفترة من الوقت وكتبنا logic معين يقوم بتغيير قيمة هذه الخاصية من true إلى false بشكل ديناميكي، ففي هذه الحالة وحالات أخرى فإنه وعلى الرغم من ان هذه الخاصية تتغير في service لكن باقي components التي تقرأ هذه الخاصية لا تحس بهذا التغيير، ولحل هذه المشكلة والمشكلة السابقة وغيرها من المشاكل التي قد تواجهك من هذا النوع، لابد من إيجاد آلية أو طريقة نستطيع بها مراقبة هذه المتغيرات على مستوى كامل التطبيق وفي حال حدوث أي تغير لأي سبب كان ننفذ مهمة معينة، كأن إذا قام المستخدم بتغيير المسار كما ذكرنا سابقاً، فلن يُسمح له بفتح أي صفحة مادام المتغير isLoggedIn قيمته false، وهذه الآلية هي ما يُسمى Subjects وهي ما سوف نتكلم عنه في الجزء التالي.

### 2.7.3. التواصل بين components عن طريق service بواسطة Subjects:

Subjects تنتمي إلى مكتبة Rxjs الضخمة والتي تقوم مهمتها بالأساس في التعامل مع البيانات الغير تزامنية القادمة إلى التطبيق، وهي ليست فقط للجافا سكربت وإطار العمل الخاص بها Angular وإنما يستخدمها مجموعة من اللغات مثل java أو c#، أما Subjects فتحتوي أربع أنواع هي (Subject – BehaviorSubject – ReplaySubject – AsyncSubject)، والذي يهمنا منها جميعاً هي BehaviorSubject وتقوم بمراقبة متغير معين وإرجاع آخر قيمة لهذا المتغير، ولكي تُرجع لنا هذه القيمة لابد ان نعمل لها subscribe لقناة على YouTube لكي يتم ارسال آخر التحديثات التي تتم على القناة، فهنا نمرر متغير إلى BehaviorSubject ومن ثم نعمل subscribe لهذا المتغير وعند أي تغيير يتم على هذا المتغير ننفذ مهمة معينة، ونشارك هذا المتغير مع جميع components، ولتوضيح لنقوم بإعطاء مثال، وأول خطوة نقوم بها هي تعريف متغير في ملف service على انه BehaviorSubject، ونمرر له قيمة منطقية، كالتالي:

## ملف login.service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})

export class LoginService {

  isLoggedIn = new BehaviorSubject<boolean>(false);

  constructor() { }
}
```

نلاحظ اننا حددنا نوعه على انه boolean واعطيناه قيمة مبدئية false، والآن نريد ان نراقب هذا المتغير في جميع التطبيق ونعمل له subscribe وفي حال حدوث أي تغيير ننفذ مهمة معينة، وليس هنالك أفضل مكان من ملف class لي Root Component، كالتالي:

## ملف app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { LoginService } from '../core/login.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  constructor(private loginService: LoginService, private router: Router) { }

  ngOnInit(): void {
    this.loginService.isLoggedIn.subscribe(value => {
      value ? this.router.navigate(['/list']) : this.router.navigate(['/login']);
    });
  }
}
```

نلاحظ اننا عملنا subscribe للمتغير (الخاصية) السابق وهو يُرجع لنا قيمة استقبلناها في متغير اسميته value، وهي القيمة التي نريد ان نراقبها بحيث إذا كانت true نقوم بتوجيه المستخدم إلى صفحة list وإذا تغير وأصبح false نقوم بتوجيه المستخدم إلى صفحة login، وايضاً نلاحظ اننا قمنا بكتابة هذا logic في الدالة ngOnInit وقد قلنا سابقاً ان هذه الدالة يتم تنفيذها مرة واحدة عند بداية بناء أي component، ولكن مع subscribe فالأمر يختلف، لأنه سوف يبقى يُراقب جميع التغيرات وينفذ الكود مع كل تغيير لذلك وجب التنبيه عزيزي المتعلم لكي لا يحدث لديك أي لبس في الموضوع، اما تغيير قيمة هذا المتغير تتم كالتالي:

#### ملف login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { LoginService } from '../core/login.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})

export class LoginComponent implements OnInit {

  constructor(private loginService: LoginService) { }

  ngOnInit(): void { }

  login() {
    this.loginService.isLoggedIn.next(true);
  }
}
```

نلاحظ اننا نستخدم الدالة next لكي نمرر القيمة، وهنا قمنا بتغييرها إلى true في حال قام المستخدم بالضغط على زر تسجيل الدخول، اما في حال قام المستخدم بالضغط على زر تسجيل الخروج فنقوم بتغيير القيمة من true إلى false، كالتالي:

#### ملف users-list.component.ts

```
import {
  AfterViewInit,
  ChangeDetectorRef,
  Component,
  OnInit,
  ViewChild
} from '@angular/core';
import { ApiService } from '../core/api.service';
import { User } from '../user';
import { Router } from '@angular/router';
import { FilterUsersListComponent } from './filter-users-list/filter-users-list.component';
import { FilterUsersListService } from '../core/filter-users-list.service';
import { LoginService } from '../core/login.service';

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
})

export class UsersListComponent implements OnInit, AfterViewInit {
  @ViewChild(FilterUsersListComponent) filterUsersList: FilterUsersListComponent;
  users: User[];
  constructor(
    private apiService: ApiService,
    private router: Router,
```

```

    private filterUsersListService: FilterUsersListService,
    private cdr: ChangeDetectorRef,
    private loginService: LoginService
  ) { }

  ngOnInit(): void {
    this.users = this.apiService.getAllUsers();
  }

  ngAfterViewInit(): void {
    this.filterUsersList.listFilter = this.filterUsersListService.filteredList;
    this.cdr.detectChanges();
  }

  onChangesValue(value: string) {
    this.filterUsersListService.filteredList = value;
    if (value) {
      const result = this.users.filter(user =>
        user.name.toLocaleLowerCase().indexOf(value.toLocaleLowerCase()) !== -1);
      this.users = result;
    } else {
      this.users = this.apiService.getAllUsers();
    }
  }

  editUser(id: number) {
    this.router.navigate(['./edit', id]);
  }

  viewUser(id: number) {
    this.router.navigate(['./view', id]);
  }

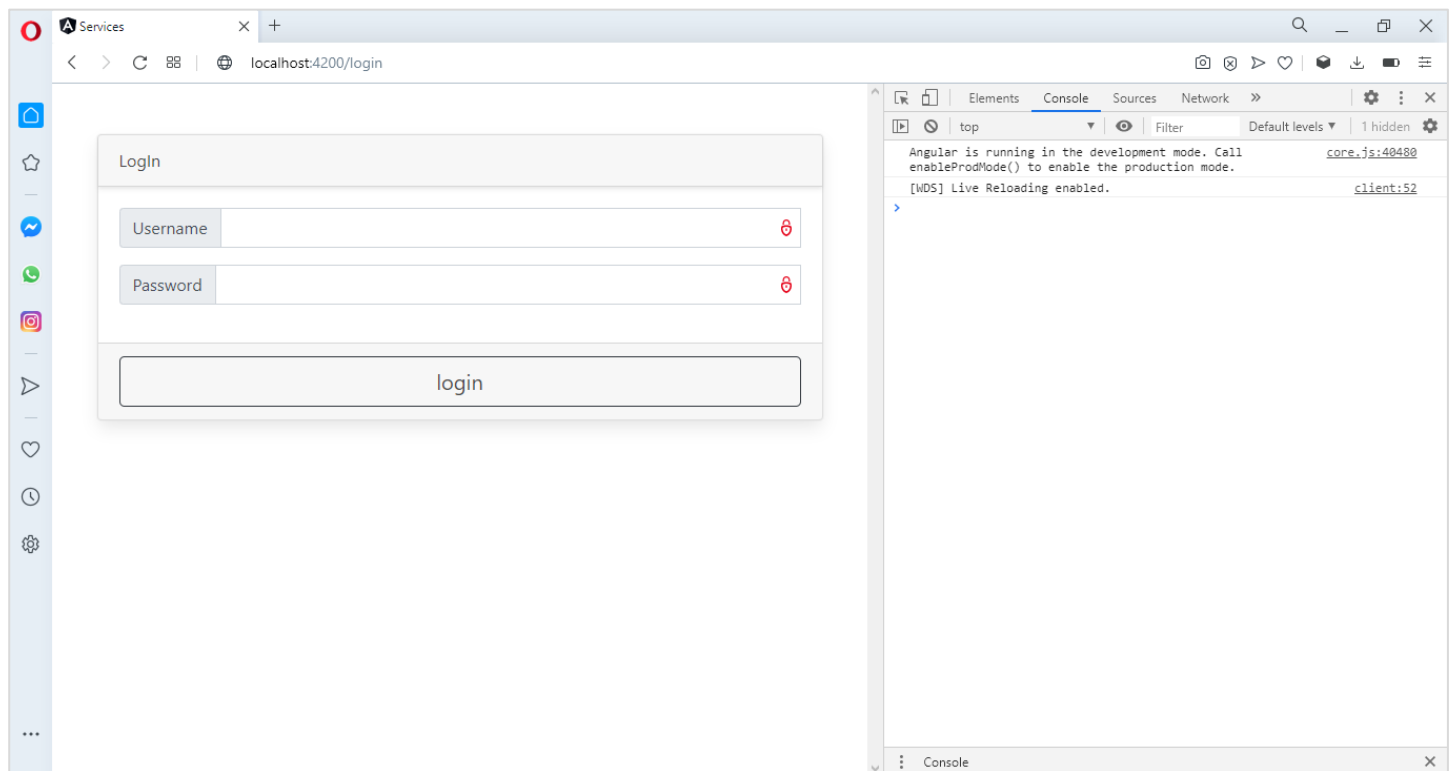
  deleteUser(id: number) {
    this.apiService.deleteUser(id);
  }

  logout() {
    this.loginService.isLoggedIn.next(false);
  }
}

```

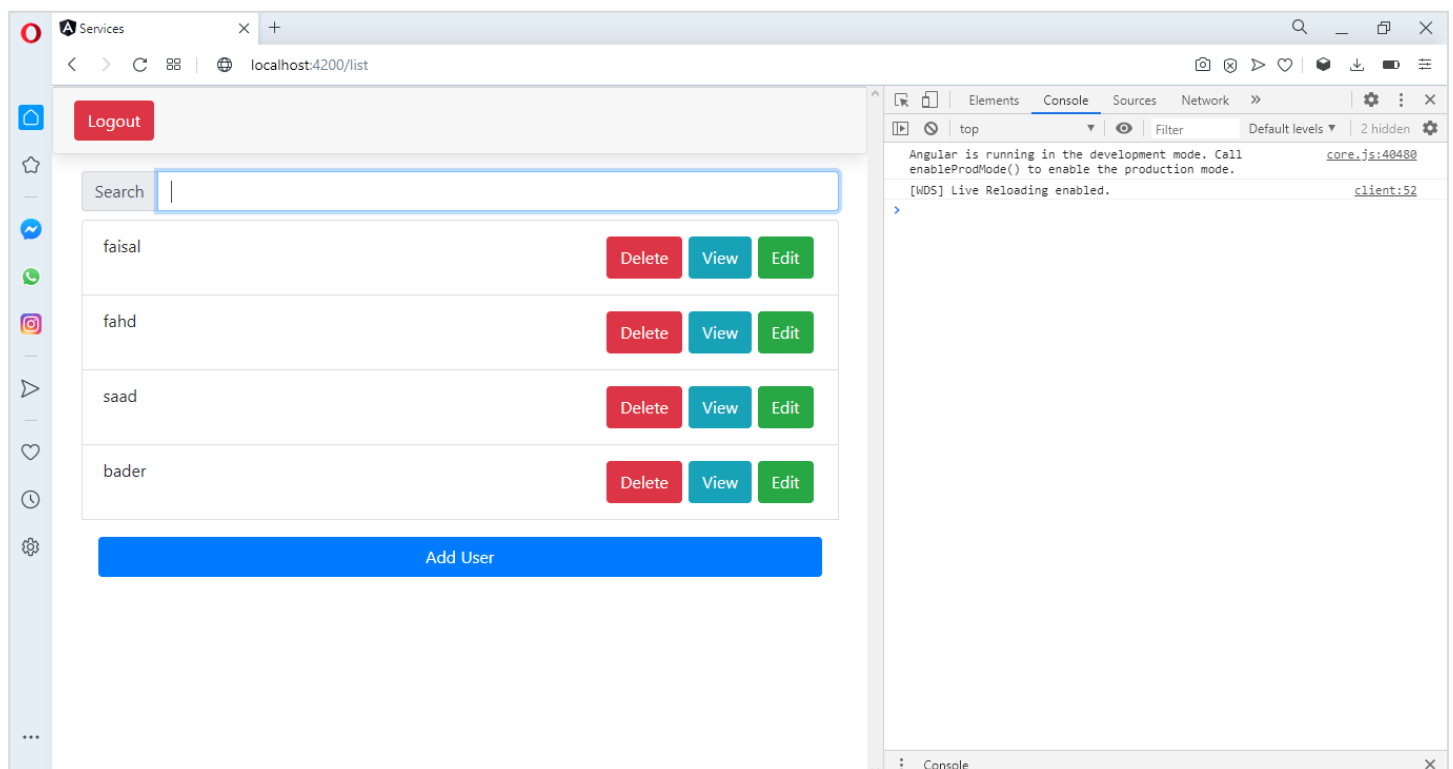


والآن لنحفظ التغييرات ونذهب إلى المتصفح ونرى النتيجة، كالتالي:



لنقوم بتغيير المسار في الرابط من login إلى list، ولنرى النتيجة:

سوف نلاحظ انه منعنا من الانتقال إلى هذه الصفحة لأن يراقب هذه القيمة ونحن قلنا له في حال كانت قيمة هذا المتغير false فانتقل إلى المسار login، الآن لنقوم بتغيير قيمة هذا المتغير إلى true عن طريق الضغط على زر login، كالتالي:



وبذلك تعلمنا كيف نستخدم Service لتواصل بين components سواء كانت بالطريقة البسيطة او عن طريق Subjects.

أما في الجزء التالي فسوف نتكلم عن جزء مهم أيضاً وهو Service Scope.

### :Services Scope .8.3

والمقصود فيه أين نريد ان يتم تنفيذ Service؟ ومتى نريد؟ ومدة صلاحيته؟

وللإجابة عن اين ومتى ومدة الصلاحية، فهي مرتبطة بمهمة هذه service فإذا كانت تنفذ فقط في component واحد فقط ولا يستفيد منها غير هذا component، ففي هذه الحالة نحذف محتويات Object ذو الاسم Injectable في ملف service ونضيفها إلى مصفوفة providers في ملف class لهذا component، ولو افترضنا ان service السابقة login.service.ts لا تُنفذ إلى في component الخاص بصفحة تسجيل الدخول، ففي هذه الحالة نحذف محتويات Object ذو الاسم Injectable، كالتالي:

```
login.service.ts ملف
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})

export class LoginService {

  isLoggedIn = new BehaviorSubject<boolean>(false);

  constructor() { }
}
```

نحذف محتويات الكائن Object السابق ليصبح بهذا الشكل، كالتالي:

```
login.service.ts ملف
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable()

export class LoginService {

  isLoggedIn = new BehaviorSubject<boolean>(false);

  constructor() { }
}
```

اما في ملف class لي component المستهدف فنضيف التالي:

```
login.component.ts ملف
import { Component, OnInit } from '@angular/core';
import { LoginService } from '../core/login.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
  providers: [LoginService]
})
```



```

}))

export class LoginComponent implements OnInit {

  constructor(private loginService: LoginService) { }

  ngOnInit(): void { }

  login() {
    this.loginService.isLoggedIn.next(true);
  }

}

```

وبذلك أصبحت هذه service مرتبطة بهذا component فقط ولا تطبق مفاهيم Singleton، وبذلك نجيب عن الأسئلة الثلاثة،

أين تُنفذ؟ في نفس component الذي اُضفناه لها.

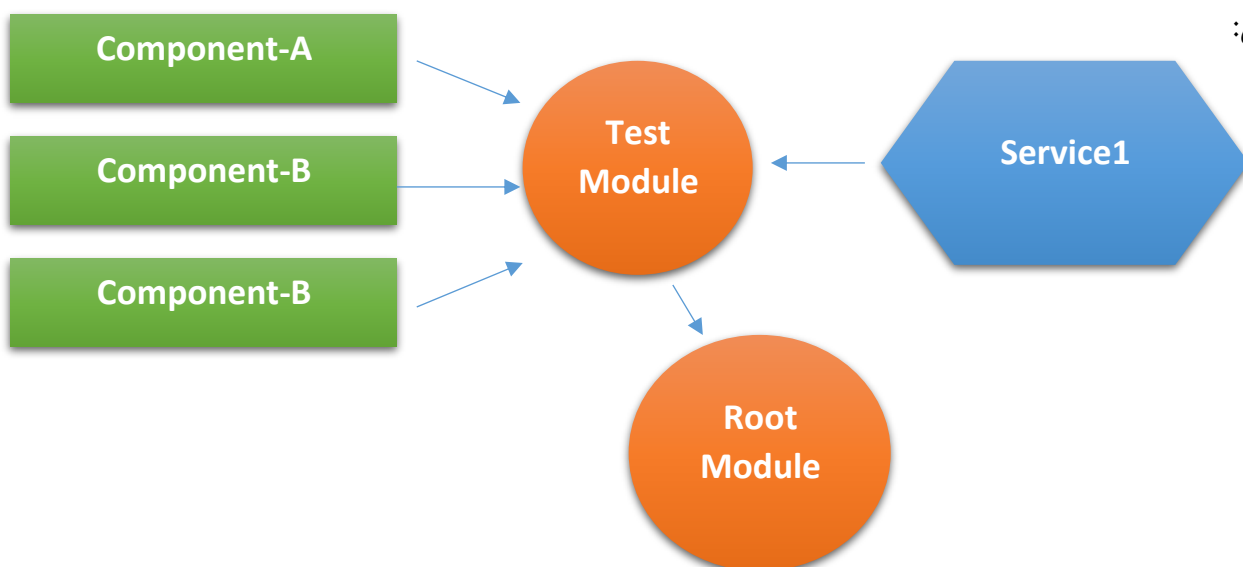
ومتى تُنفذ؟ مع بداية بناء هذا component.

ومدة الصلاحية؟ مرتبطة بمدة بقاء هذا component ومتى ما تم هدمه فإن Angular سوف يقوم بهدم Service معه.

هذا بالنسبة إلى النوع الأول، اما النوع الثاني وهو إذا كان لدينا service يتم الاستفادة منها من مجموعة من components وهذه components مرتبطة في Module فرعي واحد، فهذه الحالة أيضاً نحذف محتويات Object ذو الاسم Injectable في service وبذلك أصبحت لا تُطبق مفاهيم Singleton، ومن ثم نضيف هذه service في مصفوفة providers في ملف Module الفرعي لهذه components، وكما هو معروف ان في أي مشروع Angular هنالك Module رئيسي هو App Module أو Root Module وأي Module فرعي نقوم بإنشائه لابد من تعريفه في هذا Module، إلا في حال اردنا ان نعمل له lazy loading، ولفهم تعامل Angular مع Modules الرجاء مراجعة الكتاب الرابع من هذه السلسلة Angular Routing and Modules.

ولنفرض انه لدينا service باسم service1 ومجموعة من components (component-A – component-B – component-C) وجميع هذه components و service تم تجميعها في Module فرعي واحد وليكن اسمه على سبيل المثال test، ويمكن تمثيله

بالشكل التالي:



اما عملياً، فتكون كالتالي:

ملف service1.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()

export class LoginService {

  constructor() { }

}
```

اما ملف test، فنُعرف فيه هذه components بالإضافة إلى ملف service1، كالتالي:

ملف test.module.ts

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { componentA } from './component-A.component';
import { componentB } from './component-B.component';
import { componentC } from './component-C.component';
import { Service1 } from './service1.service';

@NgModule({
  imports: [ CommonModule ],
  declarations: [
    AppComponent,
    componentA,
    componentB,
    componentC,
  ],
  providers: [Service1],
})
export class TestModule { }
```

وأخيراً نقوم بعمل import لهذا Module في مصفوفة imports لي Root Module، كالتالي:

ملف app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { TestModule } from './test.module';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
```

```

    BrowserModule,
    AppRoutingModule,
    FormsModule,
    CoreModule,
    TestModule
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule { }

```

وبذلك أصبحت هذه service مرتبطة مع هذه components فقط، اما للإجابة عن الأسئلة الثلاثة:

أين تُنفذ؟ أي component موجود بنفس الModule الموجودة به هذه service يستطيع الوصول وتنفيذ محتوياتها.

ومتى تُنفذ؟ مع بداية التطبيق في حال ان Module الفرعي تم استدعائه في Module الرئيسي، اومتى ما تم استدعاء هذا Module الفرعي في حال تم عمل له Lazy Loading.

ومدة الصلاحية؟ من بداية بنائه إلا ان يتم انهاء التطبيق.

هذا بالنسبة لنوع الثاني، اما النوع الثالث والآخر، فهو الأكثر استخداماً وهو الوضع الافتراضي عند انشاء أي service عن طريق Angular CLI، وهو إبقاء محتويات Object ذو الاسم Injectable على وضعها الافتراضي، وتُطبق جميع مفاهيم Singleton، اما للإجابة عن الأسئلة الثلاثة، فتكون كالتالي:

أين تُنفذ؟ أي component او service في التطبيق يعمل Inject لهذه service.

ومتى تُنفذ؟ مع بداية بناء التطبيق.

ومدة الصلاحية؟ من بداية بناء التطبيق إلى أن يتم اغلاق التطبيق.

وبذلك تكون عزيزي المتعلم قد تكونت لديك ارضيه قوية لفهم services والفائدة منها وكيف يتم الاستفادة منها في التواصل مع components المختلفة، وأخيراً متى وأين ومدة صلاحية هذه services.

وبذلك أنهينا هذا الفصل وننتقل في الفصل التالي إلى مواضيع متقدمة في components.

# الفصل الرابع

مواضيع متقدمة في

Components

## 1.4. المقدمة:

في هذا الفصل سوف نستعرض بعض الجزئيات التي بعضها قد تكون مهمة ومتقدمة نوعاً ما، على سبيل المثال Change Detection و Components Dynamic.. الخ، وسوف احاول تبسيط هذه المواضيع وغيرها مع ذكر مجموعة من الأمثلة لكي تتضح الصورة لك عزيزي المتعلم، وسوف نبدأ بإذن الله بأول موضوع وهو Change Detection.

## 2.4. Change Detection:

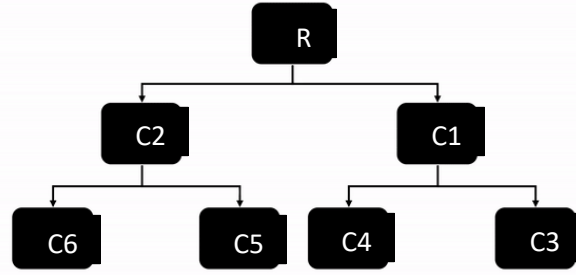
لاحظنا سابقاً ان أي تعديل على البيانات فإن Angular يراقب هذه التغيرات ويقوم بتحديث ملف template او بصيغة أدق يقوم بتحديث DOM، بناءً على التغيرات التي حدثت على هذه البيانات، مثال Property Binding او Event Binding او @Input... الخ، وجميع هذه التحديثات تتم بشكل آلي بدون أي تدخل من قبل المبرمجين، بحيث تركز مهمة المبرمج على تعريف وربط وتجهيز البيانات وتوجيهها سواء كانت بين ملفات component الواحد او بين components المختلفة، ويُترك اغلب العمل على إطار عمل Angular حيث يقوم بمراقبة هذه البيانات والتأكد من تحديث جميع عناصر DOM المرتبطة بهذه البيانات في حال حدوث أي تغيير، وهذه آلية تسمى Change Detection، وحقيقة Angular يجيد التعامل مع هذا الأمر وهو سريع جداً في مراقبة وإجراء التحديثات، وهذا النوع من Change Detection يُسمى Default، وهذا يقودنا إلى سؤال منطقي إذا هذا (نوع) فهذا معناه أن هنالك نوع آخر أو أنواع أخرى من Change Detection، وفي حقيقة الأمر هنالك نوع آخر يسمى onPush، وهذا يقودنا إلى سؤال منطقي آخر مادام النوع Default التي تتبناه Angular بشكل افتراضي لي Change Detection يقوم بعمله على اكمل وجه ويقوم بالتسهيل على المطورين من عناء كتابة Logic المعقد في مراقبة التعديلات وتحديث DOM، فما الفائدة من وجود النوع الآخر onPush؟ وللإجابة عن هذا الأمر نحتاج إلى فهم كيفية عمل النوع Default لي Change Detection في Angular، وهذه الآلية معقدة ولن أدخل في تفاصيلها وإنما سوف أحاول تبسيطها على قدر المستطاع فيما يخدم ما نريد توضيحه هنا.

يمكن تلخيص خطوات عمل Change Detection، من خلال النقاط التالية:

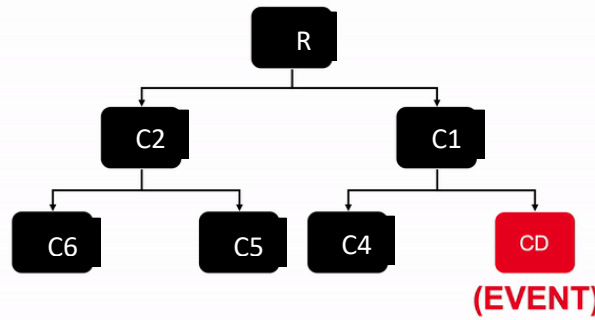
- ١) بعدما يقوم المطور بتجهيز الربط بين components المختلفة وبين ملفات class و template لكل component واتجاه وسير البيانات بين جميع أجزاء التطبيق المختلفة، فإن Angular يقوم بمراقبة البيانات (سواء كانت بيانات خارجية قادمة من قاعدة بيانات او بيانات داخلية على مستوى التطبيق فقط) وعند حدوث أي تعديل فإن Angular سوف يعثر على هذه التعديلات.
- ٢) Angular بشكل افتراضي يقوم برسم جميع components بشكل شجري، لذلك سوف يقوم بعمل check لكل component في شجرة components، من الأعلى إلى الأسفل بمعنى من Root Component إلى آخر component أبناً. ويقارنها بالبيانات المخزنة لديه في شجرة View (Angular يقوم برسم View مشابه DOM وعند حدوث أي تعديل يقارن هذا View بالDOM حيث ان View هو الذي يُخزن فيه القيم الجديدة والDOM هو الذي يُخزن فيه البيانات القديمة).

- ٣) وعند وجود أي قيم جديدة في View فإنه يقوم بتحديث DOM للمستخدم لعرض البيانات له.

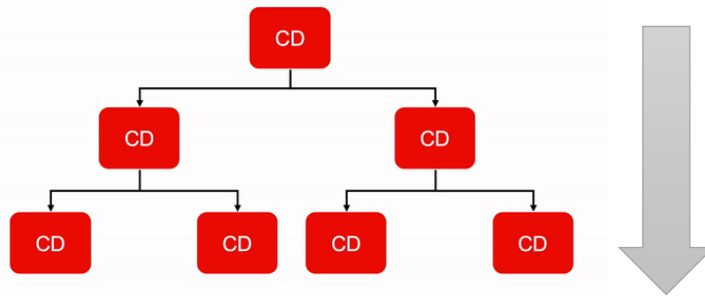
ويمكن تمثيل ما قيل سابقاً في الأشكال التالية: (حيث R يمثل Root Component و C يمثل كل component)



نلاحظ الشكل السابق يمثل شجرة components لجميع components الموجودة في التطبيق حيث يقوم Angular برسمها على شكل شجري، وعند حدوث أي تعديل في أحد هذه component وليكن هذا التعديل على شكل Event من قبل المستخدم (ضغط زر - حركات مؤشر الفأرة - كتابة نص في مربع النص - ... الخ)، كما في الشكل التالي: (حيث CD هو اختصار Change Detection)



وكما هو واضح نظام Change Detection عثر على هذا التعديل، لذلك سوف يقوم بعمل Check على جميع components من الأعلى إلى الأسفل ويقارن هذه component بما لديه في شجرة View، كالتالي:



لذلك لك ان تتخيل عزيزي المتعلم لو أنه لدينا المئات من components وبعض الأحيان قد تصل إلى الآلاف منها، فكم من الوقت يحتاجه Angular في عمل check لجميع components في حال حدوث أي تعديل ولو كان في جزئية بسيطة من التطبيق، صحيح انني قلت سابقاً انه سريع جداً، ولكن الثانية الواحدة قد تفرق في اداء التطبيق لديك، وصحيح ايضاً انه في التطبيقات المتوسطة إلى الصغيرة يُفضل استخدام هذا النوع Default، ولكن في التطبيقات الكبيرة والضخمة فتحتاج منا ان نقول لي Angular قف رجاءً وقم بتحديث هذا component فقط وبعض component المرتبطة به، ولا تقم بعمل check لجميع component في التطبيق، وهذه الآلية تُسمى onPush.

وللمعلومية في حال مررنا إلى component معين كائن أو مصفوفة (كما هو معروف الكائن والمصفوفة هما mutable) وليكن على سبيل المثال عن طريق (`@Input()`) فإن Angular يقوم بعمل check على جميع أجزاء هذا الكائن ولو كان التعديل في جزء واحد فقط، ولتوضيح لنفرض انه كان لدينا هذا الكائن المعقد والمتداخل ما بين مصفوفة وكائنات، كالتالي:

```
obj = {
  "problems": [{
    "Diabetes": [{
      "medications": [{
        "medicationsClasses": [{
          "className": [{
            "associatedDrug": [{
              "name": "asprin",
              "dose": "",
              "strength": "500 mg"
            }],
            "associatedDrug#2": [{
              "name": "somethingElse",
              "dose": "",
              "strength": "500 mg"
            }]
          }],
          "className2": [{
            "associatedDrug": [{
              "name": "asprin",
              "dose": "",
              "strength": "500 mg"
            }],
            "associatedDrug#2": [{
              "name": "somethingElse",
              "dose": "",
              "strength": "500 mg"
            }]
          }],
        }],
      }],
    }],
    "labs": [{
      "missing_field": "missing_value"
    }],
    "Asthma": [{}],
  }],
}
```

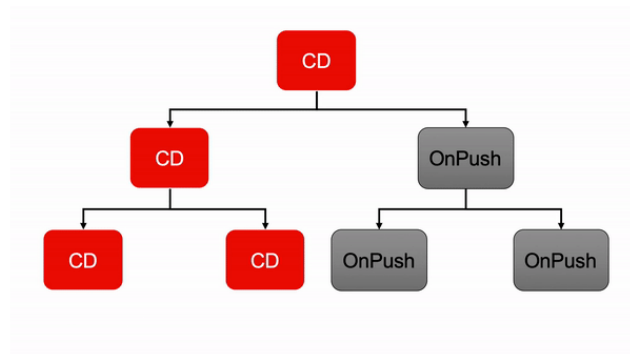
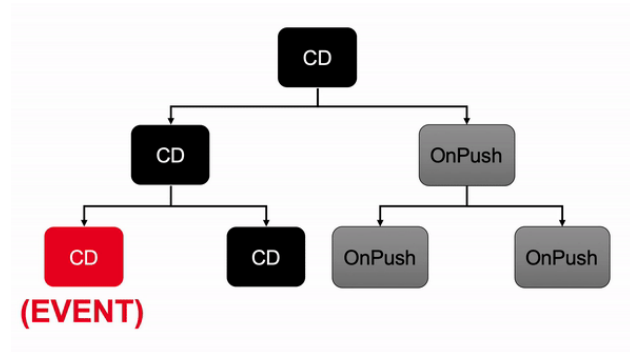
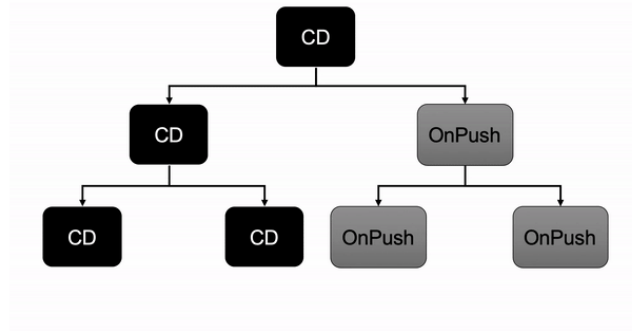
ولنفرض اننا نريد تغيير الخاصية name من القيمة aspirin إلى Panadol، فنستطيع ذلك عن طريق كتابة الأمر التالي:

```
this.obj.problems[0].Diabetes[0].medications[0].medicationsClasses[0].className[0].associatedDrug[0].name = 'Panadol'
```

وهل تعلم ماذا سوف يعمل نظام Default في Change Detection سوف يقوم بعمل check لجميع هذه الخصائص في الكائن والقيم الخاصة به، مع العلم اننا فقط قمنا بتغيير خاصية واحدة، طبعاً المثال السابق ذو قيم قليلة وقد لا يؤثر على أداء التطبيق لديك ولكن ماذا لو كان لدينا ألوف او حتى عشرات الألوف من القيم ففي هذه الحالة والحالات السابقة، إذا كنا بصدد بناء تطبيق Angular كبير فنحتاج إلى إعادة التفكير في Default وتجربة النوع الثاني `onPush`.

إذن السؤال الذي قد يترادد إلى ذهنك عزيزي المتعلم، ما هو هذا النوع؟ وماذا يقدم لنا من ميزات؟

هذا النوع نضيفه إلى ملف class لي component المستهدف، وعند وضعه كأننا نقول لي Angular لا تقم بتطبيق نظام Change Detection الافتراضي الذي لديك على هذا component وجميع components الأبناء له ان وجدت، ونستطيع تمثيله بهذا الشكل:



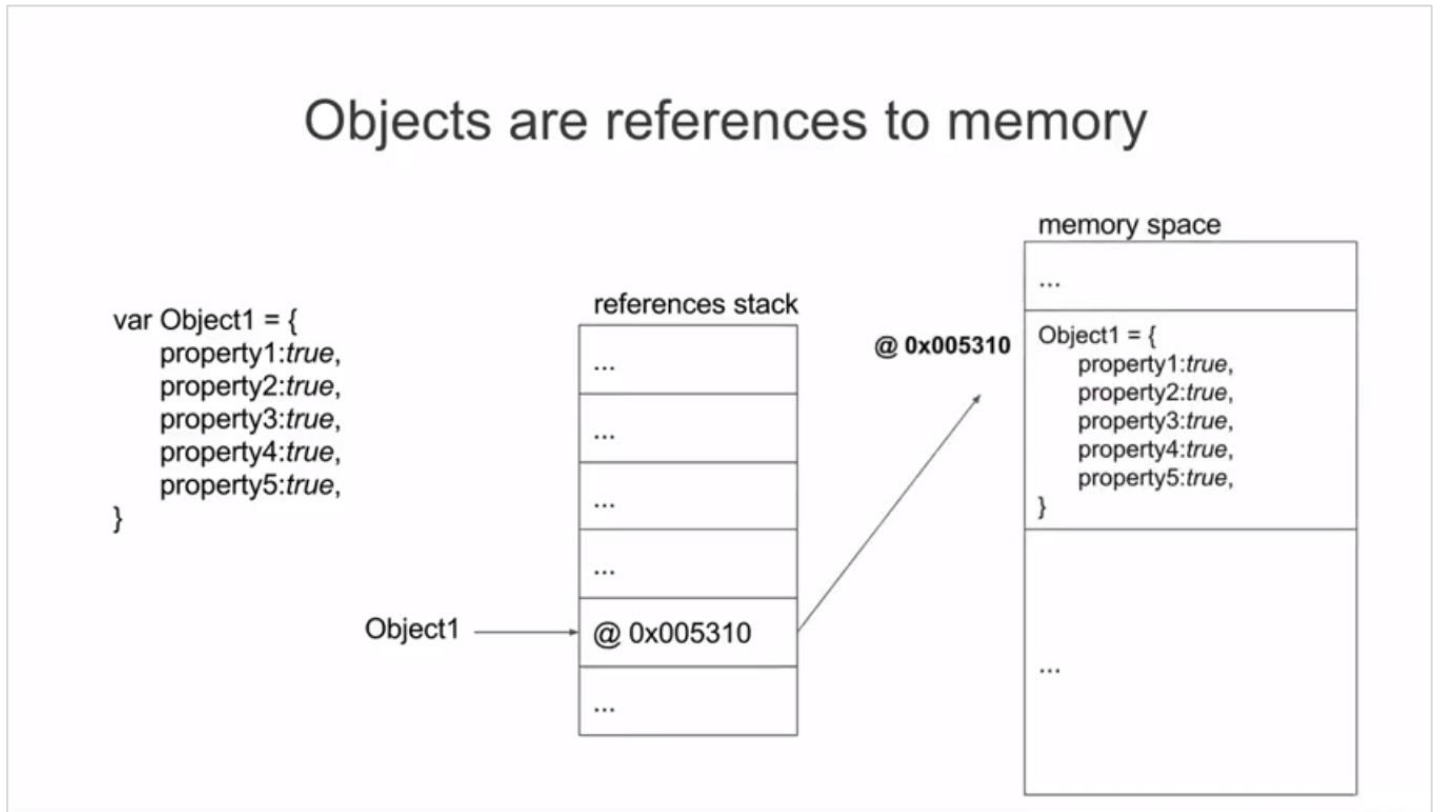
ويجب التوضيح انه باستخدامنا لهذا النوع فليس معناه ان Angular لا يقوم بعمل checks وانما يقوم بعمل check ولكن في الحالات التالية:

- ١) إذا تم تغيير Reference للقيمة الداخلة إلى component.
  - ٢) إذا تم تغيير أحد أحداث Event الخاصة بالDOM مثل (onclick-oninput-onmouseenter-... etc.) لي component المستهدف وجميع components الأبناء ان وجدت.
  - ٣) عمل check يدوياً باستخدام service ذات الاسم ChangeDetectorRef.
  - ٤) عندما تقوم (async pipe) بإرسال قيم جديدة.
- وسوف نستعرض هذه الحالات مع ذكر مثال على كل نوع.

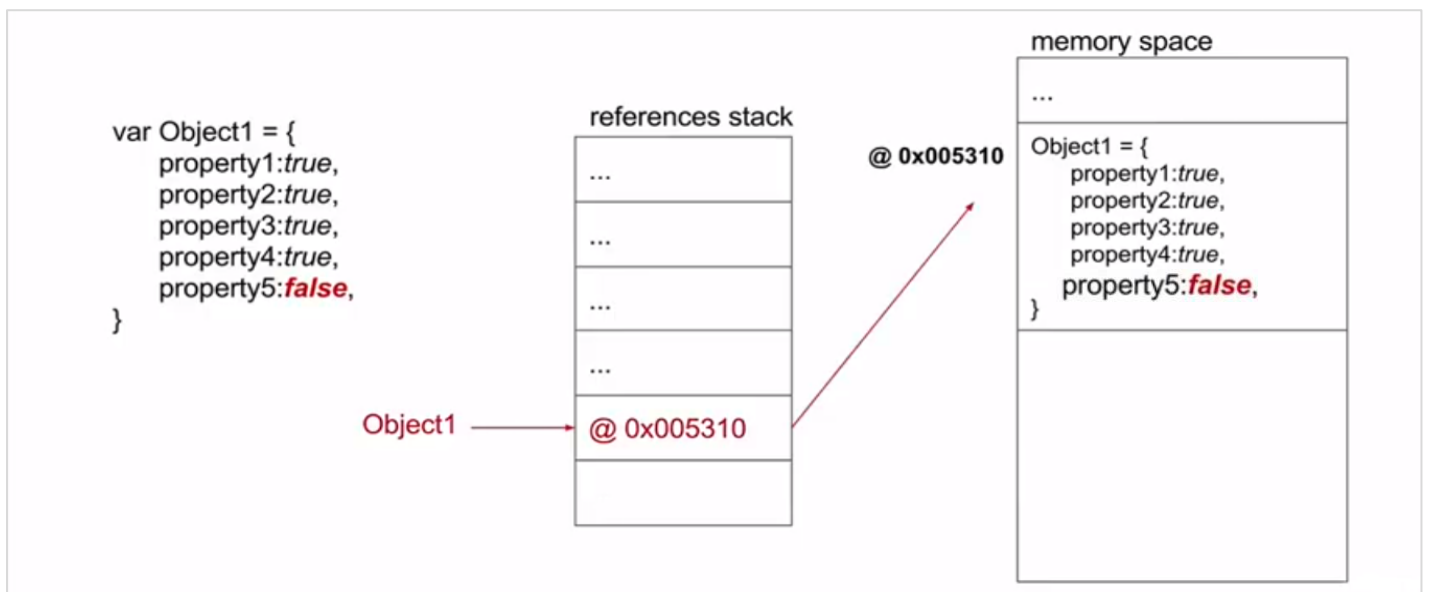


#### 1.2.4. Input Reference Changes :

الكائنات والمصفوفات تعتبر mutable أي قابلة للتغيير بعكس الأنواع الأخرى مثل الأرقام والنصوص والقيم المنطقية تعتبر immutable أي غير قابلة للتغيير، بمعنى ان المصفوفات او الكائنات عند تعريفها في JavaScript فانه يتم وضع لها مرجع Reference يشير إلى هذا الكائن في الذاكرة RAM وعند تغيير قيمة أي خاصية في هذا الكائن فإن هذه القيمة هي التي تتغير ولا يتغير Reference الخاص بهذا الكائن، لذلك هذا النوع يسمى mutable أي قابل للتغيير، ويمكن تمثيله في الشكل التالي:



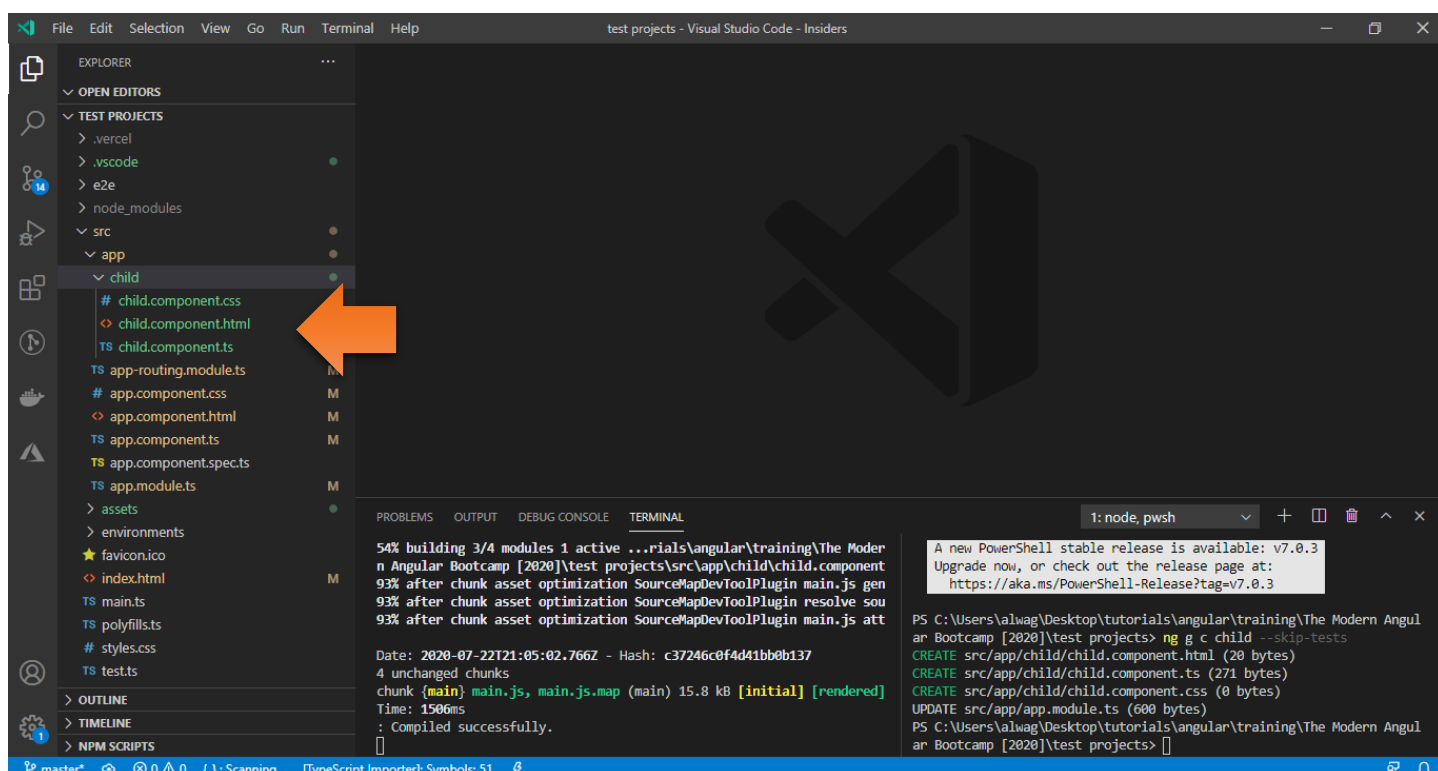
نلاحظ ان الكائن الذي عرفناه باسم Object1 اخذ Reference (@0x005310) بحيث أصبح يشير إلى هذا الكائن في الذاكرة RAM عن طريق هذا المرجع بمعنى انه أصبح كأنه عنوان لهذا الكائن في الذاكرة، ولو قمنا بتغيير أي قيمة من قيم الخصائص التي يحتويها هذا الكائن فإنه يتم تغيير الخاصية فقط ولا يتم تغيير المرجع الخاص بهذا الكائن، كما في الشكل التالي:



نلاحظ عن تغيير القيمة للخاصية ذات الاسم property5 من true إلى false فإنه سوف يتم تغيير القيمة لهذه الخاصية ولن يتم تغيير المرجع الخاص بها، بعكس الأنواع التي تعتبر immutable مثل النص string او الرقم number او القيمة المنطقية boolean، وغيرها من الأنواع الأخرى، فهذه الأنواع تعتبر غير قابلة للتعديل وعند تعديل قيمها فإنه سوف يتم إعطائها رقم مرجعي آخر مختلف عن الأول، فمثلاً لو قمنا بتعريف متغير على انه نص x = "angular" فإنه سوف يتم إعطائه رقم مرجعي Reference في الذاكرة مثله مثل الكائن والمصفوفة ولكن عند تغيير هذه القيمة لنفس المتغير كأن تكون x = "react" ففي هذه الحالة سوف يتم إعطائه رقم مرجعي Reference آخر يختلف عن الأول، وعند إرجاع القيمة "angular" x = "angular" فسوف يتم إرجاع الرقم المرجعي القديم وهكذا.

ولكن السؤال المهم هو كيف استفيد من هذه المعلومات واقوم بتوظيفها في Change Detection وخصوصاً النوع onPush، وفي الحقيقة نستفيد مما قلناه سابقاً في أنه في حال استخدامنا لنوع onPush فإن نظام Change Detection لا يلاحظ إلا المتغيرات التي تم تغيير Reference لها بمعنى انه سوف يعمل Detected للأنواع التي تندرج على انها immutable فقط، اما الأنواع التي تعتبر mutable فإن نظام Change Detection في Angular لن يلاحظ وجودها نهائياً، والسبب في ذلك عزيزي المتعلم انه لو رجعت إلى المثال السابق لوجدت اننا ذكرنا ان Angular في الوضع الافتراضي يقوم بعمل check على الكائن وجميع الخصائص الفرعية الموجودة بداخله، وقد ذكرنا ان هذا الامر في التطبيقات الكبيرة قد يقلل من أداء التطبيق لديك لذلك عند استخدام النوع onPush كأننا نقول لي Angular قم فقط بعمل check فقط في حال تغيير المرجع، وعند عدم تغيير المرجع فتجاهل الأمر تماماً، لذلك في حال اردنا ان نمرر كائن او مصفوفة من النوع mutable إلى component معين عن طريق @Input()، فنحتاج إلى تغيير Reference لهذا الكائن.

ولنكتفي من السرد النظري المطول، ولنقوم بالتطبيق العملي، لذلك قم عزيزي المتعلم بإنشاء مشروع Angular جديد واضف فيه component جديد وليكن اسمه child، بحيث يصبح بالشكل التالي:



ولنضيف selector الخاص بهذا component إلى ملف template في Root Component، لكي يصبح هو الأب، كالتالي:

ملف app.component.html

```
<app-child></app-child>
```

الآن لنقم بتعريف الكائن في المثال السابق في ملف class في Root Component وبنفس الوقت لننشئ دالة تقوم بتغيير قيمة إحدى خواص هذا الكائن وليكن اسم الدالة changeValue، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {
  obj = {
    problems: [{
      Diabetes: [{
        medications: [{
          medicationsClasses: [{
            className: [{
              associatedDrug: [{
                name: 'asprin',
                dose: '',
                strength: '500 mg'
              }],
              'associatedDrug#2': [{
                name: 'somethingElse',
                dose: '',
                strength: '500 mg'
              }],
            }],
          }],
          className2: [{
            associatedDrug: [{
              name: 'asprin',
              dose: '',
              strength: '500 mg'
            }],
            'associatedDrug#2': [{
              name: 'somethingElse',
              dose: '',
              strength: '500 mg'
            }],
          }],
        }],
      }],
    },
    labs: [{
```

```

        missing_field: 'missing_value'
      }]
    }],
    Asthma: [{}]]
  }
};

constructor() { }

ngOnInit(): void { }

changeValue() {
  this.obj.problems[0].
    Diabetes[0].
    medications[0].
    medicationsClasses[0].
    className[0].
    associatedDrug[0].
    name = 'Panadol';
}
}

```

الآن لنقم بإنشاء زر في ملف template لي Root Component بحيث إذا تم الضغط عليه يتم تنفيذ هذه الدالة، وبنفس الوقت لنقوم بإرسال الكائن obj إلى component الأب، كالتالي:

ملف app.component.html

```

<button (click)="changeValue()">Change Value</button>
<app-child [obj]="obj"></app-child>

```

الآن لنقم باستقبال هذا الكائن في component الأب، كالتالي:

ملف child.component.ts

```

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})

export class ChildComponent implements OnInit {

  @Input() obj: any;

  constructor() { }

  ngOnInit(): void { }
}

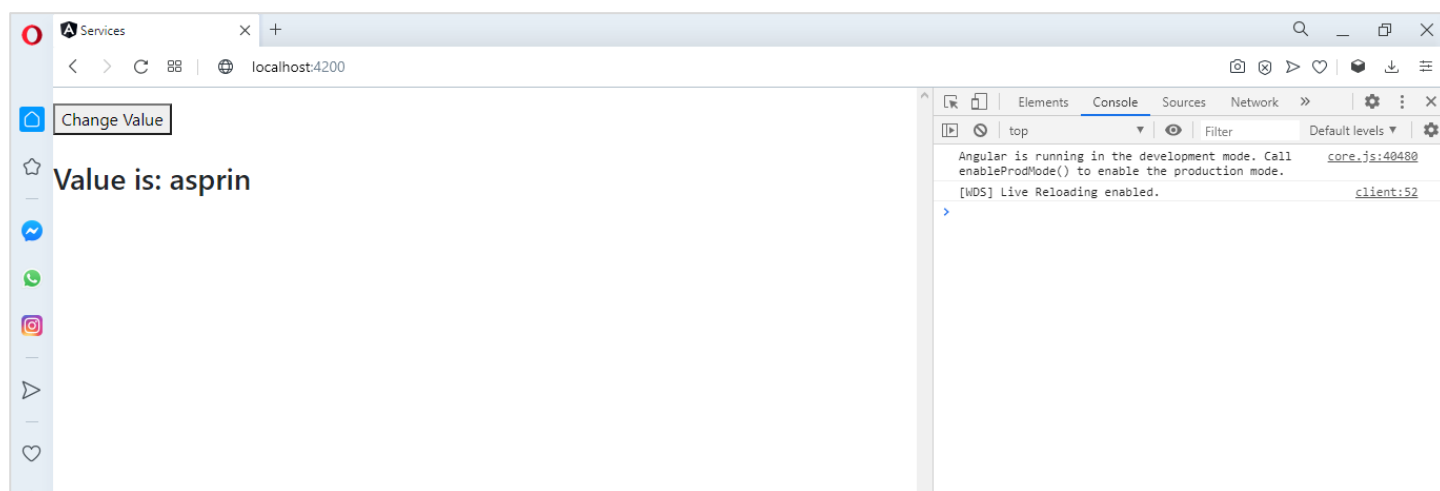
```

نلاحظ اننا قمنا باستقبال الكائن في متغير اسميته obj، ومن ثم سوف نقوم بعمل bind لهذا الكائن في ملف template لهذا component الأب، كالتالي:

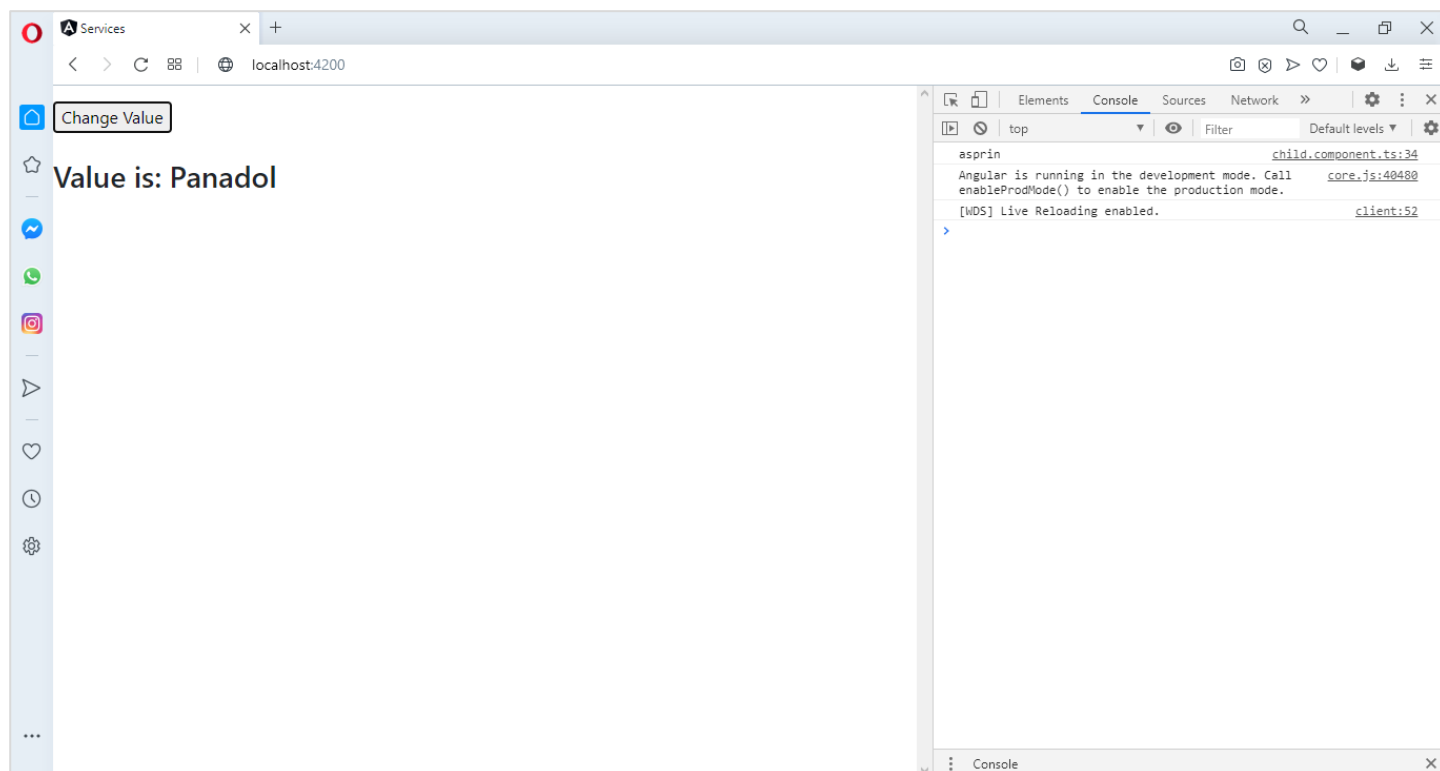
ملف child.component.html

```
<br />
<br />
<h3>
  Value is:
  {{ this.obj.problems[0].Diabetes[0].medications[0].medicationsClasses[0]
    .className[0].associatedDrug[0].name }}
</h3>
```

أخيراً لنقم بحفظ التعديلات ونذهب إلى المتصفح ونرى النتيجة، كالتالي:



نلاحظ ظهرت لنا القيمة الأساسية asprin، الآن لنضغط على الزر ونرى النتيجة:



نلاحظ النظام الافتراضي لي Change Detection في Angular قام بالكشف ان هنالك تغير في هذا الكائن وقام بعمل check لجميع أجزاء هذا الكائن وقام بتحديث القيمة المحددة، ولكن ماذا لو قمنا بتغيير النوع من Default إلى OnPush، كالتالي:

```
ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})

export class ChildComponent implements OnInit {

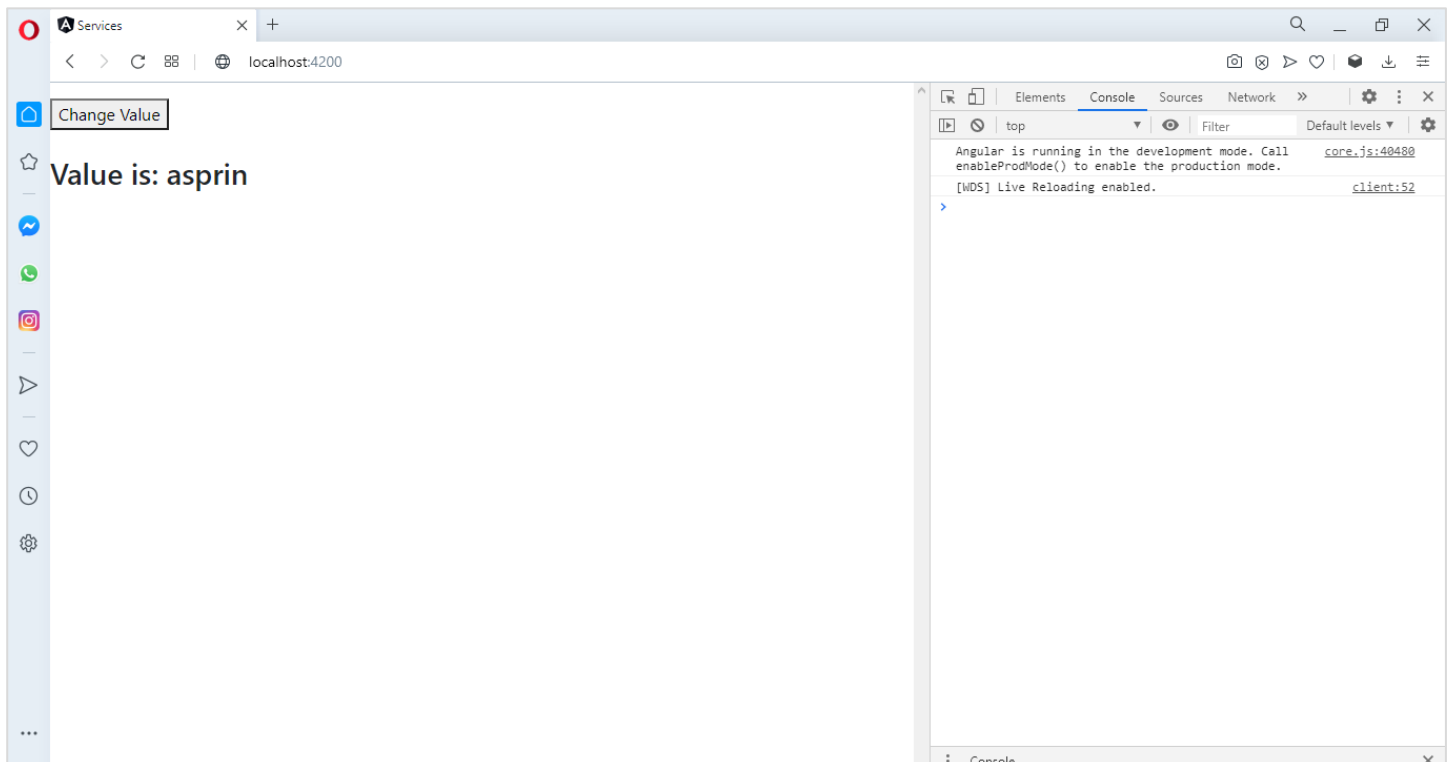
  @Input() obj: any;

  constructor() { }

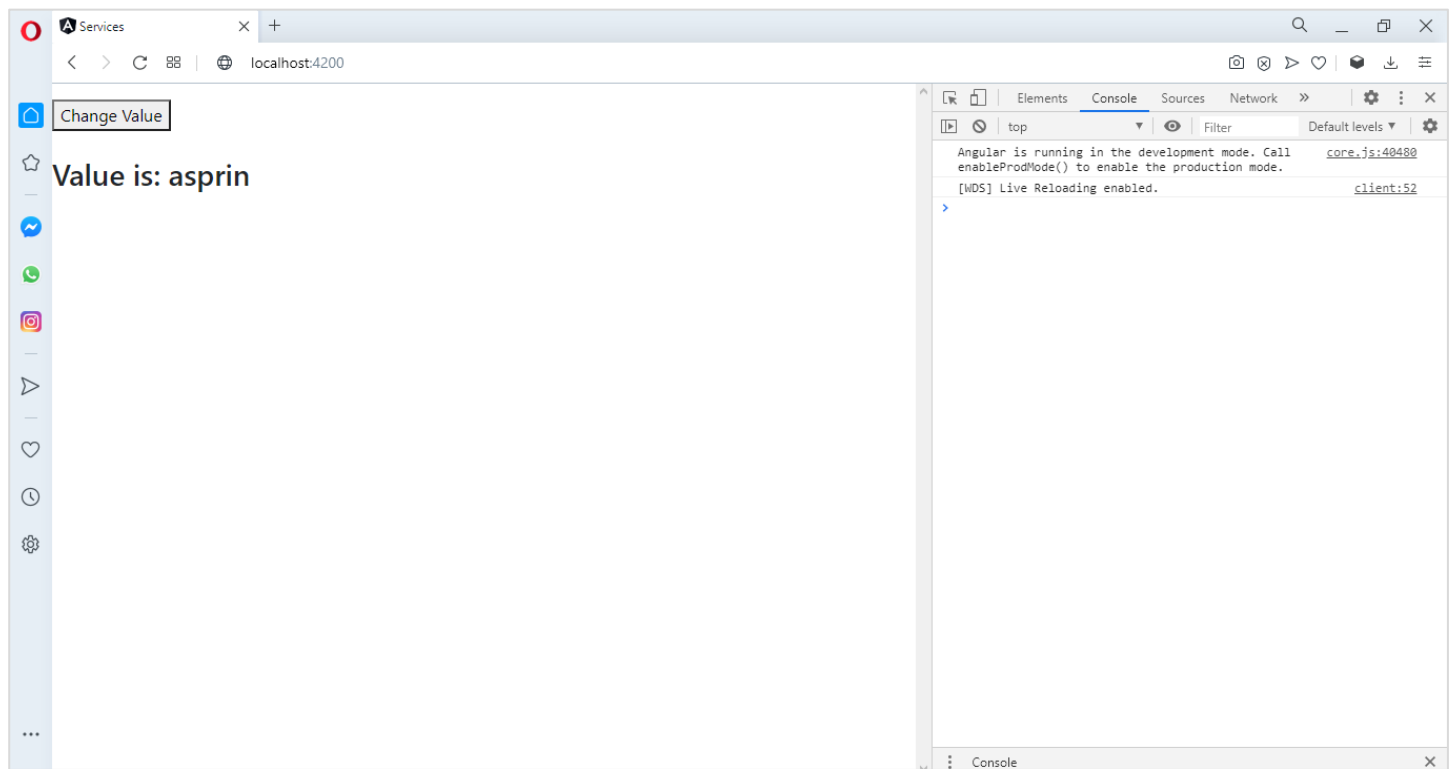
  ngOnInit(): void { }

}
```

الآن لنقم بحفظ التعديلات ونذهب إلى المتصفح ونرى النتيجة، كالتالي:



ولنضغط على الزر ولنرى النتيجة:



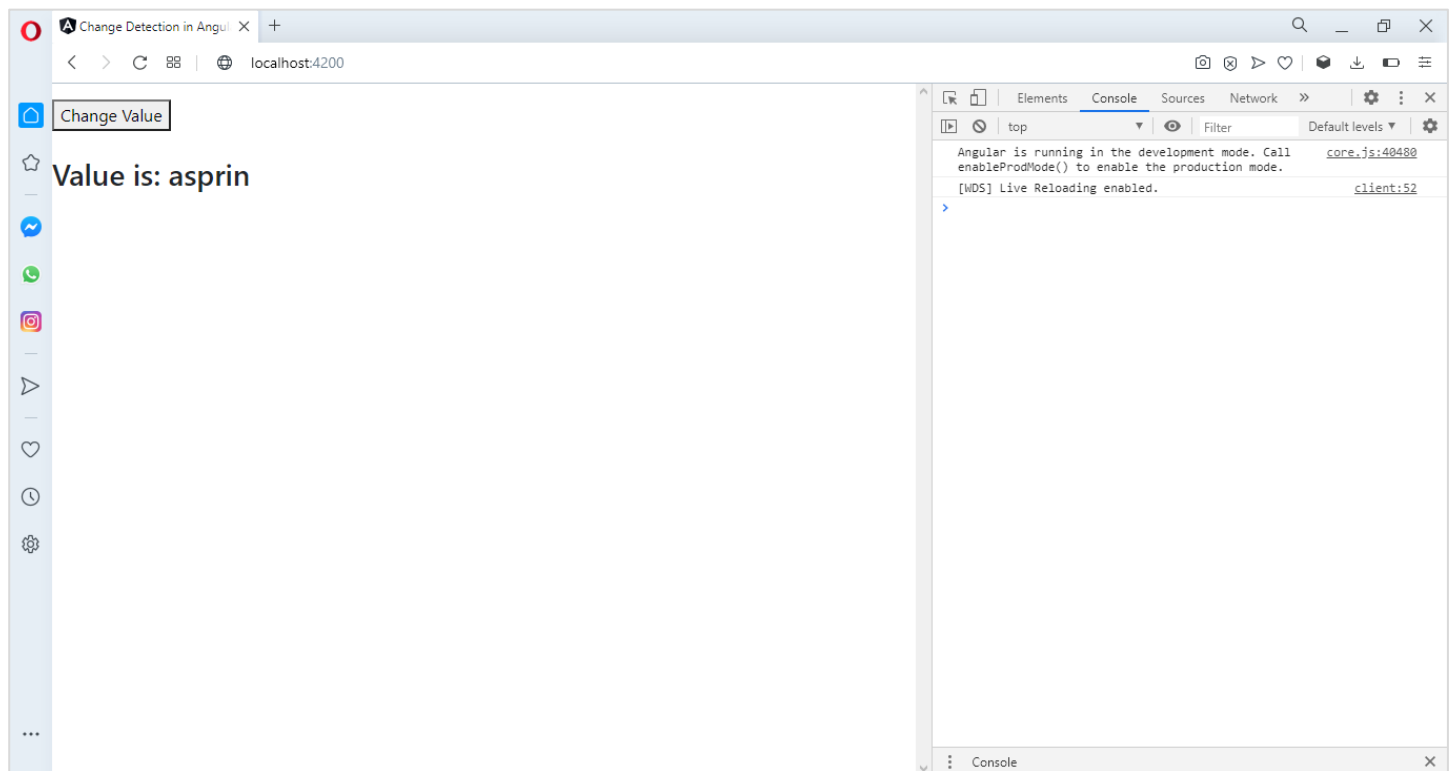
سوف نلاحظ ان Change Detection لم يكتشف هذا التغيير على الرغم من ان القيمة في الكائن تغيرت لكن نظام Change Detection لم يقوم بتحديث ملف template لعرضها للمستخدم، والسبب كما ذكرنا سابقاً انه في النوع OnPush يلاحظ فقط عند تغير References للكائن او المصفوفة.

لذلك سوف نقوم بتغيير Reference لهذا الكائن، وهناك طرق متعددة للقيام بهذا الأمر ابسطها هو كتابة الأمر التالي في الدالة changeValue في ملف class لي Root Component، كالتالي:

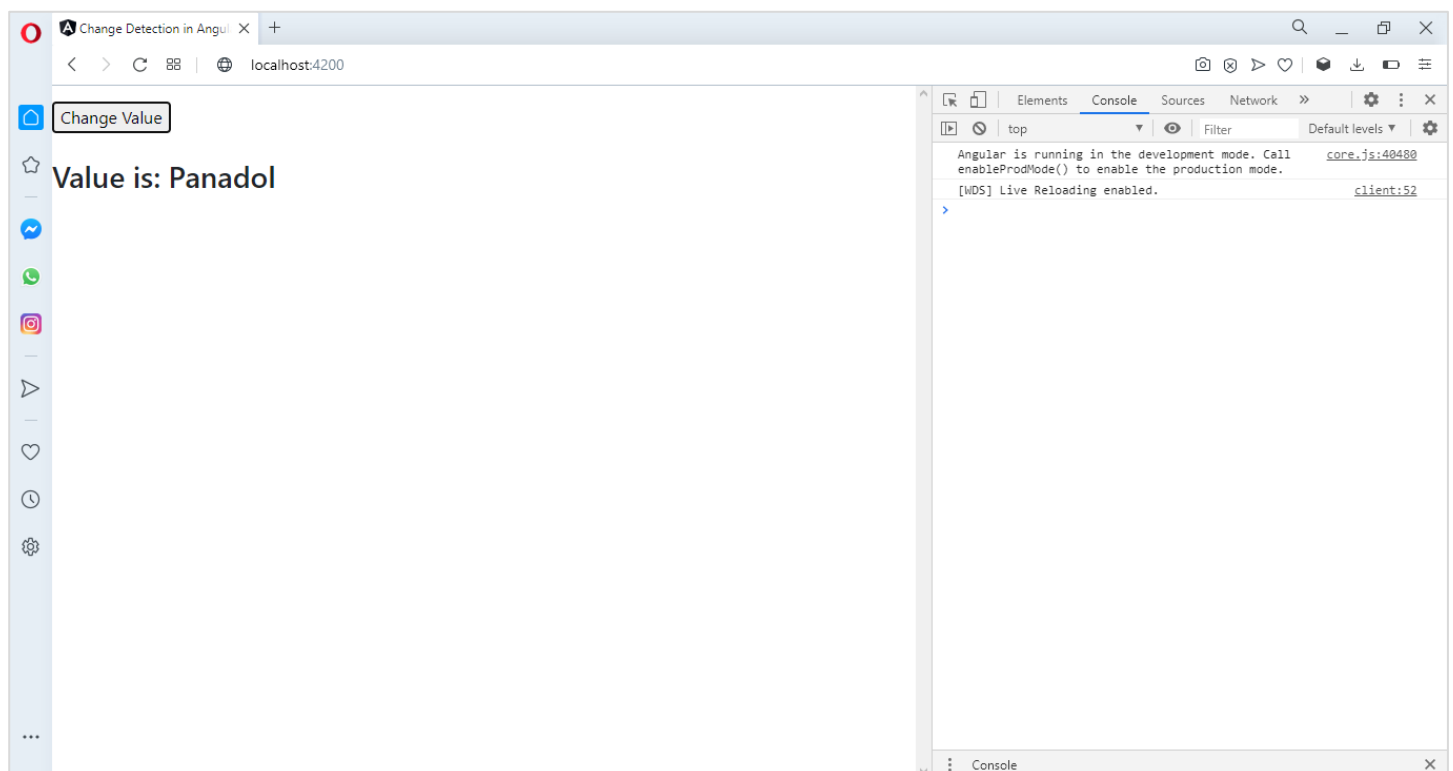
```
جزء من ملف app.component.ts

changeValue() {
  this.obj = Object.assign({}, this.obj);
  this.obj.problems[0].
    Diabetes[0].
    medications[0].
    medicationsClasses[0].
    className[0].
    associatedDrug[0].
    name = 'Panadol';
}
```

الآن لنذهب إلى المتصفح ونقوم بالتجربة بعدما كتبنا هذا الأمر، كالتالي:



لنضغط على الزر ونرى النتيجة:



نلاحظ النوع OnPush في نظام Change Detection اكتشف التعديل وقام بتحديث ملف template للمستخدم بناءً على البيانات الجديدة.

اما الأنواع التي هي immutable فلا تحتاج منّا إلى أي عمل إضافي فهي تعمل مع OnPush بدون أي مشاكل لأن أي قيمة جديدة نضيفها لهذه الأنواع فإنه سوف يقوم بتغيير Reference الخاص بها، ولتأكيد لنقوم بتعريف متغير على انه نص string في ملف class لي Root Component ونمرر هذه الخاصية إلى الأبن، كالتالي:



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

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
```

```
export class AppComponent implements OnInit {
```

```
  str = 'Faisal';
```



```
  obj = {
    problems: [{
      Diabetes: [{
        medications: [{
          medicationsClasses: [{
            className: [{
              associatedDrug: [{
                name: 'asprin',
                dose: '',
                strength: '500 mg'
              }],
              'associatedDrug#2': [{
                name: 'somethingElse',
                dose: '',
                strength: '500 mg'
              }]
            }]
          }],
          className2: [{
            associatedDrug: [{
              name: 'asprin',
              dose: '',
              strength: '500 mg'
            }],
            'associatedDrug#2': [{
              name: 'somethingElse',
              dose: '',
              strength: '500 mg'
            }]
          }]
        }],
        labs: [{
          missing_field: 'missing_value'
        }],
        Asthma: [{}]
      }]
    }];
  };
};
```

```

constructor() { }

ngOnInit(): void { }

changeValue() {

    this.str = 'Fahd';

    this.obj = Object.assign({}, this.obj);
    this.obj.problems[0].
        Diabetes[0].
        medications[0].
        medicationsClasses[0].
        className[0].
        associatedDrug[0].
        name = 'Panadol';
}
}

```

اما في ملف template فنقوم بتغيير التالي:

ملف app.component.html

```

<button (click)="changeValue()">Change Value</button>
<app-child [obj]="obj" [str]="str"></app-child>

```

ونستقبل هذا المتغير str في component الأب، كالتالي:

ملف child.component.ts

```

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})

export class ChildComponent implements OnInit {

  @Input() obj: any;
  @Input() str: string;

  constructor() { }

  ngOnInit(): void { }
}

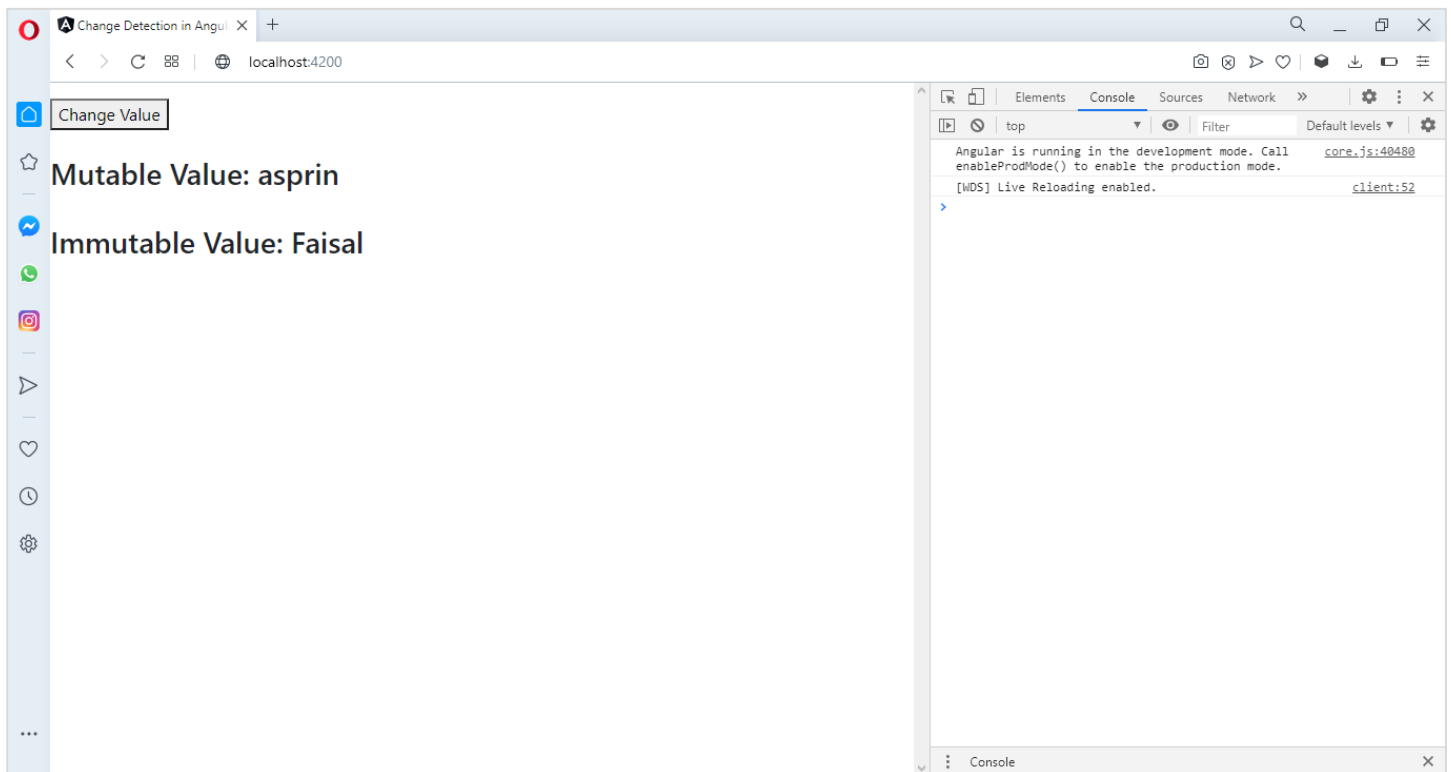
```

وأخيراً نعمل bind لهذا المتغير في ملف template، كالتالي:

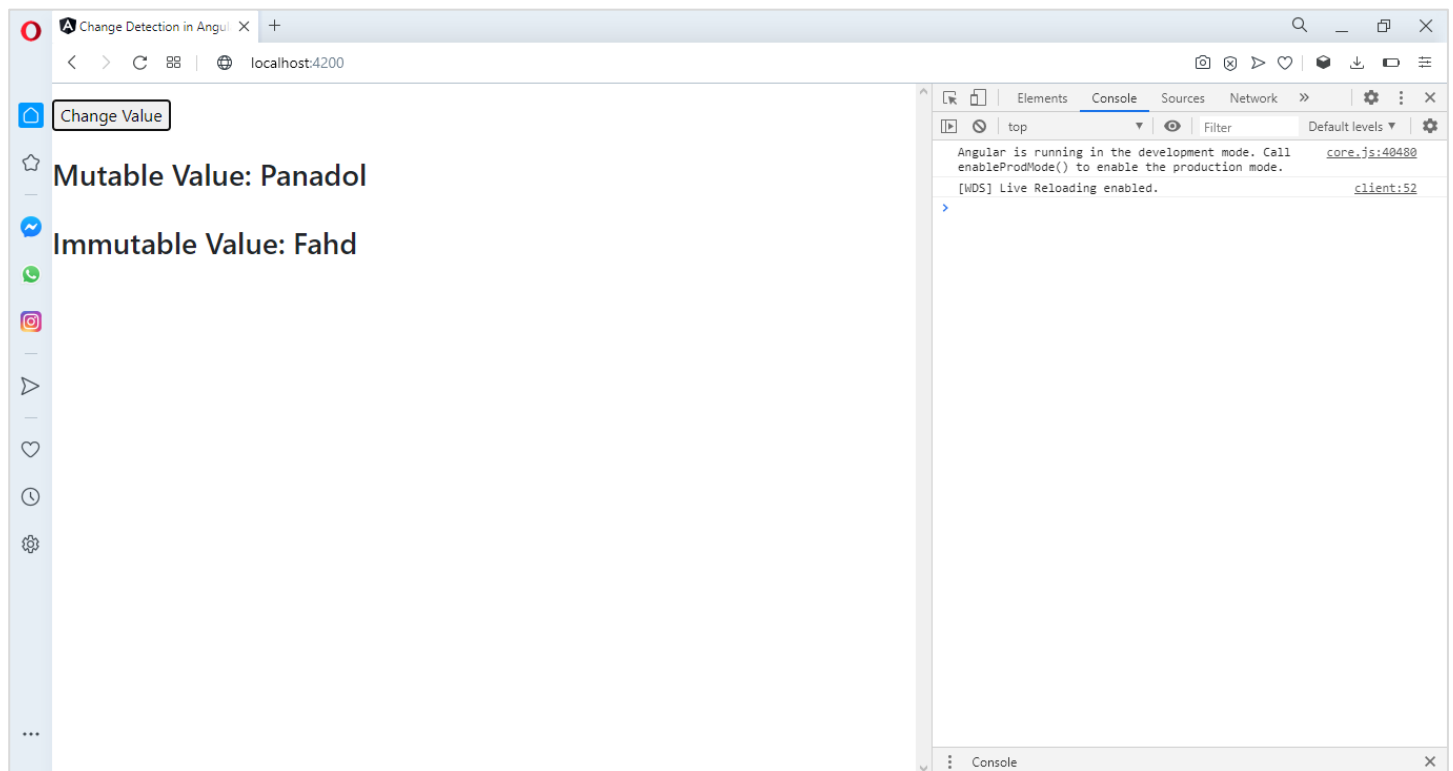
ملف child.component.html

```
<br />
<br />
<h3>
  Mutable Value:
  {{
    obj.problems[0].Diabetes[0].medications[0].medicationsClasses[0]
      .className[0].associatedDrug[0].name
  }}
</h3>
<br />
<h3>Immutable Value: {{ str }} </h3>
```

الآن لنشاهد النتيجة، في المتصفح:



الآن لنقوم بالضغط على الزر، ونرى النتيجة:



نلاحظ الوضع OnPush اكتشف التغيير في النوع string لأنه Immutable وقام بتحديث ملف template.

ولتوضيح هنالك طريقة أخرى في حال أردنا ان لا نغير Reference للكائن او المصفوفة، وذلك عن طريق عمل Check يدوياً، وسوف نتكلم عنه لاحقاً بإذن الله.

ملاحظة: في حال تمرير نوع immutable ونوع mutable فإن Angular سوف يقوم بالكشف على التغييرات الحاصلة على الكائن بشرط ان يتم تغيير قيمة النوع immutable في نفس الحدث او الدالة التي تُجري التغيير على قيم النوع mutable، بمعنى بما اننا في المثال السابق قمنا بتمرير متغير نصي ففي هذه الحالة لا نحتاج إلى ان نقوم بتغيير Reference الخاص به، وبنفس الوقت كلا التغييرين يتمان في حدث واحد وهو click (الضغط على الزر) وفي دالة واحدة وهي ChangeValue().

#### 2.2.4 Event Handler Is Triggered

وهنا يُقصد فيها أن النوع OnPush يكتشف أي حدث Event من أحداث DOM، مثل الضغط على زر الفأرة او حركة المؤشر وغيرها من الأحداث الأخرى، ولتوضيح في المثال السابق نلاحظ انه عند الضغط على زر في component الأب ان هذا الحدث اكتشفه OnPush، لذلك منعاً لتكرار فنكتفي بالمثال السابق.

#### 3.2.4 Async Pipe

النوع OnPush يقوم بالكشف عن جميع البيانات الغير تزامنية Asynchronous في حال استخدامنا لي Pipe ذو الاسم async. ولتوضيح لنعطي مثال، وسوف نقوم بتمثيل بيانات غير تزامنية عن طريق Observable وBehaviorSubject، مع العلم انني سوف أطبق المثال على نفس المشروع السابق مع حذف جميع الأكواد فيه، كالتالي:

ملف app.component.ts

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

```

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  stream: BehaviorSubject<{}> = new BehaviorSubject<{}>({ name: 'Faisal' });
  user = this.stream.asObservable();

  constructor() { }

  ngOnInit(): void { }

  changeName() {
    this.stream.next({ name: 'Fahd' });
  }
}

```

اولاً يجب عليك ان تعلم عزيزي المتعلم انه لا يتوجب عليك فهم الأسطر البرمجية في الأعلى، وانما اعرف اننا فقط قمنا بتخليق بيانات وهمية على شكل كائن وقمنا بإسناده إلى user على شكل Observable، لنحاكي البيانات الغير تزامنية.

اما الدالة فتقوم بتغيير القيمة للكائن للخاصية name، وسوف نمرر هذا المتغير user إلى component الأب، كالتالي:

ملف app.component.html

```

<button (click)="changeName()">Change Name</button>
<app-child [user]="user"></app-child>

```

وكما جرت العادة سوف نستقبل هذا المتغير في component الأب، كالتالي:

ملف child.component.ts

```

import { Component, OnInit, Input, ChangeDetectionStrategy } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent implements OnInit {
  @Input() user;
  constructor() { }
  ngOnInit(): void { }
}

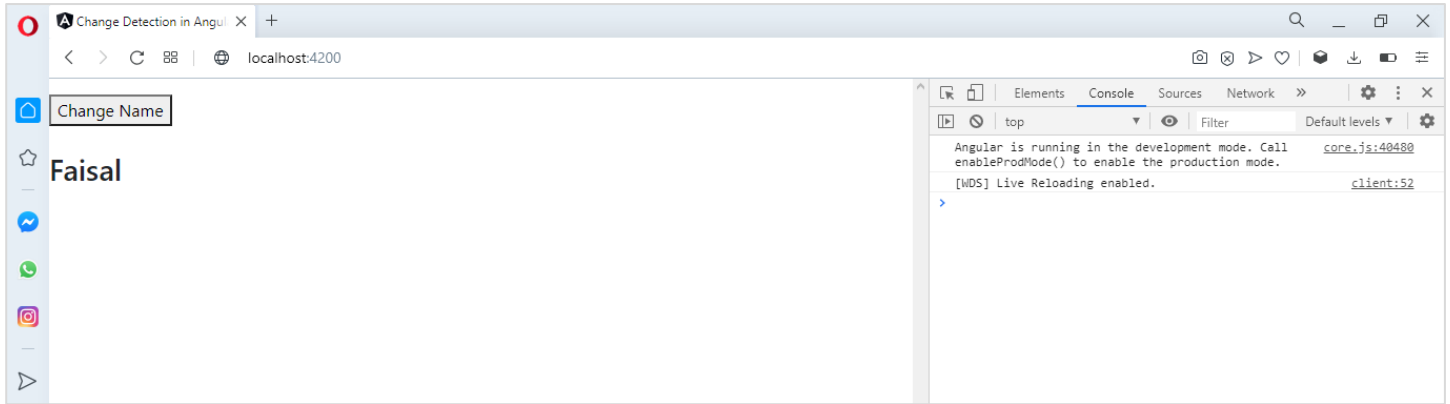
```

وأخيراً نعمل bind لهذا الكائن user إلى ملف template مستخدمين الـ pipe ذو الاسم async لكي نستطيع عرض البيانات، كالتالي:

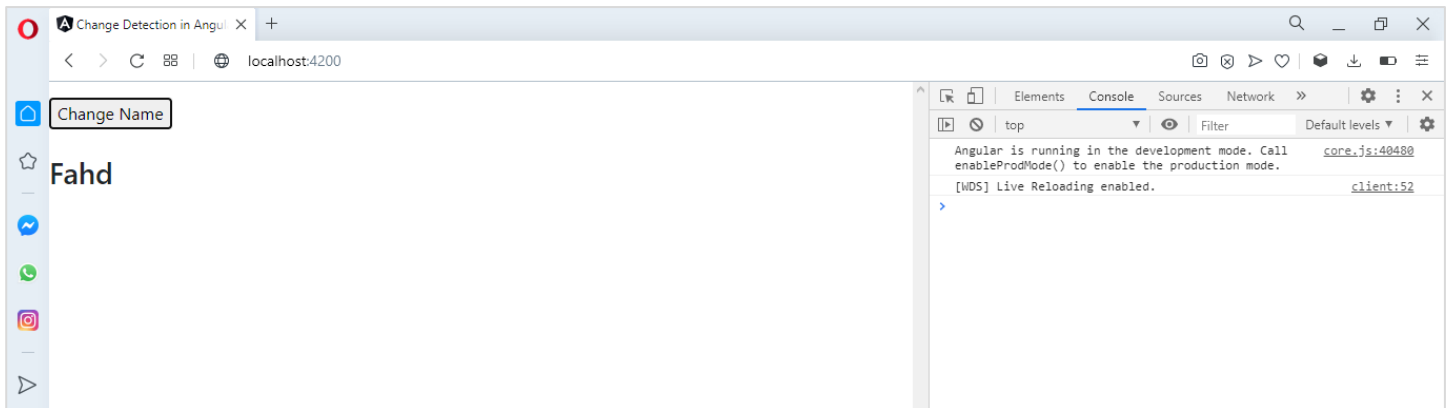
ملف child.component.html

```
<br />
<br />
<h3> {{ (user | async).name }} </h3>
```

ولنرى النتيجة، في المتصفح، كالتالي:



ولنضغط على الزر ولنرى النتيجة:



نلاحظ ان النوع OnPush أكتشف التغيير وهو بذلك مشابهه لي Default في async pipe، وما يميز async pipe انها تقوم بعمل subscription بشكل تلقائي ولا نحتاج ان نقوم بها يدوياً.

#### 4.2.4. Manual check and change detection

يقدم إطار العمل Angular للمطورين القدرة على التحكم بنظام Change Detection بشكل يدوي، سواء بإيقاف عمل check نهائياً او بعمله مره واحدة عند تنفيذ سطر او أسطر برمجية، ... الخ، عن طريق service جاهزة اسمها ChangeDetectorRef وتحتوي بداخلها على مجموعة من الدوال Methods، التي تعطي القدرة على التحكم بوقت وكيفية عمل check من عدمه، وسوف نتطرق لبعض هذه الدوال مع إعطاء امثلة لكلاً منها بإذن الله.

#### 1.4.2.4. الدالتين detach وdetectChanges

الدالة detach تُستخدم في منع نظام Change Detection بنوعيه (Default – OnPush) من عمل check او الكشف عن أي تغييرات عن component معين او مجموعة من components، ومن استخداماتها على سبيل المثال لو كان لدينا مجموعة من البيانات كأن تكون قائمة ضخمة من بيانات مستخدمين في (Dashboard) لموقع تواصل اجتماعي ضخمة يحتوي على ملايين

من المستخدمين بحيث كلما قام مستخدم جديد بتسجيل بياناته يتم إضافة بيانات هذا المستخدم في القائمة في Dashboard في نفس الوقت، فلك ان تتخيل اولاً كم عدد المستخدمين الذين يقومون بالتسجيل بنفس اللحظة وتُضاف بياناتهم في القائمة وايضاً لك ان تتخيل كم component موجود في هذا Dashboard ولك ان تتخيل مع النوع Default كم مرة سوف يقوم نظام Change Detection بالعمل على تحديث جميع components مع كل إضافة لبيانات مستخدم جديد او عندما يقوم المستخدم بتعديل بياناته، او بالنسبة لنوع OnPush يتم تحديث component الخاص بعرض قائمة المستخدمين مع جميع components الأبناء ان وجدت، وهذا كله سوف يرهق التطبيق ويقلل من كفاءته واداءه.

ولجميع هذه الأسباب وغيرها طُرح مجموعة من الحلول منها استخدام النوع OnPush وب نفس الوقت نستخدم الدالة detach لمنع Change Detection من العمل نهائياً لهذا component وجميع الأبناء المرتبطة به، ومن ثم في حال وصل لدينا مثلاً بيانات ١٠٠ مستخدم نقوم بتفعيل الدالة الثانية detectChanges يدوياً لتشغيل النظام Change Detection وتحديث ملف template لعرض البيانات لمدير التطبيق في Dashboard.

ولتوضيح لنعطي مثال، لنفرض انه لدينا مؤقت يرسل كل جزء من الثانية جزء من الوقت لذلك سوف نستخدم detach لكي نمنع نظام Change Detection من العمل ومن ثم نستخدم الدالة detectChanges لكي يفعل الكشف عن التغييرات إذا أصبح الوقت ثانية، بمعنى كل ثانية نقوم بتفعيل نظام Change Detection يدوياً، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  ChangeDetectorRef,
  DoCheck,
  ChangeDetectionStrategy
} from '@angular/core';
import { interval, Subscription } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})

export class AppComponent implements OnInit, DoCheck {

  time: number;
  timeSubscription: Subscription;
  constructor(private ref: ChangeDetectorRef) {
    this.ref.detach();
  }

  ngDoCheck(): void {
    console.log(this.time);
  }
}
```

1

2

### 3

```
ngOnInit(): void {
  interval(10).pipe(
    map(time => time * 10)
  ).subscribe(time => {
    this.time = time;
    if (this.time % 1000 === 0) {
      this.ref.detectChanges();
    }
  });
}
```

(1) قمنا بعمل inject لي service ذات الاسم ChangeDetectorRef لكي نستفيد من الدوال Methods التي تحتويها، وأول دالة كانت detach حيث قمنا باستدعائها في دالة constructor وهي اول دالة يتم تنفيذ محتوياتها عند بناء أي component، وكأنا نقول لي Angular لا تقم بتفعيل نظام Change Detection في هذا component.

(2) هنا فقط أحببت ان اطبع قيمة المتغير time في الدالة ngDoCheck لكي فقط أريك كمية البيانات التي قد تمر إلى هذا component وكيف انه مع كل جزء من هذه البيانات سوف يقوم بعمل check لجميع components tree بالطريقة العادية.

(3) اما هنا فنقوم بتوليد بيانات الوقت بأجزاء الثانية الواحدة ونعمل لها subscribe ونخزن قيمة كل جزء من هذه الأجزاء في المتغير time ومن ثم نضع شرط في حال وصل عدد هذه الأجزاء إلى 1000 جزء بمعنى انه وصل إلى ثانية واحدة (كما هو معروف ان الثانية تساوي ١٠٠٠ ملي جزء من الثانية) فقم بتفعيل نظام Change Detection يدوياً عن طريق استدعاء الدالة .detectChanges

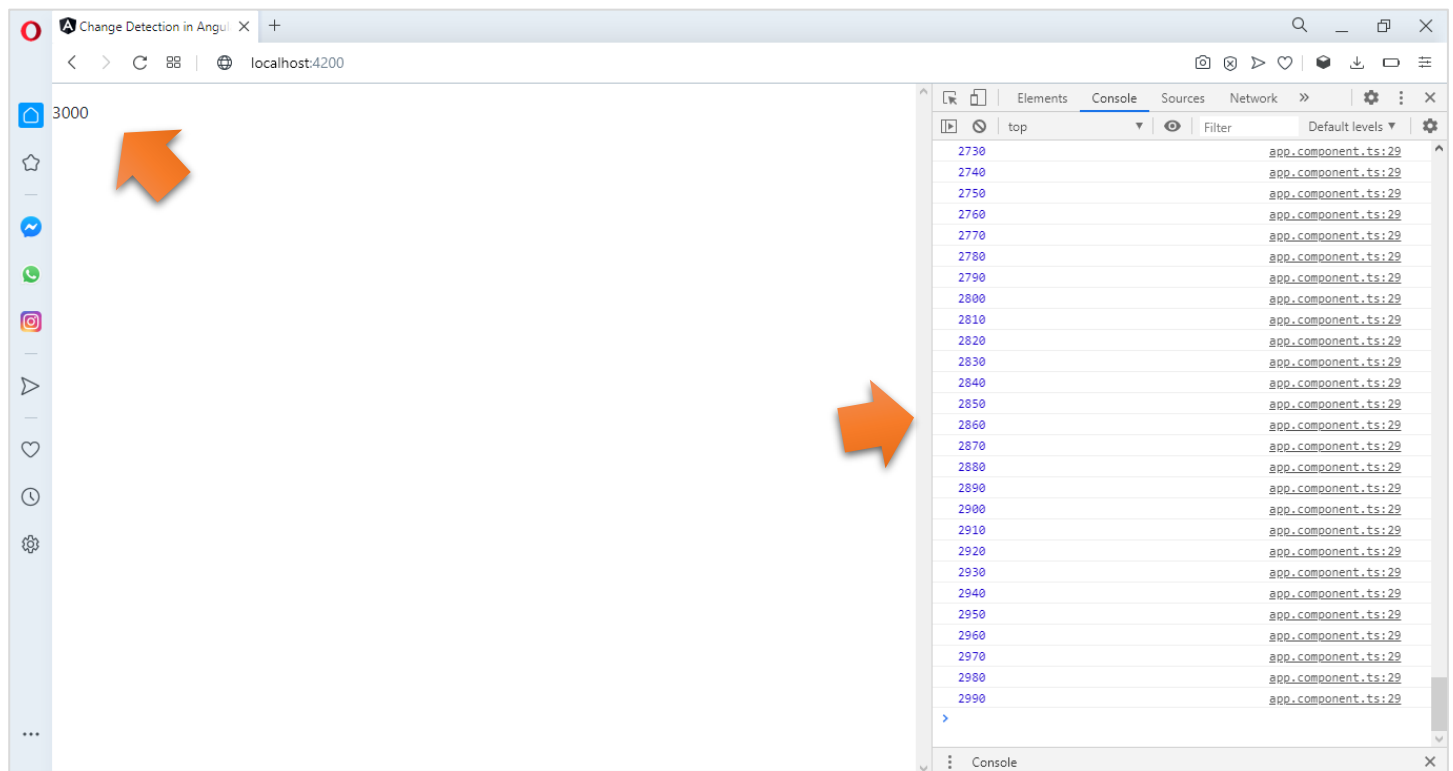
الآن لنقم بعمل bind للمتغير time في ملف template، كالتالي:

ملف app.component.html

```
<p>{{ time }}</p>
```

ولنذهب إلى المتصفح ولنرى النتيجة:





انظر للجزء الأيمن سوف ترى الكم الهائل من البيانات التي يتم توليدها ولك ان تتخيل عزيزي المتعلم كم عملية check وتحديث لي template سوف تتم ان لم نقوم بإيقاف عمل نظام Change Detection، لكيلا يؤثر على أداء التطبيق.

اما الجزء الايسر فنلاحظ ان ملف template لا يتم تحديثه إلا كل ثانية فقط وذلك لأننا قمنا بوضع شرط ونتيجة هذا الشرط هو تفعيل نظام Change Detection يدوياً عن طريق detectChanges.

ونستطيع الذهاب خطوة إلى الأمام والاستغناء عن detectChanges وتحديث DOM مباشرة لكي نستغني إعادة تفعيل نظام Change Detection ونحن كل الذي نريده هو تحديث عنصر محدد او مجموعة عناصر محددة في component معين، ويتم ذلك من خلال إعطاء Template Reference لعنصر DOM وهو في حالتنا هذه `<p></p>` ومن ثم عن طريق ViewChild نصل إلى هذا العنصر عن طريق ملف class كما تعلمنا سابقاً ومن ثم عن طريق استخدام Rnderer2، نقوم بتغيير القيمة لهذا العنصر مباشرة في DOM عن طريق اسناد القيمة time له، او مباشرة عن طريق nativeElement في حال لا نريد استخدام Renderer2، كالتالي:

```
this.clock.nativeElement.textContent = `${time}`;
```

بحيث ان clock هو template reference للعنصر `<p></p>`.

مع العلم انه يتم استبدال السطر السابق بالسطر الخاص بالدالة detectChanges.

وبذلك نستطيع ان نقول ان هاتين الدالتين يعملان في الغالب مع بعض، إلا في بعض الحالات التي نستخدم فيها الدالة detectChanges بمفردها، ومنها الذي صادفناها سابقاً في الجزء الخاص بطريقة اتصال component مع نفسه عن طريق services، حيث ظهر لنا الخطأ التالي:

**Expression has changed after it was checked... etc.**

وهذا الخطأ معناه بشكل عام وبصيغة مبسطة بعيدة عن تعقيد المصطلحات، ان Angular يُخبرنا انه قام باكتشاف تغيير معين في class او state (البيانات)، سواء كان هذا التغيير ناتج عن Event او اتصال لجلب بيانات من server، ... الخ، وانه قام بتفعيل نظام Change Detection وعند انتهائه وجد دالة معينة قامت بتغيير هذه state مرة أخرى فلذلك يخبرنا انه لا يُريد إعادة تفعيل نظام Change Detection مرة أخرى، وهناك مجموعة من الحلول منها وليس (افضلها) هو استخدام الدالة detectChanges في (الدالة التي اغضبت Angular علينا)، لكي نجبره لتفعيل نظام Change Detection.

ومن الحلول على سبيل المثال وضع الكود في دالة setTimeout، او باستخدام المكتبة ngZone وهذه الأخيرة قد يكون لنا معها وقفة بإذن الله.

وبذلك نكون انتهينا من هاتين الدالتين، وسوف ننتقل في الجزء التالي إلى دالة من استخداماتها انها تغنيانا عن تغيير المرجع للكائن.

#### 2.4.2.4. الدالة markForCheck:

هذه الدالة تعمل مع النوع OnPush فقط وتقوم بعمل check لنفس component ولا تهتم بالـ components الأبناء ان وجدت، وتقوم فكرتها اننا نقوم بعمل mark لجزء محدد من الكود بحيث نطلب من Angular ان يقوم بتشغيل نظام Change Detection في النوع OnPush في هذا الجزء المحدد فقط، حتى ولو كان هذا الجزء لا يخضع للحالات التي يتم فيها عمل check في هذا النوع (etc. - input change reference - async pipe).

ومن استخداماته في أحد الأمثلة السابقة قمنا بتمرير كائن إلى مصفوفة وقلنا في حال استخدمنا النوع OnPush فإن نظام Change Detection لن يلاحظ أي تغييرات على هذا الكائن إلا في حال قمنا بتغيير المرجع Reference له، لذلك نستطيع استخدام هذه الدالة بدون لا نقوم بتغيير المرجع وسوف يلاحظ نظام Check Detection سوف يكتشف هذه التغييرات ويجريها، لذلك لنقوم بتعديل المثال السابق، لكي يتلاءم مع التعديلات الجديدة، كالتالي:

جزء من ملف app.component.ts

```
changeValue() {
  this.obj = Object.assign({}, this.obj);
  this.obj.problems[0].
    Diabetes[0].
    medications[0].
    medicationsClasses[0].
    className[0].
    associatedDrug[0].
    name = 'Panadol';
}
```

Logic السابق هو الموجود في المثال ونلاحظ الأمر الذي اصفناه لتغيير المرجع للكائن، لذلك سوف نستبدل هذا الامر بهذه الدالة، فقط، كالتالي:

جزء من ملف app.component.ts

```
changeValue() {
  this.obj.problems[0].
    Diabetes[0].
    medications[0].
```

```

medicationsClasses[0].
  className[0].
  associatedDrug[0].
  name = 'Panadol';
  this.ref.markForCheck();
}

```

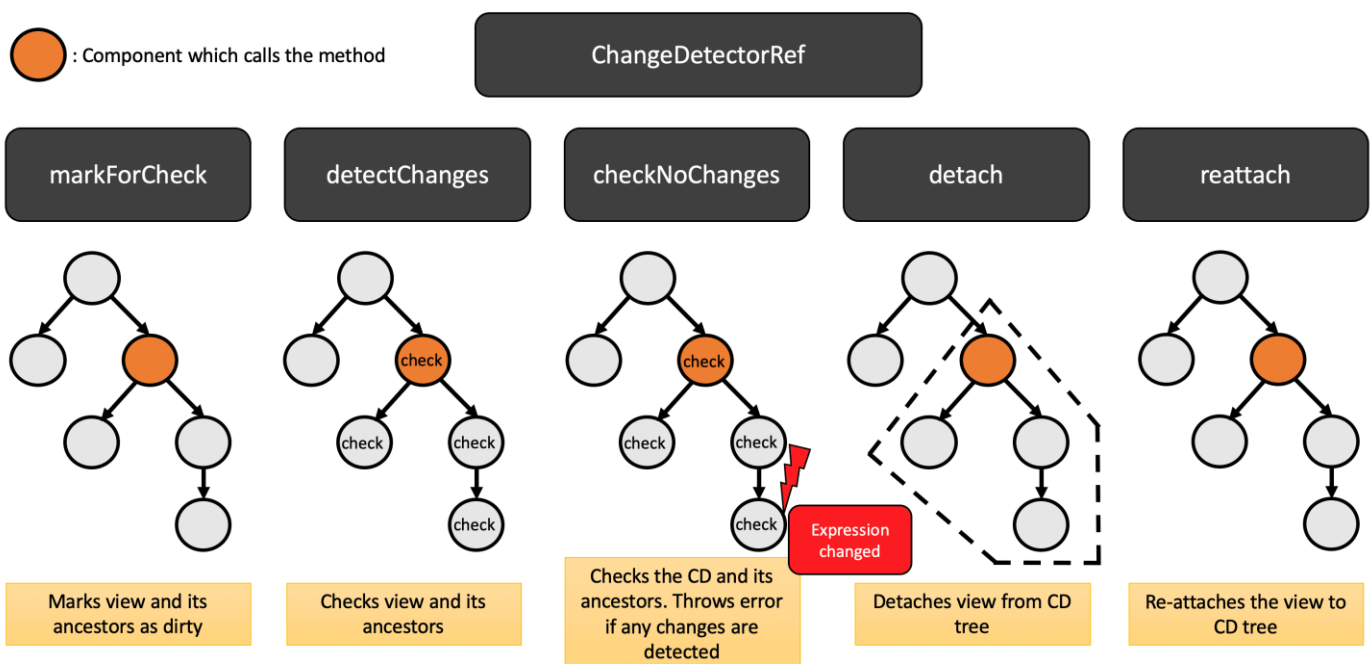
ولو قمنا بتشغيل التطبيق سوف نلاحظ انه يعمل بدون أي مشاكل.

#### 3.4.2.4 الدالة reattach:

وهي دالة تقوم بإرجاع نظام Change Detection على وضعه الافتراضي.

#### 4.4.2.4 الفرق بين دوال ChangeDetectorRef:

يمكن تلخيص الفروق بين هذه الأنواع من خلال الشكل التالي: (المصدر <https://www.mokkapps.de/>)

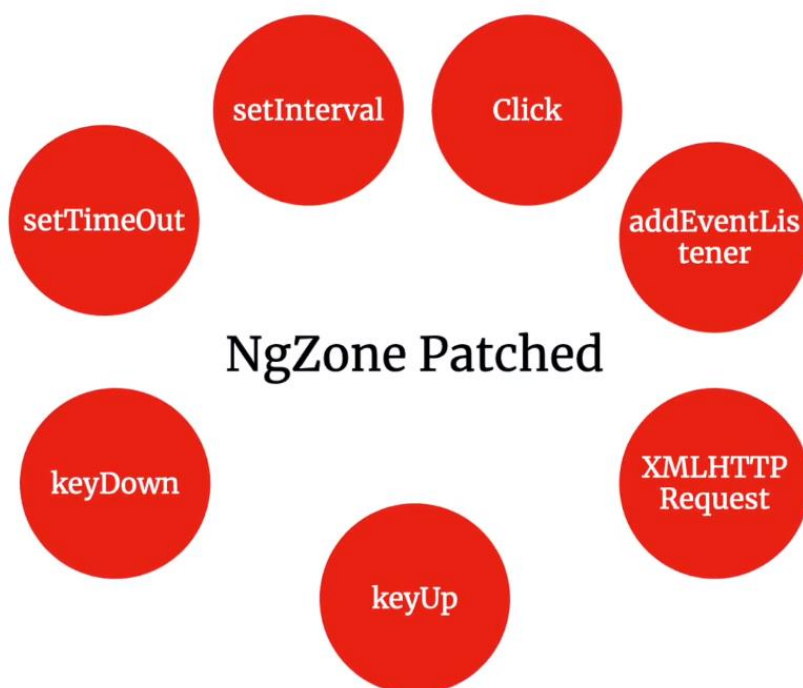


#### 5.2.4 NgZone:

هي عبارة عن service مبنية على مكتبة zone.js وهذه المكتبة هي التي تعمل على إدارة نظام Change Detection في إطار عمل Angular، وتحتوي هذه service على بعض الدوال التي يمكن ان تساعد المطورين في أعمالهم، وسوف أتكلم عن دالتين الأولى هي run والثانية هي runOutsideAngular، ومهمة الدالة الأولى هي لتنفيذ كود خارجي داخل نظام Change Detection، ومن امثلته كما هو معروف ان Angular يُستخدم في اكثر من إطار او بيئة عمل مثل NativeScript او Electron وغيره من بيئات العمل او ممكن نستخدم بعض المكتبات الخارجية التي تتعامل مع البيانات الغير تزامنية Asynchronous، فهذه المكتبات او البيئات قد لا تخضع لنظام Change Detection ونريد ان نضيف هذا الكود إلى هذا النظام ففي هذه الحالة نستخدم هذه الدالة، اما الدالة الثانية فهي واضحة من اسمها في حال اردنا ان ننفذ كود معين لا يُطبق فيه قواعد نظام Change Detection.

ونستطيع ان نقول بصيغة مبسطة لتوضيح فقط، ان هذين الدالتين تشبهان الدالتين detach و reattach حيث الدالة runOutsideAngular في NgZone تقابل الدالة detach في ChangeDetectorRef والدالة run في NgZone تقابل reattach في ChangeDetectorRef.

وكما قلنا سابقاً ان Change Detection والذي هو في الحقيقة عبارة عن الواجهة لي مكتبة zone.js، تستطيع الكشف عن اغلب التعديلات سواء كانت Events او طلبات Request لسيرفر... الخ، ونستطيع تمثيل هذا الامر بالشكل التالي:



ولتوضيح لنعطي مثال على الدالة الثانية، ولنفرض انه لدينا بيانات غير تزامنية يتم جلبها عن طريق اتصال HTTP Request، ولنفرض ان هذه البيانات ضخمة ويتم جلبها بكل لحظة، فهذه الحالة نستطيع ان نرفع من أداء التطبيق بأن نقوم بتنفيذ هذا الكود خارج نطاق مكتبة zone.js، لذلك كنوع من التدريب لنقوم بإنشاء service باسم users وهذه service تتصل بي API عن طريق موقع jsonplaceholder والذي يقدم مجموعة من البيانات للاختبار والتجربة، كالتالي:

ملف users.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class UsersService {

  constructor(private http: HttpClient) { }

  getUsers(): Observable<any[]> {
```

```

    return this.http.get<any[]>('https://jsonplaceholder.typicode.com/users');
  }
}

```

ولنقوم بحقن هذه service في ملف class في Root Component ولنقوم بإنشاء دالة وليكن اسمها runSomeLogic ومهمتها الاتصال بهذه service وجلب البيانات، ومن ثم نقوم بإضافة متغير يقوم بوظيفة العد ونضيفه في دالة lifecycle hooks ذات الاسم ngDoCheck، وقد قلنا سابقاً أن أي عملية check تتم في التطبيق يتم استدعاء هذه الدالة، لذلك نريد أن نعرف كم عملية check تتم داخل التطبيق، كالتالي:

ملف app.component.ts

```

import {
  Component,
  OnInit,
  ChangeDetectionStrategy,
  DoCheck,
} from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class AppComponent implements OnInit, DoCheck {

  private counter = 0;

  constructor(private userService: UsersService) {}

  ngOnInit(): void {}

  ngDoCheck(): void {
    console.log(`Number Of Checks: ${this.counter++}`);

    runSomeLogic() {
      console.log('Logic ...');
      this.userService.getUsers().subscribe((data) => {
        this.zone.runOutsideAngular(() => {
          console.log(data);
        });
      });
    };
  }
}

```

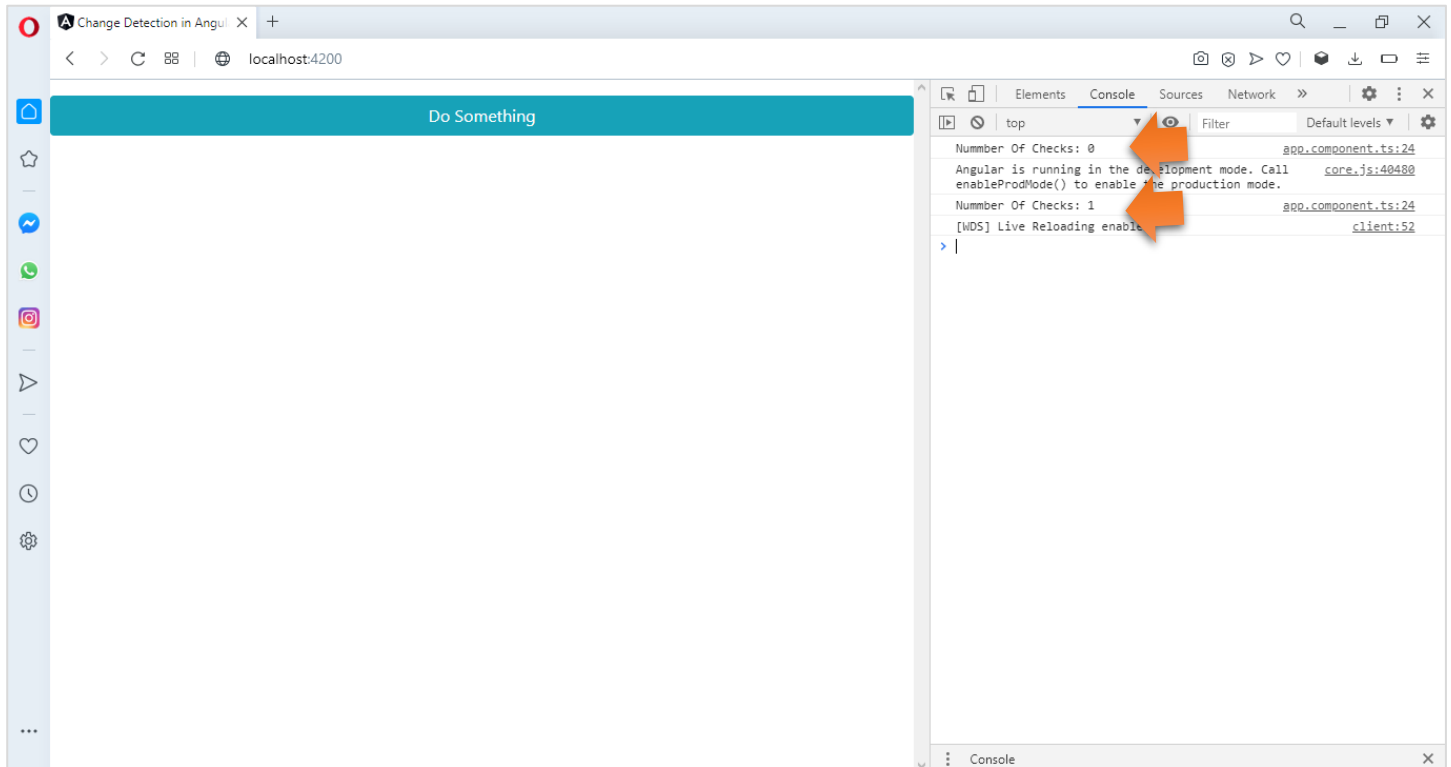


وأخيراً لننشئ زر لكي ينفذ الدالة runSomeLogic، كالتالي:

ملف app.component.html

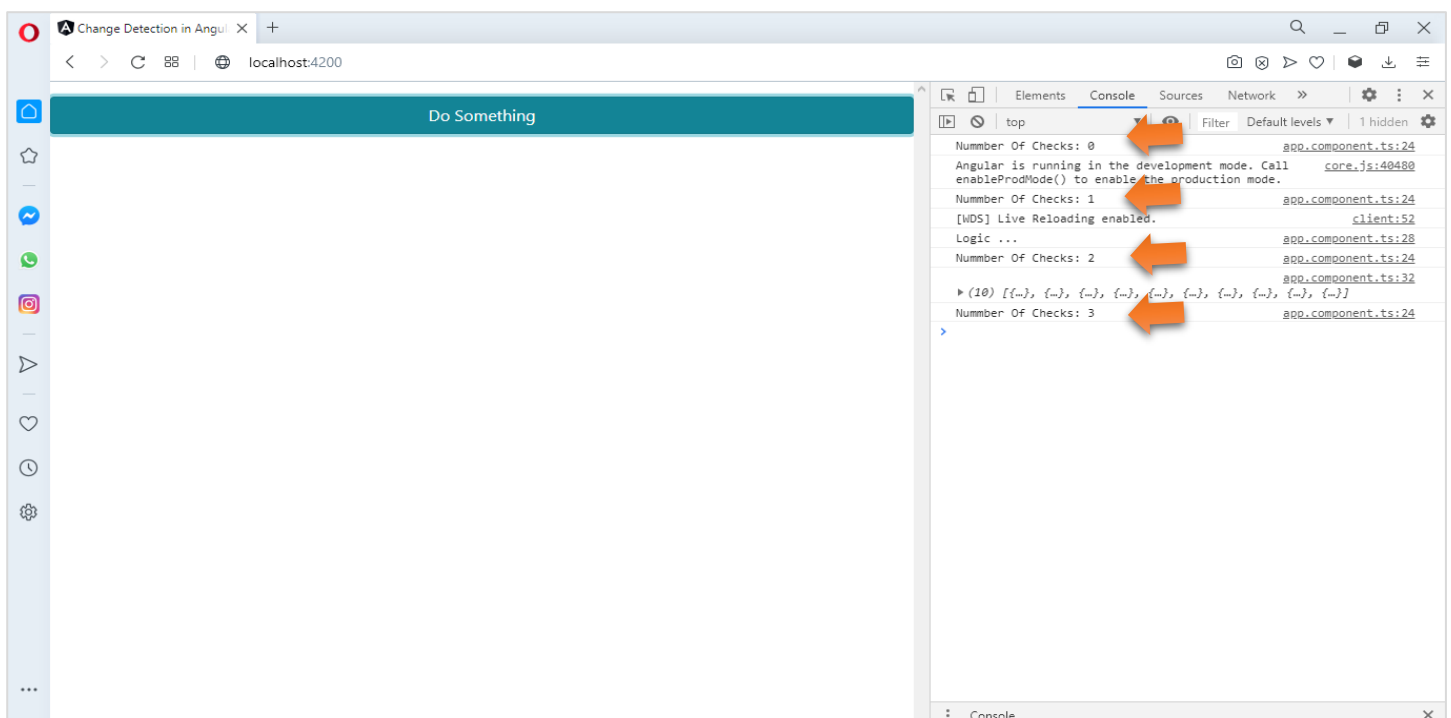
```
<button class="btn btn-info w-100" (click)="runSomeLogic()">
  Do Something
</button>
```

الآن لنقم بتشغيل التطبيق ولتري كم عملية check تتم على التطبيق قبل ان نستخدم NgZone، كالتالي:



نلاحظ انه فقط لدينا component واحد وقد تم تنفيذ عمليتين checks الآن لنضغط على الزر لكي ننفذ محتويات الدالة

runSomeLogic، كالتالي:



نلاحظ عند الضغط على الزر تم تنفيذ الدالة runSomeCode حيث قام في البداية بتنفيذ الامر console.log("Logic ...") ومن ثم قام بعمل check وعندما حدث اتصال HTTP Request قام بتفعيل check مرة أخرى، لذلك نستطيع التقليل من عمل check من خلال تنفيذ الكود الخاص بالاتصال بالservice لجلب البيانات خارج نظام Change Detection، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  ChangeDetectionStrategy,
  DoCheck,
  NgZone,
} from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class AppComponent implements OnInit, DoCheck {

  private counter = 0;

  constructor(private usersService: UsersService, private zone: NgZone) {}

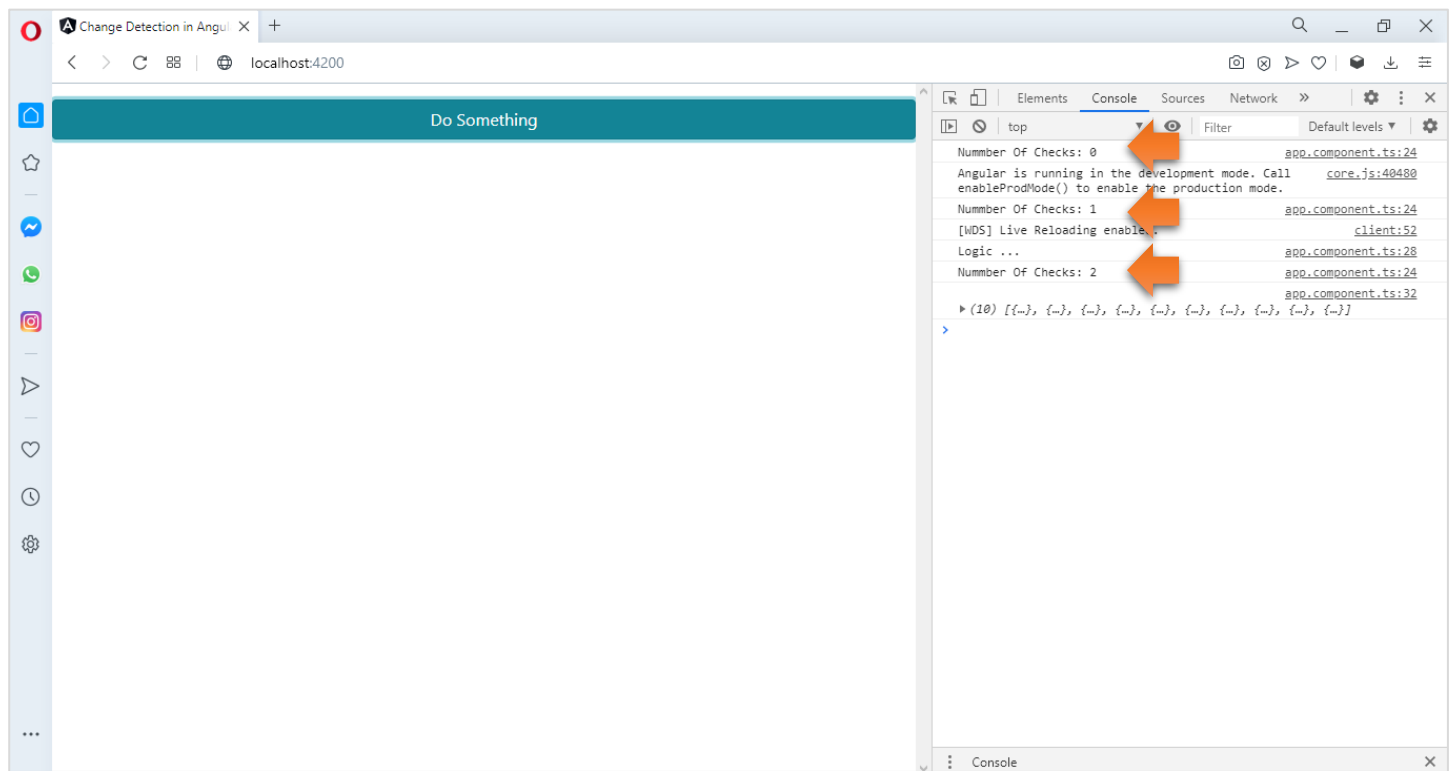
  ngOnInit(): void {}

  ngDoCheck(): void {
    console.log(`Number Of Checks: ${this.counter++}`);
  }

  runCodeOutsideAngularZone() {
    console.log('Logic ...');
    this.zone.runOutsideAngular(() => {
      this.usersService.getUsers().subscribe((data) => {
        this.zone.runOutsideAngular(() => {
          console.log(data);
        });
      });
    });
  }
}
```

نلاحظ اننا قمنا بوضع الكود الخاص بالاتصال بالservice داخل الدالة runOutsideAngular وهذه الدالة تستقبل باراميتر وهو عبارة عن دالة وداخل هذه الدالة قمنا بكتابة Logic، كما نستطيع كتابة هذا Logic في دالة خارجية ومن ثم تنفيذ هذه الدالة هنا.

الآن لنقم بحفظ هذه التعديلات ولنذهب إلى المتصفح ونضغط على الزر ولنرى النتيجة، كالتالي:



نلاحظ ان الكود الخاص بجلب البيانات لم يعمل له check، وبذلك نكون قللنا من عمل check لكي نرفع من أداء التطبيق. وهذا مثال آخر من الموقع الرسمي لإطار العمل Angular، تم فيه استخدام كلا الدالتين، وسوف استعرض الكود ومن ثم أعلق على بعض الاسطر منه لتوضيحه، كالتالي:

```

app.component.ts ملف
import {
  Component,
  OnInit,
  ChangeDetectionStrategy,
  DoCheck,
  NgZone,
} from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  changeDetection: ChangeDetectionStrategy.Default,
})
export class AppComponent implements OnInit, DoCheck {

  private counter = 0;
  progress = 0;
  label: string;

  constructor(
    private usersService: UsersService,
    private zone: NgZone
  ) {}

```



```

ngOnInit(): void {}

ngDoCheck(): void {
  console.log(`Number Of Checks: ${this.counter++}`);
}

runSomeLogic() {
  console.log('Logic ...');
  this.zone.runOutsideAngular(() => {
    this.userService getUsers().subscribe((data) => {
      this.zone.runOutsideAngular(() => {
        console.log(data);
      });
    });
  });
}

private increaseProgress(doneCallback: () => void) {
  this.progress += 1;
  console.log(`Current progress: ${this.progress}%`);
  if (this.progress < 100) {
    window.setTimeout(() => {
      this.increaseProgress(doneCallback);
    }, 10);
  } else {
    doneCallback();
  }
}

processWithinAngularZone() {
  this.label = 'inside';
  this.progress = 0;
  this.increaseProgress(() => console.log('Inside Done!'));
}

processOutsideOfAngularZone() {
  this.label = 'outside';
  this.progress = 0;
  this.zone.runOutsideAngular(() => {
    this.increaseProgress(() => {
      this.zone.run(() => {
        console.log('Outside Done!');
      });
    });
  });
}

```

← 2

← 3

← 4

في النقطة الأولى (1) عرفنا متغيرين الأول يحدد مراحل تقدم Progress Bar والثاني لنعرض فيه عنوان للمستخدم في حال استخدم تنفيذ الدالة داخل نظام Change Detection او لا.

اما النقطة الثانية (2) في دالة private أي لا تُستدعى إلا داخل ملف class هذا فقط، وتستقبل باراميتراً عبارة عن دالة، ومن ثم نقوم باستدعاء المتغير الخاص بعرض تقد الشريط Progress Bar للمستخدم ونقوم بزيادته بواحد، ومن ثم نضع شرط هل قيمة هذا المتغير أقل من 100 فإذا كان فقم باستدعاء الدالة لنفسها مرة أخرى داخل دالة setTimeout لكي نحدد فقط سرعة تقدم هذا الشريط حيث مررنا له القيمة 10 ملي ثانية، وكما هو معروف ان الثانية تساوي 1000 ملي ثانية، لأنه في البرامج الواقعية هذا الشريط بالعادة يكون سرعة تقدمه مبنية على بيانات سواء يتم تحميلها أو رفعها، ولكن هنا لا يوجد لدينا هذا الأمر لذلك قمنا بمحاكاة هذا الأمر عن طريق هذه الدالة، وفي حالة وصل قيمة المتغير 100 فقم بتنفيذ الدالة التي مررناها كباراميتراً لهذا الدالة private.

والنقطة الثالثة (3) عبارة عن دالة تستدعي الدالة private التي انشأناها قبل قليل، ونمرر لها دالة console.log.

والنقطة الرابعة (4) والأخيرة وهي ما يهمنا هنا حيث نقوم باستدعاء الدالة الخاصة private ولكن هذه المرة خارج نطاق نظام Change Detection الخاص بالـ Angular، وبنفس الوقت نمرر لها الدالة run لكي نُعيد تنفيذ هذه الدالة مرة أخرى إلى نظام Change Detection، لذلك عند تنفيذ هذه الدالة فإن القيمة سوف تبدأ بي واحد ومن ثم يعمل الكود خارج نطاق Change Detection لذلك لن يتم تحديث ملف template بالقيم الجديدة، وعند الانتهاء ووصول القيمة إلى 100 (تبعاً لشرط الذي وضعناه لدالة private) فإنه سوف يتم تنفيذ الدالة run لإعادة تشغيل نظام Change Detection مرة أخرى، وعند تشغيله سوف يجد قيمة المتغير progress هي 100 لذلك سوف يقوم بتحديث ملف template ويضع الرقم 100.

اما الآن لنقم بإضافة الكود الخاصة في ملف template، كالتالي:

```
app.component.html ملف
<button class="btn btn-info w-100" (click)="runSomeLogic()">
  Do Something
</button>

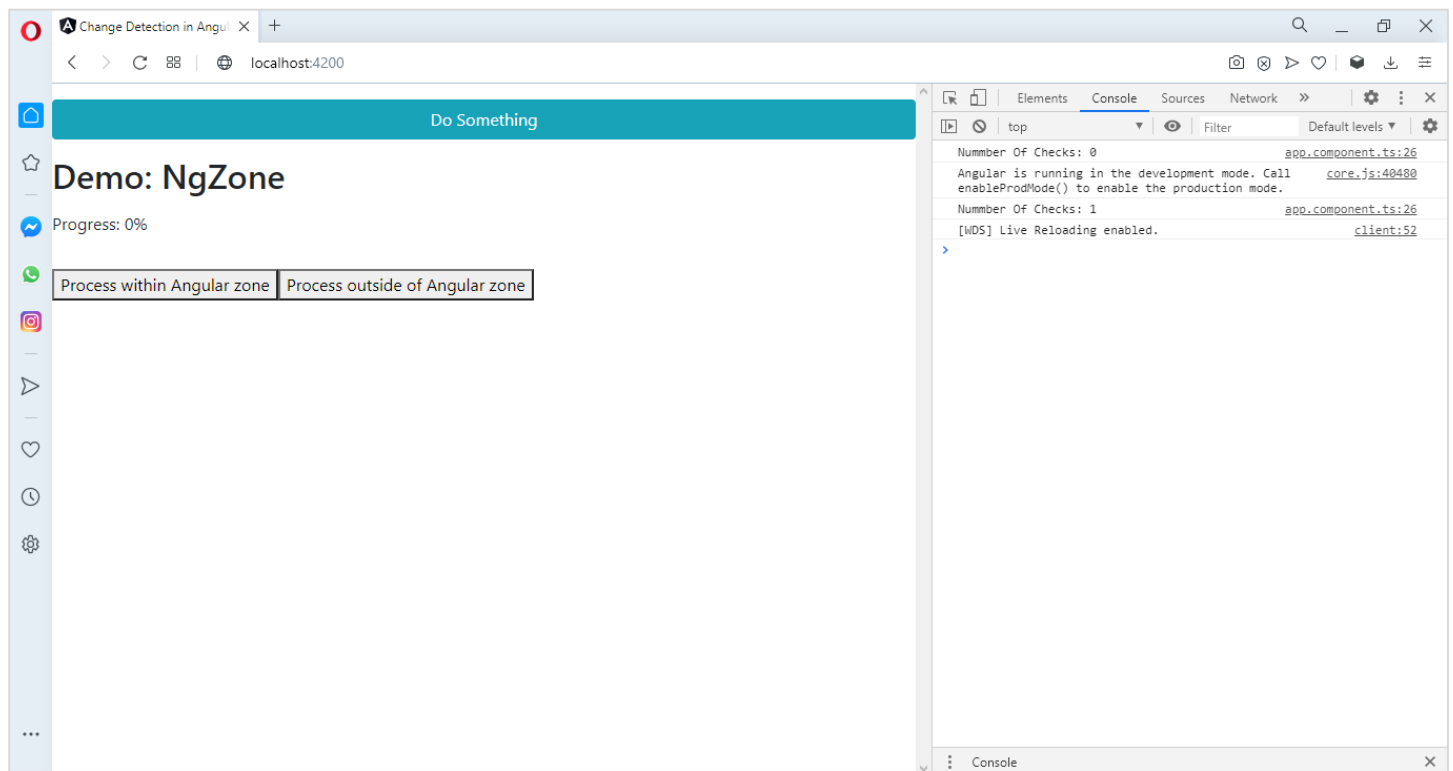
<h2>Demo: NgZone</h2>

<p>Progress: {{ progress }}%</p>
<p *ngIf="progress >= 100"> Done processing {{ label }} of Angular zone! </p>

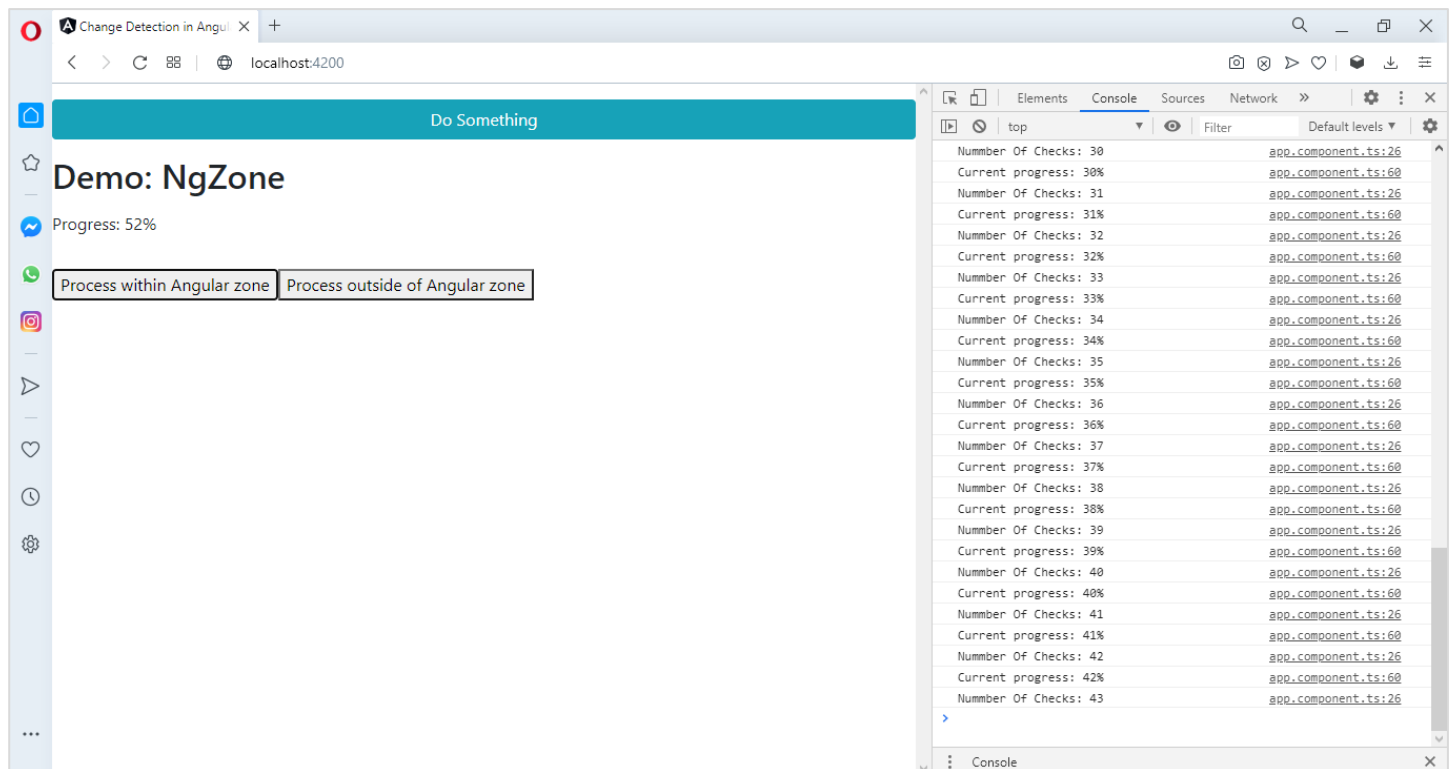
<button (click)="processWithinAngularZone()">
  Process within Angular zone
</button>
<button (click)="processOutsideOfAngularZone()">
  Process outside of Angular zone
</button>
```

نلاحظ وجود زرین واحد ينفذ الدالة التي تستدعي هي الأخرى الدالة private بالوضع الطبيعي، والأخرى تنفذ الدالة التي تستدعي الدالة private خارج نظام Change Detection، وبكل تأكيد عملنا bind لكلاً من المتغيرين progress و label.

الآن لنذهب إلى المتصفح ونرى النتيجة، كالتالي:



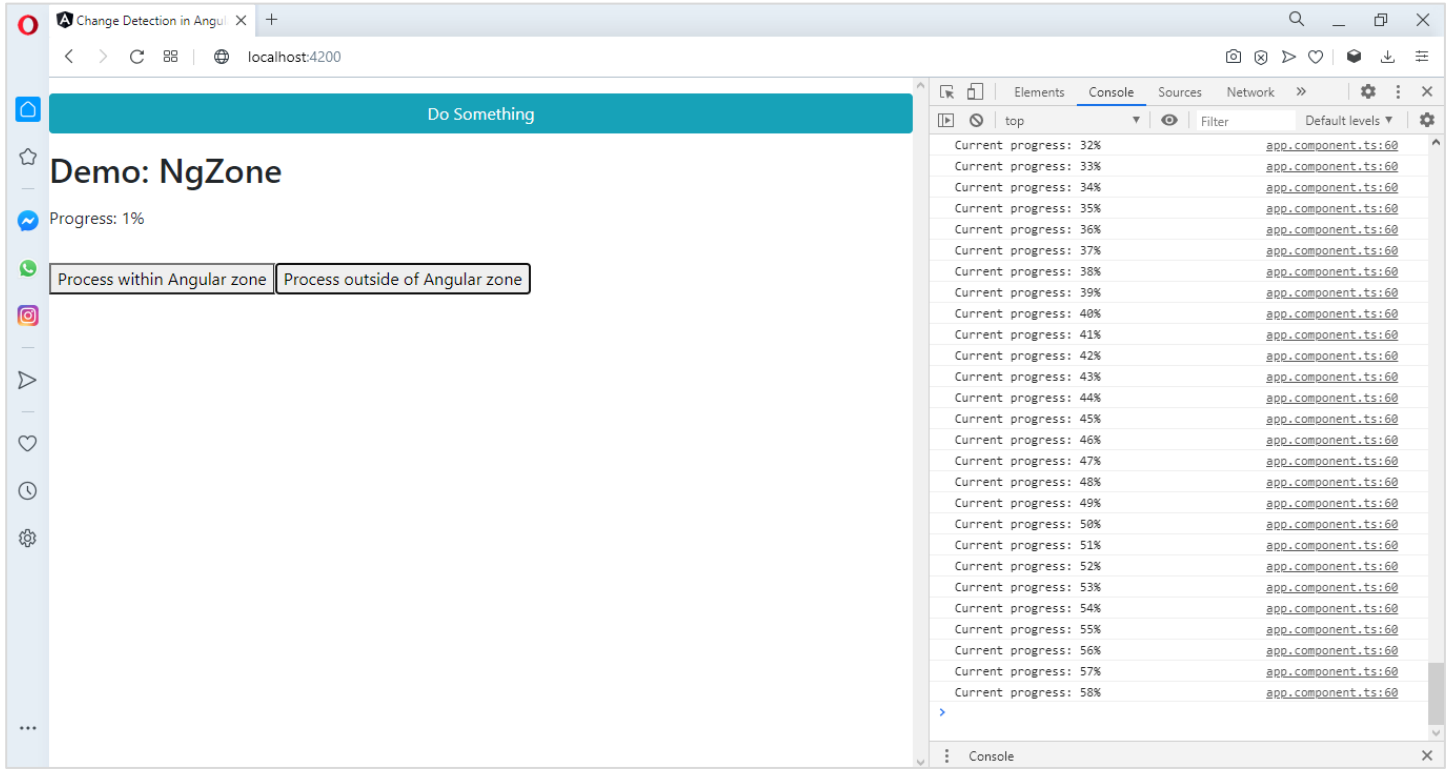
الآن لنضغط على الزر Process within Angular Zone، والتي تنفذ الدالة بوضعها الطبيعي او لنقل الافتراضي، ونرى النتيجة:



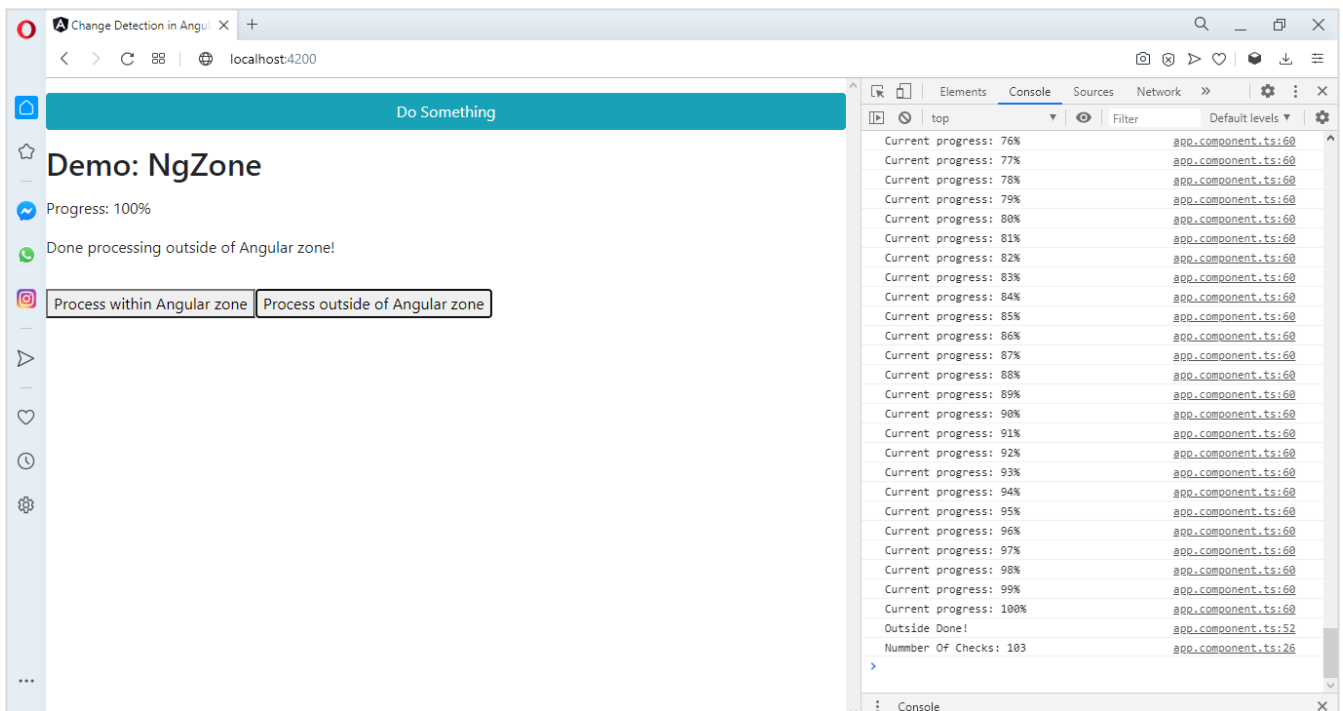
نلاحظ انه في الوضع الافتراضي يتم عمل check مع كل تغيير في قيمة المتغير progress وبناءً عليه يتم تحديث القيمة في ملف template للمستخدم، ولو نظرنا في console عند الانتهاء ووصول الرقم إلى 100، فإن عدد checks هو 101 أي انه قام بتشغيل نظام Change Detection أكثر من 100 مرة، وبالطبع كما قلنا سابقاً انه في التطبيقات الصغيرة إلى المتوسطة هذا

الأمر لا يعني أننا كمطورين، ولكن عند وصول تطبيقاتنا إلى المستوى الكبير والضخم ففي هذه الحالة يجب الانتباه إلى هذه الأمور لكي نرفع من أداء التطبيق الخاص بنا.

وبنفس الآلية لنقم بالضغط على الزر Process outside of Angular Zone، لتنفيذ الكود خارج نطاق نظام Change Detection، ومن ثم ارجاعه هذا النظام عند وصول القيمة إلى 100، كالتالي:



نلاحظ عند بداية الضغط على هذا الزر قام بعمل check ووجد قيمة المتغير هي واحد (1) لذلك قام بتحديث template ووضع القيمة واحد، ومن ثم توقف عن تحديث القيم في ملف template لأننا قلنا نفذ هذا الكود خارج نطاق Change Detection مع ان قيمة المتغير progress تزداد كما هو واضح في console يمين الشاشة، وعند الانتهاء لنرى ماذا يحدث، كالتالي:

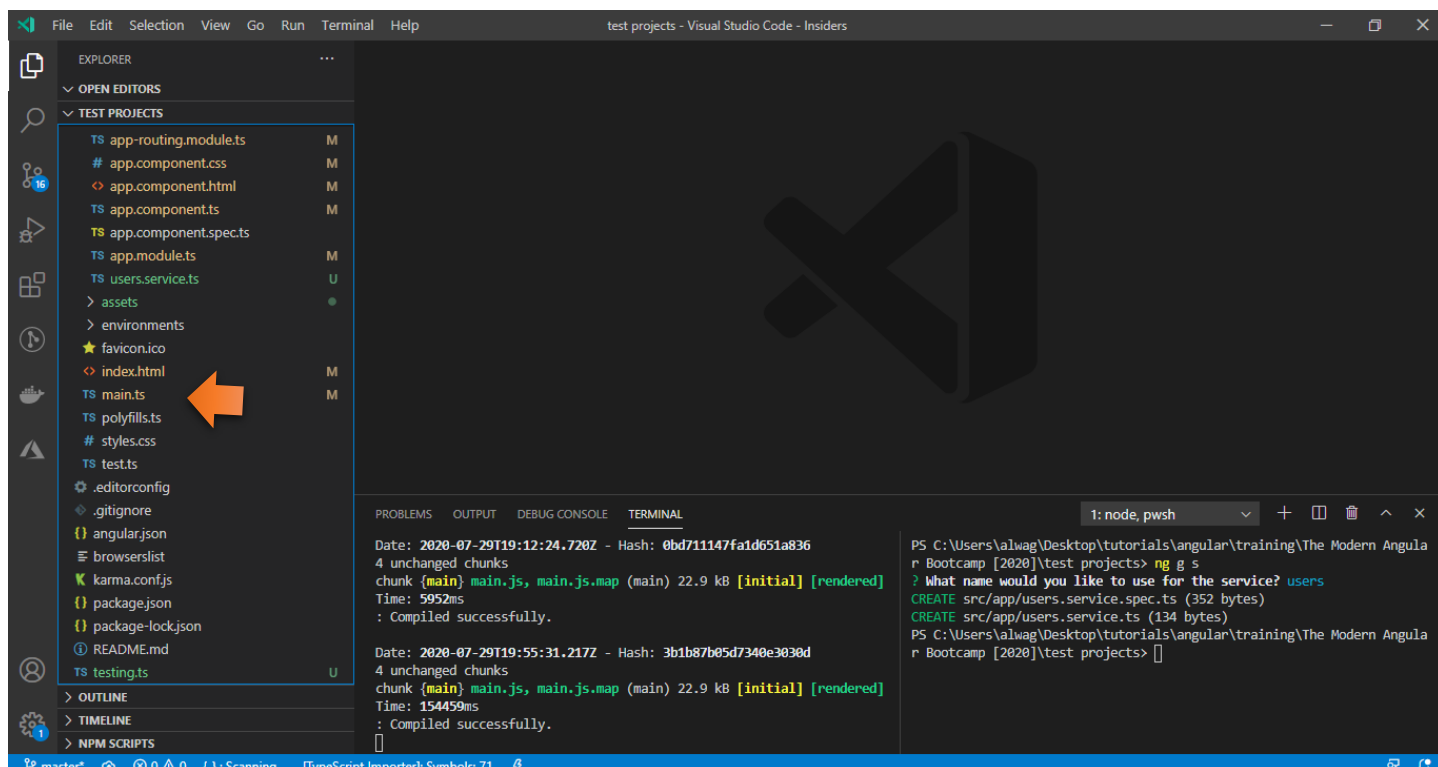


نلاحظ ان عند وصول القيمة إلى 100 تم تحديث ملف template وإظهار القيمة للمستخدم، لأننا أعدنا تفعيل نظام Change Detection عندما استخدمنا الدالة run، ولو نظرنا في console على يمين الشاشة لوجدنا ان عدد مرات checks هي 103، حيث ان 101 ناتجة عن ضغطنا لزر الأول سابقاً و2 هي الناتجة عن الضغط على الزر الثاني، بمعنى آخر انه هنا عمل check مرتين فقط المرة الأولى عندما كانت قيمة المتغير (1) والمرة الثانية عندما كانت قيمة المتغير (100)، ونستطيع ان نقول بصيغة أخرى كأننا نقول لي Angular قم بقراءة اول قيمة لهذا المتغير ومن ثم قم بتجاهل جميع القيم الأخرى وعند الوصول إلى قيمة معينة فعل نظام الكشف عن التغييرات لديك وقم بتحديث وإظهار هذه القيمة للمستخدم، وبذلك نكون قللنا من عمليات checks من 100 مرة إلى 2 مرتين فقط.

وخلاصة القول في NgZone ونظام Change Detection، انه يجب الحذر عند التعامل معه ومعرفة متى وأين وكيف يجب إيقاف او تفعيل هذا النظام، ومعرفة أي أنواع التغييرات التي يتم اكتشافها وإيها التي لا يتم اكتشافها إلا يدوياً، وبالمجمل نستطيع ان نقول إذا كانت البيانات لا تستدعي الحاجة لعرضها للمستخدم بشكل مستمر فنستطيع إيقاف هذا النظام مؤقتاً (خصوصاً في التطبيقات الكبيرة) إلى أن نصل إلى قيمة معينة ونعيد تفعيله مرة أخرى لكيلا نُرهق التطبيق بتحديثات متتالية لملف template.

## 6.2.4. الغاء NgZone:

نستطيع إيقاف NgZone نهائياً من التطبيق، وذلك من خلال الملف main.ts:



ومن ثم نضيف الأمر التالي:

ملف main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
```

```
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic()
  .bootstrapModule(AppModule, { ngZone: 'noop' })
  .catch((err) => console.error(err));
```

ولو قمنا بالضغط على الزر الخاص بتنفيذ الدالة داخل نطاق نظام Change Detection سوف نلاحظ التالي:

كما نلاحظ الجزء الأيسر لم يتم تحديث القيمة في ملف template على الرقم ان قيمة المتغير progress في تزايد كما هو ملاحظ في الجزء الأيمن في console، وذلك بالطبع بسبب اننا قمنا بتعطيل نظام Change Detection بالكامل.

وبالطبع هذا الأمر لا يُنصح فيه، وانما ذكرته هنا كمعلومة فقط، لذلك لنقوم بحذف هذا الامر من ملف main.ts:

ملف main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule).catch((err) => console.error(err));
```

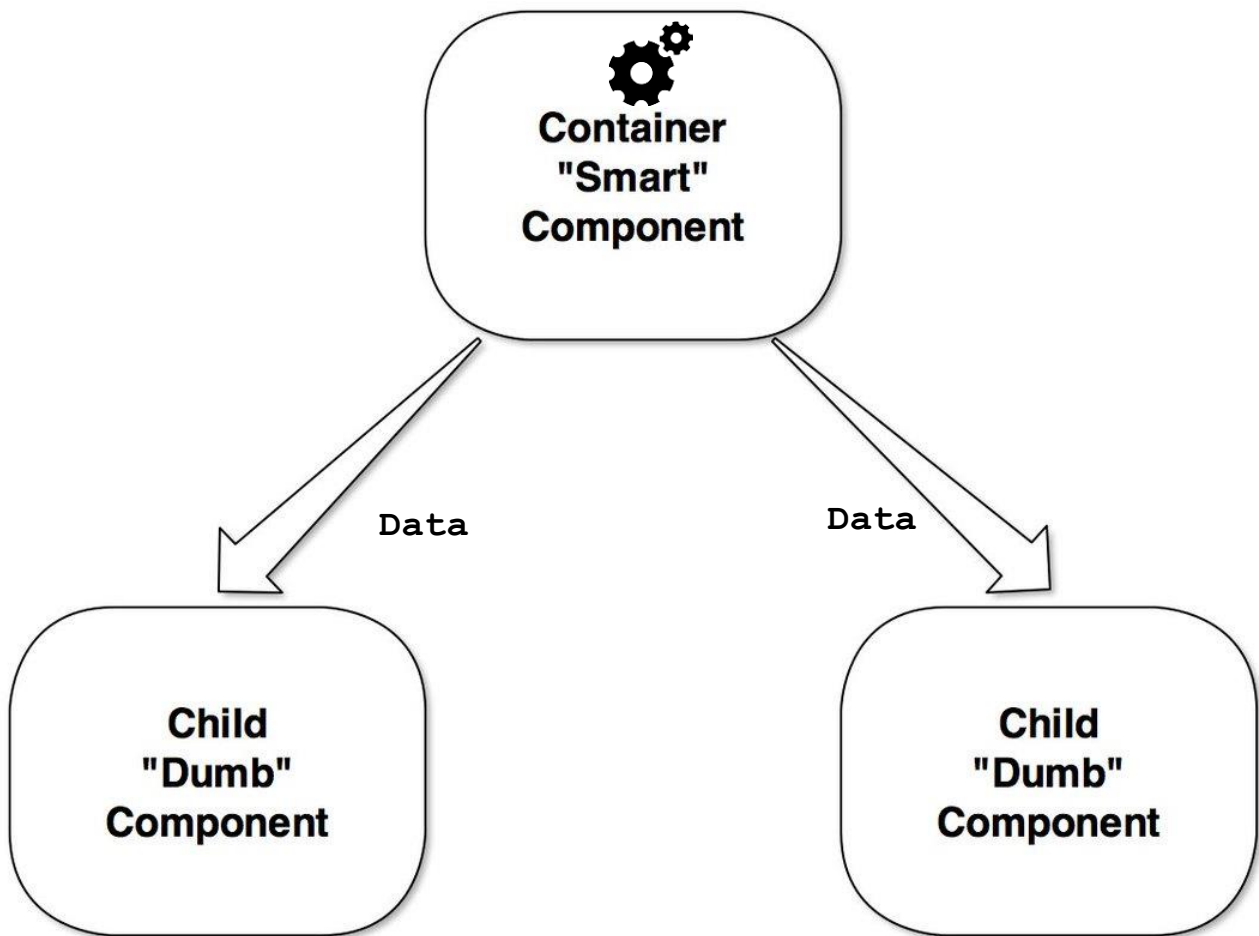
### 3.4 Smart & Dumb Components

وهي عبارة عن approach او بصيغة أخرى هو عبارة Design Pattern، وهي طريقة لتنظيم الكود فقط لا غير، لصيانة والتطوير وخصوصاً في التطبيقات الكبيرة التي تحتاج إلى تنظيم معين، كما انه يطلق عليها مسميات أخرى فال Smart Component يطلق عليه ايضاً (application-level components, container components or controller components)، اما Dumb Component فيطلق عليه ايضاً (pure components or presentation Components).

اما مفهومه فهو بسيط فال Smart Component مهمته الاتصال بال Services او الاتصال مباشرة بقاعدة البيانات عن طريق api في حال عدم الرغبة باستخدام service، بمعنى أخرى هو يقوم بعمل إدارة manage للبيانات القادمة إلى التطبيق لذلك يسمى container او control بحيث يكون اب لي Dump Component ويمرر البيانات بعد إدارتها لها.

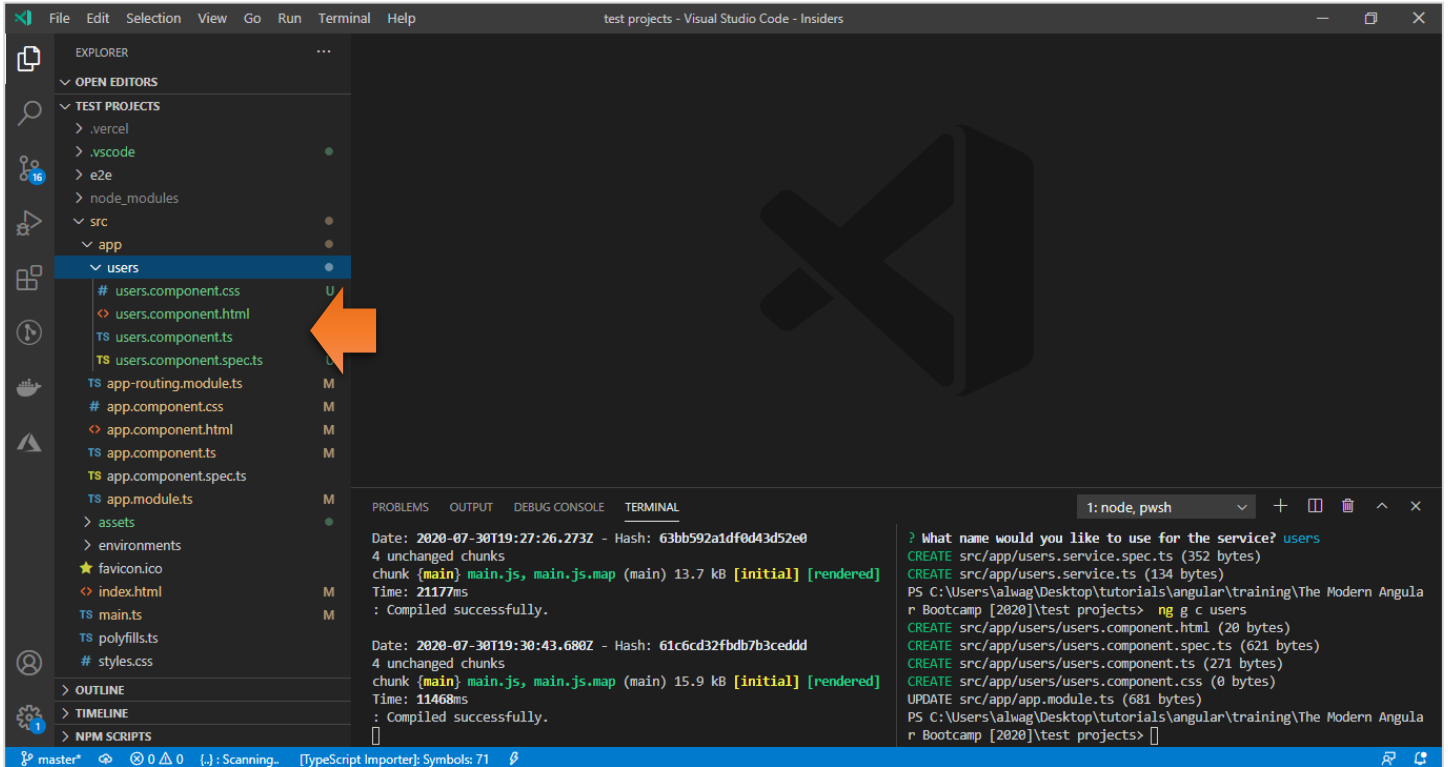
اما Dump Component فمهمته فقط استقبال البيانات عن طريق ()@Input وعرضها فقط وارسال الاحداث Events التي تحدث من المستخدم إلى Smart Component عن طريق ()@Output، لذلك تسمى pure او presentation.

ويمكن توضيحها بالشكل التالي:



وكما قلنا سابقاً أن smart هو المسئول عن عمل manage للبيانات سواء بالاتصال service او بالاتصال بشكل مباشر api.

ولتوضيح لنعطي مثال، لكن في البداية لنقوم بإنشاء مشروع جديد، وننشئ أول smart component وليكن اسمه users، كالتالي:



وبما ان هذا component هو الذي يُدير البيانات ولجعل المثال بسيط سوف أقوم بالاتصال بapi عن طريق نفس component وليس عن طريق service، كالتالي:

```
users.component.ts ملف
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.css'],
})
export class UsersComponent implements OnInit {
  users: Observable<any[]>;

  constructor(private http: HttpClient) {}

  ngOnInit(): void {
    this.users = this.http
      .get<any[]>('https://jsonplaceholder.typicode.com/users')
      .pipe(map((data) => data));
  }
}
```

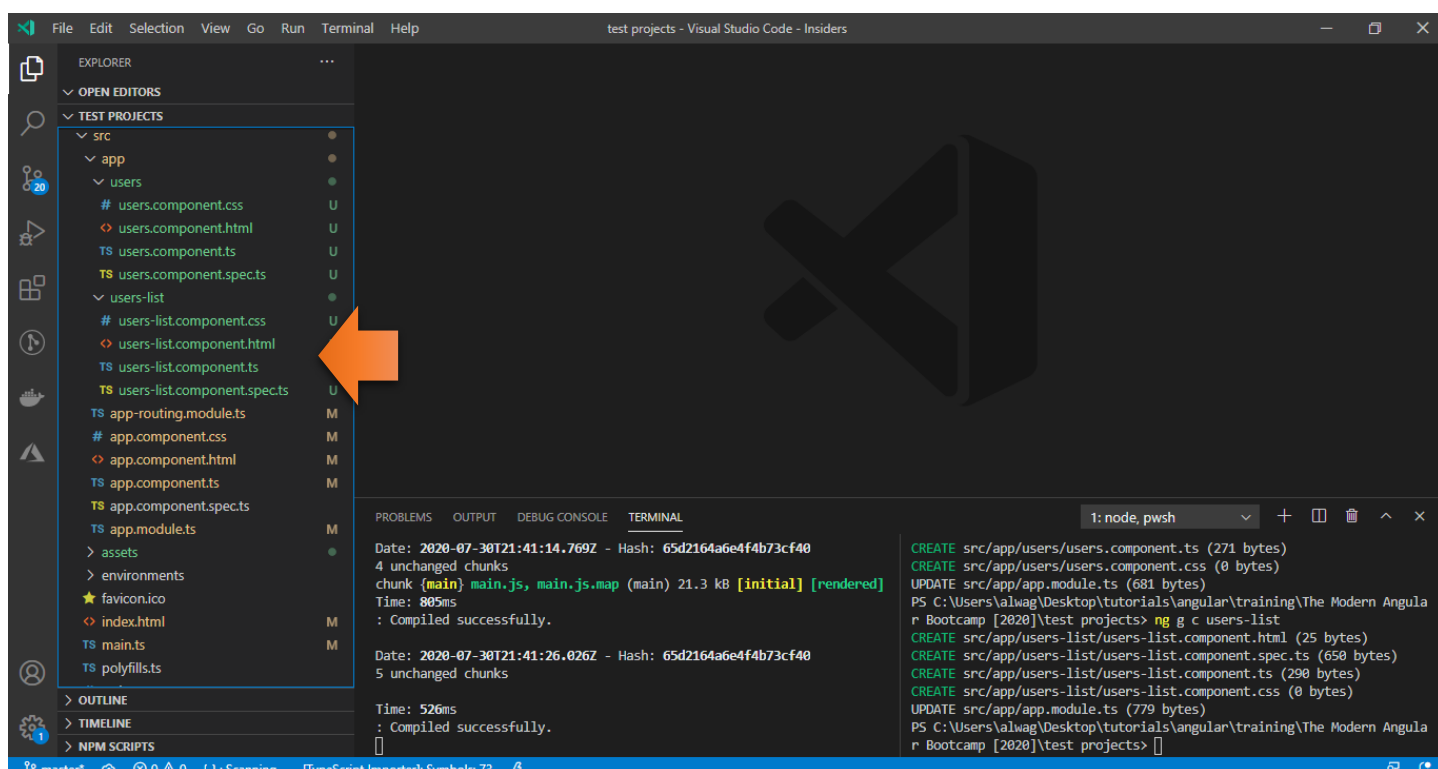


بما ان هذا component هو Smart Component لذلك نلاحظ اننا عملنا إدارة manage للبيانات وذلك من خلال الاتصال الذي اجريناه عن طريق http للاتصال بي api معين لجلب مجموعة من بيانات المستخدمين وتخزينها في متغير اسميته users. وبكل تأكيد لا ننسى ان نضيف selector الخاص بهذا component في Root Component، كالتالي:

ملف app.component.html

```
<app-users></app-users>
```

الآن لنقم بإنشاء Dump Component لكي يقوم بعرض البيانات التي عملنا لها manage في Smart Component، وليكن اسمه users-list، كالتالي:



وهذا Dump Component هو في حقيقة الأمر يعتبر أبن لي Smart Component لذلك سوف نستقبل هذه البيانات عن طريق @Input()، كالتالي:

ملف users.component.ts

```
<app-users-list [users]="users | async"></app-users-list>
```

ملف users-list.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
})
export class UsersListComponent implements OnInit {
```

```

@Input() users: any[];
constructor() {}
ngOnInit(): void {}
}

```

كما ينصح أن يتم تغيير Change Detection لي أي Dump Component لنوع OnPush، لأن هذا النوع يقوم فقط بعرض البيانات عن طريق @Input()، وكما هو معروف ان النوع OnPush يكشف التغييرات التي تحدث للبيانات عن طريق @Input()، كالتالي:

ملف users-list.component.ts

```

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

@Component({
  selector: 'app-users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
})

export class UsersListComponent implements OnInit {
  @Input() users: any[];

  constructor() {}

  ngOnInit(): void {}
}

```

والآن لنقوم بعرض البيانات في ملف template لهذا component، كالتالي:

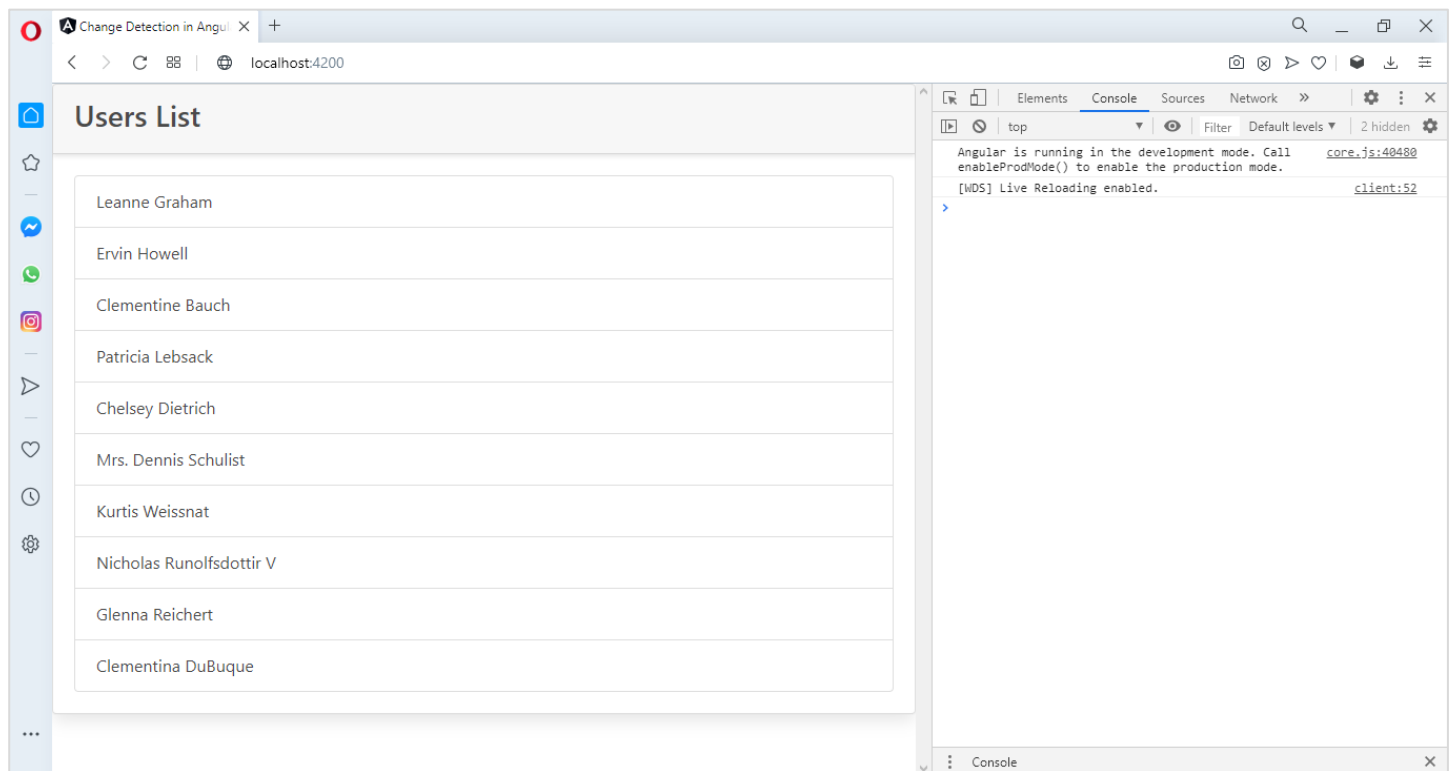
ملف users-list.component.ts

```

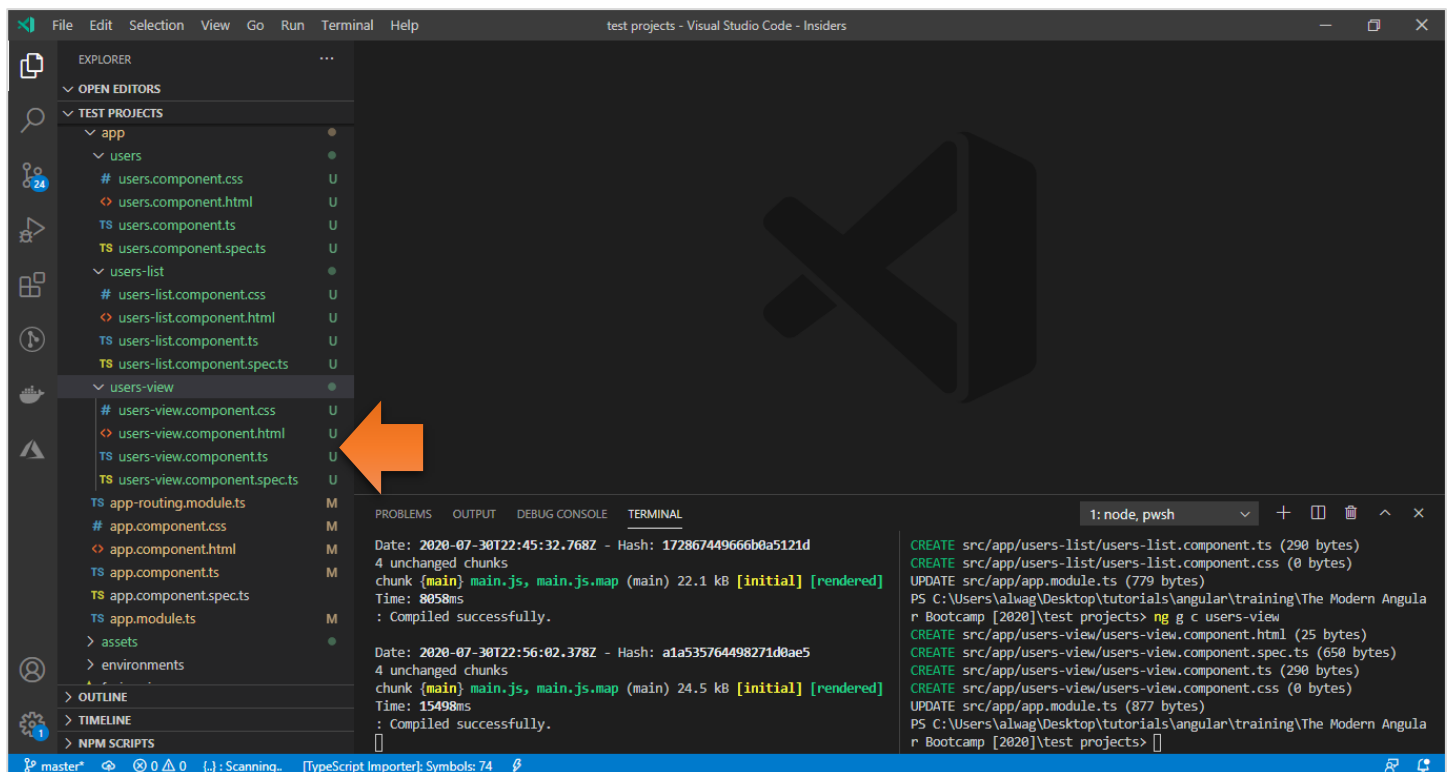
<div class="card">
  <div class="card-header">
    <h3>Users List</h3>
  </div>
  <div class="card-body">
    <ul class="list-group">
      <li class="list-group-item" *ngFor="let user of users">{{ user.name }}</li>
    </ul>
  </div>
</div>

```

والآن لنحفظ التعديلات ونذهب إلى المتصفح لنرى النتيجة، كالتالي:



كما نستطيع ان ننشأ Dump Component آخر بحيث يكون اخ للأول ويأخذ البيانات منه، وليكن اسم هذا component هو users-view، كالتالي:



ولنضيف selector الخاص بهذا component إلى الـ Dump Component الأخ له، ونعمل له Loop عن طريق ngFor ومع كل Loop نمرر البيانات إلى component المستهدف لكي قوم بعرض البيانات عوضاً عن users-list، كالتالي:

ملف users-list.component.html

```
<div class="card">
  <div class="card-header">
```

```

    <h3>Users List</h3>
  </div>
  <div class="card-body">
    <app-users-view *ngFor="let user of users" [user]="user"></app-users-view>
  </div>
</div>

```



#### ملف users-view.component.ts

```

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

@Component({
  selector: 'app-users-view',
  templateUrl: './users-view.component.html',
  styleUrls: ['./users-view.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class UsersViewComponent implements OnInit {
  @Input() user: any;

  constructor() {}

  ngOnInit(): void {}
}

```

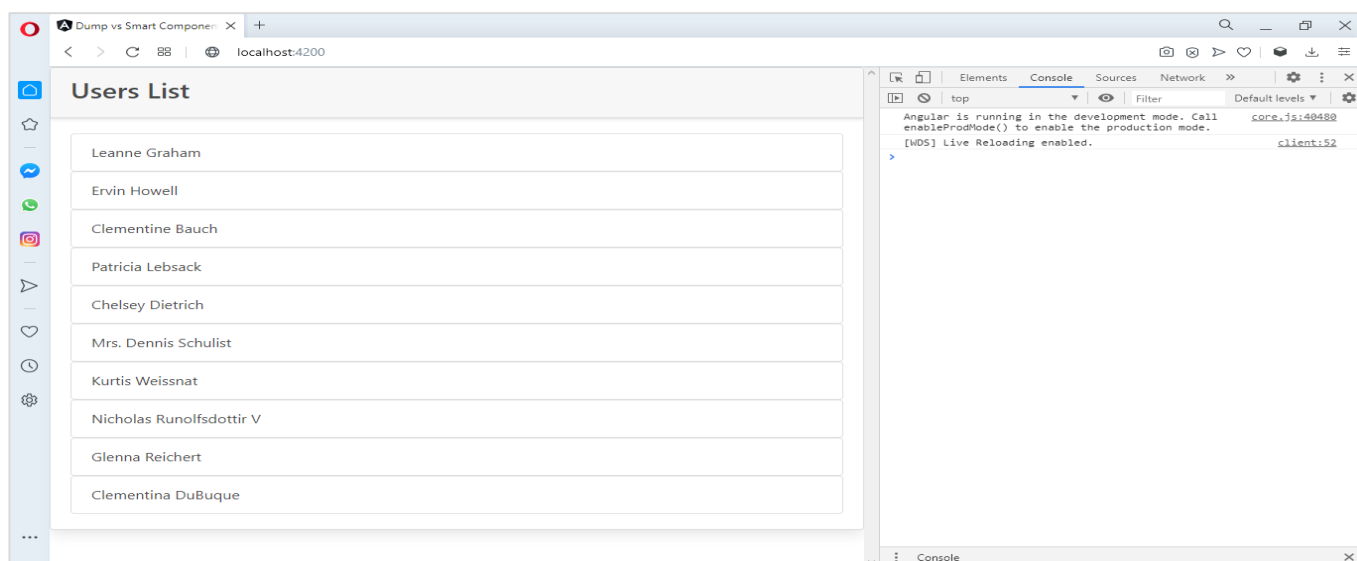
#### ملف users-view.component.html

```

<ul class="list-group">
  <li class="list-group-item">{{ user.name }}</li>
</ul>

```

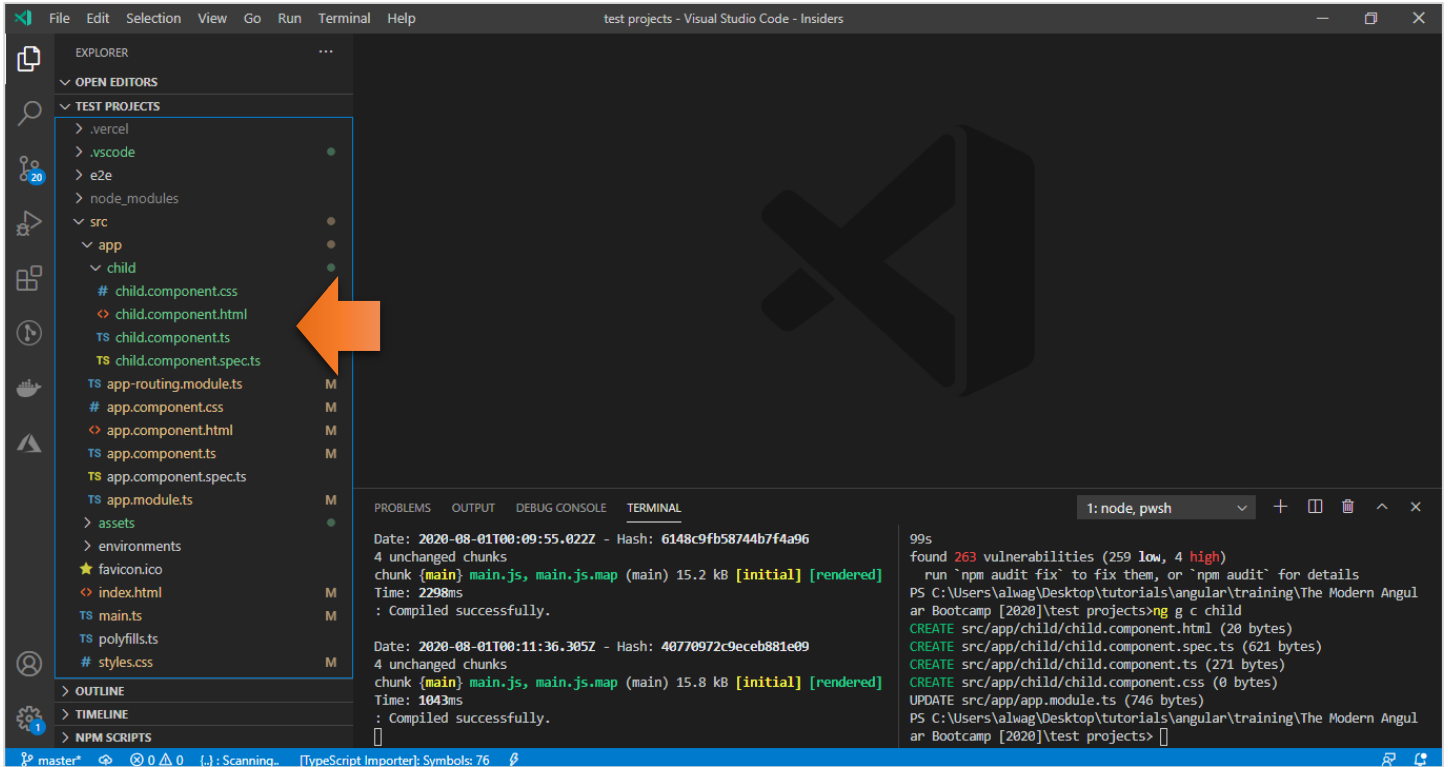
والنتيجة:



نلاحظ ان النتيجة التي تظهر للمستخدم هي نفسها، لذلك اعيد ما قلته في بداية كلامي ان هذه الميزات هي مجرد تنظيم للكود فقط لا غير.

## 4.4. Content Projection :

تعلمنا سابقاً في حال أردنا ان نمرر بيانات من component الأب إلى الأبن فأننا نستخدم @Input لكن ماذا لو أردنا ان نمرر عنصر HTML بجميع محتوياته او component آخر ايضاً بجميع محتوياته، ففي هذه الحالة نستخدم Content Projection وهي طريقة تتيح لنا إضافة عنصر HTML او component آخر من component الأب إلى الأبن مباشرة عن طريق ملف template، وتتم هذه الطريقة باستخدام الوسم `<ng-content></ng-content>` في component الأبن، ولتوضيح لنعطي مثال، ولنقم بإنشاء component جديد وليكن اسمه child، كالتالي:



وأول خطوة من خطوات عمل content projection هي إضافة الوسم `<ng-content></ng-content>` إلى ملف template في component الأبن، كالتالي:

```
ملف child.component.html

<ng-content></ng-content>
```

اما ثاني خطوة فهي إضافة selector الخاص بي component الأبن على شكل وسم HTML في ملف template في component الأب، والذي هو في حالتنا هذه الـ Root Component، كالتالي:

```
ملف app.component.html

<app-child></app-child>
```

وآخر خطوة نضيف عنصر HTML الذي نريد إضافته إلى component الأبن، بين هذا الوسم ولنفرض انه `<h1></h1>`، كالتالي:

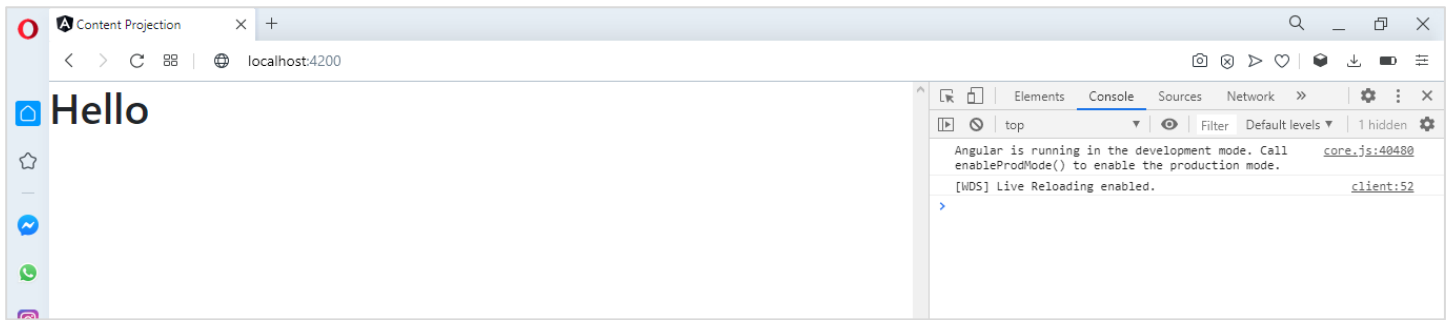
## ملف app.component.html

```
<app-child>
  <h1>Hello</h1>
</app-child>
```

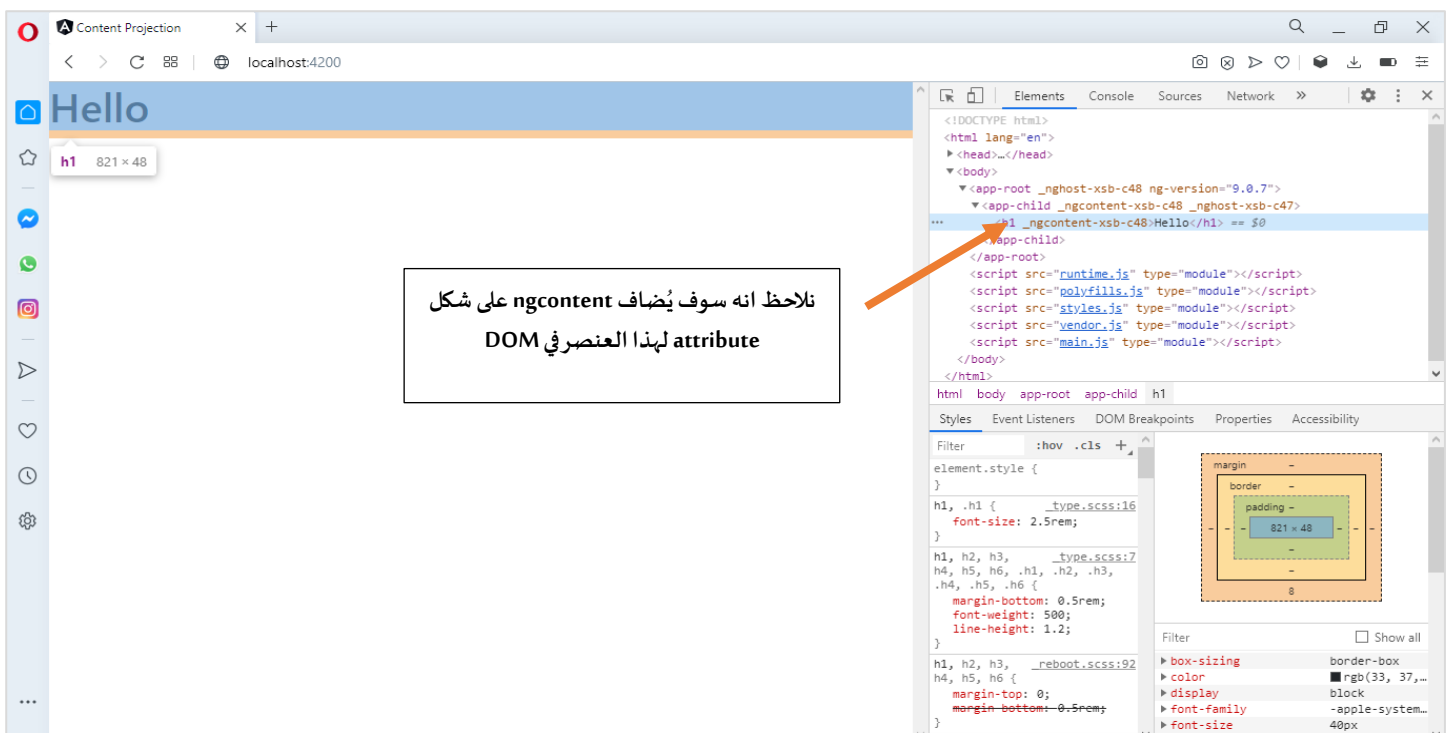
والذي سوف يقوم به Angular في هذه الحالة هو البحث عن الوسم `<ng-content></ng-content>` في component الأب  
ويستبدله بالوسم او العنصر `<h1></h1>`، كما في الشكل التالي:



اما النتيجة فتكون، كالتالي:



ولو عملنا inspect لهذا العنصر، لوجدنا التالي:



#### 1.4.4. Multi ng-content:

المثال السابق في حال كان لدينا عنصر واحد ونريد إضافته ولكن ماذا لو كان لدينا أكثر من عنصر طبعاً نستطيع باستخدام ng-content واحدة فقط ان نمرر جميع العناصر او الوسوم ولكن ماذا لو اردنا ان نربط لكل عنصر ng-content خاص به، ففي هذه الحالة لدينا عدة طرق نستطيع ربط كل ng-content مع الوسم او العنصر الذي نريد استبدالها به، وهي:

- باستخدام tag selector.
- باستخدام class selector.
- باستخدام id selector.
- باستخدام attribute selector.

ولنبداً بأول واحد وهو tag selector، كالتالي:

#### 1.1.4.4. Project using Tag Selector:

ونقوم الفكرة في هذه الحالة بالربط عن طريق اسم الوسم او العنصر بشكل مباشر في ng-content، فمثلاً لو كان لدينا وسم h1 ووسم آخر button، فنقوم بالربط بكتابة أسماء هذه الوسوم في ng-content، كالتالي:

ملف child.component.html

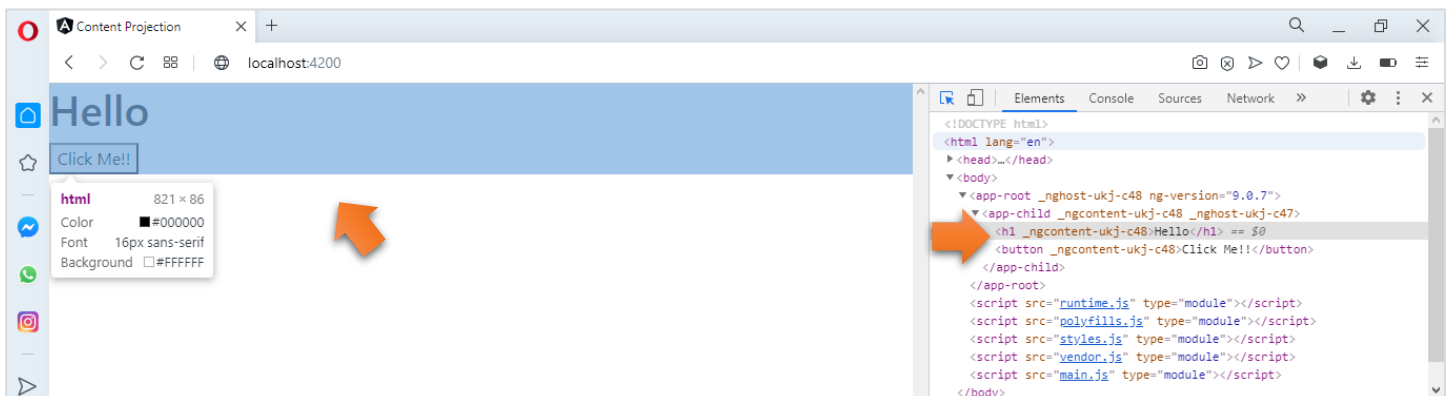
```
<ng-content selector="h1"></ng-content>
<ng-content selector="button"></ng-content>
```

وفي component الأب نضيف هذه الوسوم، كالتالي:

ملف app.component.html

```
<app-child>
  <h1>Hello</h1>
  <button>Click Me!!</button>
</app-child>
```

اما النتيجة فتكون:



ولكن يعيب هذه الطريقة هي انه سوف يقوم باستبدال جميع h1 و button وهذه قد تسبب لنا مشاكل وخصوصاً إذا أردنا نعمل تنسيقات.

#### 2.1.4.4: Project using class selector

وهذه الطريقة يتم الربط عن طريق كلاسات CSS، سواء كان موجود سابقاً كأن يكون كلاس CSS من ضمن مكتبة معينة (مثل bootstrap) واضفناه للعنصر او حتى كلاس فارغ وانما نضيفه فقط لعملية الربط كالتالي:

ملف child.component.html

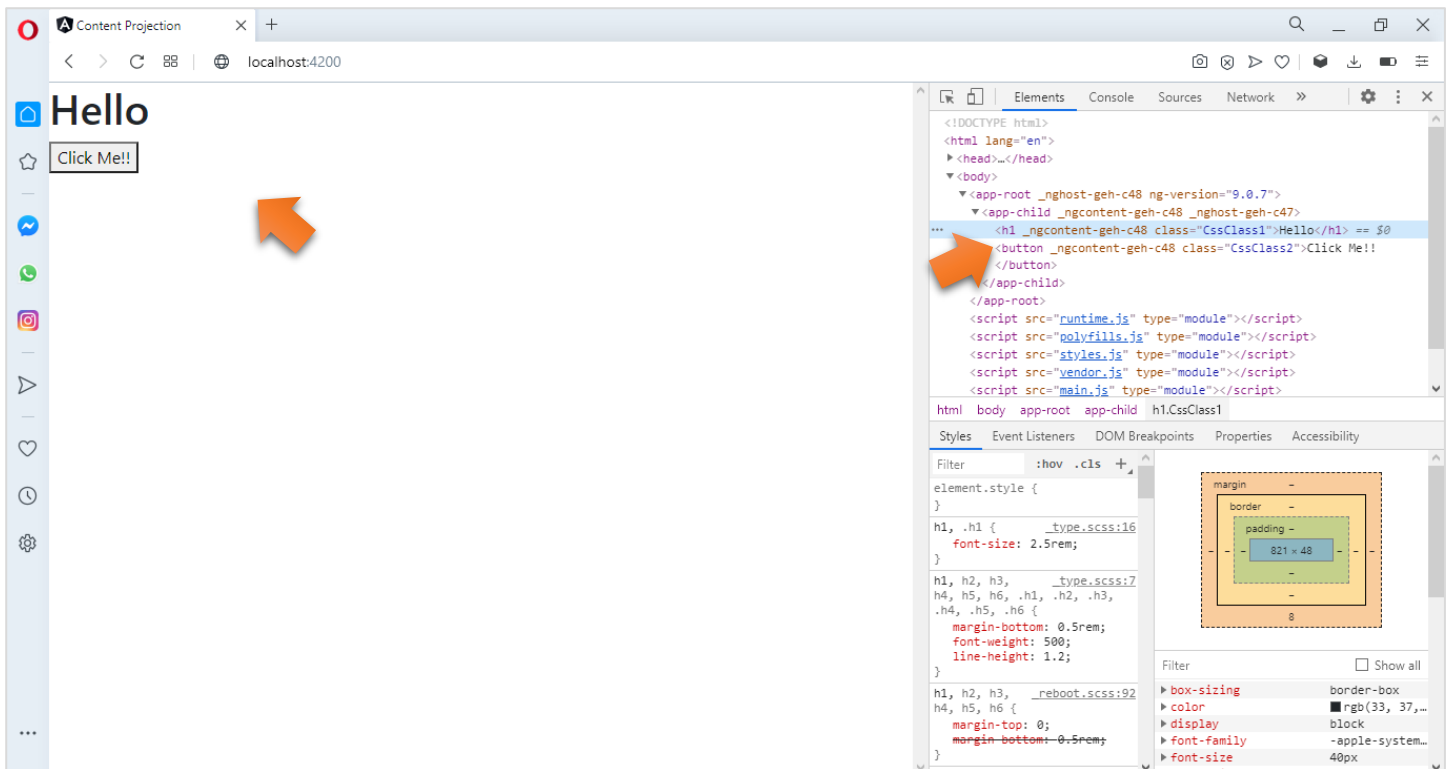
```
<ng-content select=".CssClass1"></ng-content>
<ng-content select=".CssClass2"></ng-content>
```

ونلاحظ لابد ان نضيف نقطة قبل اسم الكلاس واستخدمنا select بدلاً من selector، اما في component الأب فنضيف هذه الكلاسات، كالتالي:

ملف app.component.html

```
<app-child>
  <h1 class="CssClass1">Hello</h1>
  <button class="CssClass2">Click Me!!</button>
</app-child>
```

اما النتيجة، كالتالي:





### :Project using id selector .3.1.4.4

اما هذا النوع فيتم الربط باستخدام id للعنصر، كالتالي:

ملف child.component.html

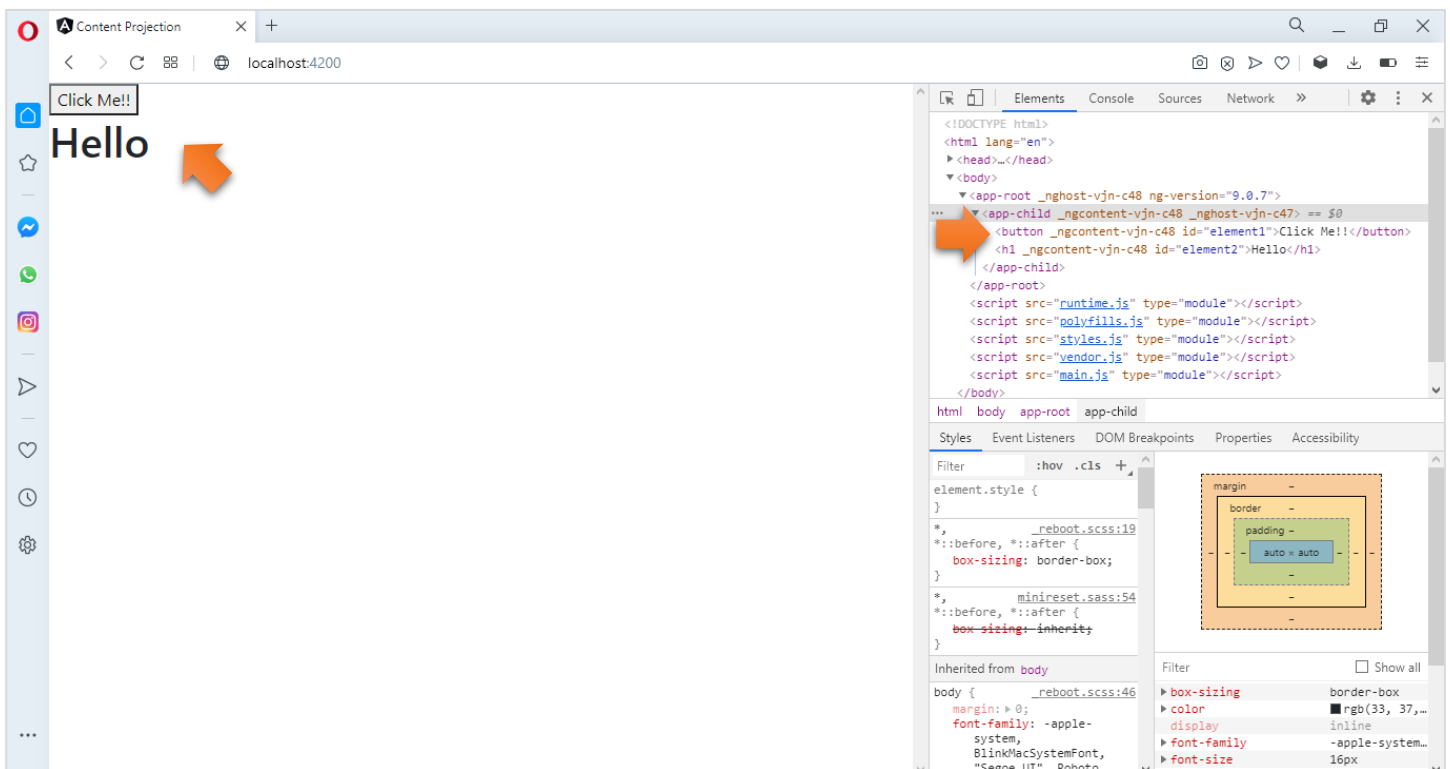
```
<ng-content select="#element1"></ng-content>
<ng-content select="#element2"></ng-content>
```

نلاحظ اضعنا # قبل الاسم المُعطى، اما component الاب فنضيف id، ولكن هذه المرة سوف استبدل العناصر بحيث اجعل element1 للزر و element2 للعنصر h1، لنرى ماذا سوف يحدث، كالتالي:

ملف app.component.html

```
<app-child>
  <h1 id="element2">Hello</h1>
  <button id="element1">Click Me!!</button>
</app-child>
```

اما النتيجة، كالتالي:



نلاحظ انه وضع الزر قبل h1 مع ان h1 قبل الزر في component الأب ولكنه هنا اعتمد على ترتيب ng-content في ملف template لي component الأب.

#### :Project using attribute selector .4.1.4.4

وهنا نستخدم الربط عن طريق الخصائص للعنصر سواء كانت من الخصائص default (مثل autocomplete) او أي خاصية نكتبها نحن فقط للربط، كالتالي:

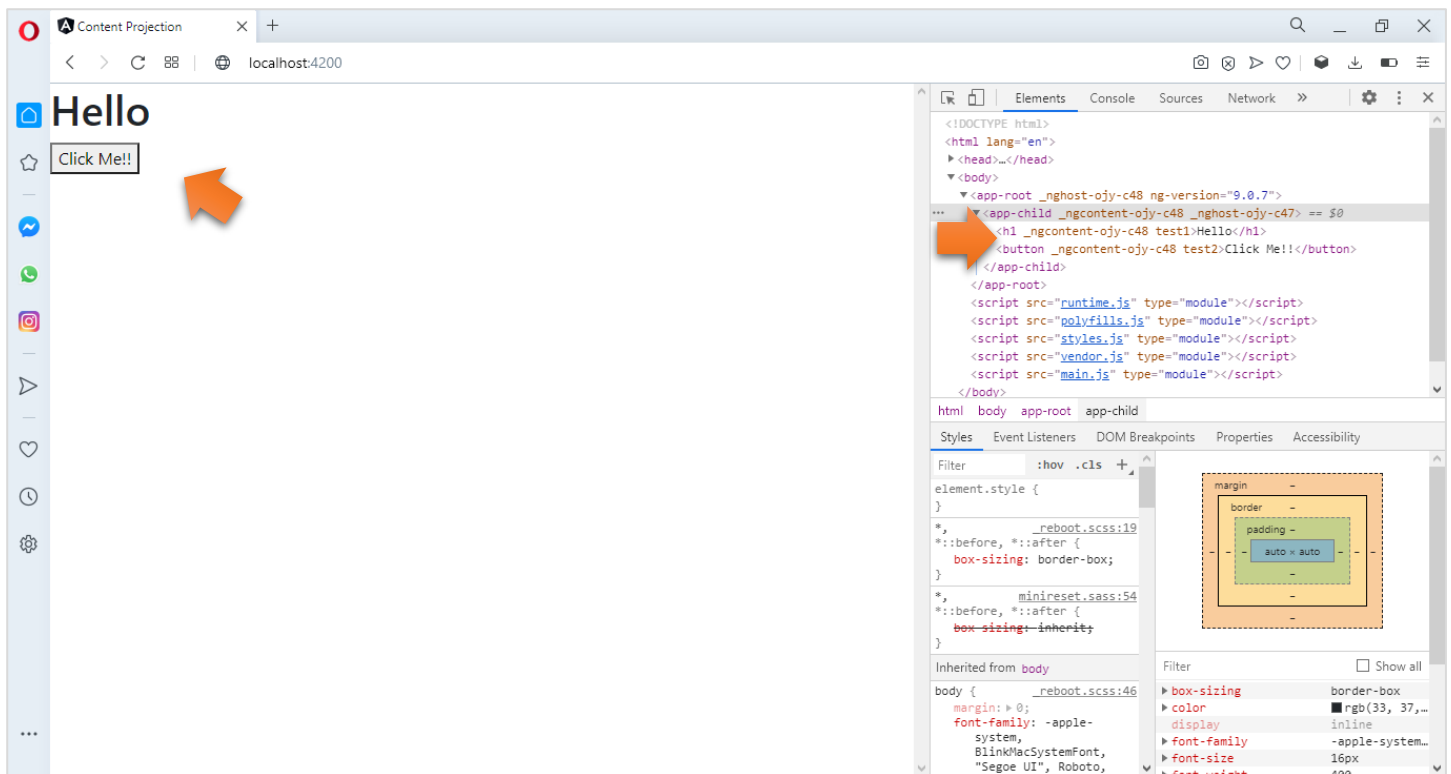
ملف child.component.html

```
<ng-content select="[test1]"></ng-content>
<ng-content select="[test2]"></ng-content>
```

ملف app.component.html

```
<app-child>
  <h1 test1>Hello</h1>
  <button test2>Click Me!!</button>
</app-child>
```

اما النتيجة، فتكون كالتالي:



هنالك بعض الملاحظات يجب الانتباه لها، في حال كان لدينا ربط ng-content بكل عنصر، وهنالك بعض العناصر لم يتم ربطها فإن هذه العناصر لن تظهر ولن يتم بناءها، اما في حال كان لدينا خمس عناصر وتم ربط عنصرين باثنين ng-content ويوجد ng-content ثالث لم يتم فيه أي ربط فان Angular سوف يستبدله بالعناصر الأخرى التي لم تربط، والcomponent ينطبق عليه ما ينطبق على العناصر او وسوم HTML في حال أردنا اضافته من component الأب إلى component الابن.

## :Styling Projected Content .2.4.4

اهم جزء لتنسيق أي عنصر في CSS هو كيفية الوصول إلى هذا العنصر (الوسم) المراد تطبيق التنسيق عليه، لذلك قدمت لنا Angular أمرين تسهل علينا الوصول إلى العنصر وتطبيق التنسيق عليه وهما `host` و `::ng-deep` ولتوضيح لنقوم بتطبيق بعض التنسيقات على المثال السابق، حيث لدينا عنصرين هما `h1` و `button` وعملنا لهم `Content Projection` في `component` الأبن ذو الاسم `child`، وليكن الربط عن طريق `class selector`، كالتالي:

ملف `app.component.html`

```
<app-child>
  <h1 class="test1">Hello</h1>
  <p class="test1">Faisal</p>
  <button class="test2">Click Me!!</button>
</app-child>
```

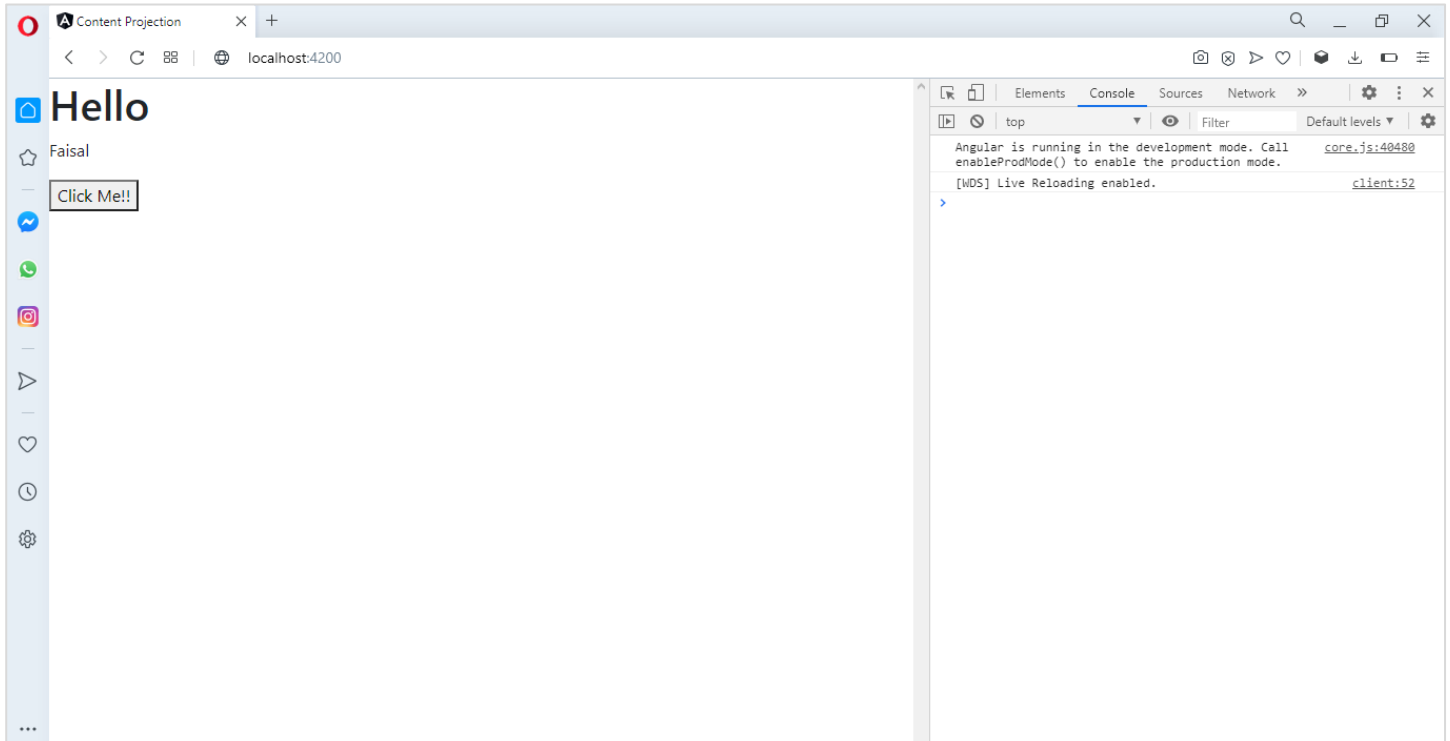
نلاحظ انني قمت فقط بإضافة عنصر جديد `p` وربطته هو والعنصر `h1` بنفس `ng-content` عن طريق الكلاس `test1`.

ملف `child.component.html`

```
<ng-content select=".test1"></ng-content>
<ng-content select=".test2"></ng-content>
```

اما هنا لم نقم بإضافة أي شيء وانما فقط ارجعنا الربط عن طريق `class selector`.

اما النتيجة، فتكون كالتالي:



الآن لنقوم بإجراء بعض التنسيقات، على هذه العناصر في ملف `style` او `css` لي `component` الأبن.

اول خطوة هي الوصول إلى العنصر او الوسم المستضيف لهذه العناصر وهو بكل تأكيد `<app-child></app-child>` لأن Angular سوف يستبدل هذه العناصر بدلاً من `ng-content` في `component` الأبن.

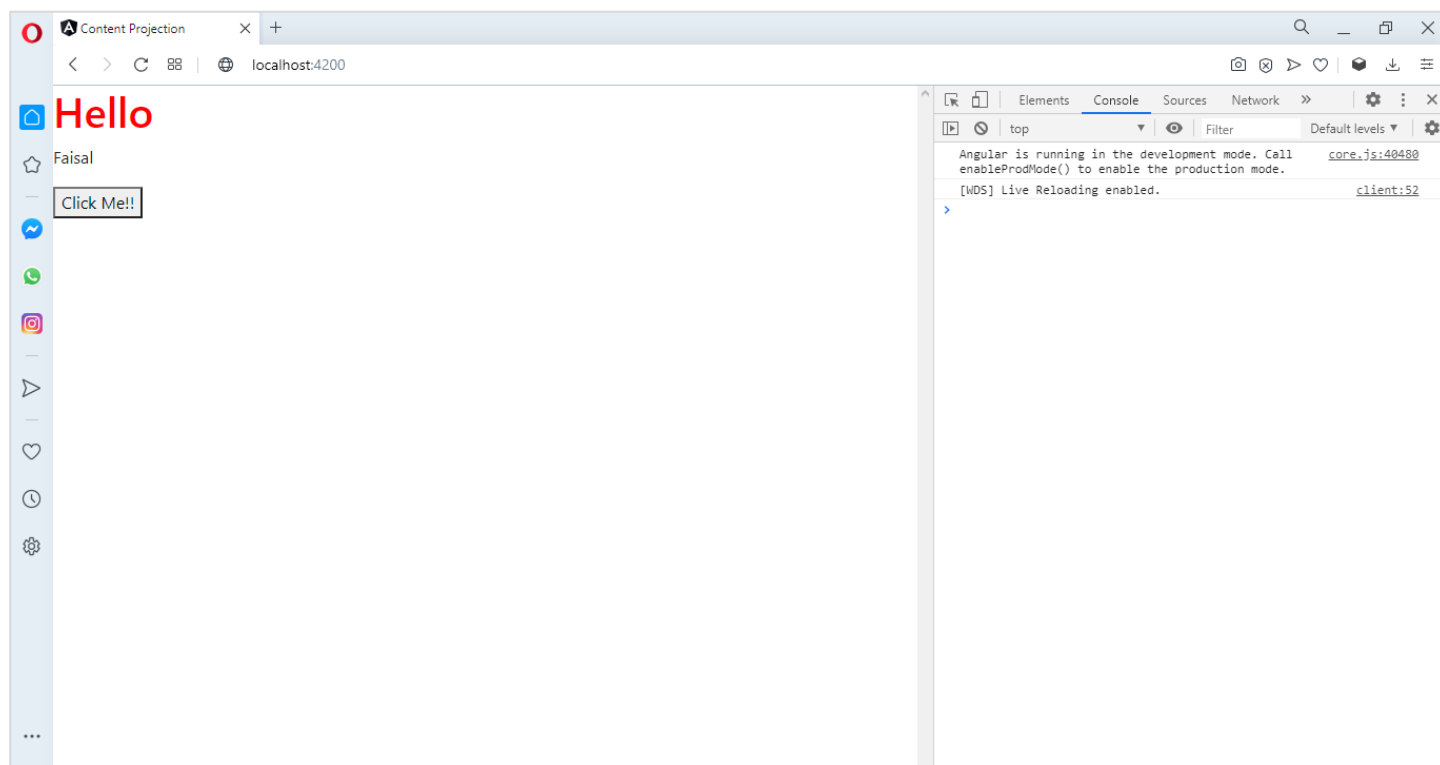
ونستطيع الوصول إلى هذا العنصر المستضيف بكتابة الأمر `host`: ومن ثم نكتب الأمر الثاني `ng-deep`: أي تعمق أو ادخل إلى هذا الوسم أو العنصر، وأخيراً نقوم بكتابة أي عنصر من هذه العناصر نريد تطبيق التنسيق عليه، سواء بكتابة اسم العنصر أو اسم الكلاس أو `id` الخاص به،

ملف `child.component.css`

```
:host ::ng-deep h1 {  
  color: red;  
}
```

الكود السابق بمعناه كأننا نقول لي Angular اذهب إلى العنصر المستضيف `host` والذي هو `<app-child></app-child>` وابحث بداخله عن أي عنصر `h1` وقم بتطبيق التنسيق وهو تغيير لون الخط إلى الأحمر، لذلك سوف يبحث في جميع `ng-content` (وفي مثالنا هنا يوجد اثنين من `ng-content`) عن أي عنصر `h1` ومن ثم يقوم بتغيير اللون إلى الأحمر.

أما النتيجة فتكون كالتالي:

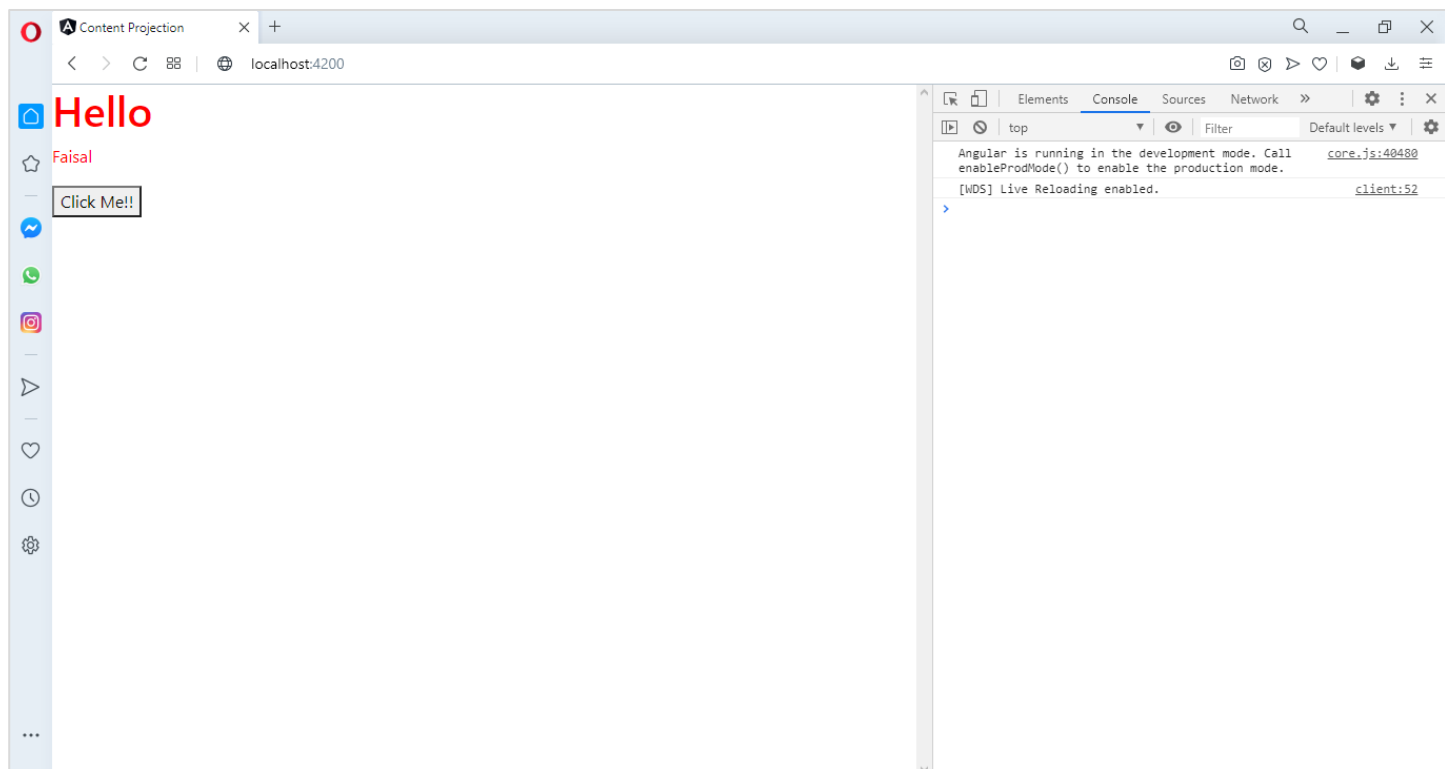


نلاحظ انه قام بتغيير لون الخط للوسم `h1`، وبما انه لا يوجد الا وسم واحد فقد قام بتطبيق التنسيق عليه.

لكن ماذا لو اردنا ان نقوم بتغيير `ng-content` محدد بجميع محتوياته من العناصر، فنستطيع القيام بذلك عن طريق كلاس `css` الخاص بكل `ng-content`، فلو نلاحظ اننا قمنا بربط `ng-content` بالعناصر عن طريق `class` محدد وهو `test1` و `test2` لذلك سوف نستفيد من هذه الكلاسات للوصول إلى العناصر وتطبيق العناصر عليها، كالتالي:

ملف `child.component.css`

```
:host ::ng-deep .test1 {  
  color: red;  
}
```



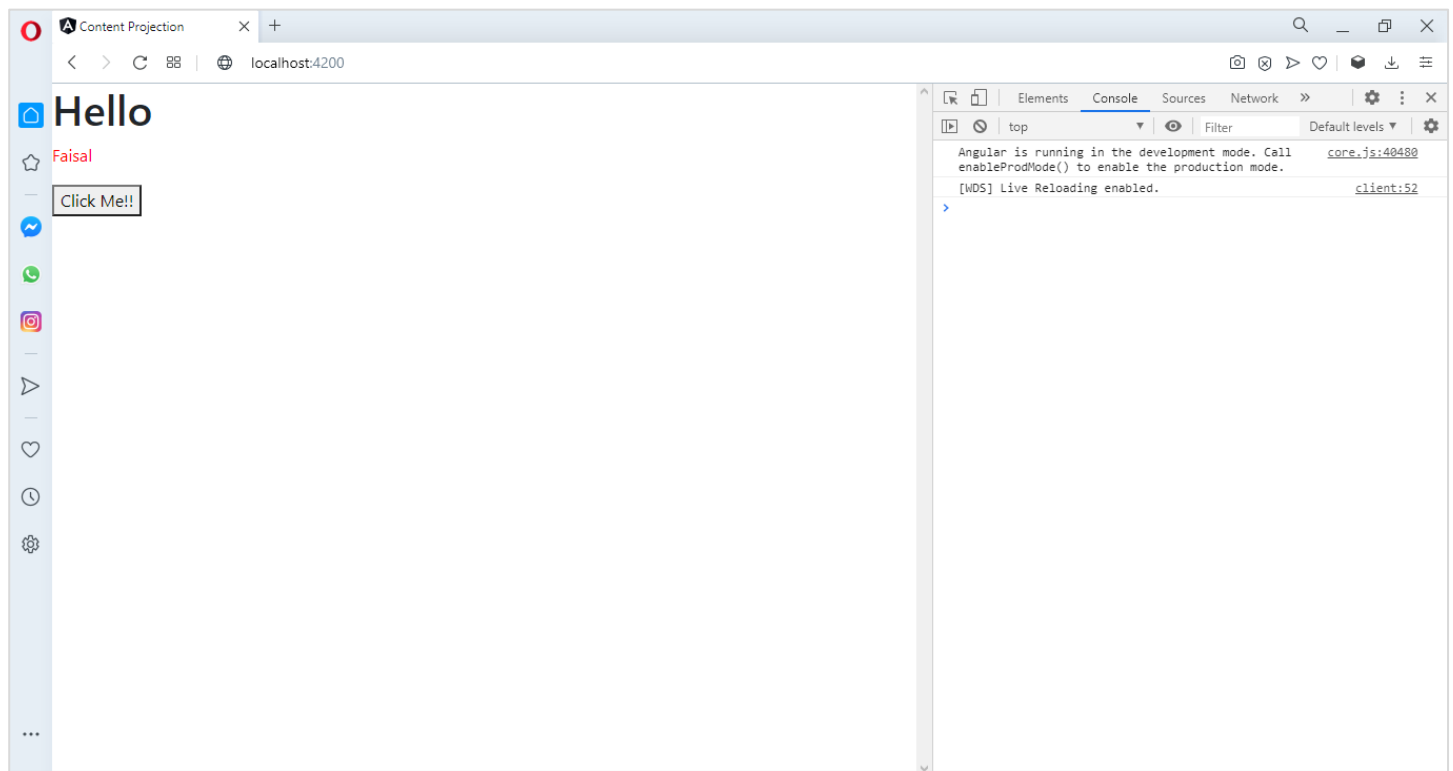
نلاحظ تم تطبيق التنسيقات على جميع العناصر التي تم ربطها بالـ `ng-content` الذي تم ربطه بالكلاس `test1`، لكن ماذا لو اردنا ان نُغير عنصر محدد من مجموعة عناصر مربوطة بنفس `ng-content`، مثلاً نلاحظ ان كلا العنصرين `h1` والعنصر `p` مرتبطين في نفس `ng-content`، ونستطيع القيام بذلك عن طريق إضافة كلاس آخر للعنصر المستهدف وليكن `p` وعن طريق هذا الكلاس نقوم بالوصول إليه، كالتالي:

ملف `app.component.ts`

```
<app-child>
  <h1 class="test1">Hello</h1>
  <p class="test1 test3">Faisal</p>
  <button class="test2">Click Me!!</button>
</app-child>
```

ملف `child.component.css`

```
:host ::ng-deep .test3 {
  color: red;
}
```

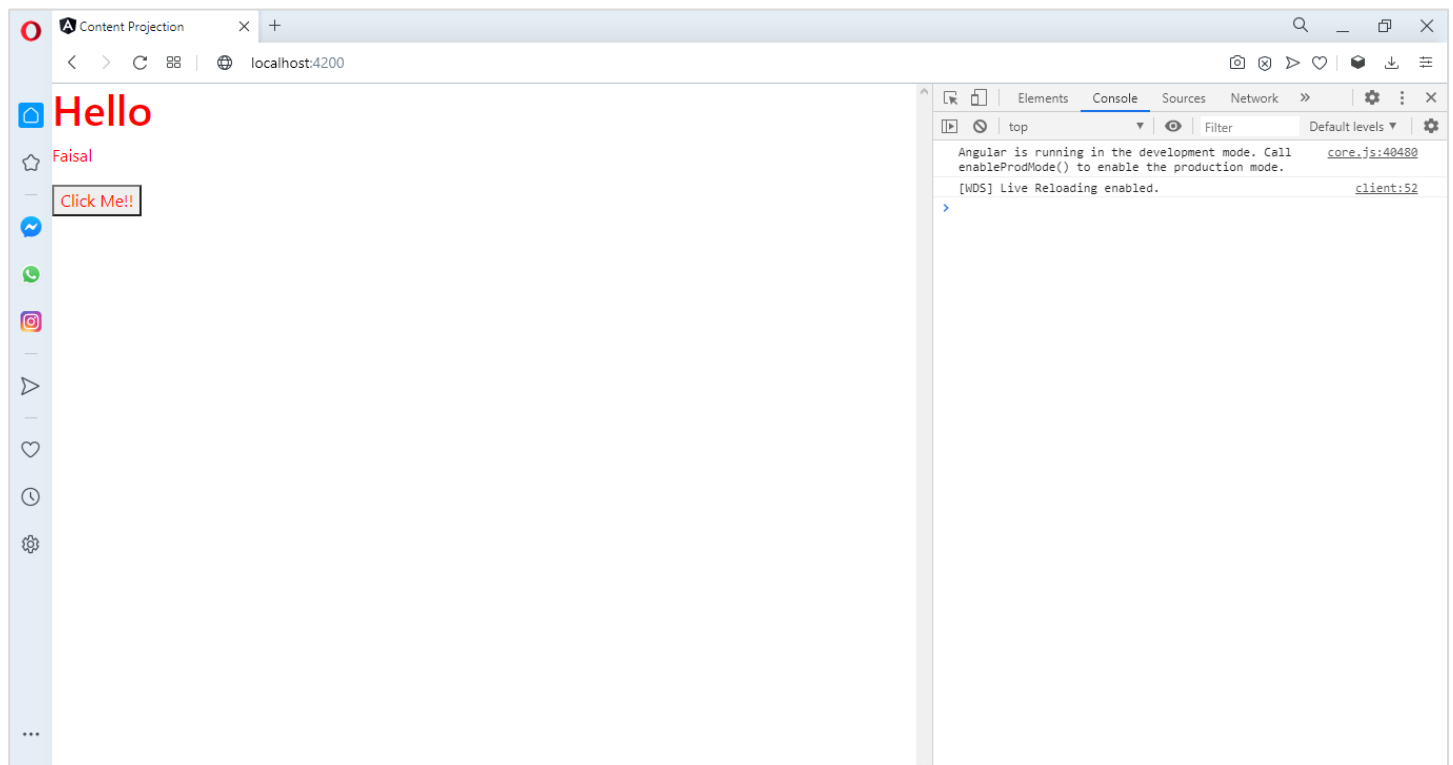


نلاحظ استطعنا الوصول إلى العنصر (الوسم) المحدد وتطبيق التنسيق عليه.

اما لو اردنا ان نقوم بتطبيق تنسيق معين على كافة العناصر بمختلف ng-content، فنستطيع ذلك عن طريق كلاسات css، كالتالي:

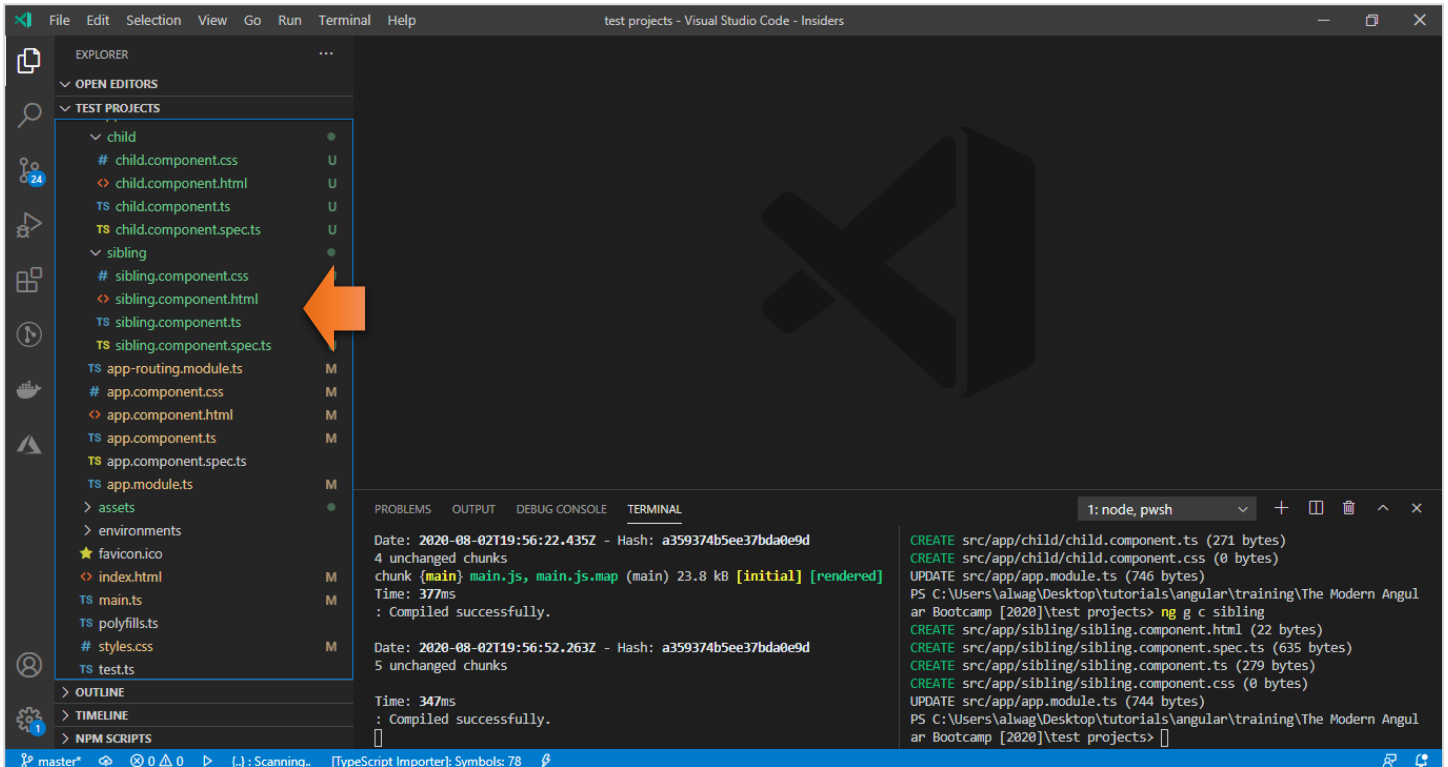
ملف child.component.css

```
:host ::ng-deep .test1,
:host ::ng-deep .test2 {
  color: red;
}
```



### :Access Projected Content from Child Component Class .3.4.4

نستطيع الوصول إلى جميع العناصر والمحتويات التي عملنا لها Project عن طريق ملف class لي component الأب، ويتم ذلك بنفس الآلية التي كُنّا نستخدمها سابقاً ViewChild و ViewChildren ولكن هنا نستخدم ContentChild و ContentChildren، ولتوضيح لنقوم بإعطاء مثال، وليكن نفس المثال السابق، وهو وجود ثلاث عناصر واحد h1 والثاني p وكلاهما مرتبطين بنفس ng-content اما الثالث فهو button ولنضيف ايضاً component آخر وليكن اسمه sibling لنضيفه من الأب إلى الأب، كالتالي:



#### ملف app.component.html

```
<app-child>
  <h1 class="test1">Hello</h1>
  <p class="test1 test3">Faisal</p>
  <button class="test2">Click Me!!</button>
  <app-sibling></app-sibling>
</app-child>
```

#### ملف child.component.html

```
<ng-content select=".test1"></ng-content>
<ng-content select=".test2"></ng-content>
<ng-content selector="app-sibling"></ng-content>
```

اما الآن لنقوم بشرح كيفية الوصول إلى أحد هذه العناصر عن طريق ملف class لي component الأب وليكن هذا العنصر هو h1، وأول خطوة نقوم بها هي إعطاء هذا العنصر Template Reference، وليكن اسمه H1Element كالتالي:

#### ملف app.component.html

```
<app-child>
  <h1 class="test1" #H1Element>Hello</h1>
```

```
<p class="test1 test3">Faisal</p>
<button class="test2">Click Me!!</button>
<app-sibling></app-sibling>
</app-child>
```

أما في ملف class لي component الأب، فنقوم بالخطوة التالية:

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})

export class ChildComponent implements OnInit {
  @ContentChild('H1Element') h1Element: ElementRef;

  constructor() { }

  ngOnInit(): void { }
}
```

نلاحظ انه مشابهه لطريقة تعاملنا مع ViewChild ولكن هنا باستخدام ContentChild، والسؤال هنا في أي دالة من دوال Lifecycle Hooks يتم كتابة Logic الخاص بالتعامل مع هذا العنصر، فلدينا في ViewChild نكتب Logic في دالة ngAfterViewInit اما Logic الخاص بالInput فيفضل استخدامه في الدالة ngOnChanges، والإجابة على هذا السؤال تُعيدنا إلى الجزء الخاص بدوال Lifecycle Hooks حيث قلنا أن الدالة ngAfterContentInit و ngAfterContentChecked يتم استدعائهما قبل الدالتين ngAfterViewInit و ngAfterViewChecked، والدالتين الأولى مرتبطتين بالمحتوى content المضاف إلى هذا component من خارجه عن طريق @ContentChild و @ContentChildren، لذلك سوف نستخدم الدالة ngAfterContentInit لتعامل مع أي Logic عن طريق @ContentChild و @ContentChildren، اما الدالة ngAfterContentChecked فهي لتعامل مع Change Detection، ولنقوم بكتابة Logic يقوم بقراء محتوى العنصر h1، كالتالي:

ملف child.component.ts

```
import {
  Component,
  ContentChild,
  OnInit,
  AfterContentInit,
  ElementRef,
```



```

} from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})

export class ChildComponent implements OnInit, AfterContentInit {
  @ContentChild('H1Element') h1Element: ElementRef;

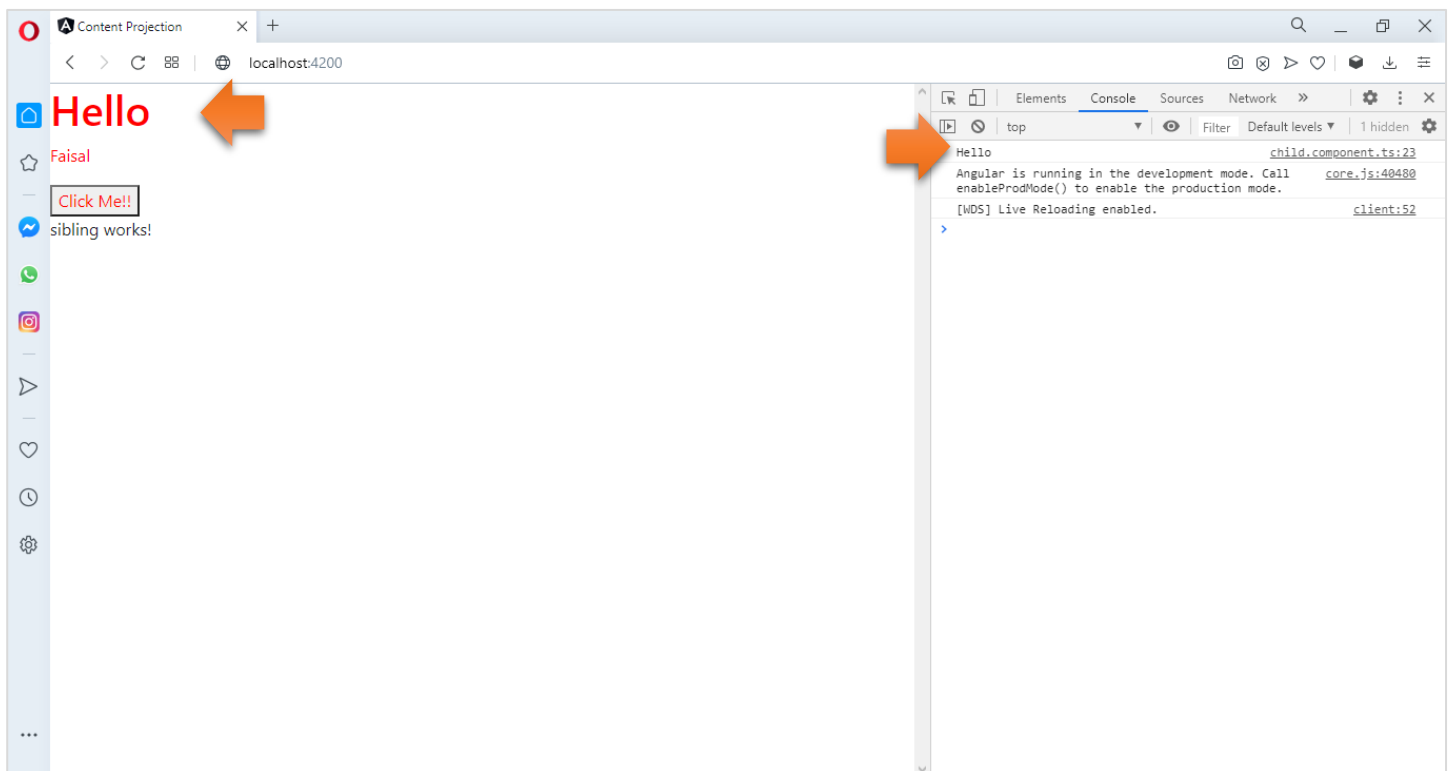
  constructor() { }

  ngOnInit(): void { }

  ngAfterContentInit(): void {
    console.log(this.h1Element.nativeElement.textContent);
  }
}

```

اما النتيجة، فهي كالتالي:



ألاّن لنزيد الجرعة قليلاً لنقوم بالوصول إلى ملف class في component ذو الاسم sibling عن طريق ملف class في component الأب، وبما اننا عملنا Content Projection في component ذو الاسم sibling فنستطيع بذلك ان نعمل له @ContentChild داخل component الأب، كالتالي:

```

// ملف child.component.ts
import { SiblingComponent } from '../sibling/sibling.component';
import {
  Component,

```

```

ContentChild,
OnInit,
AfterContentInit,
ElementRef,
} from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})
export class ChildComponent implements OnInit, AfterContentInit {
  @ContentChild('H1Element') h1Element: ElementRef;
  @ContentChild(SiblingComponent) siblingComponent: SiblingComponent;

  constructor() { }

  ngOnInit(): void { }

  ngAfterContentInit(): void {
    console.log(this.h1Element.nativeElement.textContent);
  }
}

```



والآن لنقوم بإضافة خاصية (متغير) داخل ملف class لي component ذو الاسم sibling وليكن اسمها name ومن ثم نعمل لها binding عن طريق interpolation في ملف template لنفس component، كالتالي:

ملف sibling.component.ts

```

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

@Component({
  selector: 'app-sibling',
  templateUrl: './sibling.component.html',
  styleUrls: ['./sibling.component.css']
})
export class SiblingComponent implements OnInit {

  name = 'Hello From Sibling Component';

  constructor() { }

  ngOnInit(): void {

  }
}

```

ملف sibling.component.html

```
<p>{{ name }}</p>
```

الآن لنرجع إلى ملف class لي component الأب ونقوم بالوصول مباشرة إلى هذه الخاصية name ولنقوم بتغيير قيمتها، وليكن عن طريق زر وهذا الزر عند الضغط عليه يُنفذ دالة معينة، وهذه الدالة تقوم بتغيير هذه الخاصية، كالتالي:

ملف child.component.html

```
<ng-content select=".test1"></ng-content>
<ng-content select=".test2"></ng-content>
<ng-content selector="app-sibling"></ng-content>
<button (click)="changeName()">
  Change Name Property From Child Component
</button>
```

```
import { SiblingComponent } from '../sibling/sibling.component';
import {
  Component,
  ContentChild,
  OnInit,
  AfterContentInit,
  ElementRef,
} from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})
export class ChildComponent implements OnInit, AfterContentInit {
  @ContentChild('H1Element') h1Element: ElementRef;
  @ContentChild(SiblingComponent) siblingComponent: SiblingComponent;

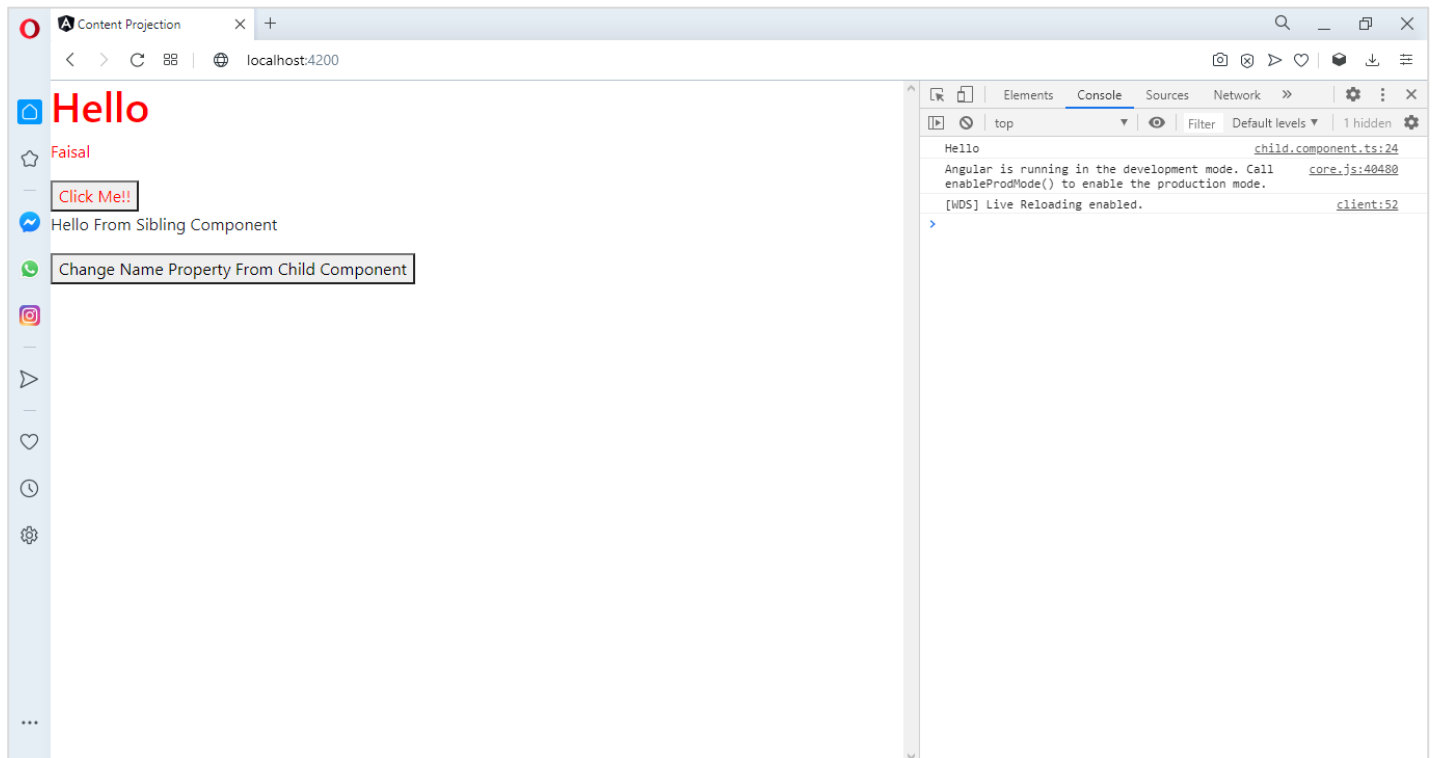
  constructor() { }

  ngOnInit(): void { }

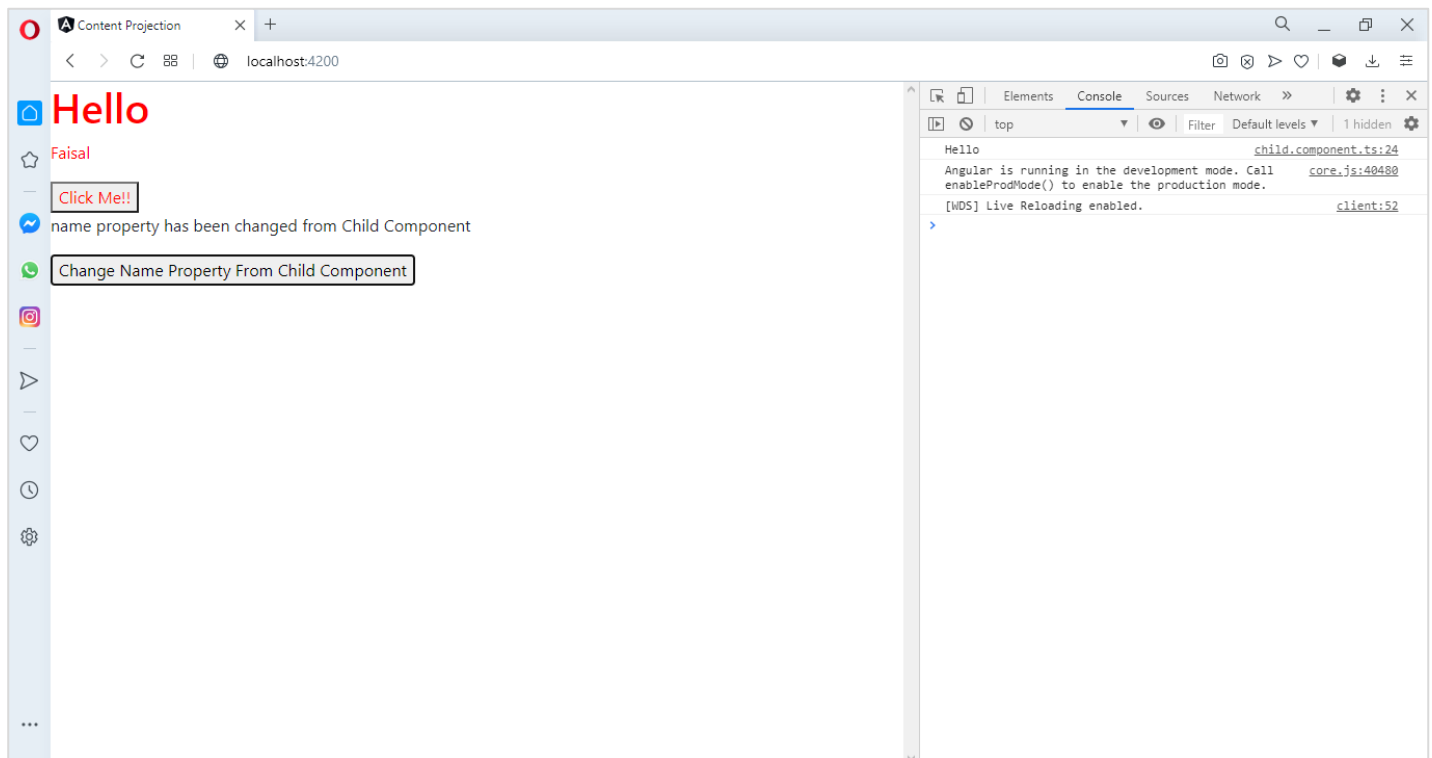
  ngAfterContentInit(): void {
    console.log(this.h1Element.nativeElement.textContent);
  }

  changeName() {
    this.siblingComponent.name = 'name property has been changed from Child Component';
  }
}
```

اما النتيجة فتكون كالتالي:



الآن لنضغط على الزر ولنرى النتيجة:



نلاحظ تم تغيير الخاصية، وبذلك تعلمنا كيف نصل من ملف class إلى ملف class آخر عن طريق `@ContentChild`، ولكن ماذا لو كان لدينا مجموعة من العناصر ونريد التعامل معها جميعاً ففي هذه الحالة لا بد ان نستخدم `@ContentChildren`، وهي مشابهة أيضاً بكافة تفاصيلها مع `@ViewChildren`، لذلك منعاً لتكرار سوف أقوم بذكرها فقط بدون الغوص بالتفاصيل، وأول خطوة نقوم بها هي إضافة Template Reference مشابه لأكثر من عنصر، كالتالي:

ملف `app.component.html`

`<app-child>`

```

<h1 class="test1" #H1Element>Hello</h1>
<p class="test1 test3" #Elements>Faisal</p>
<button class="test2" #Elements>Click Me!!</button>
<app-sibling></app-sibling>
</app-child>

```

ومن ثم نعمل Access لهذه العناصر عن طريق ملف class لي component الأب من طريق @ContentChildren، وبنفس الوقت نعمل console لهذه العناصر، كالتالي:

```

ملف child.component.ts
import { SiblingComponent } from '../sibling/sibling.component';
import {
  Component,
  ContentChild,
  OnInit,
  AfterContentInit,
  ElementRef,
  QueryList,
  ContentChildren
} from '@angular/core';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})
export class ChildComponent implements OnInit, AfterContentInit {
  @ContentChild('H1Element') h1Element: ElementRef;
  @ContentChild(SiblingComponent) siblingComponent: SiblingComponent;
  @ContentChildren('Elements') elements: QueryList<ElementRef>;
  constructor() { }

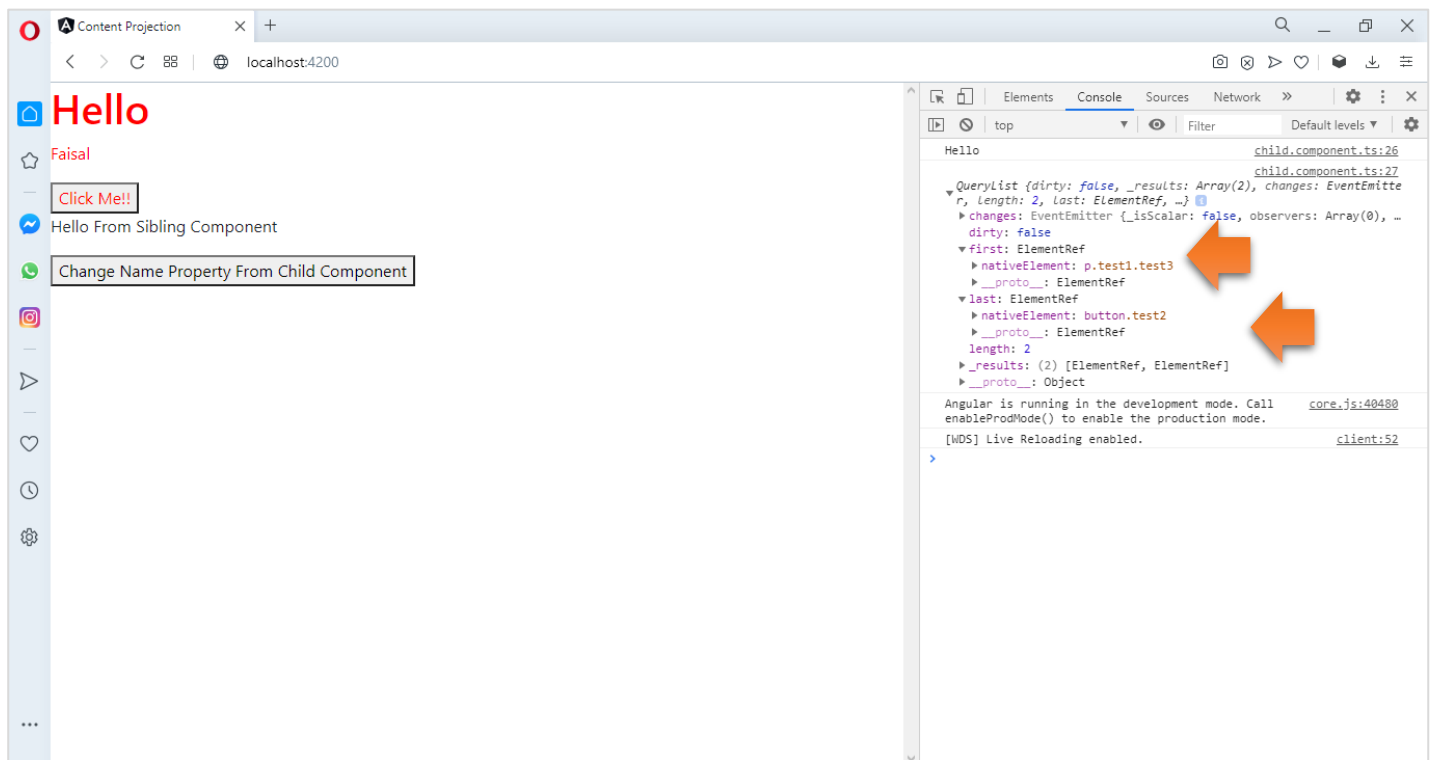
  ngOnInit(): void { }

  ngAfterContentInit(): void {
    console.log(this.h1Element.nativeElement.textContent);
    console.log(this.elements);
  }

  changeName() {
    this.siblingComponent.name = 'name property has been changed from Child Component';
  }
}

```

اما النتيجة فهي كالتالي:



نلاحظ العنصرين تم طباعتهم في console وبذلك تستطيع ان تعمل فيهم ما تشاء سواء بعمل Loop عن طريق forEach او  
نعمل لها subscribe ومراقبة التغيرات التي تطرأ عليها، الخ، كما تعلمنا سابقاً في @ContentChildren.

## 5.4 ng-container & ng-template :

ng-template هو وسم (عنصر) خاص بي Angular ويستخدم عادة مع Structure Directives مثل \*ngIf او \*ngSwitch،  
ولعل من أشهر استخداماتها عند استخدام else مع \*ngIf (ولفهم أعمق عن Structure Directives الرجاء مراجعة الكتاب  
الثالث من هذه السلسلة Angular Pipes and Directives)، ولتوضيح لنعطي المثال التالي:

```

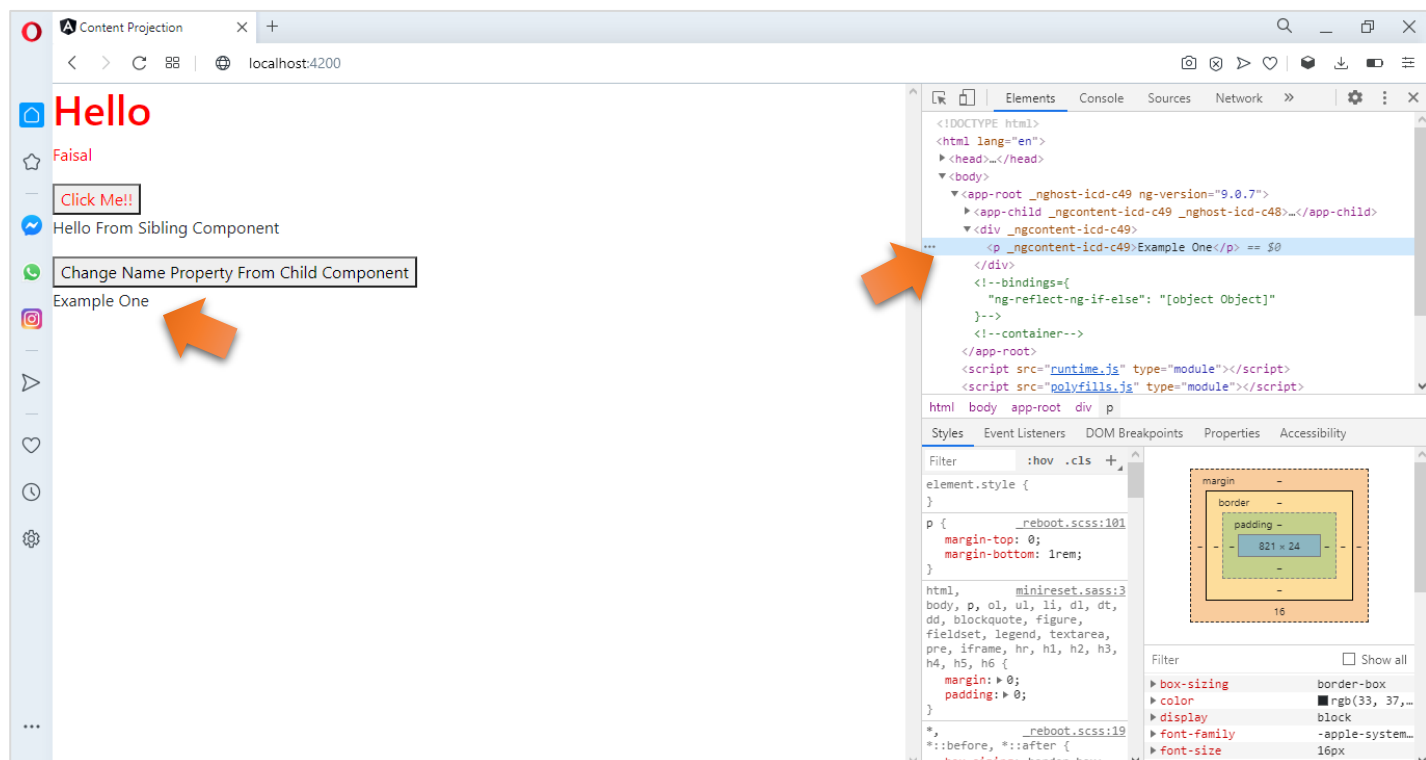
ملف app.component.html
<app-child>
  <h1 class="test1" #H1Element>Hello</h1>
  <p class="test1 test3" #Elements>Faisal</p>
  <button class="test2" #Elements>Click Me!!</button>
  <app-sibling></app-sibling>
</app-child>

<div *ngIf="true; else test">
  <p>Example One</p>
</div>

<ng-template #test>
  <p>Example Two</p>
</ng-template>

```

نلاحظ انه في المثال السابق وضعنا شرط على العنصر div بحيث إذا كان القيمة true قم ببناء هذا العنصر بالإضافة إلى جميع ما يحتويه بداخله وفي مثالنا هنا يوجد فقط العنصر p اما إذا كان false فقم ببناء العنصر الذي يحتوي على Template Reference ذو الاسم #test، ولنشاهد النتيجة على المتصفح:



نلاحظ تم بناء div و p في DOM، الآن لنقم بتغيير القيمة من true إلى false، كالتالي:

```

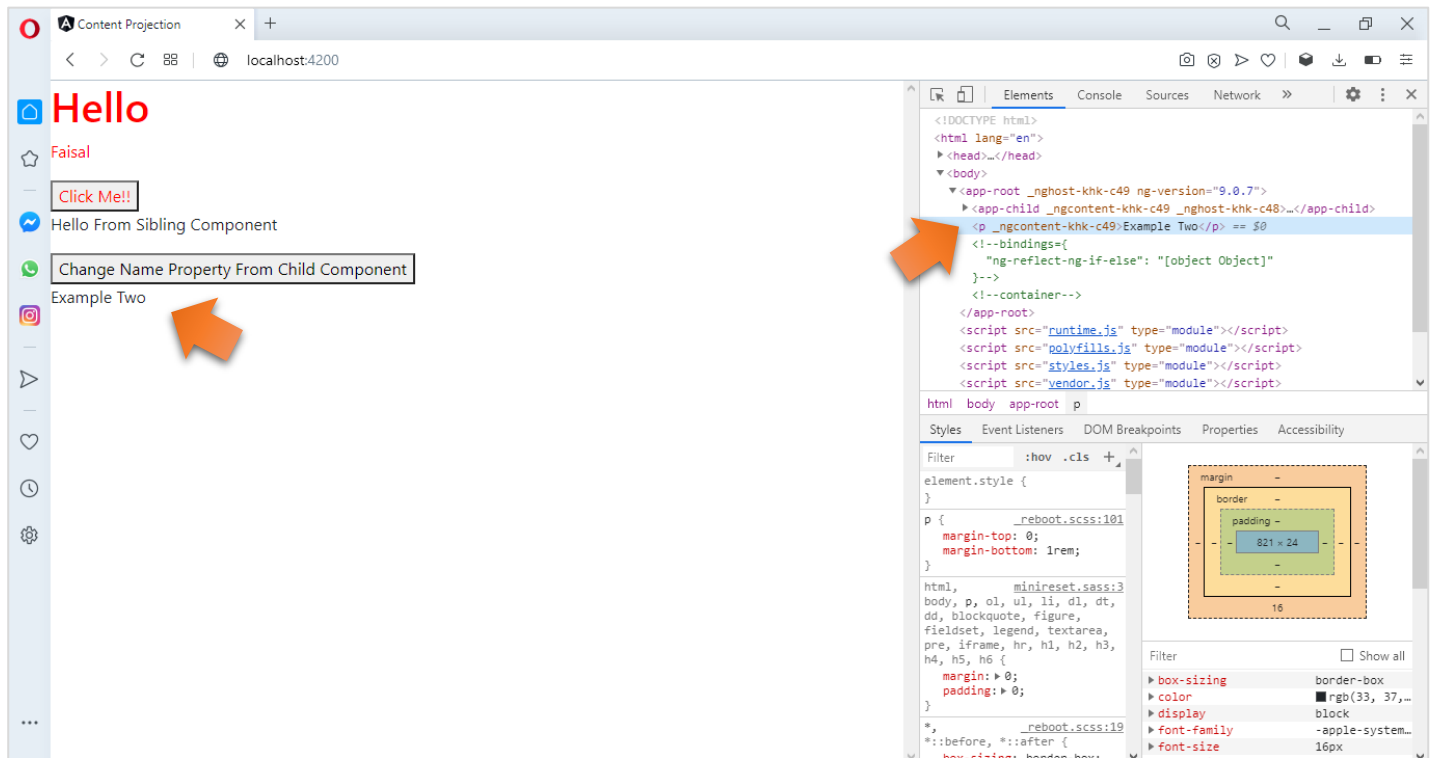
app.component.html ملف
<app-child>
  <h1 class="test1" #H1Element>Hello</h1>
  <p class="test1 test3" #Elements>Faisal</p>
  <button class="test2" #Elements>Click Me!!</button>
  <app-sibling></app-sibling>
</app-child>

<div *ngIf="false; else test">
  <p>Example One</p>
</div>

<ng-template #test>
  <p>Example Two</p>
</ng-template>

```

اما النتيجة في المتصفح فتكون كالتالي:



أتمنى منك عزيزي المتعلم ان تكون ذو نظرة ثاقبة ولاحظت ان العنصر ng-template لم يتم بناءه في DOM وانما يتم استخدامه فقط بشكل افتراضي وهذا الأمر يُعتبر من أهم خصائصه.

الآن لنقم بتغيير الوسم من ng-template إلى ng-container، مع إرجاع قيمة الشرط \*ngIf إلى true ولنرى ما الذي سوف يحدث، كالتالي:

```

ملف app.component.html
<app-child>
  <h1 class="test1" #H1Element>Hello</h1>
  <p class="test1 test3" #Elements>Faisal</p>
  <button class="test2" #Elements>Click Me!!</button>
  <app-sibling></app-sibling>
</app-child>

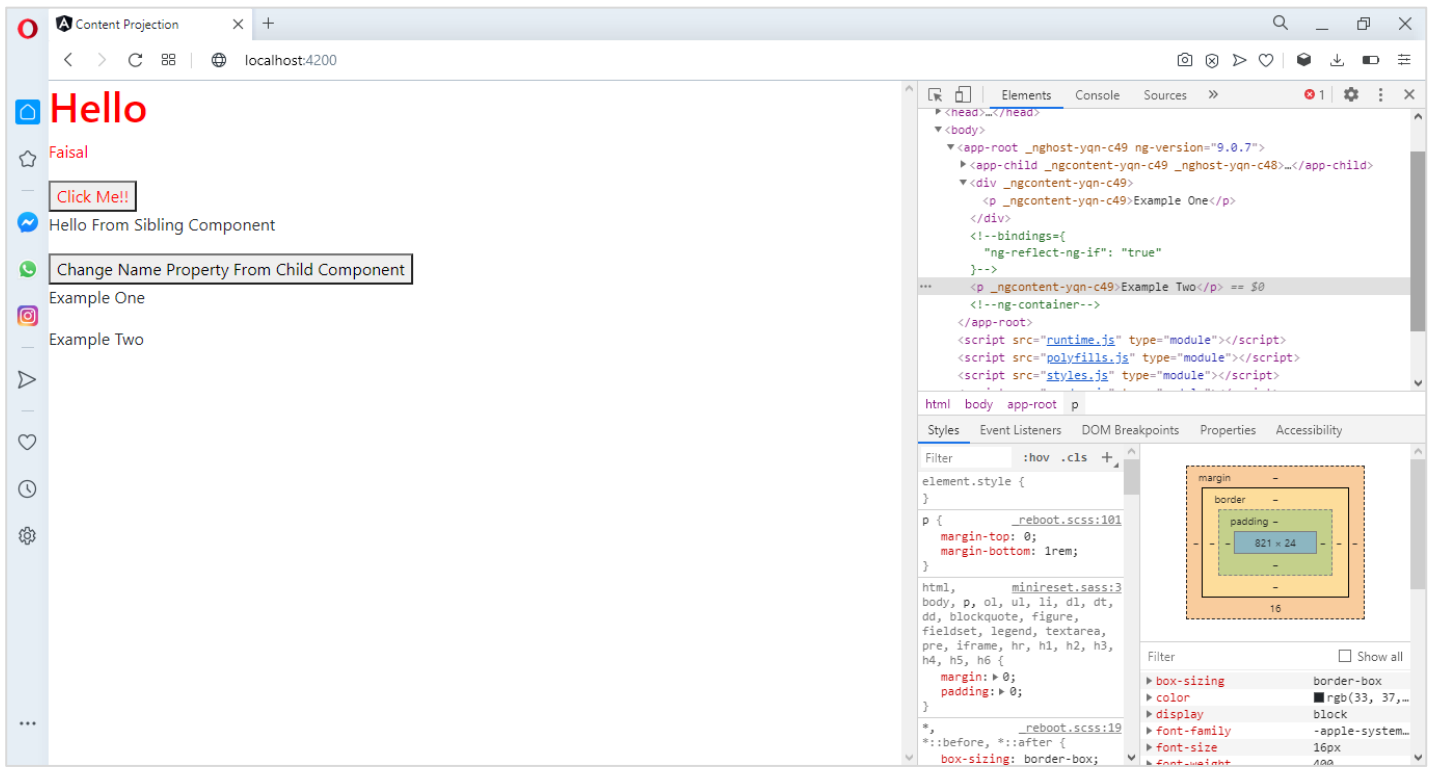
<div *ngIf="true; else test">
  <p>Example One</p>
</div>

<ng-container #test>
  <p>Example Two</p>
</ng-container>

```

اما النتيجة في المتصفح، فتكون:





نلاحظ ان كلا العنصرين تم بنائهم في DOM، اما ng-container ايضاً هي الأخرى لم يتم بناءها في DOM، ومن هنا نستنتج ان ng-template نستخدمها في حال أردنا ان نخفي Markup معين ونظهره في حال تحقق شرط معين اما ng-container فيتم استخدامها لعمل grouping (تجميع) لعناصر معينة، وكلا هذين العنصرين ng-template و ng-container لـ Angular لن يقوم ببنائهم في DOM، وانما هما افتراضيتان، اما استخدامات ng-container فهي متعددة منها في حال كان لدينا أكثر من Structure Directive لنفس العنصر فنستخدم ng-container، ولنعطي مثال لتوضيح:

```

app.component.html ملف
<app-child>
  <h1 class="test1" #H1Element>Hello</h1>
  <p class="test1 test3" #Elements>Faisal</p>
  <button class="test2" #Elements>Click Me!!</button>
  <app-sibling></app-sibling>
</app-child>

<div *ngIf="true; else test">
  <p>Example One</p>
</div>

<ng-template #test>
  <p>Example Two</p>
</ng-template>

<ng-container *ngFor="let item of [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]">
  <p *ngIf="item % 2 === 0">{{ item }}</p>
</ng-container>

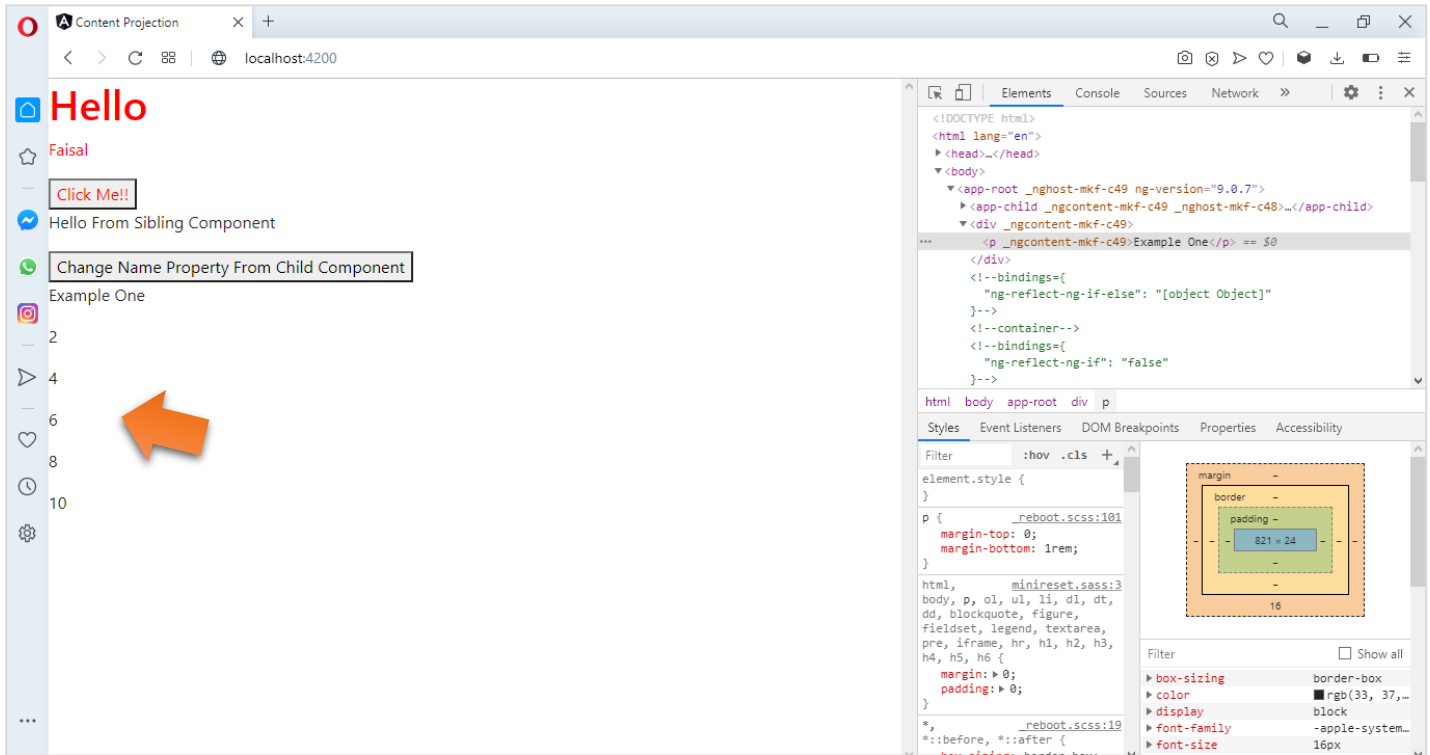
```

نلاحظ انه لدينا اثنين Structure Directives واحد ngFor والثاني ngIf لذلك لا نستطيع كتابتهما بالطريقة التالية:

```
<p *ngFor="let item of [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]" *ngIf="item % 2 === 0">
  {{ item }}
</p>
```

لذلك نقوم بفصلهما باستخدام ng-container.

اما النتيجة فتكون بالطريقة التالية:



نلاحظ قام بعمل Loop ومن ثم اظهر فقط الاعداد الزوجية بحسب الشرط الذي وضعناه.

## 6.4. NgTemplateOutlet:

وتستخدم هذه الميزة في الأساس لعرض جزء معين من ملف template في أماكن متفرقة في نفس الملف، فمثلاً لو كان لدينا شعار Logo لمؤسسة معينة، وهذا الشعار يتكرر عرضه بنفس ملف template أكثر من مرة، فبدلاً من كتابة الكود الخاص بهذا الشعار أكثر من مرة نقوم بكتابة مرة واحدة ومن ثم نستدعيه في أي جزء نريده من هذا الملف، مع العلم ان الكود Markup الخاص بالشعار نكتبه في ng-template اما الاستدعاء لهذا الكود في أجزاء الملف المختلفة فنستخدم ng-container مع إضافة ngTemplateOutlet\*، ولتوضيح لنعطي مثال ولنفرض انه لدينا شعار معين يتم عرضه في ثلاث أجزاء من الصفحة في الجزء العلوي وفي الجزء السفلي وفي الجزء الخاص بنموذج معين، ولنضع الصورة في مجلد assets، وسوف استعين بمكتبة bootstrap، لتسهيل اجراء التنسيق وبناء الصفحة، كالتالي:

وفي ملف template نضيف Markup التالي:

```
app.component.html
<nav class="navbar navbar-dark bg-dark nav-fill">
  <a class="navbar-brand" href="#">
    
```

1

```

        width="30"
        height="30"
        class="d-inline-block align-top"
    />
    Logo
</a>
<a class="btn btn-info" href="#">Section One</a>
<a class="btn btn-info" href="#">Section Two</a>
<a class="btn btn-info" href="#">Section Three</a>
</nav>

<div class="card w-75 m-auto">
    <div class="card-body">
        <div class="card-header">
            <a class="navbar-brand" href="#">
                
                Logo
            </a>
        </div>
        <h5 class="card-title"> </h5>
        <p class="card-text">
            Some quick example text to build on the card title and make up the bulk
            of the card's content.
        </p>
        <a href="#" class="btn btn-primary">Go somewhere</a>
    </div>
</div>

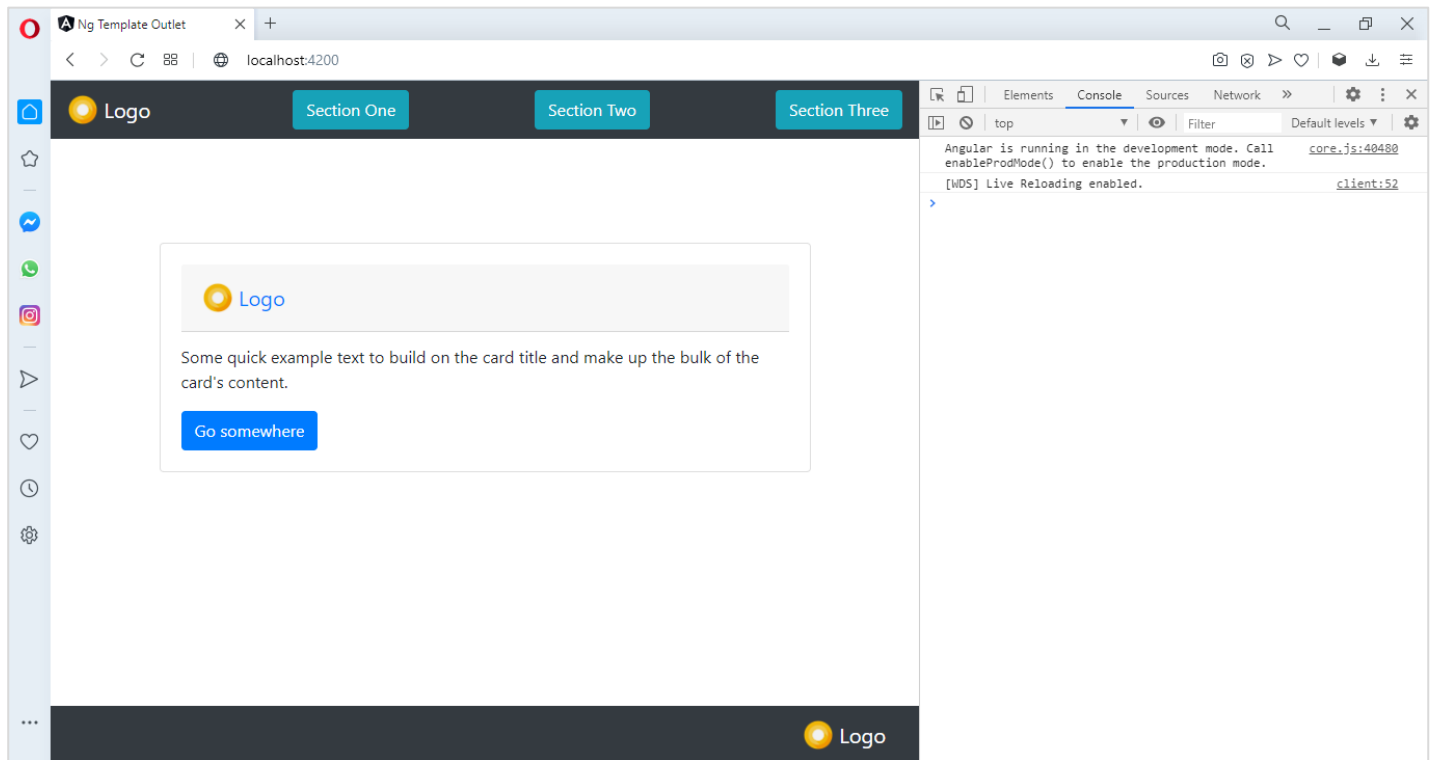
<nav class="navbar fixed-bottom navbar-dark bg-dark d-flex flex-row-reverse">
    <a class="navbar-brand" href="#">
        
        Logo
    </a>
</nav>

```

2

3

نلاحظ ان الجزء الخاص بعرض الشعار تم تكراره في ثلاث مواضع، بحيث تكون النتيجة في المتصفح كالتالي:



الآن لنستخدم ميزة ngTemplateOutlet، كالتالي:

```

app.component.html ملف
<nav class="navbar navbar-dark bg-dark nav-fill">
  <ng-container *ngTemplateOutlet="logo"></ng-container> 1
  <a class="btn btn-info" href="#">Section One</a>
  <a class="btn btn-info" href="#">Section Two</a>
  <a class="btn btn-info" href="#">Section Three</a>
</nav>

<div class="card w-75 m-auto">
  <div class="card-body">
    <div class="card-header">
      <ng-container *ngTemplateOutlet="logo"></ng-container> 2
    </div>
    <h5 class="card-title"> </h5>
    <p class="card-text">
      Some quick example text to build on the card title and make up the bulk
      of the card's content.
    </p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>

<nav class="navbar fixed-bottom navbar-dark bg-dark d-flex flex-row-reverse">
  <a class="navbar-brand" href="#">
    <ng-container *ngTemplateOutlet="logo"></ng-container> 3
  </a>
</nav>

<ng-template #logo>
  <a class="navbar-brand" href="#">

```

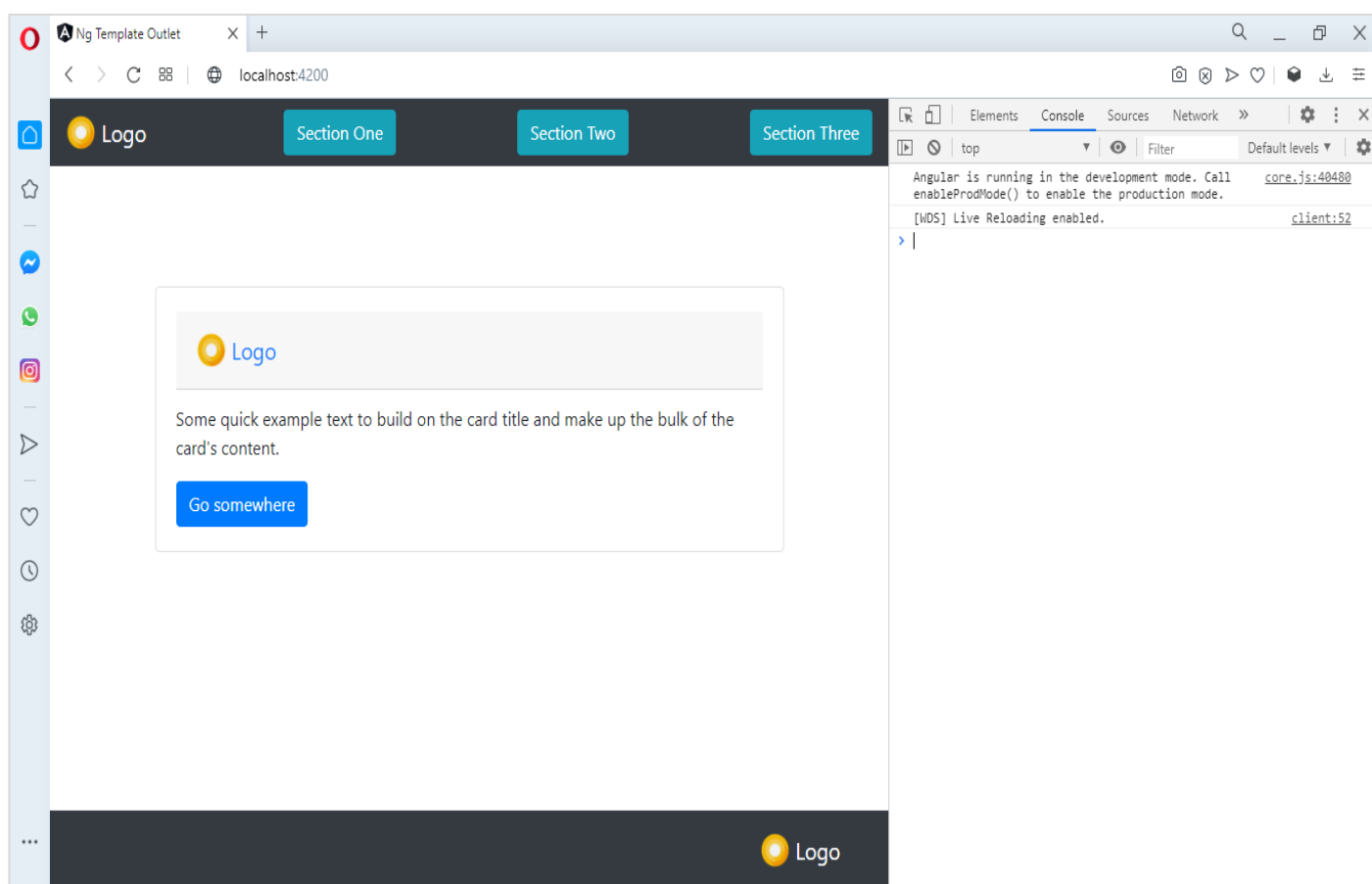
```


Logo
</a>
</ng-template>

```

نلاحظ اننا في النقطة 4 قمنا بكتابة Markup الخاص بعرض الصورة مرة واحدة في ng-template وبنفس الوقت اعطيناه Template Reference باسم logo، اما في النقاط 1,2,3 قمنا باستدعاء هذا Markup عن طريق ng-container وربطنا كل واحد منها بـ Template Reference الخاص بي ng-template عن طريق \*ngTemplateOutlet.

والنتيجة سوف تظهر كما هي:



كما ان إمكانيات هذه الميزة لا تتوقف عند تكرار جزء معين أكثر من مرة في template، بل تُعطينا إمكانيات أخرى منها ان نقوم بربطها مع متغيرات في ملف class، لكي تقوم بعرض بيانات ديناميكية، كأن يكون الشعار بياناته نأخذها من قاعدة بيانات، كالتالي:

ملف app.component.ts

```

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

@Component({
  selector: 'app-root',

```

```

    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
  })

export class AppComponent implements OnInit {

  logoInfo = { url: 'assets/Logo.png', name: 'Logo' };

  constructor() { }

  ngOnInit(): void { }

}

```

نلاحظ قمنا بتعريف كائن يحتوي على مجموعة بيانات (بطبيعة الحال في التطبيقات الواقعية هذه البيانات نأخذها من قاعدة بيانات عن طريق الاتصال بي service معينة) بحيث ان الشعار سوف يعتمد على هذه البيانات لعرض بياناته، وتتم هذه الطريقة بتعريف متغيرات خاصة بي ng-template باستخدام صيغة خاصة (let-Variable Name) ونقوم بربطها بالمتغيرات (الخصائص) الموجودة في الكائن logoInfo وبنفس الوقت نمرر هذه المتغيرات باستخدام طرق binding التي تعلمناها سابقاً سواء Interpolation او Property Binding إلى العناصر التي سوف تقوم بعرض الشعار، وآخر جزئية نقوم بها هي كما ربطنا Template Reference ذو الاسم logo بكل \*ngTemplateOutlet ايضاً هنا نقوم بربط كل \*ngTemplateOutlet بالكائن logoInfo عن طريق الخاصية context، كالتالي:

ملف app.component.html

```

<nav class="navbar navbar-dark bg-dark nav-fill">
  <ng-container *ngTemplateOutlet="logo; context: logoInfo"></ng-container>
  <a class="btn btn-info" href="#">Section One</a>
  <a class="btn btn-info" href="#">Section Two</a>
  <a class="btn btn-info" href="#">Section Three</a>
</nav>

<div class="card w-75 m-auto">
  <div class="card-body">
    <div class="card-header">
      <ng-container *ngTemplateOutlet="logo; context: logoInfo"></ng-container>
    </div>
    <h5 class="card-title"> </h5>
    <p class="card-text">
      Some quick example text to build on the card title and make up the bulk
      of the card's content.
    </p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>

<nav class="navbar fixed-bottom navbar-dark bg-dark d-flex flex-row-reverse">
  <a class="navbar-brand" href="#">
    <ng-container *ngTemplateOutlet="logo; context: logoInfo"></ng-container>
  </a>

```

```

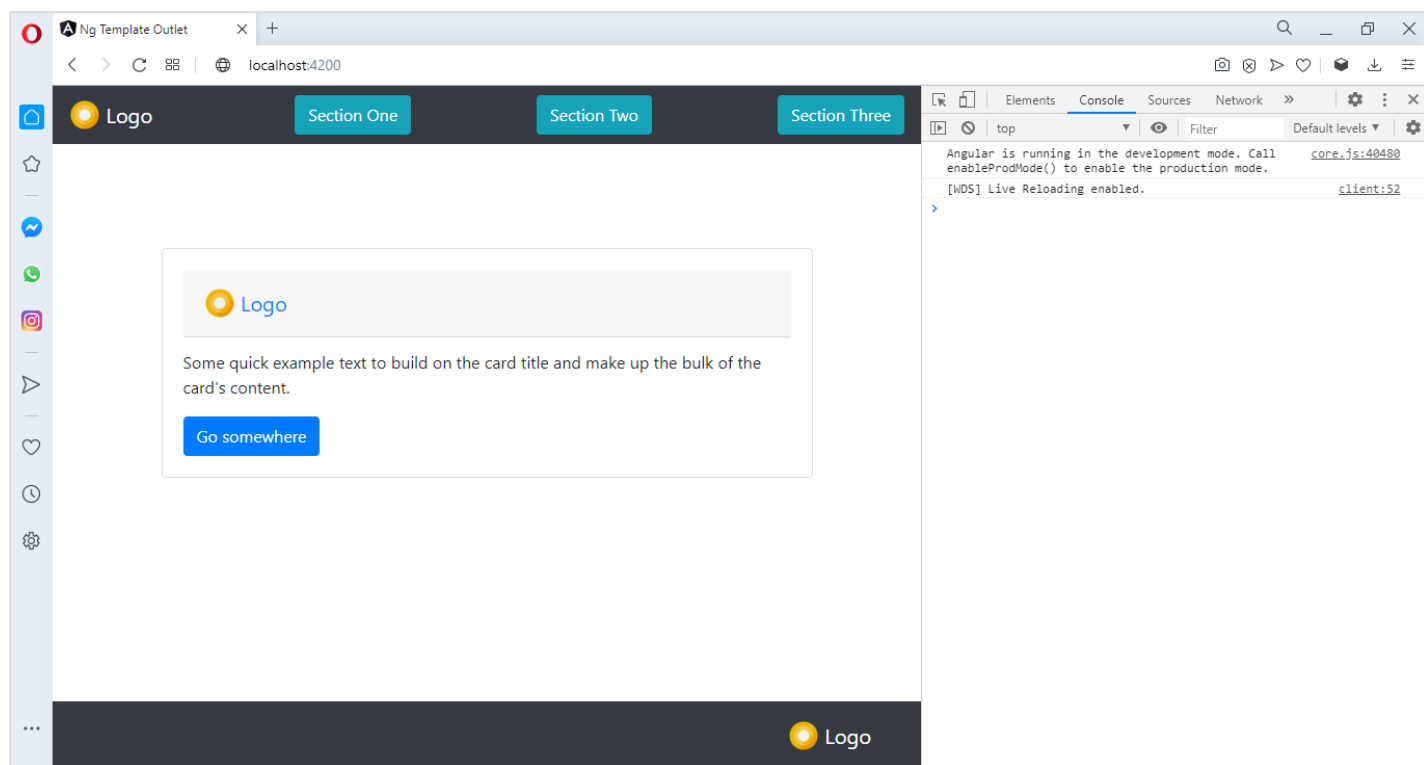
</nav>

<ng-template #logo let-logoUrl="url" let-logoName="name"> 4
  <a class="navbar-brand" href="#">
    <img
      [src]="[logoUrl]" 5
      width="30"
      height="30"
      class="d-inline-block align-top"
    />
    {{ logoName }} 6
  </a>
</ng-template>

```

في النقطة 4 قمنا بتعريف متغيرين هما logoUrl و logoName وقمنا بربطهما بالخاصيتين الموجودتين في الكائن logoInfo وهما url و name، أما النقطة 5 و6 قمنا بعمل bind لهذه المتغيرات، أما النقاط 1,2,3 فقمنا بربط الكائن logoInfo بكل \*ngTemplateOutlet عن طريق الخاصية context.

والنتيجة سوف تكون في المتصفح، نفس سابقها، ولكن هنا أصبحت البيانات الخاصة بشعار ديناميكية.



ولكن ماذا لو أردنا ان يكون هنالك قيمة افتراضية، بحيث إذا كان هنالك متغير قمنا بتعريفه ولكن لم نقوم بربطه بأي خاصية من خصائص الكائن فسوف تكون قيمته هذه القيمة الافتراضية، وللقيام بهذا الامر نستخدم \$implicit في الكائن logoInfo ونعطيه أي قيمة بحيث تكون هي القيمة الافتراضية لأي متغير لم نقوم بربطه بأي خاصية في هذا الكائن، كالتالي:

```

app.component.ts ملف
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',

```

```

templateUrl: './app.component.html',
styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  logoInfo = { url: 'assets/Logo.png', name: 'Logo', $implicit: 'Default Logo' };

  constructor() { }

  ngOnInit(): void { }

}

```

ولنفرض أن المتغير `logoName` لم نقم بربطه بأي خاصية، فانه بهذه الحالة سوف يأخذ القيمة الافتراضية في `$implicit`.  
كالتالي:

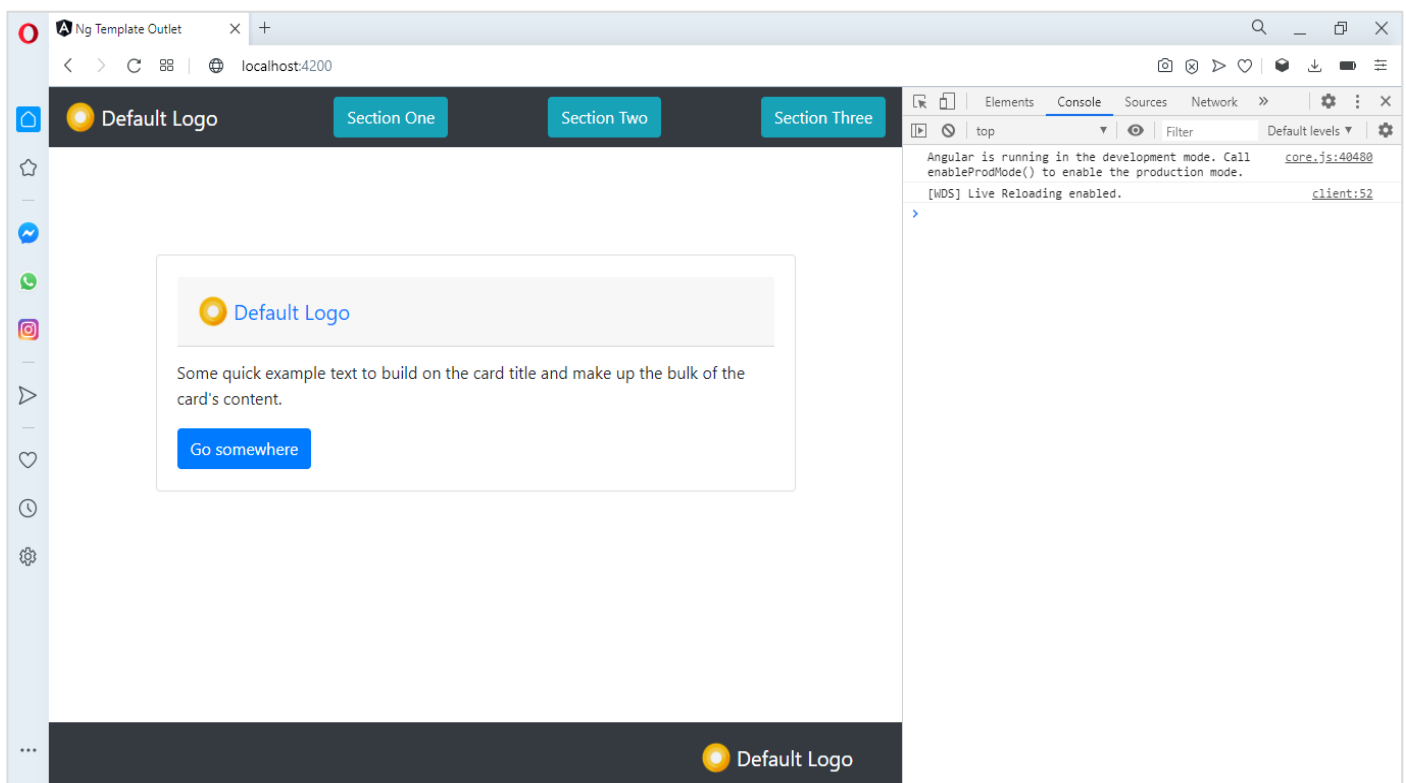
جزء من ملف `app.component.html`

```

<ng-template #logo let-logoUrl="url" let-logoName>
  <a class="navbar-brand" href="#">
    <img
      [src]="[logoUrl]"
      width="30"
      height="30"
      class="d-inline-block align-top"
    />
    {{ logoName }}
  </a>
</ng-template>

```

والنتيجة سوف تكون كالتالي:





## 7.4. Component Styles:

قد تطرقنا في السابق إلى `host`: و `ng-deep`:: وسوف نتطرق هنا بإذن الله بالإضافة إلى هذين الأمرين إلى الأمر `host-context`: وبعض النقاط الأخرى.

وقبل البدء بالغوص بتفاصيل تعامل Angular مع Styles نريد أن نوضح مفهوم مهم وهو Web Components.

### 1.7.4. Web Components:

عندما تحدثنا في مدخل إلى Angular Components لم نشير إلى نقطة مهمة وهي أن Angular لم يستفرد بهذا المفهوم وليس هو السباق لها وإنما في الحقيقة هو مبني على Web Components حيث تم إصدار في عام ٢٠١٥ م تقريباً مكتبة Polymer التي تساعد المطورين ببناء تطبيقاتهم بناءً على مفهوم Web Components، ومن هذا المنطلق قامت أغلب مكتبات واطر عمل JavaScript مثل React وVue وبالتأكيد Angular على فكرة تقسيم التطبيق إلى مجموعة من المكونات components الصغيرة وكل مكون يؤدي مهمة معينة ويتفاعل مع باقي المكونات وفق طرق معينة.

إذن من خلال ما سبق نستطيع أن نشير أن كل مكون ويب عبارة عن وحدة في واجهة المستخدم مسؤولة عن مهمة أو مهام محددة، وكل مكون يقوم بتغليف (Encapsulate) حالته الخاصة واكواد HTML (القالب – template) و CSS الخاصة به. وهذا هو ما تحاول مكونات الويب Web Components فعله بعيداً عن أي إطار أو مكتبة، وبذلك تكون فائدة Web Components بشكل عام، هي:

- التغليف والعزل: أي كل Component يحتوي على جميع ملفات HTML و CSS وأكواد JavaScript الخاصة به والتي لا يمكن الوصول إليها إلا عن طريق هذه الملفات مع إمكانية التواصل مع المكونات Components الأخرى وفق قواعد معينة وطرق مختلفة تختلف من مكتبة إلى مكتبة أخرى أو من إطار عمل إلى إطار عمل آخر كما قمنا بالتفصيل سابقاً في Angular Components Communication.
- إعادة الاستخدام: ومعناه أن هذه Components نستطيع إعادة استخدامها في أماكن متفرقة لنفس التطبيق أو نعمل Export لها ونستخدمها في تطبيقات أخرى.

و Web Components بشكل عام يعتمد على أربع ركائز أساسية، يمكن تلخيصها بالشكل التالي:



وهذه الركائز هي عامة ولا تختص بإطار عمل عن آخر وانما هي تختص بالذات بي Web Component، وقد تختلف اطر العمل بالاعتماد على هذه الركائز، فمنها من تبناها جميعاً ومنها من تبني بعضها واتاح للمطور حرية والمرونة في استخدامها جميعاً في تطبيقه او استخدام بعضها كAngular، وسوف أقوم بشرح موجز عن هذه الركائز، كالتالي:

- Custom Elements ومعناها ان لكل component اسم او selector خاص به، مشابه لأسماء العناصر (الوسوم) الخاصة بي HTML، واتوقع عزيزي المتعلم ان هذا الامر ليس بالشئ الجديد لديك لأننا تعاملنا معها سابقاً وبشكر متكرر في Angular Component.
- View او Template وهو المكان الذي نضيف فيه أكواد HTML الخاصة بالcomponent وايضاً هذا الجزء اشبعناها شرحاً في هذا الكتاب.
- Shadow DOM وهذه الركيزة هي التي قد يكون فيها غموض لديك، مع العلم انك تعاملت معها او بالأصح Angular قام بتنفيذها بالنيابة عنك بشكل ديناميكي، وفكرتها بكل بساطة ان شجرة DOM التي تحتوي جميع العناصر (الوسوم) سواء كانت custom او مبنية بالأساس في لغة HTML، تسمح بإن ننفذ أي style لأي عنصر من هذه العناصر بمعنى لو قمنا بعمل style لعنصر معين وليكن على سبيل المثال `<div></div>` فإن هذا التنسيق سوف يُطبق على جميع Divs في التطبيق ولو كانت في components آخر بشرط ان يكون هذا component قد تم بناءه، وهذا بالطبع غير مقبول، ومن هنا يأتي دور Shadow DOM حيث يقوم بتغليف كل component في شجرة DOM بحيث لا نستطيع الوصول إلى أي عنصر وإجراء أي عملية تنسيق Style عليه إلا من داخل component فقط الذي يحتويه، وهذه الركيزة معتمدة في Web Components ولكن في الأطر والمكتبات الأخرى تم استخدام طرق أخرى مثل React يتم استخدام VDOM اما Angular فلديه نظامه الخاص الذي يحاكي Shadow DOM مع إمكانية استخدام Shadow DOM في حال رغب المطور في ذلك.
- HTML imports والمقصود بها وجود آلية لاستدعاء component داخل component آخر او ملفات Styles الخاصة بكل component، مثال:

```
<link rel="import" href="component.html">
```

وهنا ليس المقام لشرح جميع جزئيات وتفصيل Web Components وانما يهمننا آلية تنفيذ Web Components داخل إطار عمل Angular، ولقد تكلمنا عن أغلب طرق وآليات تعامل Angular مع Components، فمثلاً الركيزة الأولى سوف نلاحظ ان لكل Component يوجد Selector خاص به وهو ما يمثل Custom Elements، اما الركيزة الثانية وهي Template فهو ملف HTML الخاص بكل Component الذي يحتوي على جميع Markup الخاص بعرض البيانات للمستخدم، والركيزة الرابعة فهي نلاحظ اننا نستطيع وضع Selector الخاص بي Component معين داخل Component آخر وكأننا نقوم باستدعاء هذا Component داخل Component الآخر وبنفس الوقت لو نلاحظ ان لكل Component ملف Style وملف Template اللذان نستدعيهما في ملف Class الخاص بهذا Component، بقي لدينا الركيزة الثالثة وهي Shadow DOM وهذا الامر يقوم به Angular بشكل تلقائي بالنيابة عنا ولكن كما قلنا ان Angular لديه نظامه الخاص لطرق الوصول إلى العناصر وإجراء التنسيق Styles عليها، وهو ما سوف نطرق إليه في النقطة التالية.

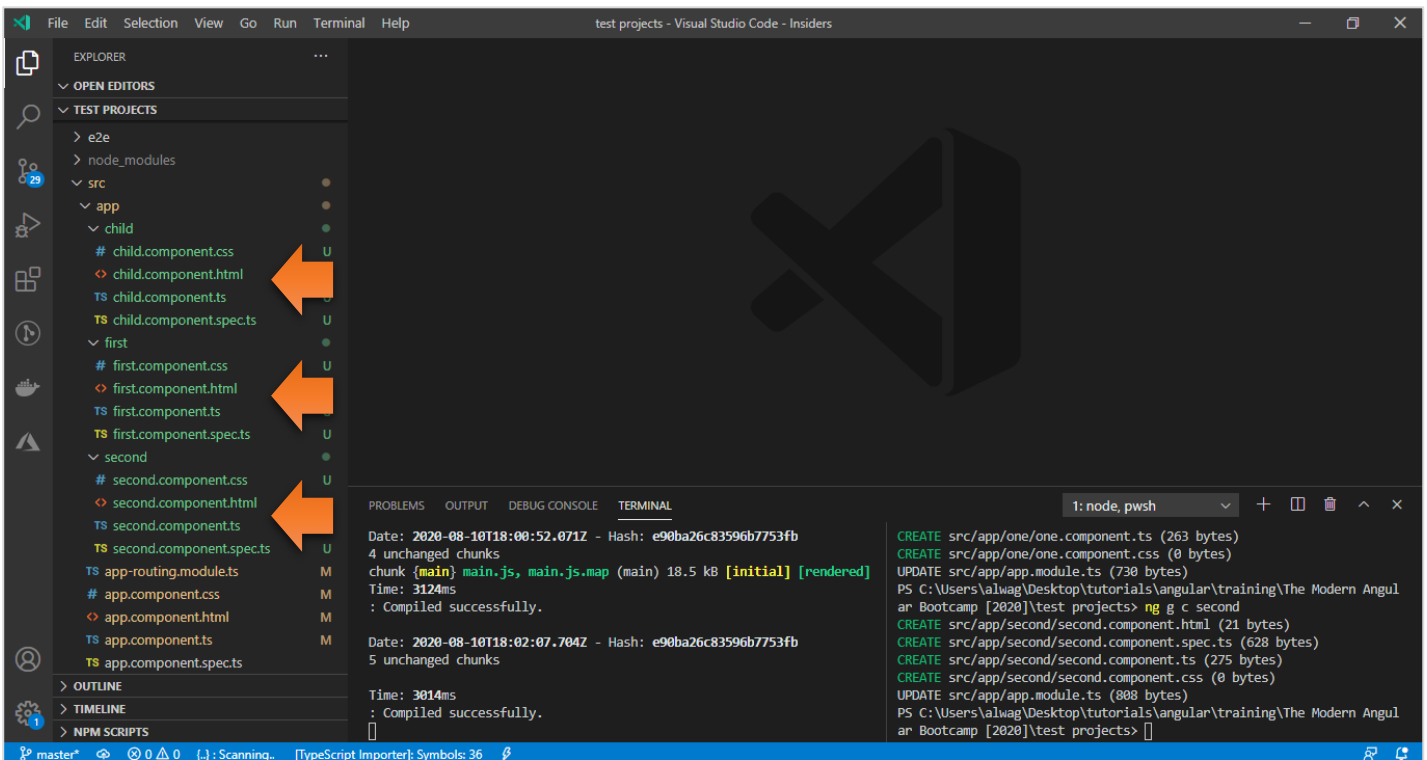
## :View Encapsulation .2.7.4

يقدم Angular ثلاث طرق لتعامل مع Styling لعناصر DOM وطرق تغليفها، وهي Emulated, ShadowDom, None، مع العلم أن الطريقة الافتراضية هي Emulated. وسوف نتطرق إلى جميع هذه الأنواع بإذن الله.

### :None .1.2.7.4

هذا النوع يقوم بإلغاء Shadow DOM نهائياً ويتيح التعامل بشكل مباشر مع شجرة DOM، وكما أشرنا سابقاً أن في حال عدم وجود Shadow DOM فإن أي تنسيق نقوم به على عنصر معين فإنه يطبق على جميع العناصر المشابهة في التطبيق في أي component متى ما تم بناء هذا component في الذاكرة.

ولتوضيح لنقوم بإنشاء ثلاثة components واحد باسم first والثاني باسم second والثالث باسم child، كالتالي:



الآن لنقم بإضافة عنصر `<h1></h1>` لجميع components، كالتالي:

ملف app.component.html

```
<h1>Root Component</h1>
```

ملف first.component.html

```
<h1>First Component</h1>
```

ملف second.component.html

```
<h1>Second Component</h1>
```

ملف child.component.html

```
<h1>Child Component</h1>
```

الآن لنقم بإضافة selector الخاص بي child إلى first وبنفس الوقت نضيف selector الخاص بكلاً من first و second إلى Root Component، كالتالي:

ملف first.component.html

```
<h1>First Component</h1>
```

```
<app-child></app-child>
```

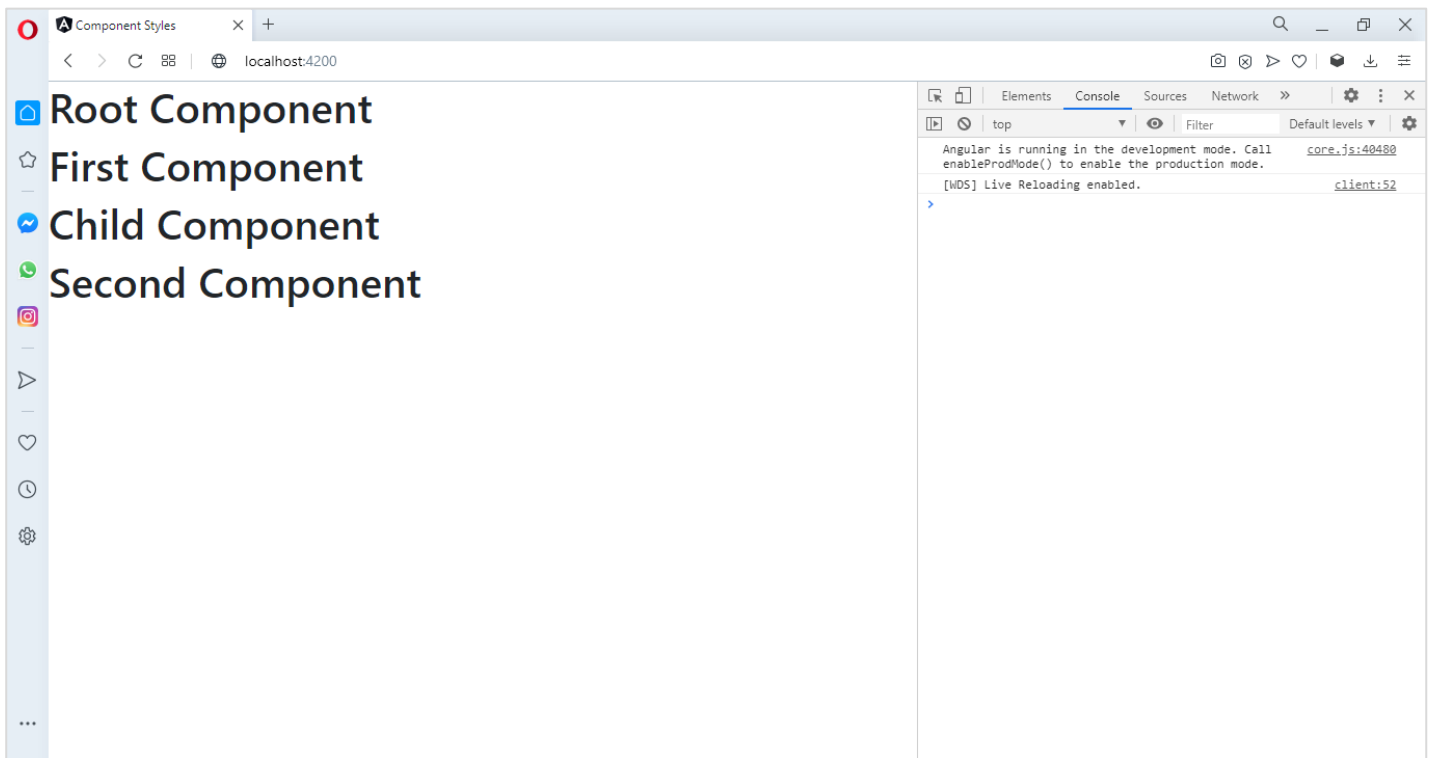
ملف app.component.html

```
<h1>Root Component</h1>
```

```
<app-first></app-first>
```

```
<app-second></app-second>
```

والنتيجة في المتصفح، كالتالي:



إلى الآن لم نقم بعمل أي شيء يخص View Encapsulation، ولنقم بتطبيق اول نوع وهو None، ولنقم بإضافتها لأي component، وليكن child، وبنفس الوقت لنقوم بعمل تنسيق Style معين للعنصر <h1></h1> في ملف style الخاص بهذا component، كالتالي:

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  encapsulation: ViewEncapsulation.None
})

export class ChildComponent implements OnInit {

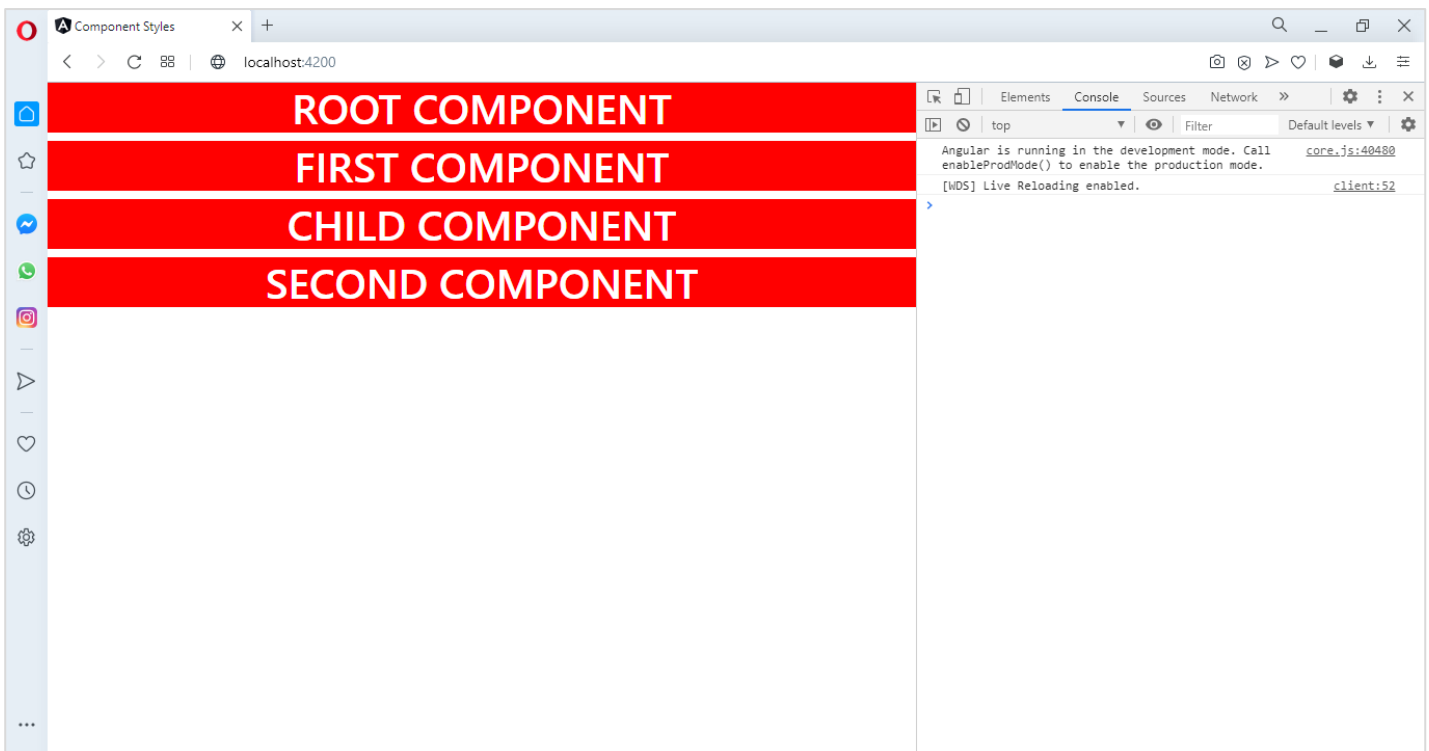
  constructor() { }

  ngOnInit(): void { }
}
```

ملف child.component.css

```
h1 {
  background: red;
  color: white;
  text-transform: uppercase;
  text-align: center;
}
```

والنتيجة في المتصفح، هي:



نلاحظ انه باستخدامنا None فإن التنسيق تم تطبيقه على جميع عناصر h1 الموجودة في التطبيق وهو المتوقع في حالتنا هذه لأننا أصبحنا نتعامل مع شجرة عناصر DOM بشكل مباشر، مع العلم انه أولاً لم نضيف التنسيق إلا في ملف style الخاص

بي child وثانياً لم نضيف الأمر None إلا في ملف class لي child component ومع هذا تم التطبيق على جميع components الأخرى.

ونعيد ما ذكرناه سابقاً بأن Angular في هذه الحالة لا يُنفذ ويطبق التنسيقات إلى للعناصر التي تم بناءها في الذاكرة وهو الذي قمنا به في المثال السابق، ولتوضيح لنقم بعمل زر إذا تم الضغط عليه يقوم ببناء second component بحيث عن بداية تشغيل التطبيق يتم بناء كلاً من root و first و child، وسوف نستخدم Router للقيام بهذا الأمر، كالتالي:

ملف app-routing.component.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { SecondComponent } from './second/second.component';

const routes: Routes = [
  {path: 'second', component: SecondComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

وفي Root Component نضيف الزر ونعمل navigate إلى path ذو الاسم second والذي يقوم ببناء وعرض second Component، وبفس الوقت لابد ان نستبدل <app-second></app-second> بي <router-outlet></router-outlet> كالتالي:

ملف app.component.html

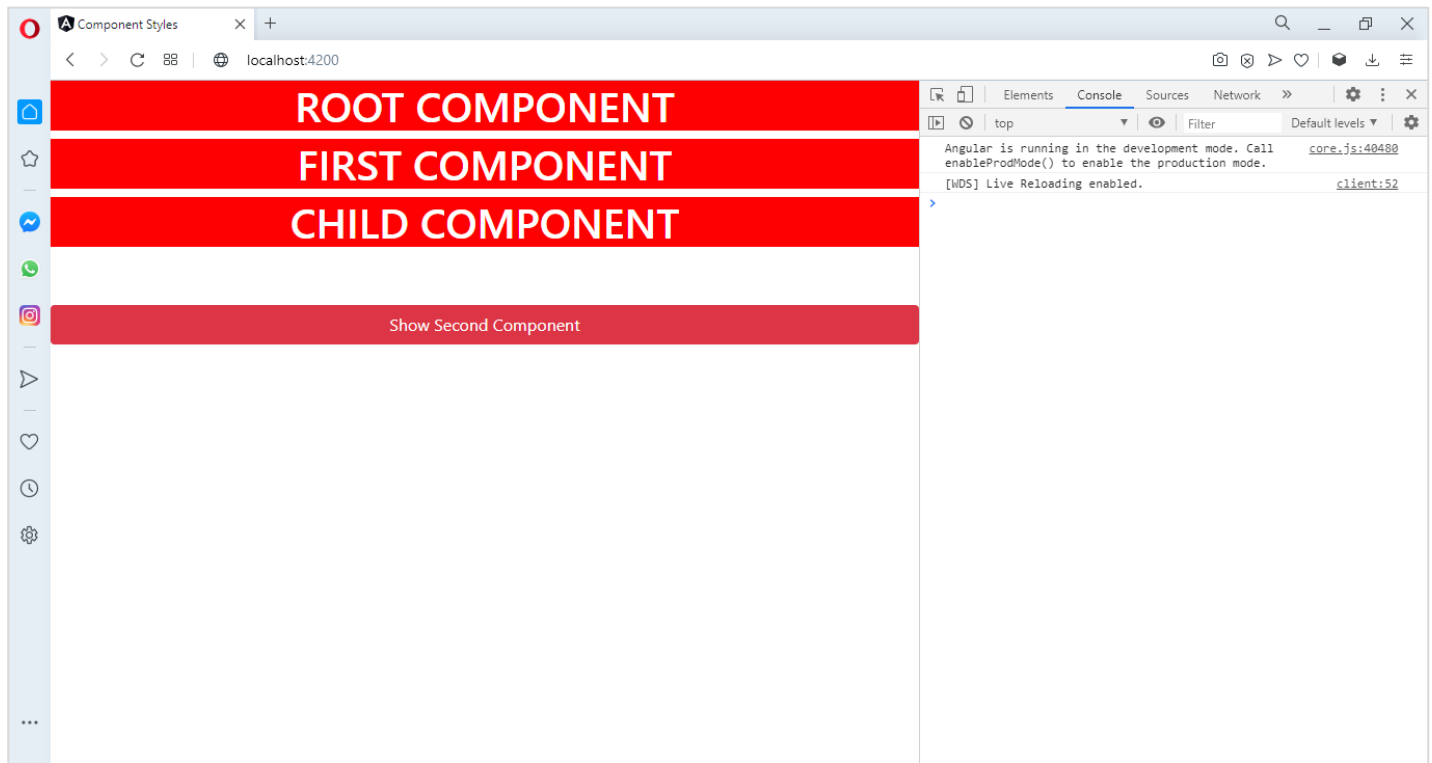
```
<h1>Root Component</h1>

<app-first></app-first>

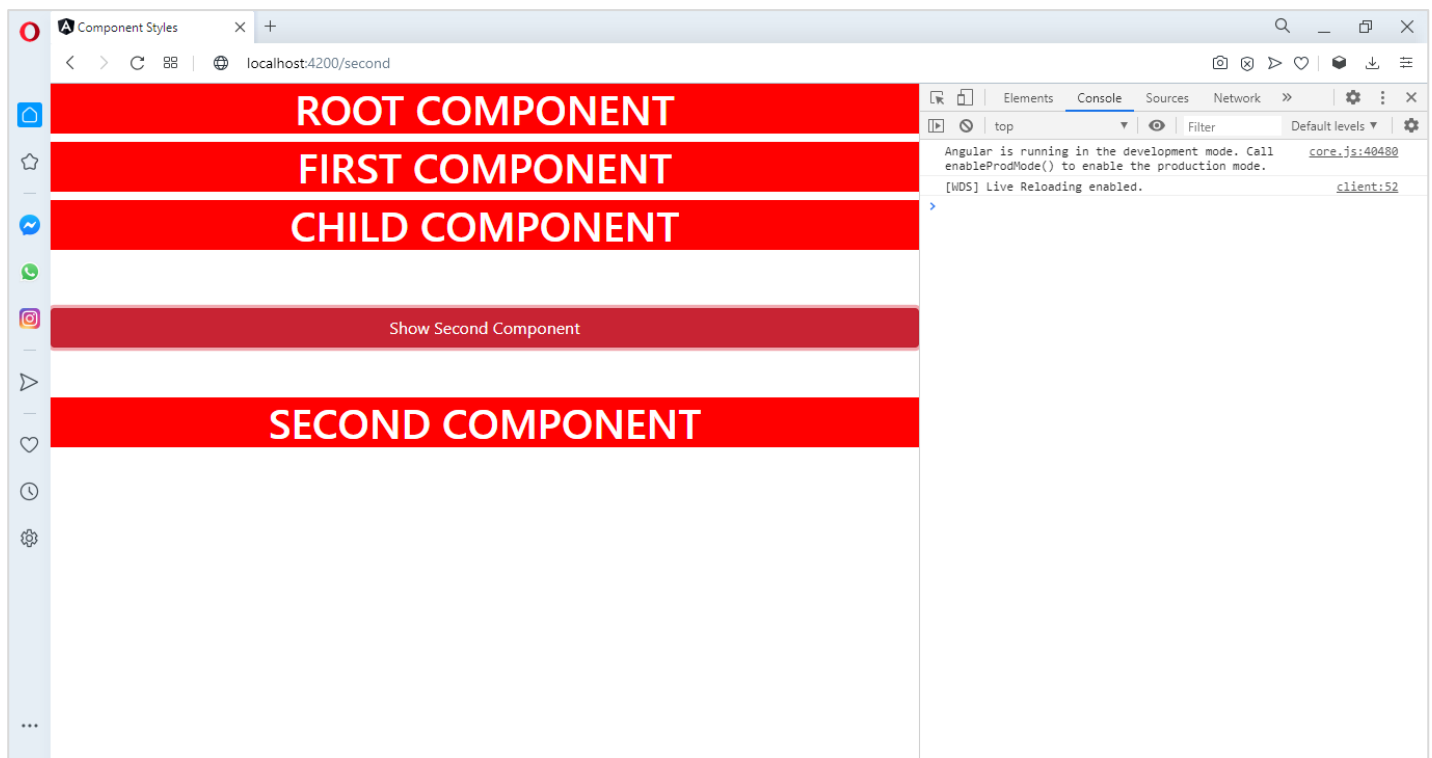
<a class="btn btn-danger btn-large w-100 mt-5 mb-5" routerLink="./second">
  Show Second Component
</a>

<router-outlet></router-outlet>
```

ولنشاهد النتيجة في المتصفح:



نلاحظ تم بناء جميع components ما عدا second، وبما انه لم يتم بناءه فمن البديهي وطبيعة الحال لن يضيفه وعناصره إلى شجرة DOM وبالتالي لن يُطبق التنسيق عليه، ولنقم الآن بالضغط على الزر ونرى النتيجة:



نلاحظ عند الضغط على الزر قام ببناء component ووجد بداخله العنصر h1 وبعد إضافة هذا العنصر إلى شجرة DOM تلقائياً قام بتطبيق التنسيق عليه.

وهذا هي الطريقة القديمة في بناء المواقع الإلكترونية وتطبيقات الويب حيث كنا نتعامل مع شجرة DOM بشكل مباشر قبل ظهور Web Component و Shadow DOM.

#### :ShadowDom .2.2.7.4

عند اختيارنا لهذا الأمر فإننا سوف نتعامل مع Shadow DOM الأساسي أو Native الموجود في Web Component، أما العمل والسلوك الذي يُحدثه عن تطبيقه في Angular هو تطبيق التنسيق على component وجميع الأبناء Selectors التي يحتويها، وهو بذلك يقوم بعمل تغليف أو Encapsulation للعنصر المحدد أو لنقل بصيغة أدق لي component المحدد وما يتفرع منه من components أبناء ان وجدت، بحيث ان تطبيق أي تنسيق على أي عنصر والعناصر المشابهة له لا يتعدى هذا component والأبناء فقط.

ولتوضيح لنقم بنقل أكواد التنسيق التي اضعناها في ملف child.component.css إلى ملف style في first وبنفس الوقت نحذف الامر None من ملف class في child component وأخيراً نضيف الامر ShadowDom إلى ملف class في first، كالتالي:

ملف child.component.css

ملف style الخاص بي component ذو الاسم child فارغ بعدما نقلنا الكود الخاص بالتنسيق منه.

ملف first.component.css

```
h1 {
  background: red;
  color: white;
  text-transform: uppercase;
  text-align: center;
}
```

ملف style في component ذو الاسم first بعدما نقلنا الكود الخاص بالتنسيق إليه.

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})

export class ChildComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}
```

ملف class في component ذو الاسم child بعدما قمنا بحذف الامر None منه.



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

@Component({
  selector: 'app-first',
  templateUrl: './first.component.html',
  styleUrls: ['./first.component.css'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class FirstComponent implements OnInit {

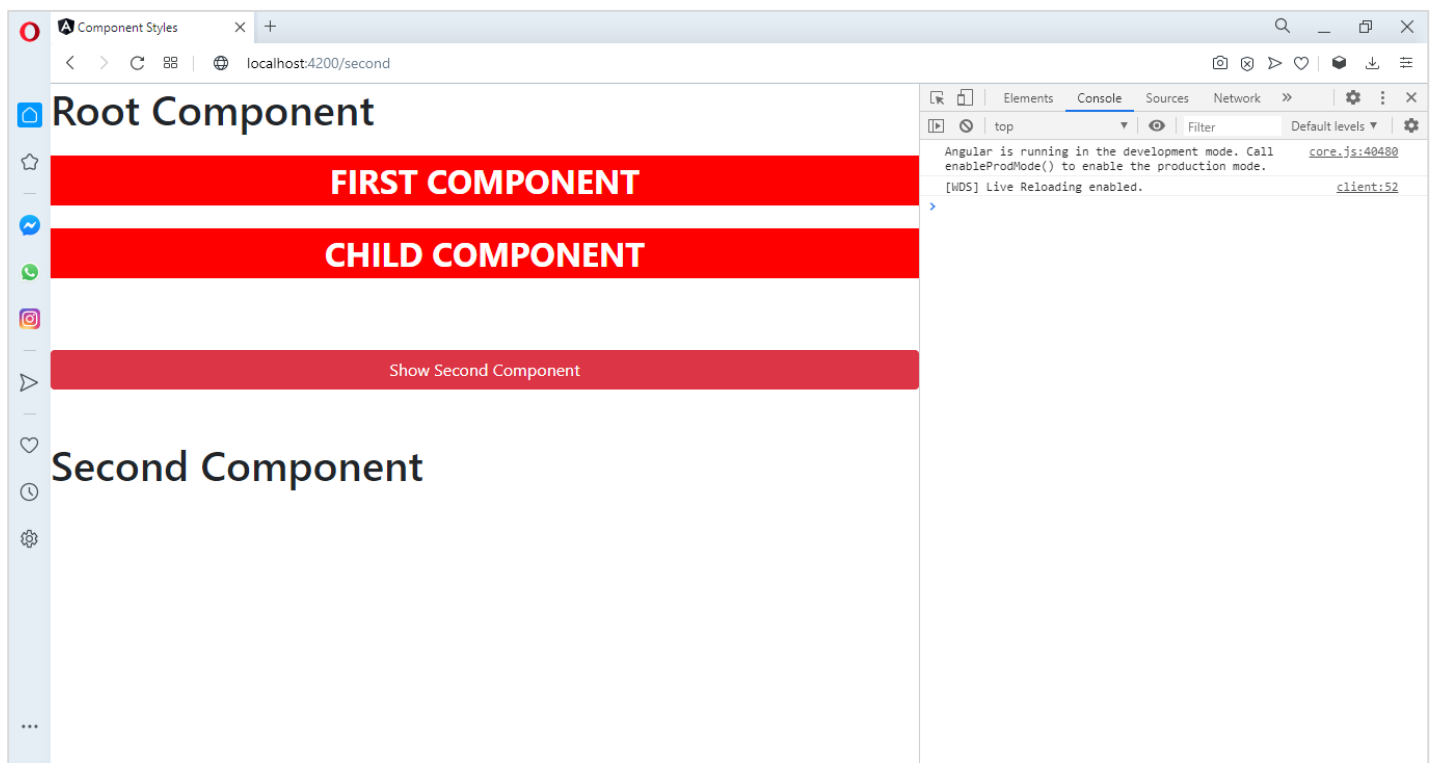
  constructor() { }

  ngOnInit(): void {
  }

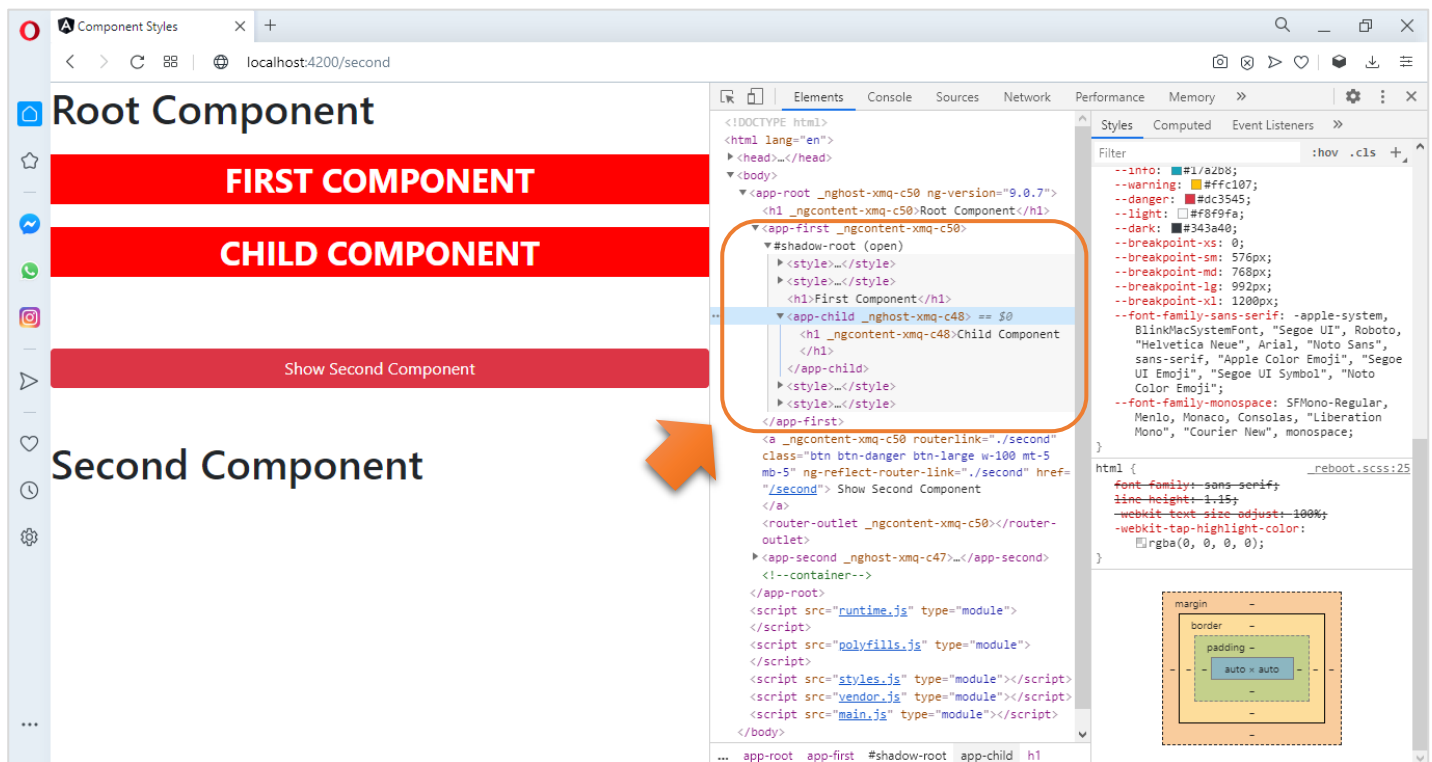
}
```

ملف class لي component ذو الاسم first بعدما اضعنا الامر ShadowDom إليه.

والآن لنذهب إلى المتصفح ونرى النتيجة، كالتالي:



نلاحظ انه قام بتغليف وتطبيق التنسيق على عناصر h1 الموجودة في first و child فقط، ولو عملنا inspect لهذه العناصر لنرى سلوكها في DOM لوجدنا انه أضاف الامر (open) #shadow-root بمعنى ان هذه العناصر تم عمل تغليف لها، كالتالي:



ولكن يُعيب Shadow DOM في Angular هو انه يقوم بتغليف component الأب وجميع الأبناء وهذا غير محبب فنريد ان يكون التغليف لكل component على حدا، وهذا يقودنا إلى آخر نوع وهو Emulated.

#### :Emulated. 3.2.7.4

وهو النوع الافتراضي في Angular وهو يُحاكي Shadow DOM، وقام بحل مشكلتين الأولى عمل تغليف لكل component على حدا والثانية عدم القلق من كون المتصفحات التي قد يعمل بها تطبيقك تدعم Shadow DOM من عدمه.

ولتوضيح لنقوم بتغيير الامر ShadowDom في class لي component ذو الاسم first إلى الامر Emulated، كالتالي:

```

first.component.ts ملف
import { Component, OnInit, ViewEncapsulation } from '@angular/core';

@Component({
  selector: 'app-first',
  templateUrl: './first.component.html',
  styleUrls: ['./first.component.css'],
  encapsulation: ViewEncapsulation.Emulated
})
export class FirstComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}

```

ويجب الإشارة انه سواء قمنا بكتابة الامر Emulated او لم نقم بكتابة شيء فالأمر سيان لان هذا الامر هو الافتراضي، كالتالي:

```

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

```

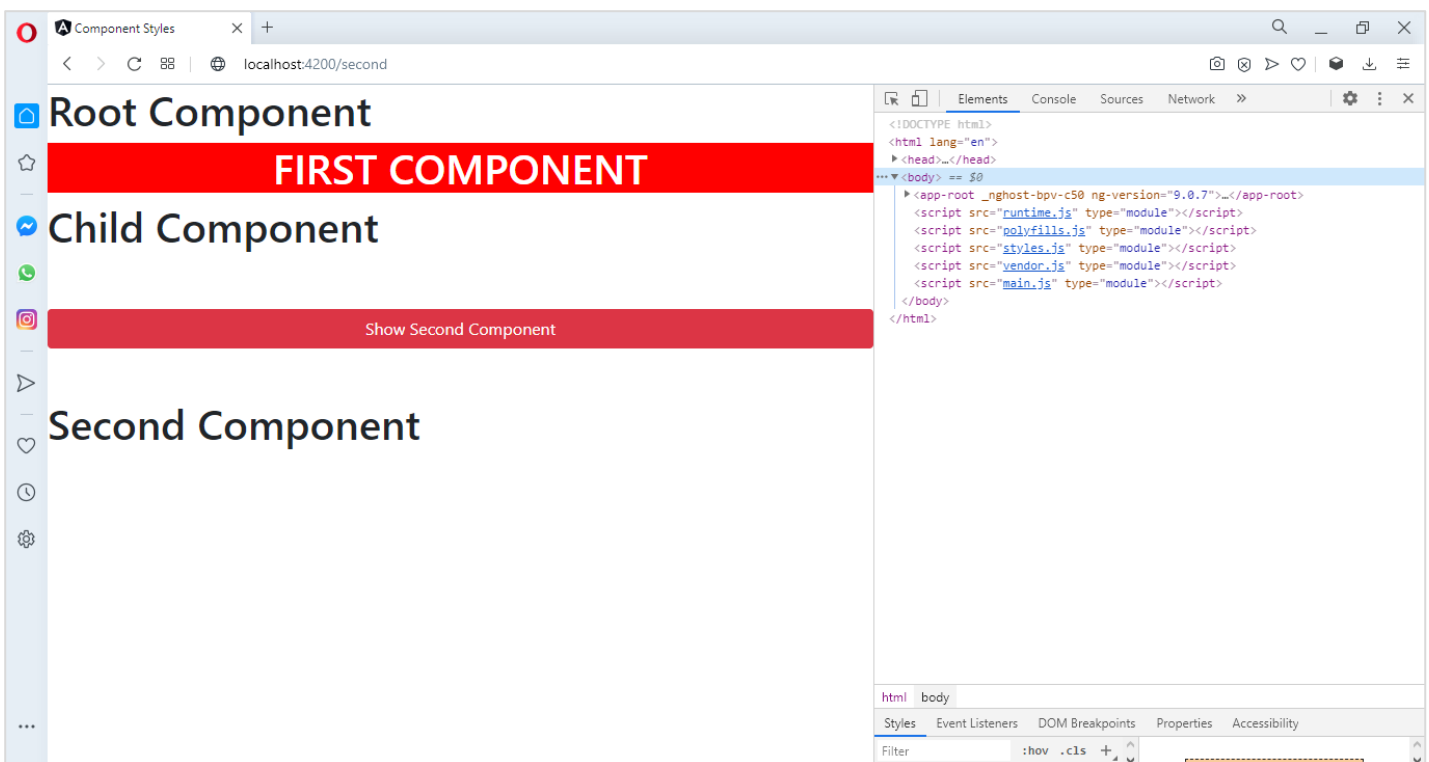
```
@Component({
  selector: 'app-first',
  templateUrl: './first.component.html',
  styleUrls: ['./first.component.css']
})
export class FirstComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

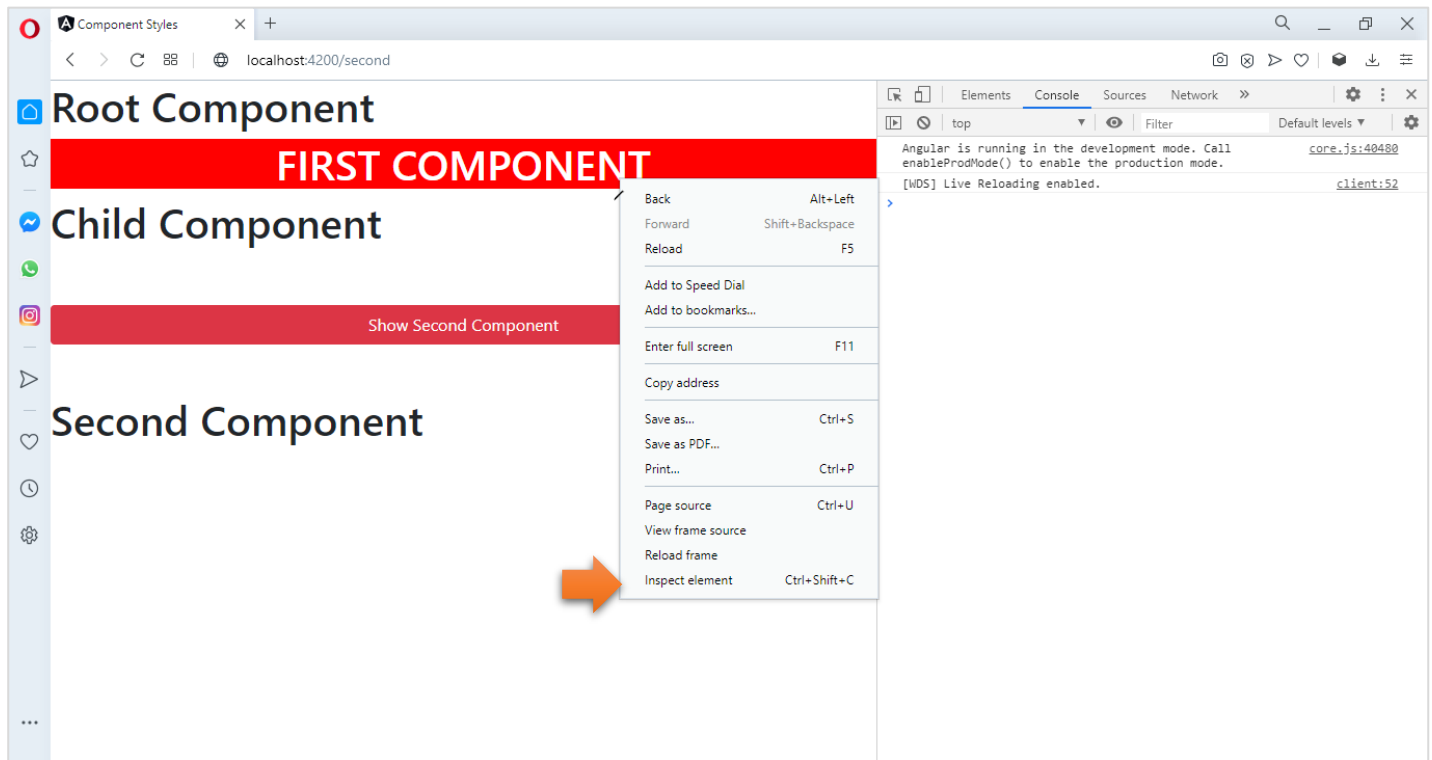
}
```

اما النتيجة في المتصفح فتكون كالتالي:

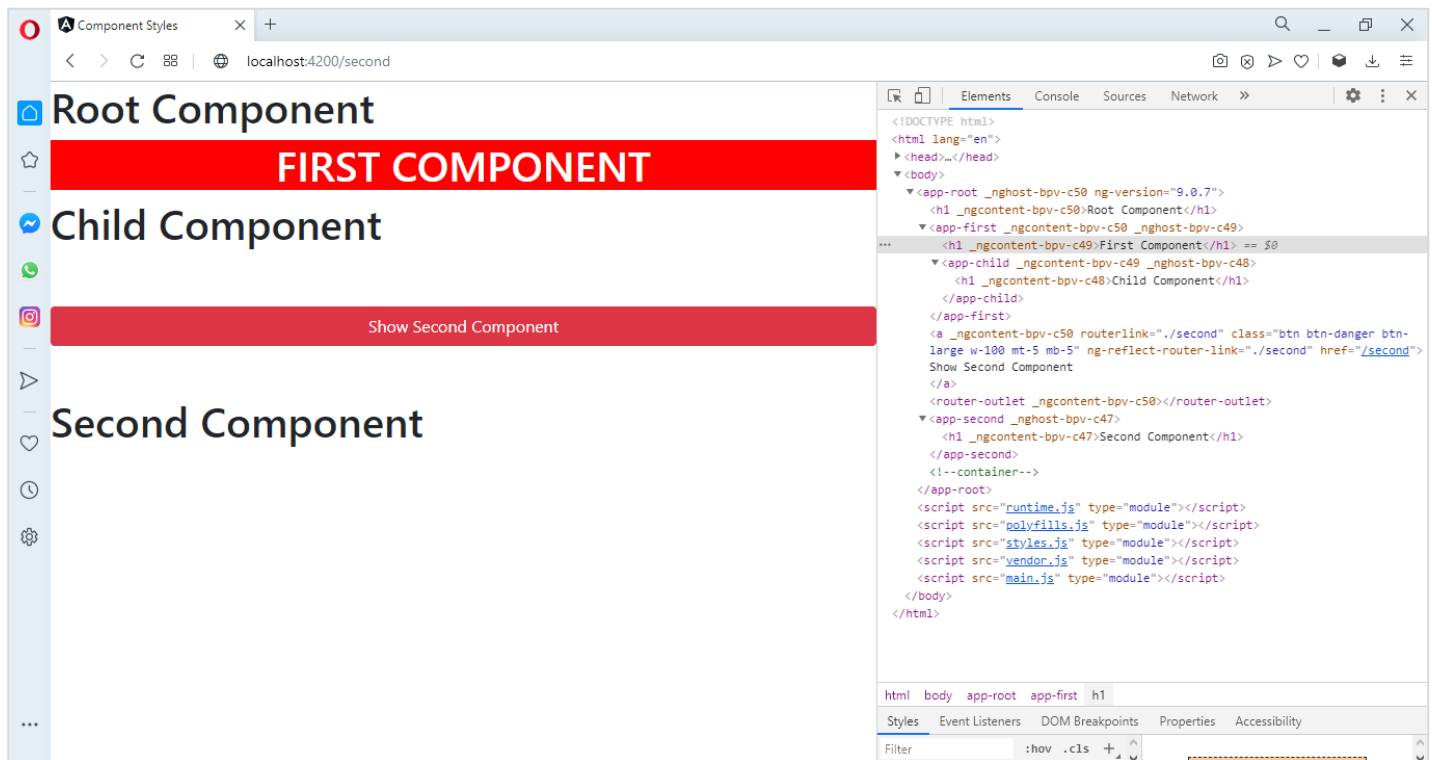


نلاحظ انه قام بعمل تغليف لعناصر component ذو الاسم first بحيث ان أي تنسيق يكون في حدود هذا component فقط ولا يتعداها إلى العناصر الأخرى، ونفس الآلية لبقية components في التطبيق.

والسؤال الذي يطرح نفسه هو كيف قام Angular بعمل هذا التغليف وفي الحقيقة الإجابة بسيطة، حيث عند بداية أي بناء لأي component يقوم Angular بوضع خاصية Attribute لهذا component في DOM وبنفس الوقت يقوم بوضع نفس هذه الخاصية لجميع العناصر التي يحتويها هذا component، ولتوضيح لنعمل على سبيل المثال inspect لي component ذو الاسم first، وذلك عن طريق الضغط بزر الفأرة الأيمن على العنصر ومن ثم اختيار inspect من القائمة المنسدلة، كالتالي:



وسوف تظهر لنا النتيجة التالية:



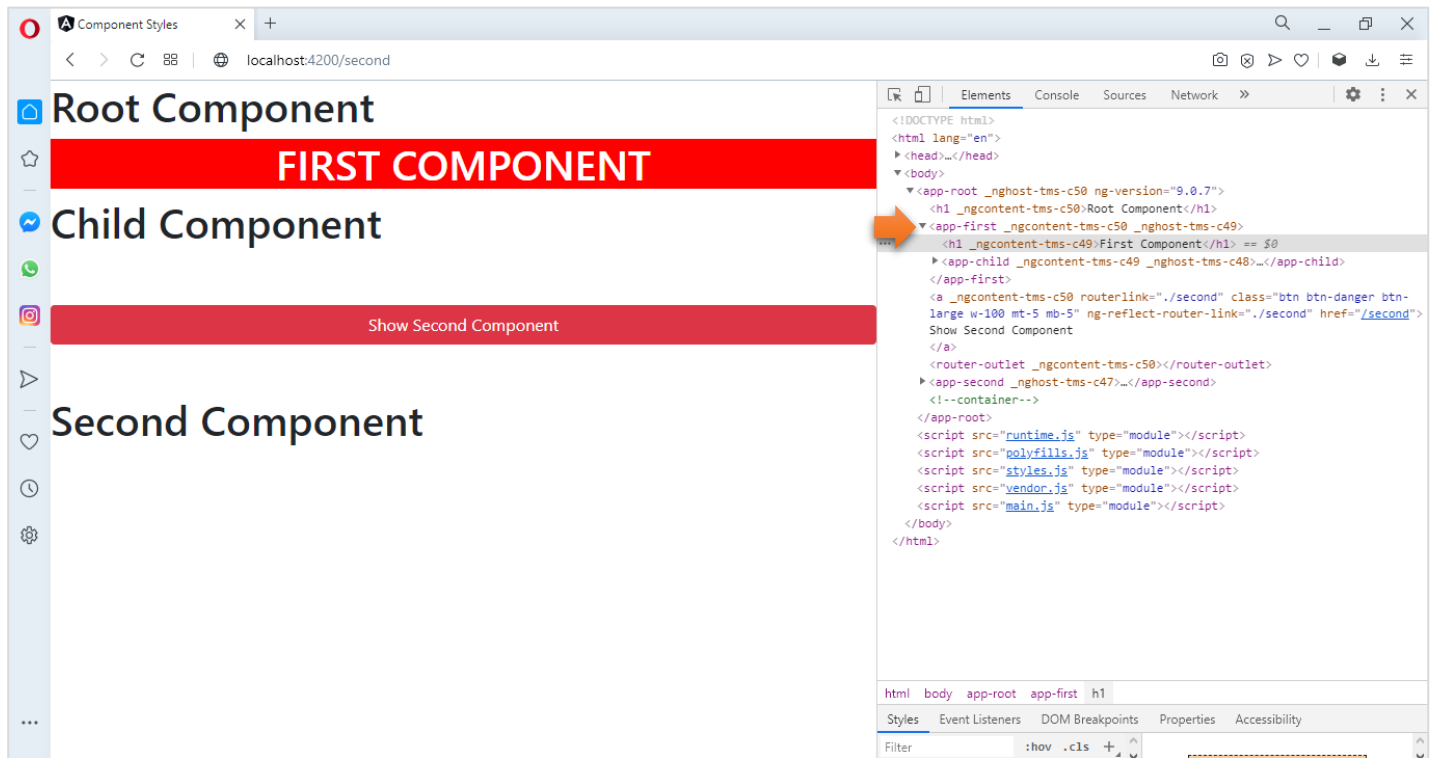
ولنقم بتكبيرها لكي نشرحها بشكل أفضل، كالتالي:

```

<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <app-root _ngghost-bpv-c50 ng-version="9.0.7">
      <h1 _ngcontent-bpv-c50>Root Component</h1>
      <app-first _ngcontent-bpv-c50 _ngghost-bpv-c49>
        <h1 _ngcontent-bpv-c49>First Component</h1> == $0
        <app-child _ngcontent-bpv-c49 _ngghost-bpv-c48>
          <h1 _ngcontent-bpv-c48>Child Component</h1>
        </app-child>
      </app-first>
      <a _ngcontent-bpv-c50 routerlink="./second" class="btn btn-danger btn-large w-100 mt-5 mb-5" ng-reflect-router-link="./second" href="/second">
        Show Second Component
      </a>
      <router-outlet _ngcontent-bpv-c50></router-outlet>
    </app-root>
    <app-second _ngghost-bpv-c47>
      <h1 _ngcontent-bpv-c47>Second Component</h1>
    </app-second>
    <!--container-->
  </body>
</html>

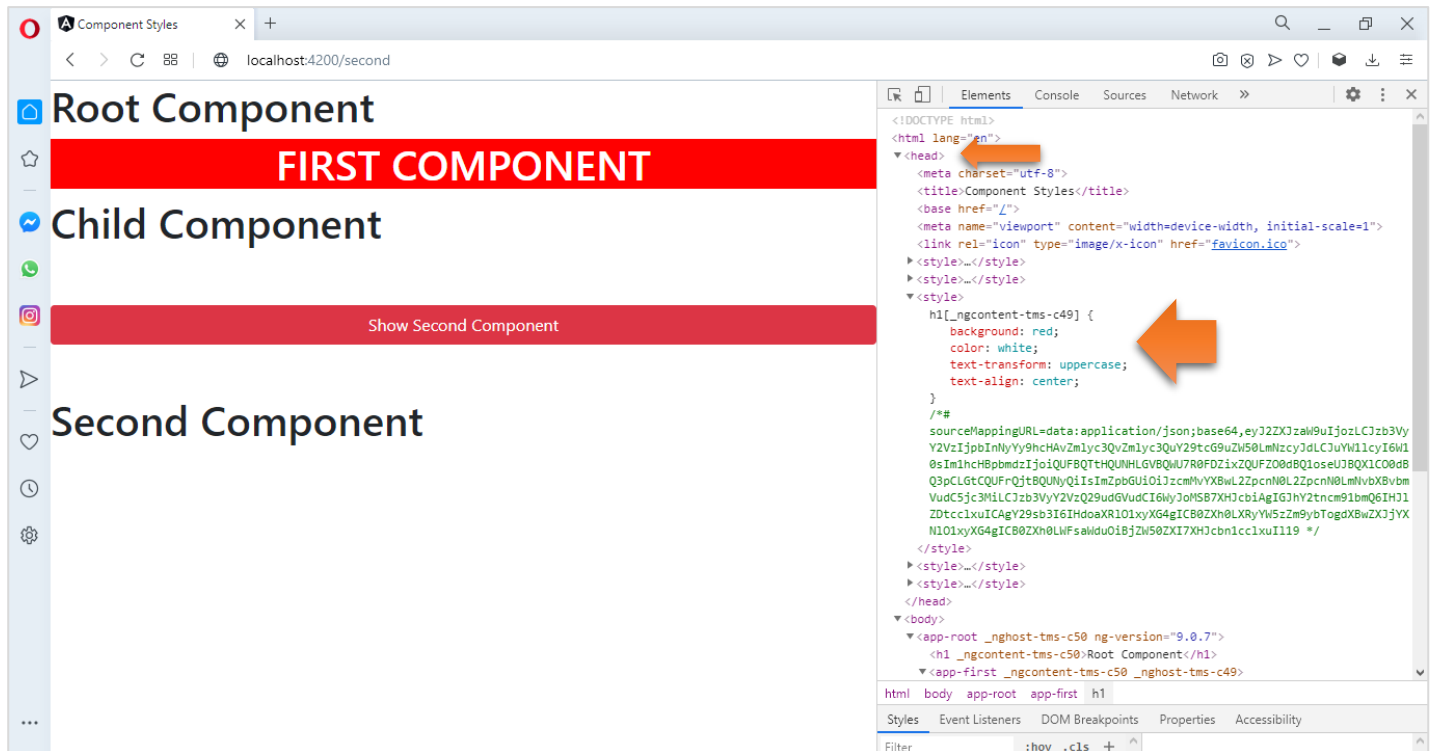
```

نلاحظ وجود الخاصيتين `_ngghost-bpv-c49` و `_ngcontent-bpv-c50` حيث ان الأولى تشير إلى ان هذا component والذي هو في DOM بحقيقة الأمر عبارة عن عنصر او وسم ولكنه custom element ونشير اليه هنا بالوسم `app-first` بانه تابع او ابن لي component الذي يحمل الخاصية `c50` ولو نظرنا إلى أي component هذا لوجدنا انه هو Root Component، اما الخاصية الثانية فتشير إلى ان هذا العنصر `app-first` هو عبارة عن host او مُضيف يحتوي هو الآخر على عناصر بداخله، وفي مثالنا هنا لا يحتوي إلا على عنصر واحد وهو `h1` حيث يوجد به هو الآخر خاصية هي `_ngcontent-bpv-c49` والتي تشير إلى ان هذا العنصر ينتمي إلى component الذي رقمه `c49` والذي هو `app-first` وهكذا بقية العناصر وcomponents الأخرى، حيث يتم تغليفها عن طريق إضافة خاصية مشتركة تجمع كل component وجميع العناصر التي يستضيفها او يحتويها، مع العلم ان هذه الخاصية غير ثابتة بحيث مع كل بناء لأي component يتم اعطائه خاصية جديدة، ولنقم بعمل تحديث لصفحة ولنرى النتيجة:



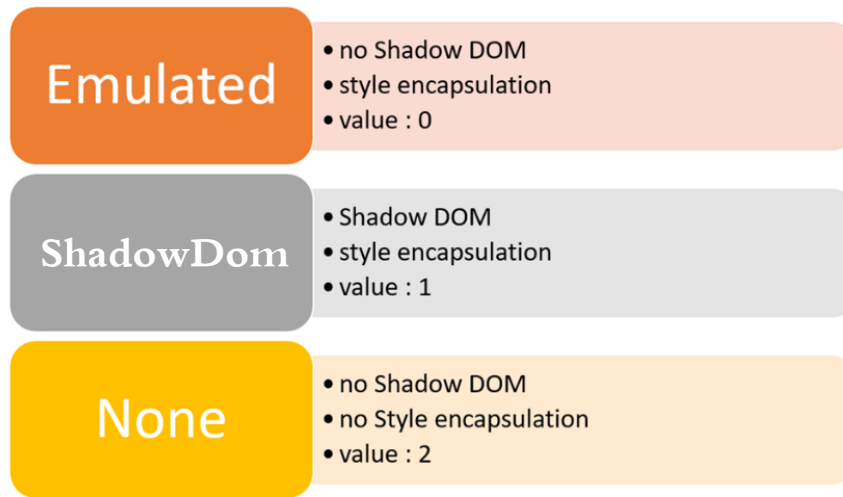
نلاحظ انه تم تغيير الخاصيتين إلى `_ngcontent-tms-c50` و `_ngghost-tms-c49`.

وبما ان Angular اوجد طريقة لعمل تغليف لكل component على حدا عن طريق إضافة خصائص موحدة لكل واحد منهما فذلك يستطيع ايضاً عن طريق هذه الخاصية تحديد أي عنصر سوف تجري عليه التنسيق، ولتوضيح لنذهب إلى head ونختار style، كالتالي:



نلاحظ العنصر `h1` تم إضافة الخاصية له `h1[_ngcontent-tms-c49]` كأننا نقول قم بتطبيق هذا التنسيق لكل `h1` موجود في component الذي يحمل الرقم `c49`.

ويمكن تلخيص جميع هذه الأنواع بالشكل التالي:



وبذلك أصبح لدينا البنية المعرفية اللازمة لفهم كيف يقوم Angular بتغليف Components والتعامل مع التنسيقات المختلفة، وسوف نتطرق في الجزء القادم إلى طرق إضافة Styles إلى component.

#### 3.7.4. طرق إضافة Styles إلى Component:

هنالك ثلاث طرق لإضافة styles إلى أي component الأولى وهي الافتراضية عن طريق كتابة أكواد التنسيق في ملف او مجموعة ملفات منفصلة ومن ثم استدعائها في ملف class لي component، ولنأخذ Root Component كمثال، كالتالي:

```
ملف app.component.ts
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}
```

نلاحظ مع كل component يتم انشاء ملف style منفصل بنفس اسم component ويتم استدعائه في ملف class عن طريق الخاصية styleUrls بين اقواس المصفوفة وهذا معناه اننا نستطيع إضافة اكثر من ملف style نستطيع اضافتهم إلى نفس component.

اما الطريقة الثانية وهي inline styling، وهي كتابة أكواد التنسيق مباشرة في ملف class وذلك عن طريق إضافة الخاصية styles ومن ثم كتابة أكواد التنسيق بين علامتي backtick، كالتالي:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  styles: [
    h1 {
      background: red;
      color: white;
      text-transform: uppercase;
      text-align: center;
    }
  ]
})
export class AppComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}
```



ونلاحظ ايضاً اننا قمنا بكتابتها بين اقواس المصفوفة بمعنى اننا نستطيع كتابة اكثر من تنسيق، وهذا النوع نقوم بإضافته في حال كان التنسيق بسيط ولا يحتوي على اسطر كثيرة.

والنوع الأخير هو بكتابة التنسيق في ملف template الخاص بالcomponent مباشرة، كالتالي:

```
<style>
  h1 {
    background: blue;
    color: white;
    text-transform: uppercase;
    text-align: center;
  }
</style>

<h1>Root Component</h1>

<app-first></app-first>

<a class="btn btn-danger btn-large w-100 mt-5 mb-5" routerLink="./second">
  Show Second Component
</a>

<router-outlet></router-outlet>
```

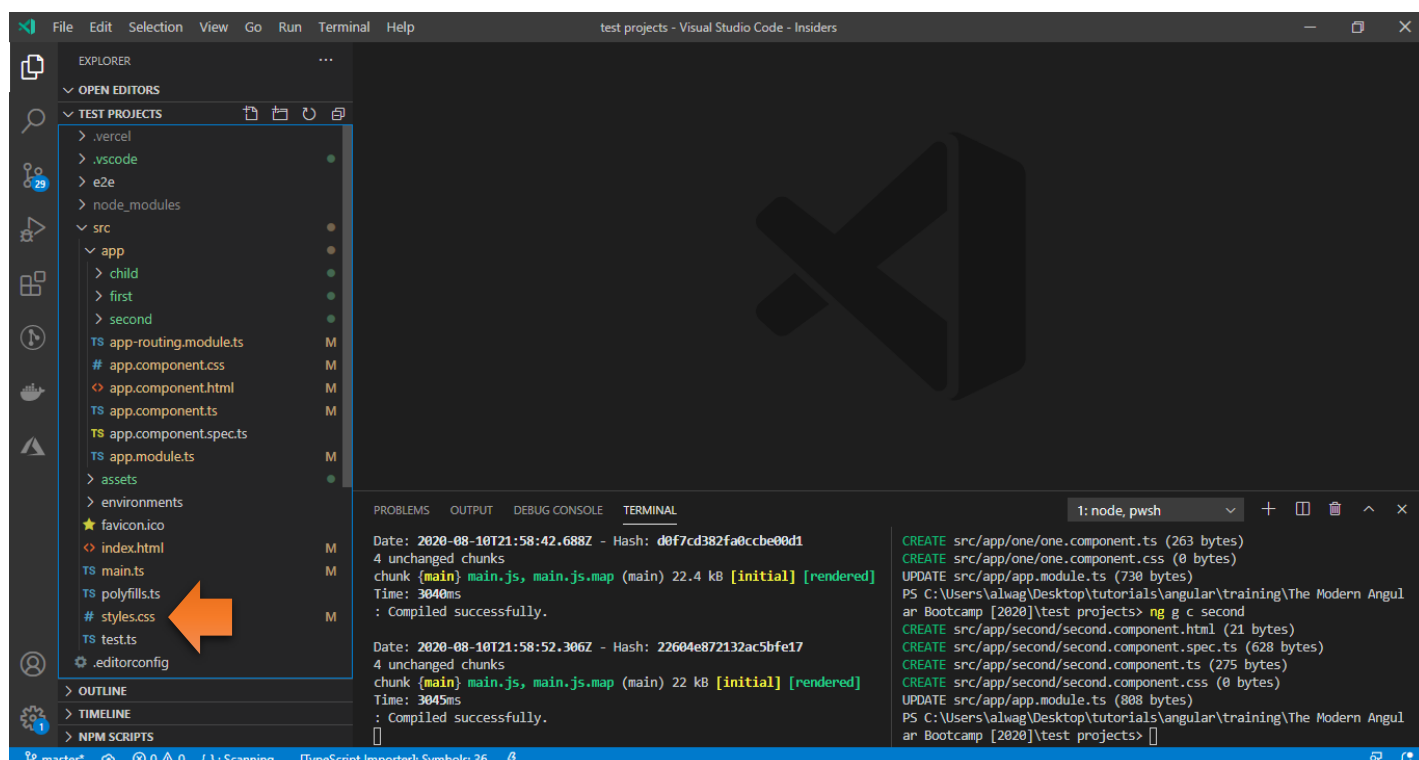




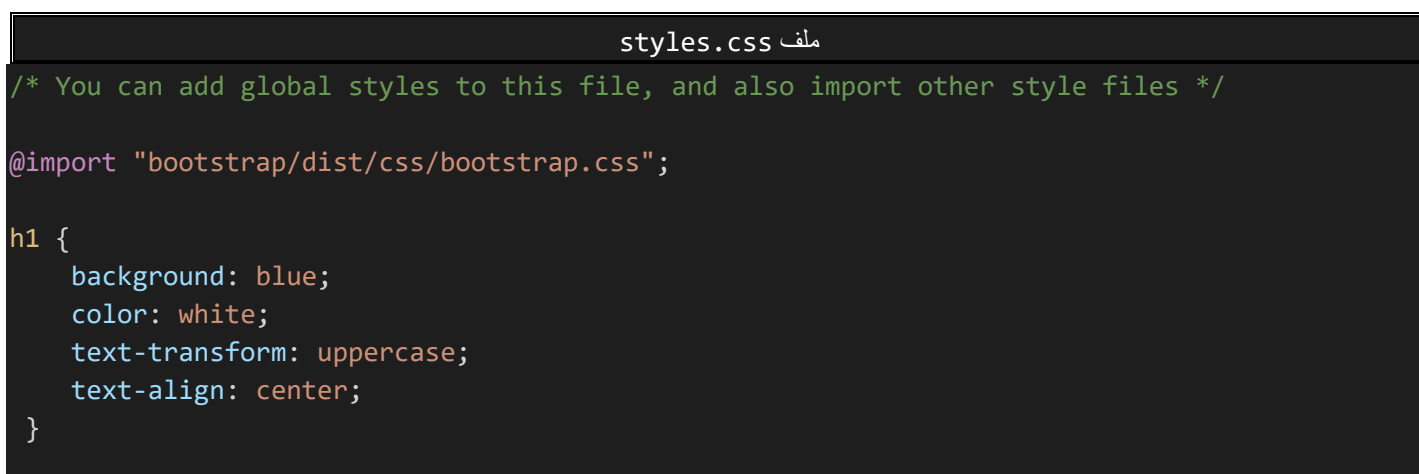
وهذا النوع هو الذي له أولوية التنفيذ، عن باقي الأنواع الأخرى ولكن فريق تطوير Angular لا ينصح باستخدام هذه الطريقة. وبذلك تعلمنا أيضاً طرق إضافة أكواد Styles إلى أي component، وبقي ان نتكلم عن كيفية إضافة أكواد styles بشكل عام بحيث يتم تطبيقها على جميع أجزاء التطبيق.

#### 4.7.4. Global Styles:

في حال اردت عزيزي المتعلم لأي سبب من الأسباب ان تضيف مجموعة من التنسيقات وتريد ان تكون global أي على مستوى كامل التطبيق وبنفس الوقت لا تريد ان تُغير نظام View Encapsulation فتستطيع ذلك عن طريق كتابة هذه الاكواد في ملف styles.css، وهو موجود في المجلد src، كالتالي:



وهذا الملف نستطيع الاستفادة منه في كتابة التنسيقات العامة وايضاً في حال اردنا ان نضيف مكتبة تنسيقات خارجية فنقوم باستدعائها في هذا الملف لكي تصبح عامة لجميع أجزاء التطبيق، كالتالي:



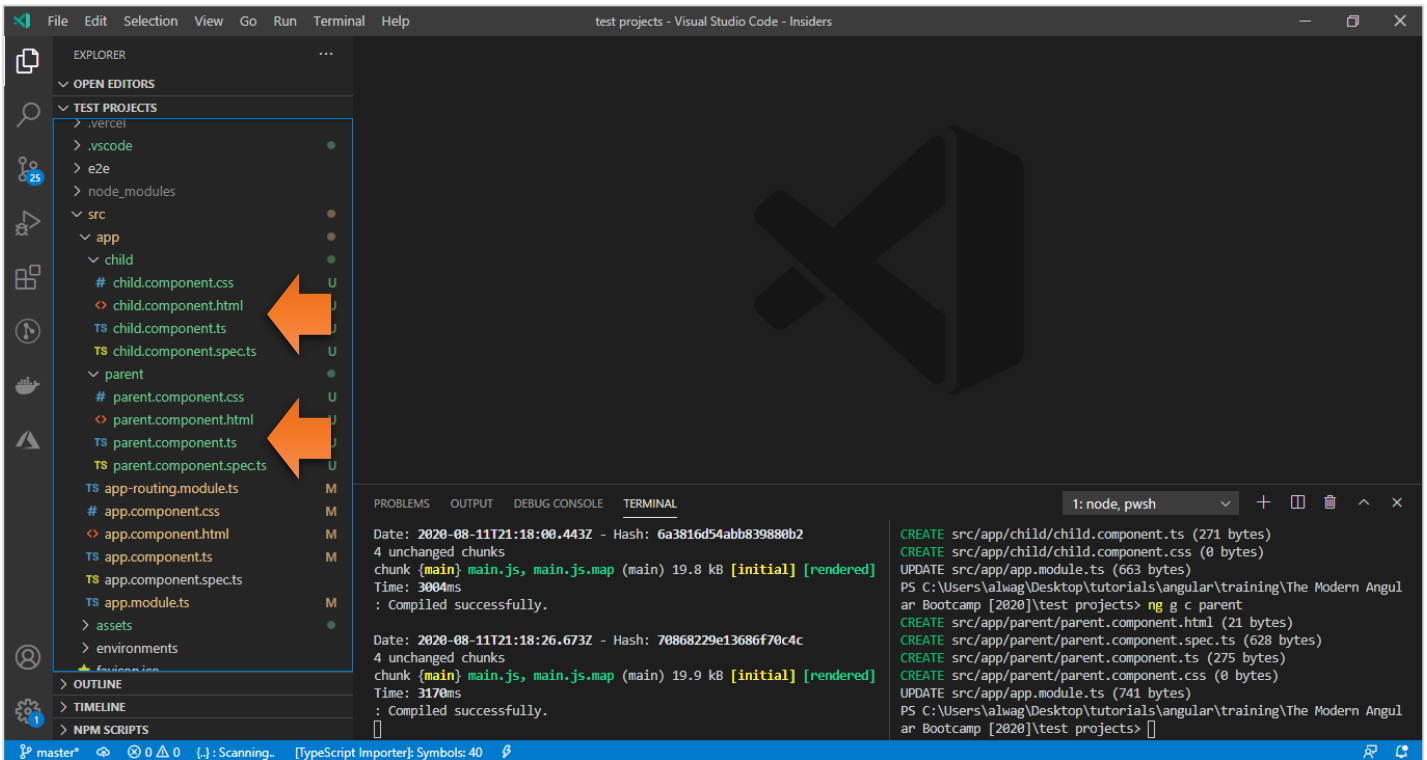
نلاحظ اننا استدعينا مكتبة bootstrap بالإضافة إلى إضافة تنسيقات إلى هذا الملف لكي تطبق على جميع h1 في التطبيق.

## 5.7.4. Component Styles Special Selectors

قدمت لنا Angular مجموعة من selectors التي تساعدنا في الوصول إلى عناصر Shadow DOM وهم `:host`, `:host-context` and `::ng-deep`، وسوف نستعرض جميع هذه selectors مع التوضيح بمثال بسيط لكلاً منها.

### 1.5.7.4. :host

وهو خاص بتنسيق component نفسه المستضيف للعناصر سواء كانت عناصر HTML او حتى عناصر أخرى custom selectors. ولتوضيح لنقم بإنشاء اثنين components واحد باسم child والثاني باسم parent، كالتالي:



ولنقم بإضافة selector الخاص بالcomponent الأب إلى Root Component والـ selector الخاص بالchild إلى parent، كالتالي:

ملف app.component.html

```
<h3>Root Component</h3>
<app-parent></app-parent>
```

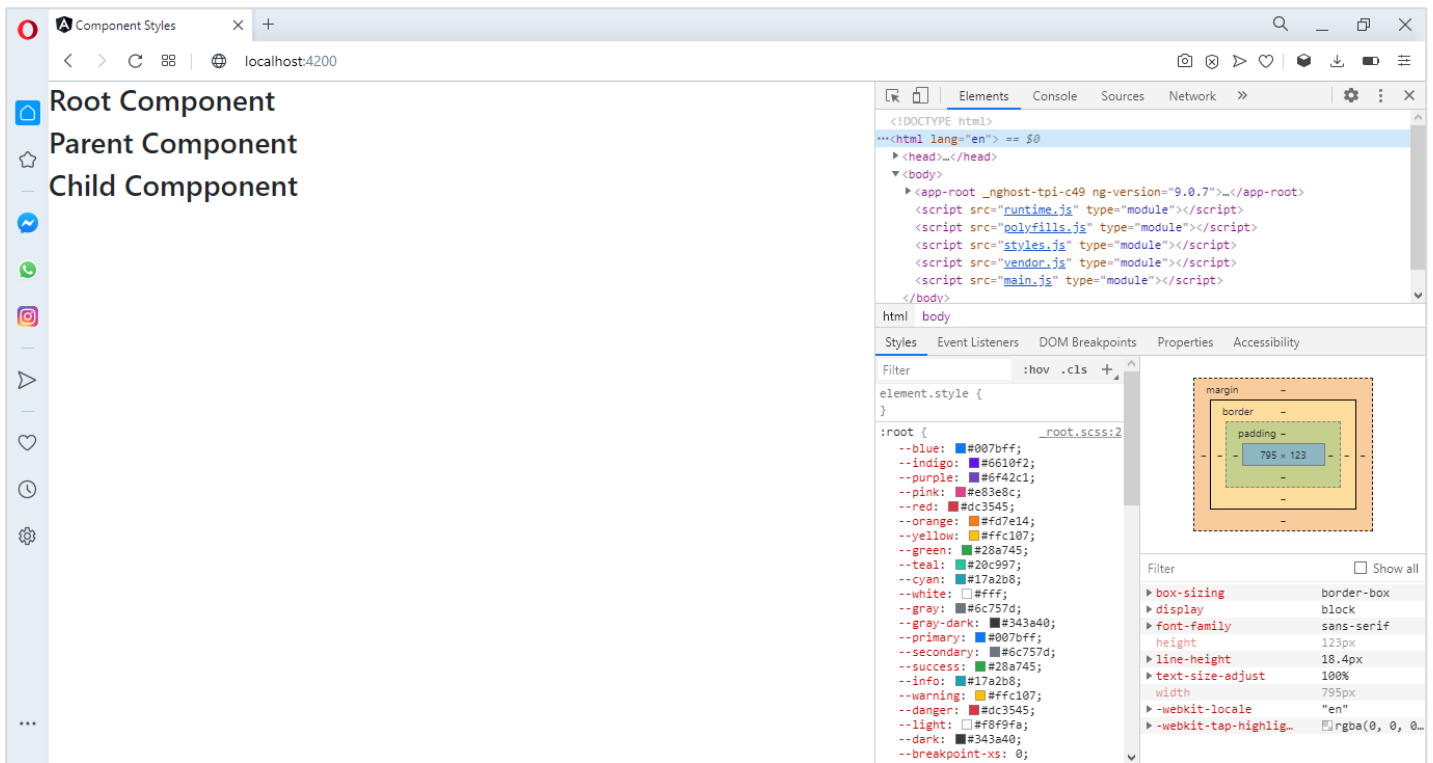
ملف parent.component.html

```
<h3>Parent Component</h3>
<app-child></app-child>
```

ملف child.component.html

```
<h3>Child Component</h3>
```

اما النتيجة في المتصفح فسوف تكون كالتالي:

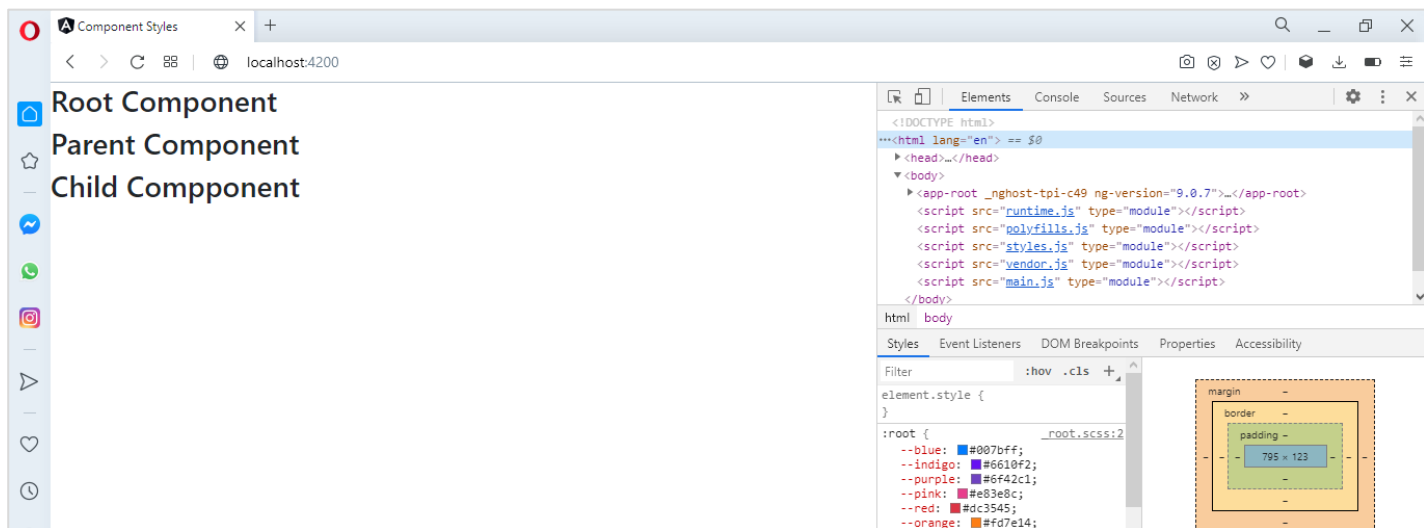


الآن لنفرض اننا نريد ان نقوم بتنسيق component ذو الاسم child نفسه الذي يحمل selector ذو الاسم <app-child> وليس العنصر h3 الذي يحتوي على الرسالة child component، ولو حاولنا كتابة اسم هذا selector مباشرة في ملف style لهذا component، فلن يُطبق عليه أى تنسيق، كالتالى:

ملف `child.component.css`

```
app-child {
  position: absolute;
  top: 10%;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}
```

ولنقم بمشاهدة النتيجة في المتصفح ولنرى النتيجة:



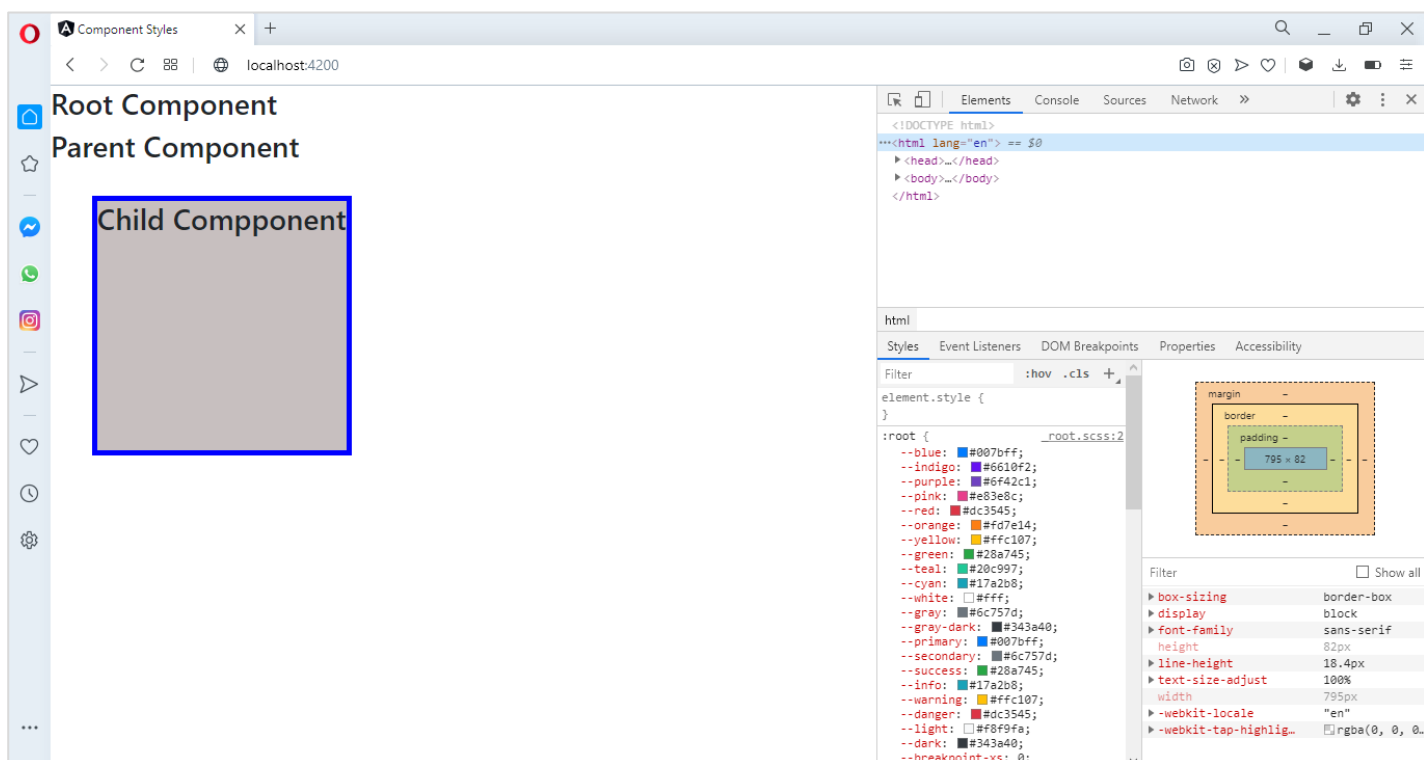
نلاحظ لم يُطبق التنسيق على هذا component لذلك نحتاج إلى special selector يشير إلى component المستضيف للعناصر وهو `:host`، والآن لنقم باستبدال اسم selector بهذا special selector، كالتالي:

```

child.component.css ملف
:host {
  position: absolute;
  top: 10%;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}

```

والنتيجة سوف تكون، كالتالي:





#### :host-context .2.5.7.4

وهي مشابهة host: ولكن هنا لا يتم التنفيذ إلا في حال وقوع شرط معين من خارج component او بصيغة أخرى من خارج host، بمعنى لو كان لدينا class css معين وليكن في Root Component وهذا class لا نضيفه إلى إذا تحقق شرط معين عن طريق class binding التي تعلمناها سابقاً، وب نفس الوقت ليكن host-context: موجودة في component آخر وليكن child، فإن host-context تراقب هذا class وعند تحقق إضافة هذا class أي تحقق الشرط فيتم تنفيذها، ولتوضيح لنقم بإنشاء class css وليكن اسمه my-color ويقوم بتغيير لون الخط لي host ذو الاسم parent، كالتالي:

ملف app.component.css

```
:host {
  position: absolute;
  top: 10%;
  margin: 5% 5%;
  border: 5px solid green;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}

.my-color {
  color: white;
}
```

نلاحظ اضعفنا هذا class والآن لنقم بتعريف متغير لكي يتم إضافة او حذف هذا class بناءً على قيمته بحيث إذا كانت true يتحقق الشرط ويتم إضافة هذا class وإذا كانت false لا يتحقق الشرط ويتم حذف هذا class، يجب الإشارة انه في التطبيقات الواقعية هذا الشرط يكون مبني على بيانات واقعية بناءً عليها يتم تغيير حالة الشرط من false إلى true اما هنا فنستخدم فقط لكي نشرح مفهوم host-context:، كالتالي:

ملف app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  changeColor = false;

  constructor() { }

  ngOnInit(): void { }
}
```

بعدها قمنا بتعريف المتغير وإعطاءه قيمة مبدئية false ننتقل الآن إلى ملف template لهذا component ونضيف زر مهمته تغيير قيمة المتغير من true إلى false او من false إلى true مع كل ضغطة، وبنفس الوقت نعمل class binding، كالتالي:

ملف app.component.html

```
<button (click)="changeColor = !changeColor">change color</button>
<h3>Root Component</h3>
<app-parent [class.my-color]="changeColor"></app-parent>
```

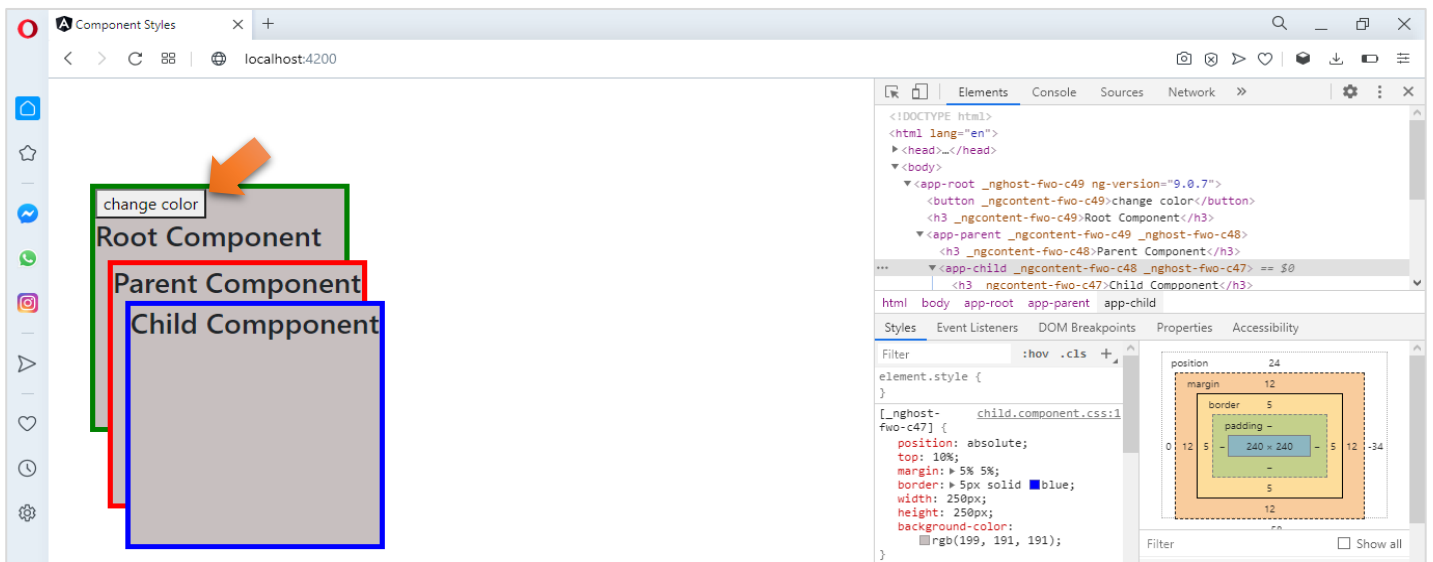
الآن نريد مراقبة حالة هذا class والذي هو my-color في component آخر بحيث إذا تمة اضافته نقوم بتنفيذ كود css، كالتالي:

ملف child.component.css

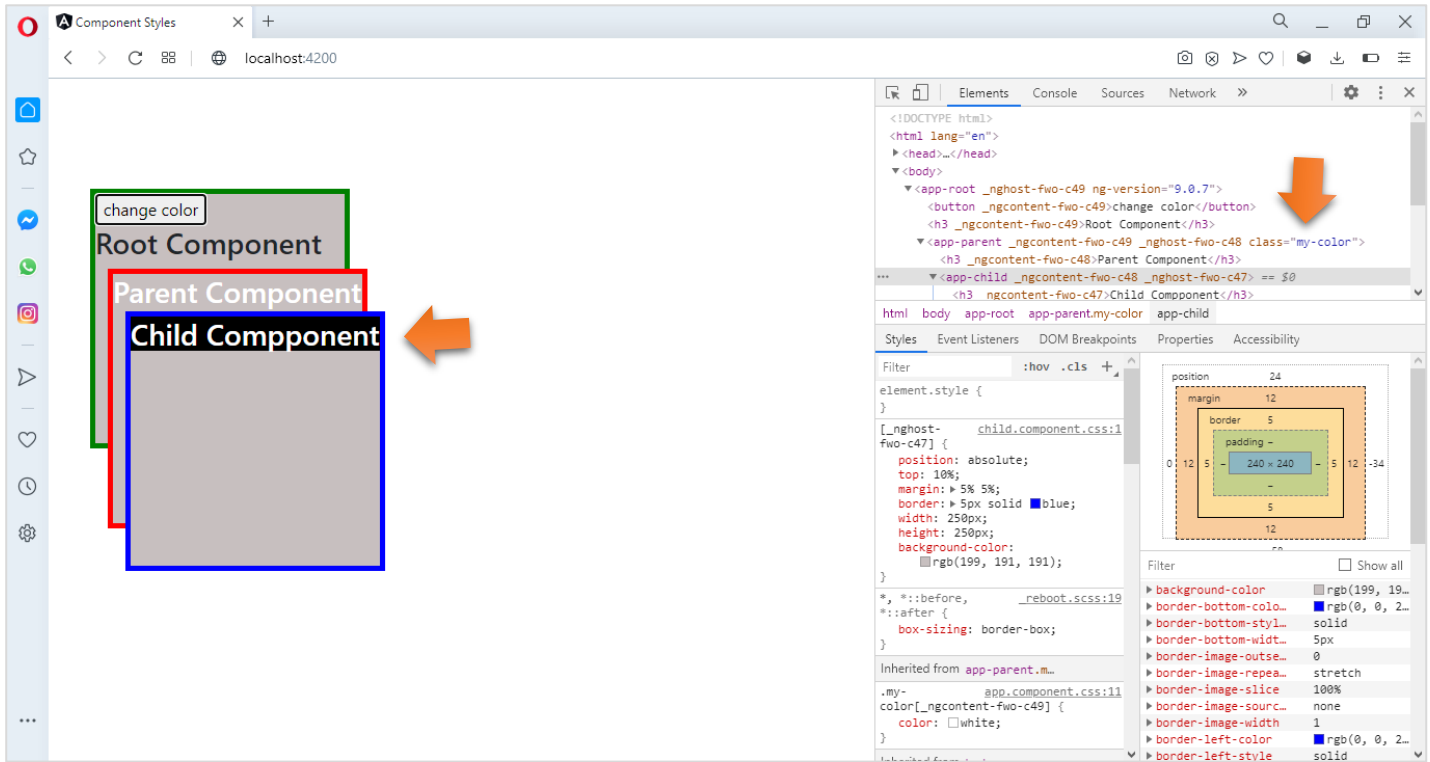
```
:host {
  position: absolute;
  top: 10%;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}

:host-context(.my-color) h3 {
  background-color: black;
  color: white;
}
```

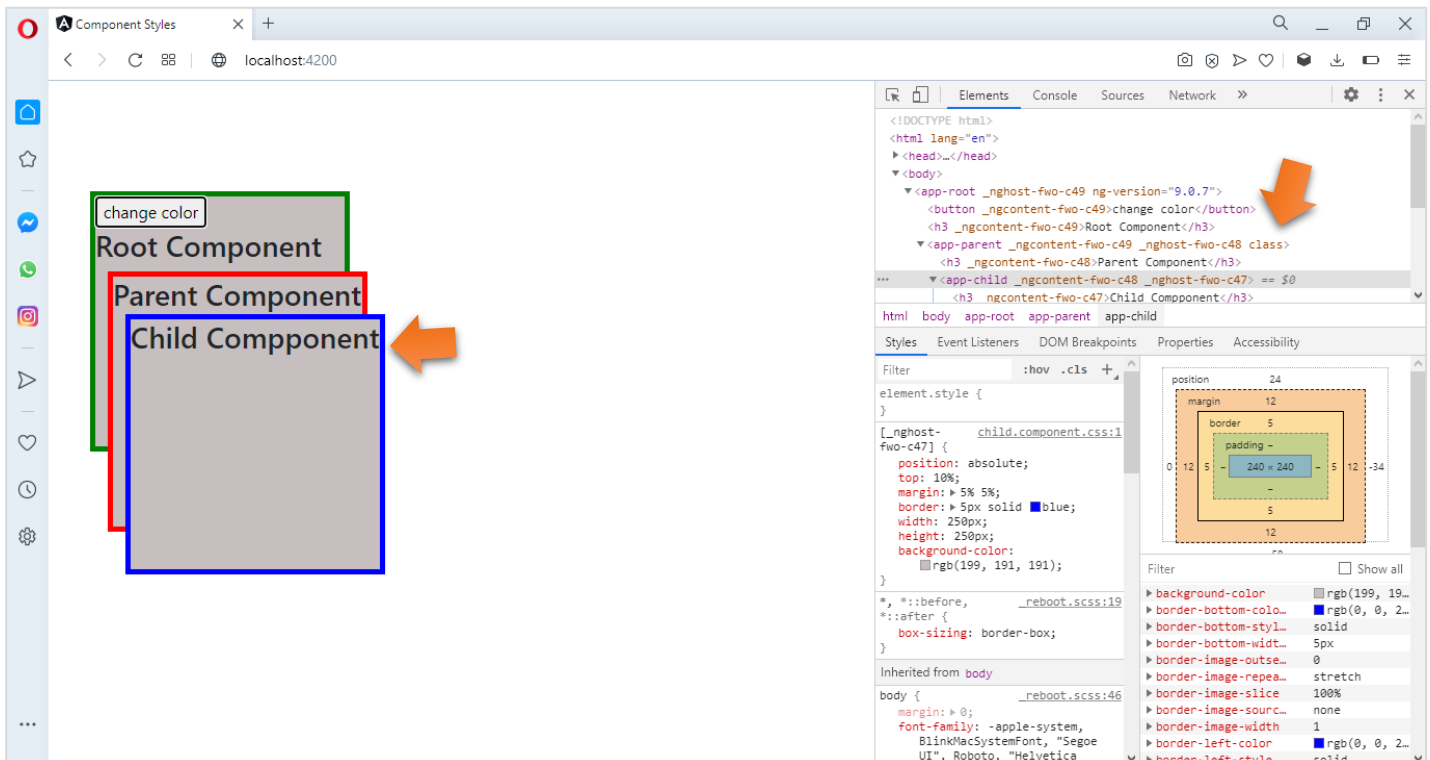
نلاحظ راقبنا class css السابق باستخدام host-context: وفي حال تحقق الشرط (مع العلم ان الشرط موجود خارج هذا component لذلك اشرت في بداية حديثي انه يتم تنفيذ هذا الامر في حال وقوع شرط معين خارج component) يتم تنفيذ كود css معين على العنصر h3 ولك حرية اختيار أي عنصر تريد موجود في هذا component سواء بكتابة اسمه مباشرة او id الخاص به او حتى اسم class آخر يحتويه هذا العنصر.



## لنقم بالضغط على الزر ونرى النتيجة:



نلاحظ بعد الضغط على الزر تمت إضافة class css ذو الاسم my-color كما هو مبين في يمين الصفحة، وبنفس الوقت تم اكتشاف هذا التغيير في component آخر هو child وتم تطبيق بعض أكواد css على العنصر h3 الذي يحتويه هذا component كما هو مؤشر عليه في السهم على يسار الصفحة، عند الضغط على الزر مرة أخرى فإن قيمة changeColor من true إلى false وبذلك سوف يتم حذف class css ذو الاسم my-color وبناءً عليه سوف يتم الغاء أكواد css التي تم تطبيقها سابقاً على العنصر h3، كالتالي:



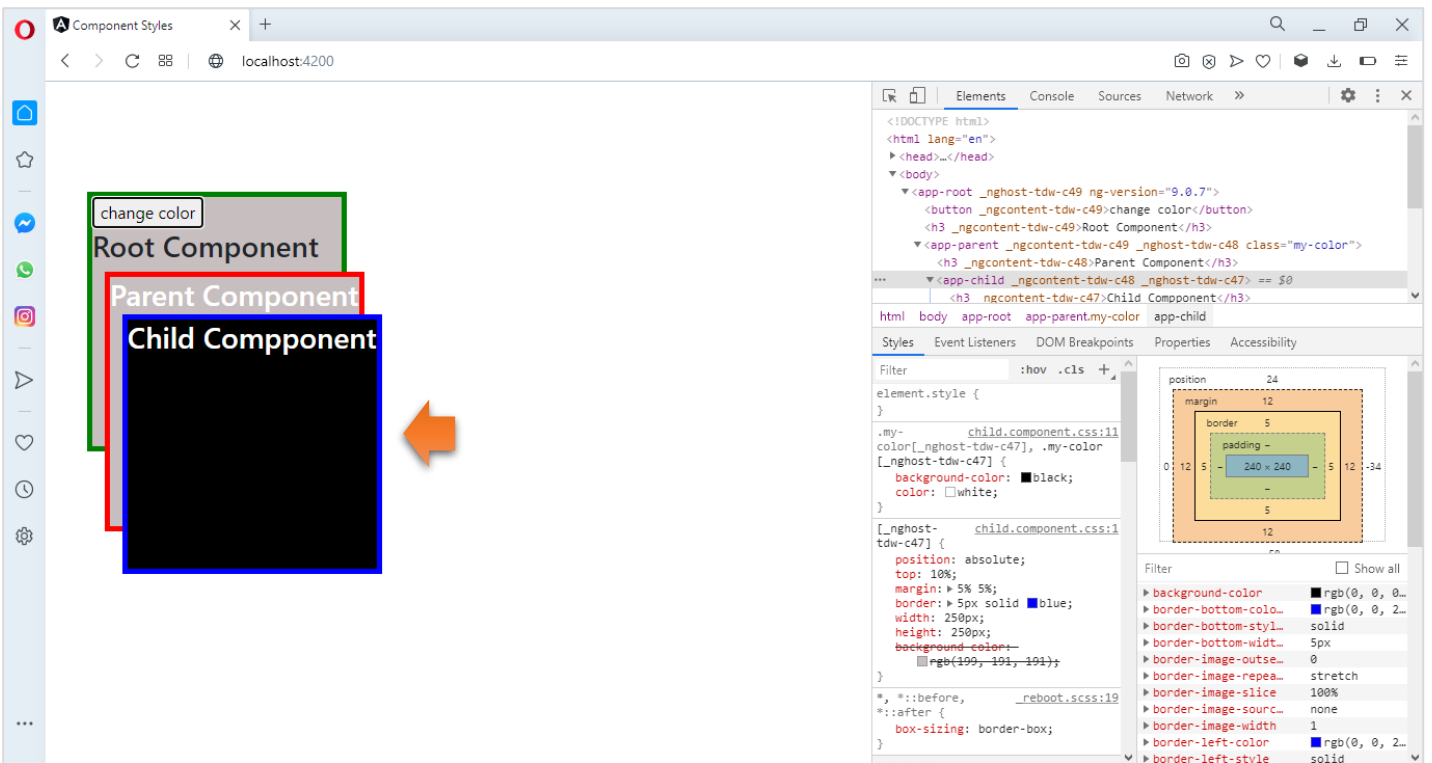


أما في حال اردنا ان نطبق مجموعة تنسيقات على host نفسه والذي هو في مثالنا هذا child component وليس العنصر h3 فنقوم فقط بحذف اسم هذا العنصر، والإبقاء على الامر (my-color).host-context، كالتالي:

```
child.component.css ملف
:host {
  position: absolute;
  top: 10%;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}

:host-context(.my-color) {
  background-color: black;
  color: white;
}
```

والنتيجة، في المتصفح تكون كالتالي:



نلاحظ التنسيق تم تطبيقه على host الذي هو component نفسه وليس العنصر الذي بداخله.

### 3.5.7.4 ::ng-deep

وهذا الامر deprecated ولكن إلى لحظة تأليف هذا الكتاب لا يزال اطار عمل Angular يدعم هذه الخاصية، بحسب الموقع

الرسمي <https://angular.io/guide/component-styles#deprecated-deep--and-ng-deep>

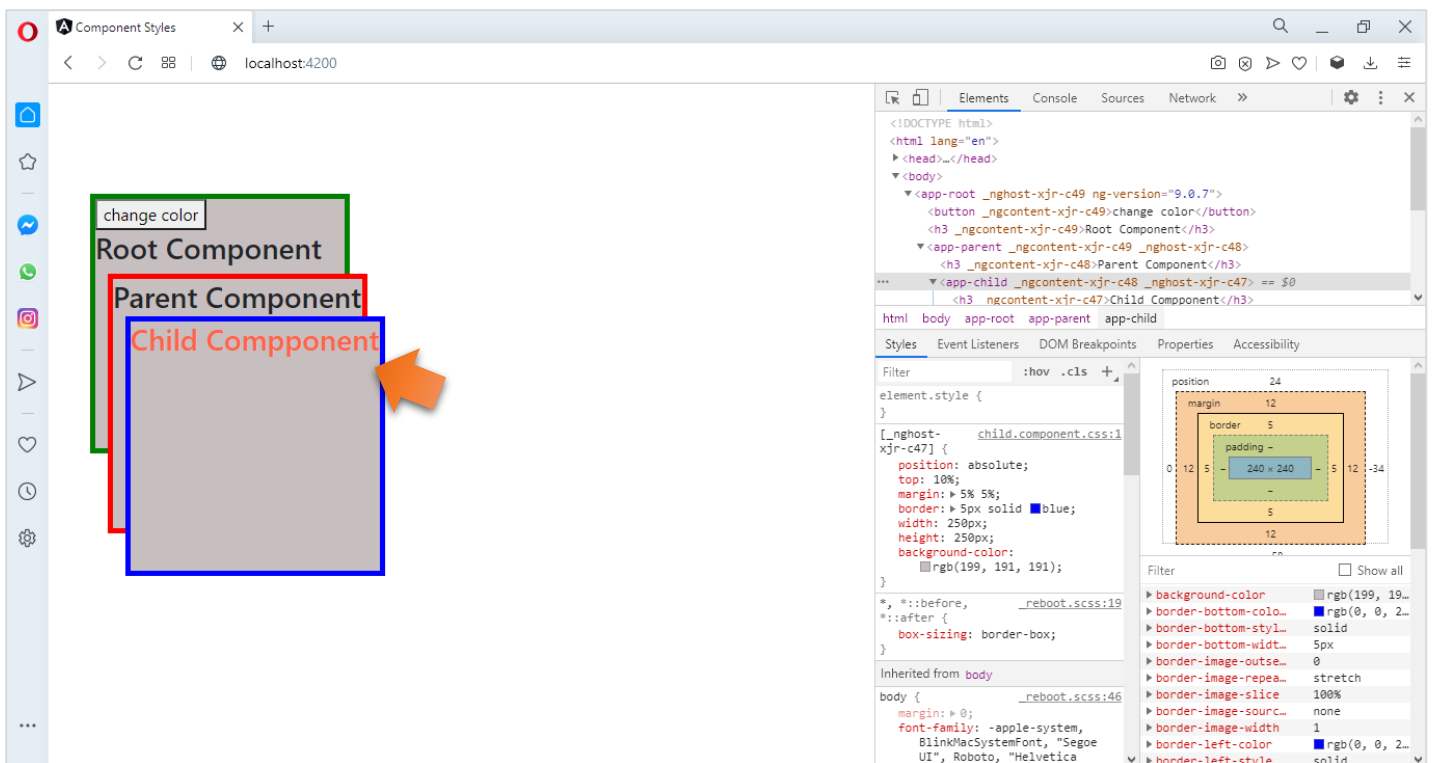
ووظيفته هو الوصول إلى عنصر او مجموعة عناصر في component الأب وتطبيق تنسيقات معينة عليها عن طريق component الأب.

ولنفرض اننا نريد الوصول إلى العنصر h3 الموجود في component الأب وتطبيق تنسيق معين عليه، ونستطيع عمل ذلك بالذهاب إلى component الأب وبالتحديد ملف style لهذا component وكتابة الكود التالي:

```
parent.component.css
ملف
:host {
  position: absolute;
  top: 10%;
  margin: 20% 5%;
  border: 5px solid Red;
  width: 250px;
  height: 250px;
  background-color: rgb(199, 191, 191);
}

:host app-child ::ng-deep h3 {
  color: tomato;
}
```

والنتيجة في المتصفح تكون كالتالي:



وبذلك قمنا بتغطية اغلب المفاهيم الخاصة Component Styles، وبقي ان نشير أن Angular يتيح للمطورين استخدام جميع الأنماط سواء sass او scss ولا يقتصر فقط على css التقليدية.

## 8.4. Renderer2 Service:

من الأمور المهمة لتطوير الويب هي القدرة على الوصول إلى العناصر في DOM والتحكم والتعديل بها وبخصائصها، وقدمت لنا Angular طريقتين الأولى، ElementRef وقد اشرنا لها سابقاً وهي تتيح لنا الوصول إلى جميع عناصر DOM والتحكم بجميع خصائصه بشكل مباشر، ولكن فريق تطوير Angular لا ينصح باستخدام هذه الطريقة ولعل من أهم الأسباب هو ان كود Angular يتم استخدامه في أكثر من platform مثل لبرمجة تطبيقات الموبايل عن طريق ionic او تطبيقات سطح المكتب عن طريق electron واخيراً تطبيقات الويب عن طريق إطار عمل Angular نفسه، لذلك يُفضل ان نستخدم بدلاً منها service جاهزة اسمها Renderer2، حيث نصل إلى العنصر المراد التحكم به عن طريق ElementRef ونمرره إلى هذه service التي تحتوي مجموعة من الدوال نستطيع عن طريقها التحكم بخصائص هذا العنصر.

وسوف نتكلم أن اغلب الخصائص والدوال التي تحتويها هذه service مع التوضيح بالأمثلة.

### 1.8.4. createElement() & createText() & appendChild():

هذه الثلاث دوال لها من اسمها نصيب فالأولى createElement تقوم بإنشاء عنصر HTML والثانية تقوم بإنشاء نص ونستطيع ان نستخدمها بدلاً من الخاصيتين innerHTML او textContent اما الدالة الثالثة فتقوم بإضافة عنصر داخل عنصر آخر، سواء كان هذا العنصر موجود بالأساس او قمنا بإنشائه ديناميكياً عن طريق الدالة createElement ونستخدمها ايضاً مع النصوص لإضافة النص المنشأ عن طريق الدالة createText إلى عنصر معين، ولتوضيح لنقوم بإعطاء مثال، ولنفرض انه لدينا قائمة ul فارغة وهناك زر وهذا الزر كل ما نقوم بالضغط عليه سوف يقوم بإنشاء عنصر li جديد وبنفس الوقت يُنشئ نص ويضيف هذا النص للعنصر المنشئ سابقاً، كالتالي:

ملف app.component.html

```
<div>
  <button class="btn btn-primary" (click)="addItem()">Add Item</button>
</div>
<ul #ulElement> </ul>
```

نلاحظ وجود الزر وهذا الزر عند الضغط عليه يُنفذ الدالة addItem (Event Binding) ونلاحظ ايضاً انه لدينا القائمة ul وقد اعطيناها Template Reference باسم #ulElement، ونلاحظ ايضاً ان هذه القائمة فارغة لا تحتوي على أي li.

الآن لنقم بالوصول إلى هذا العنصر ul في ملف class عن طريق ViewChild @ViewChild، كما تعلمنا سابقاً، وبنفس الوقت نكتب محتويات الدالة addItem، كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
```

```

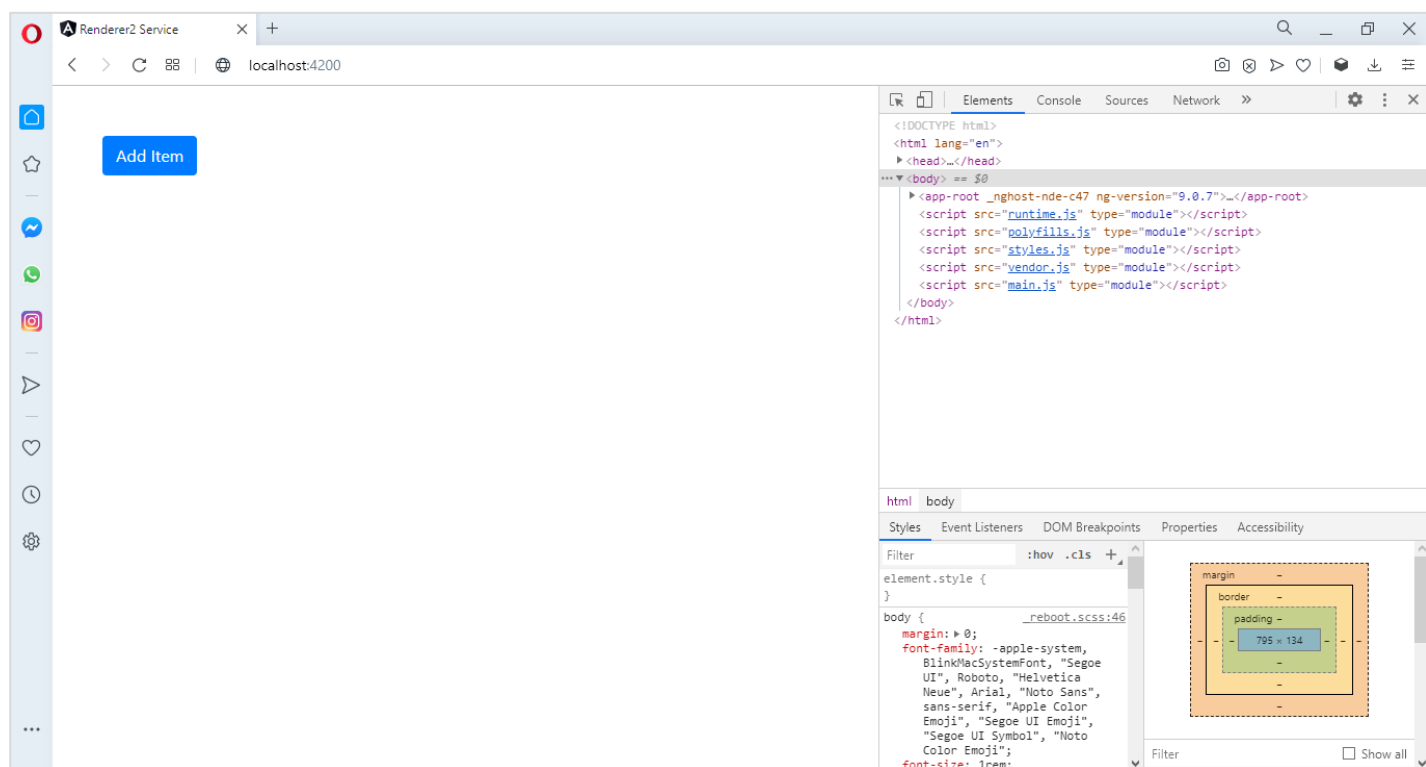
@ViewChild('ulElement') private element: ElementRef;
private count = 0;

constructor(private renderer: Renderer2) { }

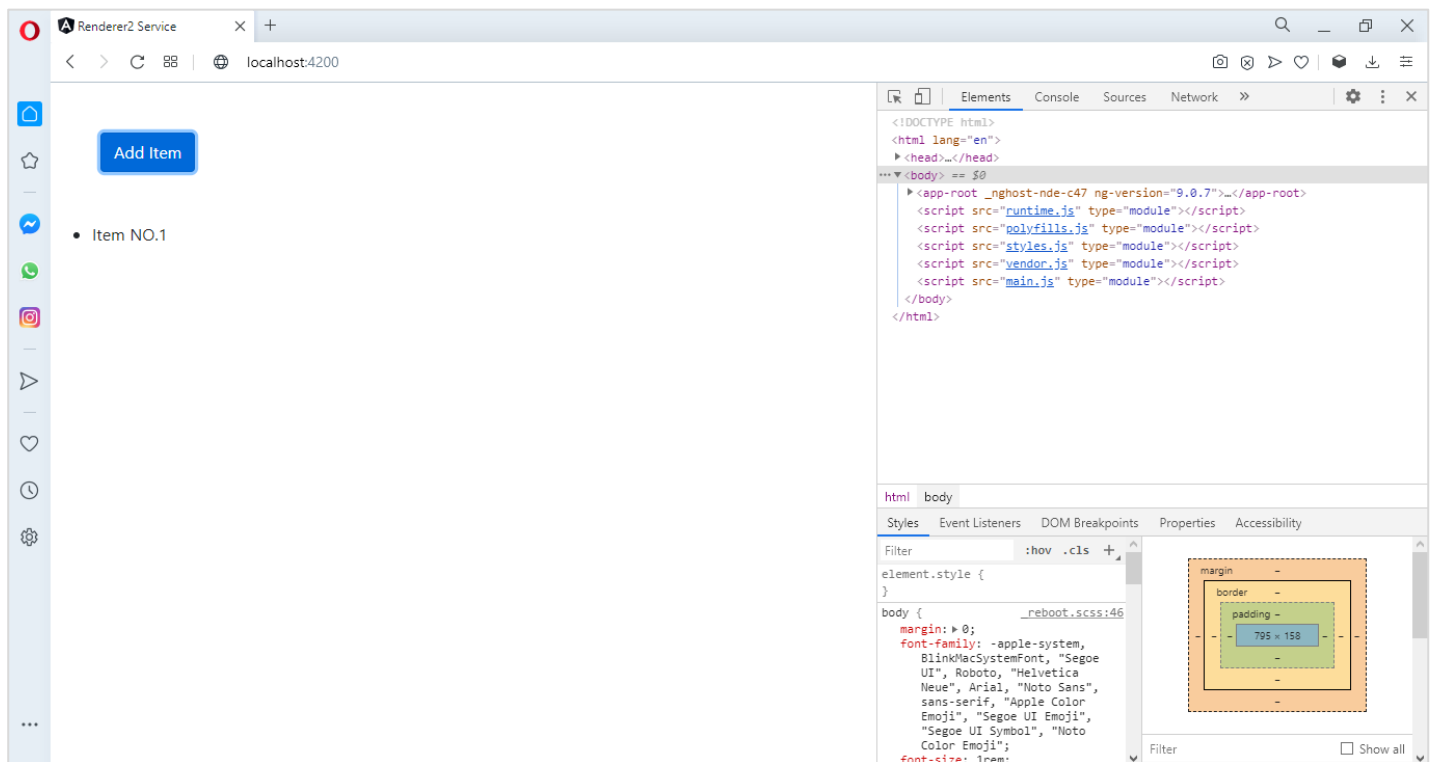
addItem() {
  this.count++;
  const el = this.renderer.createElement('li');
  const text = this.renderer.createText(`Item NO.${this.count}`);
  this.renderer.appendChild(el, text);
  this.renderer.appendChild(this.element.nativeElement, el);
}
}

```

محتوى الدالة هو الذي يهمنا حيث ان المتغير count فقط لإعطاء رقم جديد مع كل ضغطة على الزر وهو لا يؤثر على ما نريد شرحه وتبينانه هنا وانما فقط اصفته لتجميل، السطر الذي يليه عرفنا ثابت باسم el بحيث نُخزن فيه العنصر li الجديد الذي نريد انشاءه، والسطر الذي يليه ايضاً قمنا بتعريف ثابت لتخزين النص الجديد، اما السطرين الأخيرين فواحد لإضافة النص إلى العنصر والثاني لإضافة العنصر إلى القائمة ul، اما النتيجة في المتصفح فتكون كالتالي:



لنقوم بالضغط على الزر ولنرى النتيجة:



نلاحظ عند الضغط على الزر انشأ العنصر والنص وأضاف النص للعنصر ومن ثم أضاف العنصر بشكله النهائي إلى القائمة ul، ولو قمنا بتكرار الضغط فسوف يقوم بإنشاء عنصر جديد مع كل ضغطة.

## 2.8.4. insertBefore()

ومن اسمها تضيف عنصر قبل آخر عنصر مُضاف، ولتوضيح لنقوم بإضافة زر آخر وينفذ دالة وليكن اسمها insertItemBefore، كالتالي:

```

ملف app.component.html
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <ul #ulElement> </ul>
</div>

```

نلاحظ قمت بإعادة كتابة Markup حيث استخدمت بعض كلاسات مكتبة bootstrap لإضفاء لمسة جمالية بسيطة، والآن لنكتب محتويات هذه الدالة في ملف class، كالتالي:

```

import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

```

```

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2) { }

  addItem() {
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.appendChild(this.element.nativeElement, el);
  }

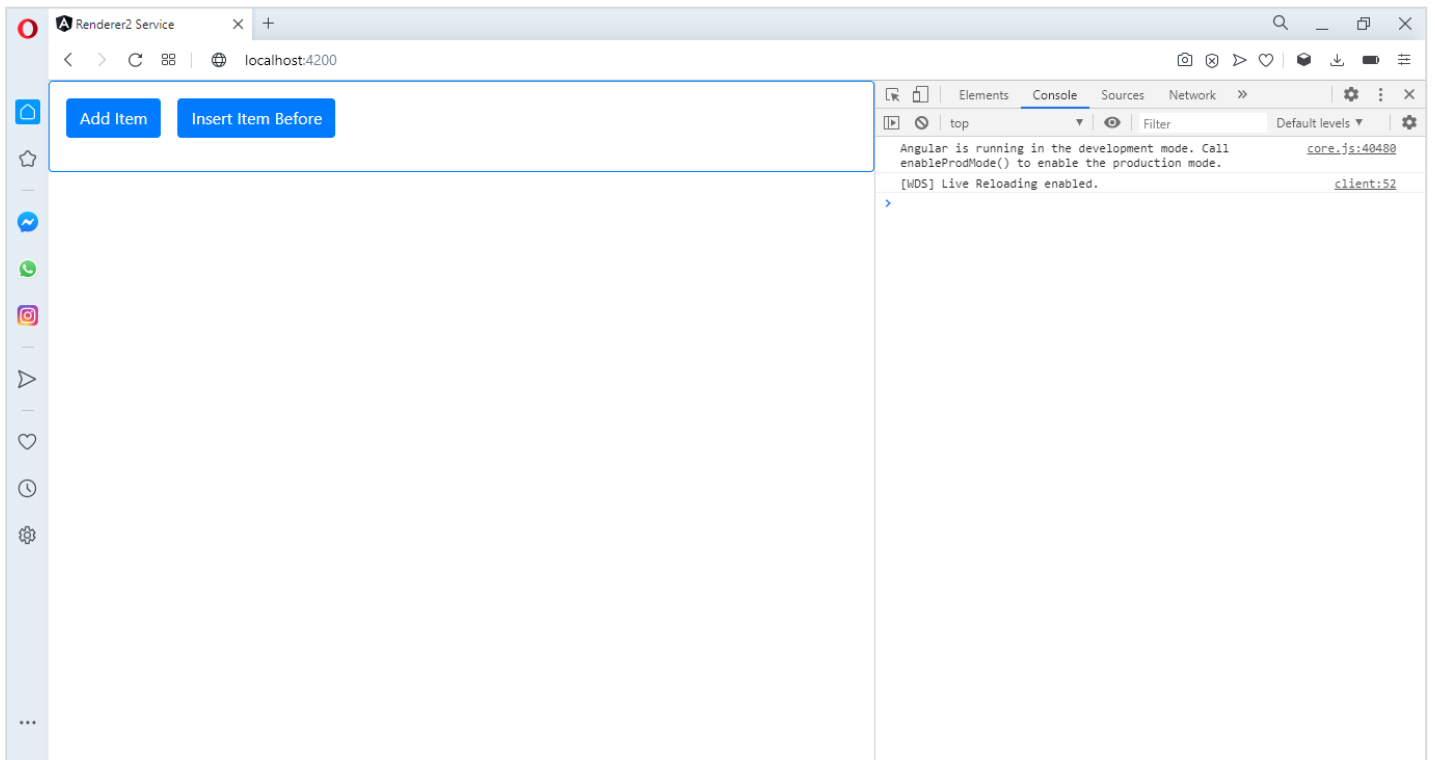
  insertItemBefor() {
    const index = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.insertBefore(this.element.nativeElement, el, refChild[index]);
  }
}

```

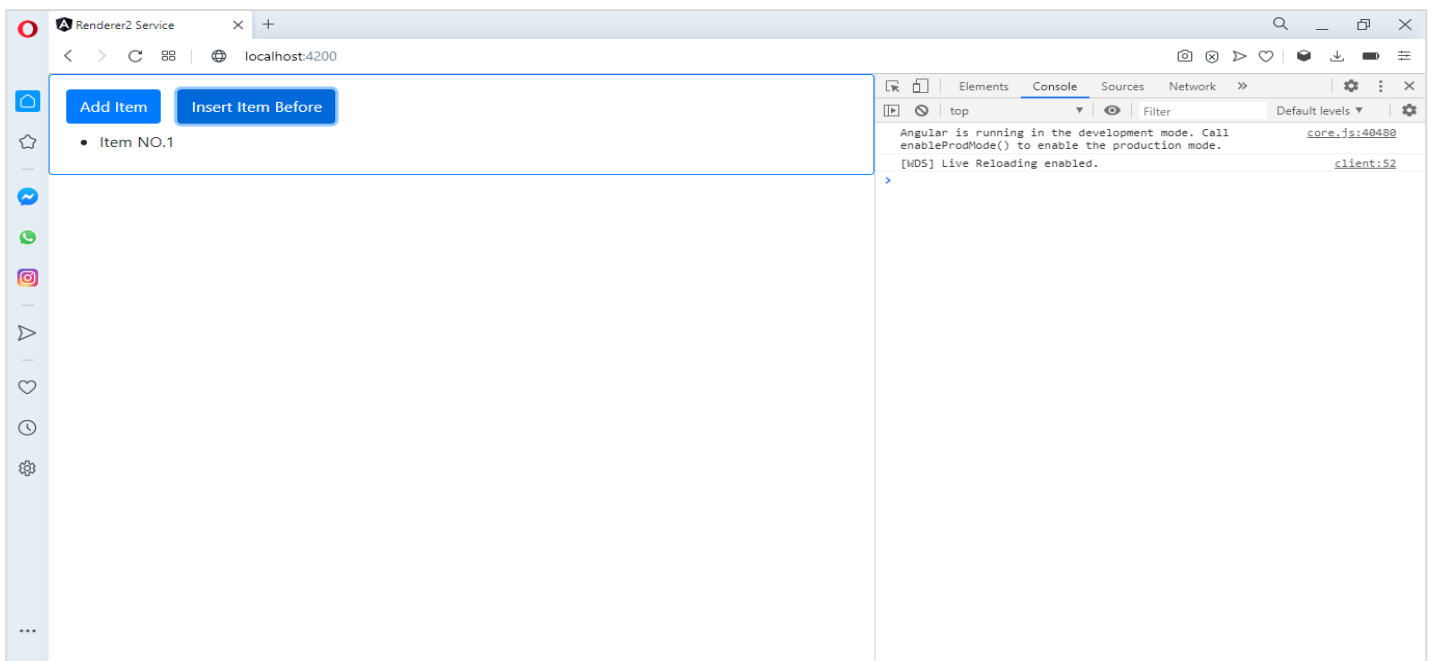
الاسطر الجديدة هي ما تم التأشير عليه بالأسهم اما باقي الاسطر فقد تم شرحها سابقاً، حيث اول سطر قمنا بتعريف ثابت يأخذ رقم آخر قيمة في المتغير count ويشرح منها واحد لكي نستخدم هذا الثابت لاحقاً لتحديد موقع آخر عنصر li في القائمة ul، مع العلم انه يجب وضعه قبل السطر this.count++ وفي حال اردت وضعه بعد هذا السطر فتقوم بطرح 2 وليس 1.

السطر الثاني الذي تم التأشير عليه بالسهم خاص بالحصول على جميع العناصر li (الأبناء) لي العنصر ul (الأب) والذي قمنا بالوصول له سابقاً باستخدام Template Reference وعن طريق ViewChild وهذا السطر سوف يُرجع لنا مصفوفة من عناصر li.

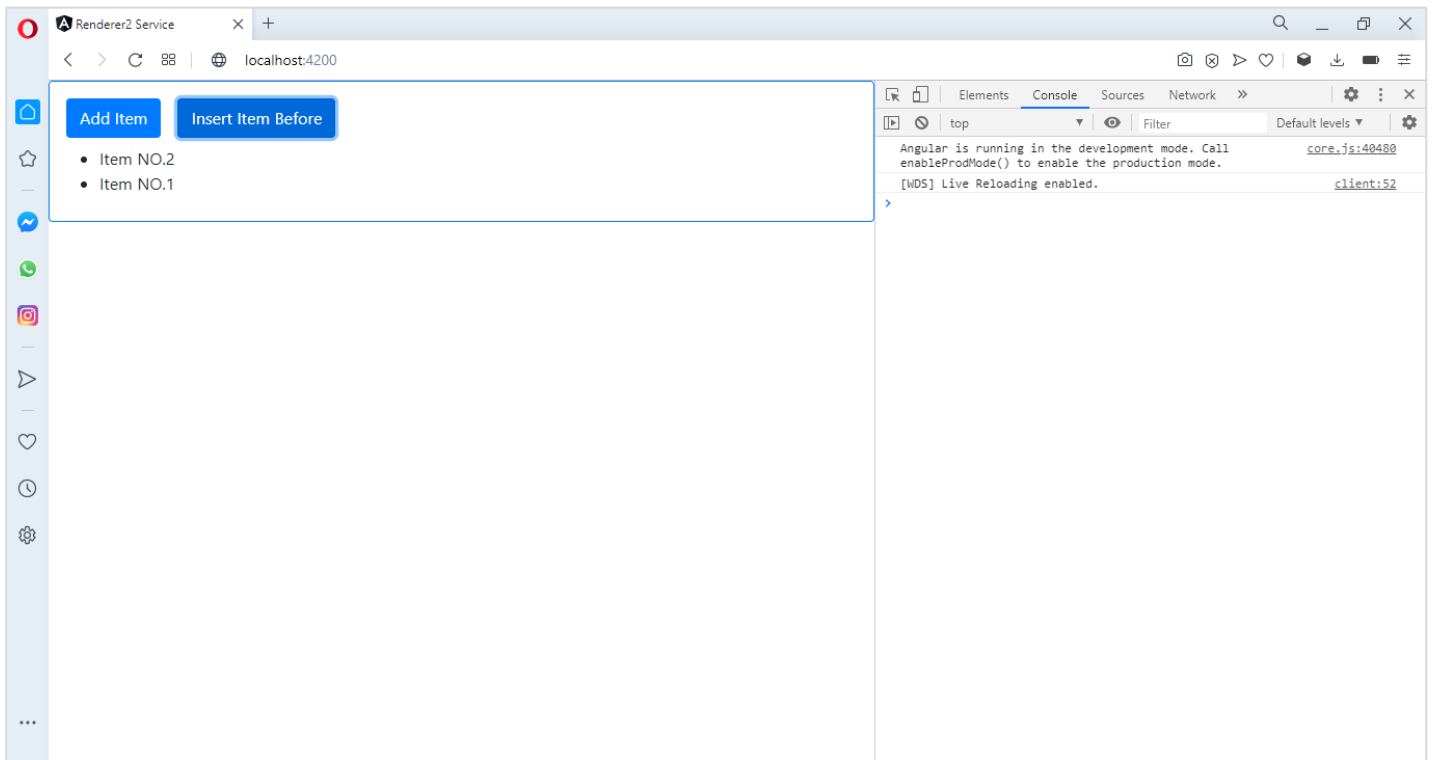
وآخر سطر هو لإضافة العنصر آخر عنصر عن طريق الدالة insertBefore حيث تستقبل ثلاث بارامترات الأول هو العنصر الأب الذي نريد إضافة هذا العنصر بداخله، والثاني هو العنصر نفسه الذي نريد اضافته والأخير هو آخر عنصر موجود بمصفوفة العناصر لكي نضيف هذا العنصر قبله، اما الآن لنشاهد النتيجة في المتصفح:



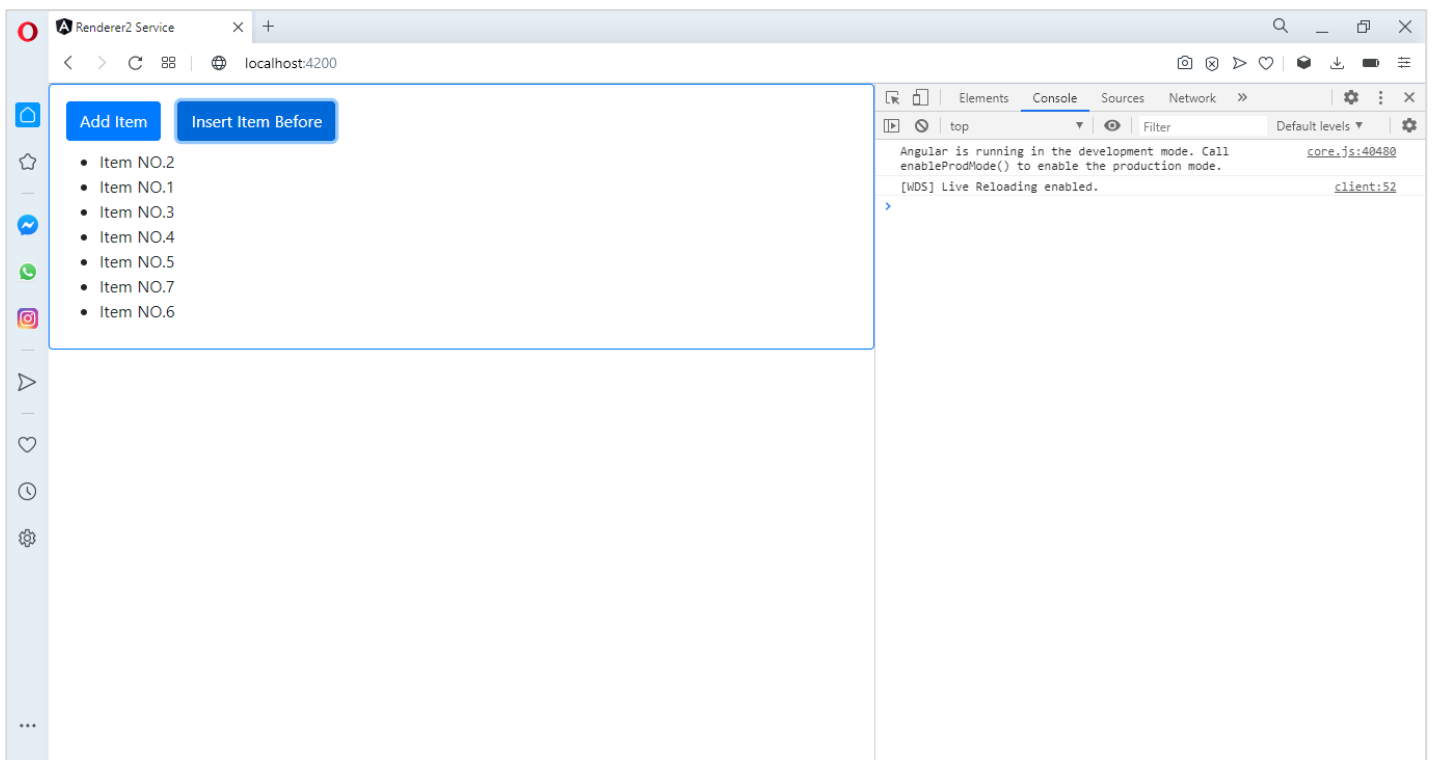
في البداية لا يوجد أي عنصر لذلك سواء ضغطنا على الزر Add Item او Insert Item Before جميعهما يؤديان نفس النتيجة،  
كالتالي:



حيث قمت بالضغط على الزر Insert Item Before وتم إضافة أول عنصر، ولكن لنشاهد النتيجة عند الضغط على نفس الزر مرة أخرى:



نلاحظ أضف العنصر الجديد قبل آخر عنصر مضاف، الآن لنقم بتجربة أخرى ولنقم بالضغط على الزر الآخر Add Item كذا مرة ومن ثم لنقوم بالضغط على الزر Insert Item Before، كالتالي:



نلاحظ عندما قمنا بالضغط على زر Add Item قام ببدء العد من عند آخر قيمة سابقة وهي 2 حيث بدأ بالرقم 3 ومن ثم عندما وصل إلى الرقم 6 قمت بالضغط على الزر Insert Item Before أضف العنصر رقم 7 قبل آخر عنصر موجود.



### 3.8.4.removeChild()

هذه الدالة تقوم بحذف العنصر او مجموعة عناصر، حيث تستقبل ثلاث باراميترات اثنان ضروريات والأخير اختياري، الأول هو العنصر الأب الذي نريد حذف العناصر الأبناء الخاصة به والثاني هو خاص بالعناصر او العنصر الابن المراد حذفه، ولتوضيح لنقوم بإضافة زر بحيث إذا قمنا بالضغط عليه يقوم بحذف عنصر القائمة ul مع جميع ما يحتويه من عناصر أبناء، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <ul #ulElement> </ul>
</div>
```

نلاحظ وجود الزر والذي يحتوي على الدالة deleteItem، وبنفس الوقت يجب ملاحظة ان العنصر ul هو ابن للعنصر div، وكما قلنا سابقاً ان الدالة تحتاج إلى باراميتين ضروريتين هما العنصر الأب والعنصر الأبن، لذلك لدينا مجموعة حلول منها ان نضع Template Reference للعنصر الاب div ومن ثم في ملف class وعن طريق ViewChild نستقبل هذا Template Reference، او نستخدم دالة جاهزة موجودة في ElementRef، ولكي نقوم بتغطية مفاهيم جديدة سوف استخدم الطريقة الثانية لأن الأولى استخدمناها كثيراً، كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;
  constructor(private renderer: Renderer2) { }

  addItem() {
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.appendChild(this.element.nativeElement, el);
  }
}
```

```

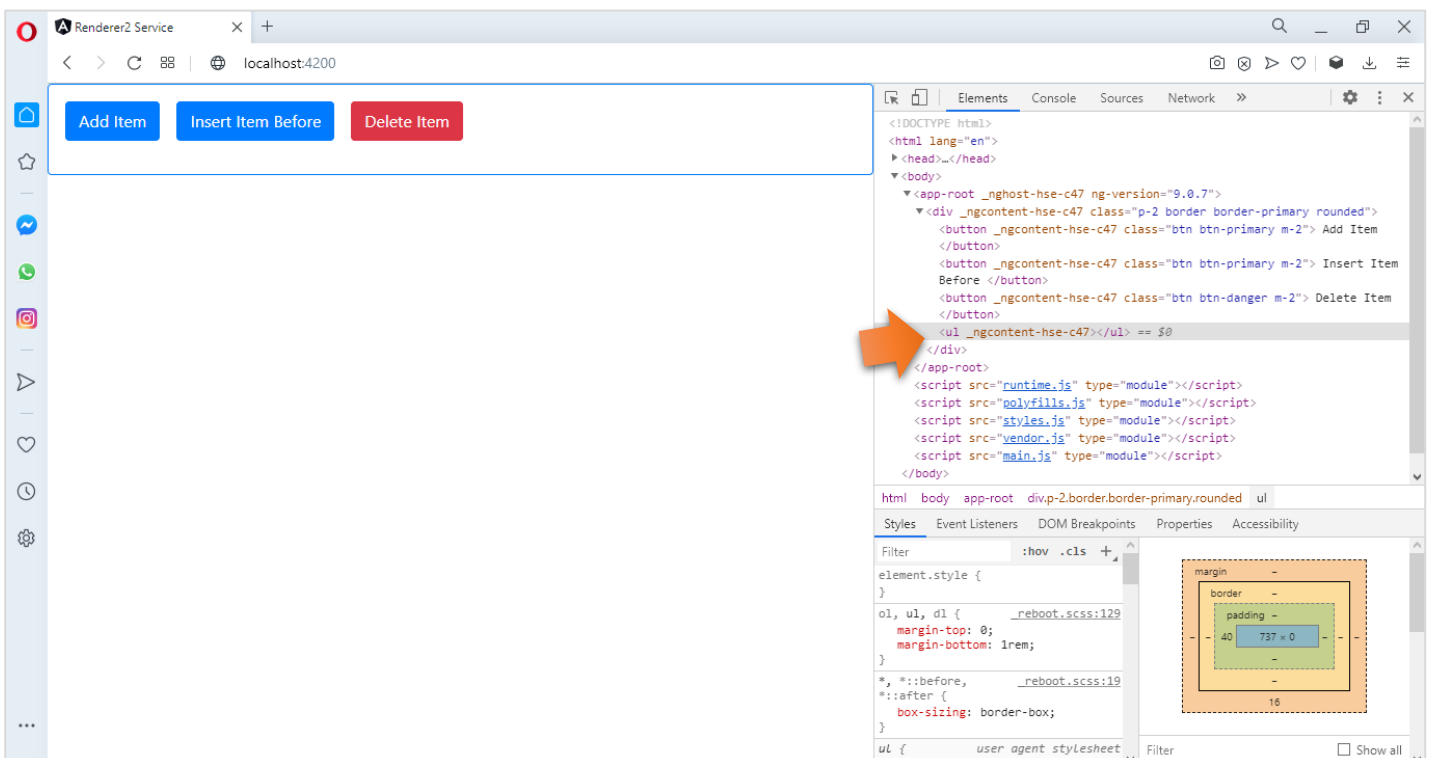
}

insertItemBefor() {
  const newCount = this.count - 1;
  this.count++;
  const el = this.renderer.createElement('li');
  const text = this.renderer.createText(`Item NO.${this.count}`);
  const refChild = this.element.nativeElement.children;
  const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
  this.renderer.appendChild(el, text);
  this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
}

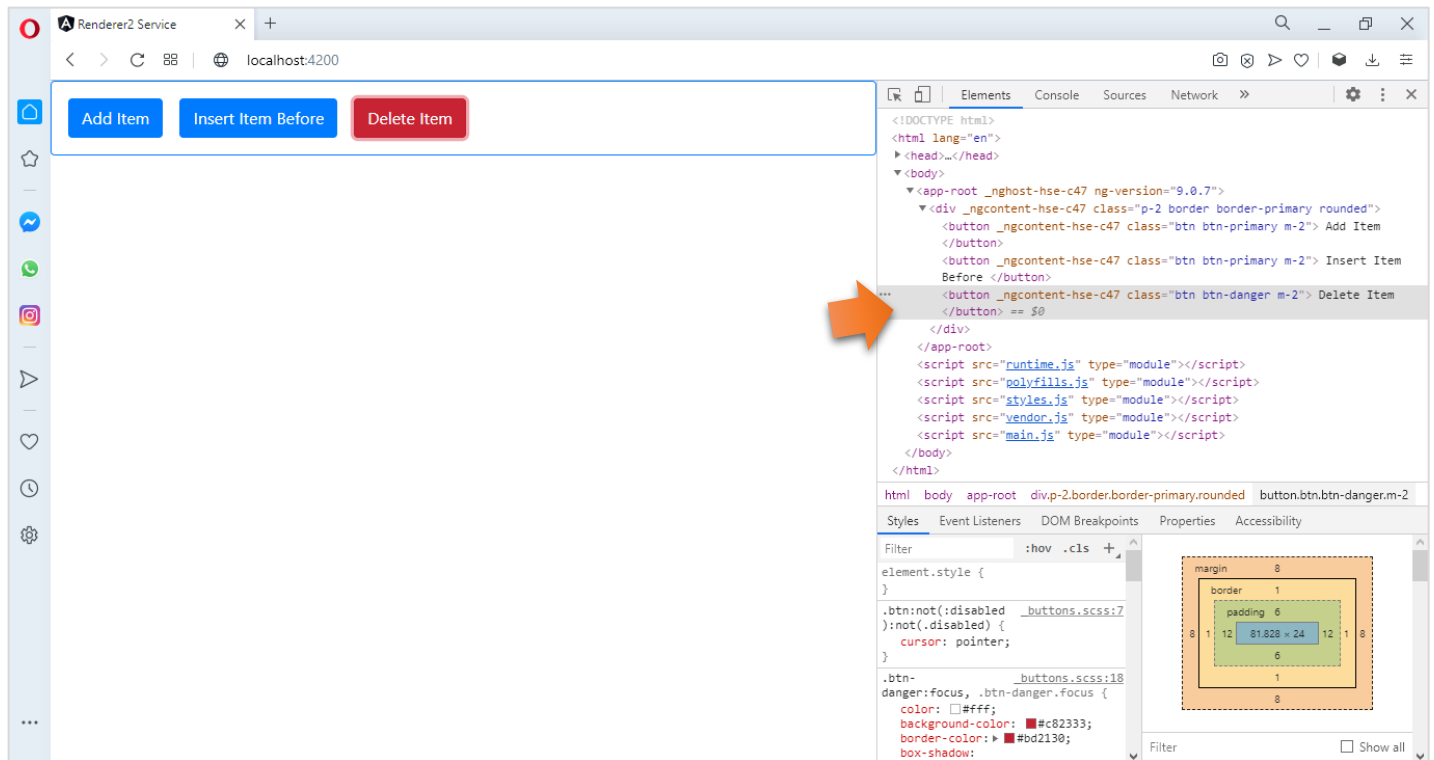
deleteItem() {
  const el = this.element.nativeElement;
  this.count = 0;
  if (el) {
    this.renderer.removeChild(el.parentNode, el);
  }
}
}

```

ما يهمنا هو محتوى الدالة deleteItem حيث عرفنا ثابت يشير إلى العنصر ul وبما أننا سوف نحذف القائمة ul وجميعها لذلك نقوم بتصفير العداد (المتغير count) ومن ثم نضع شرط لتأكد من أن هذا العنصر موجود لأن هنالك احتمال أن يقوم المستخدم بالضغط على الزر ويتم حذف العنصر ul ومن ثم يقوم بالضغط مرة أخرى ففي هذه الحالة سوف تظهر رسالة خطأ لأن العنصر المراد حذفه غير موجود، وآخر سطر استخدمنا الدالة removeChild لحذف العنصر ومررنا لها بارامترين الأول هو الأب للعنصر ul وفي حالتنا هذه هو العنصر div حيث استخدمنا الخاصية الجاهزة parentNode التي أشرنا لها سابقاً لكي نصل إلى العنصر الأب، أما البارامتر الثاني فهو العنصر نفسه المراد حذفه.

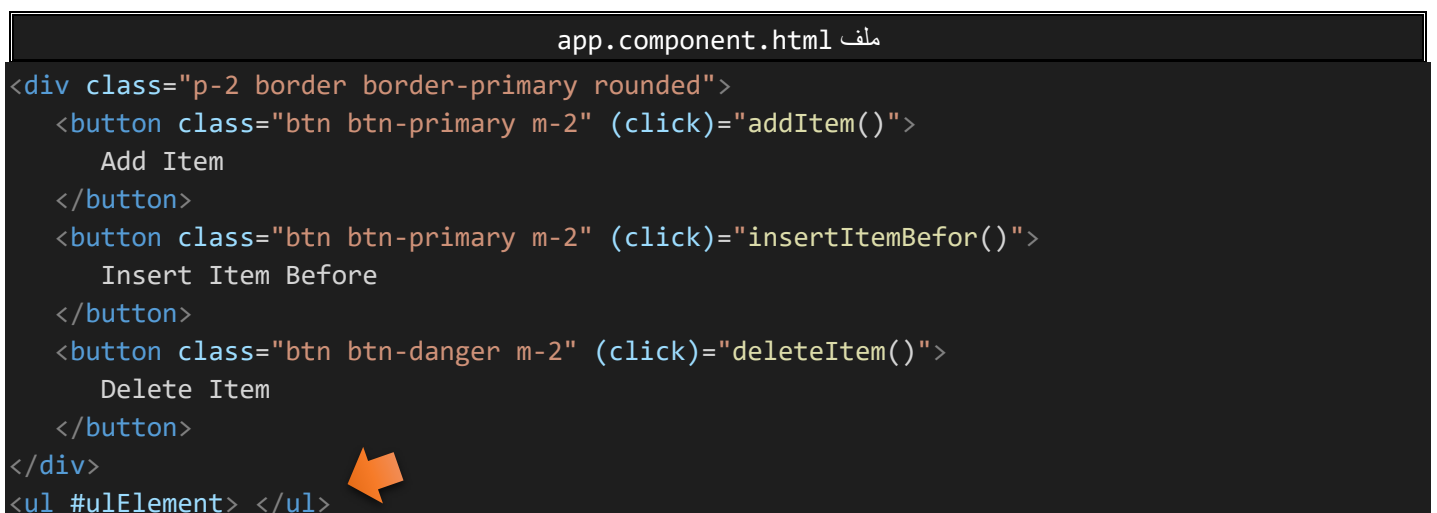


نلاحظ وجود العنصر ul في DOM الآن لو قمنا بالضغط على الزر Delete Item فسوف يقوم بحذف العنصر من DOM،  
كالتالي:



نلاحظ ان العنصر غير موجود في DOM بعدما قمنا بالضغط على الزر، مع العلم ان هذه الطريقة سوف تحذف العنصر  
وجميع العناصر الأبناء التي يحتويها ان وجدت، ولو حاولنا إضافة أي عنصر بالضغط على الزرين Add Item او الآخر فلن  
يقوما بأي شيء وحقيقة هذا هو المتوقع لأننا انشأنا العناصر li بشكل ديناميكي وعملنا لها append للعنصر ul وبهذه الطريقة  
سوف تظهر للمستخدم ولكن عند حذف ul فلن يتم إضافة أي عنصر li.

ولكن لنفرض ان هذا العنصر ul ليس له أب، كالتالي:



نلاحظ ان العنصر ul الآن اصبح خارج div، وفي الحقيقة في حال angular ليس هنالك عنصر ليس له أب، فمثل حالتنا هذه  
فالأب هو selector المستضيف لهذا العنصر، وفي حالتنا هذه هو <app-root></app-root> لذلك لو رجعنا إلى المتصفح

وقمنا بالتجربة فإن الكود السابق في الدالة deleteItem سوف يعمل بلا أي مشاكل، والفرق الوحيد هنا ان الأب بدلاً من ان يكون div أصبح app-root.

وهناك طريقة أخرى للوصول إلى selector المستضيف لهذه العناصر بشكل مباشر، وذلك عن طريق عمل inject في service ذات الاسم ElementRef في منغير معين عن طريق constructor، وبذلك أصبح هذا المتغير يشير بشكل مباشر إلى selector المستضيف لهذه العناصر والذي هو في مثالنا هذا app-root، كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  addItem() {
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.appendChild(this.element.nativeElement, el);
  }

  insertItemBefor() {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.parentElement.nativeElement;
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }
}
```

ولو قمنا بتشغيل التطبيق فسوف تظهر لنا نفس النتيجة.

ولكن ماذا لو اردنا ان نحذف العناصر الأبناء li داخل العنصر الأب ul، فهذه الحالة لا بد ان نستخدم حلقة loop لأن الأمر removeChild لا يحذف مجموعة عناصر دفعة واحدة وانما يحذف عنصر واحد بكل مرة

قد تتساءل عزيزي المتعلم، انني قلت لك سابقاً انه عند الضغط على الزر سوف يقوم بحذف العنصر ul وجميع العناصر التي يحتويها ان وجدت، وبنفس الوقت قلت هنا ان هذه الدالة تقوم بحذف عنصر واحد فقط!! ولتوضيح الدالة removeChild تحذف عنصر واحد وهو في مثالنا السابق ul فهي بهذه الحالة تتعامل مع عنصر واحد ولا تهتم بما يحتويه من تفرعات وعناصر، وبشكل تلقائي فإن المتصفح عند حذف أي عنصر فإنه سوف يقوم بحذف أي شيء يحتويه، اما لو كان لدينا مثلاً اثنان ul او اكثر فإنه لن يقوم بحذفه مرة واحدة وانما في كل مرة يحذف ul لذلك نحتاج إلى حلقة Loop للمرور على جميع العناصر ومن ثم حذفها جميعاً وهذا ما سوف نفعله مع مجموعة العناصر li

ولتوضيح لنقوم بإعطاء مثال، حيث سوف نضيف زر يقوم بحذف جميع عناصر li، وليكن اسمه Delete All Items ويُنفذ دالة وليكن اسمها deleteAllItems، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
</div>
<ul #ulElement> </ul>
```

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }
```

```

addItem() {
  this.count++;
  const el = this.renderer.createElement('li');
  const text = this.renderer.createText(`Item NO.${this.count}`);
  const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
  this.renderer.appendChild(el, text);
  this.renderer.appendChild(this.element.nativeElement, el);
}

insertItemBefor() {
  const newCount = this.count - 1;
  this.count++;
  const el = this.renderer.createElement('li');
  const text = this.renderer.createText(`Item NO.${this.count}`);
  const refChild = this.element.nativeElement.children;
  const textButton = this.renderer.createText(`Delete Item NO.${this.count}`);
  this.renderer.appendChild(el, text);
  this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
}

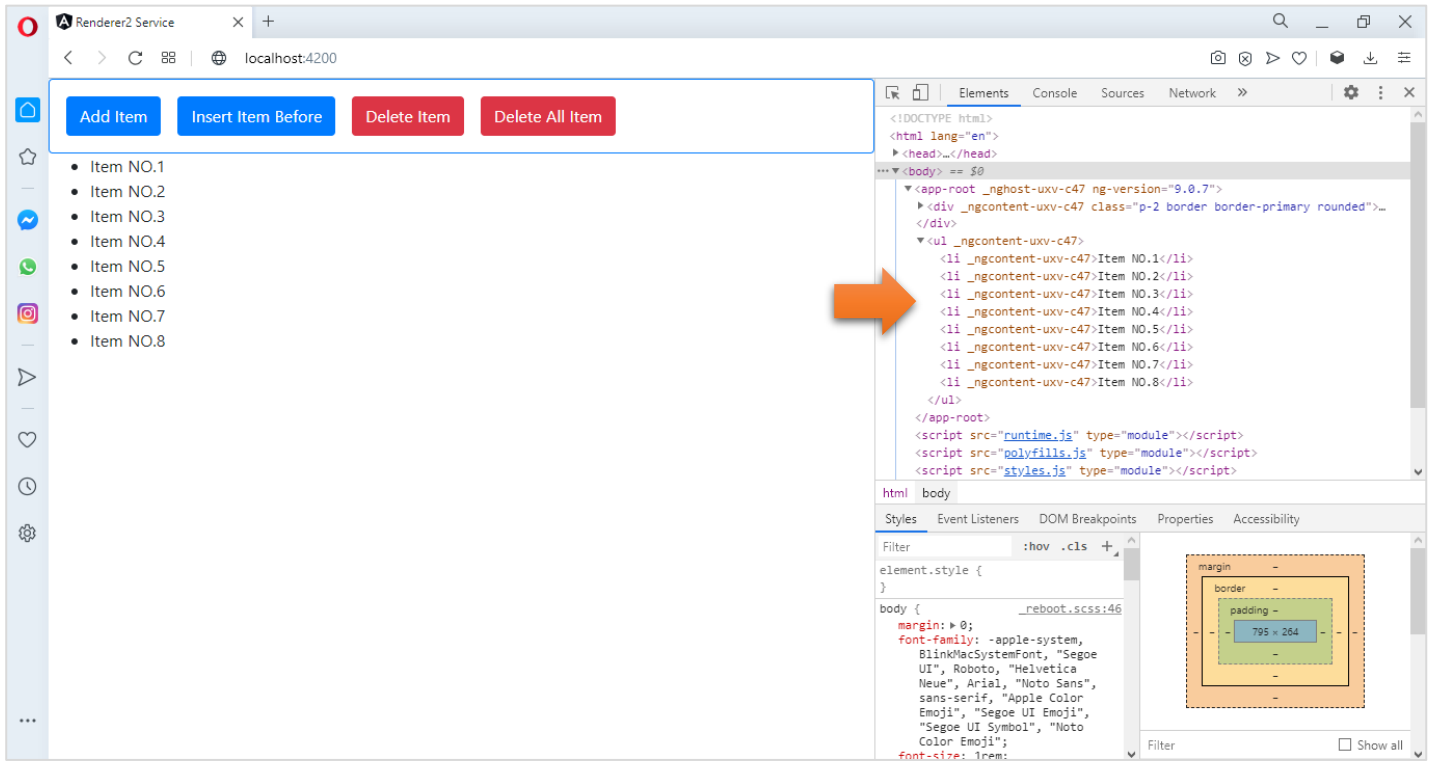
deleteItem() {
  const el = this.element.nativeElement;
  const parent = this.parentElement.nativeElement;
  this.count = 0;
  if (el) {
    this.renderer.removeChild(parent, el);
  }
}

deleteAllItems() {
  const el = this.element.nativeElement;
  this.count = 0;
  while (el.firstChild) {
    this.renderer.removeChild(el, el.lastChild);
  }
}
}

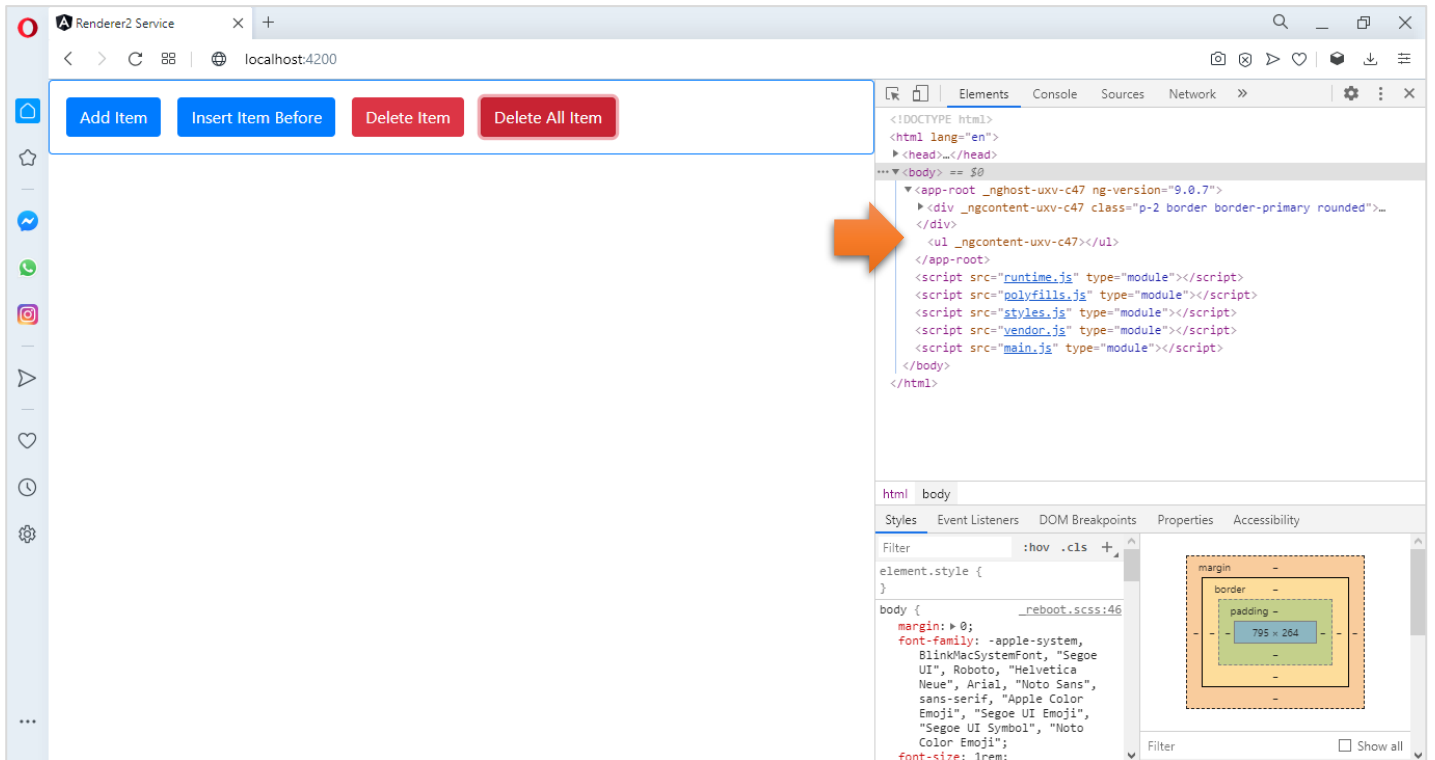
```



نلاحظ هنا قمنا بعمل حلقة Loop بحيث انه كلما كان هنالك عنصر أولي قم بعمل Loop وينفذ الدالة التي في كل مرة تقوم بحذف آخر عنصر إلى أن تصل في النهاية ان يكون هنالك عنصر واحد وهو الأولي والأخير بنفس الوقت وسوف يقوم بحذفه ايضاً، اما النتيجة في المتصفح فتكون كالتالي:



في البداية نقوم بإضافة بعض العناصر كما هو موجود بالصورة السابقة ونلاحظ أيضاً العناصر موجودة في DOM، ومن ثم نقوم بالضغط على زر Delete All Items، وسوف يتم حذف جميع العناصر كالتالي:



نلاحظ العناصر تم حذفها جميعاً، والعنصر الأب aa إلى الآن موجود وبذلك نستطيع إضافة عناصر أخرى له في حال ضغطنا على أزارا إضافة العناصر.

#### 4.8.4. parentNode():

في المثال السابق استطعنا الوصول إلى العنصر الأب باستخدام ElementRef، ولكن نستطيع استخدام دالة جاهزة هي parentNode ونمرر لها العنصر الذي نريد ان نصل للأب الخاص به، كالتالي:

جزء من ملف app.component.ts

```
deleteItem() {  
  const el = this.element.nativeElement;  
  const parent = this.renderer.parentNode(el);  
  this.count = 0;  
  if (el) {  
    this.renderer.removeChild(parent, el);  
  }  
}
```

والنتيجة هي نفسها.

#### 5.8.4. nextSibling():

كما انه يمكننا الوصول إلى العنصر الأب فأيضاً نستطيع الوصول إلى العنصر الأخ، لذلك لنقم بإنشاء زر جديد لقراءة العنصر الأخ لأول عنصر li في القائمة ul ويقوم بعرضه في رسالة alert، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">  
  <button class="btn btn-primary m-2" (click)="addItem()">  
    Add Item  
  </button>  
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">  
    Insert Item Before  
  </button>  
  <button class="btn btn-danger m-2" (click)="deleteItem()">  
    Delete Item  
  </button>  
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">  
    Delete All Item  
  </button>  
  <button class="btn btn-info m-2" (click)="selectSibling()">  
    Select Sibling  
  </button>  
</div>  
<ul #ulElement> </ul>
```

اما محتوى الدالة، فتكون كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
})
```



```

}))

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  addItem() {
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createTextNode(`Item NO.${this.count}`);
    const textButton = this.renderer.createTextNode(`Delete Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.appendChild(this.element.nativeElement, el);
  }

  insertItemBefor() {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createTextNode(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    const textButton = this.renderer.createTextNode(`Delete Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
  }

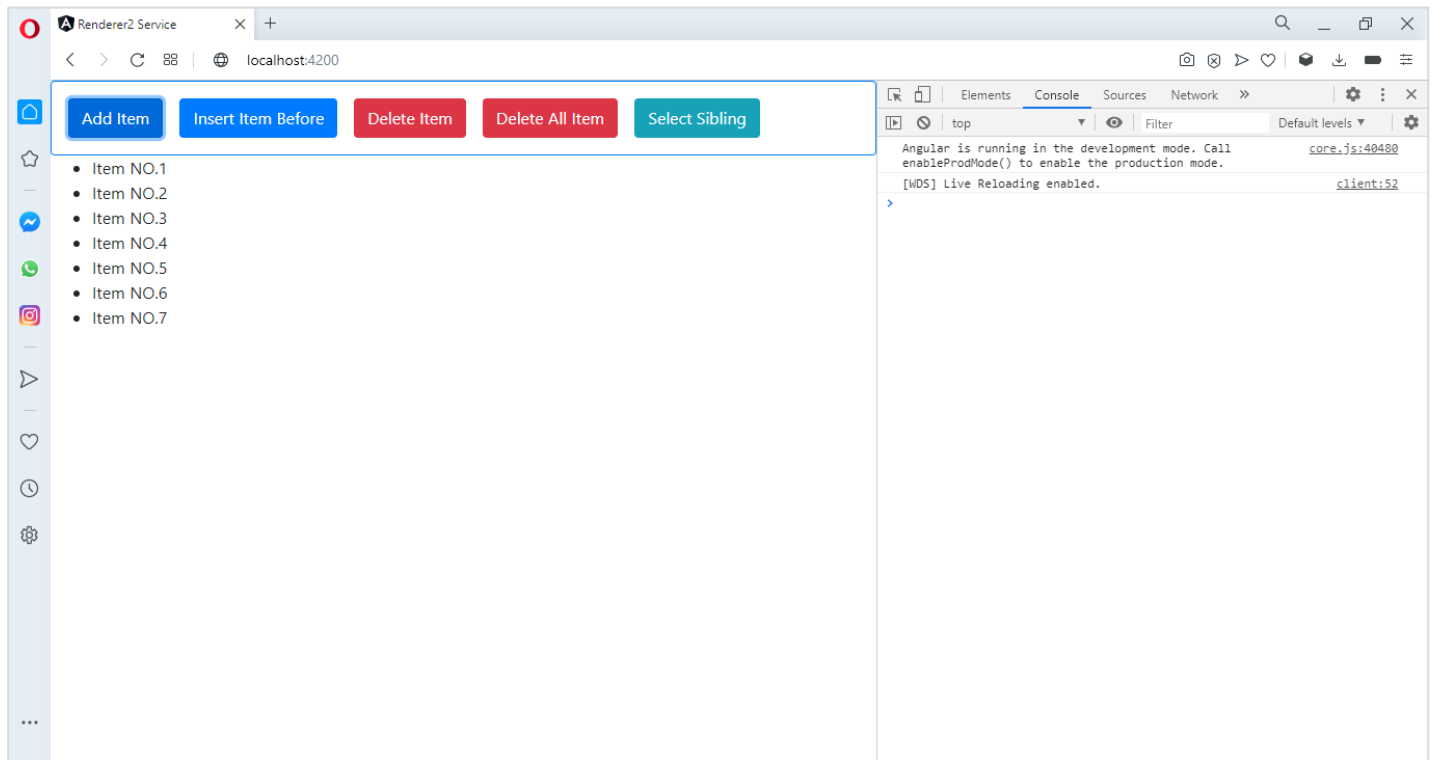
  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }

  deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
      this.renderer.removeChild(el, el.lastChild);
    }
  }

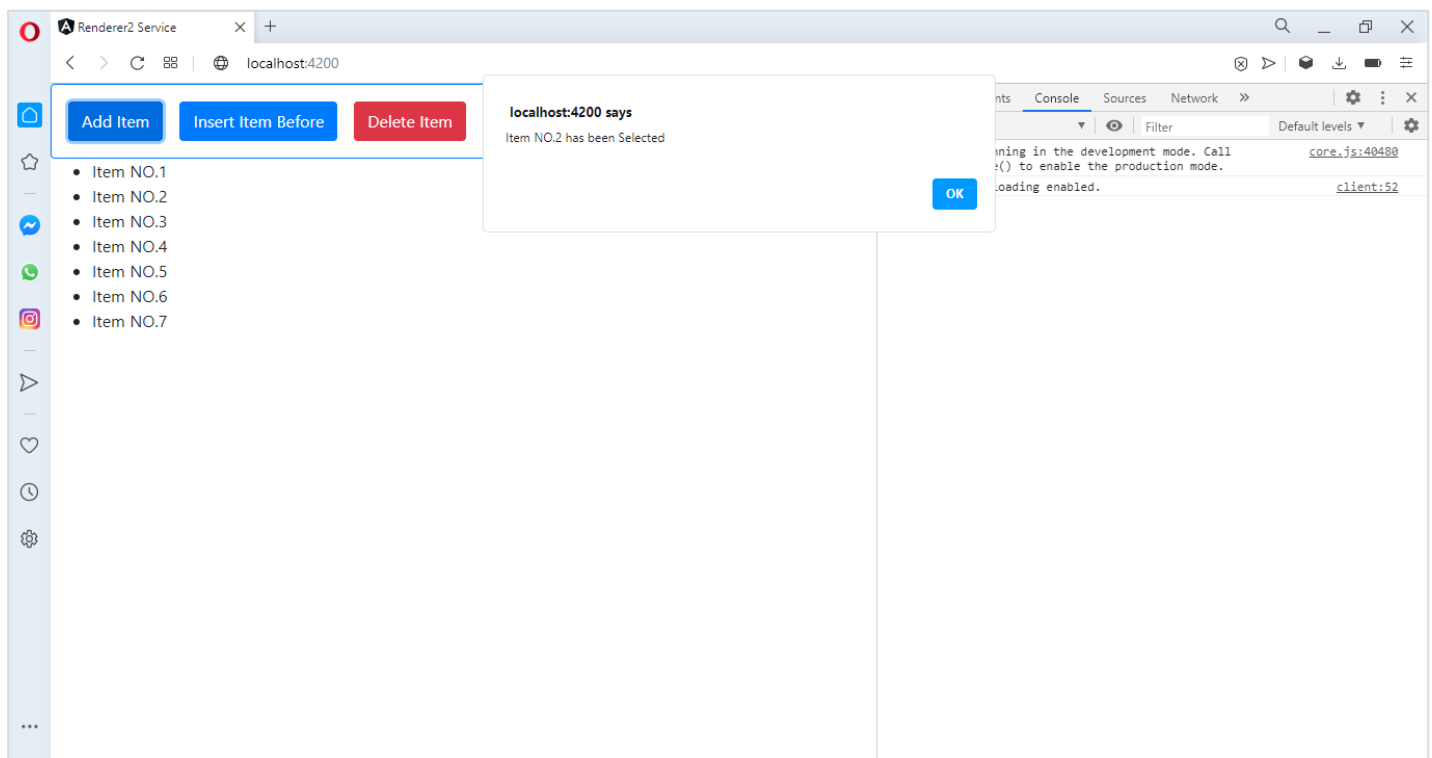
  selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
  }
}

```

والنتيجة في المتصفح تكون كالآتي:



الآن لنضغط على زر Select Sibling ولنرى النتيجة:



نلاحظ ظهرت لنا رسالة تنبيه تخبرنا انه تم عرض العنصر رقم 2 وهو العنصر الأخ لي العنصر رقم 1.

#### 6.8.4. :addClass()

هذه الدالة تسمح بإضافة كلاسات CSS، ولكن يُعيب هذه الدالة هو انها لا تسمح بإضافة إلا كلاس واحد لكل امر ولو احتجت كلاس آخر فيجب كتابته في أمر آخر عن طريق استخدام هذه الدالة مرة، ولتوضيح سوف نقوم بتعديل المثال السابق وذلك بإضافة بعض كلاسات مكتبة bootstrap عند انشاء li في الدالة addItem والدالة insertItemBefore، كالتالي:

```

import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  addItem() {
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.appendChild(this.element.nativeElement, el);
  }

  insertItemBefore() {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }

  deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
      this.renderer.removeChild(el, el.lastChild);
    }
  }
}

```

```

selectSibling() {
  const el = this.element.nativeElement;
  const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
  alert(`${sibling.textContent} has been Selected`);
}
}

```

نلاحظ اننا احتجنا امرين في كل مرة لكي نضيف كلاسين، وهناك ملاحظة مهمة كما نلاحظ ان الدالتين الخاصتين بإضافة العناصر فيهما أكثر من سطر برمجي متشابه لذلك منعاً لتكرار سوف نضيف دالة جديدة وليكن اسمها createElement وننقل لها هذه الاسطر المتشابه، ومن ثم نستدعي هذه الدالة في الدالتين السابقتين كالتالي:

ملف app.component.ts

```

import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {

```

```

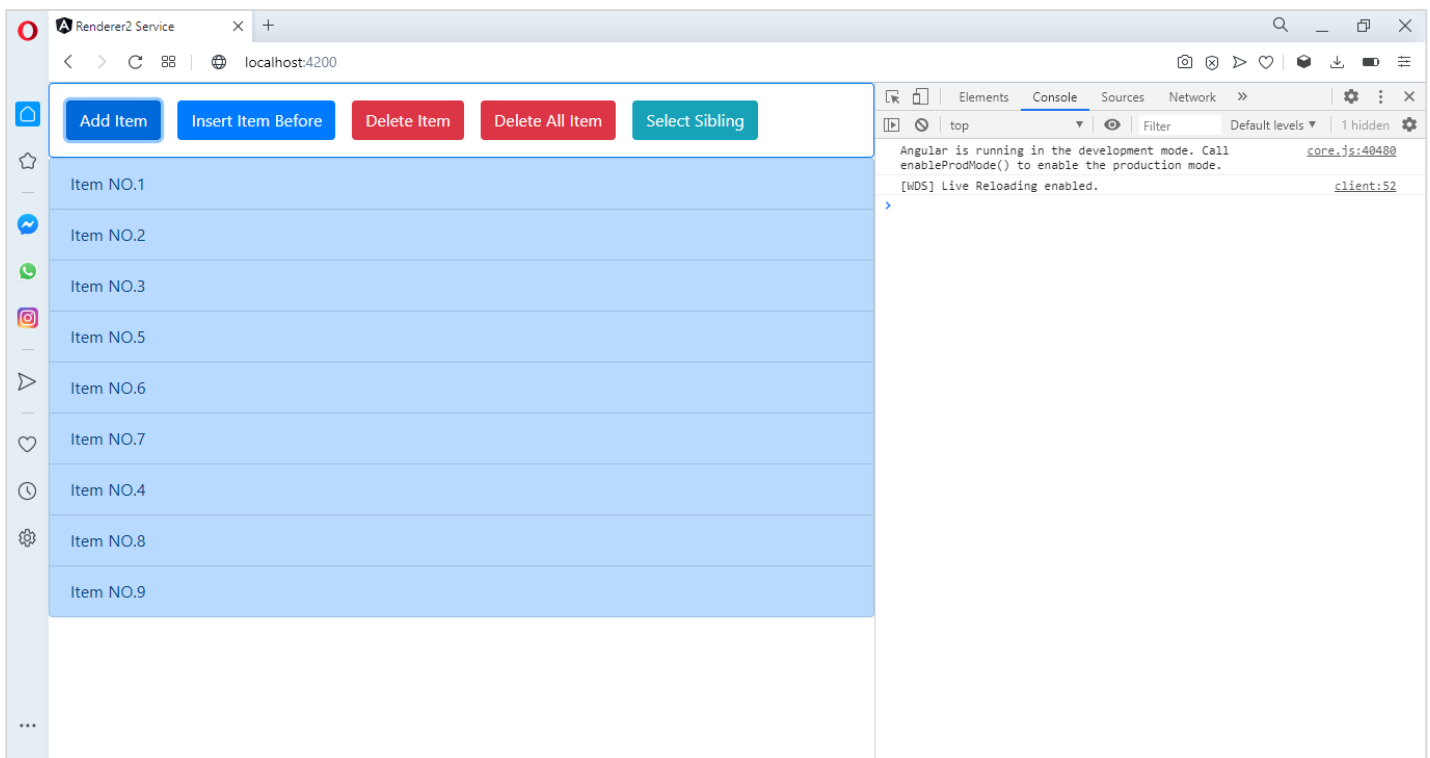
const el = this.element.nativeElement;
const parent = this.renderer.parentNode(el);
this.count = 0;
if (el) {
    this.renderer.removeChild(parent, el);
}
}

deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
        this.renderer.removeChild(el, el.lastChild);
    }
}

selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
}
}

```

اما النتيجة في المتصفح، كالتالي:



#### :removeClass() .7.8.4

ومن اسمها تقوم بحذف كلاس معين من العنصر، ولتوضيح لنقم بإضافة زر جديد بحيث يقوم بإعادة تنسيق العناصر إلى شكلها السابق، وذلك عن طريق حذف هذه الكلاسات، وايضاً هنا نحتاج إلى حلقة Loop، وكنوع من التدريب سوف استخدم

حلقة Loop غير التي استخدمناها removeChild حيث مع هذه الدالة استخدمنا while اما هنا سوف استخدم for-of،  
كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defualtListStyle()">
    the Default List Style
  </button>
</div>
<ul #ulElement class="list-group"> </ul>
```



ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    }
  }
}
```

```

    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

addItem() {
  this.createElement('add');
}

insertItemBefor() {
  this.createElement('insert');
}

deleteItem() {
  const el = this.element.nativeElement;
  const parent = this.renderer.parentNode(el);
  this.count = 0;
  if (el) {
    this.renderer.removeChild(parent, el);
  }
}

deleteAllItems() {
  const el = this.element.nativeElement;
  this.count = 0;
  while (el.firstChild) {
    this.renderer.removeChild(el, el.lastChild);
  }
}

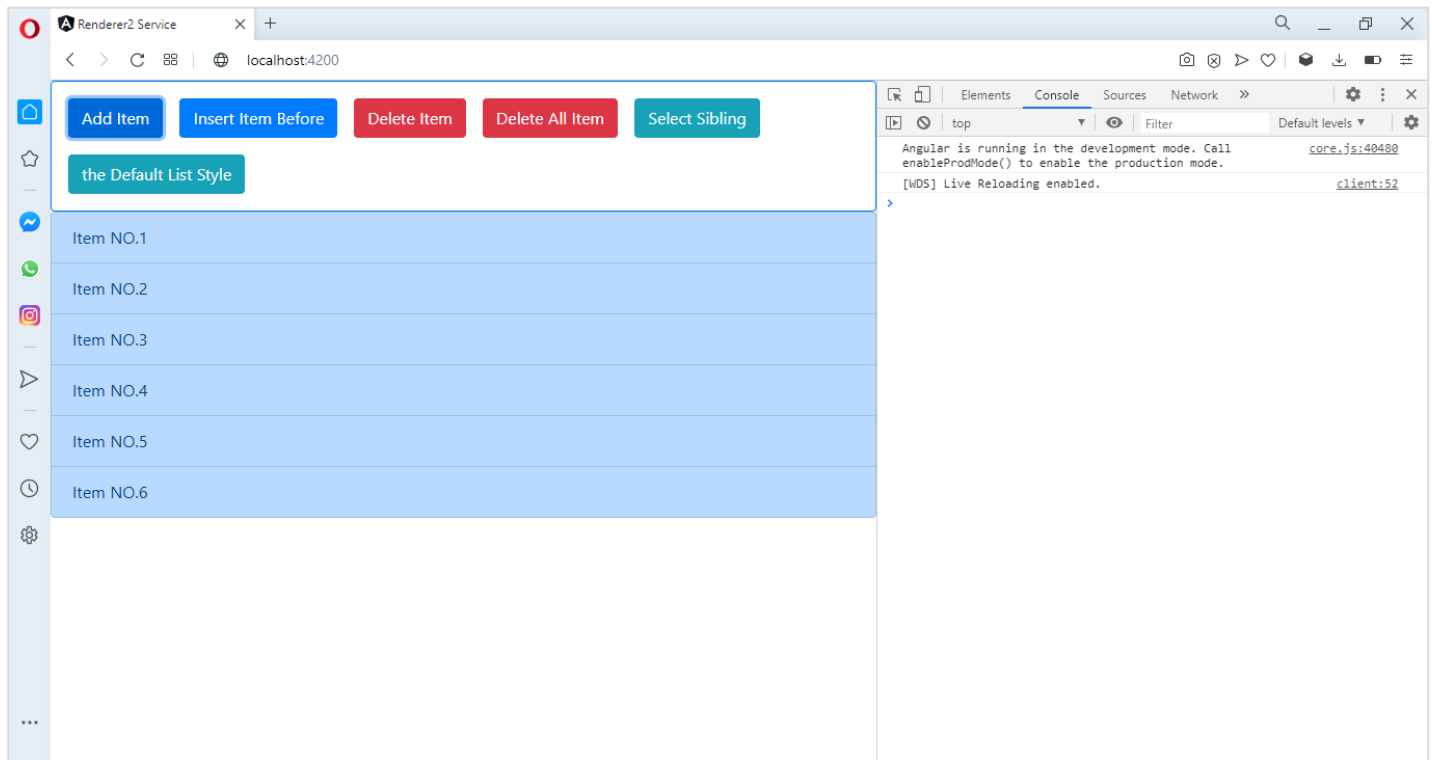
selectSibling() {
  const el = this.element.nativeElement;
  const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
  alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
  const elements: HTMLLIElement[] = this.element.nativeElement.children;
  for (const element of elements) {
    this.renderer.removeClass(element, 'list-group-item');
    this.renderer.removeClass(element, 'list-group-item-primary');
  }
}
}

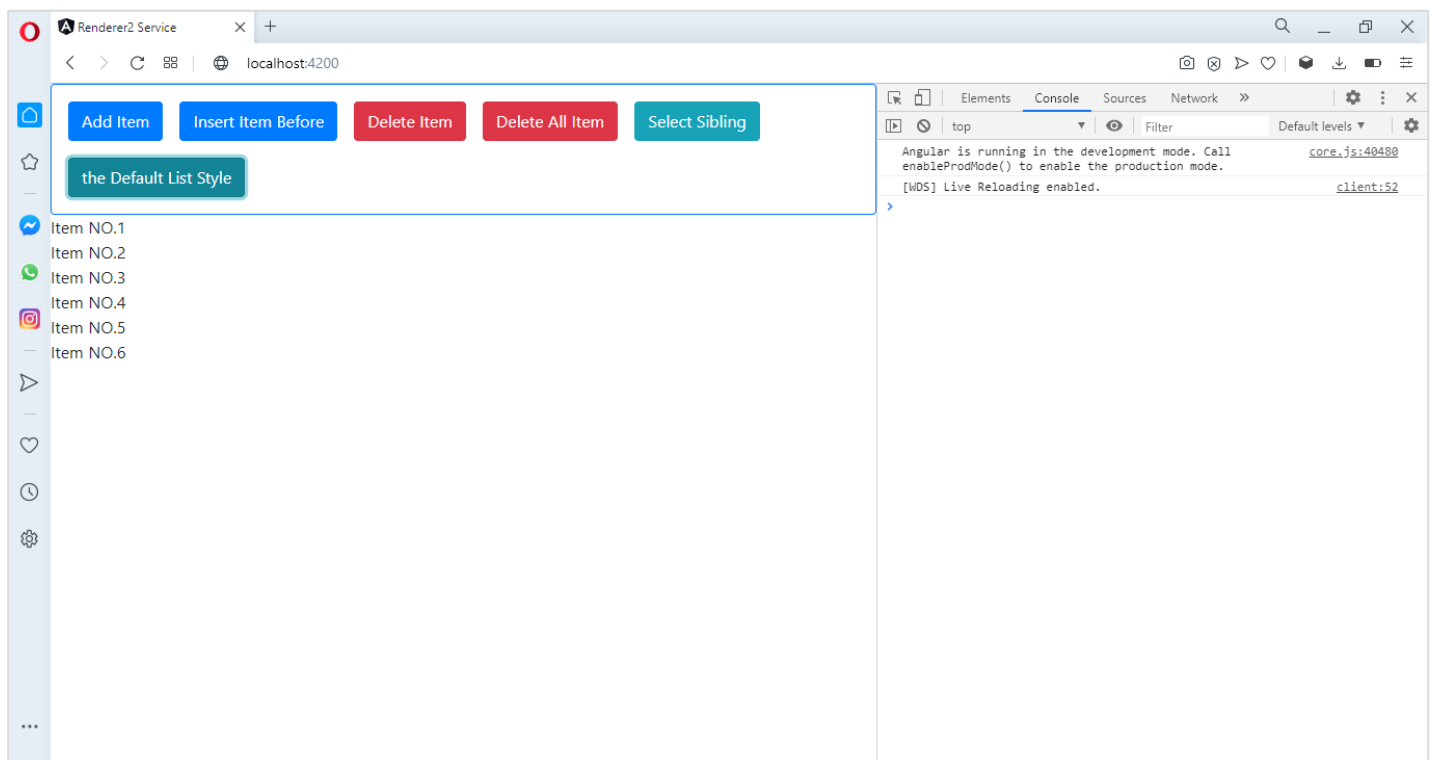
```



اما النتيجة في المتصفح، فتكون كالتالي:



الآن لنقوم بالضغط على زر the Default List Style، ولنرى النتيجة:



نلاحظ تم حذف الكلاسات ورجعت القائمة لوضعها الافتراضي.

#### 8.8.4. :setStyle() / removeStyle()

وعلى نفس السياق في `addClass` و `removeClass` هذين الدالتين أيضاً يقومان بإضافة `style` إلى عنصر محدد أو حذف `style` من هذا العنصر، وهي أيضاً مشابهة من حيث أنه لا بد من إضافة أكثر من أمر في حال أردنا إضافة أكثر من `style` للعنصر، وتستقبل الدالة `setStyle` ثلاث عناصر ضرورية هم العنصر المراد إضافة `style` إليه وخاصية `css` وأخيراً قيمة هذه الخاصية،



اما removeStyle فتستقبل بارامترين ضروريان هما العنصر وخاصية css فقط، وسوف نقوم بإضافة margin لي العناصر li، وبنفس الوقت سوف نضيف الدالة الخاصة بحذف هذا style، كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  @ViewChild('ulElement') private element: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }
}
```

```

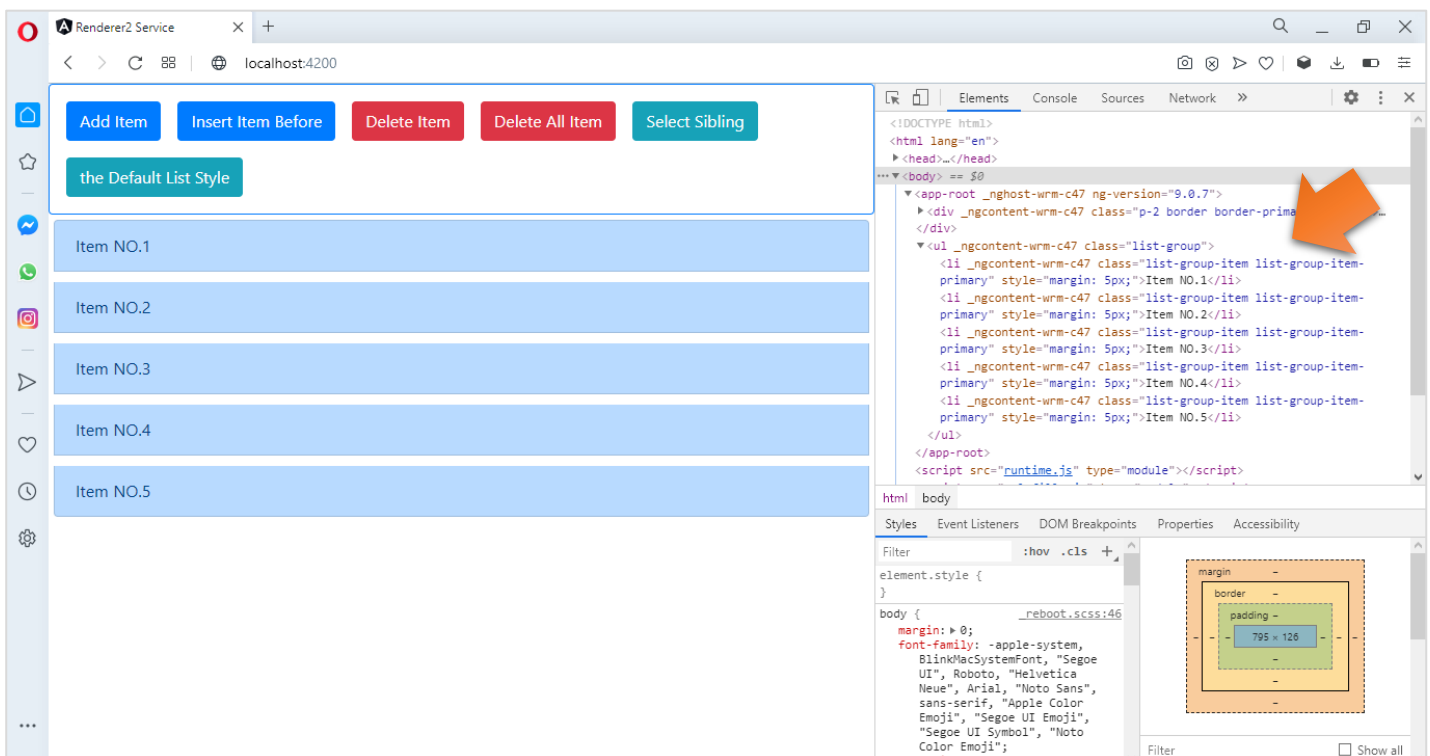
deleteAllItems() {
  const el = this.element.nativeElement;
  this.count = 0;
  while (el.firstChild) {
    this.renderer.removeChild(el, el.lastChild);
  }
}

selectSibling() {
  const el = this.element.nativeElement;
  const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
  alert(`${sibling.textContent} has been Selected`);
}

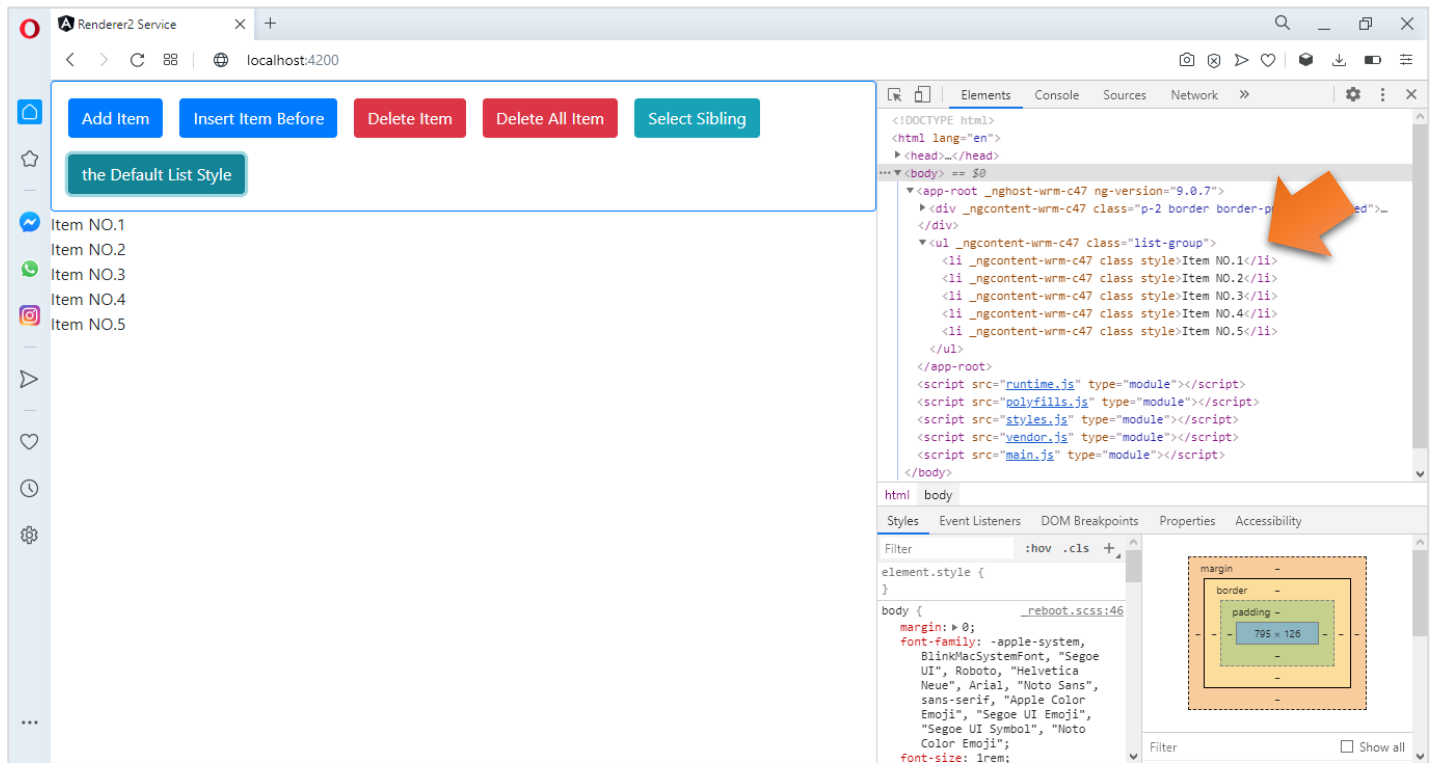
defaultListStyle() {
  const elements: HTMLLIElement[] = this.element.nativeElement.children;
  for (const element of elements) {
    this.renderer.removeClass(element, 'list-group-item');
    this.renderer.removeClass(element, 'list-group-item-primary');
    this.renderer.removeStyle(element, 'margin');
  }
}
}

```

اما النتيجة، فتكون كالتالي:



نلاحظ العناصر في يمين الشاشة وقد أُضيفت لها خاصية margin، والآن لنضغط على زر Default List Style، كالتالي:



وايضاً نلاحظ بعدما قمنا بالضغط على الزر تم حذف جميع style والكلاسات.

#### 9.8.4. :setAttribute() / removeAttribute()

قبل ان نبين وظيفة هاتين الدالتين وكيفية استخدامهما لابد ان نوضح ماهي السمة Attribute والخاصية Property في عناصر HTML، مع العلم انني قمت بتوضيح الفرق في الجزء الخاص بي Property Binding وهنا سوف افصل فيها قليلاً لأن وجدت فيها لبس كثير لدى المبتدئين في تطوير الويب.

Attribute هي السمات التي يحتويها عنصر HTML قبل عرض هذا العنصر في DOM، ولكن عندما يقوم المتصفح بقراءة الصفحة الخاصة بك عزيزي المتعلم فإنه سوف يقوم بتحويل كل عنصر إلى كائن Object، ومن بديهيات التعامل مع الكائنات في Javascript ان كل كائن يحتوي على خصائص Property ودوال (وظائف) methods، لذلك سوف يقوم المتصفح بعمل نسخة من هذه السمات Attribute الموجودة في العنصر والتعامل معها على انها خصائص بحيث تكون هذه الخصائص مستقلة تماماً عن السمات، فمثلاً لو قمنا بالتعامل مع السمة value في العنصر  ولنفرض اننا اسندنا قيمة لهذه السمة ولتكن Faisal بحيث يكون شكل العنصر كالتالي  فإنه عند بداية تشغيل هذه الصفحة فإن المتصفح سوف يقوم بتحويل هذا العنصر إلى كائن من النوع HTMLInputElement ومن ثم يقوم بقراءة هذه السمة value ووضع قيمتها Faisal في خاصية لهذا الكائن تحمل نفس اسم السمة value، وأخيراً تصبح هذه الخاصية Property الجديدة value والتي تحمل القيمة Faisal مستقلة بشكل كامل عن السمة Attribute ذات الاسم value، ولو قام المستخدم على سبيل المثال بتغيير القيمة في مربع الادخال Input فإنه سوف يُغير في قيمة الخاصية Property وليس السمة Attribute التي لاتزال تحتفظ بقيمتها والتي هي Faisal.

لكن هنالك سؤال هل جميع السمات لها خصائص مشابهة لها في DOM في الحقيقة هنالك بعض السمات لا يوجد لها خصائص في DOM ولذلك كنا نستخدم مع هذه السمات attr في Attribute Binding، وهنالك ايضاً بعض الخصائص

DOM ليس لها سمات مثل `textContent`، فمثلاً لا نستطيع كتابة `<div textContent="Hello Word"></div>` وانما نستطيع كتابة `<div>Hello Word</div>` وعندما يتم تحويل هذا العنصر إلى كائن من النوع `HTMLDivElement` في DOM سوف يتم انشاء خاصية جديدة باسم `textContent` ويُسند لها القيمة `Hello Word`.

وبعد هذه المقدمة المطولة لنعطي مثال على استخدام الدالتين `setAttribute` و `removeAttribute`، ولنفرض انه لدينا جدول وهذا الجدول يحتوي على مجموعة من عناصر `td` وهذه العناصر تحتوي على سمة `attribute` وهي `colspan` وهذه السمة ليس لها خاصية `Property` في الكائن الخاص بها في DOM، لذلك في حال اردنا اضافتها ديناميكياً من خلال الكود فلا بد ان نستخدم الدالة `setAttribute` حيث ان هذه الدالة تستقبل ثلاث بارامترات الأول هو العنصر الذي نريد ان نضيف له هذه السمة والثاني هي اسم السمة والثالث هو قيمة هذه السمة، كالتالي:

ملف `app.component.html`

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defualtListStyle()">
    the Default List Style
  </button>
  <button class="btn btn-info m-2" (click)="setAttr()">
    Set colspan Attribute
  </button>
</div>
<ul #ulElement class="list-group"> </ul>
<table>
  <tr>
    <th>Month</th><th>Savings</th>
  </tr>
  <tr>
    <td>January</td><td>$100</td>
  </tr>
  <tr>
    <td>February</td><td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>
```

نلاحظ اضعفنا لآحد عناصر td اسم Template Reference لكي نستطيع الوصول له في ملف class ومن ثم نضعف له السمة colspan، حيث يتم إضافة هذه السمة عن طريق الزر الذي قمت المؤشر عليه بالسهم حيث يُنفذ دالة باسم setAttr، كالتالي:

ملف app.component.ts

```
import { Component, ViewChild, ElementRef, Renderer2 , AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {

  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
  }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
```

```

    this.count = 0;
    if (el) {
        this.renderer.removeChild(parent, el);
    }
}

deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
        this.renderer.removeChild(el, el.lastChild);
    }
}

selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
    const elements: HTMLLIElement[] = this.element.nativeElement.children;
    for (const element of elements) {
        this.renderer.removeClass(element, 'list-group-item');
        this.renderer.removeClass(element, 'list-group-item-primary');
        this.renderer.removeStyle(element, 'margin');
    }
}

setAttr() {
    const td: HTMLTableElement = this.tdColspan.nativeElement;
    this.renderer.setAttribute(td, 'colspan', '2');
    console.log(td);
}
}

```



في البداية استقبلنا Template Reference عن طريق ViewChild ومن ثم طبعت هذا العنصر في console قبل لا نضيف له العنصر لكي نشاهد الفرق قبل إضافة السمة وبعد إضافة السمة وهو محتوى الدالة setAttr حيث قمنا باستخدام setAttribute ومررنا العنصر الذي نريد إضافة له السمة والسمة وهي colspan وأخيراً القيمة وهي اثنان وأخيراً قمت بطباعة العنصر مرة أخرى في console لكي نشاهد الفرق بعد إضافة السمة.

وفي ملف style لتنضيف بعض الحدود borders، كالتالي:

ملف app.component.css

```

table,
th,
td {
    border: 1px solid black;
}

```

والآن لنشاهد النتيجة في المتصفح، كالتالي:

Month	Savings
January	\$100
February	\$80
Sum:	\$180

نلاحظ العنصر تم طباعته في console ولا يحتوي على السمة والجدول ونلاحظ الخلية التي نريد ان نضيف لها السمة، الآن لنضغط على الزر Set colspan Attribute ولنرى النتيجة، كالتالي:

Month	Savings
January	\$100
February	\$80
Sum: \$180	

نلاحظ العنصر في console تمت إضافة السمة colspan له وب نفس الوقت تم تطبيق هذه السمة على الجدول، مع العلم انه لو حاولنا استخدام الدالة setProperty مع السمة colspan فلن يحدث شيء لأن هذه الدالة تتعامل فقط مع خصائص الكائن في DOM، لذلك يجب الانتباه ومعرفة الخصائص من السمات.

اما الدالة removeAttribute فمن اسمها تقوم بحذف السمة من العنصر، لذلك لنقوم بإضافة زر لحذف العنصر بحيث ينفذ دالة ولتكن اسمها removeAttr، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefore()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defaultListStyle()">
    the Default List Style
  </button>
  <button class="btn btn-success m-2" (click)="setAttr()">
    Set colspan Attribute
  </button>
  <button class="btn btn-danger m-2" (click)="removeAttr()">
    Remove colspan Attribute
  </button>
</div>

<ul #ulElement class="list-group"> </ul>

<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>
```

اما ملف class فنقوم بكتابة محتوى الدالة، كالتالي:



```

import { Component, ViewChild, ElementRef, Renderer2, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {
  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
  }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }
}

```

```

deleteAllItems() {
  const el = this.element.nativeElement;
  this.count = 0;
  while (el.firstChild) {
    this.renderer.removeChild(el, el.lastChild);
  }
}

selectSibling() {
  const el = this.element.nativeElement;
  const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
  alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
  const elements: HTMLLIElement[] = this.element.nativeElement.children;
  for (const element of elements) {
    this.renderer.removeClass(element, 'list-group-item');
    this.renderer.removeClass(element, 'list-group-item-primary');
    this.renderer.removeStyle(element, 'margin');
  }
}

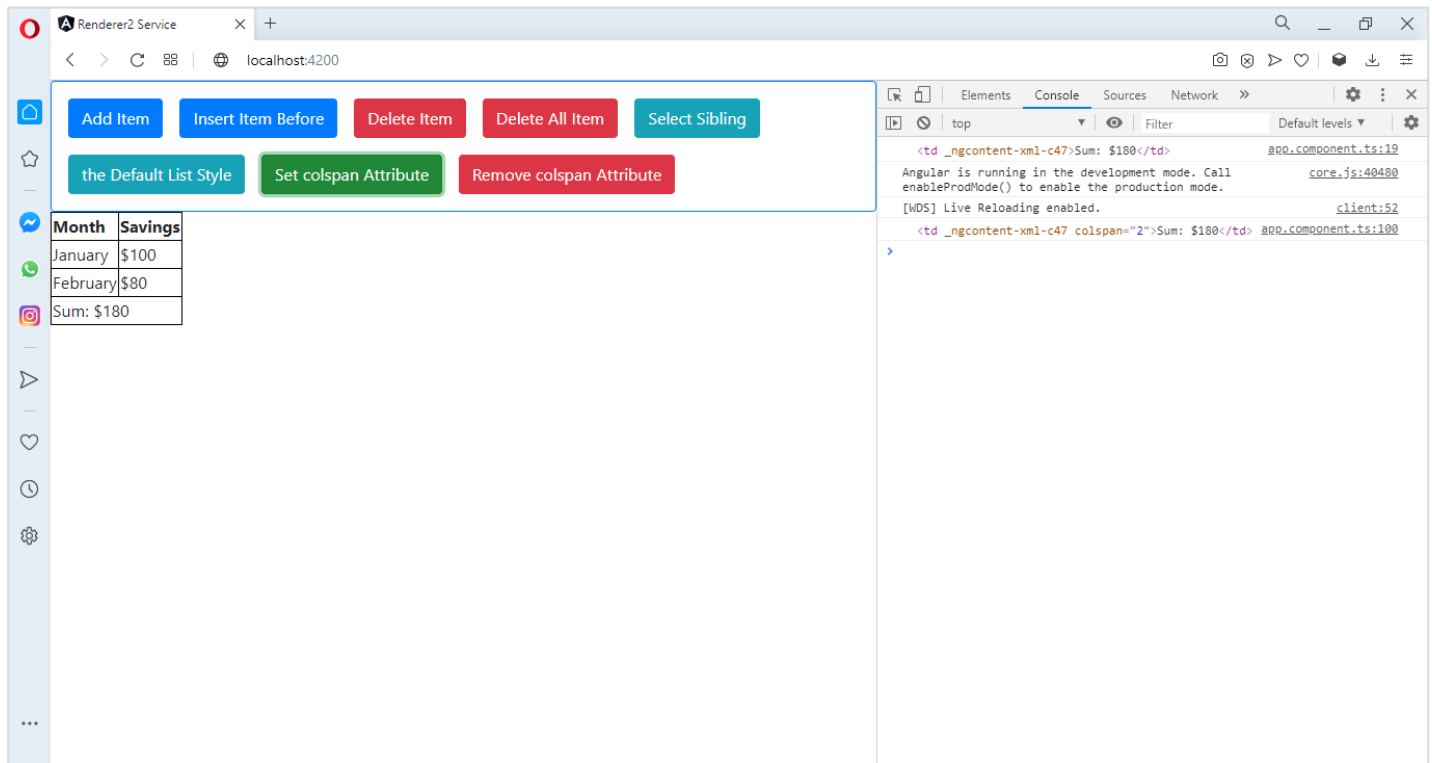
setAttr() {
  const td: HTMLTableElement = this.tdColspan.nativeElement;
  this.renderer.setAttribute(td, 'colspan', '2');
  console.log(td);
}

removeAttr() {
  this.renderer.removeAttribute(this.tdColspan.nativeElement, 'colspan');
  console.log(this.tdColspan.nativeElement);
}
}

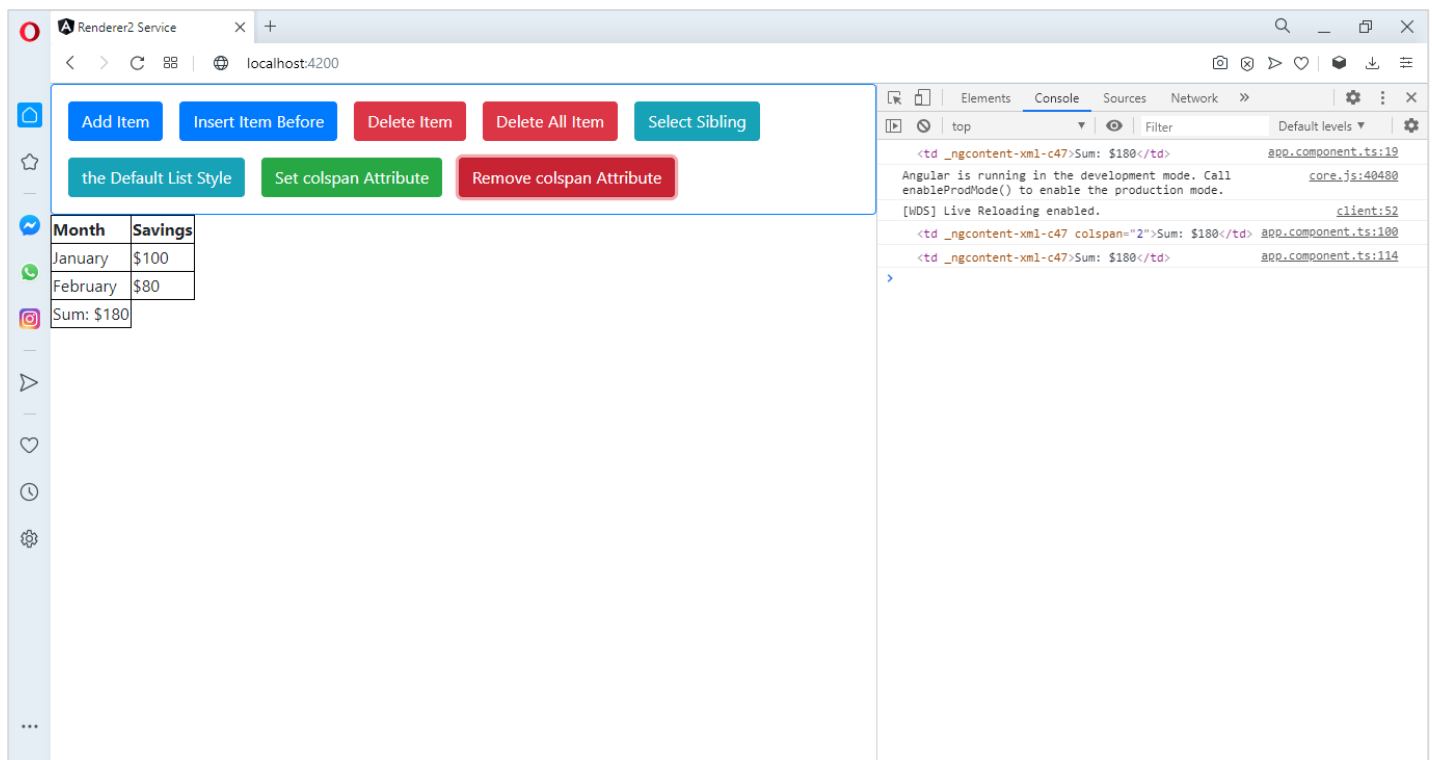
```



الآن لنقم بمشاهدة النتيجة في المتصفح، كالتالي:



نلاحظ قمنا بالضغط على الزر Set colspan Attribute لإضافة السمة colspan إلى العنصر، والآن لنقوم بالضغط على الزر Remove colspan Attribute ولنرى النتيجة:



نلاحظ تم حذف السمة colspan.

#### 10.8.4 .setProperty():

تقوم هذه الدالة بالتعامل مع خصائص العناصر بعد تحويلها إلى كائنات في DOM، ومن الأمثلة الخاصة بـ textContent ولتوضيح لنعطي مثال لنفرض انه لدينا عنصر div ونريد ان نضيف نص لهذا div عن طريق الخاصية textContent، كالتالي:

```

<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defualtListStyle()">
    the Default List Style
  </button>
  <button class="btn btn-success m-2" (click)="setAttr()">
    Set colspan Attribute
  </button>
  <button class="btn btn-danger m-2" (click)="removeAttr()">
    Remove colspan Attribute
  </button>
  <button class="btn btn-success m-2" (click)="setProp()">
    set textConteny Property
  </button>
</div>

<ul #ulElement class="list-group"> </ul>

<div #div></div>

<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>

```

```

import { Component, ViewChild, ElementRef, Renderer2, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {
  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('div') private div: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
    console.log(this.div.nativeElement);
  }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
      this.renderer.removeChild(parent, el);
    }
  }
}

```

```

    }
  }

  deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
      this.renderer.removeChild(el, el.lastChild);
    }
  }

  selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
  }

  defaultListStyle() {
    const elements: HTMLLIElement[] = this.element.nativeElement.children;
    for (const element of elements) {
      this.renderer.removeClass(element, 'list-group-item');
      this.renderer.removeClass(element, 'list-group-item-primary');
      this.renderer.removeStyle(element, 'margin');
    }
  }

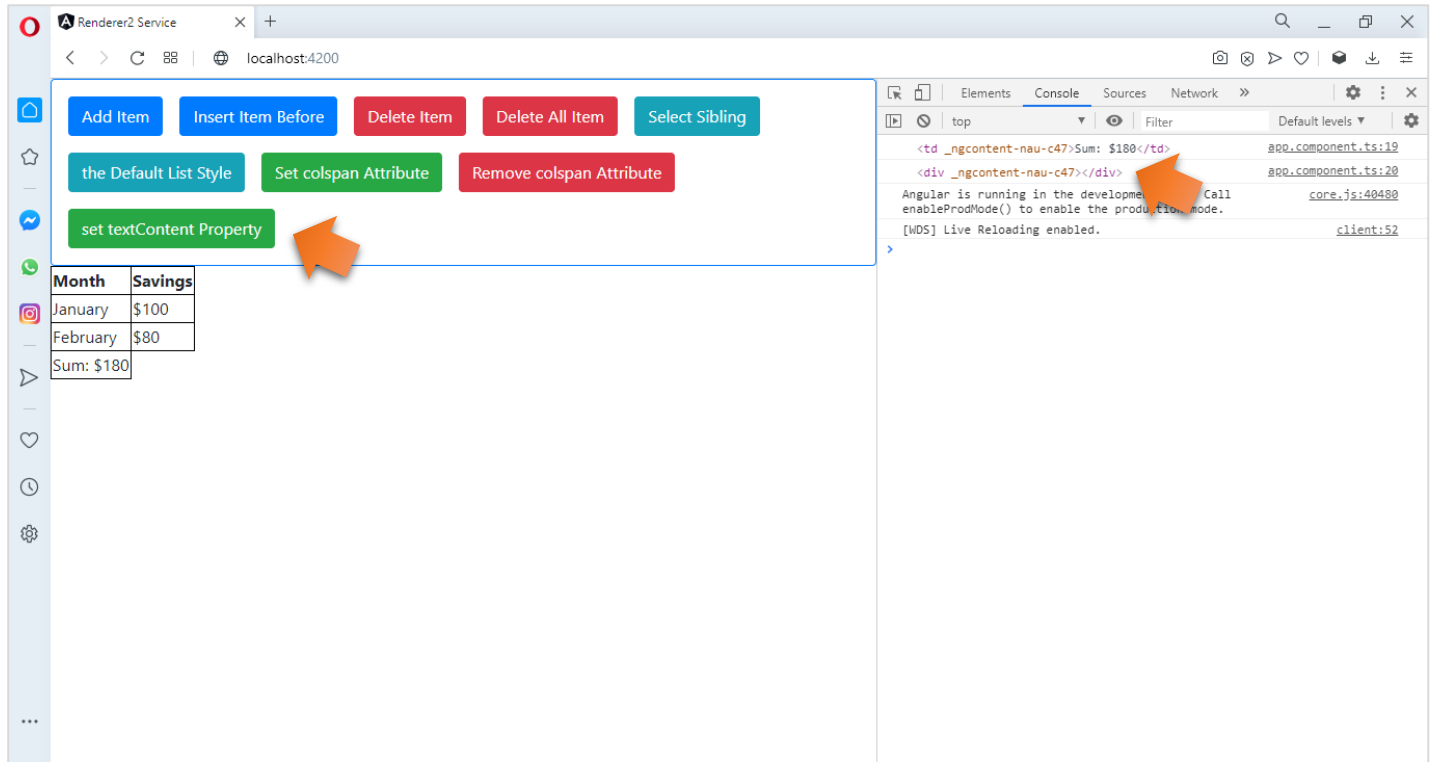
  setAttr() {
    const td: HTMLTableElement = this.tdColspan.nativeElement;
    this.renderer.setAttribute(td, 'colspan', '2');
    console.log(td);
  }

  removeAttr() {
    this.renderer.removeAttribute(this.tdColspan.nativeElement, 'colspan');
    console.log(this.tdColspan.nativeElement);
  }

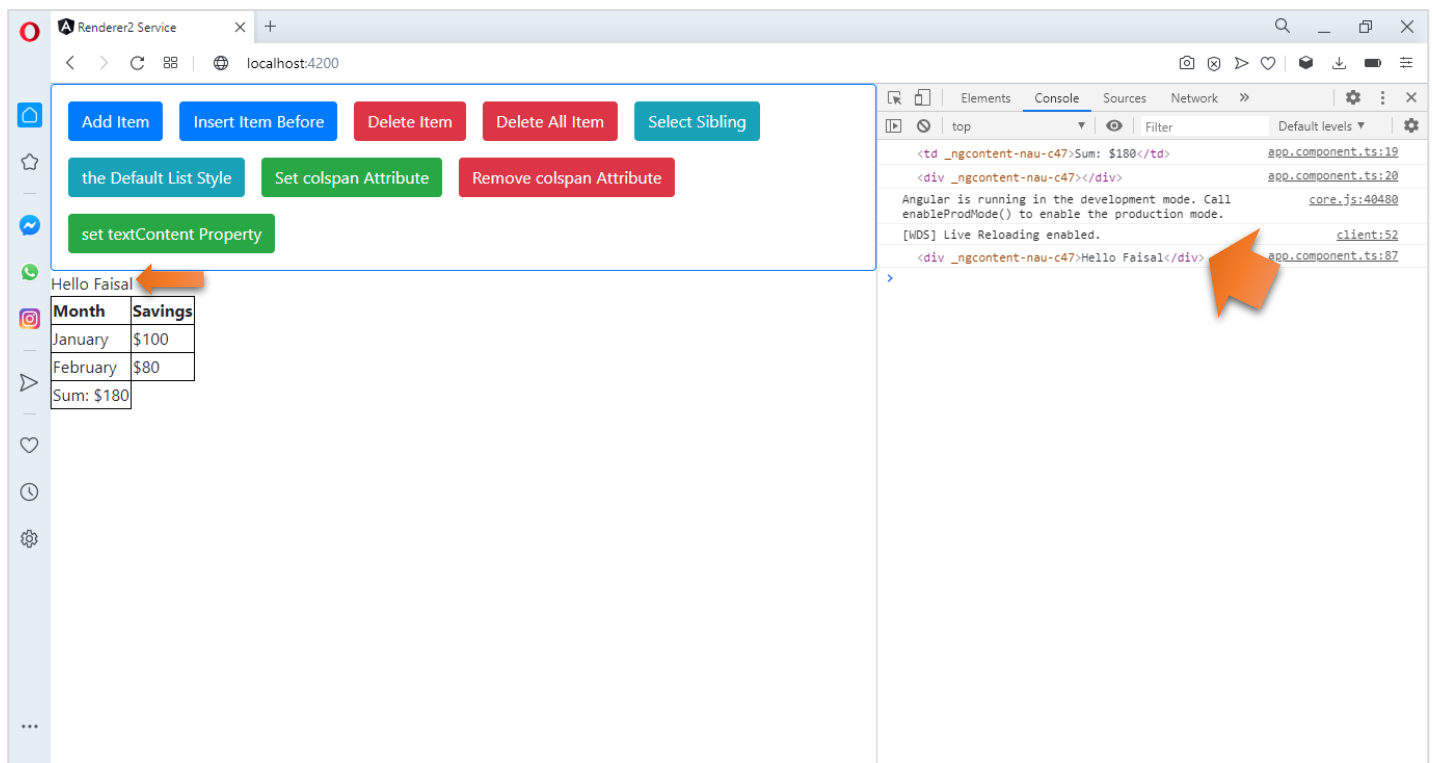
  setProp() {
    const div: HTMLDivElement = this.div.nativeElement;
    this.renderer.setProperty(div, 'textContent', 'Hello Faisal');
    console.log(div);
  }
}

```





الآن لنضغط على الزر ولنرى النتيجة:



نلاحظ تم إضافة الخاصية textContent إلى الكائن HTMLDivElement في DOM، ولو قمنا بإستبدال الدالة setProperty بالدالة setAttribute فلن يحدث شيء لأنه سوف يعامل الخاصية textContent على انها سمة ويُضيفها إلى العنصر ولكن لن يظهر أي شيء في المتصفح لأنه لا يوجد سمة Attribute تحمل هذا الاسم، وتستطيع عزيز المتعلم تجربة هذا الأمر بنفسك.

ملاحظة: لا يوجد دالة لحذف Property بنفس الطريقة في removeAttribute.

ولتوضيح الصورة بشكل اكبر لنقوم بإعطاء مثال آخر، قلنا في المقدمة ان هنالك سمات ليس لها خصائص وهنالك خصائص ليس لها سمات ولكن هنالك سمات لها خصائص مشابهه ليس مرتبطة بها ولكن مشابهه لها ومنها value، لذلك سوف نعطي مثال لنرى في حال استخدمنا كلا الدالتين مع value ماذا سوف يحدث لذلك لنقم بإنشاء مربع نص ولنعطيه Template Reference وبنفس الوقت لنقوم بإنشاء زرین واحد ينفذ الدالة setProperty والثاني يُنفذ الدالة setAttribute، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defualtListStyle()">
    the Default List Style
  </button>
  <button class="btn btn-success m-2" (click)="setAttr()">
    Set colspan Attribute
  </button>
  <button class="btn btn-danger m-2" (click)="removeAttr()">
    Remove colspan Attribute
  </button>
  <button class="btn btn-success m-2" (click)="setProp()">
    set textContent Property
  </button>
  <button class="btn btn-outline-dark m-2" (click)="setValueProp()">
    set Value Property
  </button>
  <button class="btn btn-outline-dark m-2" (click)="setValueAttr()">
    Set Value Attribute
  </button>
</div>
<ul #ulElement class="list-group"> </ul>
<hr />
<div #div></div>
<hr />
<input #input type="text" value="Faisal" />
<hr />
<table>
  <tr>
    <th>Month</th>
```



```

    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>

```

والآن لنقوم بكتابة محتوى الدالتين في ملف class لهذا component، كالتالي:

ملف app.component.ts

```

import { Component, ViewChild, ElementRef, Renderer2, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {
  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('div') private div: ElementRef;
  @ViewChild('input') private input: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
    console.log(this.div.nativeElement);
    console.log(this.input.nativeElement);
  }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO. ${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {

```

```

        this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
        this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
}

addItem() {
    this.createElement('add');
}

insertItemBefor() {
    this.createElement('insert');
}

deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
        this.renderer.removeChild(parent, el);
    }
}

deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
        this.renderer.removeChild(el, el.lastChild);
    }
}

selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
    const elements: HTMLLIElement[] = this.element.nativeElement.children;
    for (const element of elements) {
        this.renderer.removeClass(element, 'list-group-item');
        this.renderer.removeClass(element, 'list-group-item-primary');
        this.renderer.removeStyle(element, 'margin');
    }
}

setProp() {
    const div: HTMLDivElement = this.div.nativeElement;
    this.renderer.setProperty(div, 'textContent', 'Hello Faisal');
    console.log(div);
}

setAttr() {

```

```

const td: HTMLElement = this.tdColspan.nativeElement;
this.renderer.setAttribute(td, 'colspan', '2');
console.log(td);
}

removeAttr() {
  this.renderer.removeAttribute(this.tdColspan.nativeElement, 'colspan');
  console.log(this.tdColspan.nativeElement);
  this.renderer.removeAttribute(this.div.nativeElement, 'textContent');
  console.log(this.div.nativeElement);
}

setValueProp() {
  const input: HTMLInputElement = this.input.nativeElement;
  this.renderer.setProperty(input, 'value', 'Saad');
  console.log(input);
}

setValueAttr() {
  const input: HTMLInputElement = this.input.nativeElement;
  this.renderer.setAttribute(input, 'value', 'Fahd');
  console.log(input);
}
}

```

كما هو واضح الدالة `setValueProp` تقوم بإضافة قيمة للخاصية `value` في الكائن `HTMLInputElement` في DOM، اما الدالة `setValueAttr` تقوم بتغيير القيمة للسمة `value` في العنصر `<input />`. اما النتيجة في المتصفح، فتكون كالتالي:

نلاحظ العنصر في يمين الشاشة وقد اضعنا له السمة `Attribute` واعطيناها قيمة مبدئية وهي `Faisal`، اما السهم الذي على يسار الشاشة فيشير إلى الكائن `HTMLInputElement` بعد تحويل العنصر إليه، وقام المتصفح بقراءة كافة السمات وانشاء

خصائص Property مقابلة لها واعطاها نفس الاسم، ومن هذه السمات التي تم انشاء خاصية مقابلة لها هي value وب نفس الوقت قام بقراءة القيمة الموجودة لهذه السمة وأسندها للخاصية، وكما قلنا سابقاً أن هذه الخصائص منفصلة عن السمات بحيث أي تعديل على الخصائص لا يؤثر على السمات، ولتأكد من ذلك لنقوم بالضغط على زر Set Value Property لنقوم بتغيير الخاصية في DOM، كالتالي:

نلاحظ عند الضغط على الزر تم تغيير قيمة الخاصية value في DOM وتم تغيير القيمة في الكائن على يسار الشاشة، اما السمة في العنصر لاتزال كما هي محتفظة بقيمتها المبدئية وهي Faisal، الآن لنقوم بالضغط على الزر الثاني Set Value Attribute لكي نقوم بتغيير السمة value في العنصر وليس الخاصية value في الكائن في DOM، ولنرى النتيجة:

نلاحظ تم تغيير السمة في العنصر على يمين الشاشة ولكن الخاصية لاتزال محتفظة بقيمتها السابقة، لذلك يجب الانتباه عند التعامل مع الخصائص والسمات ومعرفة كيفية التفريق بينهما ومتى يتم استخدام الدوال `setProperty` و `setAttribute`، ويجدر الإشارة انه في حال قمنا بإعطاء السمة قيمة مبدئية كما فعلنا في هذا المثال ومن ثم قمنا بالضغط على الزر تغيير السمة وليس الخاصية في البداية فإنه سوف يقوم بتغيير السمة وتحديث الخاصية معاً، ولكن عن أي تغيير لهذه الخاصية ومن ثم محاولة تغيير السمة فانه لا يؤثر على الخاصية، وتستطيع التجريب عزيزي المتعلم بنفسك بعد تحديث الصفحة وذلك بالضغط في البداية على زر `Set Value Attribute` ومن ثم سوف تشاهد ان كلاً من السمة والخاصية تغيرتا وبعدها قم بالضغط على زر `Set Value Property` وسوف تشاهد ان الخاصية تغيرت اما السمة فلم تتأثر وبقيت محتفظة بقيمتها السابقة وبعدها قم بالضغط مرة أخرى على زر `Set Value Attribute` ويوف تلاحظ ان الخاصية لم تتأثر ولعل السبب في هذا يرجع انه في حال عدم تغيير القيمة المبدئية في السمة ومن ثم محاولة تغيير السمة فإن هنالك رابط معين بالقيم يؤثر على الخاصية ولكن عند أي تغيير لهذه الخاصية فإن هذا الرابط يختفي.

#### 11.8.4. `selectRootElement()`:

وقد تعاملنا مع هذه الدالة سابقاً عندما استخدمنا `focus` مع مربع الادخال، ولكن لم نقم بشرحها، وهذه الدالة تستقبل بارامتر واحد وهو العنصر الذي نريد الوصول إليه، حيث تقوم بالوصول إلى عنصر محدد بشكل مباشر سواء عن طريق كلاس `css` موجود داخل هذا العنصر او عن طريق اسم العنصر مباشرة او عن طريق إعطاء `Template Reference` للعنصر المستهدف ومن ثم نستقبله في ملف `class` عن طريق `@ViewChild`، لكن يجب الانتباه إلى مجموعة من النقاط المهمة:

- في حال تمرير اسم العنصر مباشرة (`div, input, p, ... etc.`) وكان هنالك أكثر من واحد بنفس ملف `template` لنفس `component` فإنه سوف يقوم بإرجاع اول عنصر موجود.
- في حال كان هذا العنصر يحتوي على مجموعة من العناصر الأبناء واستخدمنا الدالة `selectRootElement` للوصول إلى هذا العنصر فإنه سوف يتم حذف جميع العناصر الأبناء.
- في حال كان المقصد فقط من استخدامها هو الوصول إلى العنصر فقط فعندئذ يُفضل استخدام `Template Reference` و `ViewChild`.

ولتوضيح لنقوم بإعطاء مثال، لنفرض انه لدينا زر عند الضغط عليه يتم عمل `focus` لعنصر `input` وبما انه ليس لدينا إلا عنصر `input` واحد فقط في ملف `template` فسوف استخدم الاسم مباشرة، كالتالي:

ملف `app.component.html`

```
<div class="p-2 border border-primary rounded">
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
```

```

<button class="btn btn-danger m-2" (click)="deleteAllItems()">
  Delete All Item
</button>
<button class="btn btn-info m-2" (click)="selectSibling()">
  Select Sibling
</button>
<button class="btn btn-info m-2" (click)="defaultListStyle()">
  the Default List Style
</button>
<button class="btn btn-success m-2" (click)="setAttr()">
  Set colspan Attribute
</button>
<button class="btn btn-danger m-2" (click)="removeAttr()">
  Remove colspan Attribute
</button>
<button class="btn btn-success m-2" (click)="setProp()">
  set textContent Property
</button>
<button class="btn btn-outline-dark m-2" (click)="setValueProp()">
  set Value Property
</button>
<button class="btn btn-outline-dark m-2" (click)="setValueAttr()">
  Set Value Attribute
</button>
<button class="btn btn-secondary m-2" (click)="focusInput()">
  Focus
</button>
</div>
<ul #ulElement class="list-group"> </ul>
<hr />
<div #div></div>
<hr />
<input #input type="text" value="Faisal" />
<hr />
<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td colspan="2">Sum: $180</td>
  </tr>
</table>

```



ولنكتب محتوى الدالة focusInput في ملف class، كالتالي:

```

import { Component, ViewChild, ElementRef, Renderer2, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {
  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('div') private div: ElementRef;
  @ViewChild('input') private input: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  private count = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
    console.log(this.div.nativeElement);
    console.log(this.input.nativeElement);
  }

  createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
      this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
      this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
  }

  addItem() {
    this.createElement('add');
  }

  insertItemBefor() {
    this.createElement('insert');
  }

  deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
  }
}

```

```

    if (el) {
        this.renderer.removeChild(parent, el);
    }
}

deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
        this.renderer.removeChild(el, el.lastChild);
    }
}

selectSibling() {
    const el = this.element.nativeElement;
    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
    const elements: HTMLLIElement[] = this.element.nativeElement.children;
    for (const element of elements) {
        this.renderer.removeClass(element, 'list-group-item');
        this.renderer.removeClass(element, 'list-group-item-primary');
        this.renderer.removeStyle(element, 'margin');
    }
}

setProp() {
    const div: HTMLDivElement = this.div.nativeElement;
    this.renderer.setProperty(div, 'textContent', 'Hello Faisal');
    console.log(div);
}

setAttr() {
    const td: HTMLTableElement = this.tdColspan.nativeElement;
    this.renderer.setAttribute(td, 'colspan', '2');
    console.log(td);
}

removeAttr() {
    this.renderer.removeAttribute(this.tdColspan.nativeElement, 'colspan');
    console.log(this.tdColspan.nativeElement);
    this.renderer.removeAttribute(this.div.nativeElement, 'textContent');
    console.log(this.div.nativeElement);
}

setValueAttr() {
    const input: HTMLInputElement = this.input.nativeElement;
    this.renderer.setAttribute(input, 'value', 'Fahd');
    this.renderer.setValue(input, 'anwr');
    console.log(input);
}

```



```

setValueProp() {
  const input: HTMLInputElement = this.input.nativeElement;
  this.renderer.setProperty(input, 'value', 'Saad');
  console.log(input);
}

focusInput() {
  this.renderer.selectRootElement('input').focus();
}
}

```

والنتيجة في المتصفح، كالتالي:

نلاحظ وجود مؤشر الفأرة في مربع الادخال بعدما قمنا بالضغط على الزر.

هذا في حال أردنا الوصول إلى العنصر عن طريق الاسم، ونستطيع الوصول إلى العنصر عن طريق كلاس css، كالتالي:

```

app.component.html جزء من ملف
<button class="btn btn-secondary m-2 test" (click)="focusInput()">
  Focus
</button>

```

test هو كلاس فارغ انا قمت بكتابته هنا فقط لكي أستطيع الوصول إليه عن طريق الدالة selectRootElement، كالتالي:

```

focusInput() {
  this.renderer.selectRootElement('.test').focus();
}

```

والنتيجة هي نفسها.

ومن النقاط التي اشرت لها سابقاً ويجب الانتباه لها هي انه في حال كان لدينا اكثر من عنصر فإنه سوف يأخذ اول عنصر وبنفس الوقت إذا كان لدينا عنصر لديه أبناء فإنه سوف يقوم بحذف جميع الأبناء لهذا العنصر، وهاذين النقطتين موجودتين بالعنصر div حيث يوجد اكثر من عنصر div وبنفس الوقت div الأول يحتوي على مجموعة من العناصر الأبناء وهم الأزرار، كالتالي:

```

app.component.html ملف
<div class="p-2 border border-primary rounded"> ← بداية div الأول
  <button class="btn btn-primary m-2" (click)="addItem()">
    Add Item
  </button>
  <button class="btn btn-primary m-2" (click)="insertItemBefor()">
    Insert Item Before
  </button>
  <button class="btn btn-danger m-2" (click)="deleteItem()">
    Delete Item
  </button>
  <button class="btn btn-danger m-2" (click)="deleteAllItems()">
    Delete All Item
  </button>
  <button class="btn btn-info m-2" (click)="selectSibling()">
    Select Sibling
  </button>
  <button class="btn btn-info m-2" (click)="defualtListStyle()">
    the Default List Style
  </button>
  <button class="btn btn-success m-2" (click)="setAttr()">
    Set colspan Attribute
  </button>
  <button class="btn btn-danger m-2" (click)="removeAttr()">
    Remove colspan Attribute
  </button>
  <button class="btn btn-success m-2" (click)="setProp()">
    set textContent Property
  </button>
  <button class="btn btn-outline-dark m-2" (click)="setValueProp()">
    set Value Property
  </button>
  <button class="btn btn-outline-dark m-2" (click)="setValueAttr()">
    Set Value Attribute
  </button>
  <button class="btn btn-secondary m-2 test" (click)="focusInput()">
    Focus
  </button>
</div> ← نهاية div الأول
<ul #ulElement class="list-group"> </ul>
<hr />
<div #div></div> ← الثاني div
<hr />
<input #input type="text" value="Faisal" />
<hr />

```

```

<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>

```

نلاحظ div الأول مؤشر بالسهمين من بدايته ونهايته، وبنفس الوقت هنالك div الثاني، الآن لنقم بتغيير طريق الوصول إلى اسم العنصر ونختار بالتحديد العنصر div، كالتالي:

جزء ملف app.component.ts

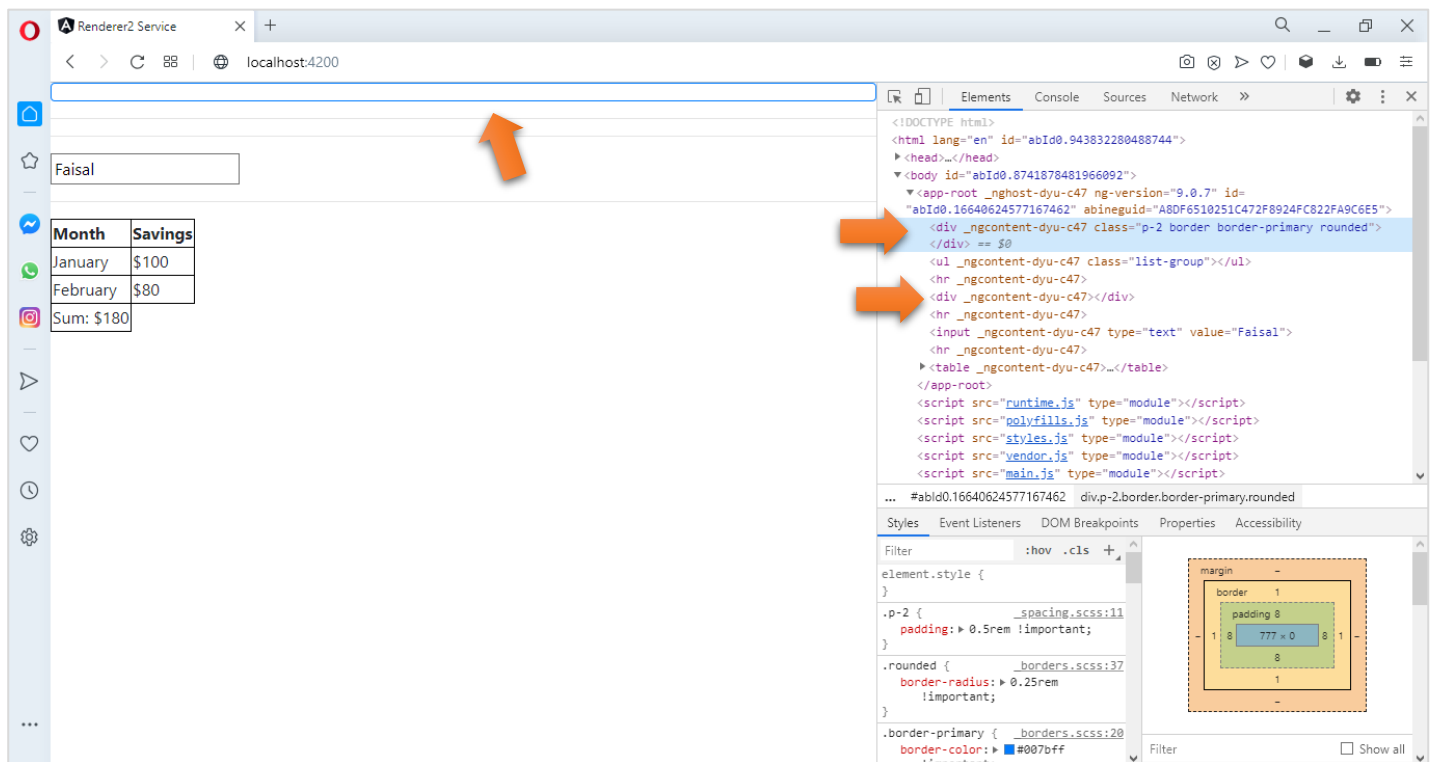
```

focusInput() {
  this.renderer.selectRootElement('div');
}

```

سوف نلاحظ عن تشغيل هذه الدالة وتنفيذ السطر البرمجي فإنه سوف يقوم بقراءة اول div واجهه وبنفس الوقت سوف يُعيد هذا div مع حذف جميع العناصر الأبناء ان وجدت، كالتالي:

الآن لنضغط على الزر Focus، ولنرى النتيجة، كالتالي:



نلاحظ div الأول موجود لكن العناصر الأبناء وهي في حالتنا هذه هي الأبناء، قد تم حذفها، لذلك يجب الحذر عند التعامل مع هذه الدالة وإذا اضطررنا إلى استخدامها ان نستخدمها في العناصر التي ليس لها أي عناصر أبناء، وآخر جزء اريد ان اوضحه انه في حال اردنا ان نستخدم هذه الدالة فقط لكي نصل للعنصر فهنا يُفضل استخدام Template Reference مع ViewChild، كالتالي:

جزء من ملف app.component.ts

```
focusInput() {
  const el: HTMLElement = this.renderer.selectRootElement('.test');
  console.log(el);
}
```

نلاحظ اننا استخدمنا الدالة فقط للوصول إلى العنصر الذي يحتوي على كلاس css ذو الاسم test وتخزين هذا العنصر في ثابت اسميته el، ففي هذه الحالة لا يُفضل استخدام هذه الدالة والاستعاضة عنها بـ Template Reference و ViewChild، وحقيقة أرى ان يتم تغيير هذه الدالة باسم removeChildren لأنني أرى ان افضل استخدام لهذه الدالة هي حذف العنصر او العناصر الأبناء.

#### 12.8.4 .listen():

وتقوم هذه الدالة بمراقبة حدث Event معين وعند وقوع هذا الحدث على عنصر محدد تقوم بتنفيذ دالة ما، وتستقبل ثلاث بارامترات الأول العنصر والثاني الحدث والثالث الدالة التي نريد تنفيذها، وقد وجدت مثال بأحد المراجع والجميل فيه انه يستخدم بعض الدوال السابقة الموجودة في Renderer2، وفكرته بكل بساطة هو عند مرور مؤشر الفأرة على العنصر p في كل مرة يظهر رسالة، وهناك عدد 10 يتم اظهار رسالة انه وصلت للحدث الأعلى، كالتالي:

ملف app.component.html

```
<div class="p-2 border border-primary rounded">
```

```

<button class="btn btn-primary m-2" (click)="addItem()">
  Add Item
</button>
<button class="btn btn-primary m-2" (click)="insertItemBefor()">
  Insert Item Before
</button>
<button class="btn btn-danger m-2" (click)="deleteItem()">
  Delete Item
</button>
<button class="btn btn-danger m-2" (click)="deleteAllItems()">
  Delete All Item
</button>
<button class="btn btn-info m-2" (click)="selectSibling()">
  Select Sibling
</button>
<button class="btn btn-info m-2" (click)="defualtListStyle()">
  the Default List Style
</button>
<button class="btn btn-success m-2" (click)="setAttr()">
  Set colspan Attribute
</button>
<button class="btn btn-danger m-2" (click)="removeAttr()">
  Remove colspan Attribute
</button>
<button class="btn btn-success m-2" (click)="setProp()">
  set textContent Property
</button>
<button class="btn btn-outline-dark m-2" (click)="setValueProp()">
  set Value Property
</button>
<button class="btn btn-outline-dark m-2" (click)="setValueAttr()">
  Set Value Attribute
</button>
<button class="btn btn-secondary m-2 test" (click)="focusInput()">
  Focus
</button>
</div>
<ul #ulElement class="list-group"> </ul>
<hr />
<div #div></div>
<hr />
<input #input type="text" value="Faisal" />
<hr />
<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>

```

```

    <td>February</td>
    <td>$80</td>
  </tr>
  <tr>
    <td #tdColspan>Sum: $180</td>
  </tr>
</table>

<p #listen>
  Hover to see some magic!
</p>
{{ counter }}

```



وفي ملف class، نستقبل هذا Template Reference وبنفس الوقت نكتب الاسطر البرمجية في الدالة ngAfterViewInit، كالتالي:

ملف app.component.ts

```

import { Component, ViewChild, ElementRef, Renderer2, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements AfterViewInit {
  @ViewChild('ulElement') private element: ElementRef;
  @ViewChild('div') private div: ElementRef;
  @ViewChild('input') private input: ElementRef;
  @ViewChild('tdColspan') private tdColspan: ElementRef;
  @ViewChild('listen') private elRef: ElementRef;

  private count = 0;
  private toggle = false;
  counter = 0;

  constructor(private renderer: Renderer2, private parentElement: ElementRef) { }

  ngAfterViewInit(): void {
    console.log(this.tdColspan.nativeElement);
    console.log(this.div.nativeElement);
    console.log(this.input.nativeElement);

    this.renderer.listen(this.elRef.nativeElement, 'mouseover', () => {
      this.toggle = !this.toggle;
      this.counter++;
      const currentElement = this.elRef.nativeElement;
      const firstText = this.renderer.createText('Hover to see new text! (Hover me)');
      const secondText = this.renderer.createText('Text changed! (Hover me)');
      const thirdText = this.renderer.createText('Reached maximum count!');
      this.renderer.selectRootElement(currentElement);
      if (this.counter < 10) {

```

بداية الدالة

```

        this.toggle ? this.renderer.appendChild(currentElement, secondText) :
            this.renderer.appendChild(currentElement, firstText);
    } else {
        this.renderer.appendChild(currentElement, thirdText);
        this.counter = 10;
    }
});  نهاية الدالة
}

createElement(type: string): void {
    const newCount = this.count - 1;
    this.count++;
    const el = this.renderer.createElement('li');
    const text = this.renderer.createText(`Item NO.${this.count}`);
    const refChild = this.element.nativeElement.children;
    this.renderer.appendChild(el, text);
    this.renderer.addClass(el, 'list-group-item');
    this.renderer.addClass(el, 'list-group-item-primary');
    this.renderer.setStyle(el, 'margin', '5px');
    if (type === 'add') {
        this.renderer.appendChild(this.element.nativeElement, el);
    } else if (type === 'insert') {
        this.renderer.insertBefore(this.element.nativeElement, el, refChild[newCount]);
    }
}

addItem() {
    this.createElement('add');
}

insertItemBefor() {
    this.createElement('insert');
}

deleteItem() {
    const el = this.element.nativeElement;
    const parent = this.renderer.parentNode(el);
    this.count = 0;
    if (el) {
        this.renderer.removeChild(parent, el);
    }
}

deleteAllItems() {
    const el = this.element.nativeElement;
    this.count = 0;
    while (el.firstChild) {
        this.renderer.removeChild(el, el.lastChild);
    }
}

selectSibling() {
    const el = this.element.nativeElement;

```

```

    const sibling: HTMLLIElement = this.renderer.nextSibling(el.firstChild);
    alert(`${sibling.textContent} has been Selected`);
}

defaultListStyle() {
    const elements: HTMLLIElement[] = this.element.nativeElement.children;
    for (const element of elements) {
        this.renderer.removeClass(element, 'list-group-item');
        this.renderer.removeClass(element, 'list-group-item-primary');
        this.renderer.removeStyle(element, 'margin');
    }
}

setProp() {
    const div: HTMLDivElement = this.div.nativeElement;
    this.renderer.setProperty(div, 'textContent', 'Hello Faisal');
    console.log(div);
}

setAttr() {
    const td: HTMLTableElement = this.tdColspan.nativeElement;
    this.renderer.setAttribute(td, 'colspan', '2');
    console.log(td);
}

removeAttr() {
    this.renderer.removeAttribute(this.tdColspan.nativeElement, 'colspan');
    console.log(this.tdColspan.nativeElement);
    this.renderer.removeAttribute(this.div.nativeElement, 'textContent');
    console.log(this.div.nativeElement);
}

setValueAttr() {
    const input: HTMLInputElement = this.input.nativeElement;
    this.renderer.setAttribute(input, 'value', 'Fahd');
    this.renderer.setValue(input, 'anwr');
    console.log(input);
}

setValueProp() {
    const input: HTMLInputElement = this.input.nativeElement;
    this.renderer.setProperty(input, 'value', 'Saad');
    console.log(input);
}

focusInput() {
    const el: HTMLTableElement = this.renderer.selectRootElement('.test');
    console.log(el);
}
}

```



نلاحظ استخدمنا الدالة selectRootElement لكي نحذف جميع النص text الأبناء مع كل مرة يتم مرور مؤشر الفأرة على العنصر ومن ثم نضيف نص جديد، والآن لنشاهد النتيجة في المتصفح كالتالي:

the Default List Style Set colspan Attribute Remove colspan Attribute

set textContent Property set Value Property Set Value Attribute Focus

Faisal

Month	Savings
January	\$100
February	\$80
Sum:	\$180

Hover to see some magic!

0

نلاحظ الرسالة الظاهرة، ولنقم الآن بتمرير مؤشر الفأرة فوق هذا النص، كالتالي:

the Default List Style Set colspan Attribute Remove colspan Attribute

set textContent Property set Value Property Set Value Attribute Focus

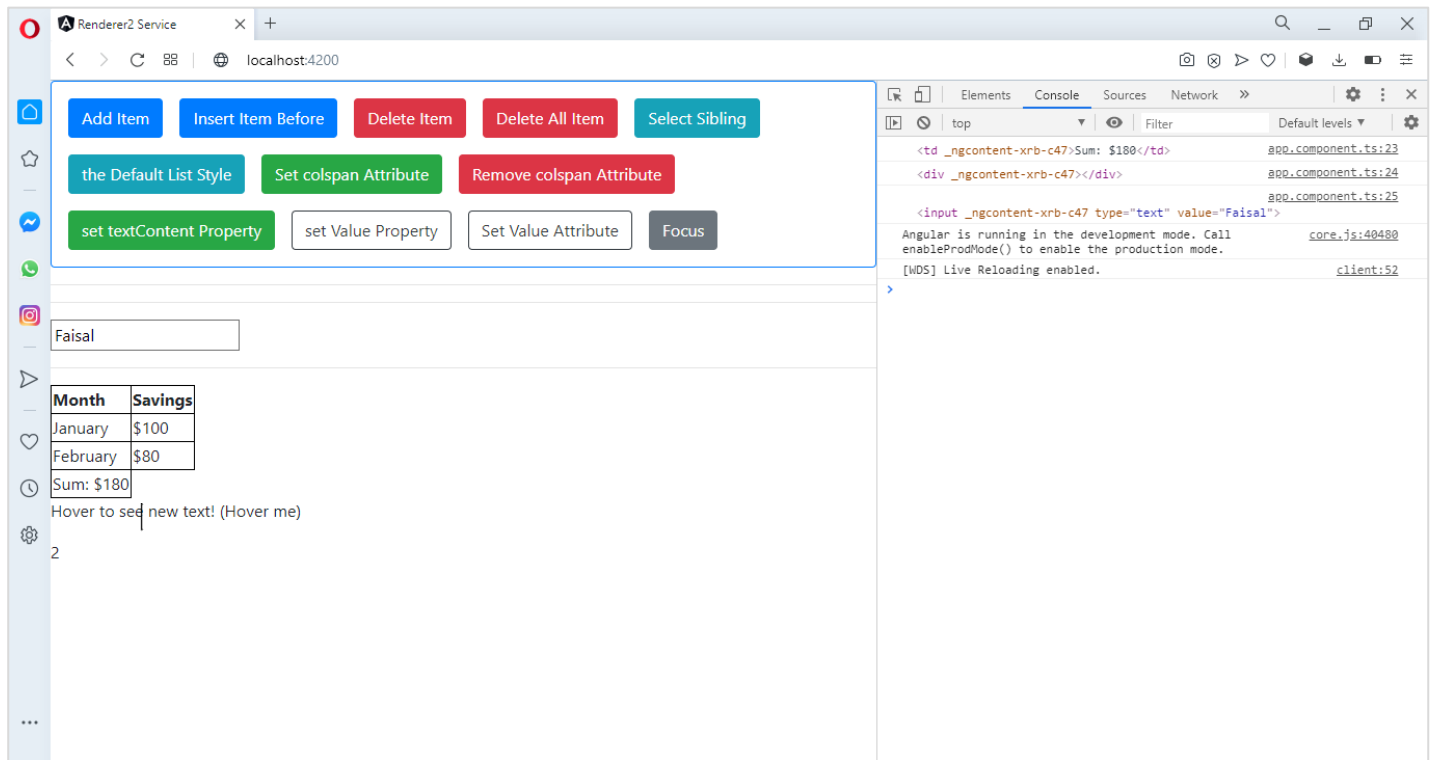
Faisal

Month	Savings
January	\$100
February	\$80
Sum:	\$180

Text changed! (Hover me)

1

ولنمرر مؤشر الفأرة مرة أخرى، ولنرى النتيجة:



وبذلك نكون قمنا بتغطية اغلب الدوال الموجودة في Renderer2 وكيفية الاستفادة منها للتعامل مع DOM، وفي القسم التالي سوف نتكلم عن موضوع لا يقل أهمية وهو التعامل مع components الديناميكية.

## 9.4. ViewChild Token:

تعاملنا كثيراً مع ViewChild وكنا نمرر لها بارامتر واحد وهو اسم العنصر او اسم component او Template Reference، ولكن هنالك بارامتر ثاني (اختياري) وهو read، حيث نستطيع تمرير له مجموعة من القيم، كالتالي:

- TemplateRef
- ElementRef
- ViewContainerRef
- Provider

الأول الثاني غالباً ما يحدث بهما اللبس حيث ان الأول خاص بالعنصر <ng-template></ng-template> وطرق التعامل معها، ولتوضيح، لنقوم بإعطاء مثال، كالتالي:

ملف app.component.html

```
<ng-template #temp> </ng-template>
```

نلاحظ قمنا بإعطاء ng-template متغير Template Reference باسم #temp، الآن سوف نقوم بقراءة هذا المتغير على انه TemplateRef، وهو generic أي لا بد ان نعطيه نوع فرعي وليكن any، كالتالي:

```
import {
  Component,
  AfterViewInit,
  ViewChild,
  ElementRef,
  TemplateRef
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

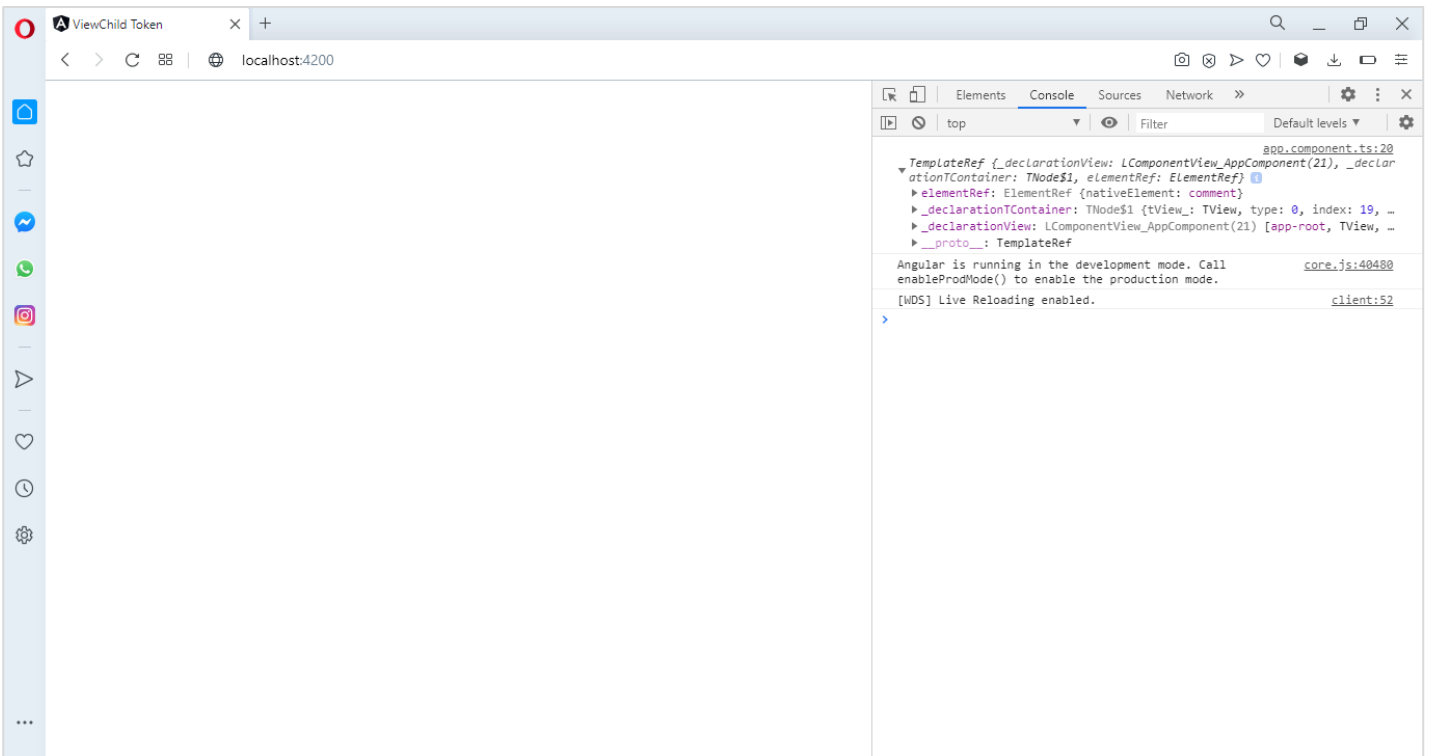
export class AppComponent implements AfterViewInit {

  @ViewChild('temp', { read: TemplateRef }) temp: TemplateRef<any>;

  constructor() { }

  ngAfterViewInit(): void {
    console.log(this.temp);
  }
}
```

نلاحظ مررنا بارامتر آخر على شكل كائن تحتوي على الخاصية read ومن ثم القيمة وهي TemplateRef كأننا نقول له قم بقراءته على أنه Template، وآخر خطوه قمنا بها هي تعريف متغير لنخزن فيه هذه القيم وهو أيضاً من النوع TemplateRef وبما أنه generic مررنا له النوع any، وأخيراً قمنا بطباعة هذا المتغير في console، والنتيجة في المتصفح، كالتالي:



اما الآن لنقم بتغيير القراءة من TemplateRef إلى ElementRef، ولنرى النتيجة:

```
import {
  Component,
  AfterViewInit,
  ViewChild,
  ElementRef,
  TemplateRef
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

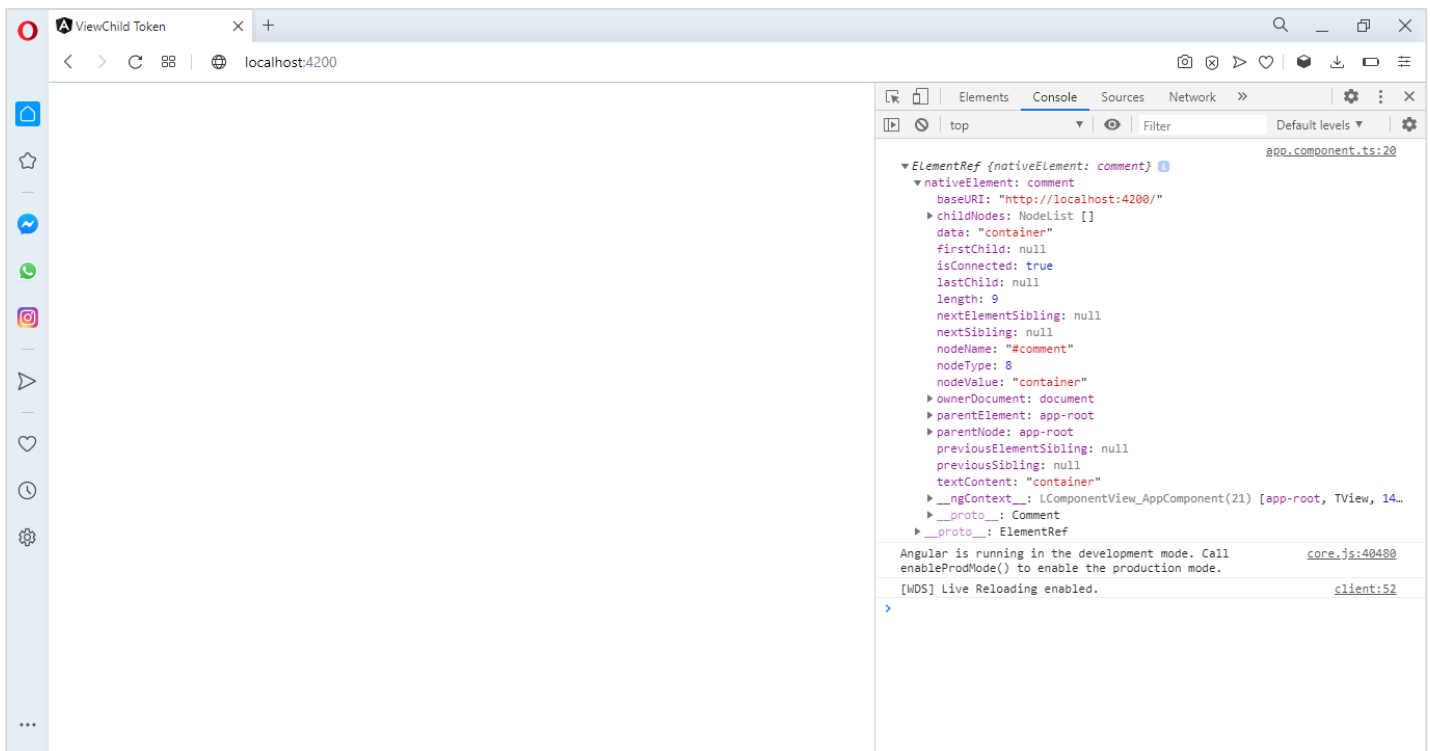
export class AppComponent implements AfterViewInit {

  @ViewChild('temp', { read: ElementRef }) temp: ElementRef;

  constructor() { }

  ngAfterViewInit(): void {
    console.log(this.temp);
  }
}
```

وهنا كأننا نقول له قم بقراءة العنصر ng-template على انه عنصر فقط وليس على انه Template، كالتالي:



نلاحظ انه تعامل معه على انه عنصر فقط مثله مثل أي العناصر الأخرى.

ولكن ماذا عن ViewContainerRef، وسوف نؤجل الكلام فيها إلى الجزء الخاص بي Dynamic Components، أما Provider فنستخدمه لقراءة أي قيمة نضيفها لي Provider الخاص بالـ Component الأب مثل Service أو Directive، وغيرها، ولتوضيح، لنفرض انه لدينا component ابن باسم child، كالتالي:

ملف child.component.html

```
<h4> I am child </h4>
```

اما ملف class فنضيف الكود التالي:

ملف child.component.ts

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  providers: [
    { provide: 'provide', useValue: 'any value' }
  ]
})

export class ChildComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }

}
```



والآن نريد ان نقرأ القيمة any value من component الأب، ومن الحلول استخدام ViewChild في ملف class لي component الأب ونمرر له component الابن، ونجعله يقرأ هذا provide، كالتالي:

ملف app.component.ts

```
import {
  Component,
  AfterViewInit,
  ViewChild,
  ElementRef,
  TemplateRef
} from '@angular/core';

import { ChildComponent } from './child/child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
```

```

}))

export class AppComponent implements AfterViewInit {

  @ViewChild(ChildComponent, { read: 'provide' }) providerToken: string;

  constructor() { }

  ngAfterViewInit(): void {

  }

}

```

ولنقوم بقراءة هذه القيمة وعرضها للمستخدم في ملف template، كالتالي:

ملف app.component.html

```

<app-child></app-child>

<strong>provide</strong> token from child: <b>{{ providerToken }}</b>

```

ولنشاهد النتيجة في المتصفح، كالتالي:

The screenshot shows a web browser window with the URL `localhost:4200`. The page content displays "I am child" and "provide token from child: any value". An orange arrow points to the text "any value". The browser's developer console is open, showing an error: "ERROR Error: ExpressionChangedAfterItHasBeenCheckedError: core.js:6185 Expression has changed after it was checked. Previous value: 'undefined'. Current value: 'any value'". The error stack trace includes: `at throwErrorIfNoChangesMode (core.js:8092)`, `at bindingUpdated (core.js:19773)`, `at interpolation1 (core.js:19917)`, `at eotextInterpolate1 (core.js:24182)`, `at Module.eotextInterpolate (core.js:24153)`, `at AppComponent_Template (app.component.html:3)`, `at executeTemplate (core.js:11949)`, `at refreshView (core.js:11796)`, `at refreshComponent (core.js:13229)`, and `at refreshChildComponents (core.js:11527)`. At the bottom of the console, it says "Angular is running in the development mode. Call enableProdMode() to enable the production mode." and "[WDS] Live Reloading enabled."

نلاحظ قمنا بقراءة هذه القيمة فقط من component الأب، ولو تلاحظ ظهرت لنا رسالة خطأ، كنوع من التدريب سوف اترك لك عزيزي المتعلم إيجاد حل لهذا الخطأ وسوف اعطيك تلميح بسيط واهمس بإذنك اني تكلمت عن هذا الخطأ في القسم الخاص Change Detection.

## 10.4 .NgComponentOutlet:

نستطيع إضافة أي component داخل أي component آخر عن طريق إضافة selector الخاص بالcomponent الأب في component الأب في ملف template، ولكن ماذا لو اردنا ان نضيف هذا component برمجياً، عن طريق ملف class بشكل ديناميكي، ونستطيع القيام بهذا الأمر عن طريق NgComponentOutlet حيث تستقبل باراميتران الأول وهو ضروري وهو عبارة عن component الذي نريد عرضه اما الثاني فهو اختياري وهو عبارة عن ثلاث قيم الأول injector و content و NgModuleFactory، وسوف نتكلم بإذن الله عن هذه القيم جميعاً من خلال الأمثلة.

وأول مثال سوف يكون بسيط وهو عبارة عن اثنان من components الأول وهو child والثاني هو root، بحيث يكون هنالك زر في component الأب root إذا تم الضغط عليه يتم إضافة component الأب إلى هذا component الأب، كالتالي:

ملف child.component.html

```
<h3>Hi from Child Component</h3>
```

وفي ملف template لي component الأب والذي هو في حالتنا هذه Root Component نقوم بإنشاء زر حيث عند الضغط عليه يُنفذ دالة معينة، وبنفس الوقت نكتب الأمر NgComponentOutlet في العنصر ng-container، كالتالي:

ملف app.component.html

```
<button (click)="generateComponent()">Render Dynamic Component</button>
<ng-container *NgComponentOutlet="dynamicComponent"></ng-container>
```

نلاحظ العنصر ng-container وقد كتبنا داخله الأمر NgComponentOutlet مسبقاً بعلامة النجمة \* وأسندنا إليه متغير اسميته dynamicComponent حيث ان هذا المتغير هو الذي يحمل بداخله component الذي نريد عرضه، الآن لنذهب إلى ملف class لهذا component ونعرف المتغير السابق وبنفس الوقت نكتب محتوى الدالة generateComponent والتي تُنفذ في حال تم الضغط على الزر، كالتالي:

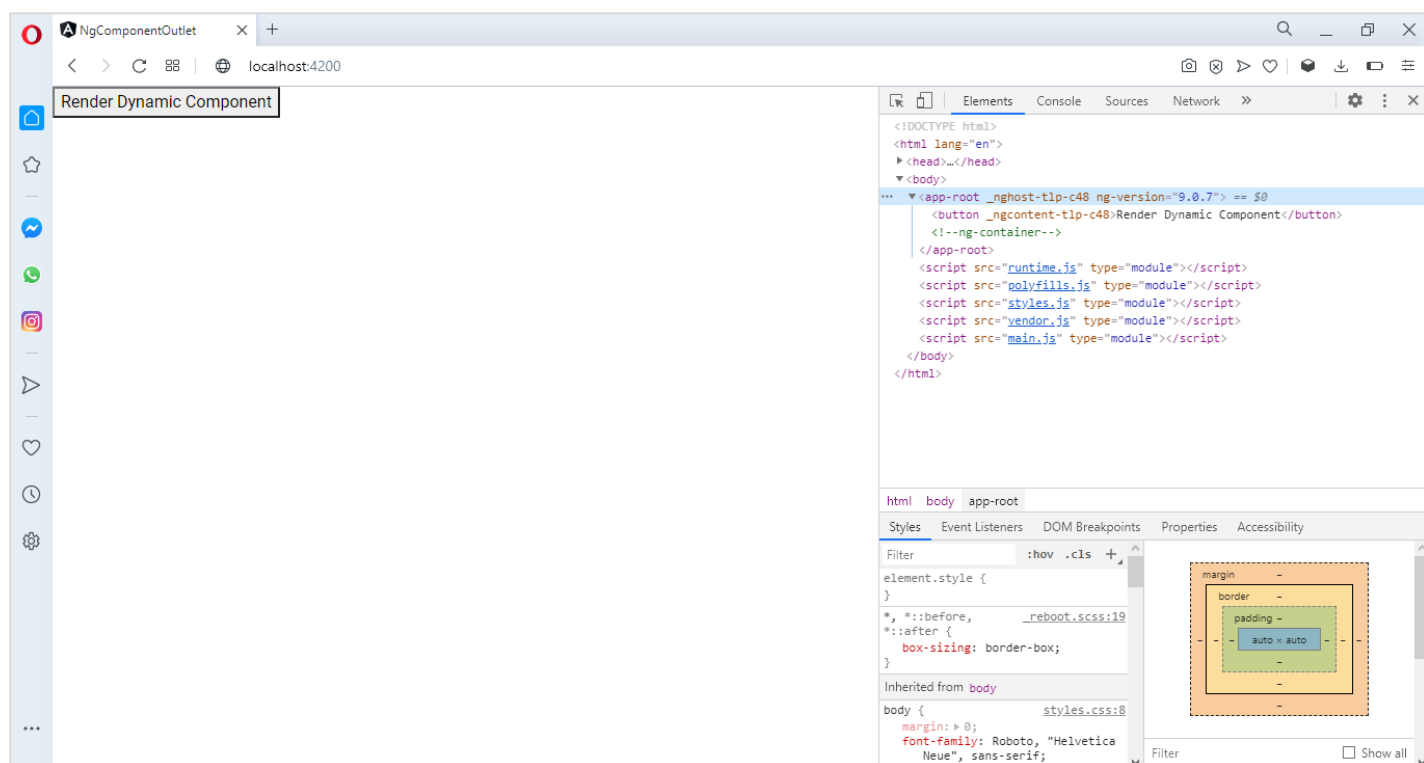
ملف app.component.ts

```
import { Component } from '@angular/core';
import { ChildComponent } from '../child/child.component';

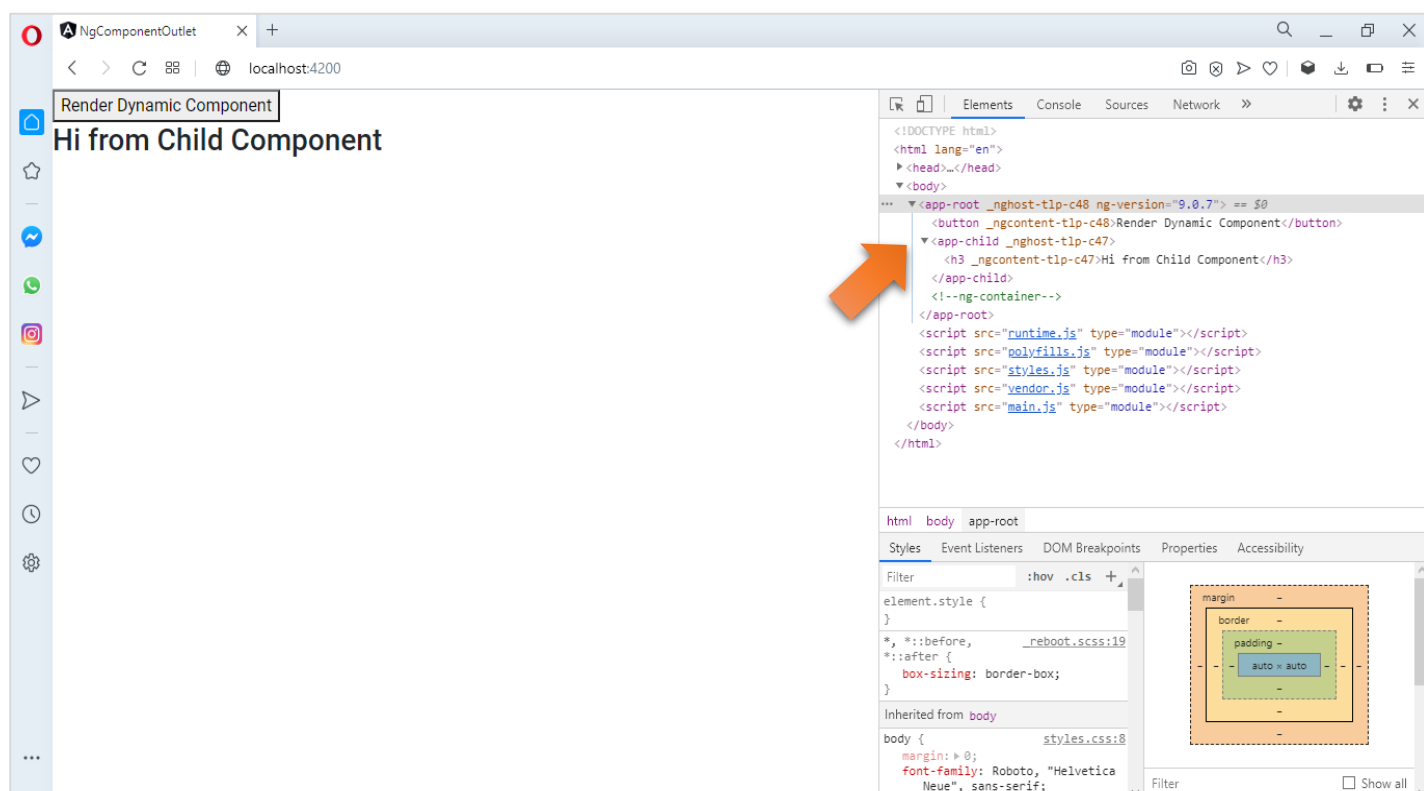
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  public dynamicComponent = null;
  generateComponent() {
    this.dynamicComponent = ChildComponent;
  }
}
```

نلاحظ قمنا بتعريف المتغير `dynamicComponent` ولا بد ان نسند لها قيمة فارغة `null` لأن لو اعطيناها قيمة مبدئية وهي `component` الذي نريد اظهاره فإنه سوف يقوم بإظهار في التطبيق عند بداية تشغيل هذا التطبيق، وهذا الوضع يخالف ما نصبوا إليه وهو عرض هذا `component` عند الضغط على الزر، ومن ثم نقوم بكتابة محتوى الدالة حيث تقوم بكل بساطة بإسناد `component` ذو الاسم `child` إلى المتغير السابق، ولنشاهد النتيجة في المتصفح، كالتالي:



نشاهد DOM في يمين الصفحة لا يحتوي على selector الخاص بالـ `component` الأبن والذي هو `app-child`، والآن لنضغط على الزر ولنشاهد النتيجة، كالتالي:





نلاحظ تمت إضافة العنصر إلى DOM كأبن لي Root Component، ولتوضيح لا يلزم استخدام ng-container فتستطيع استخدام ng-template او حتى أي عنصر من عناصر HTML كـ div مثلاً.

وايضاً لا يلزم كتابة \* قبل الامر السابق فستطيع التعامل معها كأنها Property Binding، كالتالي:

```
<ng-container [ngComponentOutlet]="dynamicComponent"></ng-container>
```

ولنتقل الآن إلى المثال التالي، ولنفرض أنه لدينا مجموعة من components نريد عرضها في مجموعة من tabs بحيث كل tab يعرض component معين بشكل ديناميكي، وسوف نستخدم Angular Material، ولمعرفة كيفية التعامل معها ولإضافة هذه المكتبة الرائعة انصحك بالذهاب إلى هذا الرابط: <https://material.angular.io/>، ولكي لا أدخلك في متاهات قد لا تحتاجها قم فقط بكتابة هذا الأمر في terminal مع التأكد أنك بنفس مسار مجلد المشروع الخاص بك:

```
ng add @angular/material
```

بعد انتهاء التحميل قم بإضافة Module الخاص بي tabs في app.module.ts، كالتالي:

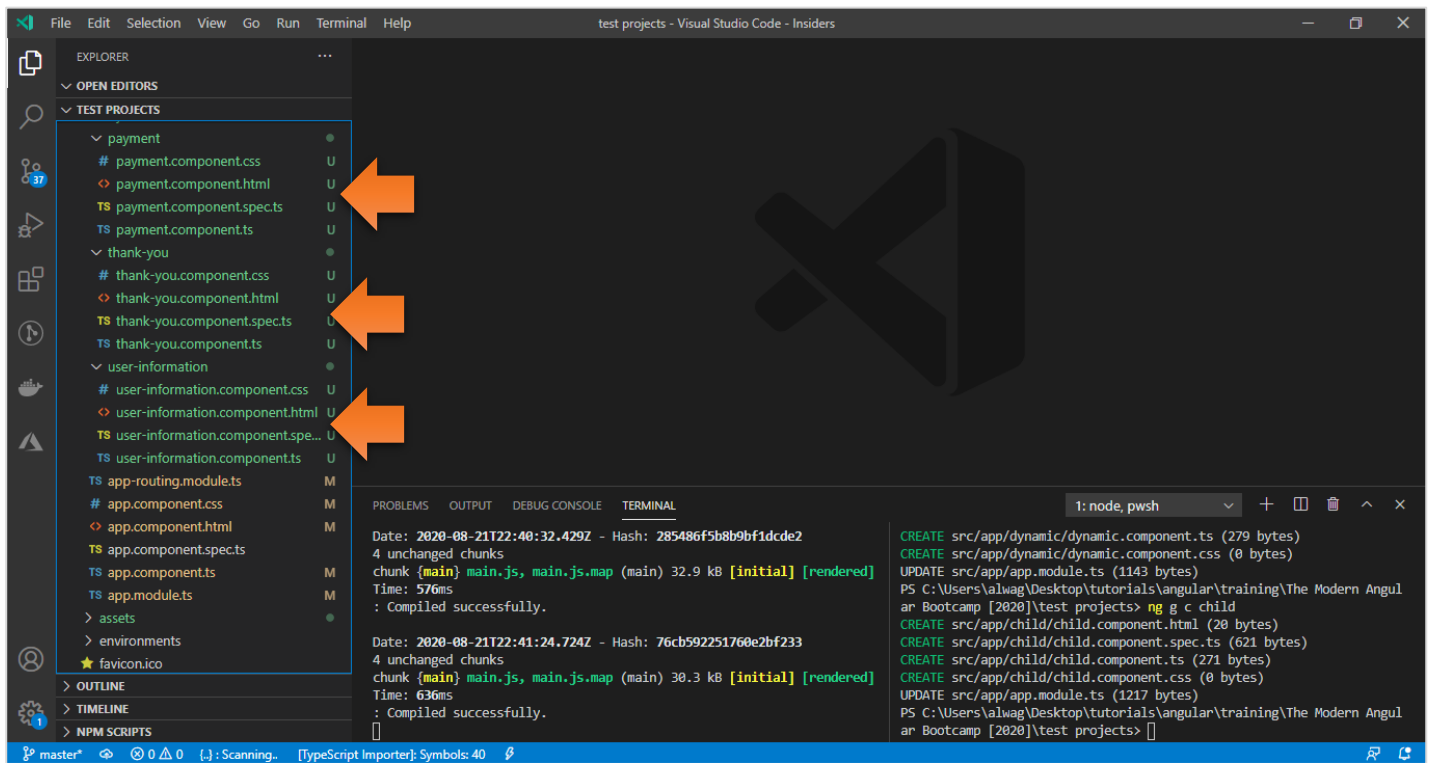
ملف app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MatTabsModule } from '@angular/material/tabs';
import { ChildComponent } from './child/child.component';

@NgModule({
  declarations: [AppComponent, ChildComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    BrowserAnimationsModule,
    MatTabsModule
  ],
  providers: [],
  bootstrap: [AppComponent],
})

export class AppModule { }
```

الآن أصبحت tabs جاهزة لاستخدامها، ولكن قبل استخدامها لنقوم بإنشاء ثلاثة components، كالتالي:



بطبيعة الحال لك حرية اختيار أسماء components التي تناسب احتياجاتك.

الآن لنقم بإنشاء مصفوفة تحتوي على مجموعة من الكائنات كل كائن هو عبارة عن عنوان والـ component الذي نريد عرضه، كالتالي:

```
app.component.ts ملف
import { Component } from '@angular/core';
import { PaymentComponent } from './payment/payment.component';
import { ThankYouComponent } from './thank-you/thank-you.component';
import { UserInformationComponent } from './user-information/user-information.component';
import { ChildComponent } from './child/child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  public dynamicComponent: ChildComponent = null;

  public dynamicTabs = [
    { label: 'User Information', component: UserInformationComponent },
    { label: 'Payment', component: PaymentComponent },
    { label: 'Thank You', component: ThankYouComponent }
  ];

  generateComponent() {
    this.dynamicComponent = ChildComponent;
  }
}
```

وفي ملف template، نقوم بعملية الربط الديناميكي بين كل component و tab الخاص به، كالتالي:

ملف app.component.html

```
<mat-tab-group>
  <mat-tab *ngFor="let tab of dynamicTabs">
    <ng-template mat-tab-label style={{ tab.label }}></ng-template>
    <ng-container *ngComponentOutlet="tab.component"></ng-container>
  </mat-tab>
</mat-tab-group>

<hr />
<hr />

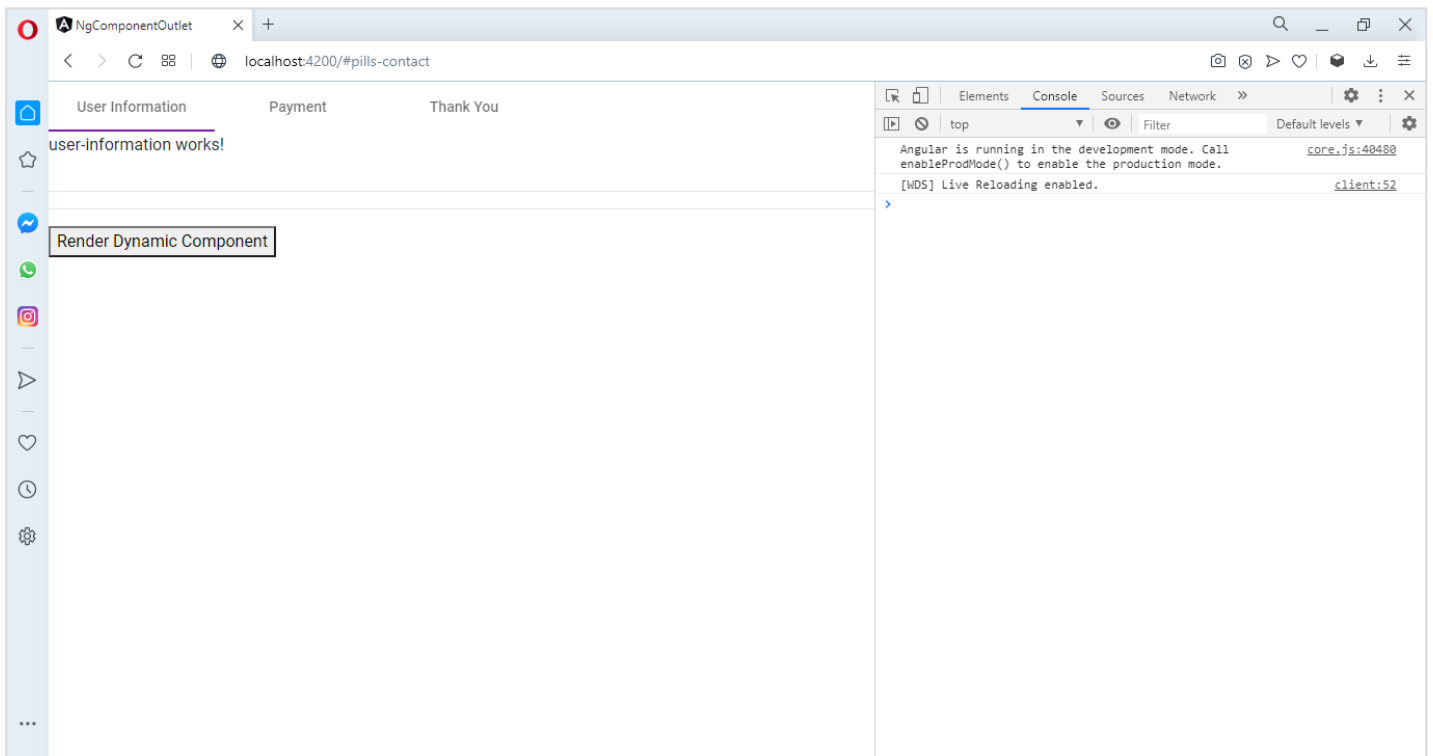
<button (click)="generateComponent()">Render Dynamic Component</button>
<ng-container [ngComponentOutlet]="dynamicComponent"></ng-container>
```

وفي ملف style نضيف التالي:

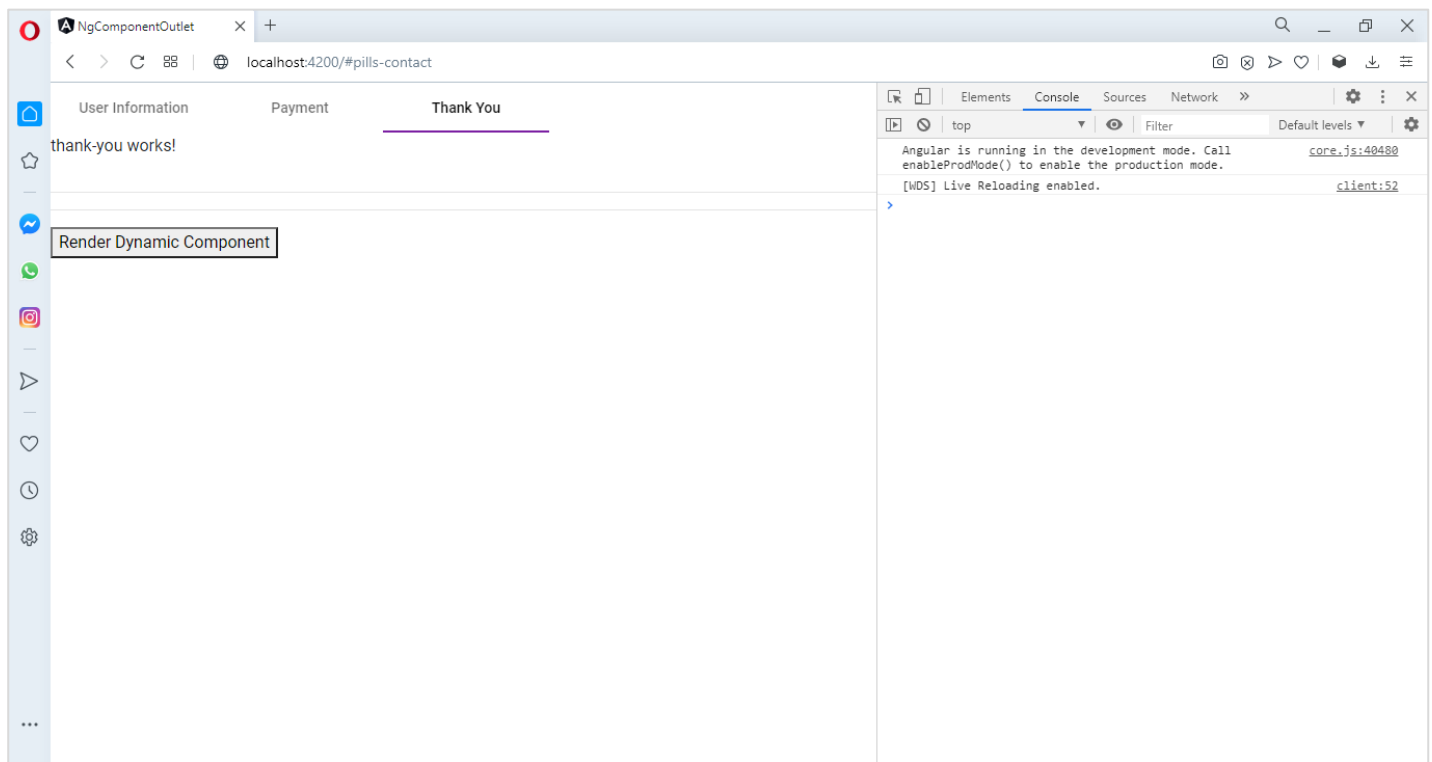
ملف app.component.css

```
:host ::ng-deep .mat-tab-label-content {
  color: black;
}
```

والنتيجة في المتصفح، كالتالي:



ولنختار تبويب tab آخر، ولنشاهد النتيجة:



نلاحظ تم إضافة مجموعة من الـ components الأبناء بشكل ديناميكي عن طريق ملف class لي component الأب.

ولكن ماذا لو كان هنالك @Input لهذا Component ولا نريد ان نقوم بعرضه بشكل ديناميكي وانما نريد ان نمرر له بيانات عن طريق هذا @Input، وهذا هو مثالنا الثالث، لذلك لنقوم بإنشاء component جديد وليكن اسمه dynamic ولنفرض انه لدينا @Input في هذا component، ونعرض هذا الاسم في ملف template، كالتالي:

ملف dynamic.component.ts

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

@Component({
  selector: 'app-dynamic',
  templateUrl: './dynamic.component.html',
  styleUrls: ['./dynamic.component.css']
})

export class DynamicComponent implements OnInit {
  @Input() name: string;

  constructor() { }

  ngOnInit(): void {
  }
}
```

ملف child.component.html

```
<h3>Hi {{ name }} from Child Component</h3>
```

الآن نريد ان نقوم بعرض هذا component بشكل ديناميكي في Root Component وبنفس الوقت لابد ان نمرر بيانات إلى هذا component.

في الطريقة التقليدية كنا سوف نستخدم هذا الطريقة في ملف template في component الأب، كالتالي:

```
<app-dynamic [name]="name"></app-dynamic>
```

ولكن هنا الوضع مختلف حيث أن هذا component الذي نشير إليه بالعنصر <app-dynamic> يتم اضافته بشكل ديناميكي أثناء وقت التشغيل التطبيق، او بصيغة أخرى عن طريق ملف class في Root Component، لذلك لابد ان نستخدم طريقة أخرى تناسب مع هذا الوضع، وهذه الطريقة هي في البارامتر الثاني وبالتحديد في الخاصية Injector، وهو جزء من نظام معقد تعتمد Angular من ضمن نواة إطار العمل، وليس هنا المقام لتفصيل فيه، ولكن سوف نستفيد من هذا النظام للوصول إلى component الديناميكي وبالأخص @Input الذي نريد تمريره إلى هذا component وتمرير قيمة له، وأول خطوة نقوم بها هي تعريف ثابت على انه Injection ومهمته بشكل مبسط أخذ القيمة من component الأب وتوصيلها إلى component الابن، ولكن لابد ان نعطيه صلاحية عن طريق Providers، كالتالي:

ملف app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { InjectionToken, NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MatTabsModule } from '@angular/material/tabs';
import { UserInformationComponent } from './user-information/user-information.component';
import { PaymentComponent } from './payment/payment.component';
import { ThankYouComponent } from './thank-you/thank-you.component';
import { DynamicComponent } from './dynamic/dynamic.component';
import { ChildComponent } from './child/child.component';

export const NAME = new InjectionToken<string>('');

@NgModule({
  declarations: [
    AppComponent,
    UserInformationComponent,
    PaymentComponent,
    ThankYouComponent,
    DynamicComponent,
    ChildComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
```

```

    BrowserAnimationsModule,
    MatTabsModule
  ],
  providers: [{ provide: NAME, useValue: '' }],
  bootstrap: [AppComponent],
})
export class AppModule { }

```



في السطر البرمجي الذي تم التأشير عليه في السهم الأول قمنا بتعريف ثابت جديد باسم NAME ومررنا له قيمة مبدئية فارغة، مع العلم انه قد تظهر لك رسالة تحذيرية ومحتواها انه من الأفضل كتابة هذا السطر البرمجي في ملف منفصل ومن ثم استدعائه هنا او إذا كان يُستخدم في component واحد نكتفي بإضافته لي providers الخاصة بهذا component فقط، ولكن لتبسيط سوف نبقى الأمر كما هو عليه، اما السطر الثاني الذي تم التأشير عليه بالسهم هو لتعريف هذا الثابت على انه provider، وبما اننا قمنا بكتابته هنا فهو بذلك أصبح متاح لجميع components على مستوى التطبيق.

الخطوة التالية هي الذهاب إلى component الأب الذي يحتوي على هذا component الديناميكي ونقوم بإجراء التعديلات التالية:

#### ملف app.component.ts

```

import { Component, Injector, ReflectiveInjector } from '@angular/core';
import { PaymentComponent } from '../payment/payment.component';
import { ThankYouComponent } from '../thank-you/thank-you.component';
import { UserInformationComponent } from '../user-information/user-information.component';
import { ChildComponent } from '../child/child.component';
import { DynamicComponent } from '../dynamic/dynamic.component';
import { NAME } from './app.module';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  public dynamicComponent = null;

  public customInjector: Injector;
  public dynamicComponentWithInjector = null;

  public dynamicTabs = [
    {
      label: 'User Information',
      component: UserInformationComponent
    },
    {
      label: 'Payment',
      component: PaymentComponent
    },
    {
      label: 'Thank You',

```



```

    component: ThankYouComponent
  }
];

constructor(private injector: Injector) {

}

generateComponent() {
  this.dynamicComponent = ChildComponent;
}

createInjector() {
  const name = 'Faisal';
  // tslint:disable-next-line: deprecation
  this.customInjector = ReflectiveInjector.resolveAndCreate(
    [{ provide: NAME, useValue: name }],
    this.injector
  );
  this.dynamicComponentWithInjector = DynamicComponent;
}
}

```

هنالك ثلاث تعديلات او بصيغة أخرى ثلاث إضافات الأولى هي متغيرين اسميت الأول customInjector وهو عبارة عن Injector الذي سوف نعمل له bind للخاصية injector في ملف template، اما الثاني فهو يمثل component الذي نريد ان نعرضه بشكل ديناميكي.

والإضافة الثانية هي عملنا inject لي service ذات الاسم Injector.

والإضافة الأخيرة هي دالة مهمتها تمرير قيمة ديناميكية معينة وفي مثالنا هذا هي القيمة Faisal الموجودة في الثابت name حيث نمررها عن طريق نظام Injection بالاعتماد على provide ذو الاسم NAME الذي قمنا بإنشاءه سابقا، ويحمل القيمة name، وجميعها مخزنة في المتغير (الخاصية) customInjector.

الآن سوف نقوم بعمل bind لهذه المتغير في ملف template، كالتالي:

```

ملف app.component.html
<mat-tab-group>
  <mat-tab *ngFor="let tab of dynamicTabs">
    <ng-template mat-tab-label>{{ tab.label }}</ng-template>
    <ng-container *ngComponentOutlet="tab.component"></ng-container>
  </mat-tab>
</mat-tab-group>

<hr />
<hr />

<button (click)="generateComponent()">Render Dynamic Component</button>
<ng-container [ngComponentOutlet]="dynamicComponent"></ng-container>

```

```

<hr />
<hr />

<button (click)="createInjector()">
  Render Dynamic Component With Injector
</button>
<ng-container *ngComponentOutlet="dynamicComponentWithInjector; injector: customInjector">
</ng-container>

```

في البداية اضفنا زر يقوم بتنفيذ الدالة السابقة والتي تقوم بقراءة قيمة وإضافتها إلى نظام Injection والمتمثل عندنا في المتغير customInjector، اما الخطوة الثانية فهي إضافة component الذي نريد اظهاره بشكل ديناميكي إلى \*ngComponentOutlet وبالنسبة للبارامتر الثاني والذي هو اختياري مررنا له customInjector. والجزء الأخير هو قراءة هذه القيمة وتميرها إلى @Input في component الديناميكي، كالتالي:

ملف dynamic.component.ts

```

import { Component, Inject, Input, OnInit } from '@angular/core';
import { NAME } from '../app.module';

@Component({
  selector: 'app-dynamic',
  templateUrl: './dynamic.component.html',
  styleUrls: ['./dynamic.component.css']
})
export class DynamicComponent implements OnInit {
  @Input() name: string;
  constructor(@Inject(NAME) private nameInjected: string) { }

  ngOnInit(): void {
    this.name = this.nameInjected;
  }
}

```

نلاحظ قمنا بقراءة provide ذو الاسم NAME في متغير اسميته nameInjected واخيراً نسند القيمة الموجودة في نظام Injection في المتغير name.

وأخر جزء هو عرض هذه القيمة في ملف template، كالتالي:

ملف dynamic.component.html

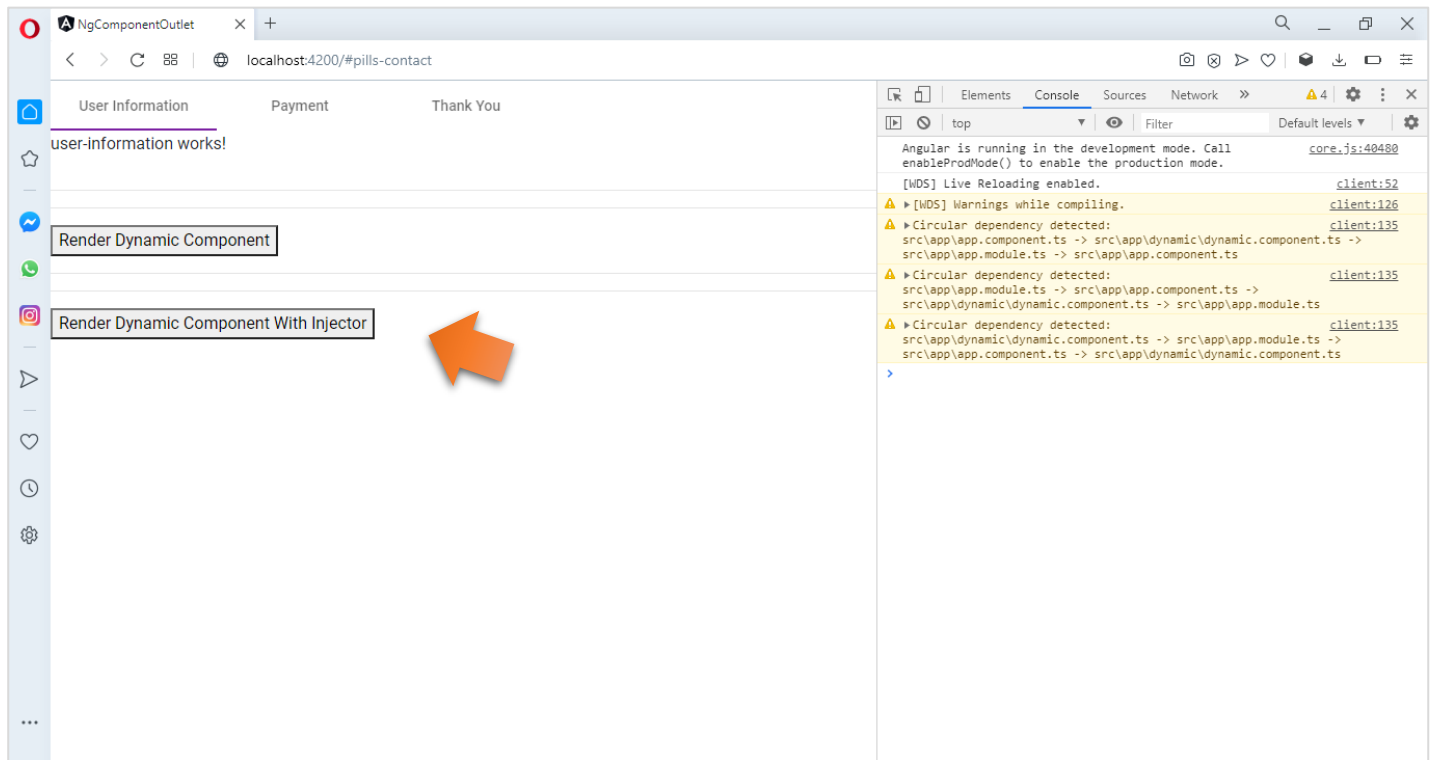
```

<h3>
  Hi <span style="color: red;">{{ name }}</span> From Dynamic Component
</h3>

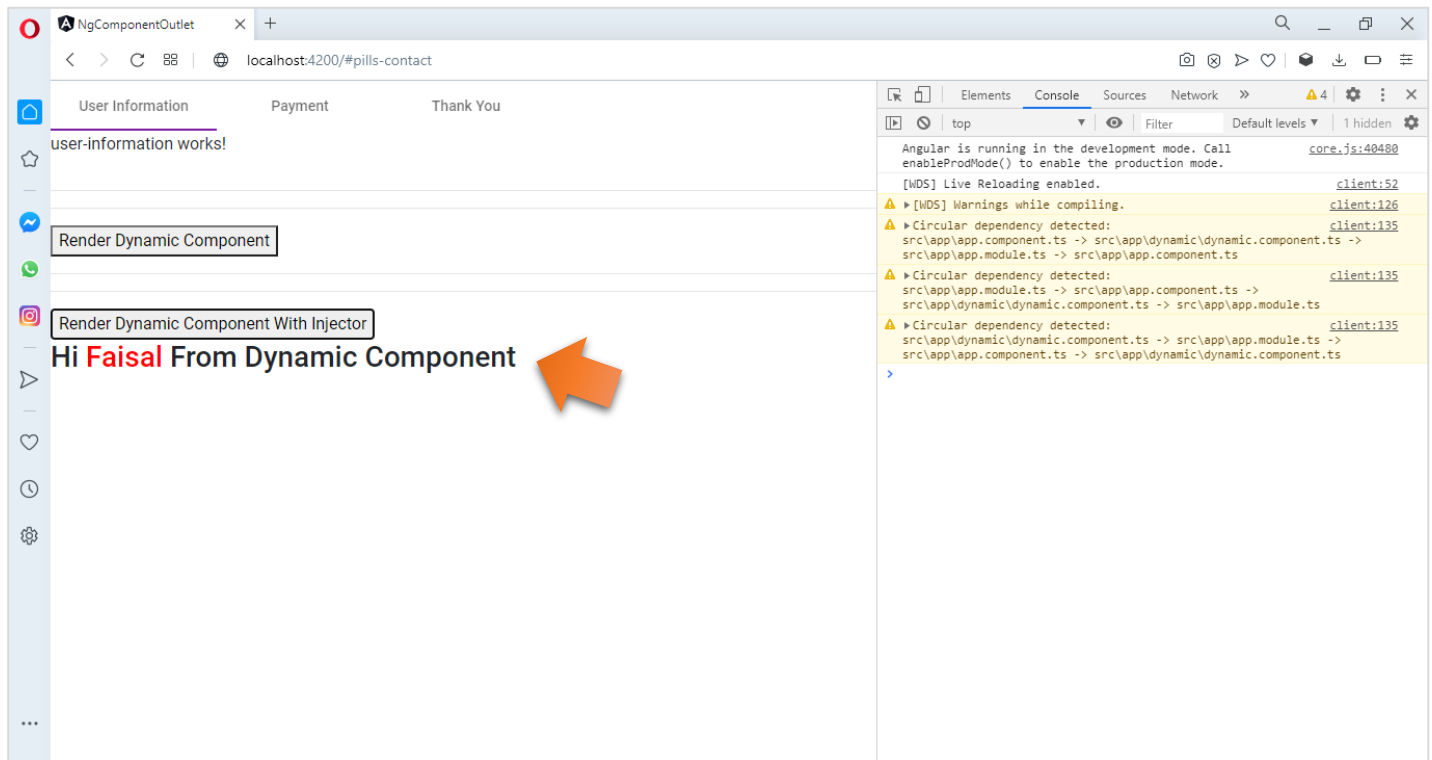
```

اما النتيجة في المتصفح، فتكون كالتالي:

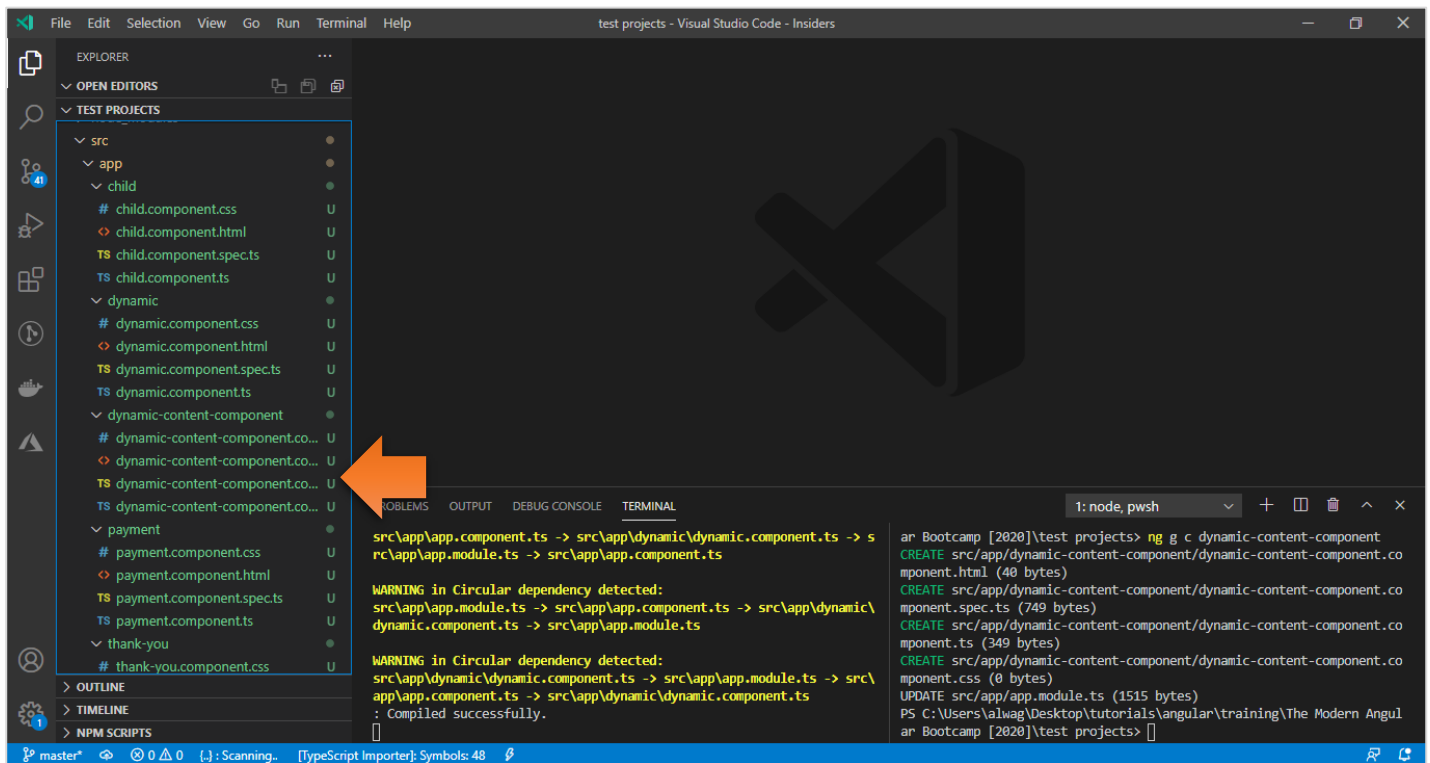




الآن لنضغط على الزر المؤشر عليه بالسهم، ولنرى النتيجة:



اما المثال التالي فهو لتوضيح طريقة استخدام ng-content مع ngComponentOutlet، ولنفرض انه لدينا component الأب ونريد ان نعرض component بشكل ديناميكي وبنفس الوقت نمرر مجموعة من contents عن طريق ng-content، وللقيام بهذا الأمر نقوم بإنشاء component جديد وليكن اسمه dynamic-content-component، كالتالي:



ولنضيف ng-content بداخل ملف template لهذا component، كالتالي:

ملف dynamic-content-component.component.html

```
<ng-content></ng-content>
<ng-content select="content2"></ng-content>
```

الآن لنذهب إلى ملف Root Component وننفذ الخطوات التالية:

ملف app.component.ts

```
import { Component, Injector, ReflectiveInjector } from '@angular/core';
import { PaymentComponent } from './payment/payment.component';
import { ThankYouComponent } from './thank-you/thank-you.component';
import { UserInformationComponent } from './user-information/user-information.component';
import { ChildComponent } from './child/child.component';
import { DynamicComponent } from './dynamic/dynamic.component';
import { NAME } from './app.module';
import {
  DynamicContentComponentComponent
} from './dynamic-content-component/dynamic-content-component.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
  public dynamicComponent = null;
  public customInjector: Injector;
  public dynamicComponentWithInjector = null;
  public outlet = null;
```

```

public dynamicTabs = [
  {
    label: 'User Information',
    component: UserInformationComponent
  },
  {
    label: 'Payment',
    component: PaymentComponent
  },
  {
    label: 'Thank You',
    component: ThankYouComponent
  }
];

constructor(private injector: Injector) {

}

generateComponent() {
  this.dynamicComponent = ChildComponent;
}

createInjector() {
  const name = 'Faisal';
  // tslint:disable-next-line: deprecation
  this.customInjector = ReflectiveInjector.resolveAndCreate(
    [{ provide: NAME, useValue: name }],
    this.injector
  );
  this.dynamicComponentWithInjector = DynamicComponent;
}

createDynamicComponentWithContent() {
  this.outlet = DynamicContentComponentComponent;
}
}

```



ملف app.component.html

```

<mat-tab-group>
  <mat-tab *ngFor="let tab of dynamicTabs">
    <ng-template mat-tab-label>{{ tab.label }}</ng-template>
    <ng-container *ngComponentOutlet="tab.component"></ng-container>
  </mat-tab>
</mat-tab-group>

<hr />
<hr />

<button (click)="generateComponent()">Render Dynamic Component</button>
<ng-container [ngComponentOutlet]="dynamicComponent"></ng-container>

<hr />

```

```

<hr />

<button (click)="createInjector()">
  Render Dynamic Component With Injector
</button>
<ng-container
  *ngComponentOutlet="dynamicComponentWithInjector; injector: customInjector"
>
</ng-container>

<hr />
<hr />

<button (click)="createDynamicComponentWithContent()">
  Render Dynamic Component With Content
</button>
<ng-container *ngComponentOutlet="outlet; content: [[first], [second]]">
</ng-container>
<div #first [hidden] = "!outlet">
  <h1>CONTENT1</h1>
  <h4>content1 text</h4>
</div>
<div #second [hidden] = "!outlet">
  <h1>CONTENT2</h1>
  <h4 id="content2">content2 text</h4>
</div>

```

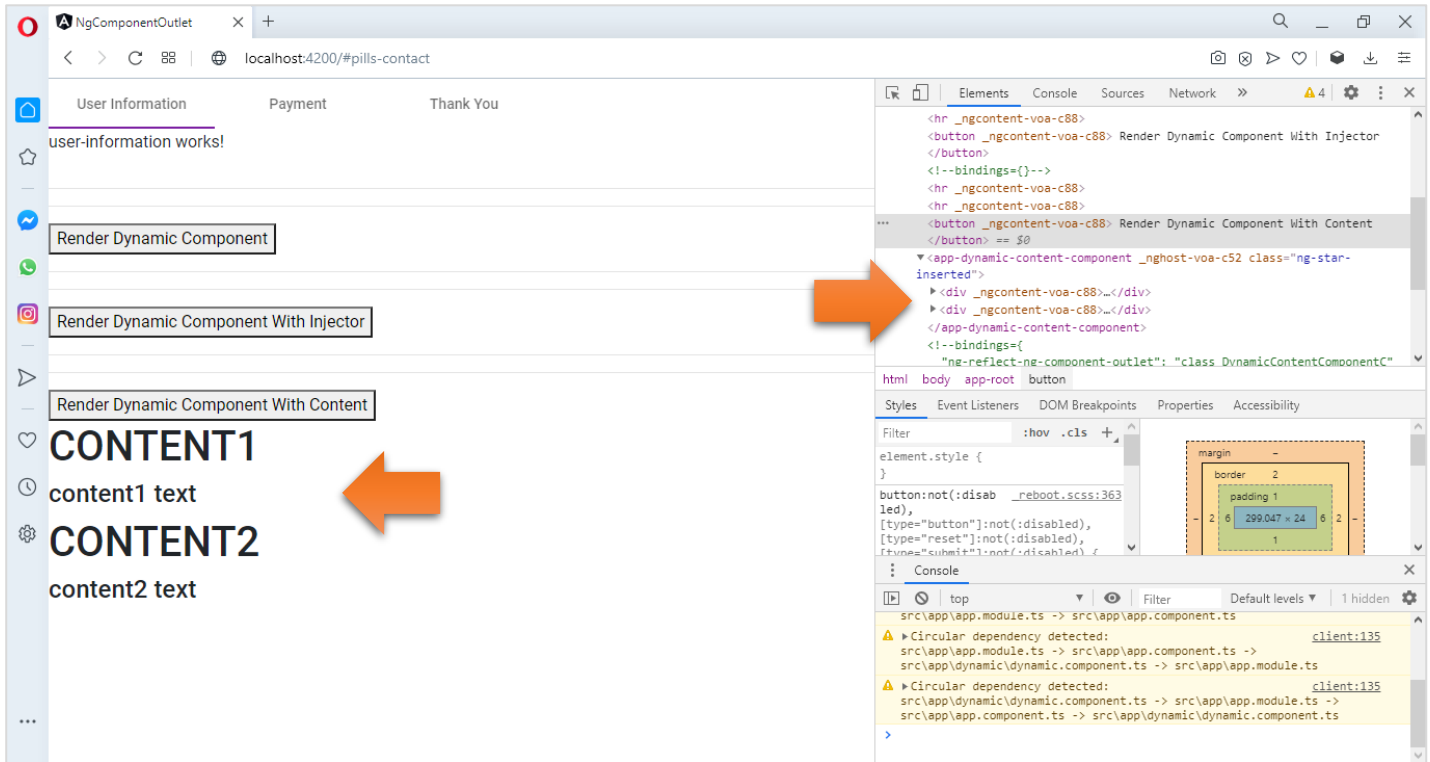


نلاحظ استخدمنا الخاصية الأخرى وهي content ومررنا لها جميع Markup الذي نريد ان يكون content إلى component الديناميكي عن طريق Template Reference، ووضعتنا شرط بسيط وهو انه لا تظهر هذا Markup إلى في حال تم بناء component الديناميكي والمتمثل بالمتغير outlet.

الآن لنشاهد النتيجة في المتصفح، كالتالي:

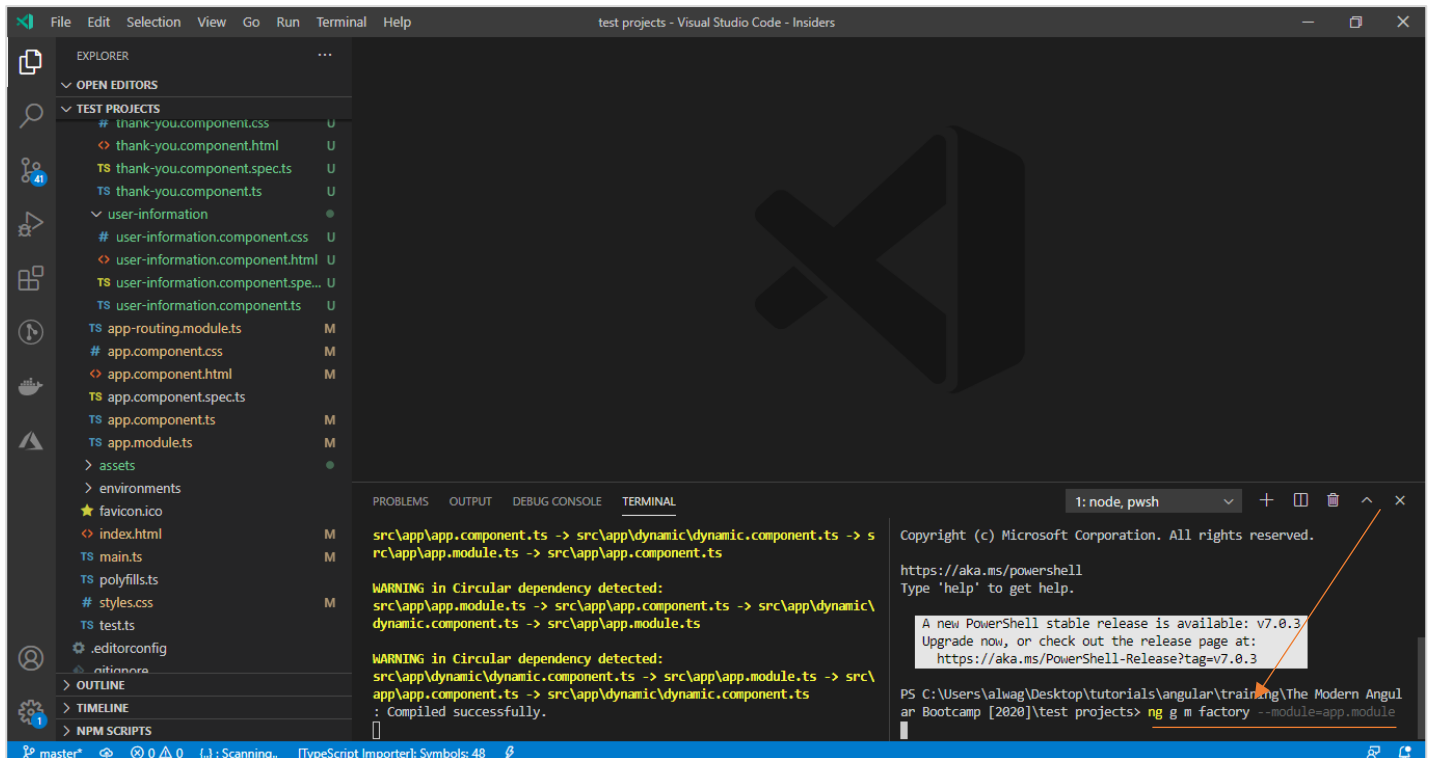
The screenshot shows a web browser window with the URL `localhost:4200/#pills-contact`. The page displays a contact form with three buttons: "User Information", "Payment", and "Thank You". Below these buttons, there are three buttons for rendering dynamic components: "Render Dynamic Component", "Render Dynamic Component With Injector", and "Render Dynamic Component With Content". The "Render Dynamic Component With Content" button is highlighted with an orange arrow. The browser's console shows several warnings related to circular dependencies in the Angular application.

ولنضغط على الزر ولنرى النتيجة، كالتالي:

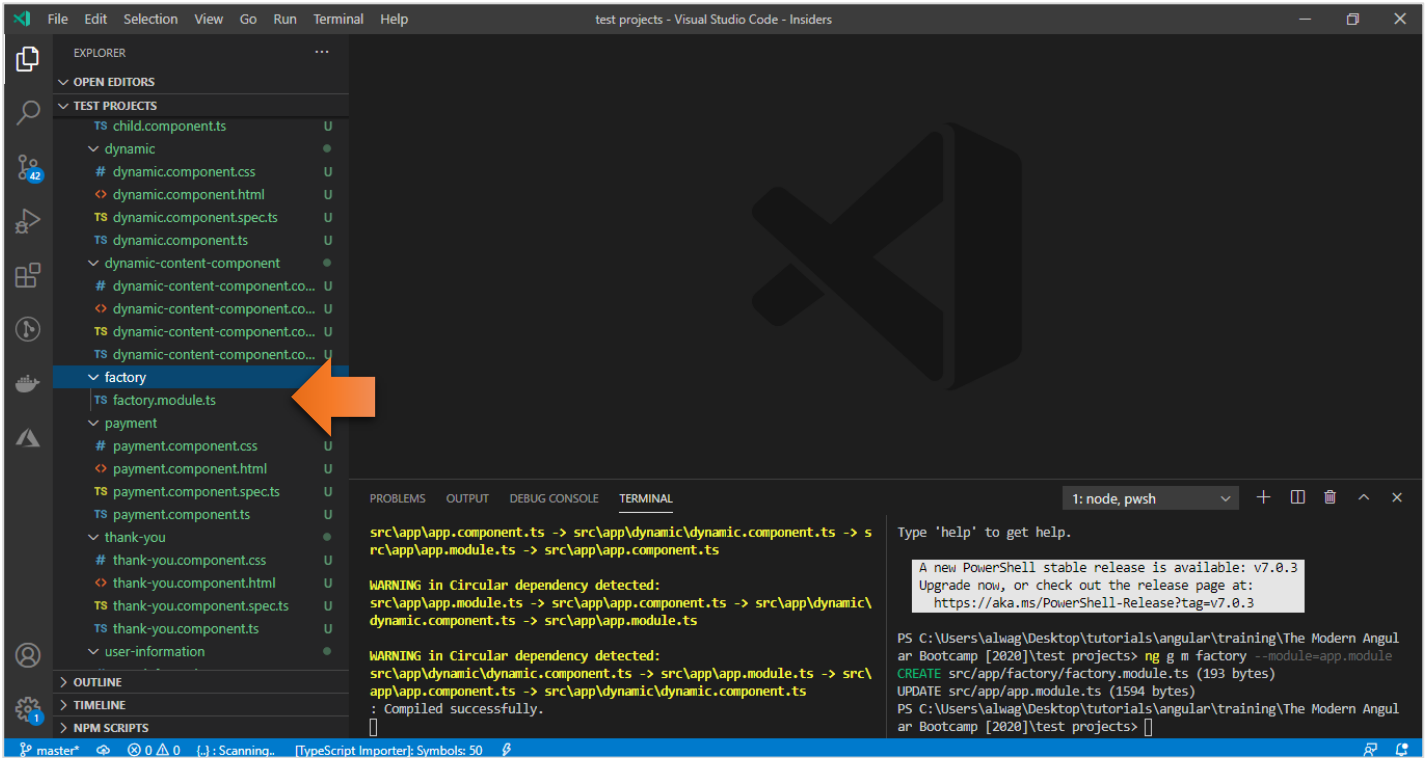


نلاحظ كما هو موجود في يمين الشاشة تم إضافة العناصر إلى DOM.

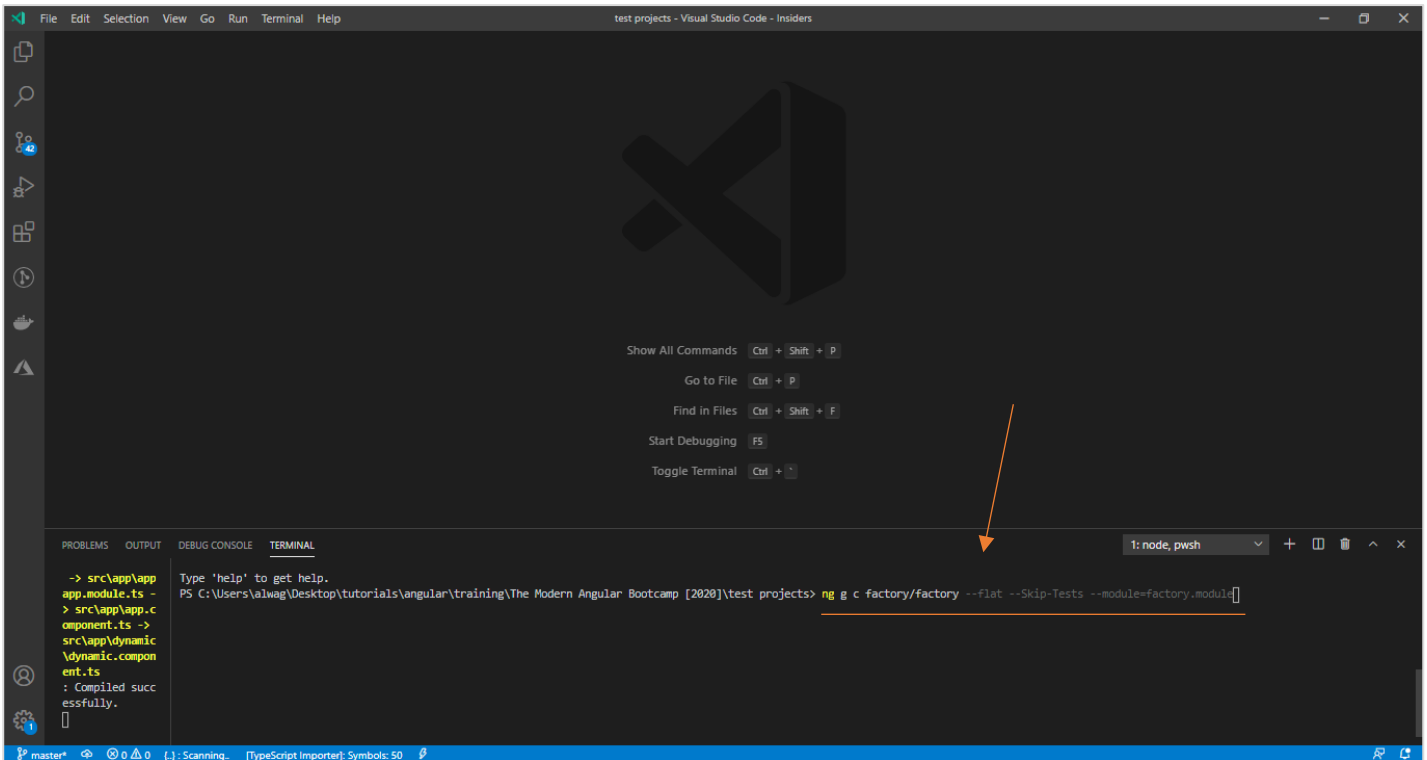
اما المثال الأخير فهو لآخر خاصية اختيارية وهي ngModuleFactory وتكمن الفكرة اننا نستطيع عرض بعض components الموجودة في Module آخر وليس شرط ان تكون موجودة في Module الرئيسي (ولفهم أعمق عن Module انصحك بقراءة كتابي Angular Routing and Module وهو جزء من هذه السلسلة)، ولتوضيح لنقوم بإنشاء Module جديد وليكن اسمه factory عن طريق كتابة هذا الأمر في terminal، كالتالي:



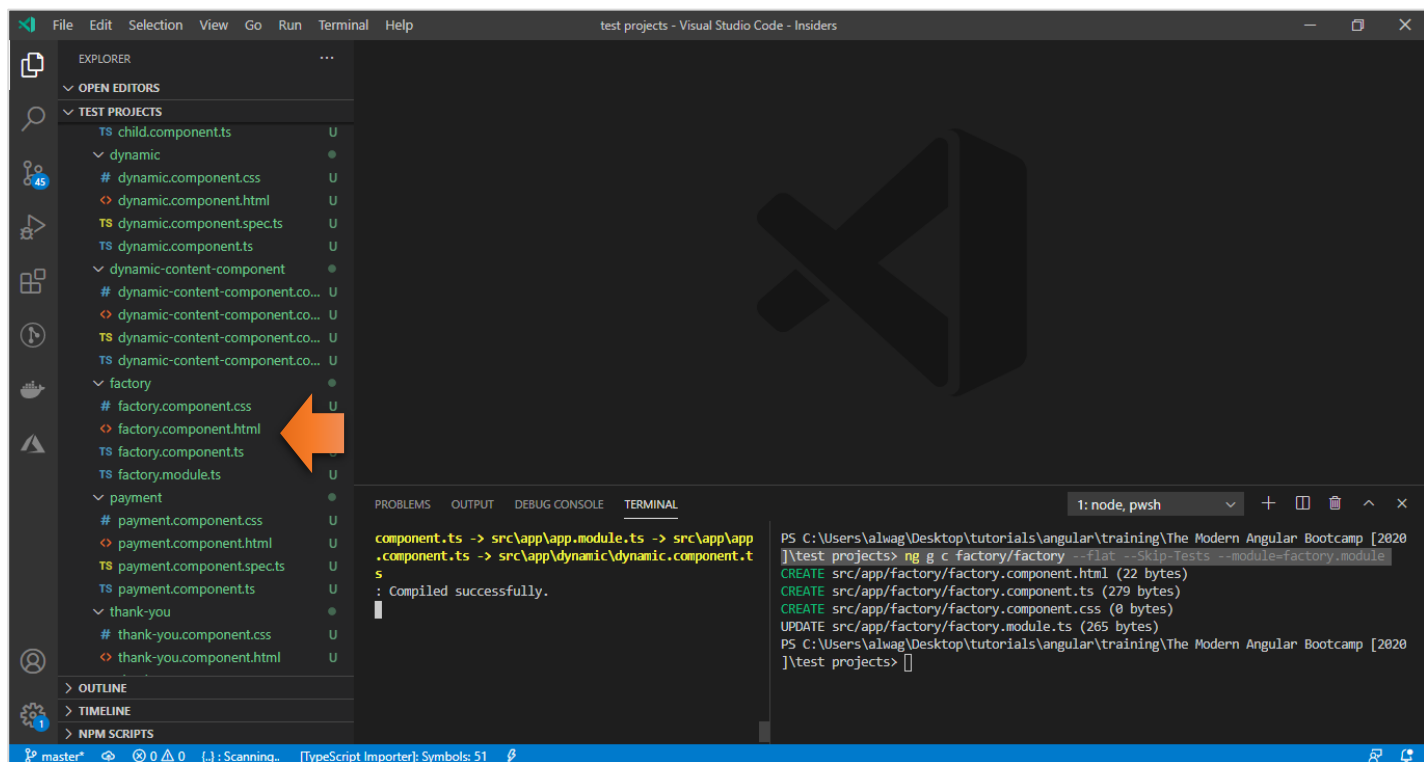
ثم نضغط Enter من لوحة المفاتيح، بحيث يتم انشاء هذا Module، وتسجيله في app.module.ts الرئيسي، كالتالي:



نلاحظ انشأ لنا مجلد باسم factory وبداخله module يحمل نفس الاسم، والآن لنقم بإضافة component ولكن بدلاً من ان يتم تسجيله في app.module.ts الرئيسي فأنا سوف نقوم بتسجيله في factory.module.ts، ونستطيع القيام بجميع هذه الأمور بشكل سريع عن طريق كتابة الأمر التالي:



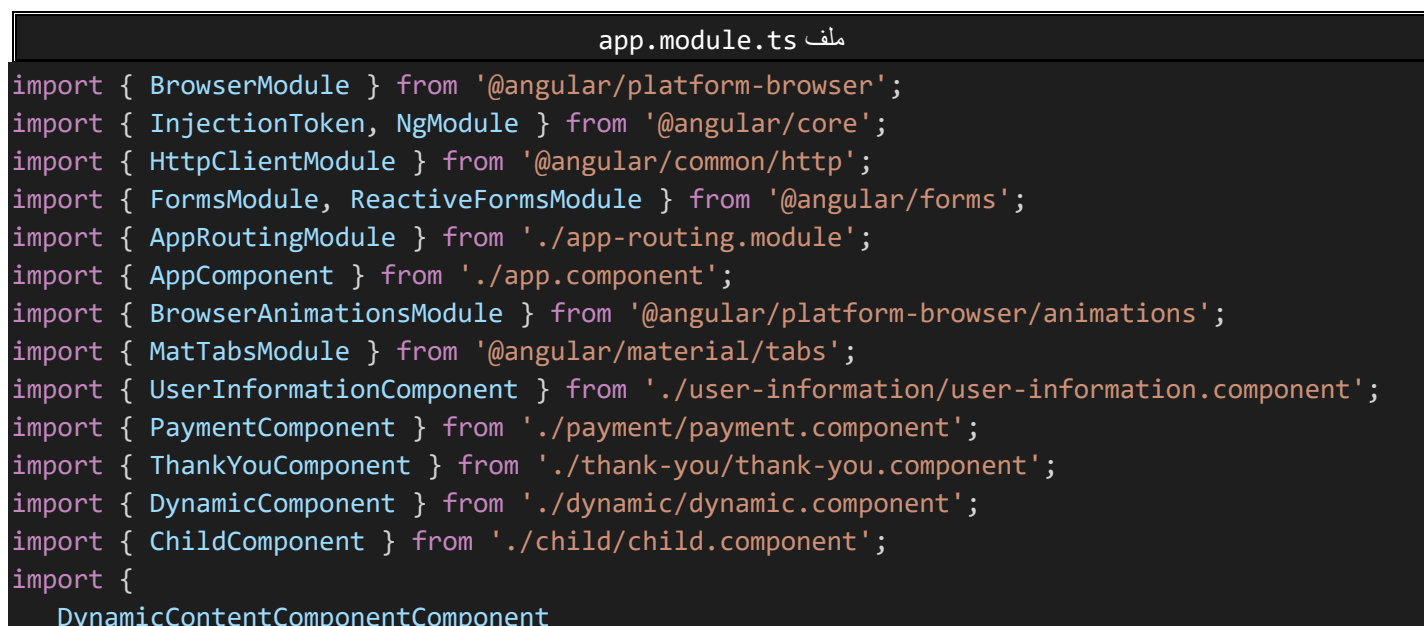
وبعدها نضغط Enter من لوحة المفاتيح، وسوف يتم انشاء هذا component مع تسجيله في factory.module.ts، كالتالي:



ولو اطلعنا على هذا module لوجدنا انه تم تسجيل هذا component بشكل تلقائي، كالتالي:



وبنفس الوقت لو اطلعنا على ملف app.module.ts فسوف نجد ان هذا module تم تسجيله ايضاً، كالتالي:



```

} from './dynamic-content-component/dynamic-content-component.component';
import { FactoryModule } from './factory/factory.module';

export const NAME = new InjectionToken<string>('');

@NgModule({
  declarations: [
    AppComponent,
    UserInformationComponent,
    PaymentComponent,
    ThankYouComponent,
    DynamicComponent,
    ChildComponent,
    DynamicContentComponentComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    BrowserAnimationsModule,
    MatTabsModule,
    FactoryModule
  ],
  providers: [{ provide: NAME, useValue: '' }],
  bootstrap: [AppComponent],
})
export class AppModule { }

```

ولنضيف بعض الكود البسيط في ملف factory.component.html، كالتالي:

ملف factory.component.html

```
<h3>Hi From Another Module</h3>
```

والنقطة المهمة هي في الجزء القادم حيث سوف نقوم بقراءة هذا Module بشكل ديناميكي، ومن ثم نعرض component الذي يحتويه، كالتالي:

ملف app.component.ts

```

import {
  Compiler,
  Component,
  Injector,
  NgModuleFactory,
  ReflectiveInjector
} from '@angular/core';
import { PaymentComponent } from './payment/payment.component';
import { ThankYouComponent } from './thank-you/thank-you.component';
import { UserInformationComponent } from './user-information/user-information.component';
import { ChildComponent } from './child/child.component';

```



```

import { DynamicComponent } from './dynamic/dynamic.component';
import { NAME } from './app.module';
import {
    DynamicContentComponentComponent
} from './dynamic-content-component/dynamic-content-component.component';
import { FactoryComponent } from './factory/factory.component';
import { FactoryModule } from './factory/factory.module';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
})

export class AppComponent {
    public dynamicComponent = null;
    public customInjector: Injector;
    public dynamicComponentWithInjector = null;
    public outlet = null;
    public factoryComponent = null;
    public customModule: NgModuleFactory<any>;
    public dynamicTabs = [
        {
            label: 'User Information',
            component: UserInformationComponent
        },
        {
            label: 'Payment',
            component: PaymentComponent
        },
        {
            label: 'Thank You',
            component: ThankYouComponent
        }
    ];

    constructor(private injector: Injector, private compiler: Compiler) {

    }

    generateComponent() {
        this.dynamicComponent = ChildComponent;
    }

    createInjector() {
        const name = 'Faisal';
        // tslint:disable-next-line: deprecation
        this.customInjector = ReflectiveInjector.resolveAndCreate(
            [{ provide: NAME, useValue: name }],
            this.injector
        );
        this.dynamicComponentWithInjector = DynamicComponent;
    }
}


```

```

createDynamicComponentWithContent() {
  this.outlet = DynamicContentComponentComponent;
}

createDynamicComponentFromAnotherModule() {
  this.customModule = this.compiler.compileModuleSync(FactoryModule);
  this.factoryComponent = FactoryComponent;
}
}

```



وفي ملف template نضيف التالي:

ملف app.component.html

```

<mat-tab-group>
  <mat-tab *ngFor="let tab of dynamicTabs">
    <ng-template mat-tab-label>{{ tab.label }}</ng-template>
    <ng-container *ngComponentOutlet="tab.component"></ng-container>
  </mat-tab>
</mat-tab-group>

<hr />
<hr />

<button (click)="generateComponent()">Render Dynamic Component</button>
<ng-container [ngComponentOutlet]="dynamicComponent"></ng-container>

<hr />
<hr />

<button (click)="createInjector()">
  Render Dynamic Component With Injector
</button>
<ng-container
  *ngComponentOutlet="dynamicComponentWithInjector; injector: customInjector"
>
</ng-container>

<hr />
<hr />

<button (click)="createDynamicComponentWithContent()">
  Render Dynamic Component With Content
</button>
<ng-container *ngComponentOutlet="outlet; content: [[first], [second]]">
</ng-container>
<div #first [hidden]="!outlet">
  <h1>CONTENT1</h1>
  <h4>content1 text</h4>
</div>
<div #second [hidden]="!outlet">
  <h1>CONTENT2</h1>
  <h4 id="content2">content2 text</h4>

```

```

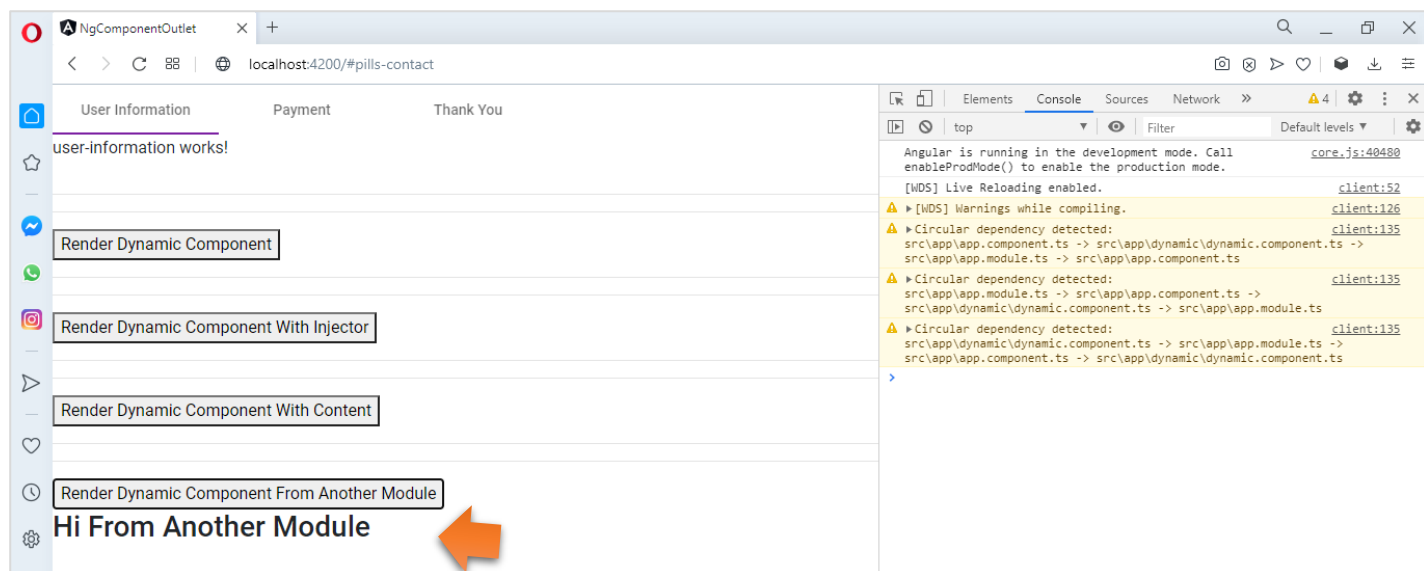
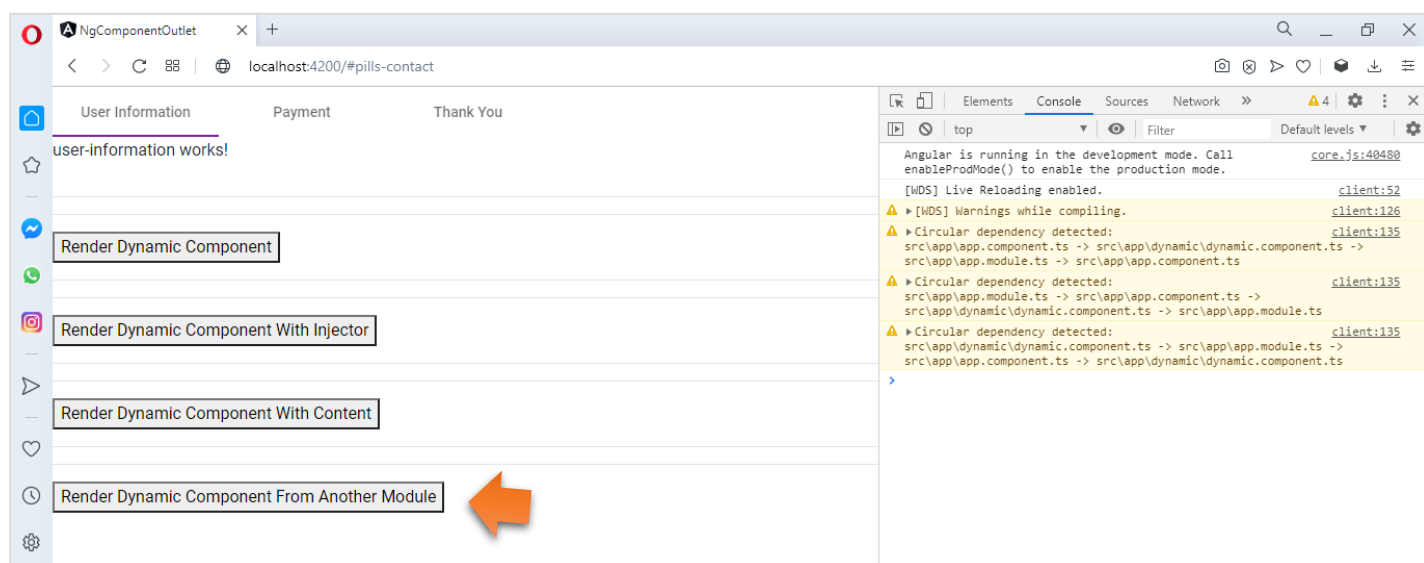
</div>

<hr />
<hr />

<button (click)="createDynamicComponentFromAnotherModule()">
  Render Dynamic Component From Another Module
</button>
<ng-container>
  *ngComponentOutlet="factoryComponent; ngModuleFactory: customModule"
>
</ng-container>

```

والآن لنشاهد النتيجة في المتصفح، كالتالي:



مع العلم انه في الإصدارات الحديثة من Angular لا نحتاج إلى هذه الخطوات جميعاً فإن Angular يمتلك من الذكاء ما يجعله يعمل بشكل تلقائي أن هذا component ينتهي إلى أي module، فقط نمرر اسم component كبارامتر اول لي \*ngComponentOutlet، ونترك الباقي لي Angular لكي يقوم بعرض هذا component لنا.

## 12.4 .Inheritance Components:

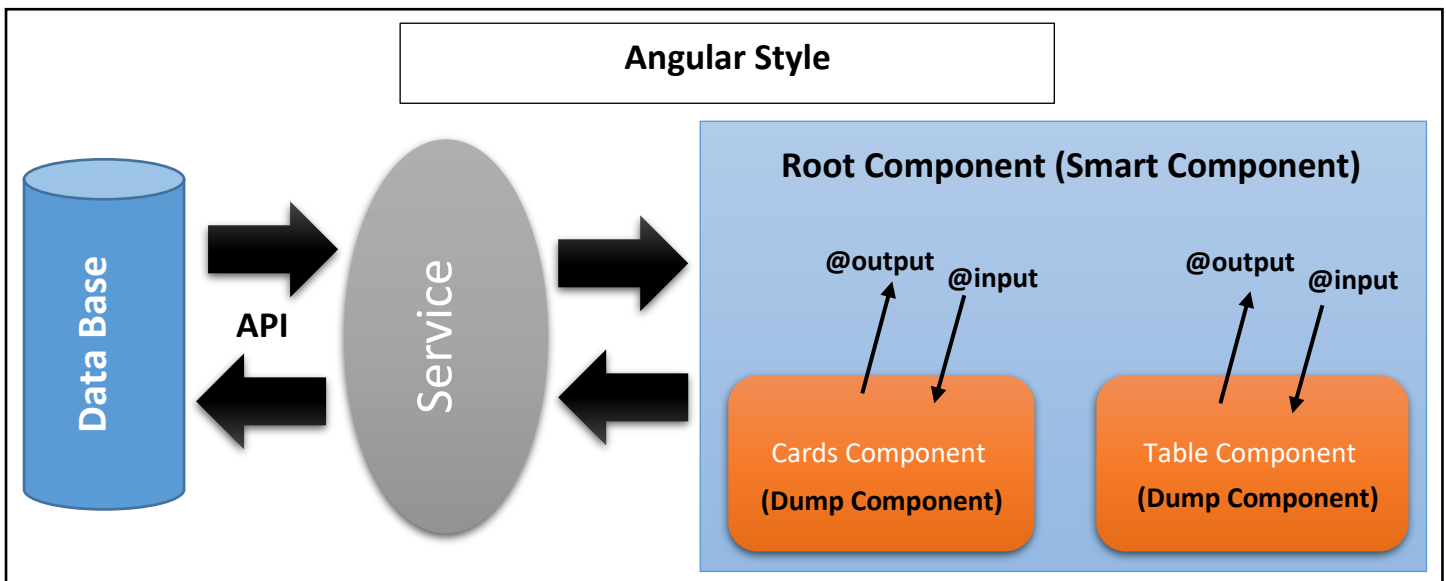
بما أن أي components بالأساس هو عبارة عن class وبما أن إطار عمل Angular يعتمد على لغة Typescript والتي هي superset لـ JavaScript والتي هي أيضاً تدعم مفاهيم البرمجة الكائنية OOP، فلذلك نستطيع تطبيق والاستفادة من هذه المفاهيم في Components الخاصة بنا ومنها مفهوم الوراثة Inheritance.

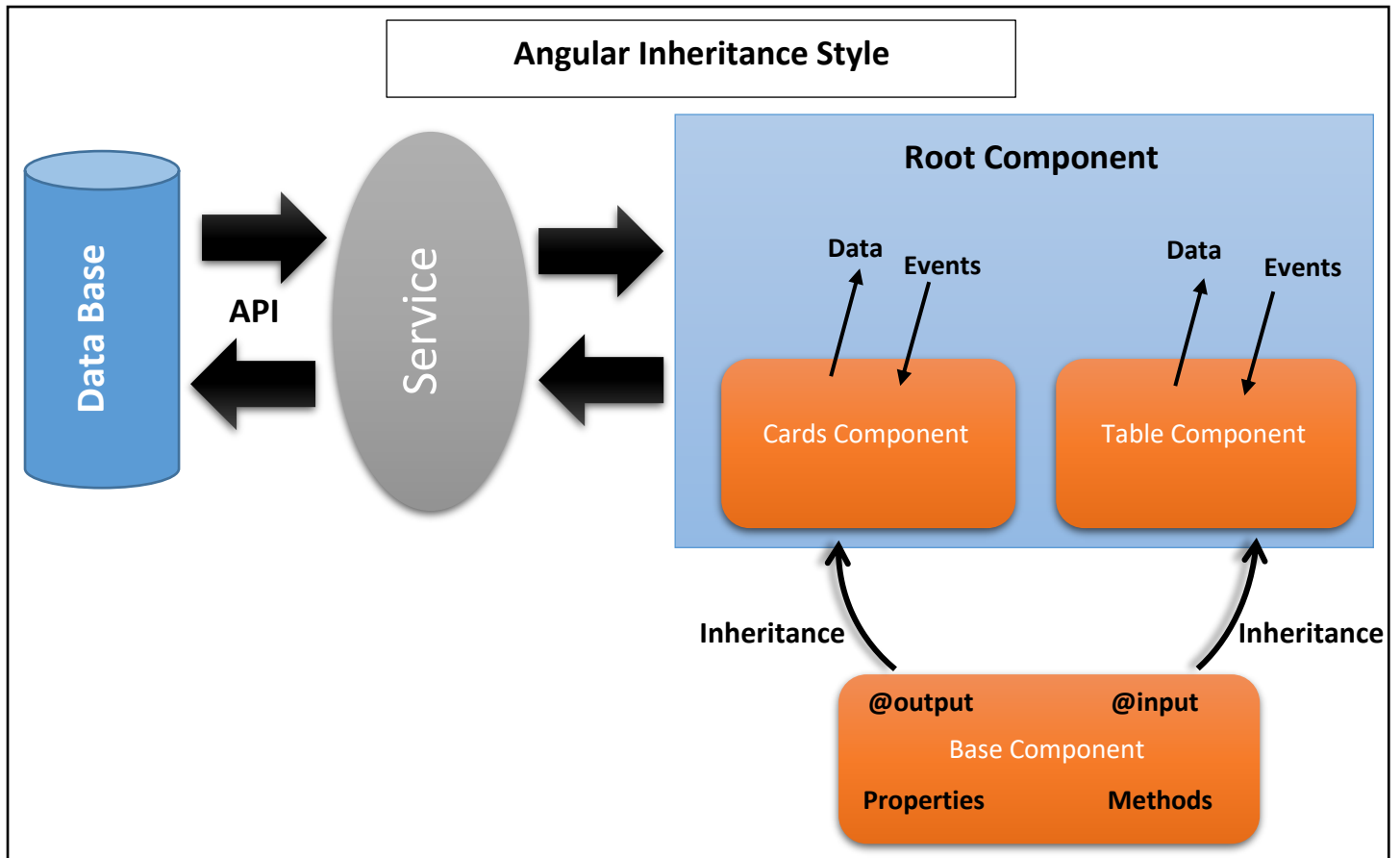
وعند تطبيق مفهوم الوراثة على components في angular يجب الانتباه إلى مجموعة من النقاط، هي:

- الذي يتم توريثه هو فقط logic الموجود في ملف class كالدوال (methods) او الخصائص (Properties) بشرط ان يكون تم تعريفها على انها public وليس private، او أي بيانات داخلية عن طريق @Input() او احداث خارجة عن طريق @Output()، ولا يتم توريث أي Markup في ملف template او أكواد css في ملف style.
- ليس كل Logic في ملف class يتم توريثه حيث ان جميع دوال lifecycle Hooks لا يتم توريثها وايضاً الخصائص في Decorator ذو الاسم @Component({}) مثل selector او templateUrl او Providers لا يتم توريثها.

اما صور استخدام الوراثة بين components المختلفة قد تكون محدودة نوعاً ما، لأن إطار عمل Angular قدم لنا كمطورين طرق متعددة لتقليل تكرار الأكواد وتنظيمها، ولكن بما أن Angular يُتيح لنا القيام بنفس الأمر بطرق متعددة فنستطيع استخدام الوراثة لتقليل تكرار Logic المتشابهة بين هذه components فنستطيع عندئذٍ ان ننشأ component يحتوي على جميع logic المتشابهة ونجعل أي component يرث هذا Logic منه، وبالتأكيد يعمل override او يضيف logic آخر خاص به بحسب الاحتياج.

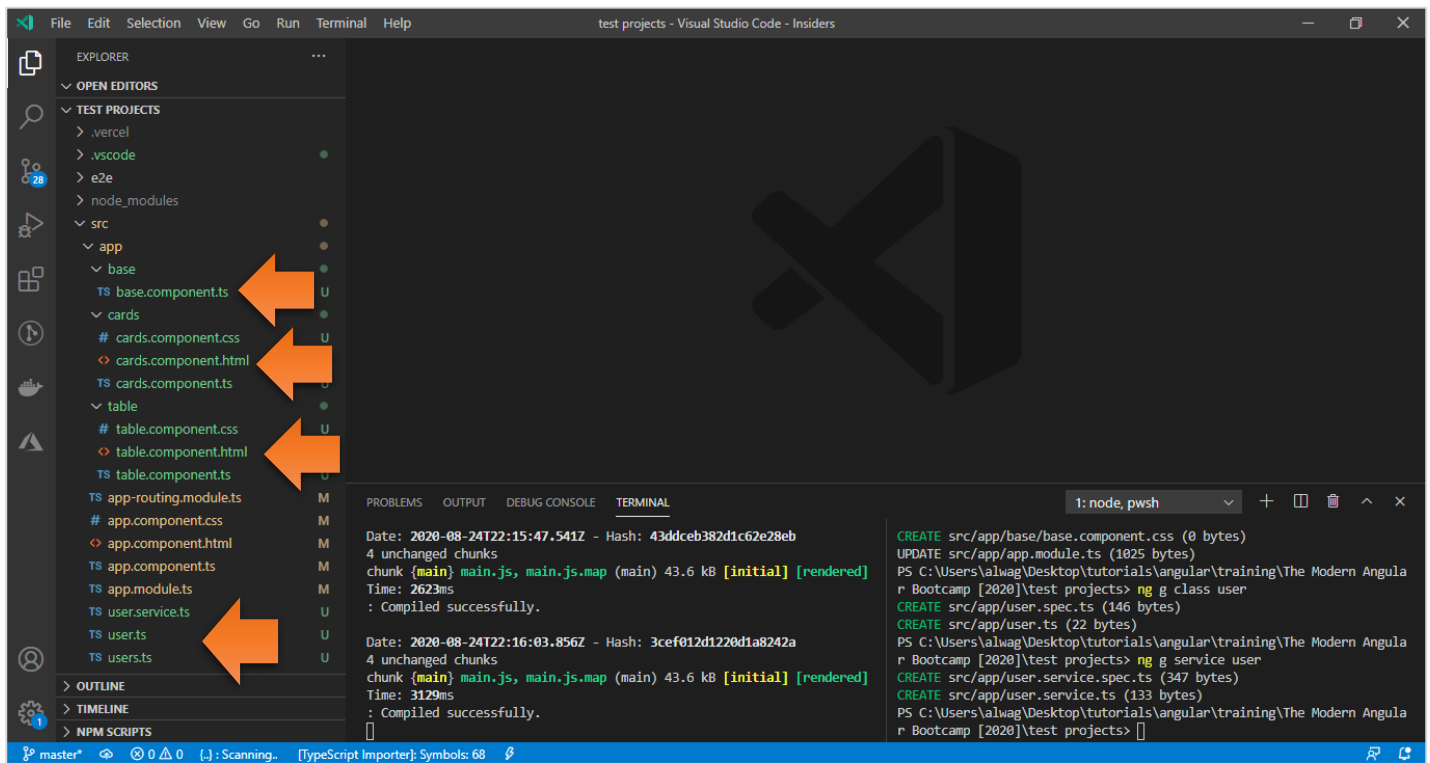
ولتوضيح لنقوم بإعطاء مثال، ولنفرض انه لدينا component باسم base حيث يحتوي على Logic الذي نريد توريثه مثل @Input و @Output و خاصية ودالة معينة، ولدينا ايضاً اثنين components أخرى تقوم بالوراثة من هذا component ذو الاسم base وليكن اسمهما cards و table، وبما ان base مهمته فقط التوريث وب نفس الوقت قلنا سابقاً انه يتم توريث ملف class فقط لذلك ليس هنالك داعي لوجود ملفي template و style لهذا base component، ويمكن تمثيل استخدام الوراثة مقابل استخدام الطرق التي تقدمها Angular لبناء هذا التطبيق بالأشكال التالية:





كما هو واضح في الشكل الأول استخدمت طريقة Smart/Dump Components لبناء التطبيق حيث لكل component نمرة input و output الخاص به و بنفس الوقت نمرة له البيانات Data التي تمثل input والاحداث Events التي تمثل output، بعكس الطريقة الثانية باستخدام الوراثة، حيث يوجد Component رئيسي واسمته Base حيث يتم تعريف جميع inputs و outputs والخصائص والدوال...الخ، اما بالنسبة لي components التي ترث هذه Functionality فإننا نمرة لها البيانات والاحداث فقط، لأن input و output قد ورثوها من Base Component.

والآن لنقوم بترجمة هذا الشكل بطريقة عملية، وفي البداية لننشئ ثلاثة components الأول باسم base والثاني باسم cards والثالث باسم table، و بنفس الوقت لنقوم بإنشاء service لنحاكي تخاطبها مع قاعدة البيانات وملف اسمته users.ts قمت بتخزين بعض البيانات لكي نحكي الاتصال بقاعدة البيانات، وأخيراً ملف user.ts لوصف هذه البيانات، كالتالي:



ولو لاحظنا ان Base Component قد قمت بحذف ملفات template و style الخاص به لأننا لا نحتاجها هنا ولكن ليس هذا الأمر الزامياً فإذا احتجت وجودهما لأي سبب من الأسباب فنبقى عليهما.

اما الملفات الثلاثة الأخيرة فقد اشرت لها سابقاً، واما محتوياتها، كالتالي:

#### ملف user.ts

```
export class User {
  id: string;
  firstName: string;
  lastName: string;
  email: string;
  address: {
    street: string;
    suite: string;
    city: string;
    zipcode: string;
  };
  phone: string;
}
```

#### ملف users.ts

```
import { User } from './user';

export const USERS: User[] = [
  {
    id: '1',
    firstName: 'Leanne',
    lastName: 'Graham',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
```

```

    suite: 'Apt. 556',
    city: 'Gwenborough',
    zipcode: '92998-3874'
  },
  phone: '1-770-736-8031 x56442'
},
{
  id: '2',
  firstName: 'Ervin',
  lastName: 'Howell',
  email: 'Shanna@melissa.tv',
  address: {
    street: 'Victor Plains',
    suite: 'Suite 879',
    city: 'Wisokyburgh',
    zipcode: '90566-7771'
  },
  phone: '010-692-6593 x09125'
},
{
  id: '3',
  firstName: 'Clementine',
  lastName: 'Bauch',
  email: 'Nathan@yesenia.net',
  address: {
    street: 'Douglas Extension',
    suite: 'Suite 847',
    city: 'McKenziehaven',
    zipcode: '59590-4157'
  },
  phone: '1-463-123-4447'
},
{
  id: '4',
  firstName: 'Patricia',
  lastName: 'Lebsack',
  email: 'Julianne.OConner@kory.org',
  address: {
    street: 'Hoeger Mall',
    suite: 'Apt. 692',
    city: 'South Elvis',
    zipcode: '53919-4257'
  },
  phone: '493-170-9623 x156'
},
{
  id: '5',
  firstName: 'Chelsey',
  lastName: 'Dietrich',
  email: 'Lucio_Hettinger@annie.ca',
  address: {
    street: 'Skiles Walks',
    suite: 'Suite 351',

```

```

        city: 'Roscoeview',
        zipcode: '33263'
    },
    phone: '(254)954-1289'
}
];

```

#### ملف user.service.ts

```

import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { User } from '../user';
import { USERS } from '../users';

@Injectable({
  providedIn: 'root'
})

export class UserService {

  constructor() { }

  getUsers(): Observable<User[]> {
    return of(USERS.map((user) => Object.assign(new User(), user)));
  }
}

```

اما محتويات ملف base.component.ts كما اشرنا سابقاً نعرف فيها جميع Logic الذي نريد توريثها، كالتالي:

#### ملف base.component.ts

```

import { Component, OnInit } from '@angular/core';
import { Input, Output, EventEmitter } from '@angular/core';
import { User } from '../user';
@Component({
  selector: 'app-base',
  template: '',
  styles: ['']
})
export class BaseComponent implements OnInit {
  public name: string;

  @Input() users: User[];
  @Output() selected: EventEmitter<User> = new EventEmitter<User>();

  constructor() { }

  ngOnInit(): void { }

  public selectUser(user: User): void {
    this.selected.emit(user);
  }
}

```



نلاحظ قمنا بتعريف خاصية باسم name وعرفنا مدخل @Input() باسم users وعرفنا ايضاً @Output() اسميته selected واخيراً دالة Methods باسم selectUser تقوم بأرسال user جديد عن طريق selected باستخدام @output.

بطبيعة الحال جميع هذه Functionality مجرد تعريفات وليس لها بيانات بمعنى ليس هنالك بيانات لي users ... الخ. وكما قلنا سابقاً ان هذه البيانات نمررها لي components التي ترث هذا Logic، والآن لنقوم بعمل وراثه في كلا components، كالتالي:

```
cards.component.ts ملف
import { Component, OnInit } from '@angular/core';
import { BaseComponent } from '../base/base.component';

@Component({
  selector: 'app-cards',
  templateUrl: './cards.component.html',
  styleUrls: ['./cards.component.css']
})
export class CardsComponent extends BaseComponent implements OnInit {
  name = 'Cards Component';

  constructor() {
    super();
  }

  ngOnInit(): void {
  }
}
```

نلاحظ علمنا extends أي وراثه من BaseComponent وبنفس الوقت علمنا override للخاصية name التي تم وراثتها، وبذلك اصبح هذا component يمتلك جميع Functionality، ولتأكد لنقم بقراءة @Input() ذو الاسم users في ملف template لهذا component، كالتالي:

```
cards.component.html ملف
<div class="panel panel-default" *ngFor="let user of users">
  <div class="panel-body">
    <div> <strong>User ID</strong> {{ user.id }} </div>
    <div> <strong>First Name:</strong> {{ user.firstName }} </div>
    <div> <strong>Last Name:</strong> {{ user.lastName }} </div>
    <div> <strong>Email:</strong> {{ user.email }} </div>
    <div>
      <button class="btn btn-success" (click)="selectUser(user)">
        Select
      </button>
    </div>
  </div>
</div>
```

نلاحظ قمنا بقراءة الخاصية users على الرغم اننا لم نقوم بتعريفها في ملف class لهذا component وبنفس الوضع بالنسبة للدالة selectUser، ولم يظهر لنا أي رسالة خطأ والسبب ان هذه الخصائص قد ورثها من BaseComponent.

وأعيد وأكرر انه لو قمنا بتشغيل التطبيق فلن تظهر أي بيانات لأننا لم نقوم بتمرير البيانات إلى هذا component الوارث، الذي قمنا بفعله إلى الآن هو تعريف Logic وورائته فقط.

وأخيراً بالنسبة لهذا component لنقم بإضفاء لمسة جمالية بسيطة عن طريق إضافة بعض الأكواد في ملف style، كالتالي:

ملف cards.component.css

```
.panel {
  margin-bottom: 20px;
  padding: 10px;
  -webkit-box-shadow: -1px 0px 13px 1px rgba(0, 0, 0, 0.18);
  -moz-box-shadow: -1px 0px 13px 1px rgba(0, 0, 0, 0.18);
  box-shadow: -1px 0px 13px 1px rgba(0, 0, 0, 0.18);
}
```

ونفس الوضع لي component الآخر table، كالتالي:

ملف table.component.ts

```
import { Component, OnInit } from '@angular/core';
import { BaseComponent } from '../base/base.component';

@Component({
  selector: 'app-table',
  templateUrl: './table.component.html',
  styleUrls: ['./table.component.css']
})

export class TableComponent extends BaseComponent implements OnInit {

  name = 'Table Component';

  constructor() {
    super();
  }

  ngOnInit(): void {
  }

}
```

ملف table.component.html

```
<div class="table-responsive">
  <table class="table table-striped table-hover table-bordered">
    <thead>
      <tr>
        <th>User ID</th>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
      </tr>
    </thead>
```

```

<tbody>
  <tr *ngFor="let user of users">
    <td>{{ user.id }}</td>
    <td>{{ user.firstName }}</td>
    <td>{{ user.lastName }}</td>
    <td>{{ user.email }}</td>
    <td>
      <button class="btn btn-success" (click)="selectUser(user)">
        Select
      </button>
    </td>
  </tr>
</tbody>
</table>
</div>

```

بعدما قمنا بتجهيز جميع Functionality والوراثة، بقي أخيراً ان نقوم بتمرير البيانات إلى هذين components، وسوف نضيف selector الخاص بهما إلى ملف template في Root Component، كالتالي:

ملف app.component.ts

```

import { Component, ViewChild, OnInit, OnDestroy } from '@angular/core';
import { User } from './user';
import { Subscription } from 'rxjs';
import { UserService } from './user.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit, OnDestroy {
  private getUsers: Subscription;
  public users: User[];
  public selectedUser: User;

  constructor(private userService: UserService) { }

  ngOnInit() {
    this.getUsers = this.userService.getUsers()
      .subscribe((users: User[]) => this.users = users);
  }

  ngOnDestroy() {
    if (this.getUsers) {
      this.getUsers.unsubscribe();
    }
  }

  setSelectedUser(user: User): void {
    this.selectedUser = user;
  }
}

```

كما نلاحظ قمنا بقراءة البيانات عن طريق service عن طريق الدالة ngOnInit و خزناها في المصفوفة users وبالتالي أصبحت هذه المصفوفة تمثل البيانات التي سوف نقوم بتمريرها إلى كلا components سواء cards او table، وأخيراً نكتب محتوى الدالة setSelectedUser والتي تمثل الحدث event ذو الاسم selected والذي هو الآخر عبارة عن @output، ولتوضيح لئلا نرى هذا الحدث في ملف template، كالتالي:

```
ملف app.component.html
<div class="container">
  <div class="alert alert-success" *ngIf="selectedUser">
    <pre>
      {{ selectedUser | json }}
    </pre>
  </div>

  <app-table
    #table
    [users]="users"
    (selected)="setSelectedUser($event)"
  ></app-table>

  <app-cards
    #cards
    [users]="users"
    (selected)="setSelectedUser($event)"
  ></app-cards>
</div>
```

نلاحظ في السهمين الثاني والثالث مررنا البيانات لي components الوارثين، سواء كان input عن طريق users او الحدث output عن طريق selected، اما السهم الأول فيعرض بيانات هذا المستخدم عند الضغط عليه، ولكن قبل استعراض النتيجة في المتصفح لنقوم بإكمال باقي أجزاء التطبيق لكي يظهر بشكله النهائي، كالتالي:

```
ملف app.component.ts
import { Component, ViewChild, OnInit, OnDestroy } from '@angular/core';
import { User } from './user';
import { Subscription } from 'rxjs';
import { BaseComponent } from './base/base.component';
import { UserService } from './user.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit, OnDestroy {
  private getUsers: Subscription;

  public currentTab: string;
  public users: User[];
  public selectedUser: User;
```

```

public componentName: string;

@ViewChild('table') table: BaseComponent;
@ViewChild('cards') cards: BaseComponent;

constructor(private userService: UserService) { }

ngOnInit() {
  this.getUsers = this.userService.getUsers()
    .subscribe((users: User[]) => this.users = users);
  this.switchTo('table');
}

ngOnDestroy() {
  if (this.getUsers) {
    this.getUsers.unsubscribe();
  }
}

switchTo(tab: string): void {
  this.currentTab = tab;
  this.selectedUser = null;

  setTimeout(() => { // Give the view some time to render
    switch (this.currentTab) {
      case 'table':
        this.currentComponentName = this.table.name;
        break;
      case 'cards':
        this.currentComponentName = this.cards.name;
        break;
    }
  });
}

setSelectedUser(user: User): void {
  this.selectedUser = user;
}
}

```



الذي اريد التنويه عليه هو ما تم التأشير عليه بالسهمين، حيث قمنا بقراءة الخاصية name التي علمنا لها override في كلا components الوارثين.

ملف app.component.html

```

<div class="container">
  <div class="btn-group" role="group">
    <button
      type="button"
      class="btn btn-default"
      [class.btn-primary]="currentTab === 'table'"
      (click)="switchTo('table')"
    >

```

```

    >
      Table
    </button>
    <button
      type="button"
      class="btn btn-default"
      [class.btn-primary]="currentTab === 'cards'"
      (click)="switchTo('cards')"
    >
      Cards
    </button>
  </div>
  <div class="alert alert-success" *ngIf="selectedUser">
    <pre>
      {{ selectedUser | json }}
    </pre>
  </div>
  <h3>View a list of users using the {{ currentComponentName }}</h3>

  <app-table
    #table
    *ngIf="currentTab === 'table'"
    [users]="users"
    (selected)="setSelectedUser($event)"
  ></app-table>

  <app-cards
    #cards
    *ngIf="currentTab === 'cards'"
    [users]="users"
    (selected)="setSelectedUser($event)"
  ></app-cards>
</div>

```

ولنضيف بعض اللمسات الجمالية في ملف style، كالتالي:

ملف app.component.css

```

.container {
  margin-top: 1rem;
}
.alert {
  margin-top: 1rem;
}
pre {
  background: black;
  color: white;
  border-radius: 5px;
  margin: 0px;
}
h3 {
  margin: 20px;
}

```

The screenshot shows a web application running on localhost:4200. The application has a sidebar with a 'Table' button selected. The main content area displays a heading 'View a list of users using the Table Component' and a table with 5 rows of user data. Each row has a 'Select' button. The console on the right shows messages from Angular and Webpack Dev Server (WDS).

User ID	First Name	Last Name	Email
1	Leanne	Graham	Sincere@april.biz
2	Ervin	Howell	Shanna@melissa.tv
3	Clementine	Bauch	Nathan@yesenia.net
4	Patricia	Lebsack	Julianne.OConner@kory.org
5	Chelsey	Dietrich	Lucio_Hettinger@annie.ca

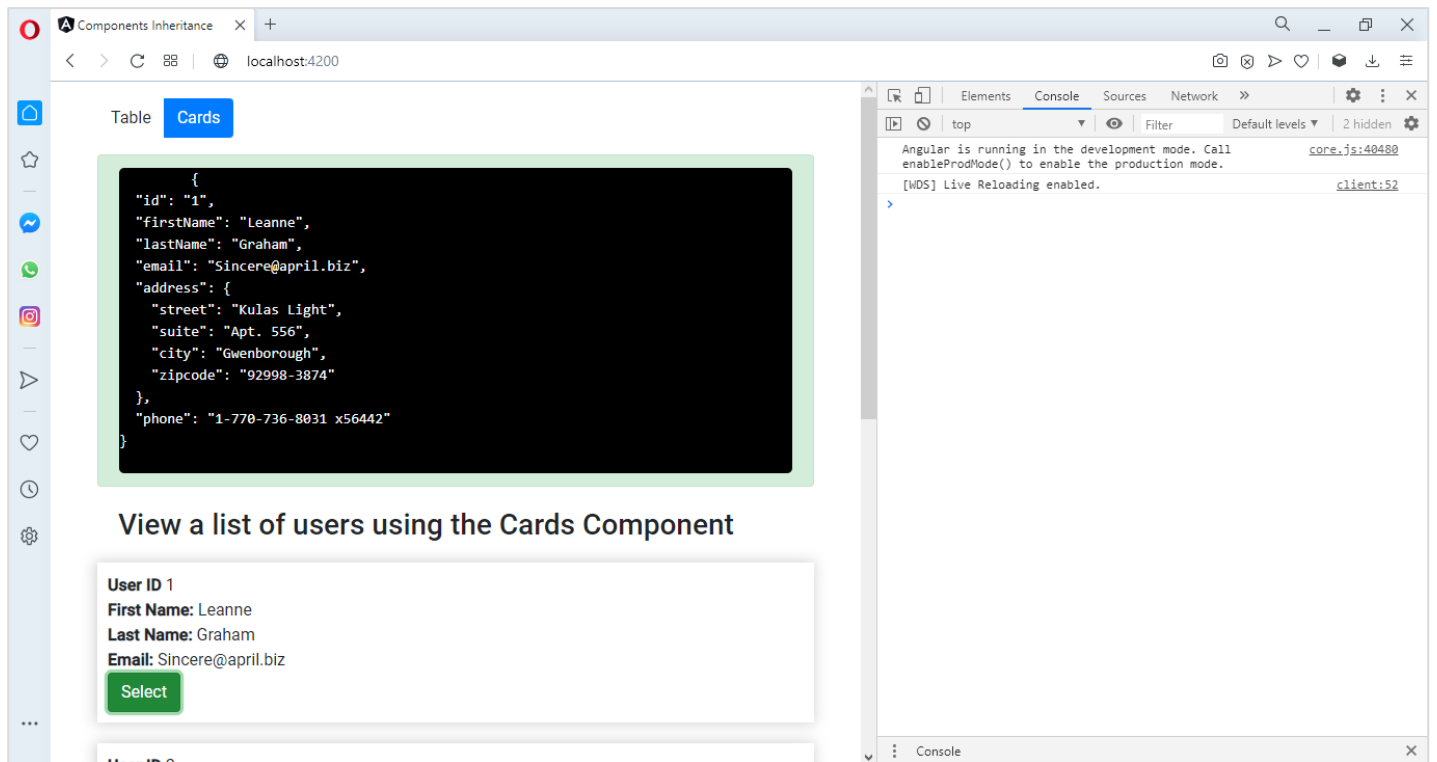
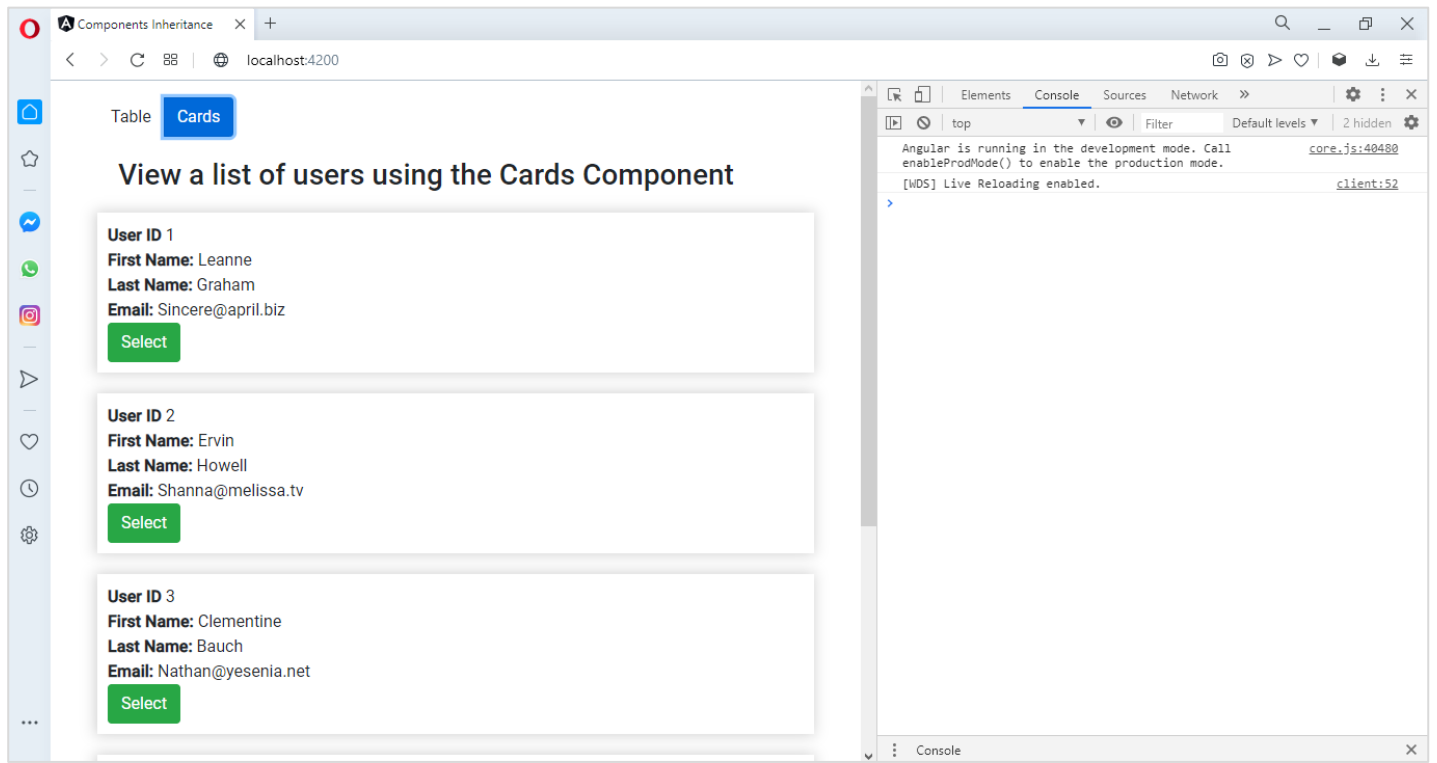
The screenshot shows the same web application, but with a JSON object highlighted in a green box. The JSON object represents the user data for the second row of the table. The table below shows the updated data, with the second row now highlighted in green.

```

{
  "id": "2",
  "firstName": "Ervin",
  "lastName": "Howell",
  "email": "Shanna@melissa.tv",
  "address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771"
  },
  "phone": "010-692-6593 x09125"
}

```

User ID	First Name	Last Name	Email
1	Leanne	Graham	Sincere@april.biz
2	Ervin	Howell	Shanna@melissa.tv



نلاحظ اننا استفدنا من مبدأ الوراثة لعرض نفس البيانات وبنفس Logic ولكن بطريقتين مختلفتين الأولى عن طريق cards والآخر عن طريق table.

ملاحظة مهمة: مصدر المثال على هذا الرابط <https://stackblitz.com/edit/angular-jxbbaq?file=app%2Fapp.component.ts>



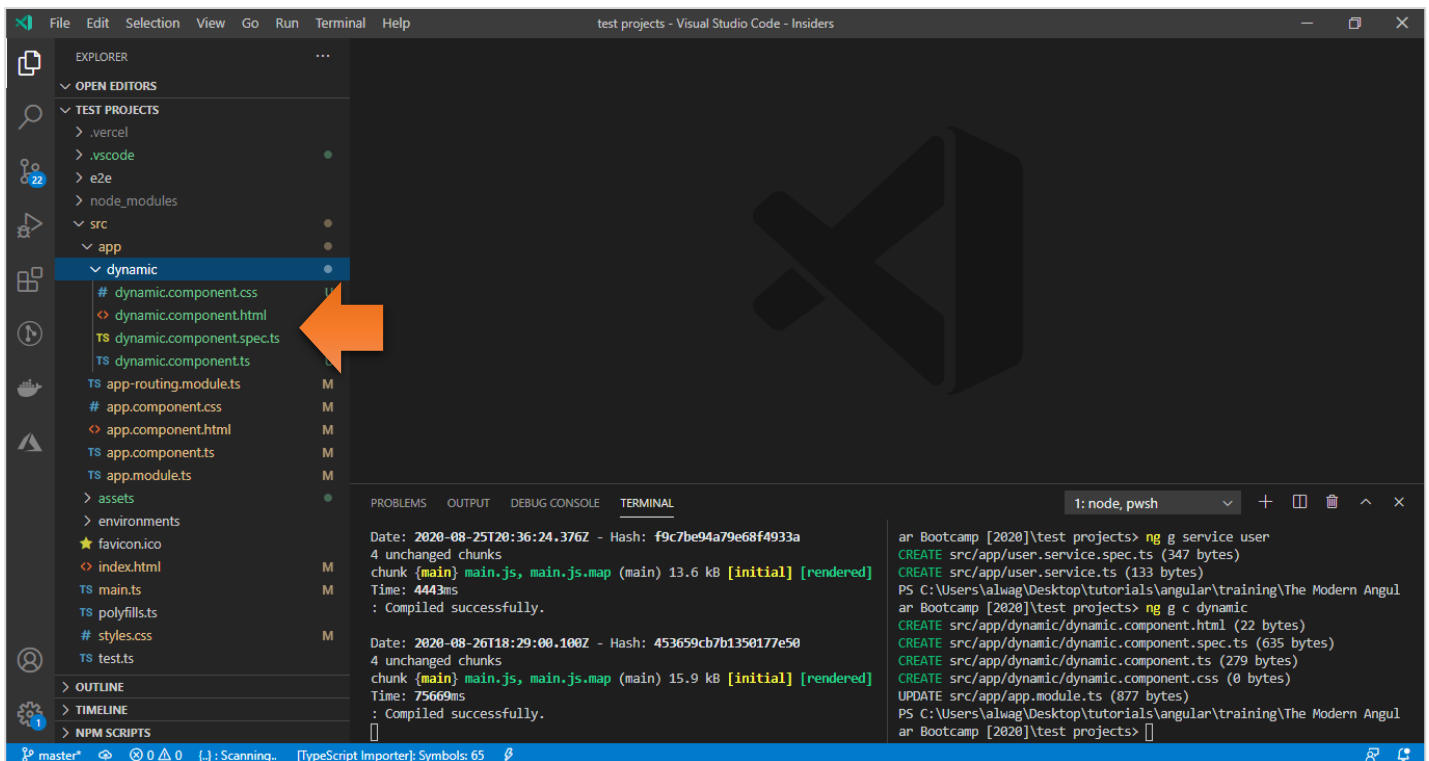
## 13.4. Dynamic Components

لقد تكلمنا سابقاً عن طريقة بناء components ديناميكية باستخدام `*ngComponentOutlet` حيث يتم عرضها ديناميكياً أثناء تشغيل التطبيق، وهنالك طرق أخرى وتعطي مرونة أكثر لتعامل مع components الديناميكية سواء من الاستخدامات البسيطة نفس النتائج التي نصل إليها عند استخدامنا `*ngComponentOutlet` إلى الاستخدامات المعقدة وهي بناء component من الصفر بشكل ديناميكي.

وسوف يكون الشرح على شكل امثلة بحيث كل مثال يشرح مفهوم معين وسوف نتدرج من الاستخدامات البسيطة إلى الوصول إلى بناء components ديناميكية بالكامل.

### 1.13.4. المثال الأول:

لنفرض انه لدينا component بسيط ونريد عرضه ديناميكياً عند الضغط على زر معين، وليكن اسمه dynamic، لذلك اول خطوة هي انشاء هذا component كالتالي:



وتكمن الفكرة انه بدلاً من استخدام `*ngComponentOutlet` كما في السابق، هنا سوف نستخدم اثنين services تحتوي على مجموعة من الأوامر والدوال البرمجية التي تُعطي للمطور مرونة وإمكانيات أكثر لتعامل مع Dynamic Components، وهذين services هما `ComponentFactoryResolver` و `ViewContainerRef`، حيث أن الأولى نستطيع عن طريقها تجهيز كافة البيانات اللازمة عن هذا component الديناميكي عن طريق استخدام الدالة `resolveComponentFactory`، ومن ثم نستخدم service الثانية لبناء component عن طريق استخدام الدالة `createComponent`، لذلك لنقم بعمل inject لهذه services في component الأب الذي نريد ان يحتوي على هذا component الأبن، كالتالي:

ملف `app.component.ts`

```
import {
```

```

    Component,
    OnInit,
    ViewContainerRef,
    ComponentFactoryResolver
} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  constructor(private vcr: ViewContainerRef, private cfr: ComponentFactoryResolver) { }

  ngOnInit() { }
}

```

بعدما قمنا بعمل inject لنقم باستخدام cfr والذي يمثل ComponentFactoryResolver حيث انه يحتوي على دالة باسم resolveComponentFactory وتستقبل بارامتر واحد وهو component الذي نريد تهيئته وجلب بياناته، حيث سنقوم في البداية بعمل console لهذا الكائن ولنرى ما يحتويه، كالتالي:

ملف app.component.ts

```

import { Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

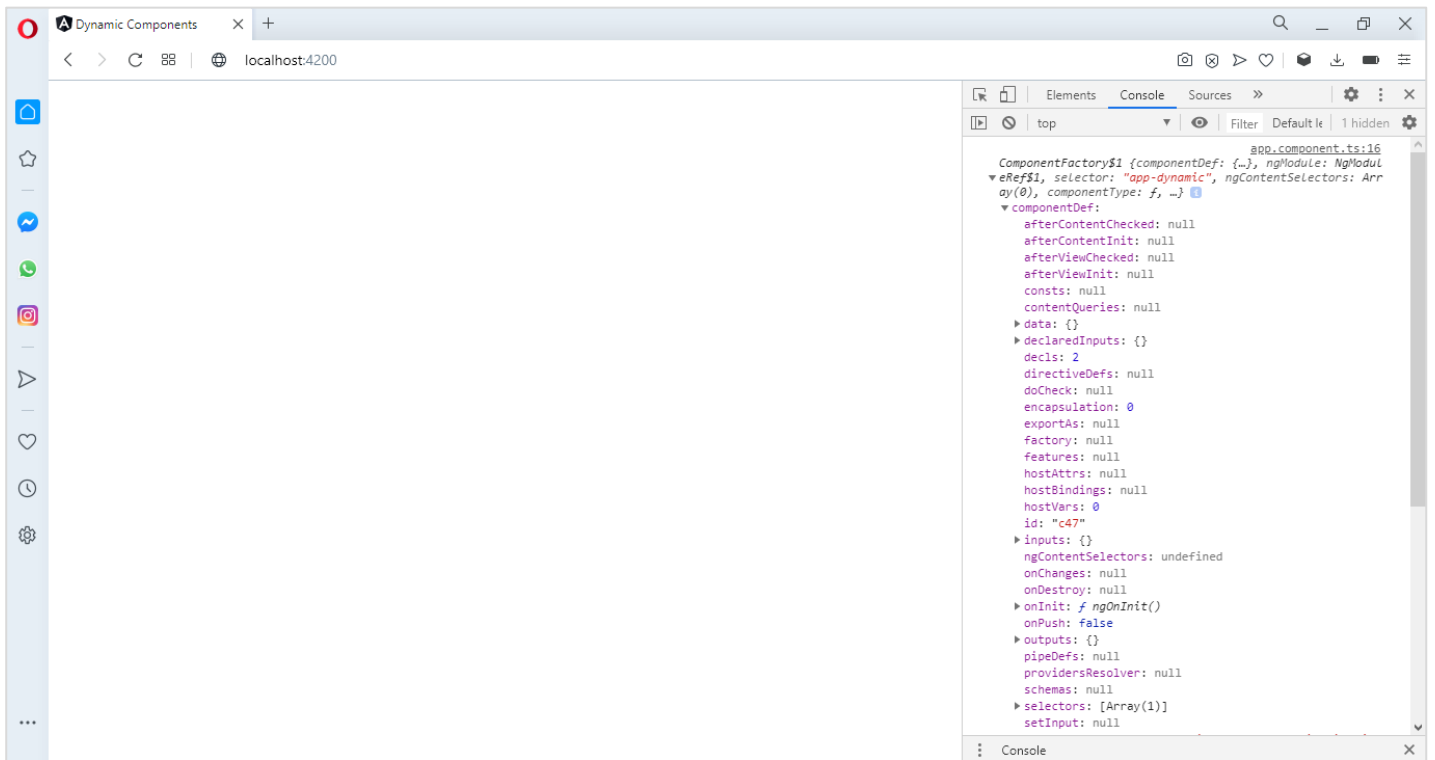
export class AppComponent implements OnInit {

  constructor(private vcr: ViewContainerRef, private cfr: ComponentFactoryResolver) { }

  ngOnInit() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    console.log(resolver);
  }
}

```

نلاحظ اننا قمنا بقراءة component الديناميكي، وقمنا بطباعته في console، اما النتيجة فتكون كالتالي:



نلاحظ على يمين الشاشة ظهرت لنا جميع الخصائص والدوال والبيانات الخاصة بي component الديناميكي ذو الاسم dynamic، وبذلك أصبح جاهز لإنشائه عن طريق ViewContainerRef وسوف نقوم بإنشاء دالة يتم تنفيذها عند الضغط على زر معين وتقوم بإنشاء هذا component، كالتالي:

```

ملف app.component.ts
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  constructor(private vcr: ViewContainerRef, private cfr: ComponentFactoryResolver) { }

  ngOnInit() {}

  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.vcr.createComponent(resolver);
  }
}

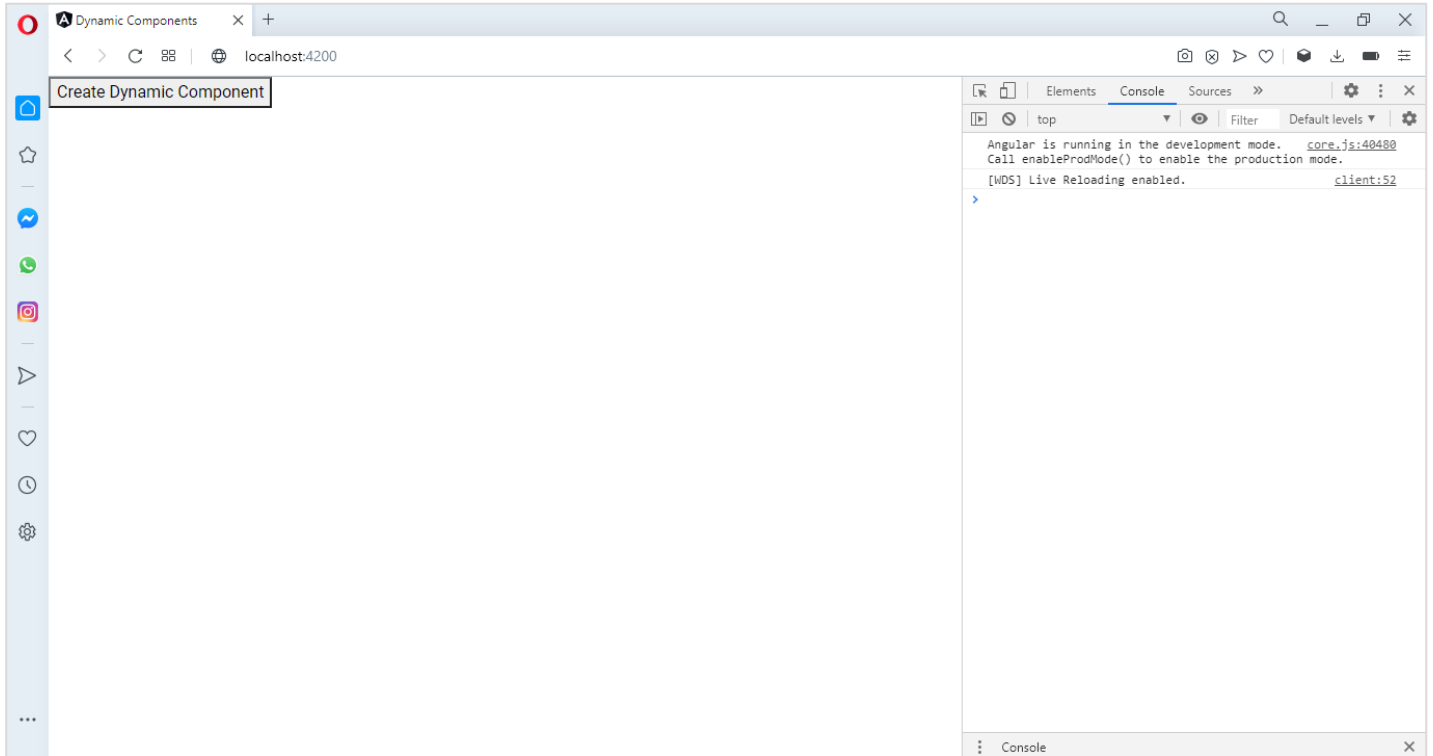
```

ولننشأ الزر ونضيف الدالة له، كالتالي:

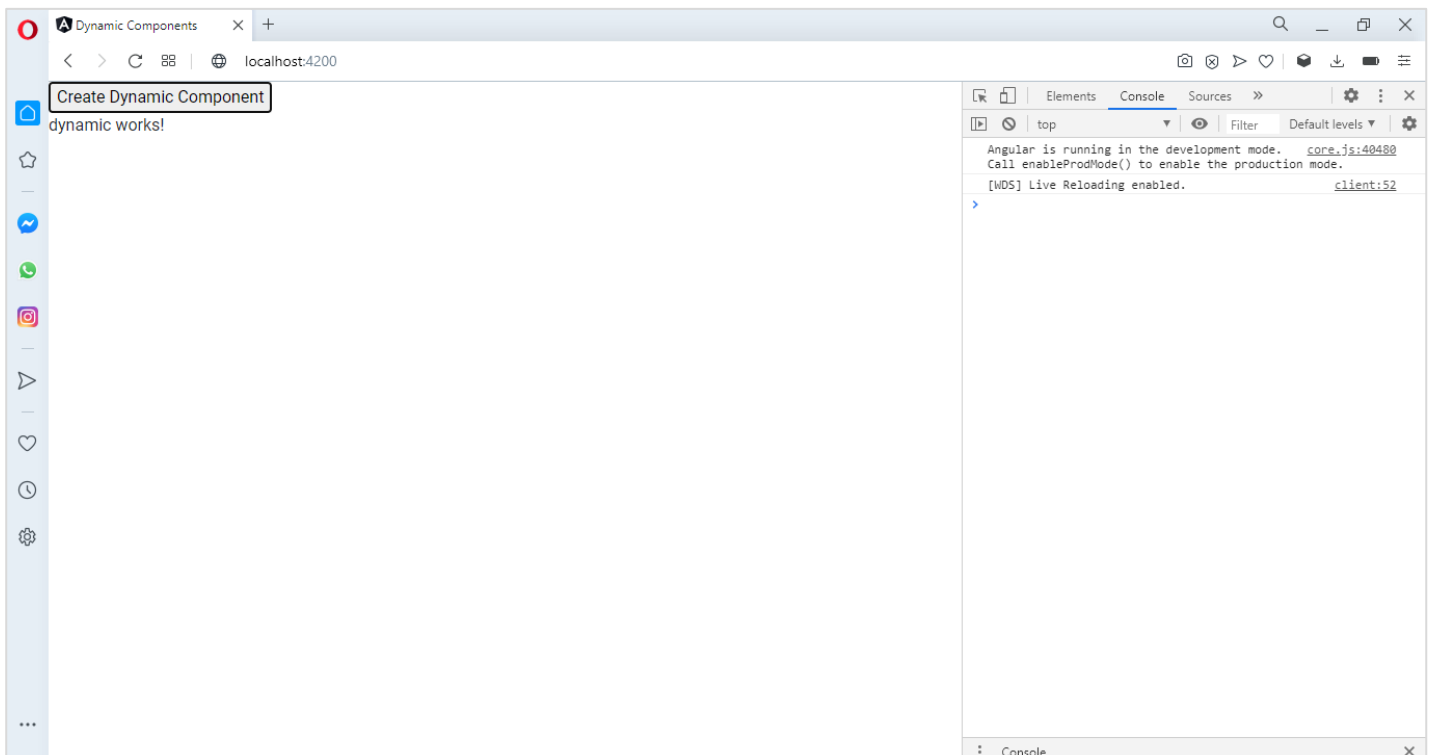
ملف app.component.html

```
<button (click)="createDynamicComponent()">Create Dynamic Component</button>
```

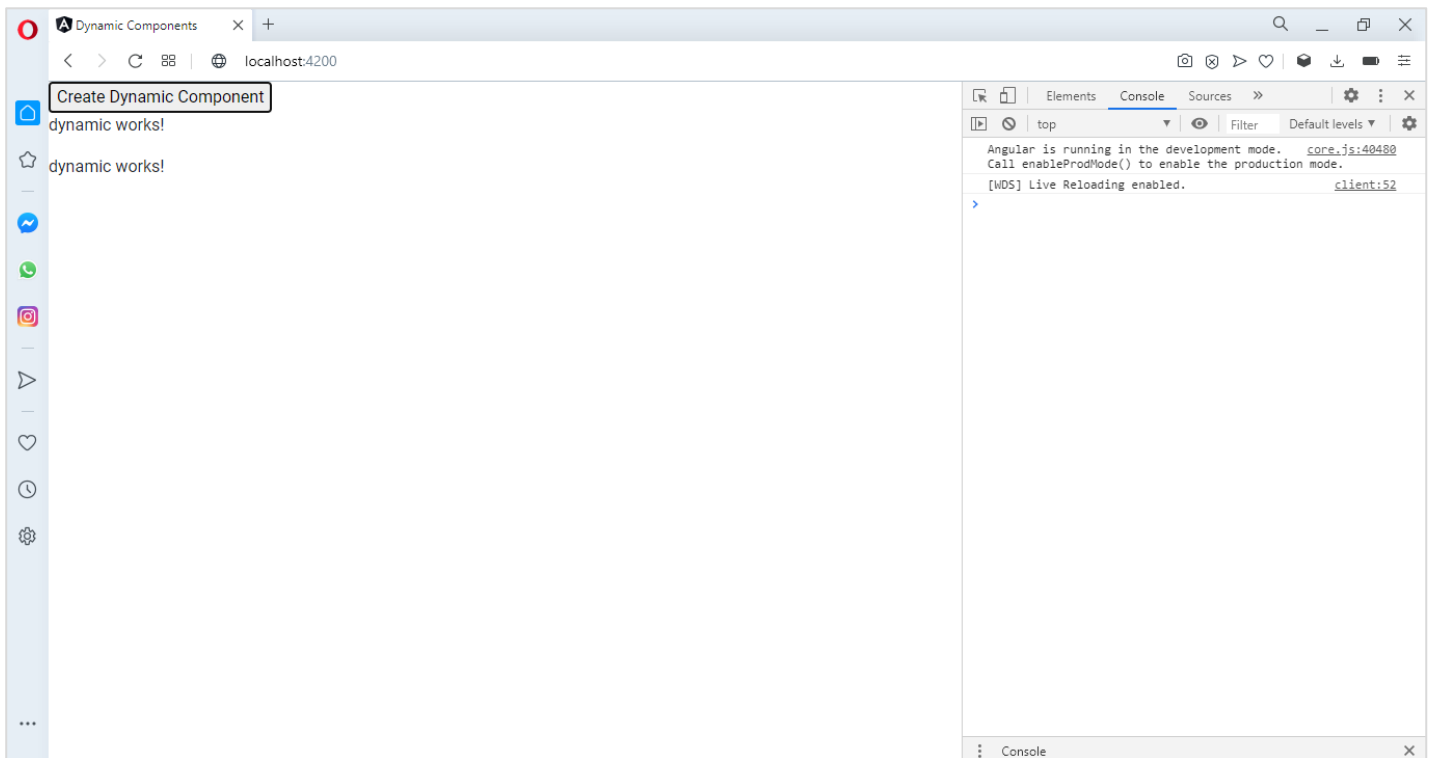
وأخيراً لنشاهد النتيجة في المتصفح، كالتالي:



والآن لنقوم بالضغط على الزر ولنرى النتيجة:



نلاحظ أضف لنا component ديناميكياً أثناء وقت التشغيل، والآن لنقوم بالضغط على الزر مرة أخرى، ولنرى النتيجة:



نلاحظ أضف لنا نسخة أخرى من هذا component، بحيث كل مرة نقوم بالضغط سوف يقوم بإضافة نسخة أخرى من هذا component ليكونوا جميعاً أخوة للأب والذي هو في مثالنا هذا Root Component، وهو بذلك يختلف `*ngComponentOutlet` الذي يقوم بإنشائه مره واحده فقط وكلما نقوم بالضغط فلن يقوم بإنشاء إلا نسخة واحدة، وفي حال اردنا ان يقوم بإنشاء نسخة واحدة فقط نضيف الأمر التالي:

```
app.component.ts ملف
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  constructor(private vcr: ViewContainerRef, private cfr: ComponentFactoryResolver) { }
  ngOnInit() {}
  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.vcr.clear();
    this.vcr.createComponent(resolver);
  }
}
```

ولو حاولنا ان نضغط الزر أكثر من مرة فإنه لن يضيف إلا نسخة واحدة فقط.

وبنفس الوقت هنالك إشكالية أخرى وهي أن إضافة هذا component تكون بعد آخر عنصر في الصفحة، وهذا أيضاً يختلف عن `*ngComponentOutlet` الذي نضيفه في مكان محدد في الصفحة، وفي حال اردنا ان نضيفه في مكان محدد في الصفحة نستخدم `ng-container` ونعطيه `Template Reference`، كالتالي:

ملف `app.component.html`

```
<ng-container #dynamic></ng-container>

<button (click)="createDynamicComponent()">Create Dynamic Component</button>
```

ولنستقبل هذا `Template Reference` في ملف `class` عن طريق `ViewChild` ونجعله يقرأه على انه `ViewContainerRef` كما تعلمنا سابقاً، ونحذف `inject`، كالتالي:

ملف `app.component.ts`

```
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver, ViewChild
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

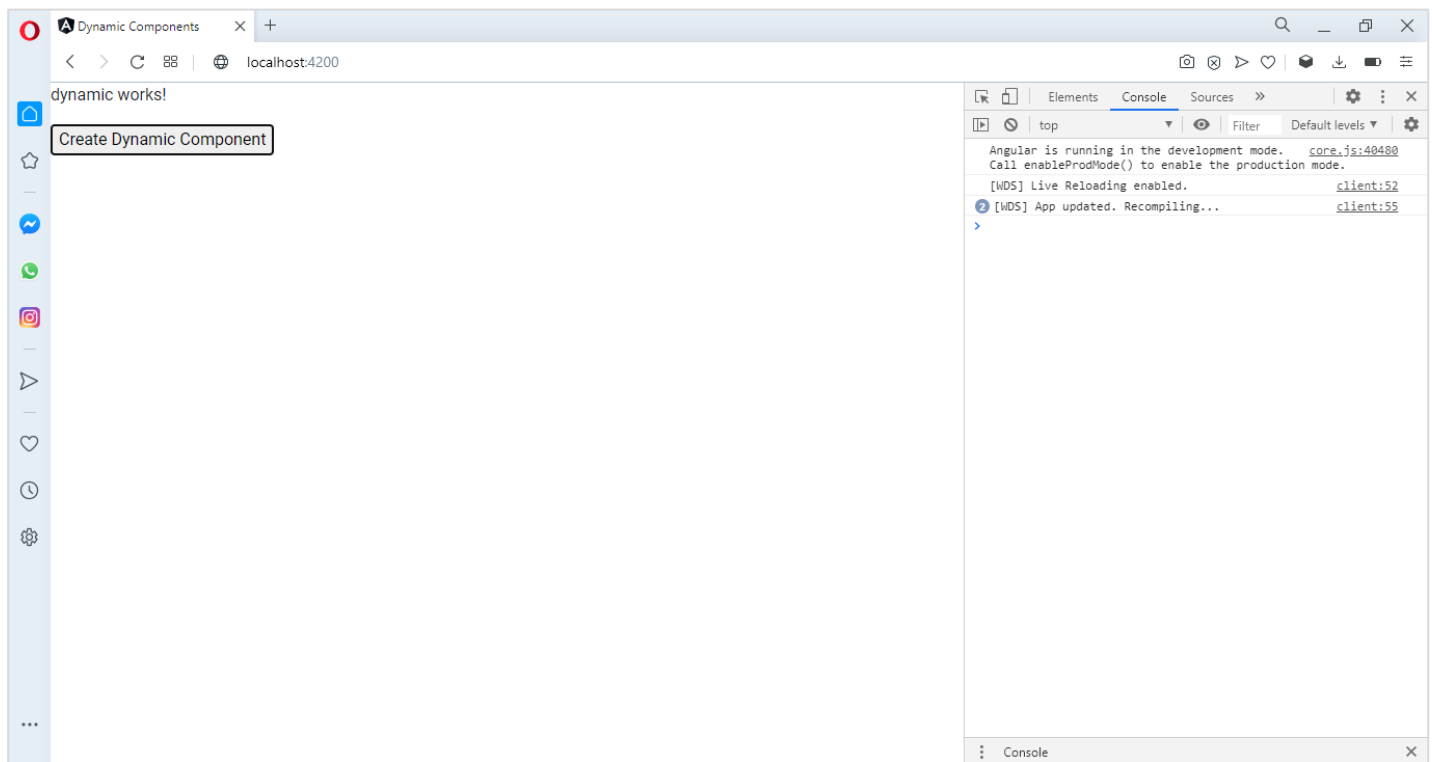
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {
  @ViewChild('dynamic', { read: ViewContainerRef }) private dynamic: ViewContainerRef;
  constructor(private cfr: ComponentFactoryResolver) { }

  ngOnInit() {}

  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.dynamic.clear();
    this.dynamic.createComponent(resolver);
  }
}
```

اما النتيجة في المتصفح، كالتالي:



## 2.13.4. المثال الثاني:

في هذا المثال نريد ان نتعلم كيف نصل إلى @Input في Dynamic Component ونقوم بتغيير قيمتها.

وسوف تلاحظ عزيزي المتعلم أن الوصول إلى Input اسهل بكثير من الوصول لها عن طريق \*ngComponentOutlet، لذلك لنقم بإنشاء Input إلى component الديناميكي وليكن اسمه name، كالتالي:

```

ملف dynamic.component.ts
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-dynamic',
  templateUrl: './dynamic.component.html',
  styleUrls: ['./dynamic.component.css']
})

export class DynamicComponent implements OnInit {

  @Input() name: string = 'Faisal';

  constructor() { }

  ngOnInit(): void {
  }

}

```

ومن ثم لنقم بعمل bind لهذا input في ملف template، كالتالي:

```
<p>
  Hi <span style="color: red;">{{ name }}</span> Dynamic Component
</p>
```

الآن لنقم بالوصول إلى هذا input عن طريق Root Component، وطريقة الوصول كما اشرنا سابقاً هي بسيطة فكل الذي سوف نقوم به هو تعريف متغير ويكون من النوع `ComponentRef<DynamicComponent>` والسبب لأن الدالة `CreateComponent` التي استخدمناها في السابق تُعيد `ComponentRef` وهو generic أي يأخذ `Component` الديناميكي والذي هو في مثالنا هذا اسميته `DynamicComponent`، كالتالي:

```
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver, ViewChild, ComponentRef
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {

  @ViewChild('dynamic', { read: ViewContainerRef }) private dynamic: ViewContainerRef;

  private cr: ComponentRef<DynamicComponent>;

  constructor(private cfr: ComponentFactoryResolver) { }

  ngOnInit() { }

  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.dynamic.clear();
    this.cr = this.dynamic.createComponent(resolver);
  }
}
```

نلاحظ في السطر البرمجي الذي تم التأشير عليه بالسهم الأول قمنا بتعريف الخاصية، اما السطر الذي تم التأشير عليه بالسهم الثاني فقمنا بحفظ القيمة التي ترجع لنا من الدالة `CreateComponent` في الخاصية التي عرفناها سابقاً.

وبذلك أصبحت هذه الخاصية تحتوي على خاصية اسمها `instance` التي تتيح لنا الوصول إلى جميع input في `DynamicComponent`، والآن لنقوم بإنشاء دالة تقوم بالوصول إلى هذا input وتغيير قيمته، كالتالي:



```
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver, ViewChild, ComponentRef
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit {
  @ViewChild('dynamic', { read: ViewContainerRef }) private dynamic: ViewContainerRef;
  private cr: ComponentRef<DynamicComponent>;
  constructor(private cfr: ComponentFactoryResolver) { }

  ngOnInit() { }

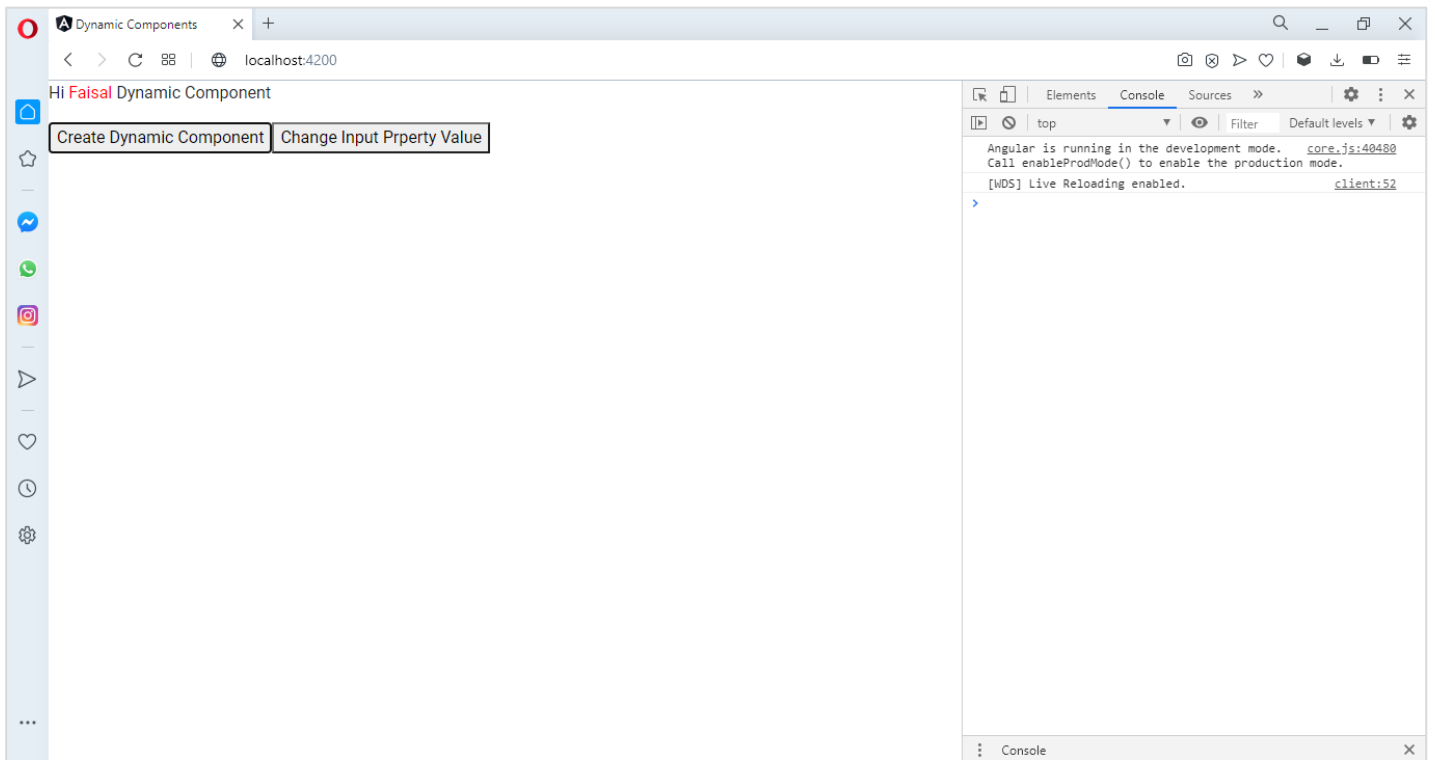
  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.dynamic.clear();
    this.cr = this.dynamic.createComponent(resolver);
  }

  changeInputProperty() {
    this.cr.instance.name = 'Fahd';
  }
}
```

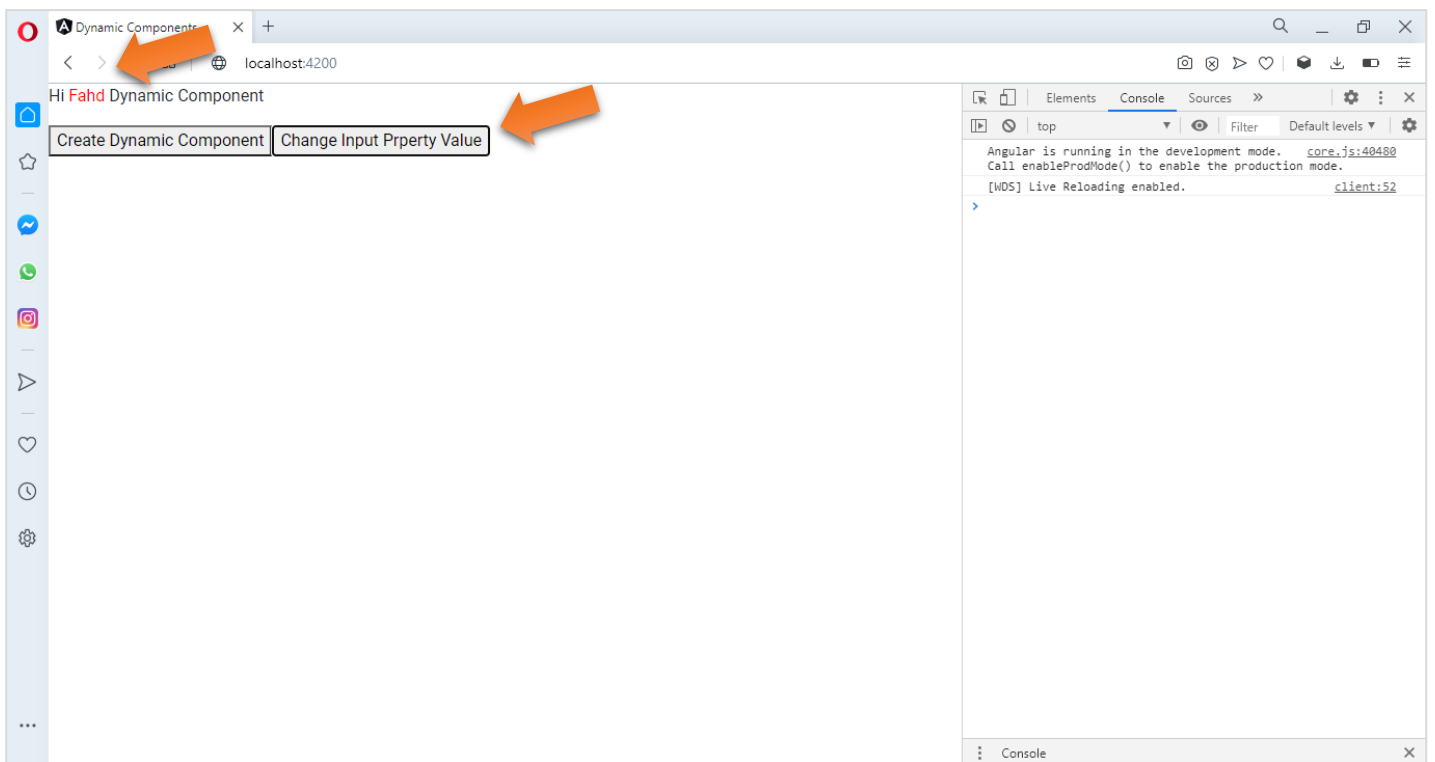
وأخيراً سوف نقوم بإنشاء زر في ملف template لكي يتم تنفيذ هذه الدالة عند الضغط عليه، كالتالي:

```
<ng-container #dynamic></ng-container>
<button (click)="createDynamicComponent()">Create Dynamic Component</button>
<button (click)="changeInputProperty()">Change Input Property Value</button>
```

ولنشاهد النتيجة في المتصفح، كالتالي:



نلاحظ عندما قمنا بالضغط على الزر الأول أضف لنا component بشكل ديناميكي، والآن لنقوم بالضغط على الزر الثاني لكي نقوم بتغيير قيمة Input الذي اسميناها name، كالتالي:



### 3.13.4. المثال الثالث:

في هذا المثال سوف نتعلم كيف نتعامل مع البيانات @Output الخارجة من component الديناميكي وكيف يتم مراقبة هذه البيانات والوصول لها وتنفيذ كود معين عند تغير قيمتها، لذلك لنقوم بإضافة Output في component الديناميكي وبنفس الوقت نعمل emit لقيمة معينة من النوع string، كالتالي:

ملف dynamic.component.ts

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

@Component({
  selector: 'app-dynamic',
  templateUrl: './dynamic.component.html',
  styleUrls: ['./dynamic.component.css']
})

export class DynamicComponent implements OnInit {

  @Input() name = 'Faisal';
  @Output() data: EventEmitter<string> = new EventEmitter<string>();

  constructor() { }

  ngOnInit(): void { }

  changeValue() {
    this.data.emit('Saad');
  }

}
```

والآن لنذهب إلى ملف template لهذا component وننشأ الزر الذي يُنفذ هذه الدالة، كالتالي:

ملف dynamic.component.html

```
<p class="w-100 p-4 border border-dark bg-light">
  Hi <span class="bg-danger text-white">{{ name }}</span> Dynamic Component
  <button (click)="changeValue()">change Output Value</button>
</p>
```

والآن لنذهب إلى ملف class في component الرئيسي الذي يحتوي على Dynamic Component ونكتب Logic اللازم لمراقبة هذا Output في component الديناميكي ومعرفة في حال حدث أي تغيير، كالتالي:

ملف app.component.ts

```
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver, ViewChild, ComponentRef
} from '@angular/core';
import { DynamicComponent } from './dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
```

```

export class AppComponent implements OnInit {
  @ViewChild('dynamic', { read: ViewContainerRef }) private dynamic: ViewContainerRef;
  private cr: ComponentRef<DynamicComponent>;
  constructor(private cfr: ComponentFactoryResolver) { }

  ngOnInit() { }

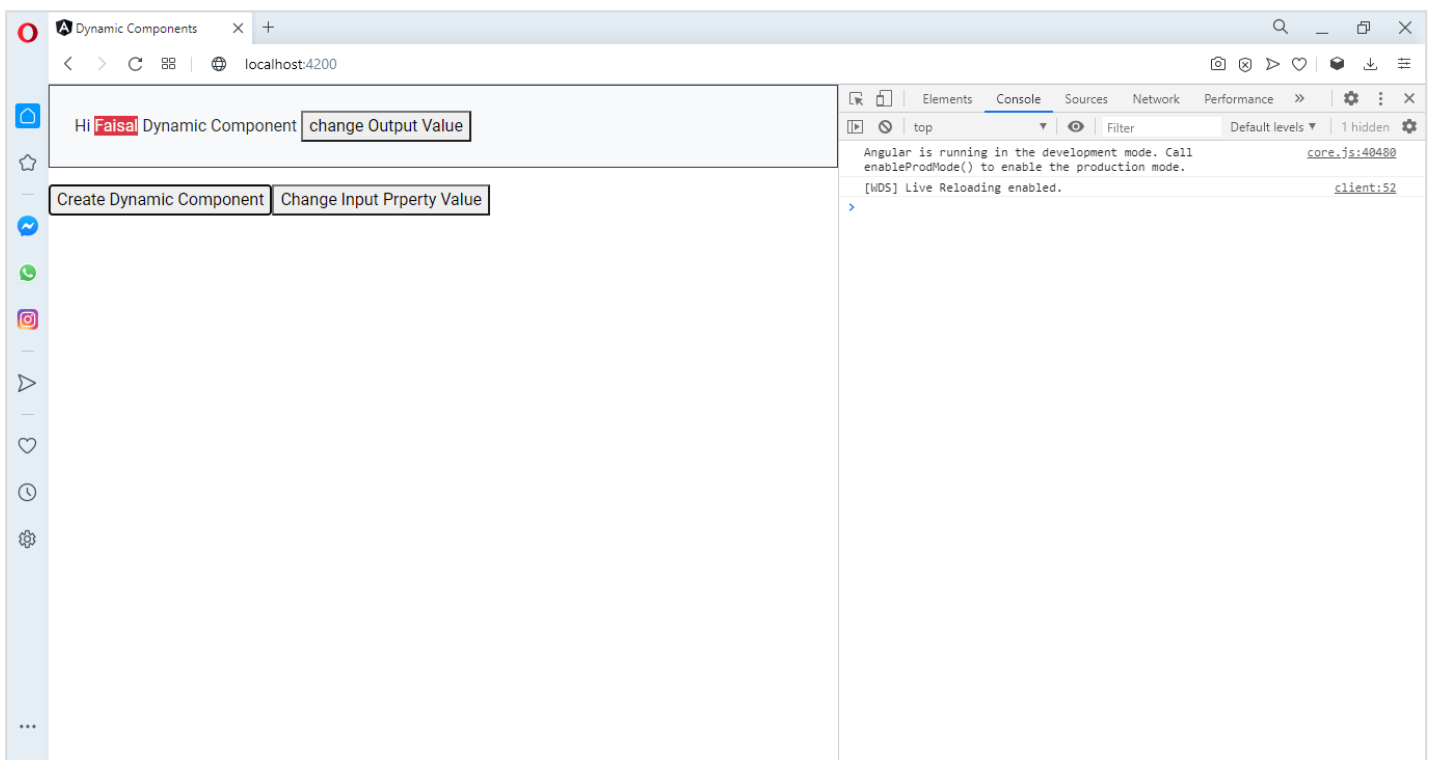
  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.dynamic.clear();
    this.cr = this.dynamic.createComponent(resolver);

    this.cr.instance.data.subscribe((value: string) => {
      console.log('@Output Value: ' + value);
    });
  }

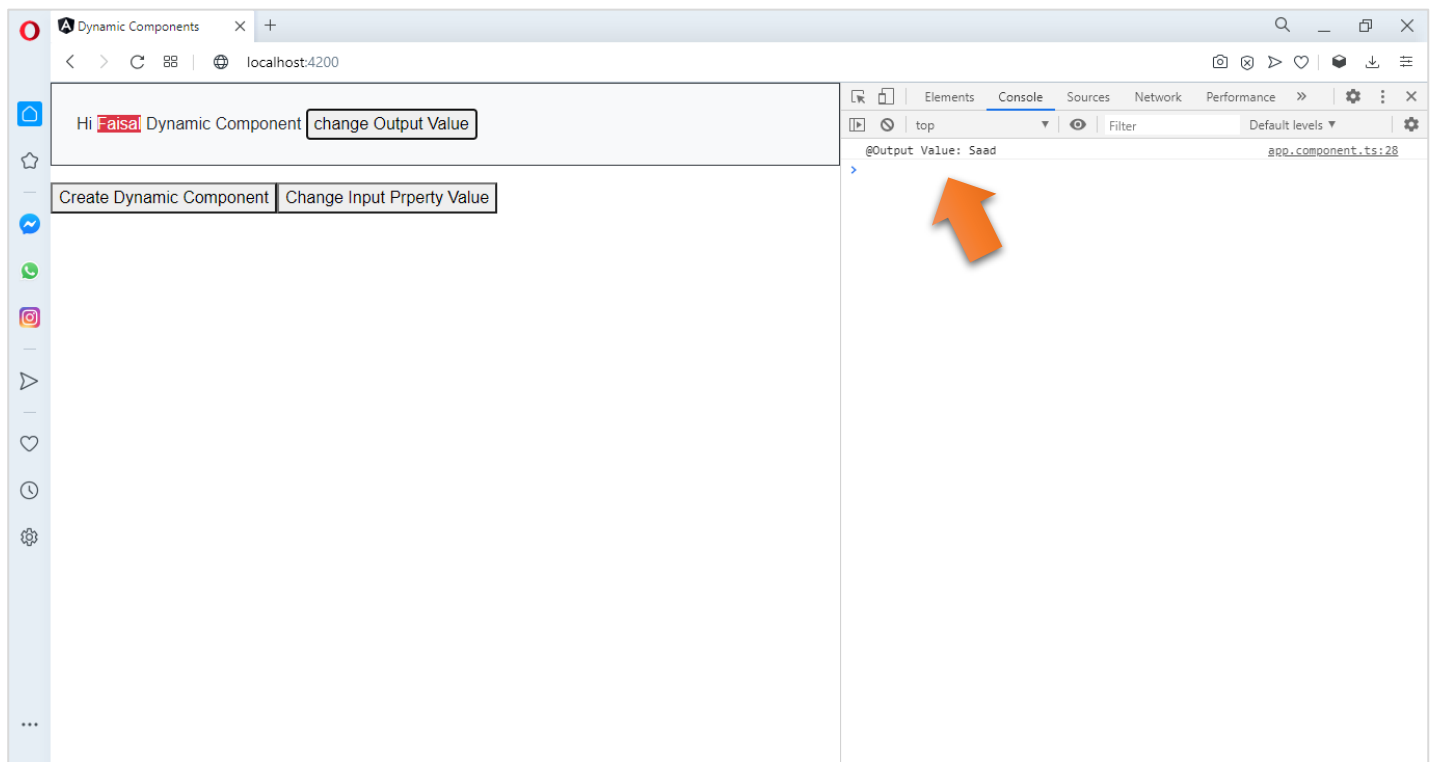
  changeInputProperty() {
    this.cr.instance.name = 'Fahd';
  }
}

```

نلاحظ عملنا subscribe لي @Output والمتمثل في الخاصية data والقيمة التي ترجع هي نفسها القيمة التي عملنا لها emit في component الديناميكي، وهنا فقط قمنا بعرض هذه القيمة في console، اما النتيجة في المتصفح فتكون كالتالي:



نلاحظ قمنا بالضغط على الزر انشاء component، وظهر لنا هذا component الديناميكي، ولنقم بالضغط على زر change output value (مع العلم ان هذا الزر موجود في component الديناميكي) لتغيير قيمة output وبنفس الوقت سوف نلتقط هذه القيمة في Root Component ونعرضها في console، كالتالي:



وأخيراً يُفضل ان نعمل destroy لهذا component الديناميكي (لا يشترط عمل ذلك)، كالتالي:

```
app.component.ts ملف
import {
  Component,
  OnInit,
  ViewContainerRef,
  ComponentFactoryResolver,
  ViewChild,
  ComponentRef,
  OnDestroy
} from '@angular/core';
import { DynamicComponent } from '../dynamic/dynamic.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent implements OnInit, OnDestroy {
  @ViewChild('dynamic', { read: ViewContainerRef }) private dynamic: ViewContainerRef;
  private cr: ComponentRef<DynamicComponent>;
  constructor(private cfr: ComponentFactoryResolver) { }

  ngOnInit() { }

  createDynamicComponent() {
    const resolver = this.cfr.resolveComponentFactory(DynamicComponent);
    this.dynamic.clear();
    this.cr = this.dynamic.createComponent(resolver);
  }
}
```

```

    this.cr.instance.data.subscribe((value: string) => {
      console.log('@Output Value: ' + value);
    });
  }

  changeInputProperty() {
    this.cr.instance.name = 'Fahd';
  }

  ngOnDestroy(): void {
    this.cr.destroy();
  }
}

```

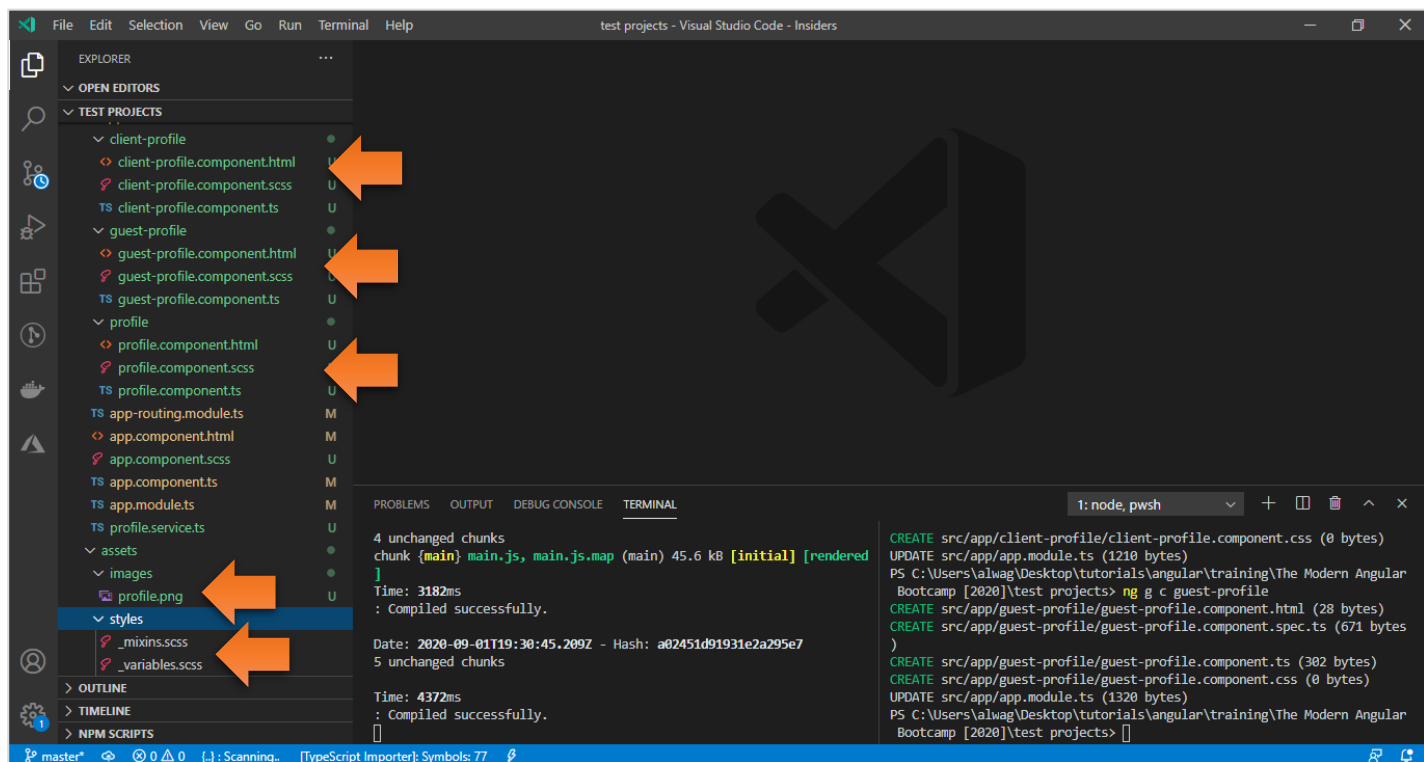
#### 4.13.4. المثال الرابع:

لننتقل خطوة إلى الامام قليلاً ونستعرض المثال الموجود في هذا الموقع <https://labs.thisdot.co/blog/loading-components-dynamically-in-angular-9-with-ivy> بعنوان [Loading Components Dynamically in Angular 9 with Ivy](https://labs.thisdot.co/blog/loading-components-dynamically-in-angular-9-with-ivy) مع بعض التصرف البسيط.

وتقوم فكرة المثال اننا نريد ان نقرر اثناء وقت تشغيل التطبيق بعرض component معين بشكل ديناميكي بناءً على حالة تسجيل الدخول للمستخدم، فإذا كان المستخدم قد قام بتسجيل الدخول فنظهر له component معين وإذا تغيرت حالته لأي سبب من الأسباب واصبح غير مسجل لدخوله إلى التطبيق فإن نظامنا سوف يقوم بعرض component آخر. طبعاً نستطيع عمل هذا الأمر بعدة طرق، أشهرها باستخدام Routing او باستخدام \*ngIf، ولكن هنا سوف نستخدم طريقة Dynamic Components كنوع من التدريب.

لذلك يُفضل إنشاء مشروع جديد، واختيار style بان يكون scss وليس css، ومن ثم نقوم بإضافة ثلاثة components واحد باسم profile بحيث يكون الحاوية التي يتم عرض باقي components الأخرى فيها بشكل ديناميكي، والثاني باسم guest-profile حيث يتم عرض هذا component بشكل ديناميكي في حال ان المستخدم لم يقم بتسجيل الدخول، والثالث باسم client-profile حيث يتم عرضه في حال ان المستخدم قام بتسجيل الدخول، وآخر جزء نقوم بإضافته هو service وليكن اسمها profile حيث سوف نكتب بها جميع Logic الخاص بالتأكد من عمليتي تسجيل الدخول/الخروج وايضاً عرض components بشكل ديناميكي.

وبعدما نقوم بإنشاء هذه components سوف ننشأ مجلدين داخل assets الأول باسم images ونضع فيه أي صورة تمثل صورة للحساب وليكن اسمها profile.png اما المجلد الآخر باسم styles ونضع فيه ملفين الأول باسم \_variables.scss والثاني باسم \_mixins.scss، بطبيعة الحال لن نتطرق للأكواد في هذين الملفين لأنه ليس هنا المقام لشرح scss او sass:



### ملف `_variables.scss`

```
$card-width: 400px;  
$avatar-width: 80px;  
$container-margin: 20px;
```

### ملف `_mixins.scss`

```
@import './variables.scss';  
  
@mixin button($color) {  
  display: inline-block;  
  padding: 0.5rem 1rem;  
  border: 1px solid $color;  
  border-bottom-color: darken($color, 10);  
  border-radius: 5px;  
  background: linear-gradient(180deg, $color, darken($color, 10));  
  color: white;  
  cursor: pointer;  
  font-family: Arial, Helvetica, sans-serif;  
  box-shadow: 1px 2px 4px rgba(0, 0, 0, 0.2);  
  font-size: 1rem;  
  
  &:hover {  
    background: $color;  
    box-shadow: 1px 4px 6px rgba(0, 0, 0, 0.2);  
  }  
  
  &:active {  
    background: darken($color, 10);  
  }  
}
```

```
@mixin card {
  box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2);
  border: 1px solid #eee;
  width: $card-width;
  padding: 1rem;
}
```

لا تهتم عزيزي المتعلم في الأكواد السابقة فهي مجرد أكواد للتنسيق فقط لا غير، واعرني انتباهك لما هو قادم، وأول جزء سوف نبدأ فيه هو ملف profile.service.ts، حيث سوف يحتوي على Logic الخاص بعرض component بشكل ديناميكي بناءً على حالة المستخدم، لذلك لنستعرض محتويات هذا الملف، ونحاول إيضاح بعض المفاهيم، كالتالي:

#### ملف profile.service.ts

```
import { ComponentFactoryResolver, Injectable, ViewContainerRef } from '@angular/core';
import { BehaviorSubject, from } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})

export class ProfileService {

  private isLoggedIn = new BehaviorSubject(false);
  public isLoggedIn$ = this.isLoggedIn.asObservable(); ①

  constructor(private cfr: ComponentFactoryResolver) { }

  private guestProfile() {
    return () =>
      import('./guest-profile/guest-profile.component').then( ②
        m => m.GuestProfileComponent
      );
  }

  private clientProfile() {
    return () =>
      import('./client-profile/client-profile.component').then( ③
        m => m.ClientProfileComponent
      );
  }

  private forChild(vcr: ViewContainerRef, cl: { loadChildren: () => Promise<any>; }) {
    return from(cl.loadChildren()).pipe(
      map((component: any) => this.cfr.resolveComponentFactory(component)), ④
      map(componentFactory => vcr.createComponent(componentFactory))
    );
  }

  login() {
    this.isLoggedIn.next(true); ⑤
  }
}
```



```

logout() {
  this.isLoggedIn.next(false); 6
}

loadComponent(vcr: ViewContainerRef, isLoggedIn: boolean) {
  vcr.clear();
  return this.forChild(vcr, { 7
    loadChildren: isLoggedIn ? this.clientProfile() : this.guestProfile()
  });
}
}

```

تعريف بعض المتغيرات لتخزين حالة المستخدم بحيث تكون القيمة true في حال قام بتسجيل الدخول والقيمة false في حال لم يتم بتسجيل الدخول.

رقم 2 و 3 في الحقيقة طريقة جيدة في جلب components التي نريد اضافتها بشكل ديناميكي، باستخدام الصيغة الديناميكية في الاستدعاءات (كما هو الحال في استدعاء Modules في Lazy Loading)، ونلاحظ انه استخدم then بمعنى استفاد من تقنية Promise.

اما رقم 4 فهي دالة private لا تُرى إلا داخل هذا class فقط، ومهمتها بكل بساطة انشاء component او بصيغة أخرى بناء component برمجياً أثناء وقت تشغيل التطبيق، ونلاحظ مررنا لها بارامتر اسمه c1 وهو عبارة عن كائن يحتوي على دالة باسم loadChildren بحيث تُعيد Promise، والسبب الذي جعلناها تُعيد Promise لأن هذا البارامتر بكل بساطة سوف يخزن قيمة إحدى components القادمتين في النقطة 2 و 3، وبما ان هذين الدالتين تُعيدان Promise لذلك جعلنا هذا البارامتر ايضاً هو الآخر يُعيد Promise.

اما النقطتين 5 و 6 فتقومان بتغيير حالة المستخدم.

وآخر نقطة وهي 7 تقوم باستدعاء الدالة الخاصة ببناء component ديناميكياً، ومن ثم تسند قيمة لي الدالة loadChildren تبعاً لقيمة المتغير isLoggedIn بحيث إذا كانت القيمة true يُسند لدالة loadChildren component الديناميكي الذي استدعينا ديناميكياً عن طريق then وتقنيات Promises والذي هو ClientProfileComponent والمتمثل بالدالة الموجودة في النقطة 2، اما في حال كانت القيمة false فنقوم بعكس العملية واسناد component الذي ينتج لنا من الدالة في النقطة 3.

والخطوة القادمة هي الذهاب إلى Profile Component والذي اشرنا سابقاً انه يعتبر بمثابة الأب او الحاوية التي نريد ان نظهر components الديناميكية خلالها، لذلك لنذهب إلى ملف template لهذا component ونضيف ng-container ونعطيه template reference حيث سوف يتم استبدال هذا ng-container بالcomponent الديناميكي الذي نريد اظهره، كالتالي:

ملف profile.component.html

```
<ng-container #profile></ng-container>
```

الآن لنستقبل هذا template reference في ملف class عن طريق @ViewChild ونقرأها على انها ViewContainerRef كما تعلمنا سابقاً، ولكن هنا هنالك إضافة وهي الخاصية static وتستقبل قيمة منطقية true او false مع العلم ان القيمة الافتراضية لها هي false أي بمعنى في حال عدم كتابتنا لها سوف يتم اسناد لها القيمة false، اما الفائدة منها فنضع قيمتها true في حال اردنا ان نقرأ القيمة في @ViewChild او @ViewChildren او حتى @ContentChild او @ContentChildren في الدالة ngOnInit، أي بمعنى آخر نريد قراءة هذه القيمة قبل لا يتم بناء View في ملف template، وله عدة استخدامات منها على سبيل المثال في حال اردنا ان نعرض جزء من view او content في ملف template بشكل ديناميكي بناءً على بيانات معينة، كما في حالتنا هذه حيث بناءً على حالة المستخدم نقرر هل نظهر Guest Component ام Client Component، كالتالي:

ملف profile.component.ts

```
import { Component, OnInit, ViewChild, OnDestroy, ViewContainerRef } from '@angular/core';
import { Subject } from 'rxjs';
import { mergeMap, takeUntil } from 'rxjs/operators';
import { ProfileService } from '../profile.service';

@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  styleUrls: ['./profile.component.scss']
})

export class ProfileComponent implements OnInit, OnDestroy {
  1
  @ViewChild('profile', { static: true, read: ViewContainerRef }) profile: ViewContainerRef;

  private destroySubject = new Subject();

  constructor(private profileService: ProfileService) { }

  ngOnInit() {
    this.profileService.isLoggedIn$.pipe(
      takeUntil(this.destroySubject),
      mergeMap(isLoggedIn =>
        this.profileService.loadComponent(this.profile, isLoggedIn)
      )
    ).subscribe();
  }

  ngOnDestroy() {
    this.destroySubject.next();
    this.destroySubject.complete();
  }
}
```

نلاحظ الخاصية static والقيمة true المسندة لها، وبنفس الوقت في الدالة ngOnInit باستدعاء الدالة loadComponent والموجودة في service التي قمنا ببنائها قبل قليل، حيث مررنا لها ng-container والمتغير الذي يمثل حالة المستخدم.

اما باقي Logic عزيزي المتعلم فلا تتعب عقلك في فهمه لكي لا يتشتت انتباهك، فهذه الأكواد رغم أهميتها ولكن ليس هنا المقام لشرحها وهي تنتمي إلى مكتبة rxjs ودوالها وأسلوب Reactive Programing، ولعلنا نفرد كتاب مختص في هذا القسم من هذه السلسلة إن شاء الله.

إلى هذه النقطة نستطيع ان نقول اننا قطعنا شوطاً كبيراً في هذا المثال وما تبقى هو بعض اللمسات البسيطة ومنها تصميم components الديناميكية وازضافة بعض Logic لها مثل زر لتغيير حالة المستخدم في كلا components، كالتالي:

ملف client.component.html

```
<section class="card">
  <figure class="card__avatar">
    
  </figure>

  <h2 class="card__title" contenteditable="true">Daniel Marin</h2>

  <p class="card__subtitle" contenteditable="true">
    Senior Software Engineer at This Dot Labs, a company specializing in
    Modern Web Technologies, designing, and developing software to help
    companies maximize efficiency in their processes.
  </p>

  <div class="card__toolbar">
    <button (click)="logout()">Logout</button>
  </div>
</section>
```

ملف client.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ProfileService } from '../profile.service';

@Component({
  selector: 'app-client-profile',
  templateUrl: './client-profile.component.html',
  styleUrls: ['./client-profile.component.scss']
})
export class ClientProfileComponent implements OnInit {

  constructor(private profileService: ProfileService) { }

  ngOnInit(): void { }

  logout() {
    this.profileService.logout();
  }
}
```

ملف client.component.scss

```
@import "../../assets/styles/mixins.scss";
```

```

.card {
  @include card();

  &__avatar {
    height: $avatar-width;
    width: $avatar-width;
    margin: 0 auto;
    border-radius: 50%;
    overflow: hidden;

    img {
      width: 100%;
      height: 100%;
      object-fit: cover;
    }
  }

  &__title {
    margin: 1rem 0 0.5rem 0;
    text-align: center;
  }

  &__subtitle {
    margin: 0 0 1rem 0;
    text-align: center;
  }

  &__toolbar {
    display: flex;
    justify-content: center;

    button {
      @include button(#a80000);
    }
  }
}

```

ملف guest.component.html

```

<section class="card">
  <div class="card__avatar">
    <div class="card__avatar__head"></div>
    <div class="card__avatar__body"></div>
  </div>

  <div class="container">
    <h2 class="card__title">Guest Profile</h2>

    <p class="card__subtitle">
      Thank you for visiting us. If you want to take your experience to the
      next level, all you need is to log in.
    </p>

    <div class="card__toolbar">

```

```

    <button (click)="login()">Login</button>
  </div>
</div>
</section>

```

#### ملف guest.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ProfileService } from '../profile.service';

@Component({
  selector: 'app-guest-profile',
  templateUrl: './guest-profile.component.html',
  styleUrls: ['./guest-profile.component.scss']
})

export class GuestProfileComponent implements OnInit {

  constructor(private profileService: ProfileService) { }

  ngOnInit(): void { }

  login() {
    this.profileService.login();
  }

}

```

#### ملف guest.component.scss

```

@import "~src/assets/styles/mixins.scss";

.card {
  display: flex;
  @include card();

  &__title {
    margin: 0 0 0.5rem 0;
  }

  &__subtitle {
    margin: 0 0 0.5rem 0;
  }

  &__toolbar button {
    @include button(#145092);
  }

  &__avatar {
    height: 80px;
    width: $avatar-width;
    border: 2px solid #bbb;
    background: #666;
    position: relative;
    overflow: hidden;
  }
}

```

```

    &__head {
      position: absolute;
      border-radius: 50%;
      background: #bbb;
      width: 35px;
      height: 35px;
      top: 15px;
      left: 22px;
    }

    &__body {
      position: absolute;
      border-radius: 50%;
      background: #bbb;
      width: 70px;
      height: 50px;
      top: 55px;
      left: 5px;
    }
  }
}

.container {
  width: $card-width - $avatar-width - $container-margin;
  margin: 0 $container-margin;
}

```

أما في Root Component، فنضيف هذا Logic، كالتالي:

ملف app.component.html

```

<h1 class="header">Dynamic components</h1>
<main class="container">
  <app-profile></app-profile>
</main>

```

ملف app.component.scss

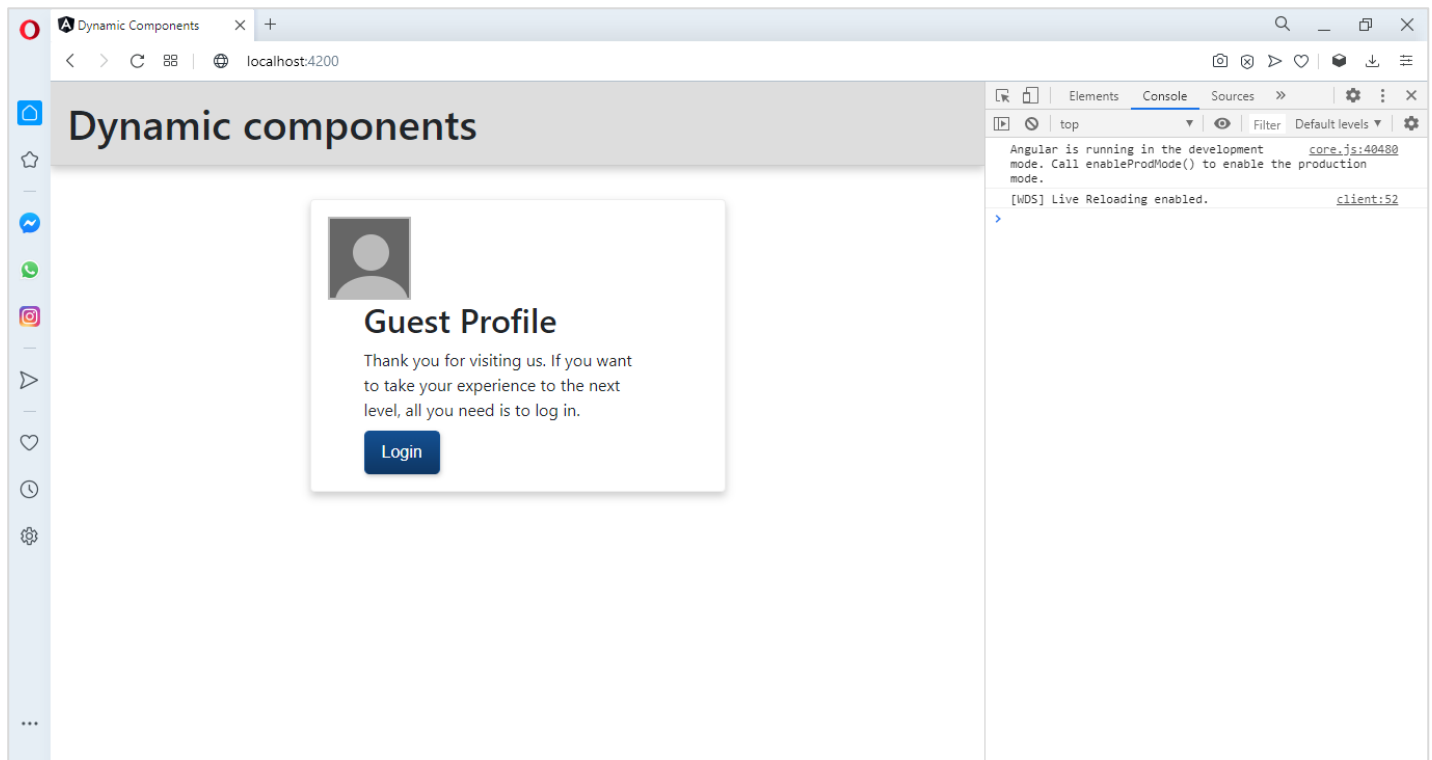
```

.header {
  background: #ddd;
  border-bottom: 1px solid #ccc;
  margin: 0;
  padding: 1rem;
  box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2);
}

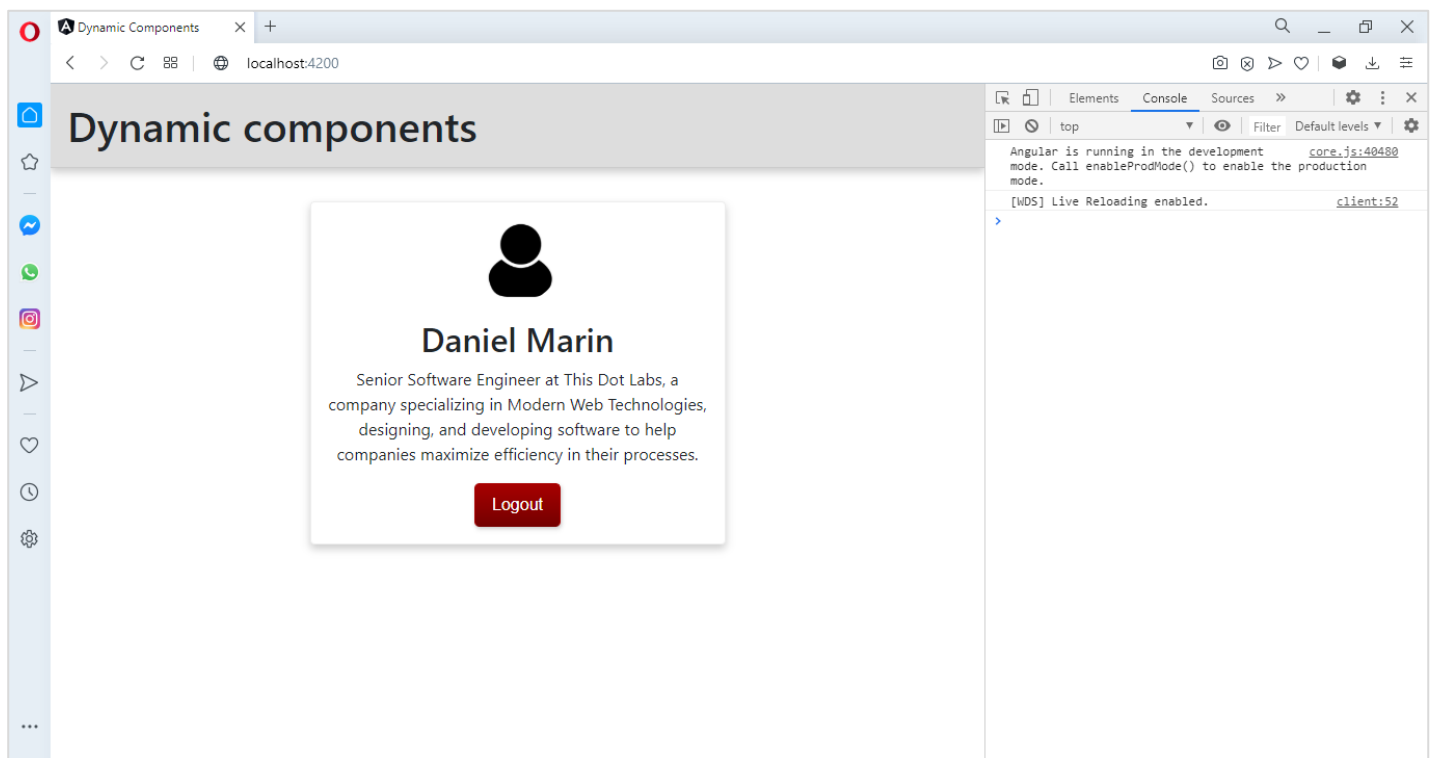
.container {
  display: flex;
  justify-content: center;
  margin-top: 2rem;
}

```

وأخيراً لنشاهد النتيجة في المتصفح، كالتالي:



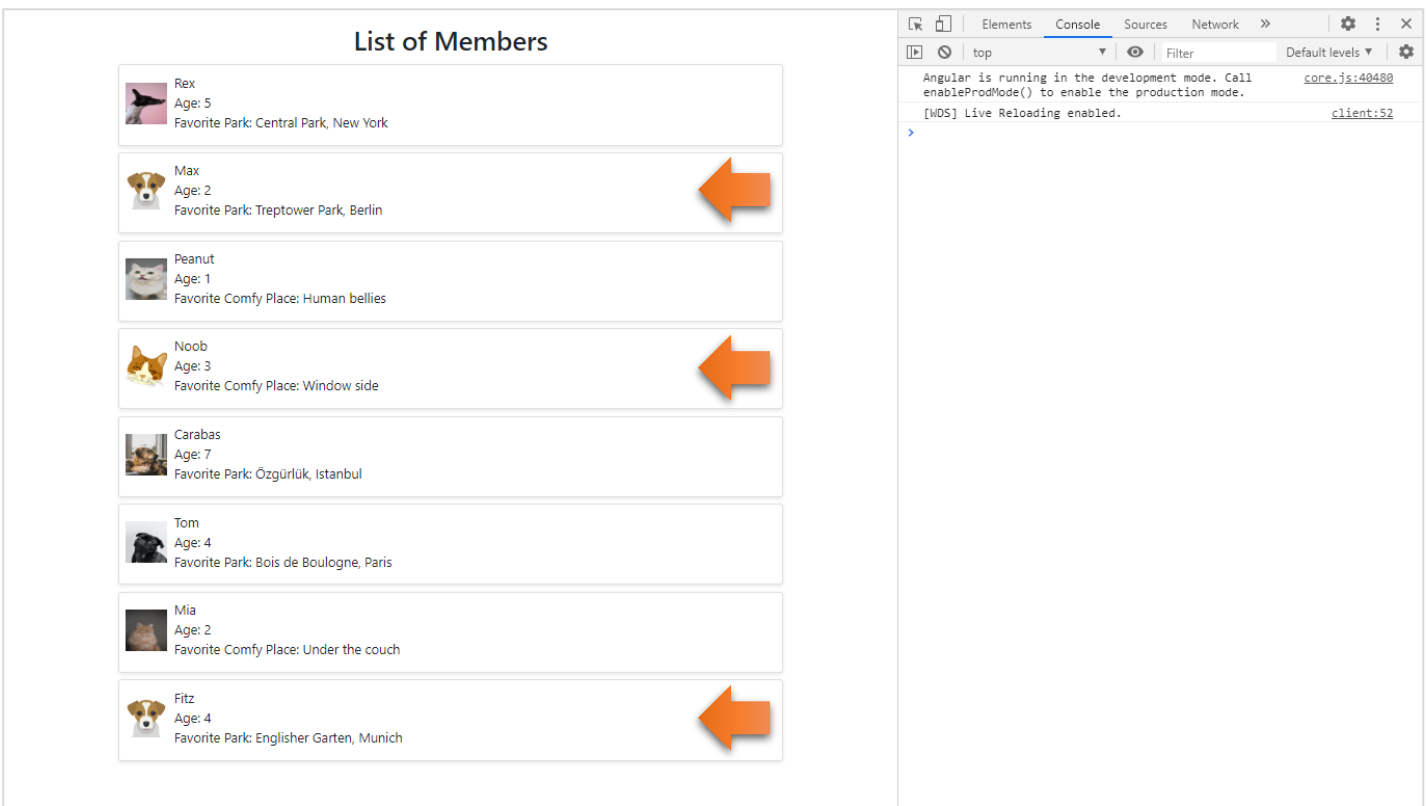
نلاحظ اظهر لنا Guest Component وذلك لأن القيمة المبدئية لحالة المستخدم هي false، ولوقمنا بتغيير القيمة المبدئية إلى true فسوف تلاحظ انه سيظهر Client Component، والآن لنقوم بالضغط على زر Login لكي نغير حالة المستخدم، كالتالي:



#### 5.13.4. المثال الخامس:

وهنا ايضاً مثال جيد ومصدره هذا الرابط <https://codeburst.io/dynamic-components-in-angular-8fd12490ab8> مع بعض التصرف البسيط، حيث تقوم فكرته في حال لدينا مجموعة من البيانات المجمعة والغير متشابهة كأن يكون لدينا بيانات كلاب وقطط (اعزكم الله) بحيث هنالك بيانات مشتركة بين كلا النوعين وايضاً هنالك بيانات خاصة بكل نوع.

بطبيعة الحال نستطيع حل هذه المشكلة بعدة طرق منها استخدام Dynamic Components حيث نقوم بإنشاء اثنين components واحد لعرض بيانات الكلاب والثاني لعرض بيانات القطط ونقوم بعرض هذه components ديناميكياً بناءً على البيانات الواردة في قائمة تجمع جميع هذه العناصر، بحيث إذا كان القط يوجد له صورة نظهرها وإذا لم يكن نظهر له صورة افتراضية خاصة بنوع القطط وبنفس الوقت في حال الكلب يحمل صورة نظهرها في القائمة وإن لم يكن نظهر له صورة افتراضية خاصة بنوع الكلاب، بحيث يكون التطبيق بشكله النهائي، كالتالي:



The screenshot shows a web application titled "List of Members" displaying a list of dogs. Each dog entry includes a small image, the dog's name, age, and favorite place. The list includes Rex (Age: 5, Favorite Park: Central Park, New York), Max (Age: 2, Favorite Park: Treptower Park, Berlin), Peanut (Age: 1, Favorite Comfy Place: Human bellies), Noob (Age: 3, Favorite Comfy Place: Window side), Carabas (Age: 7, Favorite Park: Özgürlük, Istanbul), Tom (Age: 4, Favorite Park: Bois de Boulogne, Paris), Mia (Age: 2, Favorite Comfy Place: Under the couch), and Fitz (Age: 4, Favorite Park: Englischer Garten, Munich). Three orange arrows point to the right, indicating the dynamic component logic. The browser's developer console is open, showing messages from Angular and WDS.

نلاحظ ان جميع البيانات مجمعة مع بعضها البعض وتم عرضها في قائمة واحدة مع عرض البيانات الخاصة بكل نوع وايضاً عرض البيانات المشتركة (الاسم والعمر) بين هذه الأنواع واخيراً عرض صورة افتراضية في حال عدم وجود صور للنوع المحدد كما هو مؤشر في الأسهم.

لذلك لنبدأ في تطبيق هذا المثال خطوة خطوة، وأول ما نقوم به هو انشاء ثلاث ملفات class لوصف هذه البيانات واحد باسم Pet ويحتوي على البيانات المشتركة وواحد باسم dog ويرث من class السابق ذو الاسم Pet مع إضافة البيانات الخاصة بهذا النوع وبنفس الطريقة بالنسبة لي ملف cat، كالتالي:

ملف pet.ts

```
export default class Pet {  
  name: string;
```



```

age: number;
profilePicture?: string;
}

```

نلاحظ في هذا الملف قمنا بتعريف class باسم Pet وقمنا بوصف البيانات المشتركة بين كلا النوعين، والآن لنقوم بإنشاء ملفين آخرين يقوموا بالوراثة من هذا الملف، كالتالي:

ملف cat.ts

```

import Pet from './pet';

export default class Cat extends Pet {
  favoriteComfyPlace: string;
}

```

نلاحظ في هذا الملف قمنا بإنشاء class يرث من Pet وبنفس الوقت قمنا بإضافة وصف لبيانات خاصة به، وبنفس الطريقة نطبقها على الملف الثاني، كالتالي:

ملف dog.ts

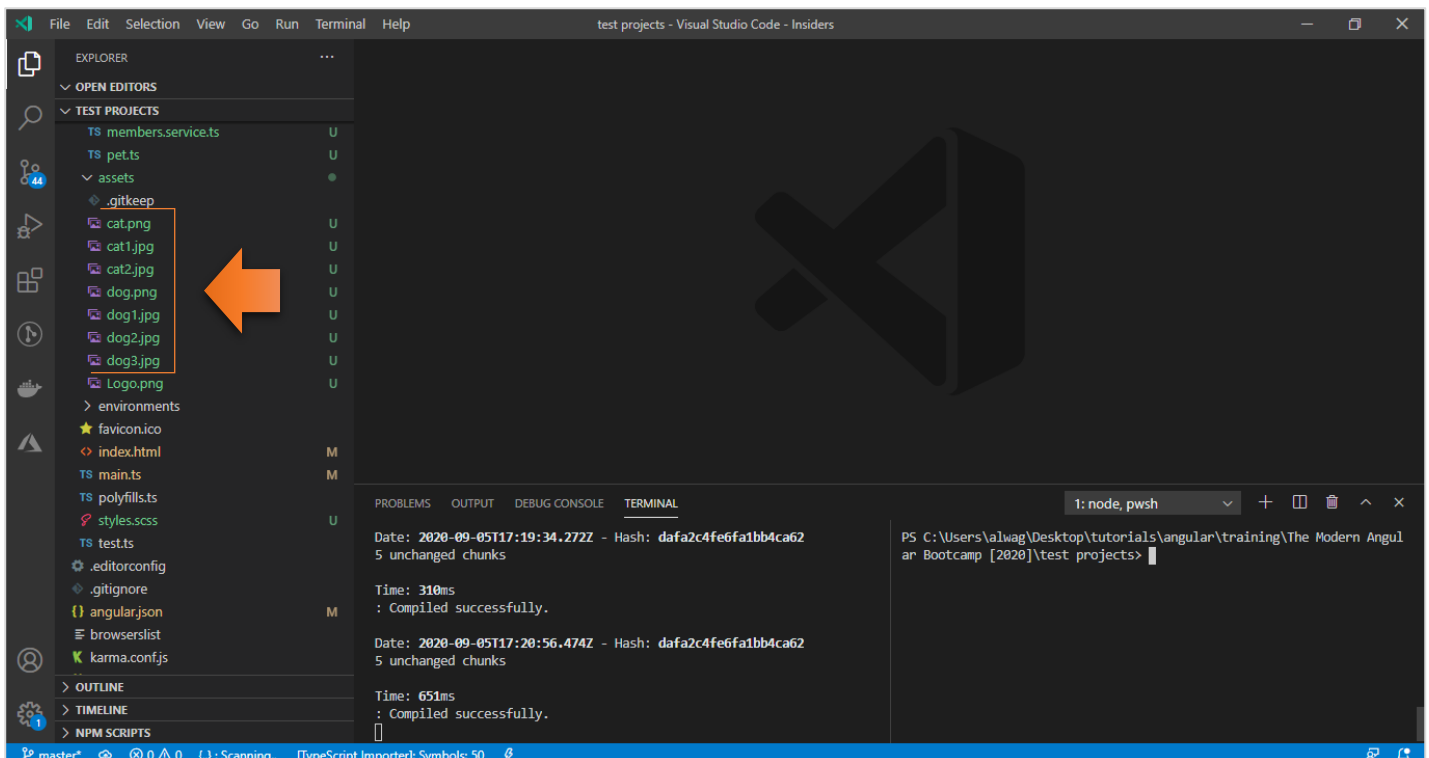
```

import Pet from './pet';

export default class Dog extends Pet {
  favoritePark: string;
}

```

بعدما قمنا بوصف البيانات لنقوم الآن بإنشاء هذه البيانات، وبطبيعة الحال في التطبيقات الواقعية هذه البيانات تكون موجودة في قاعدة بيانات وتصل فيها عن طريق api ولكن هنا لنبقى الأمور بسيطة ونقوم بإنشاء هذه البيانات يدوياً في ملف service باسم members، مع العلم ان الصورة قمت بتحميلها من الانترنت ووضعها في مجلد assets باسم (dog1.jpg – dog2.jpg – dog3.jpg) اما الصورة الافتراضية باسم dog.png وبنفس الطريقة مع صور القطط، كالتالي:



ملف members.service.ts

```
import { Injectable } from '@angular/core';
import Pet from './pet';
import Dog from './Dog';
import Cat from './cat';

@Injectable({
  providedIn: 'root'
})

export class MembersService {

  constructor() { }

  get members(): Pet[] {
    return [
      {
        name: 'Rex',
        age: 5,
        favoritePark: 'Central Park, New York',
        profilePicture: 'assets/dog1.jpg',
      } as Dog,
      {
        name: 'Max',
        age: 2,
        favoritePark: 'Treptower Park, Berlin',
      } as Dog,
      {
        name: 'Peanut',
        age: 1,
        favoriteComfyPlace: 'Human bellies',
        profilePicture: 'assets/cat1.jpg',
      } as Cat,
      {
        name: 'Noob',
        age: 3,
        favoriteComfyPlace: 'Window side',
      } as Cat,
      {
        name: 'Carabas',
        age: 7,
        favoritePark: 'Özgürlük, Istanbul',
        profilePicture: 'assets/dog2.jpg',
      } as Dog,
      {
        name: 'Tom',
        age: 4,
        favoritePark: 'Bois de Boulogne, Paris',
        profilePicture: 'assets/dog3.jpg',
      } as Dog,
    ]
  }
}
```

```

    name: 'Mia',
    age: 2,
    favoriteComfyPlace: 'Under the couch',
    profilePicture: 'assets/cat2.jpg',
  } as Cat,
  {
    name: 'Fitz',
    age: 4,
    favoritePark: 'Englischer Garten, Munich',
  } as Dog,
];
}
}

```

نلاحظ البيانات جميعها مجمعة في مكان واحد رغم اختلافها مع وصف كل بيانات بما يناسبها.

بعدما قمنا بوصف وإنشاء البيانات لنقوم الآن بتصميم components حيث نحتاج إلى اثنين components لعرضهما بشكل ديناميكي واحد باسم dog-item والآخر باسم cat-item وكلا هذين components يحتاجان إلى component يكون حاوية يحتوي على هذين components الديناميكيان ويتحكم بعرض أي component في القائمة بناءً على البيانات القادمة له، ولنجعل الأمور بسيطة لنجعل Root Component هو الحاوية بحيث نكتب فيه Logic الخاص بالاتصال بالـ service لجلب البيانات وبنفس الوقت قراءة هذه البيانات وبناءً عليها يقوم ببناء components الديناميكية، كالتالي:

ملف app.component.html

```
<h2 class="d-flex justify-content-center mt-3">List of Members</h2>
```

ملف app.component.ts

```

// tslint:disable: no-string-literal
import { Component, ComponentFactoryResolver, OnInit, ViewContainerRef } from '@angular/core';
import Cat from './cat';
import Dog from './Dog';
import Pet from './pet';
import { MembersService } from './members.service';
import { CatItemComponent } from './cat-item/cat-item.component';
import { DogItemComponent } from './dog-item/dog-item.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {

  private members: Pet[] = [];

  constructor(
    private membersService: MembersService,

```

```

private vcr: ViewContainerRef,
private cfr: ComponentFactoryResolver
) { }

ngOnInit(): void {
  this.members = this.membersService.members;

  this.vcr.clear();

  this.members.forEach((member: (Dog | Cat)) => {
    if (member['favoritePark']) {
      const resolver = this.cfr.resolveComponentFactory(DogItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Dog;
    } else if (member['favoriteComfyPlace']) {
      const resolver = this.cfr.resolveComponentFactory(CatItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Cat;
    }
  });
}
}

```

لنركز على ما هو موجود في الدالة ngOnInit حيث اول خطوة قمنا بقراءة البيانات من service وخزناها في متغير اسميناها members، ومن ثم قمنا بعمل Loop على جميع هذه البيانات وفي كل Loop نخزن القيمة في متغير اسميته member ومن ثم وضعنا شرط لنعرف هذه البيانات المحددة لأي نوع ونستطيع ذلك عن طريق وصف البيانات الذي قمنا به سابقاً حيث ان Dog يحتوي على خاصية خاصة به وهي favoritePark اما Cat فتحتوي على خاصية خاصة بها هي favoriteComfyPlace لذلك نضع هذا الشرط ونرى فإذا كانت البيانات القادمة من اول Loop تحتوي على الخاصية favoritePark فهذا معناه ان هذه البيانات تخص الكلب ومنها نقوم ببناء Component الخاص بعرض بيانات الكلاب ديناميكياً و@Input التي مررناها إلى هذا component هي نفسها البيانات القادمة لنا من اول Loop، ومن ثم سوف يعمل Loop الثاني ويقرأ البيانات ومن ثم يطبق الشروط فإذا كانت البيانات القادمة لا تحتوي على الخاصية favoritePark فسوف ينتقل إلى الشرط الثاني فإذا كانت تحتوي على الخاصية favoriteComfyPlace فهذا معناه ان هذه البيانات القادمة تخص قطعة معينة وبناءً عليها يتم بناء Component الخاص بالقطط ديناميكياً مع تمرير البيانات الخاصة به، ويتم عمل Loop إلى أن ينتهي من جميع البيانات ويكمل جميع عناصر القائمة، بحيث كل عنصر من عناصر القائمة هو عبارة عن component والذي يعرض بيانات يا كلب معين او قط معين.

والآن بعدة هذا الشرح المطول لنقوم بتصميم components الديناميكية، كالتالي:

ملف cat-item.component.ts

```

import { Component, Input, OnInit } from '@angular/core';
import Cat from '../cat';

@Component({

```

```

selector: 'app-cat-item',
templateUrl: './cat-item.component.html',
styleUrls: ['./cat-item.component.css']
}))

export class CatItemComponent implements OnInit {

  @Input() member: Cat;

  constructor() { }

  ngOnInit(): void {
  }

}

```

نلاحظ القيمة الداخلة إلى هذا component والتي تأخذ قيمتها في كل مرة نعمل Loop ونبني component ديناميكياً، كما أشرنا سابقاً

```

ngOnInit(): void {
  this.members = this.membersService.members;

  this.vcr.clear();

  this.members.forEach((member: (Dog | Cat)) => {
    if (member['favoritePark']) {
      const resolver = this.cfr.resolveComponentFactory(DogItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Dog;
    } else if (member['favoriteComfyPlace']) {
      const resolver = this.cfr.resolveComponentFactory(CatItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Cat;
    }
  });
}

```

اما ملف template فيكون، كالتالي:

ملف cat-item.component.html

```

<ul class="list-group p-1 mb-0">
  <li class="list-group-item m-auto w-75 p-0 pt-2 shadow-sm bg-white rounded">
    <img
      class="mr-2 ml-2 position-relative"
      style="top: -25px; display: inline;"
      width="50"
      height="50"
      [src]="member.profilePicture || 'assets/cat.png'"
    >

```

```

/>
<div style="display: inline-block;">
  <p class="m-0">{{ member.name }}</p>
  <p class="m-0">Age: {{ member.age }}</p>
  <p class="m-0">
    Favorite Comfy Place: {{ member.favoriteComfyPlace }}
  </p>
</div>
</li>
</ul>

```

وبنفس الطريقة في component الآخر ذو الاسم dog-item، كالتالي:

ملف dog-item.component.ts

```

import { Component, Input, OnInit } from '@angular/core';
import Dog from '../dog';

@Component({
  selector: 'app-dog-item',
  templateUrl: './dog-item.component.html',
  styleUrls: ['./dog-item.component.css']
})

export class DogItemComponent implements OnInit {

  @Input() member: Dog;

  constructor() { }

  ngOnInit(): void {
  }

}

```

وما قيل سابقاً يُقال هنا من ناحية تغذية هذا component بالبيانات عن طريق @Input، كالتالي:

```

ngOnInit(): void {
  this.members = this.membersService.members;

  this.vcr.clear();

  this.members.forEach((member: (Dog | Cat)) => {
    if (member['favoritePark']) {
      const resolver = this.cfr.resolveComponentFactory(DogItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Dog;
    } else if (member['favoriteComfyPlace']) {
      const resolver = this.cfr.resolveComponentFactory(CatItemComponent);
      const createComponent = this.vcr.createComponent(resolver);
      createComponent.instance.member = member as Cat;
    }
  });
}


```

#### ملف dog-item.component.html

```
<ul class="list-group p-1 mb-0">
  <li class="list-group-item m-auto w-75 p-0 pt-2 shadow-sm bg-white rounded">
    <img
      class="mr-2 ml-2 position-relative"
      style="top: -25px; display: inline;"
      width="50"
      height="50"
      [src]="member.profilePicture || 'assets/dog.png'"
    />
    <div style="display: inline-block;">
      <p class="m-0">{{ member.name }}</p>
      <p class="m-0">Age: {{ member.age }}</p>
      <p class="m-0">Favorite Park: {{ member.favoritePark }}</p>
    </div>
  </li>
</ul>
```

والنتيجة النهائية، تكون:


### List of Members



Rex

Age: 5


Favorite Park: Central Park, New York



Max

Age: 2


Favorite Park: Treptower Park, Berlin



Peanut

Age: 1


Favorite Comfy Place: Human bellies



Noob

Age: 3


Favorite Comfy Place: Window side



Carabas

Age: 7


Favorite Park: Özgürlük, Istanbul



Tom

Age: 4


Favorite Park: Bois de Boulogne, Paris



Mia

Age: 2

Favorite Comfy Place: Under the couch



Fitz

Age: 4

Favorite Park: Englischer Garten, Munich

Elements

Console

Sources

Network

Filter

Default levels

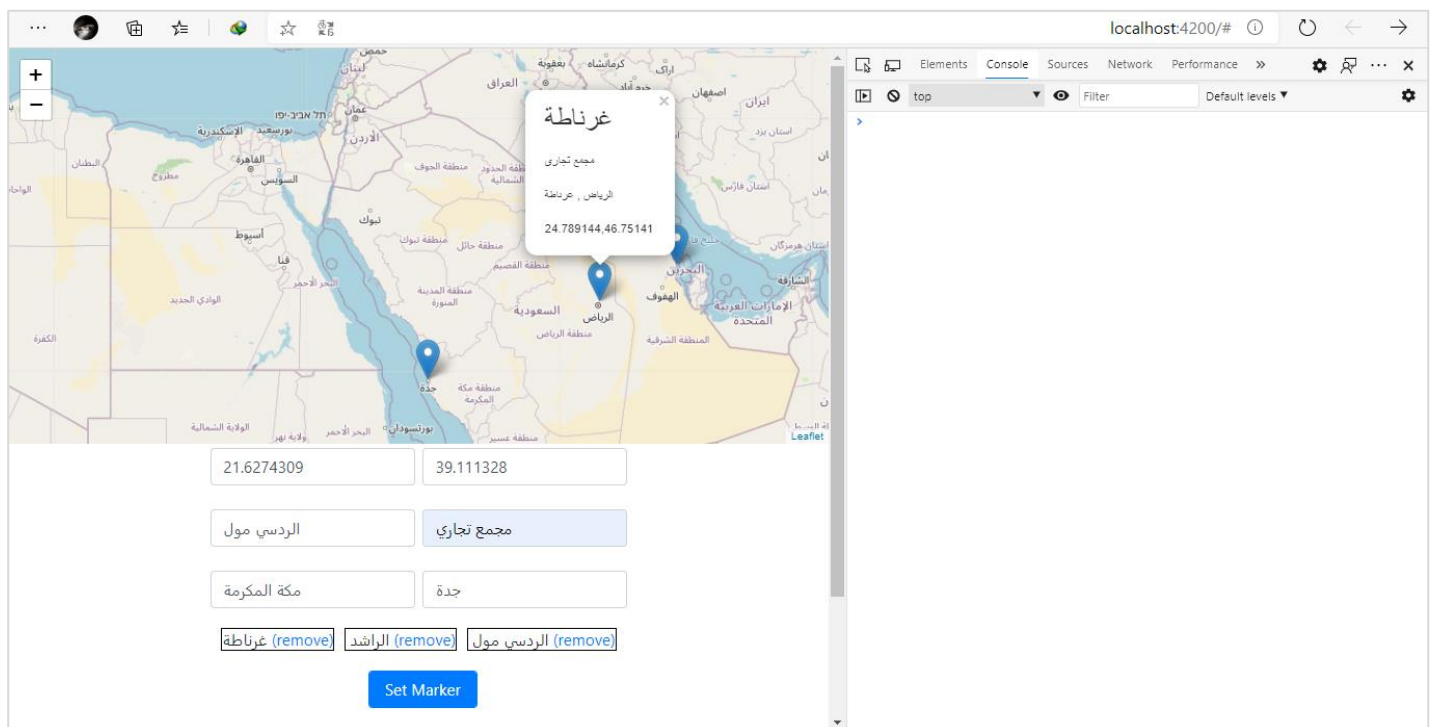
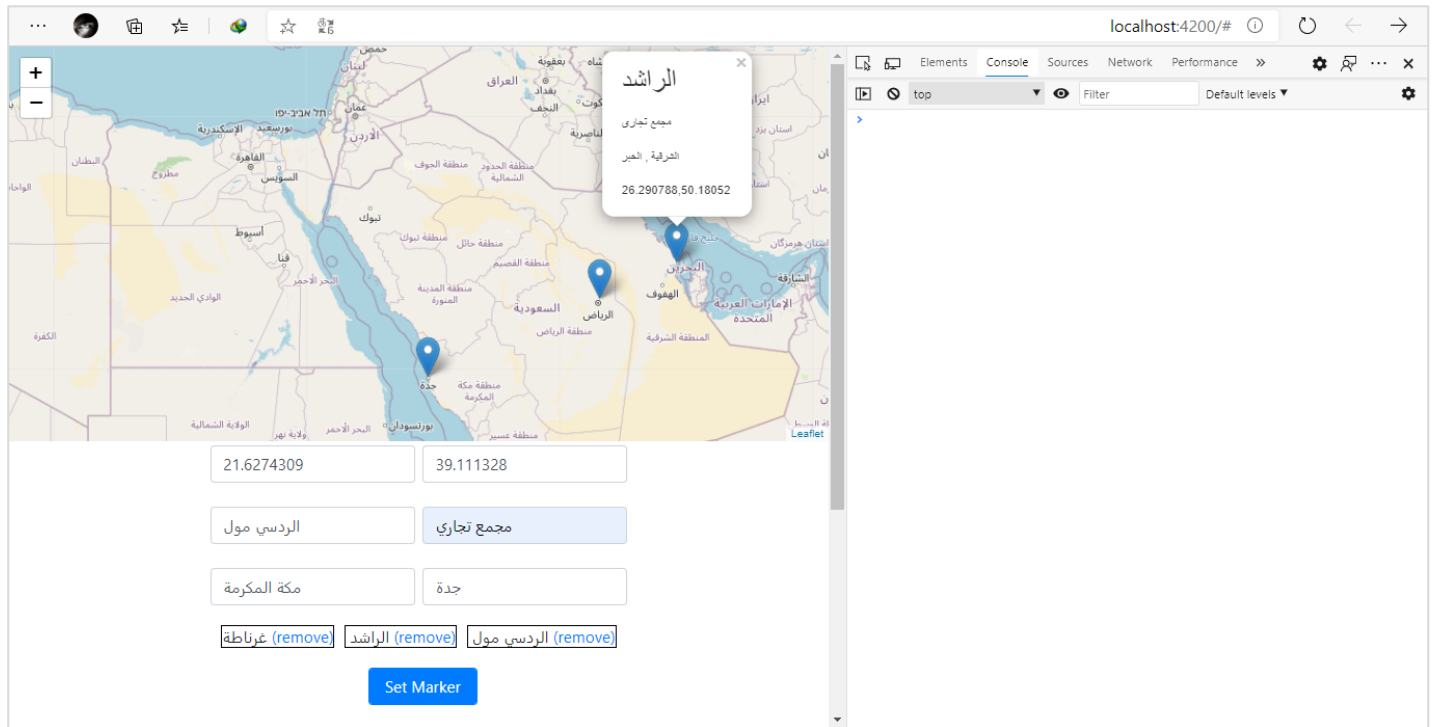
نلاحظ ظهرت لنا البيانات على شكل مجموعة من components بحيث كل عنصر في القائمة هو عبارة component مستقل بذاته، تم عرضه بشكل ديناميكي بناءً على البيانات القادمة من service والتي عملنا لها Loop بحيث مع كل Loop يتم انشاء عنصر من هذه القائمة على شكل component.

ملاحظة: استخدمت مكتبة bootstrap لتنسيق مظهر التطبيق.

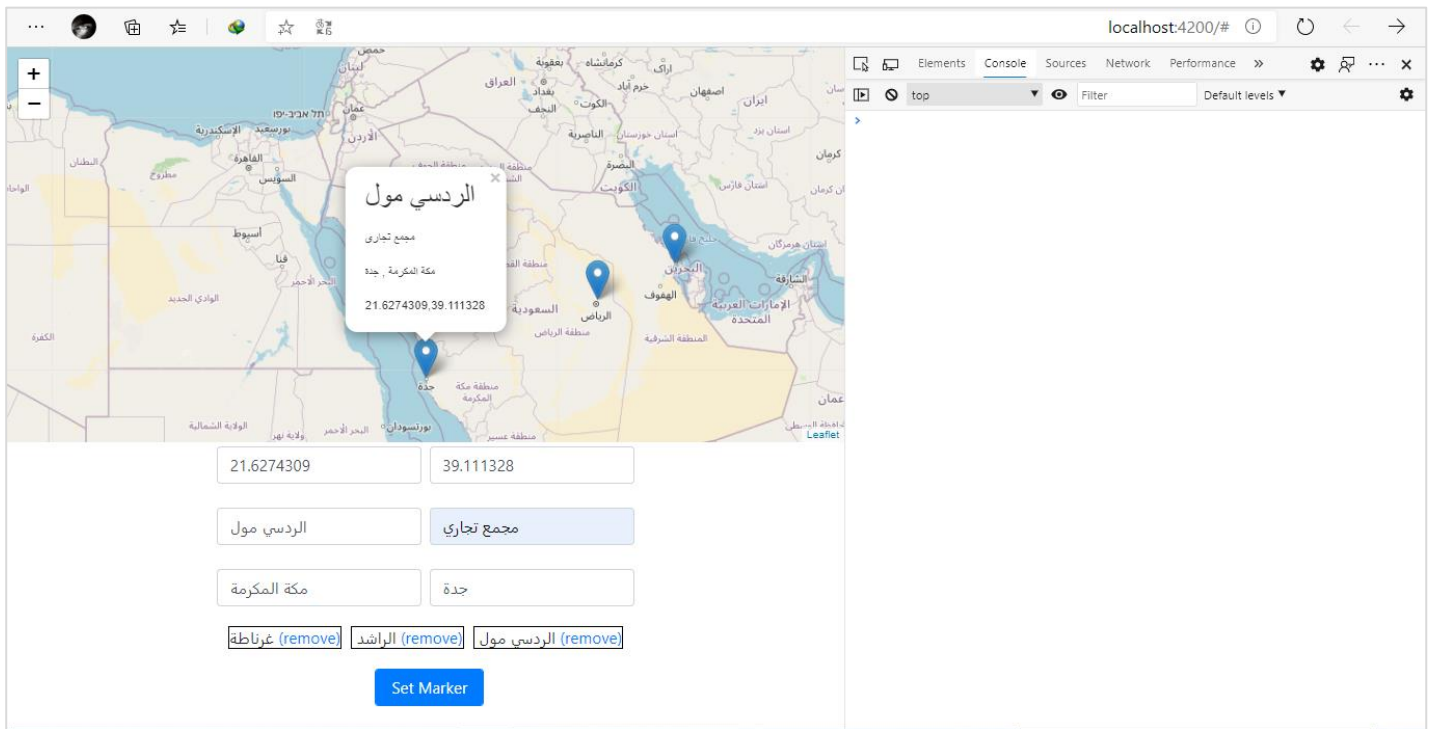
#### 6.13.4. المثال السادس:

وهو عن كيفية الاستفادة من Angular Dynamic Components من خلال مكتبة خارجية، وتقوم فكرة هذا المثال انه في حال اردنا استخدام مكتبة Leaflet لعرض الخرائط في التطبيق الخاص بنا، فإننا نريد مع اسقاط كل دبوس على الخريطة فإن لوحة البيانات التي تظهر فوق هذا الدبوس تكون عبارة عن component ديناميكي، بحيث يقوم بقراءة بيانات الموقع المُسقط عليه هذا الدبوس على الخريطة ويظهرها بشكل ديناميكي لكل دبوس على حدا.

بحيث يكون النتيجة النهائية للمثال، كالتالي:







نلاحظ ان لوحة المعلومات التي تظهر عند الضغط على كل دبوس هي المقصودة سابقاً بأنها عبارة عن component ديناميكي، بمعنى آخر انه يوجد لدينا component واحد نقوم بعرضه ديناميكياً لكل دبوس على الخريطة بحيث يستسقي بياناته في كل مره من هذا الدبوس على الخريطة، والسبب في استخدامنا Dynamic Components لأن هذه الدبابيس تضاف هي ايضاً بشكل ديناميكي كما هو واضح في الصور السابقة حيث يقوم المستخدم بإضافة بيانات الدبوس يدوياً ومن ثم يضغط على الزر Set Marker ويتم اسقاط الدبوس تلقائياً على الخريطة، لذلك نحن المطورين لا نعلم كم عدد وبيانات هذا الدبابيس لكي نجعل لكل دبوس component خاص به، ومن هذا المنطلق استفدنا من Dynamic Components بحيث مع كل دبوس يتم اضافته نقوم بتخليق نسخة من هذا component لهذا الدبوس.

وبالاختلاف عن الأمثلة الأخرى هنا نحتاج أن نحول هذا component إلى شكله native او HTML بحيث نستطيع عرضه عن طريق هذه المكتبة.

وأخيراً لنكتفي من الشرح النظري ولنبدأ بالشرح عملياً.

وأول خطوة هي انشاء مشروع Angular جديد وإضافة هذه المكتبة إليه، ولمعرفة طريقة إضافة هذه المكتبة لمشروع Angular، تستطيع مراجعة هذا الرابط <https://github.com/Asymmetrik/ngx-leaflet#leafletcenter-latlng>

وبعد تثبيت هذه المكتبة في مشروعنا سوف نحتاج إلى استدعاء ملف style الخاص بهذه المكتبة عن طريق كتابة هذا الأمر في ملف style.css، كالتالي:

```
ملف style.css
@import 'bootstrap/dist/css/bootstrap.css';
@import 'leaflet/dist/leaflet.css';
```

نلاحظ ايضاً اضفنا مكتبة bootstrap لكي نستفيد منها في بناء النموذج الخاص بإضافة بيانات الدبوس.

وبعد إضافة هذه البيانات سوف نحتاج إلى interface أو class (جميعها تؤدي نفس الغرض) لكي نقوم بعمل وصف للبيانات التي سوف تظهر في component الديناميكي، لذلك لنقوم بإضافة interface وليكن اسمه marker عن طريق كتابة هذا الأمر في terminal الخاص بـ ng g interface marker، وعند إنشاء هذا الملف نضيف الوصف إليه، كالتالي:

ملف marker.ts

```
import { LatLngExpression } from 'leaflet';

export interface IMarker {
  name: string;
  description: string;
  position: LatLngExpression;
  city: string;
  district: string;
}
```

الخاصية position هي من النوع LatLngExpression وهو من الأنواع المضمنة مع هذه المكتبة والتي تدل على أن هذه الخاصية التي تكون من هذا النوع تحمل قيم الأحداثيات لموقع الدبوس، وايضاً نلاحظ أننا أضفنا حرف (I) قبل اسم هذا interface وفي الحقيقة هذا ليس شرط وإنما عرفتم التعارف عليه بين مجتمع المطورين عند التعامل مع interfaces.

ومن ثم في الخطوة التالية لنقوم بإنشاء component الديناميكي وبما أننا قلنا سابقاً أن هذا component لا بد أن نقوم بتحويله إلى شكله native أي HTML لكي نستطيع عرضه داخل هذه المكتبة لذلك لنقوم بتسميته html-marker، وبعد إنشاء هذا component لنقوم بتمرير له @Input وهو عبارة عن كائن من النوع IMarker وهو عبارة عن interface الذي قمنا بإنشائه قبل قليل، بحيث مع كل إنشاء لدبوس معين نمرر بياناته إلى هذا component لكي يعرضها، أما الآن لنقم بتمرير @Input وتصميم صفحة template، كالتالي:

ملف html-marker.component.ts

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

@Component({
  selector: 'app-html-marker',
  templateUrl: './html-marker.component.html',
  styleUrls: ['./html-marker.component.css']
})
export class HtmlMarkerComponent implements OnInit {

  @Input() data: IMarker;

  constructor() { }

  ngOnInit(): void {
  }

}
```

نلاحظ اننا مررنا كائن اسميناه data، اما ملف template لهذا component فنجري به التعديلات التالية:

ملف html-marker.component.html

```
<h3> {{ data.name }} </h3>
<p> {{ data.description }} </p>
<p>
  <span>{{ data.city }}</span>
  ,
  <span>{{ data.district }}</span>
</p>
<p> {{ data.position }} </p>
```

وبذلك اصبح هذا component جاهز، وبقي الجزء المهم وهو طريقة بناء هذا component ديناميكياً وعرضه داخل هذا المكتبة، وسوف نكتب هذا Logic في Root Component، ولكن قبلها لنقوم بتصميم ملف template، مع العلم انني استخدمت مكتبة bootstrap لتنسيق ونهج Angular Template Driven Forms في بناء النموذج (لفهم اعمق الرجاء مراجعة كتابي Angular Forms وهو جزء من هذه السلسلة)، كالتالي:

ملف app.component.html

```
<div
  style="height: 400px; width: auto;"
  leaflet
  [leafletCenter]="latlang"
  [leafletOptions]="options"
  (leafletMapReady)="map = $event"
>
</div>
```

1

```
<form
  #form="ngForm"
  class="d-flex flex-column justify-content-center align-items-center"
>
```

2

```
  <div class="form-group form-inline">
    <input
      ngModel
      name="latitude"
      class="form-control m-1"
      type="text"
      placeholder="Latitude"
    />
    <input
      ngModel
      name="longitude"
      class="form-control m-1"
      type="text"
      placeholder="Longitude"
    />
  </div>
```

3

```

<div class="form-group form-inline">
  <input
    class="form-control m-1"
    ngModel
    name="name"
    type="text"
    placeholder="Name"
  />
  <input
    ngModel
    name="description"
    class="form-control m-1"
    type="text"
    placeholder="Description"
  />
</div>

```

4

```

<div class="form-group form-inline">
  <input
    class="form-control m-1"
    ngModel
    name="city"
    type="text"
    placeholder="City"
  />
  <input
    ngModel
    name="district"
    class="form-control m-1"
    type="text"
    placeholder="district"
  />
</div>

```

5

```

<div class="form-group">
  <span
    *ngFor="let marker of markers"
    style="border: 1px solid black; margin: 5px;">
    {{ marker.name }}
    <a href="#" (click)="removeMarker(marker)">(remove)</a>
  </span>
</div>

```

6

```

<div class="form-group">
  <button
    class="btn btn-primary form-control m-1"
    (click)="addMarker(form)">
    Set Marker
  </button>
</div>

```

7

```

</form>

```

8

- ① هذا الجزء خاص بعرض الخريطة عن طريق المكتبة Leaflet ولمعرفة تفاصيل أكثر تستطيع مراجعة الرابط السابق.
- ② بداية النموذج ونلاحظ استخدمنا أسلوب Angular Driven Forms، عن طريق إضافة Template Reference لهذا النموذج باسم form واسندنا له القيمة ngForm.
- ③ الجزء من النموذج الخاص بإدخال احداثيات الدبوس.
- ④ الجزء من النموذج الخاص بإدخال بيانات اسم ووصف الموقع للدبوس على الخريطة.
- ⑤ الجزء من النموذج الخاص بإدخال بيانات اسم المدينة والمنطقة لموقع الدبوس على الخريطة.
- ⑥ في هذا الجزء عملنا حلقة Loop على مصفوفة سوف ننشئها بعد قليل في ملف class باسم markers والفائدة من هذه المصفوفة هو لتخزين اسم الموقع ونسخة من component الديناميكي ونسخة من الدبوس المضاف لكي نستطيع حذفه مستقبلاً، ويتم تغذية هذه المصفوفة بالبيانات المطلوبة عند الضغط على زر Set Marker لإضافة الدبوس، وبنفس الوقت يتم إضافة دالة لحذف هذا الدبوس وذلك بمعرفة الكائن الذي نمرره لهذه الدالة في كل Loop يتم عمله وتخليق نسخة من هذا الزر وباقي markup.
- ⑦ في هذا الجزء يتم تنفيذ الدالة الخاصة بإضافة نسخة من كلاً من الدبوس والcomponent الديناميكي مع البيانات التي يحتويها وبنفس الوقت يتم تغذية المصفوفة markers بالبيانات اللازمة، ونلاحظ مررنا المتغير form وهو الذي قمنا بإنشائه في النقطة (1) حيث يمثل النموذج مع جميع القيم الموجودة بداخله.
- ⑧ وآخر نقطة نهاية النموذج.

وبما اننا اشرنا إلى المصفوفة markers لنقم بإنشاء ملف لوصف البيانات الموجودة في هذه المصفوفة، كما فعلنا سابقاً، لذلك لنقم بإنشاء interface وليكن اسمه marker-meta-data، ونضيف الوصف التالي:

```

marker-meta-data.ts ملف
import { ComponentRef } from '@angular/core';
import { Marker } from 'leaflet';
import { HtmlMarkerComponent } from './html-marker/html-marker.component';

export interface IMarkerMetaData {
  name: string;
  markerInstance: Marker;
  componentInstance: ComponentRef<HtmlMarkerComponent>;
}

```

هذا الملف يحتوي على ثلاث خصائص هي name وهو اسم الدبوس، ونسخة من الدبوس ونسخة من component الديناميكي، وكما قلنا سابقاً سوف نستفيد منه لحفظ الدبوس الذي يتم تخليقه ديناميكياً لحذفه مستقبلاً.

الآن بعدما قمنا بعمل وصف لهذه البيانات، لنرجع إلى ملف class لي Root Component لكتابة Logic الخاص بإضافة الدبوس والcomponent الديناميكي، كالتالي:

```
// tslint:disable: no-inferable-types no-string-literal
import { Component, ComponentFactoryResolver, OnInit } from '@angular/core';
import { icon, latLng, marker, tileLayer, Marker, LatLng } from 'leaflet';
import { IMarkerMetaData } from './marker-meta-data';
import { IMarker } from './marker';
import { HtmlMarkerComponent } from './html-marker/html-marker.component';
import { NgForm } from '@angular/forms';
import { ViewContainerRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  options = {
    layers: [
      tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png')
    ],
    zoom: 5,
    center: latLng(26.709879, 28.260116)
  };

  MARKERS_DATA: IMarker[] = [];
  markers: IMarkerMetaData[] = [];
  map: any;
  latlang: LatLng;

  constructor(
    private resolver: ComponentFactoryResolver,
    private vcr: ViewContainerRef,
  ) { }

  ngOnInit(): void {
    const iconRetinaUrl = 'assets/marker-icon-2x.png';
    const iconUrl = 'assets/marker-icon.png';
    const shadowUrl = 'assets/marker-shadow.png';
    const iconDefault = icon({
      iconRetinaUrl,
      iconUrl,
      shadowUrl,
      iconSize: [25, 41],
      iconAnchor: [12, 41],
      popupAnchor: [1, -34],
      tooltipAnchor: [16, -28],
      shadowSize: [41, 41]
    });
    Marker.prototype.options.icon = iconDefault;
  }
}
```

1

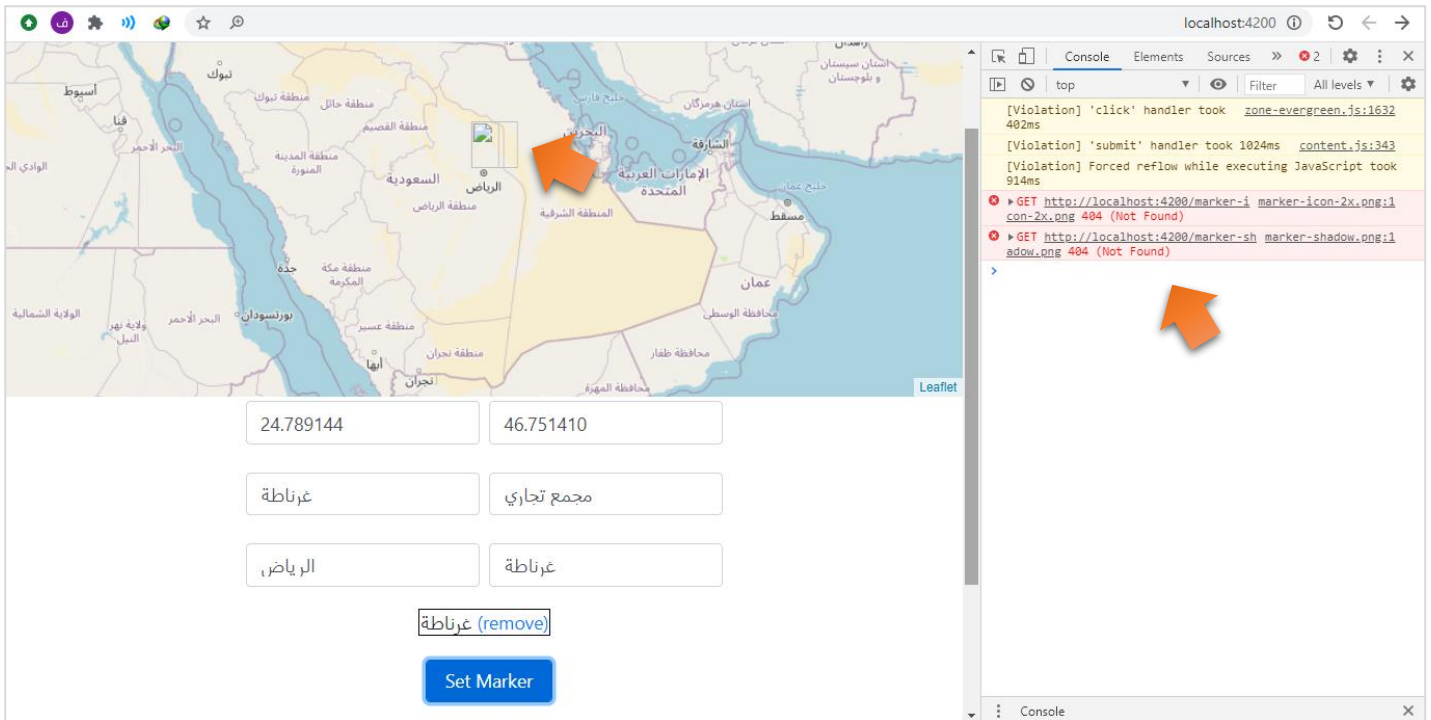
2

3

① تعريف كائن يحتوي على مجموعة من الخصائص تحتاجها المكتبة لكي تقوم ببناء الخريطة، وقد قمنا بتمرير هذا الكائن في ملف template في النقطة (1).

② تعريف مجموعة من الخصائص (المتغيرات) منها متغير اسميته MARKER\_DATA وهو عبارة عن مصفوفة من النوع IMarker وهو عبارة عن interface الذي قمنا ببنائه سابقاً، ومهمة هذا المتغير هو تخزين البيانات الخاصة بالدبوس من الاسم والموقع،.. الخ، ومن ثم تمريرها إلى component الديناميكي عن طريق @Input الذي عرفناه سابقاً لكي يقوم بعرض البيانات، ومنها أيضاً المصفوفة التي اسميناها markers وقد اشرت لها سابقاً حيث نخزن فيها اسم الموقع ونسخة من الدبوس ونسخة من component الديناميكي بحيث نستفيد منها لحذف هذا الدبوس مستقبلاً، ومنها متغير باسم map وتحتاجه المكتبة لبناء الخريطة وتأخذ قيمتها من ملف template لهذا component في النقطة (1) – (map = \$event)، ومنها متغير اسميته latlang وهو من النوع LatLang ونستفيد منه لكي نقوم بتحريك الخريطة إلى موضع الدبوس في كل مرة نضيف دبوس جديد.

③ هذا الجزء خاص بعرض الدبوس على الخريطة او بصيغة أخرى صورة او شكل الدبوس لأنه قد يعمل التطبيق بدون أي مشاكل ولكن الصورة لا تعمل، كما في الشكل التالي:



نلاحظ الصورة الخاصة بهذا الدبوس لا تظهر، مع العلم انه هنالك بعض الخطوات التي نقوم بها قبل كتابة هذا الكود لكي نستطيع حل هذه المشكلة، وللإطلاع على كيفية تنفيذ هذه الخطوات والكود السابق لحل هذه المشكلة الرجاء مراجعة هذا

الرابط <https://stackoverflow.com/questions/41144319/leaflet-marker-not-found-production-env>

جميع الذي اضعناه Logic السابق هو تهيئة لأهم خطوتين وهما الدالة addMarker التي تُنفذ عند الضغط على زر Set Marker والدالة removeMarker لحذف الدبوس المحدد، وسوف نبدأ بأول دالة وهي إضافة الدبوس للخريطة وإنشاء component ديناميكي وتمرير البيانات المحددة، كالتالي:

```
app.component.ts ملف
// tslint:disable: no-inferable-types no-string-literal
```

```

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

import { icon, latLng, marker, tileLayer, Marker, LatLng } from 'leaflet';
import { IMarkerMetaData } from './marker-meta-data';
import { IMarker } from './marker';
import { HtmlMarkerComponent } from './html-marker/html-marker.component';
import { NgForm } from '@angular/forms';
import { ViewContainerRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  options = {
    layers: [
      tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png')
    ],
    zoom: 5,
    center: latLng(26.709879, 28.260116)
  };

  MARKERS_DATA: IMarker[] = [];
  markers: IMarkerMetaData[] = [];
  map: any;
  latlang: LatLng;

  constructor(
    private resolver: ComponentFactoryResolver,
    private vcr: ViewContainerRef,
  ) { }

  ngOnInit(): void {
    const iconRetinaUrl = 'assets/marker-icon-2x.png';
    const iconUrl = 'assets/marker-icon.png';
    const shadowUrl = 'assets/marker-shadow.png';
    const iconDefault = icon({
      iconRetinaUrl,
      iconUrl,
      shadowUrl,
      iconSize: [25, 41],
      iconAnchor: [12, 41],
      popupAnchor: [1, -34],
      tooltipAnchor: [16, -28],
      shadowSize: [41, 41]
    });
    Marker.prototype.options.icon = iconDefault;
  }

```



```

}

addMarker(form: NgForm) {
  this.MARKERS_DATA.push(
    {
      name: form.value.name,
      description: form.value.description,
      position: [+form.value.latitude, +form.value.longitude],
      city: form.value.city,
      district: form.value.district,
    }
  );

  this.latlang = latLng(+form.value.latitude, +form.value.longitude); ❷

  const index = this.MARKERS_DATA.length - 1; ❸

  const factory = this.resolver.resolveComponentFactory(HtmlMarkerComponent); ❹

  const component = this.vcr.createComponent(factory); ❺

  component.instance.data = this.MARKERS_DATA[index]; ❻

  component.changeDetectorRef.detectChanges(); ❼

  const m = marker(this.MARKERS_DATA[index].position); ❽

  m.bindPopup(component.location.nativeElement).openPopup(); ❾

  m.addTo(this.map); ❿

  this.markers.push({
    name: this.MARKERS_DATA[index].name,
    markerInstance: m,
    componentInstance: component
  });
}

```

- ❶ نغذي المصفوفة الخاصة بتخزين بيانات الدبوس بالقيم التي قام المستخدم بإدخالها في النموذج على شكل كائن object.
- ❷ نقوم بتوجيه الخريطة إلى موضع الدبوس بالاستناد إلى الاحداثيات المدخلة في النموذج، وتخزين هذه القيمة في المتغير latlang وقد قمنا بعمل له property binding سابقاً في ملف template لهذا component في النقطة (1).
- ❸ بعد إضافة البيانات إلى المصفوفة في النقطة (1) نحتاج إلى معرفة index الخاص بهذا الكائن، لأننا نحتاج إلى تمرير أو الاستفادة من بعض أو كل هذه البيانات (الكائن) في الخطوات التالية.
- ❹ ❺ ❻ هذه النقاط اشبعناها شرحاً ولعل الشيء الجديد هو ما هو موجود في النقطة (6) حيث قمنا بتمرير جميع البيانات الخاصة بهذا الكائن ذو index المحدد الذي اضفناه في الأخير.

7 بما اننا نتعامل مع مكتبة خارجية لا تخضع لنظام Change Detection الخاص بالـ Angular فنحتاج إلى ان نقوم بتشغيل هذا النظام يدوياً لكي يتم تحديث مجتوى template وإظهار أي تعديلات لمستخدم التطبيق، وهناك عدة طرق للقيام بهذا الأمر منها استخدام NgZone وبالتحديد الدالة run() او باستخدام الطريق الأقصر والأبسط وكتابة هذا الأمر الموجود في هذه النقطة.

8 هنا نحتاج ان نعمل نسخة من الدبوس وذلك بالاعتماد على الاحداثيات الموجودة في الخاصية position الموجودة في هذا الكائن ذو index المحدد ومن ثم قمنا بتخزين هذه النسخة في متغير اسميناها m.

9 صحيح اننا قمنا بتخليق component ديناميكياً عن طريق أخذ نسخة منه وايضاً قمنا بتمرير البيانات له، ولكن قبل ان نمرر هذه البيانات لدبوس المحدد نحتاج إلى أن نحول هذا component إلى شكله native لأن هذه المكتبة لا تستقبل إلا نص او HTML Markup، لذلك لابد ان نقوم بتحويلها وبنفس الوقت تمرير هذه القيم بعد التحويل.

10 وأخيراً نضيف هذا الدبوس إلى الخريطة والمتمثلة في المتغير map، والذي أخذ قيمته عن طريق ملف template في النقطة (1)، لكي يظهر للمستخدم.

اما النقطة الأخيرة فهي تغذية المصفوفة markers بالبيانات، وكما قلنا سابقاً نحتاج هذه المصفوفة لكي نخزن نسخة من الدبوس ونسخة من component الديناميكي واسم الموقع، لكي نستطيع الوصول له وحذفه لاحقاً.

بعدما قمنا بشرح دالة إضافة الدبوس بقي علينا أخيراً كتابة Logic الخاص بحذف هذا الدبوس وجميع ما يرتبط به، كالتالي:

```
app.component.ts ملف
// tslint:disable: no-inferable-types no-string-literal
import {
  Component,
  ComponentFactoryResolver,
  OnInit,
} from '@angular/core';

import { icon, latLng, marker, tileLayer, Marker, LatLng } from 'leaflet';
import { IMarkerMetaData } from './marker-meta-data';
import { IMarker } from './marker';
import { HtmlMarkerComponent } from './html-marker/html-marker.component';
import { NgForm } from '@angular/forms';
import { ViewContainerRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})

export class AppComponent implements OnInit {
  options = {
    layers: [
      tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png')
    ],
  },
}
```

```

        zoom: 5,
        center: LatLng(26.709879, 28.260116)
    });

    MARKERS_DATA: IMarker[] = [];
    markers: IMarkerMetaData[] = [];
    map: any;
    latlang: LatLng;

    constructor(
        private resolver: ComponentFactoryResolver,
        private vcr: ViewContainerRef,
    ) { }

    ngOnInit(): void {
        const iconRetinaUrl = 'assets/marker-icon-2x.png';
        const iconUrl = 'assets/marker-icon.png';
        const shadowUrl = 'assets/marker-shadow.png';
        const iconDefault = icon({
            iconRetinaUrl,
            iconUrl,
            shadowUrl,
            iconSize: [25, 41],
            iconAnchor: [12, 41],
            popupAnchor: [1, -34],
            tooltipAnchor: [16, -28],
            shadowSize: [41, 41]
        });
        Marker.prototype.options.icon = iconDefault;
    }

    addMarker(form: NgForm) {
        this.MARKERS_DATA.push(
            {
                name: form.value.name,
                description: form.value.description,
                position: [+form.value.latitude, +form.value.longitude],
                city: form.value.city,
                district: form.value.district,
            }
        );
        this.latlang = LatLng(+form.value.latitude, +form.value.longitude);
        const index = this.MARKERS_DATA.length - 1;
        const factory = this.resolver.resolveComponentFactory(HtmlMarkerComponent);
        const component = this.vcr.createComponent(factory);
        component.instance.data = this.MARKERS_DATA[index];
        component.changeDetectorRef.detectChanges();
        const m = marker(this.MARKERS_DATA[index].position);
        m.bindPopup(component.location.nativeElement).openPopup();
        m.addTo(this.map);
        this.markers.push({
            name: this.MARKERS_DATA[index].name,
            markerInstance: m,

```

```

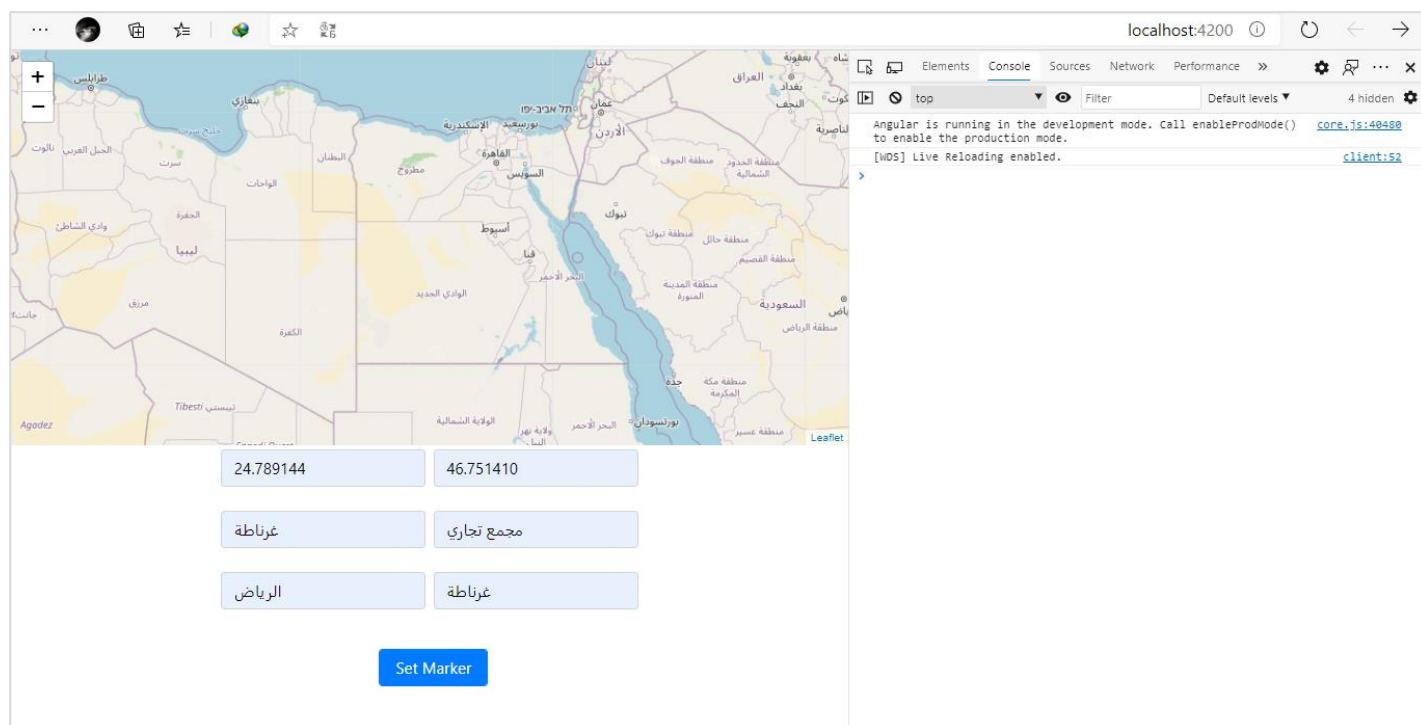
componentInstance: component
});
}

removeMarker(_marker) {
  const idx = this.markers.indexOf(_marker);
  this.markers.splice(idx, 1);
  this.MARKERS_DATA.splice(idx, 1);
  _marker.markerInstance.removeFrom(this.map);
  _marker.componentInstance.destroy();
}
}

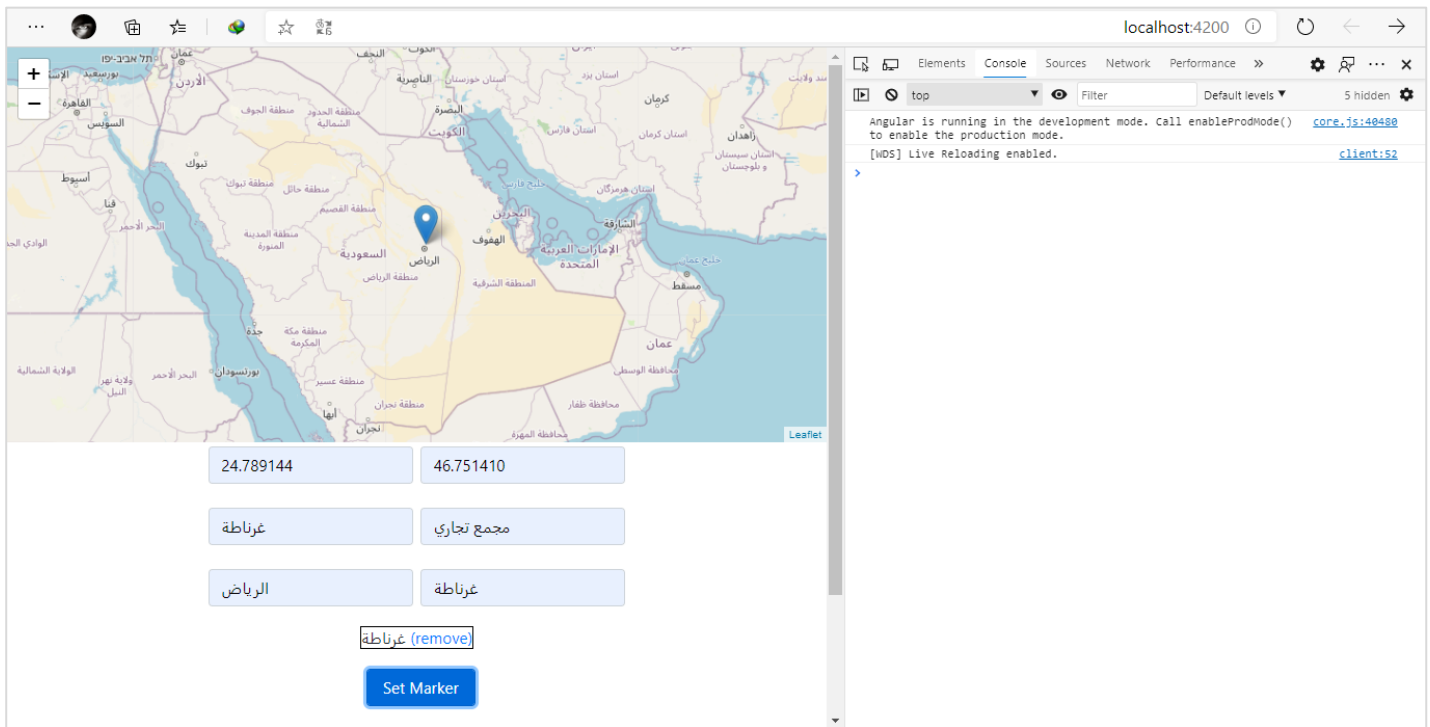
```

كما هو واضح، قمنا في البداية بالحصول على index الخاص بالدبوس المحدد والمتمثل في البارامتر \_marker (تستطيع اختيار أي اسم تريده) ومن بعدما حصلنا على index استطعنا بذلك من حذفه من كلا المصفوفتين، ومن ثم حذف نسخة الدبوس، وأخيراً حذف نسخة component الديناميكي.

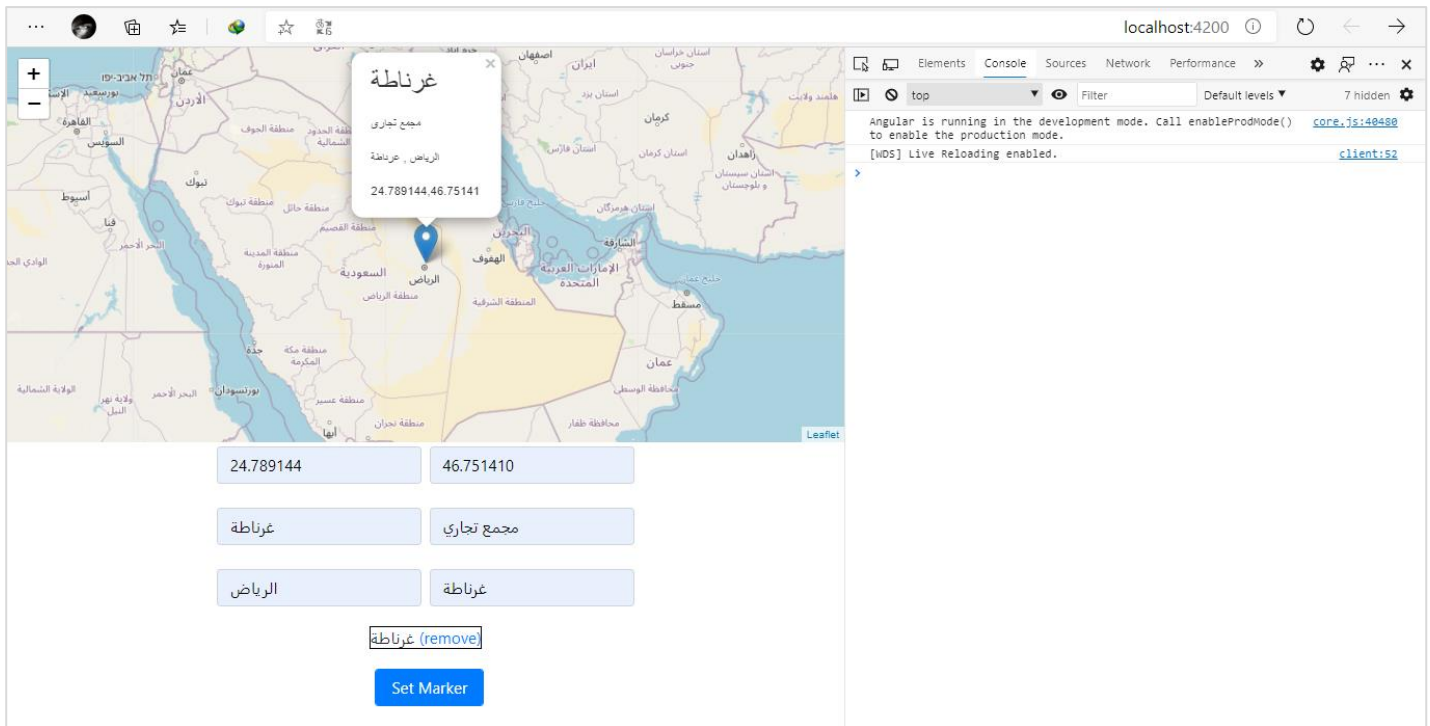
والآن لنذهب إلى المتصفح، ولنرى النتيجة، كالتالي:



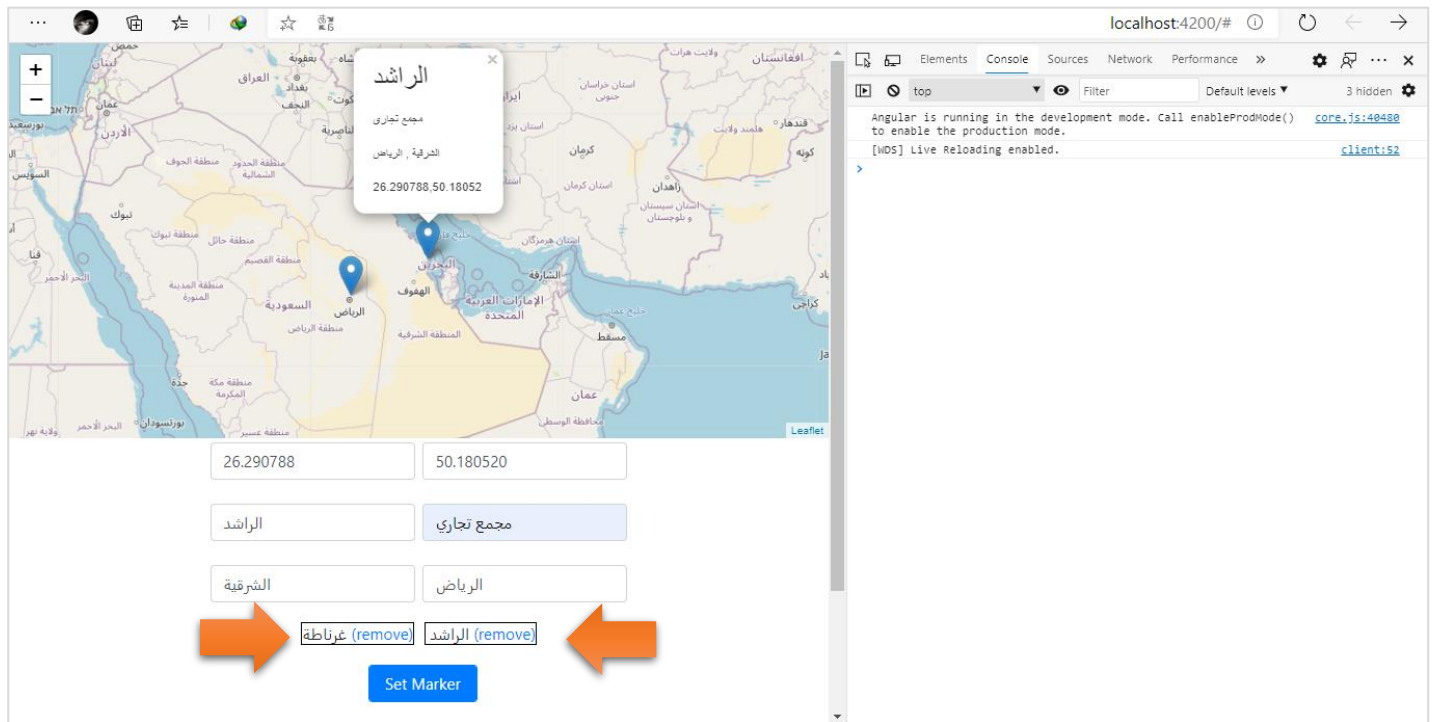
في البداية كما هو واضح قمنا بإضافة بعض البيانات إلى النموذج، والآن لنقم بالضغط على الزر Set Marker ولنرى النتيجة:



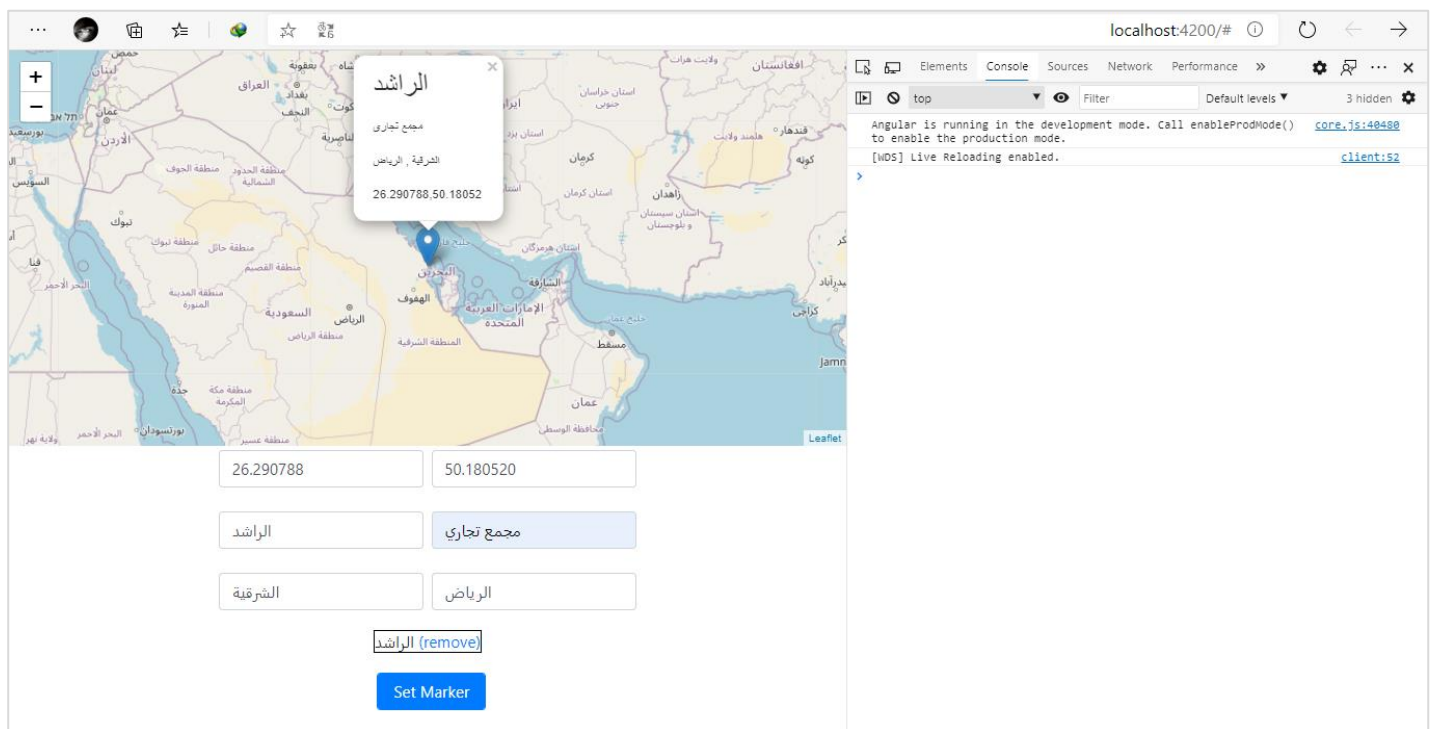
نلاحظ تم إضافة الدبوس وتوجيه الخريطة للإحداثيات المدخلة، وبنفس الوقت ظهر لنا اسم الموقع وبجانبه remove في حال اردنا حذف هذا الدبوس، والآن لنقوم بالنقر على نفس الدبوس ونرى النتيجة، كالتالي:



نلاحظ ظهر لنا component الديناميكي على شكل HTML Markup، والآن لنقوم بإضافة بيانات أخرى ولنرى النتيجة، كالتالي:



وايضاً نستطيع حذف أي من هذه الدبابيس بالضغط على remove بجانب اسم الموقع المراد حذفه.



وبذلك انهيينا هذا المثال الذي من خلاله عرفنا كيف نستفيد من Angular Dynamic Components مع المكتبات الخارجية.

# المراجع

## References

## References:

1. Freeman, Adam, **Pro Angular 9**, Apress, 2020.
2. Agarwal, Uttam, **Hands-On Full Stack Development with Angular 5 and Firebase**, Packt, 2018.
3. Delaney, Jeff, **The Angular Firebase Survival Guide**, leanpub, 2018.
4. Saha, Depasis, **Angular 7.0 for Beginners**, 2018.
5. Vijay, Santhosh, **Material for Angular Developer Course**, Leanpub, 2019.
6. Hussain, Asim, **Angular From Theory to Practice**, 2017.
7. D. Booth, Joseph, **Angular Succinctly**, Syncfusion, 2019.
8. Squad, Ninja, **Become a ninja with Angular**, 2019.
9. Steyer, Manfred, **Enterprise Angular**, Leanpub, 2020.
10. Clow, Mark, **Angular 5 Projects**, Apress 2018.
11. Nate Murray, Felipe Coury, Ari Lerner, and Carlos Taborda, **ng-book The Complete Guide to Angular**, Fullstack.io, 2018.
12. Bouchefra, Ahmed, **Practical Angular: Build your first web apps with Angular 8**, Leanpub, 2019.
13. Hajian, Majid, **Progressive Web Apps with Angular**, Apress, 2019.
14. Savkin, Victor, **Angular Router**, Leanpub, 2018.