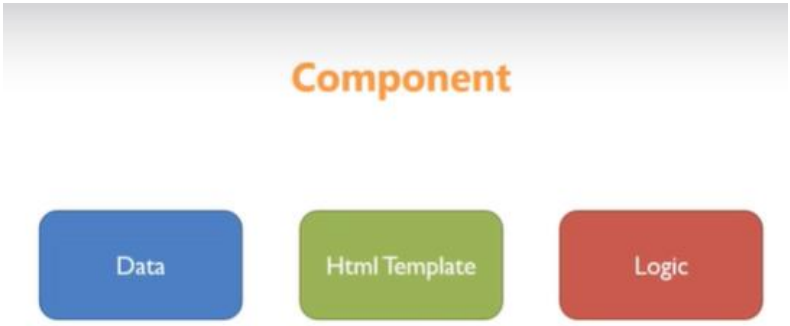
Önceki kısımda todoApp ile angular uygulamalarına bir giriş yaptık ancak, detay vermemiştik sadece nasıl kullanıldığını anlamak için öyle bir giriş yaptık. Bu word dosyasında anguların temel özelliklerini ele alarak bir movieApp yapılacak.

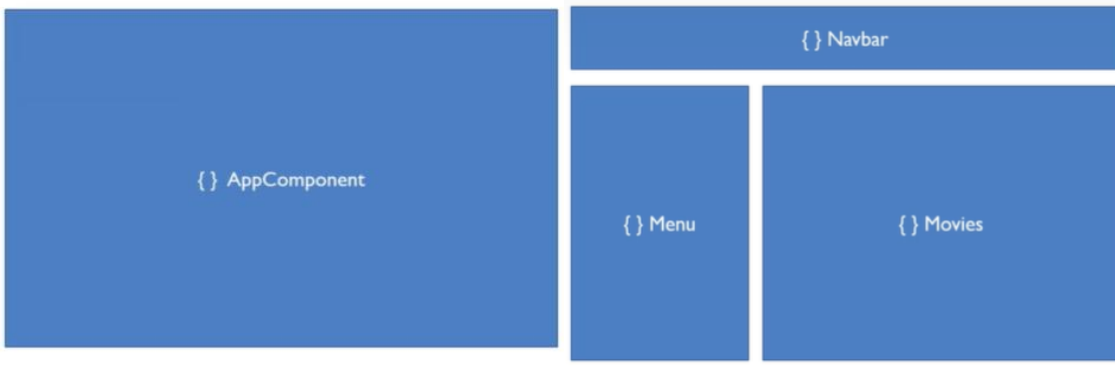
Daha sonraki kısımlar da burada ele alınan temel özellikleri derinleştirecek.

PART 1: Components

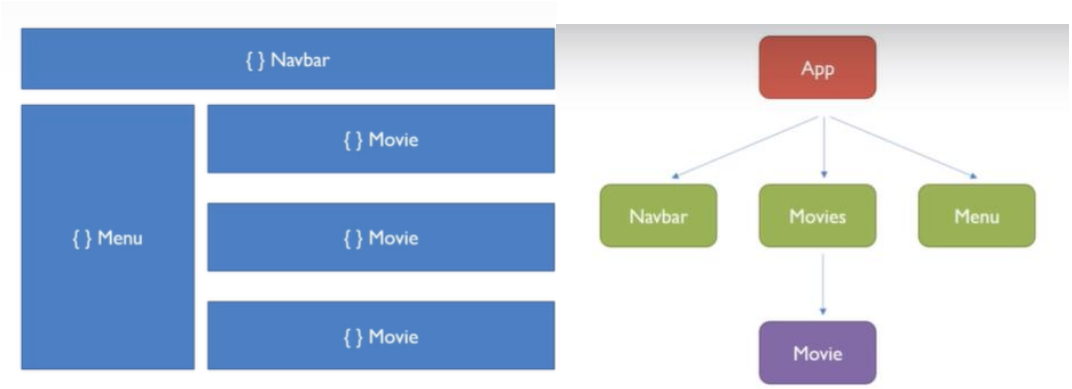
Componentler tek başına belirli bir görevi icra edebilen yapılardır, temelde bir componentin üzerinde çalıştığı bir Data yapısı vardır, bu datayı işleyip içerisinden belirli elemanları seçme işlemini yapan bir Logic yani kod vardır. Son olarak da bu bilgileri html'e aktarmak için bir Html Template kısmı vardır.



Bir uygulama yapıyoruz diyelim, uygulamanın kendisi AppComponent bizim bu AppComponent'i alt component'lere ayırmamız gerek, böylece uygulamanın farklı kısımlarını ayrı ayrı çalışırız ve işlerimiz daha rahat olur.



Hatta bu componentleri de alt componentlere ayırabiliriz mesela Movies componentini birsürü movie componenti'nin birleşimi şeklinde oluşturabilriz.



Yani component hiyerarşisi yukarıda görüldüğü gibi olur AppComponentin altına dizilirler.

Şimdi burada daha önce gördüğümüz angular'ın module, component, html yapısını tekrar edelim ve önümüzdeki başlıkta kendimiz componentler yaratalım.

Bizim uygulama çalışınca çalıştırılacak olan doyasımız main.ts, bu dosyanın içerisinde de uygulamanın tarayıcıda çalışacağı belirtilmiş ve ilk çalışacak module AppModule olarak belirtilmiş.

main.ts

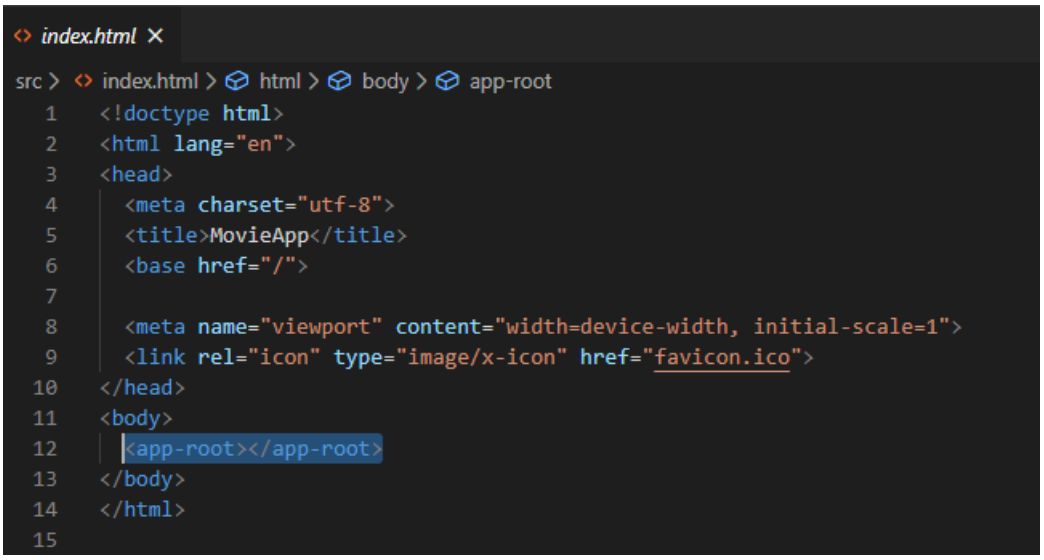
```
src > main.ts > ...
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

App.module.ts içerisinde angular'dan gelen Module'ler ile bizim yarattığımız componentler bundle edilir, burada da ilk çalıştıralacak component AppComponent olarak belirtilmiş.

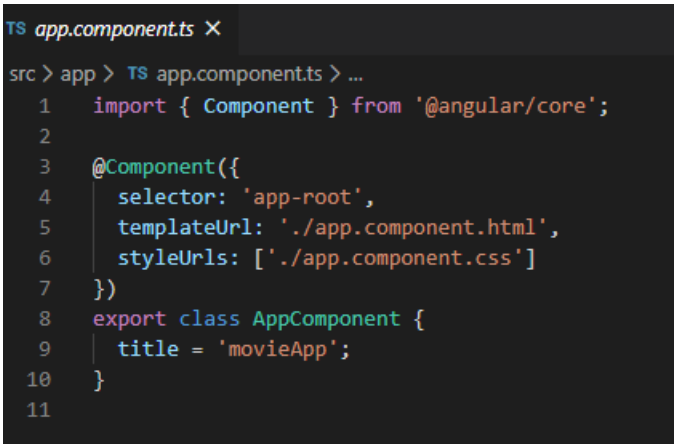
app.module.ts

```
src > app > app.module.ts > ...
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

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

```

Index.html module'ümüzün çalıştığı html sayfası, module'ün içindeki bir componente ulaşmak için de selector tag'ini kullanıyoruz. Böylece appcomponent'in html'ini buraya çağırmış oluruz.



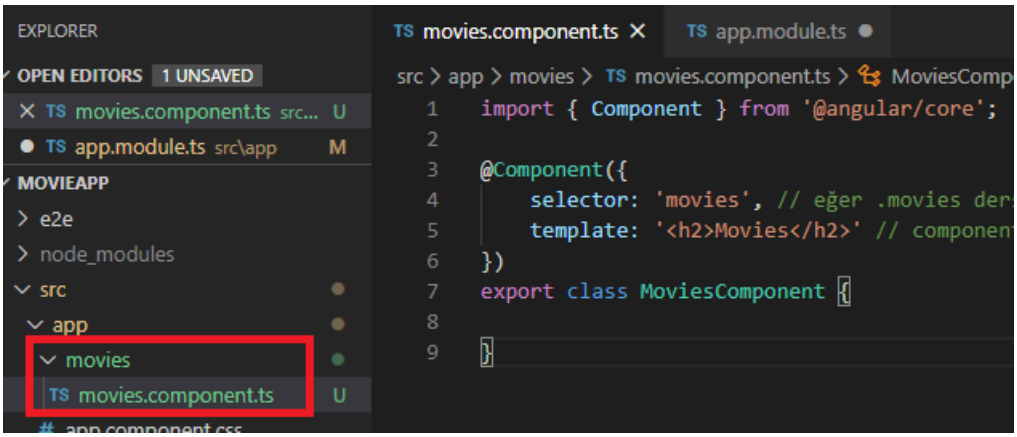
Bildiğimiz gibi, app component class'ının içerisindeki değişkenleri app component html'inde kullanabiliyoruz, bazen {{}} ile kullanıyoruz bazen ise directive'ler'in yanında "" ile kullanıyoruz.



Bunları zaten biliyoruz, bir tekrar etmiş olduk. Şimdi yeni componentleri nasıl üreteceğimize bakacağız.

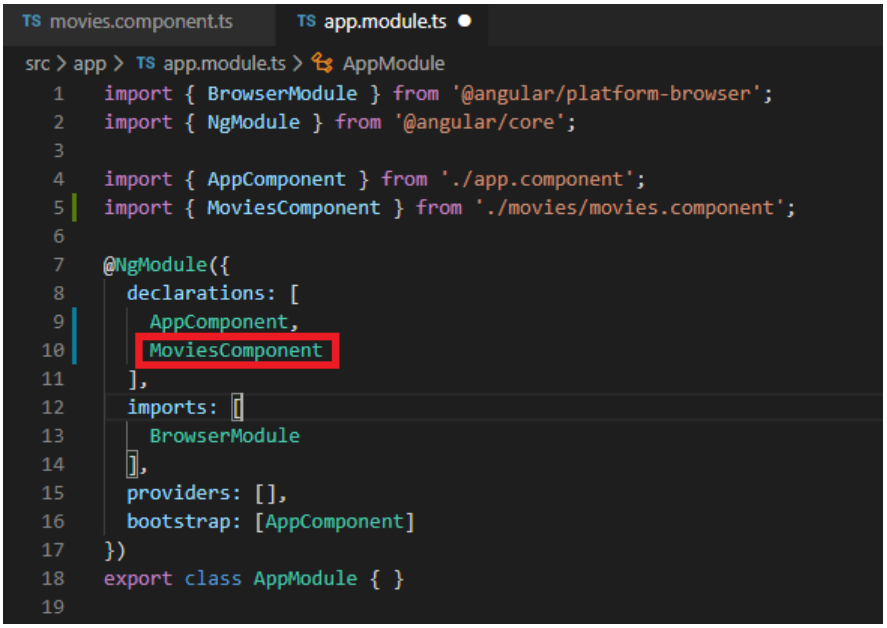
PART 2: Generating new Components

Bu kısımda yeni bir component oluşturacağız.

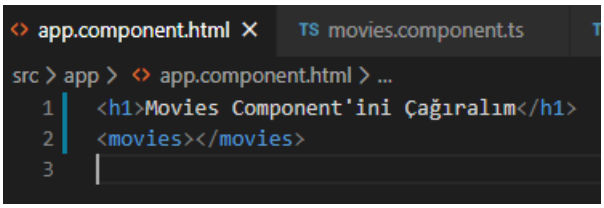


- Öncelikle app altında bir movies kalsörü oluşturduk bunun altında da bir movies.component.ts dosyası oluşturduk.
- Daha sonra yukarıdaki gibi bir MoviesComponent class'ı tanımladık henüz içi boş, ayrıca @Component decorate'i ekledik.
- Decorator'ın içinde selector'ı ve template'i belirledik, selector .movies şeklinde veya #movies şeklinde yazılırsa veya şeklinde çağırılması gerekecektir.

Sonuçta component'i tanımladık ancak, bu component'i kullanabilmek için AppModule içinde componenti declare etmeliyiz:

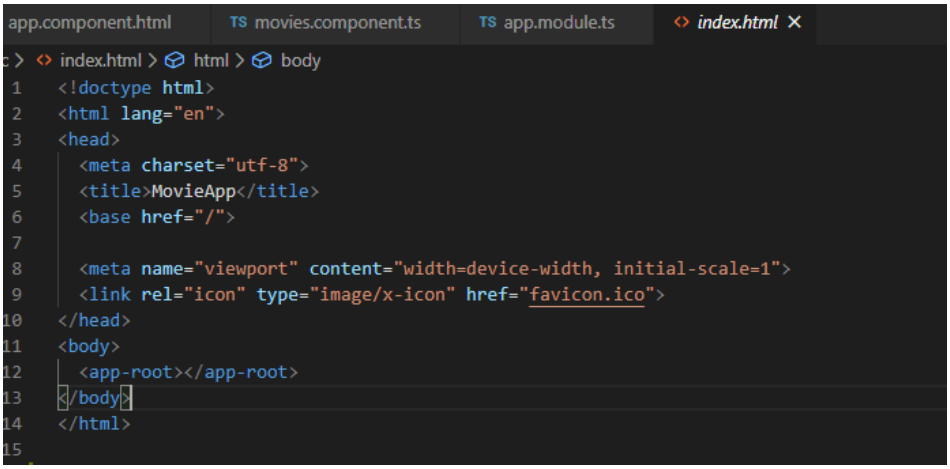


Artık app.component.html içerisinde movies component'e ulaşabiliriz. Unutma app.component.html'den app component'e de ulaşabiliyorduk.

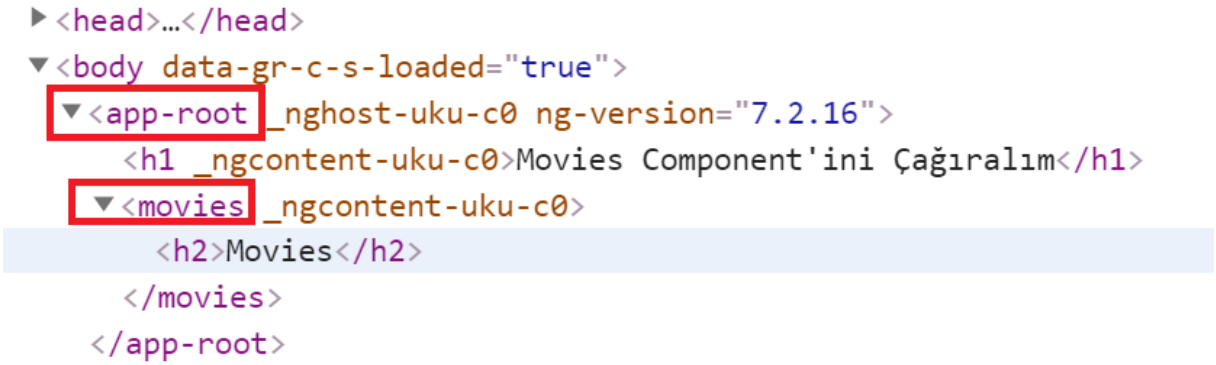


?Merak ediyorum, acaba movies.component.html oluştursak oradan da app.component'e ulaşabilir miydik?

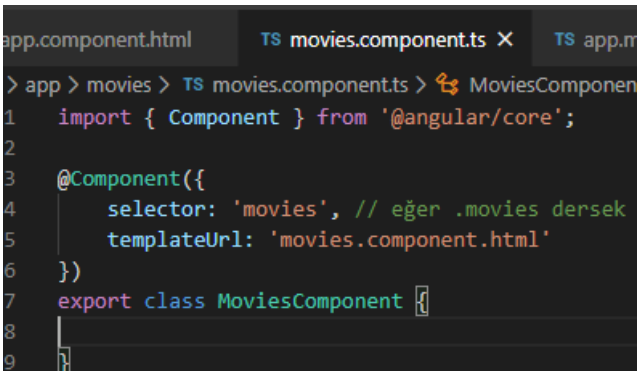
Index.html önce <app-root> etiketi ile app.component'i çağırır, app component'de kendi içinde movies component'ini çağırır.



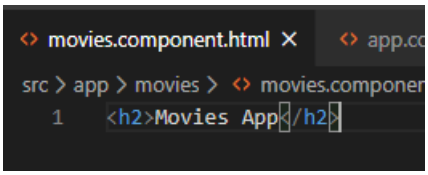
Şimdi çalışan index.html'i inspect edelim, beklediğimiz gibi önce app component'in html'i yer alıyor daha sonra onun altında çağırılan yerde diğer component'in html'i yer alıyor.



Son olarak movies.component.html sayfasını oluşturalım. Öncelikle movies.component.ts içerisinden bu html'i movies component'e link ediyoruz.



Daha sonra html sayfasının içeriğini oluşturuyoruz:



Değişen bir şey olmadı, movies.component.html'i ayırmış olduk, benzer şekilde css'i de ayırabiliriz.

PART 3: Generating new Components with CLI

Bu kısımda ilk önce extensions'dan vs code'a git history ekledik böylece, branch'leri görselleştirebiliyoruz.

Bu kısımda önceki kısımda yaptığımız elle component oluşturma işleminin kısayolunu göreceğiz, yani yeni bir klasör oluşturma içinde yeni bir component.ts dosyası oluşturma, dosyanın içini doldurma, ilgili componente html ve css sayfası bağlama ve bu comonent'i app.module'de declare etme işlemlerini tek seferde terminal kullanarak yapabiliriz. Comment Line Interface

```
\movieapp> ng g c movie
```

Generate component movie'nin kısaltması ile yani ng g c movie ile bir movie componentini oluşturabiliriz. Elbette bunu yapabilmek için önce CTRL+C ile uygulamayı durdurmamız lazım.

Şimdi öncelikle index.html içerisinde app component'i aşağıdaki gibi çağıralım.

```
<body>
  <h1>INDEX.HTML</h1>
  <app-root></app-root>
</body>
```

movie.component.html | app.component.html ×

src > app > app.component.html > ...

```
1  <h1>App Component</h1>
2  <movies></movies>
3
```

movies.component.html ● | movie.component

src > app > movies > movies.component.html > ...

```
1  <h2>Movies Component</h2>
2  <movie></movie>
3
```

movies.component.html ● | movie.component.html ×

src > app > movie > movie.component.html > ...

```
1  <p> Movie Component</p>
2
```

INDEX.HTML

App Component

Movies Component

Movie Component

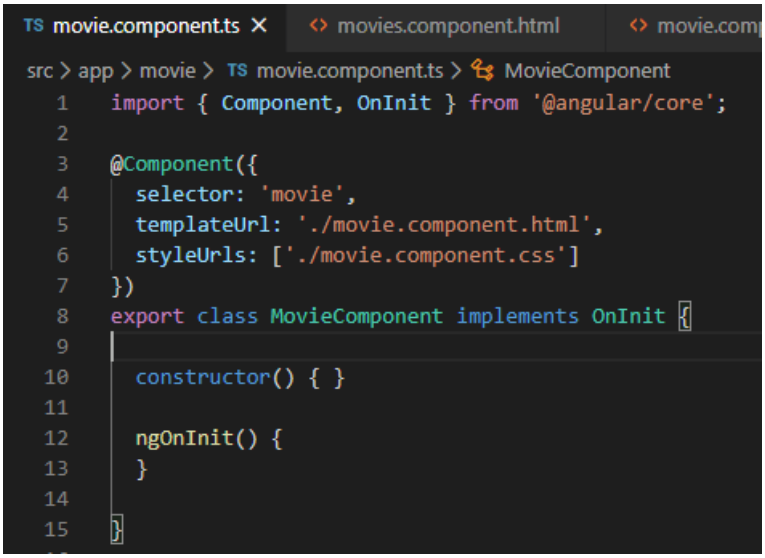
Index.html → app.component → movies.component → movie.component şeklinde bir çağırma sırası izlendi.

Anladığım kadarıyla, index.html içerisinden sadece app.component çağırılabiliyor.

Ancak, onun dışındaki componentlerden diğer tüm component'ler çağırılabiliyor, hatta meslea app.component içerisinden tekrar app.component bile çağırılabiliyor, sonsuz bir sayfa elde ediyoruz.

İstersek app.component içerisinden diğer component'leri isteğimize göre çağırırız, istersek app.component içerisinden movies'i çağırırız, movies'in içinde de movie'ler çağırırız, veya movie içinde app.component'i tekrar da çağırabiliriz vb.

Burada önemli bir diğer nokta CLI ile oluşturulan movie.components.ts'in öncekinden biraz farklı olması:



Burada bir OnInıt interface'i implement ediliyor bunun detayına sonra bakacağız.

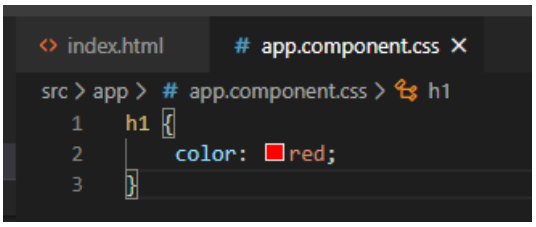
PART 4: Component'te CSS ile Çalışma & Bootstrap

Bu kısım basit olacak, CSS sayfalarını basitçe dolduracağız, bir de ilk word dosyasında bahsedilen bootstrap eklemeyi tekrarlayacağız.

Yeni olan 2 önemli şey var:

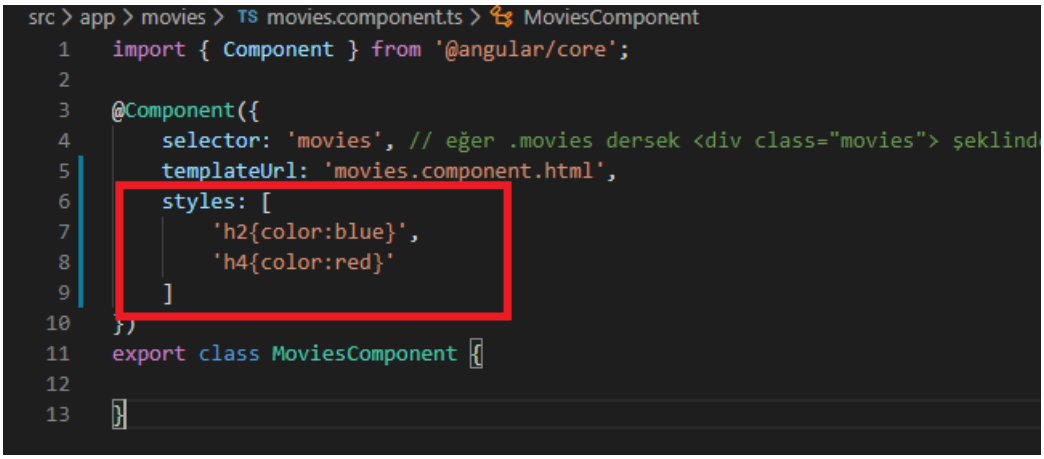
- İlki şu, bir component'in .css file'ını modify ettiğimiz zaman sadece o componenet'in .html file'ı içerisindeki etiketleri etkiler, yani örneğin app.component'in css'i sadece app.component.html'i etkiler, app.component.html içerisinden çağrılan movies.component.html'i veya diğerlerini etkilemez.
- Bir diğer husus da component'e css bilgisini external olarak component.css ile verebileceğimiz gibi, component.ts dosyasının içinde internal olarak da verebiliriz.

Örneğin:



Dersek yalnızca app.component.html sayfası içindeki h1 etiketleri kırmızı olacak, movies veya movie componentleri içindekiler etkilenmeyecek.

Aşağıda da movies.component.ts içerisinde movies component'ine internal css atanıyor. Aslında videoda [' '] arasına hem h2'yi hem de h4'ü yazdı ama bende hata verdi ayrı ayrı yazdım. Zaten bence .css uzantılı ayrı bir dosyada tanımlamak daha mantıklı.



İkinci kısımda Bootstrap eklemeyi tekrar hatırlayalım:

- Daha önceden bunu zaten yapmıştık.
- Öncelikle projenin çalışmasını CTRL+C ile durduruyoruz.
- npm install –save bootstrap
- npm install –save jquery
- bootstrap ve jquery'i `node_modules/bootstrap'e` indiriyorum ve package.json'a dahil ediyorum.
- Bir de şimdi bootstrap'i kullanacağım yerde tanımlamam gerekecek, biz tüm projede kullanmak istediğimiz için bu tanımlamayı angular.json içerisinde yapacağız:
- Angular.json içindeki architect alından styles kısmında zaten projenin tamamında kullanılan styles.css dosyası tanımlanmış bunun yanına bir de bootstrap.min.css dosyasını tanımlıyoruz bu dosya da node_modules altına indirilmişti. Ayrıca scrpipt arrayinin içini de dolduruyoruz.

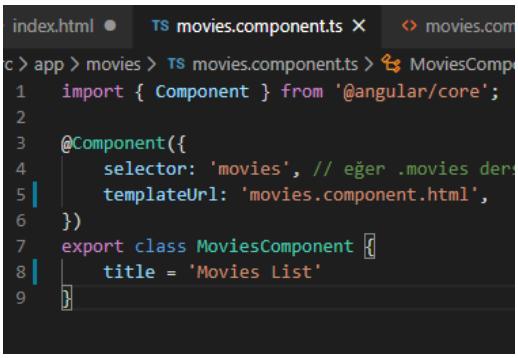
```
],
"styles": [
  "./node_modules/bootstrap/dist/css/bootstrap.css",
  "src/styles.css"
],
"scripts": [
  "./node_modules/jquery/dist/jquery.js",
  "./node_modules/bootstrap/dist/js/bootstrap.js"
]
```

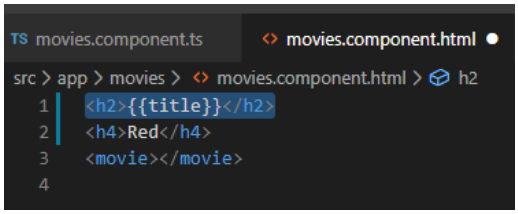
- Anladığım kadarıyla jquery'i indirmemizin sebebi de bootstrap'in jquery kullanıyor olması, tabi bu bootstrap'in js gerektiren componentleri için geçerli. Sadece css için kullanacaksan, scripts kısmına hiç gerek yok.

PART 5: Template yani COMPONENT.HTML

Bu kısımda da bir numara yok, bildiğimiz şeyler. Sadece bir component url'sinin ayrı bir html'de verilmesine alternatif olarak component.ts'de templateUrl yerine template şeklinde verilmesini göstermiş ancak bence bu useless.

Öncelikle data binding'i tekrar ediyor, movies.component.ts içinde tanımlanan bir title değişkeni movies.component.html içerisinde {{ }} içerisinde kullanılıyor:





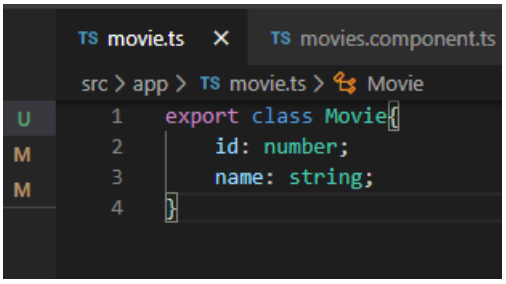
Görüldüğü gibi title değişkeni movies component'te tanımlandı ve movies.component.html içerisinde kullanıldı.

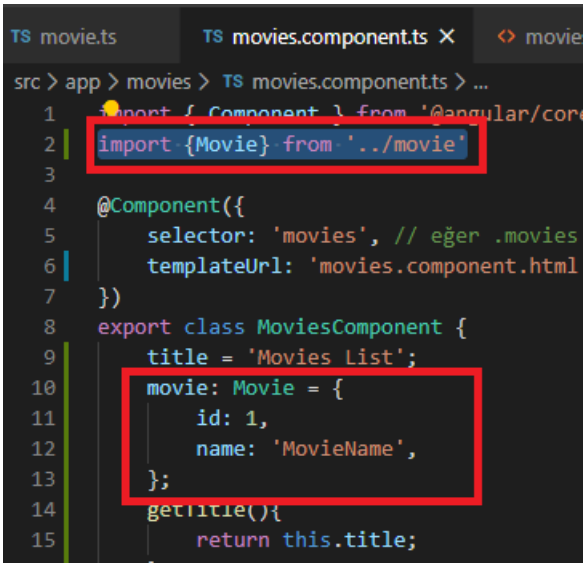
Bu movies.component.html sayfasını ayrı bir sayfada tanımlamak yerine movies.component.ts içerisinde de gösterebilirdik ancak bu bence verimsiz.

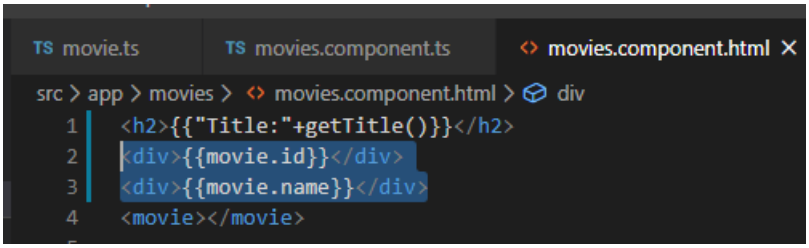
- Bunu yapmak istersek templateUrl yerine template deriz ve ``` işaretleri içerisinde html etiketlerini yazarız:

- Burada getTitle() methodunu kullanırken bir de "" içerisinde string kullanabildiğimize dikkat et.

Şimdi de yine daha önce değinilen bir şey yapalım, component içerisinde bir object oluşturalım ve bu objenin class'ı da ayrı bir ts dosyası olarak tanımlansın, yani bir model olsun ve bu modeli component'e import edip bu modelden bir obje oluşturalım:

- Öncelikle app içinde bir movie.ts oluşturuyoruz ve içine aşağıdaki gibi bir class oluşturuyoruz, farkındaysan bu bir interface gibi. Sadece obje oluştururken hangi özelliklerin olacağı söyleniyor bir constructor vesaire yok.
- 

- Ardından component.ts içinde aşağıdaki gibi bir Movie objesi yaratıyorum.
- 

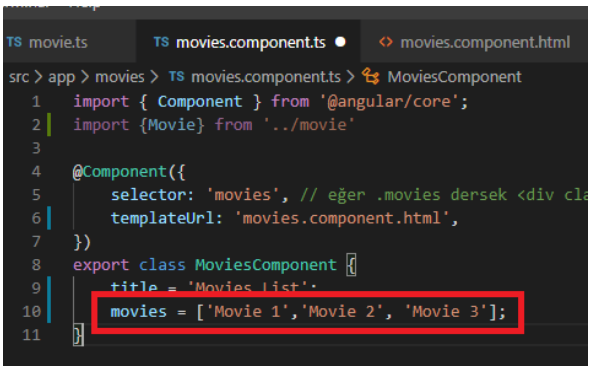
- Bu movie objesine artık component.html içerisinden ulaşılabilir.
- 

PART 6: Listeyi Template (component.html) üzerinde gösterme.

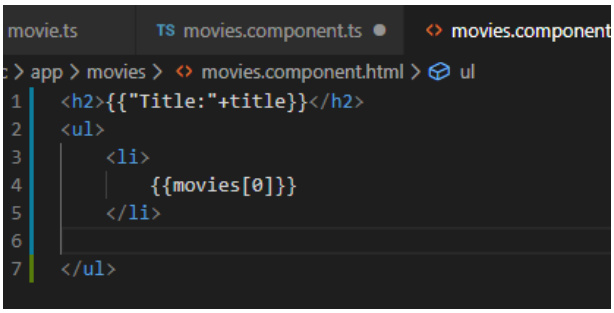
Önceki başlıkta, component.ts içinde üretilen title (string) veya movie(object) değişkenleri doğrudan {{}} ile template içerisinde kullanılabildi.

Peki ya component.ts'de üretilen bir listenin elemanlarına template'den ulaşmak istersek bunu nasıl yaparız?

Bunun yolu ngFor kullanmak. Mesela aşağıdaki gibi basit bir movies listesi olsun.



Bu listeye ulaşmak için ilk ihtimal her elemanı tek tek <li> içinde yazmak, ancak listenin eleman sayısı değişkense, veya sayısını bilmiyorsak, veya tek tek uğraşmak istemiyorsak bu yöntem useless!



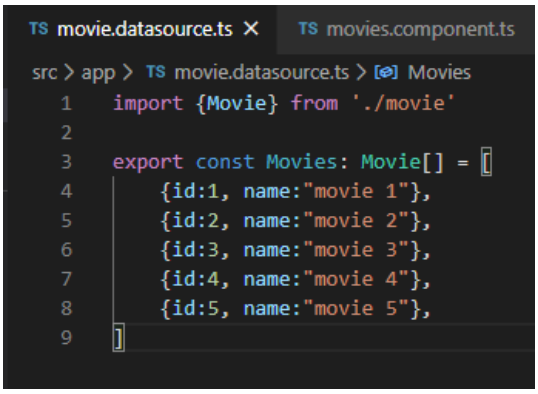
İşte *ngFor directive'i burada devreye giriyor

Burada yapılan şey şu, movies.component.html'e model olarak gönderilen movies listesi üzerinden her elamanı al ve her bir eleman için bu <li> etiketini template'e ekle, template içinde de movie'yi yazdır.

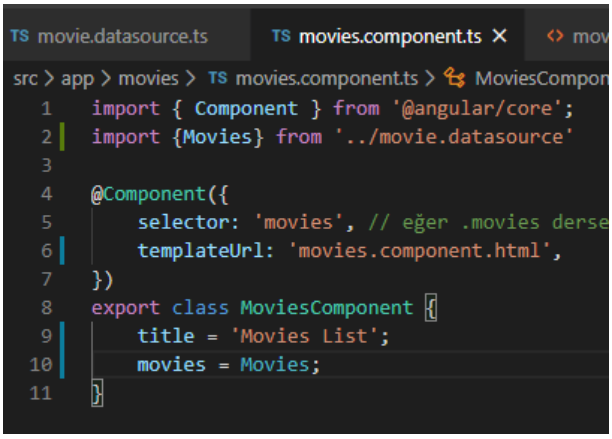
ngFor'un mantığı bu kadar.

Şimdi kalan kısımda örneği biraz daha geliştirelim, basit bir liste göndermek yerine bir object list'i template'e gönderelim. Yani Movie objelerinden oluşan bir movie list'i gönderelim.

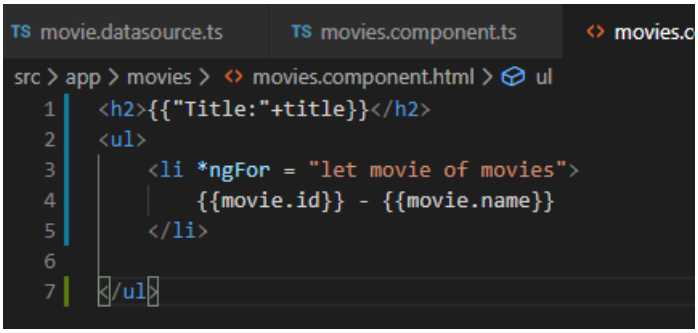
Öncelikle database'i temsil etmesi için bu movie.datasource.ts isimli bir dosya yaratıp içerisinde 5 elemanlı const bir Movies object listesi yaratıyoruz:



Daha sonra bu Movies listesini movies.component.ts içerisinde aşağıdaki gibi import ediyoruz ve movies isimli değişkenin içerisine atıyoruz.

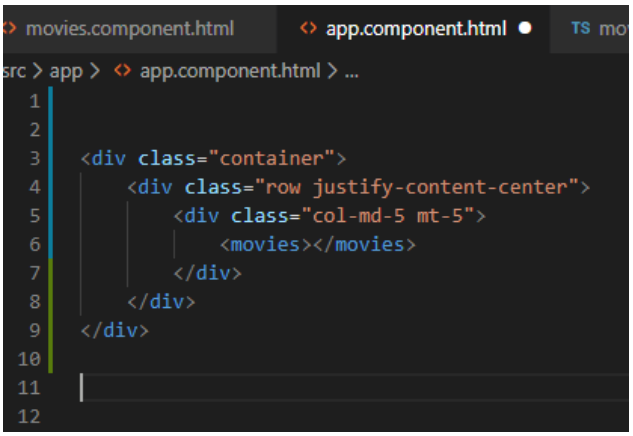


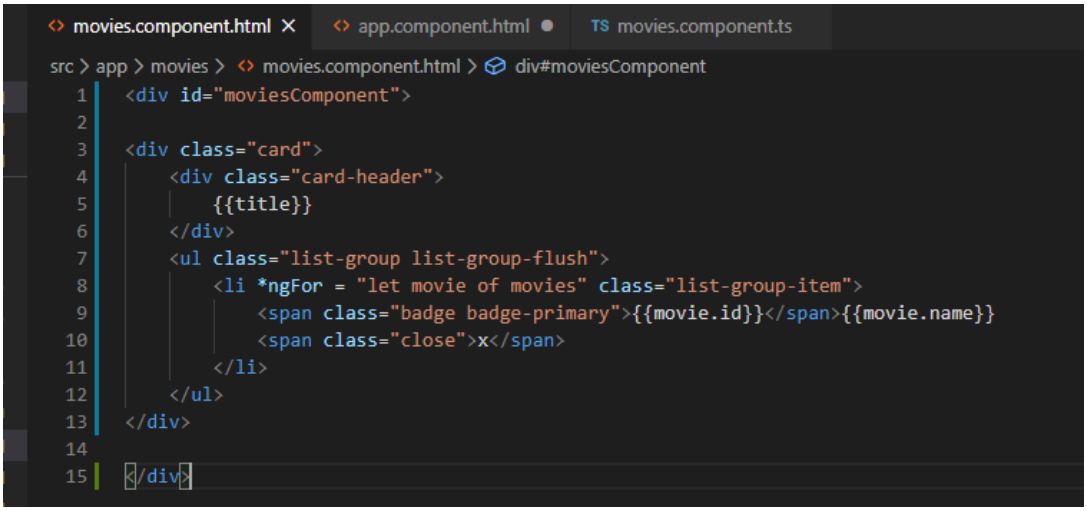
Sonuçta movies isminde bir Movie listesini model olarak oluşturmuş oluyoruz, bu modele template içerisinden ulaşabiliriz:



Listenin her bir movie objesi için id ve name li elemanı içerisinde yazdırıldı.

Şimdi uygulamayı biraz güzelleştirmek adına, bazı bootstrap class'larından yararlanıyoruz, app.component.html'i ve movies.component.html'i aşağıdaki gibi düzenliyoruz.



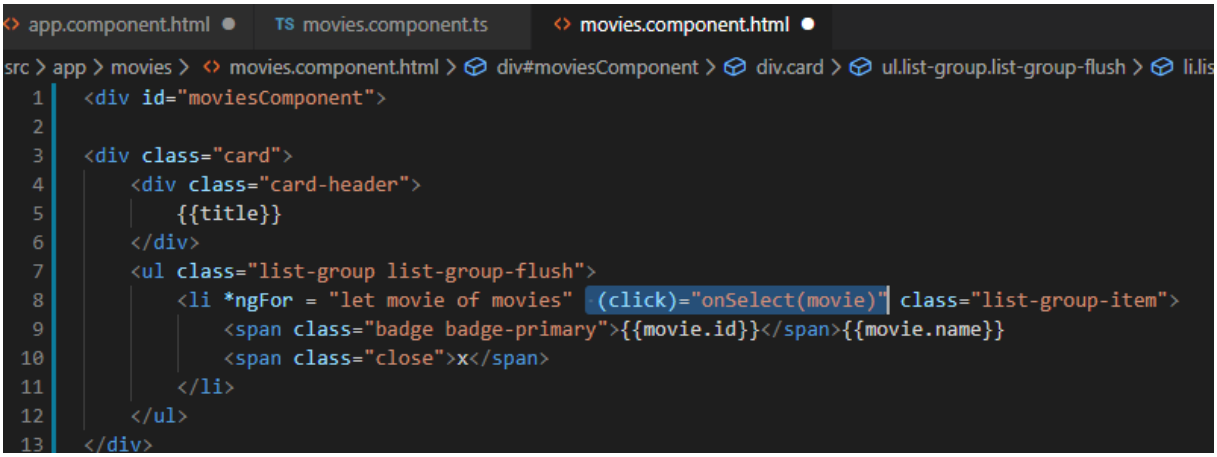


Sonuçta listeyi aşağıdaki gibi görselleştirebiliyoruz.

PART 7: Event ile Çalışma

Daha önce component.ts içerisinde tanımlanan değişkenlere component.html içinden ulaşabildiğimizi görmüştük ister {{}} ile ister ise ng directives ile bunu yapabiliyorduk.

Şimdi ise component.html içerisinden event ile bilgiyi component.ts'e göndereceğiz. Bunun için yapacağımız şey şu olacak, ngFor ile oluşturulan movie listesindeki herhangi bir liste elemanına tıklandığında, ilgili movie elemanı component.ts içerisinden çağırılan bir onSelect() methodu ile component.ts'e gönderilecek:

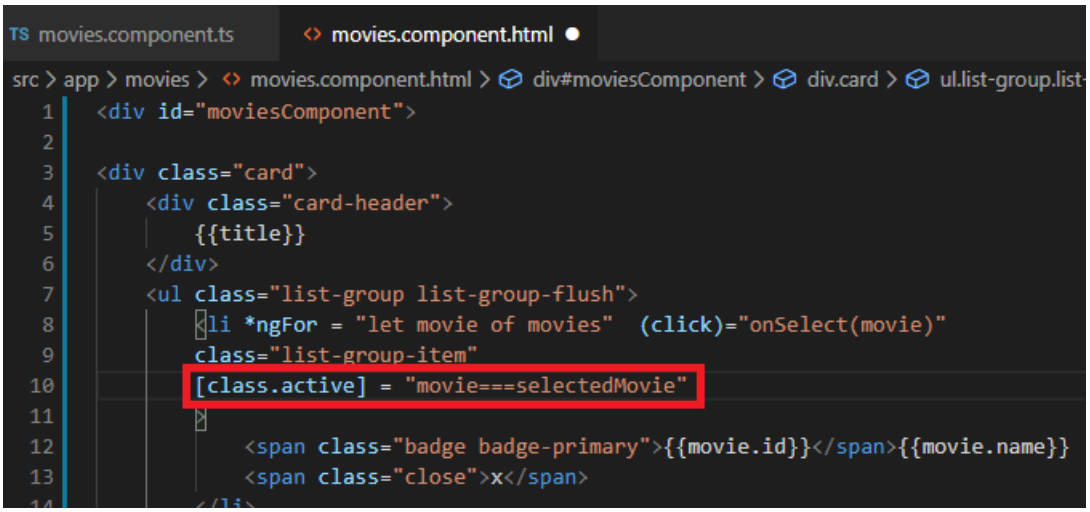


movies.component.ts'in içinde de aşağıdaki gibi onSelect() methodu ve selectedMovie attribute'u tanımlı.

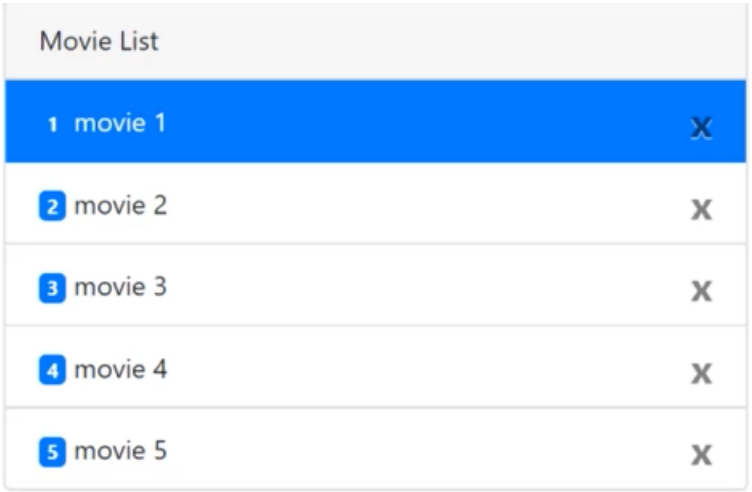
Burada şuna dikkat et, onSelect içinde geçen movie objesi selectedMovie'ye eşitlendiğinde ikisi artık aynı objeyi gösteriyor yani burada call by reference olmuş oluyor. İki ayrı obje oluşmuyor hem selectedMovie hem de parametre aynı objeyi işaret eden referanslar diye düşün!

Burada method içerisinde this.selectedMovie.name = "newName" dersek tıklanan elemanın adı değişecektir.

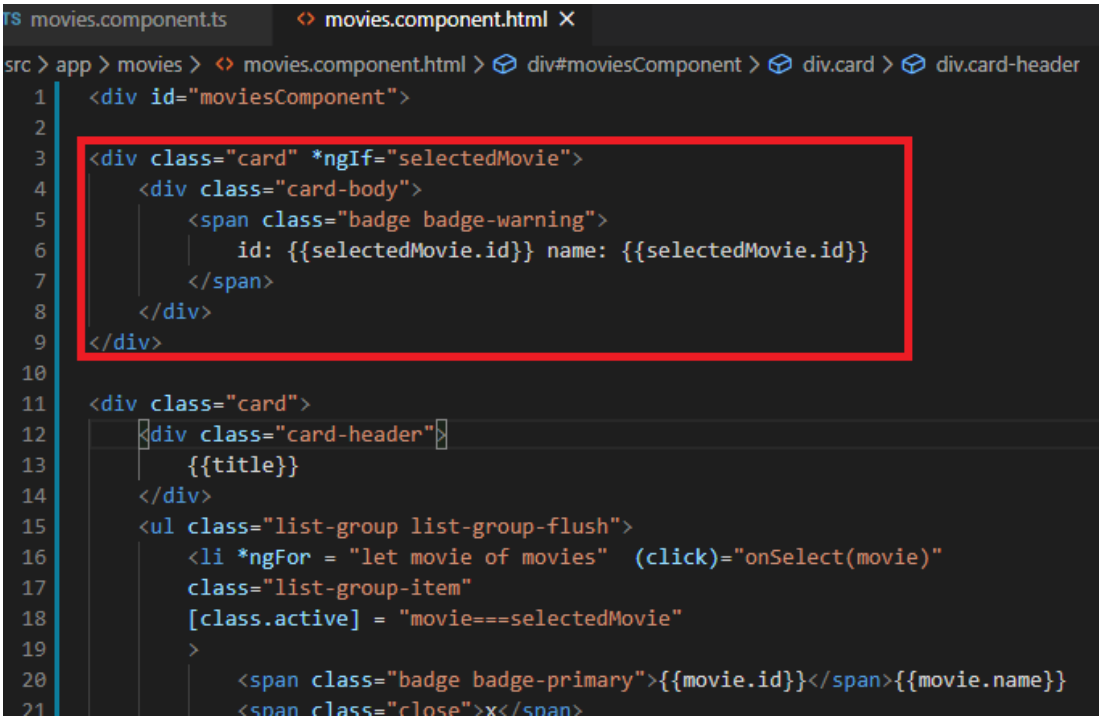
Şimdi bir başka önemli şey yapacağız, template içerisinde bir ifadenin sağlanıp sağlanmamasına göre elemana bir class ekleyeceğiz:

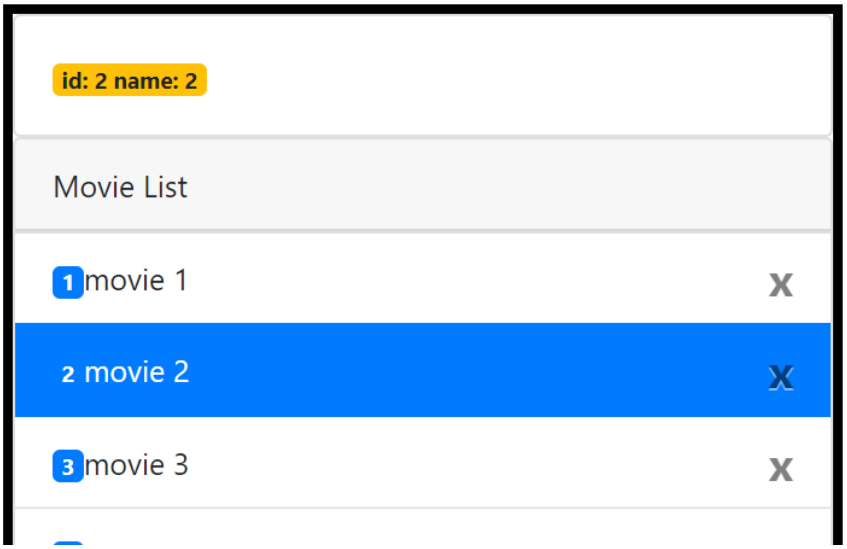


- Burada ilgili li elemanı zaten bir list-group-item class'ına sahip, buna bir active class'ı vermek istersek list-group-item active diyebiliriz ancak böyle dersek tüm elemanlar active olacak.
- Biz istiyoruz ki bu elemana active class'ını bir şartla verelim, o şartı da tırnak içerisinde yazıyoruz.
- Bu şart ilgili li elemanına karşılık gelen movie'nin selectedMovie ile aynı olması. En son tıklanan movie selected movie oluyordu, bunu click event'i ile sağlamıştık.
- O halde en son tıklanan movie'nin li elemanına active class'ı veriyoruz, böylece tıklanan li elemanının rengi değişiyor.



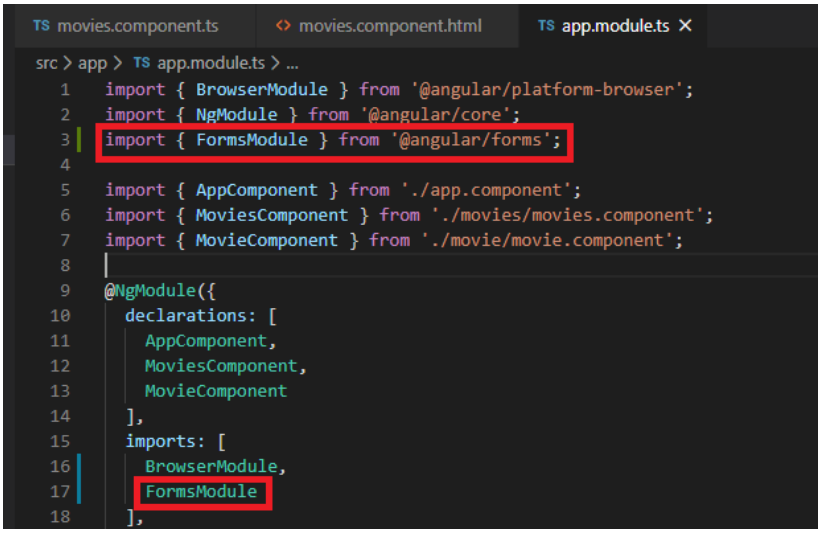
Şimdi de, movies componentinin üstüne bir kısım daha ekleyelim bu movie detay kısmı olsun burada tıklanan elemanın ismini ve id'sini gösterelim.
Tıklanan eleman zaten component.ts içerisine selectedMovie olarak alındığı için burada bu elemana tekrar {{}} şeklinde ulaşabiliriz. Bunun dışında, sayfa yeni yüklendiğinde kırmızıyla işaretlenen html kodlarını yani detay card kısmını görmek istemediğim için buraya bir ngIf directive'i ekliyorum ki eğer selectMovie yoksa buradan false dönsün ve eleman gösterilmesin.



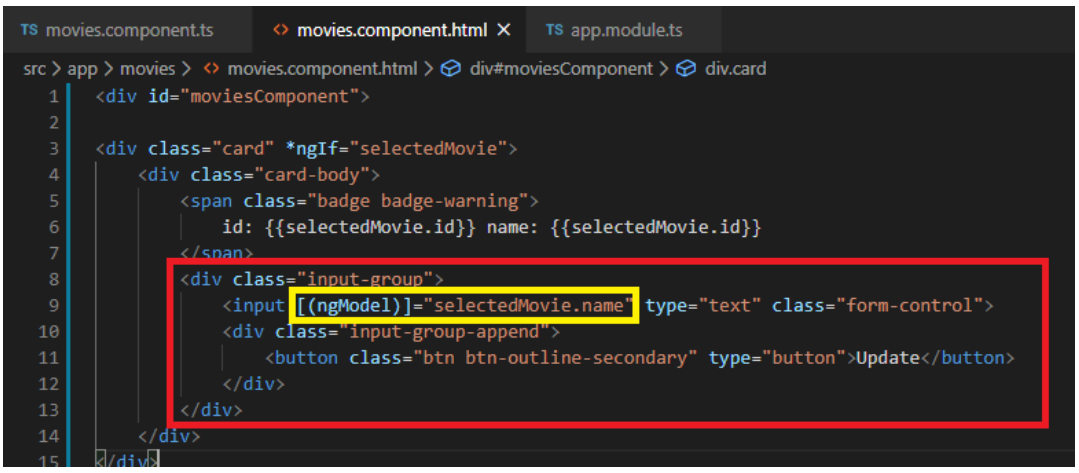


Şimdi de bu detay kısmına bir input bloğu ve update butonu ekleyelim, input bloğu ile selectedMovie'yi çift yönlü bind edelim bunun için ngModel kullanmamız gerektiğini biliyoruz.

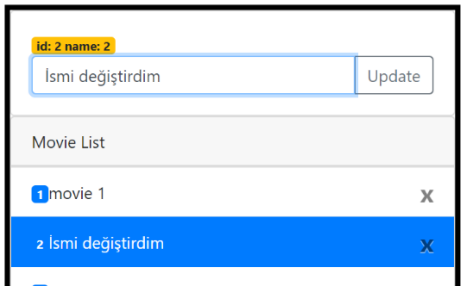
O yüzden öncelikle ngModel'i kullanabilmek için gerekli modülü app.module içerisinde import edelim:



Şimdi de movies.component.html içerisinde detay kısmına bir input-group ekleyebiliriz:



Böylece selectedMovie.name ile input'u çift yönlü bind ettik biri değişince diğeri de değişecek.



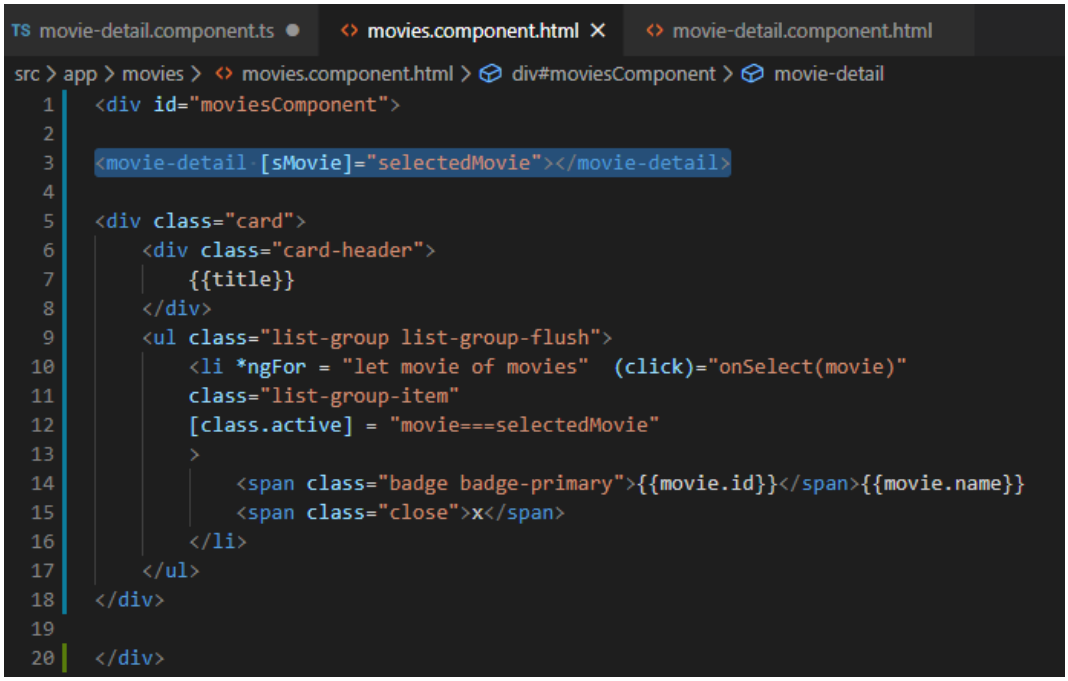
PART 8: Component'e Parametre Gönderme

Önceki kısımda movies component'in üst kısmına bir detail card kısmı eklemiştik burada selectedMovie'nin detaylarını gösterdik ve bir input elemanıyla da double sided binding yaptık.

Bu kısımda amacımız bu detail kısmı için ayrı bir component oluşturmak olacak. Component oluşturmayı zaten biliyoruz ama problem şu olacak, movies component içerisinde selectedMovie objesini kullanıyordu bu obje de movies.component.ts'den elde ediliyor ancak biz yeni bir detay component'i oluştururursak bu component movies.component.ts'e ulaşamayacak detay.component.ts'e ulaşabilecek.

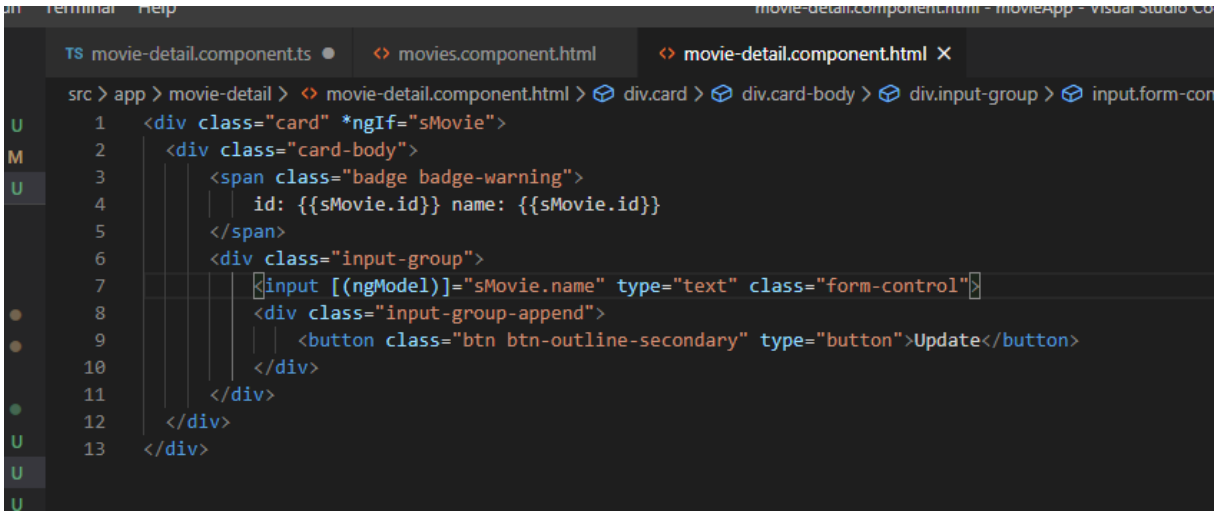
O halde bizim selectedMovie parametresini bir şekilde yeni component'e geçirmemiz gerekecek işte bu bölümde bunu anlayacağız.

- 1. Öncelikle ng g c movie-detail diyerek yeni detail component'ini yaratıyoruz.
- 2. Daha sonra, movies.component.html içerisindeki detay card kısmını kesiyoruz ve, movie-detail.component.html içerisine yapıştırıyoruz.
- 3. Bunu yaptıktan sonra movies.component.html içerisinde kesilen yere de <movie-detail> tag'ini ekliyoruz ki, aynı detail html sayfası burada gösterilmeye devam etsin.



- İyi ama bu yeni component selectedMovie'ye ihtiyaç duyuyor, bu ihtiyacını karşılamak için sMovie isminde yeni bir değişken ile selectedMovie objesi yeni componente yukarıdaki gibi aktarılıyor.
- Ancak unutma ki bu aktarma ile artık detail component.html movies.component.ts'teki değişkene ulaşıyor demek değil. İlgili değişkenin aşağıdaki gibi detail.component.ts içerisinde tanımlanması gerekiyor:

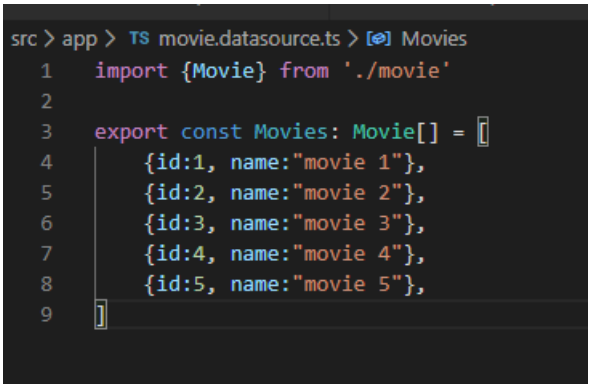
- Artık ilgili selectedMovie yeni component'e sMovie ismiyle aktarılmış oldu. Bu yeni sMovie objesi detail componentinde paşa paşa kullanılabilir:



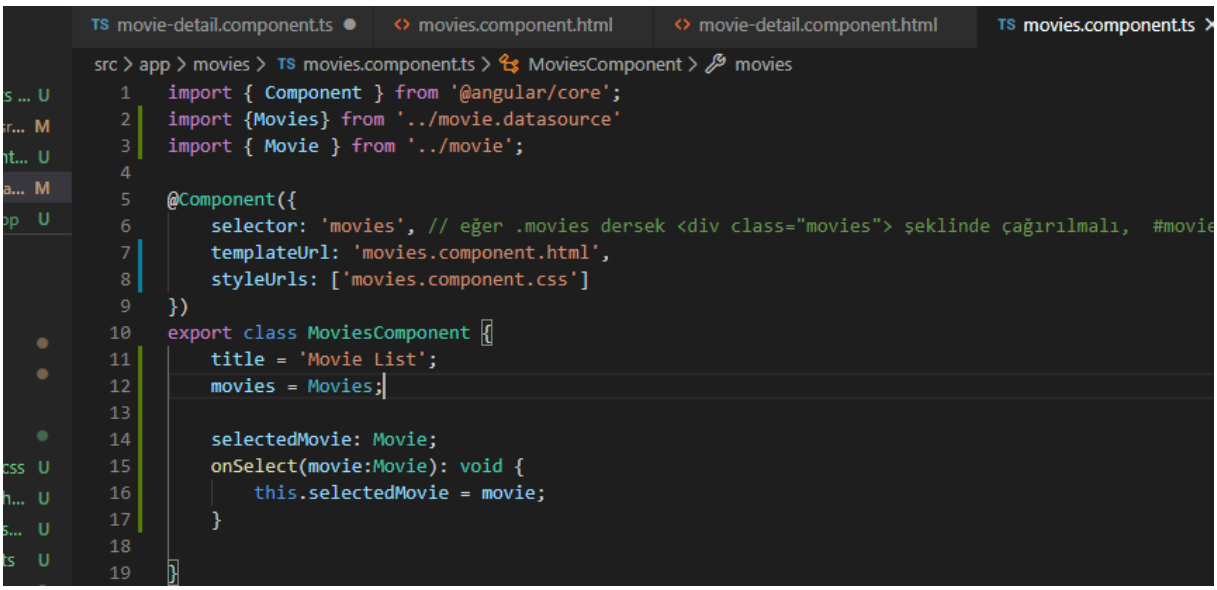
Sonuç olarak, **selectedMovie** objesi **sMovie** ismiyle **movies template** üzerinden **movie-detail.component**'e aktarıldı, bu obje **movie-detail.component.ts** içerisinde yakalandı ve **movie-detail template** içinde kullanıma hazır hale geldi.

PART 9: Service ile Çalışma

Bizim uygulamada movie database'i olarak aşağıdaki gibi bir movie.datasource.ts kullanmıştık. Bu normalde bir external database olur. API'ler aracılığı ile falan bu dataya ulaşılır.



Fakar biz bu class'ı database gibi kabul ettik ve movies.component.ts içerisinde bu class'ı import ettik yani aslında movies.component.ts içerisinde database işlemi yapmış olduk.



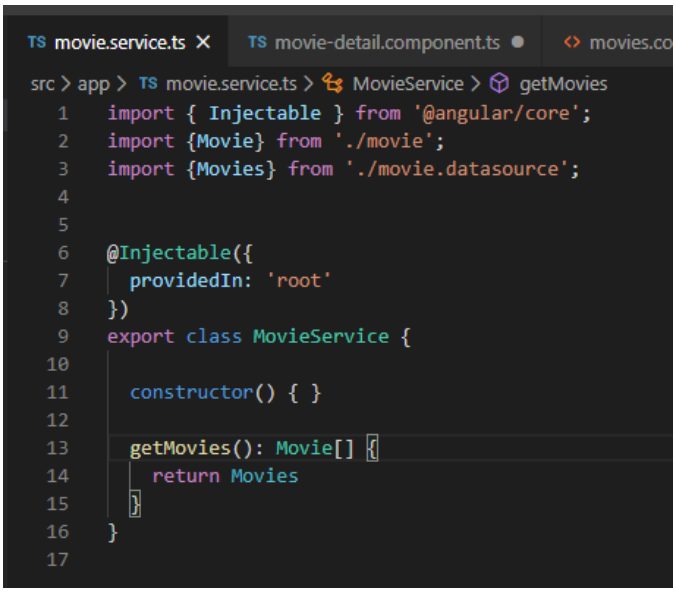
Bu doğru bir practice değil, yapılması gereken database işlemi için bir service tanımlamak ve bu service ile elde edilen data'ya tüm componentlerden erişim sağlamak

Şimdi database'den veri alma işini halletmesi için bir service oluşturacağız ve tüm component'ler bu service'in çektiği data'ya ulaşabilecek.

Bir service tanımlamak için **ng g s movie** dersek movie isminde bir service tanımlamış oluruz. Böylece bir movie.service.ts dosyası aşağıdaki gibi yaratılmış oldu.

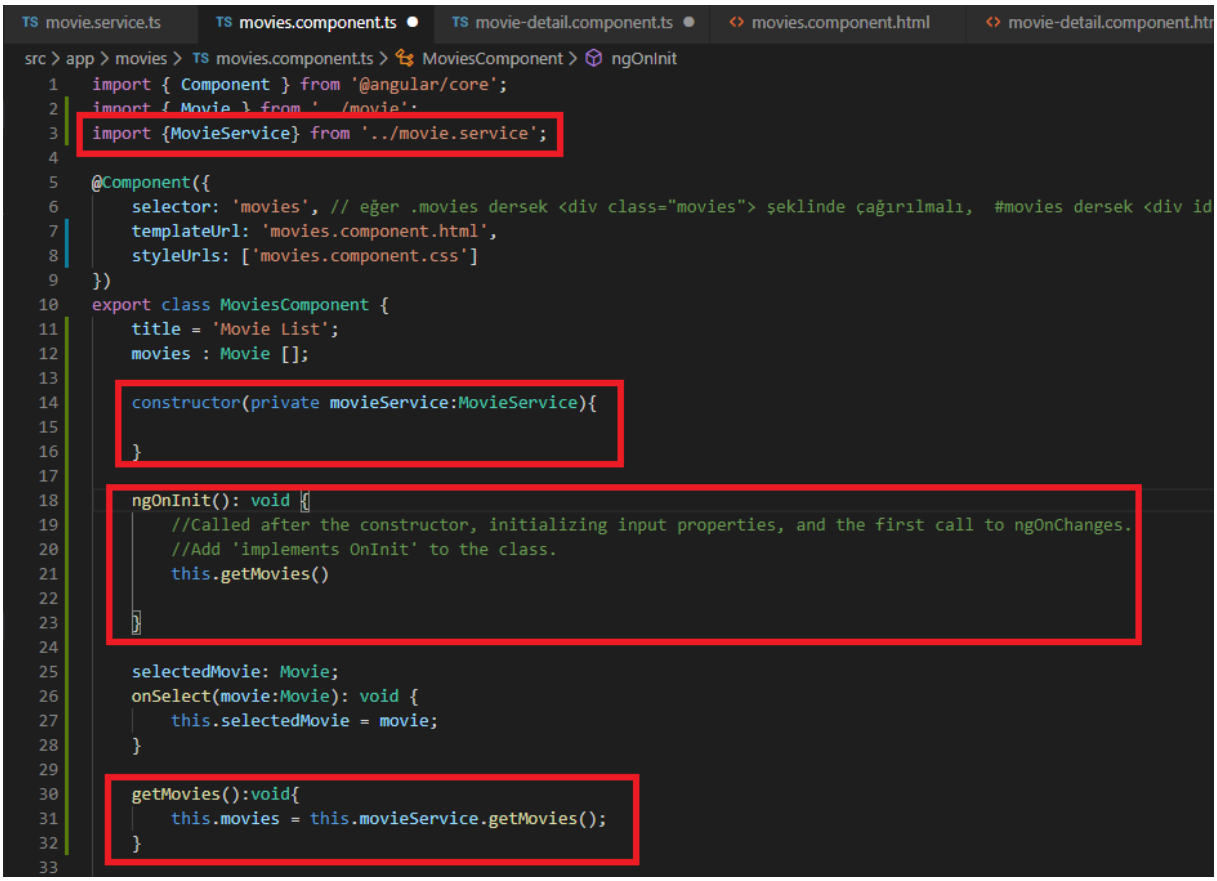
- providedIn: root diyerek uygulamanın root module'ü yani app module'ü üzerinden bu service'e ulaşılabileceğini söyler, başka module'ler olsaydı onları da belirtebilirdik.
- Injectable decorator'ının anlamı da sanıyorum dışarıda bu MovieService class'ını inject etmeye yarıyor sanıyorum.

Şimdi bu movie.service.ts'i biraz modifiye edip datasource içerisinden movies listesini burada elde edelim.



Burada datasource üzerinden movie listesi getMovies methodu ile alındı, tabi normalde burada API aracılığı ile veriler alınırdı. Sonuçta getMovies methodu service'in ulaşılabilir olduğu heryerden ulaşılabilir oldu.

Şimdi movies.component.ts içerisinde bu MovieService'i import ederlim ve getMovies methodu ile movies listesini elde edelim. Movies listesini movies değişkeninin içine atacağız.



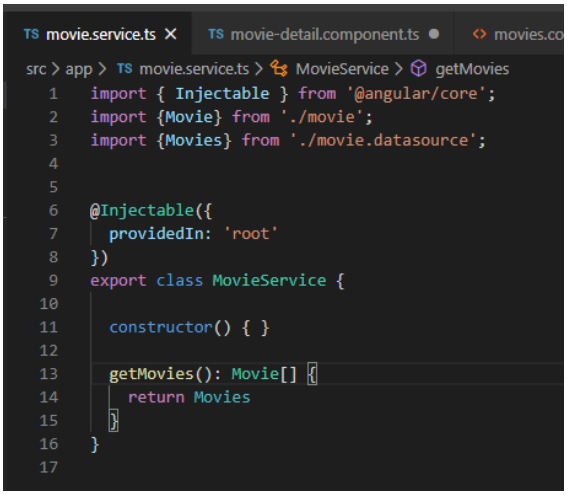
- Öncelikle MovieService'i import ediyoruz, ardından constructor içerisinde bu movieService ismiyle bir MovieService objesi yaratıyoruz ancak başına private diyoruz böylece sanırım inject etmiş oluyoruz.
- Ardından ngOnInit() methodu çalışacak bu methodun içinde de getMovies çağırılıyor ki bunun yaptığı da inject edilen movieService'in içinden movie listesini alıp objenin movies'ine atamak.

PART 9: Observable ile Çalışma

Anladığım kadarıyla database işlemleri sırasında http ile nodejs ile vesaire uğraşacağız o zaman asenkron methodlarla çalışacağız, bilgi bize asenkron şekilde gelecek bu yüzden de bir observable türünde liste geriye göndermeliyiz.

İleride service üzerinden http requesti ile talep edilen bilgi bize asenkron şekilde gelecek.

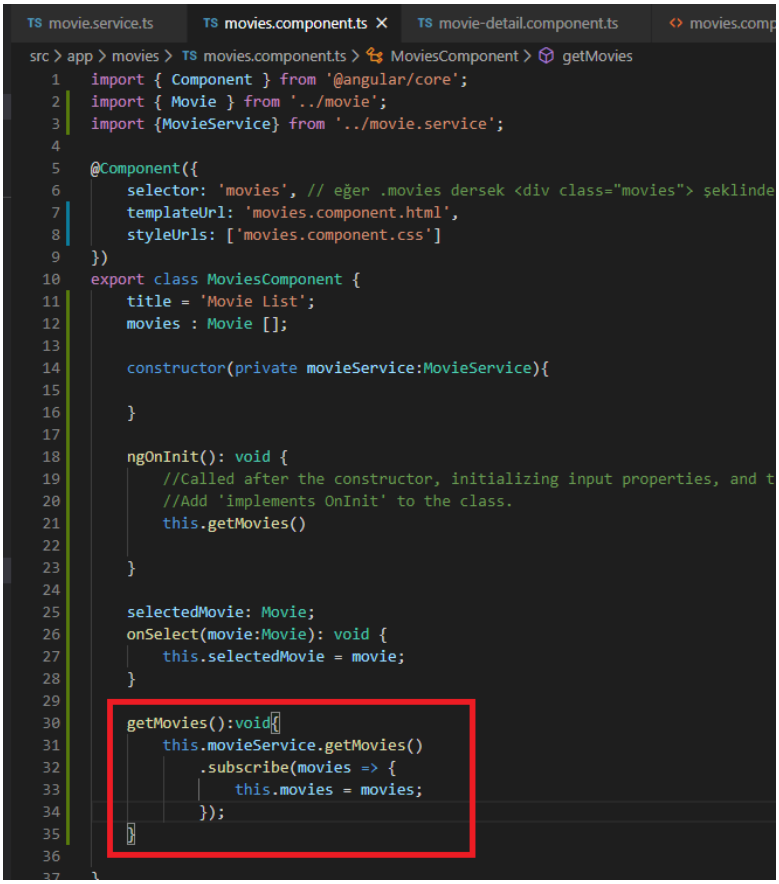
Movie.service.ts'in önceki halinde getMovies methodu ile veritabanından bilgiyi anında alıyorduk:



Örneğin component içerisinden getMovies çağırıldığında senkron olarak anında Movies return ediliyor ancak gerçek hayatta veritabanı ile uğraşırken durum bu olmaz, veritabanına anında bağlanamaz, burada bir delay olmak zorunda ve bu delay beklenmeli. Bu listeyi observable olarak almak sanırım bu işe yarıyor, liste tamamen elde edilene kadar program orada kalıyor, diğer satırlara geçmiyor.

getMovies'de yukarıdaki şekilde observable olarak Movies return edilince veritabanı delaylerinin programda bir sıkıntıya yol açmasını engellemiş oluyoruz.

Ayrıca artık service'in çağıralacağı component içerisindeki getMovies methodu da biraz değişmeli yeni method aşağıdaki gibi olacak doğrudan this.movies = this.movieService.getMovies() demek yerine aşağıdaki gibi subscribe methodunu kullanıyoruz.

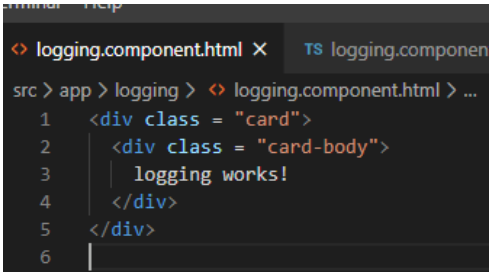


Sonuç olarak http request'leri ile veritabanı işlemleri yaparken bu kullanımlar problem yaşamamızı engelliyor.

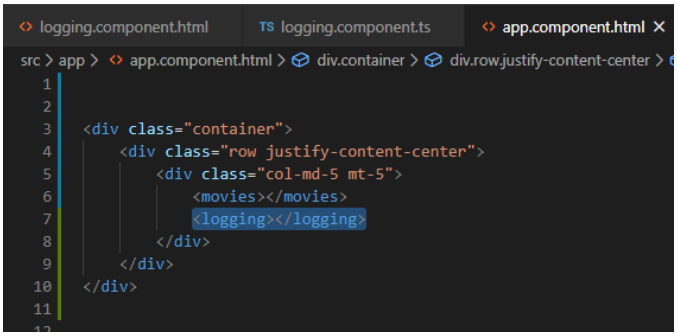
PART 10: Logging Component'inin Eklenmesi

ng g c logging diyerek logging isminde bir component'i uygulamaya ekleyelim.

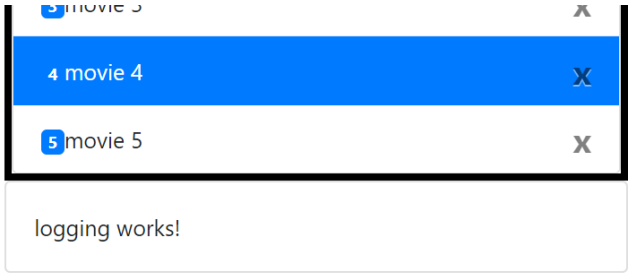
Daha sonra logging.component.html'i aşağıdaki gibi doldurduk.



Son olarak da bu elemanı app.component.html içerisinden movies'in hemen altında selector ile çağıralım:

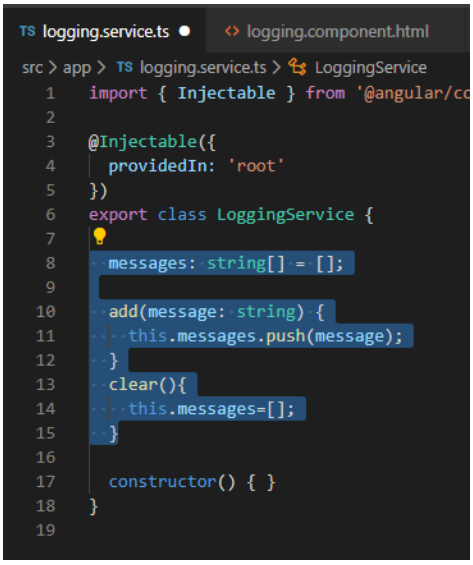


Sonuçta aşağıdaki gibi bir kısım uygulamanın app.component'ine eklenmiş oldu.



Şimdi ise ng g s logging ile uygulamaya bir logging service'i ekliyoruz.

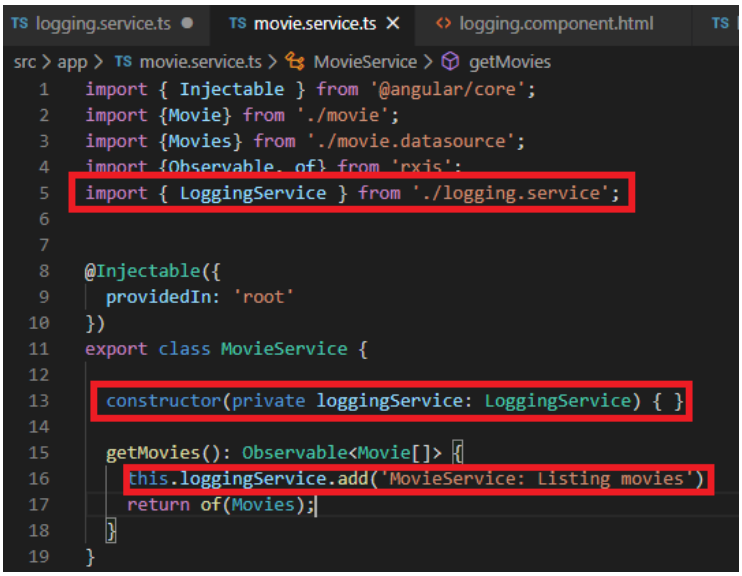
Daha sonra logging service'in içini aşağıdaki gibi dolduruyoruz:



Daha önce bahsettiğimiz gibi, service'in olayı database işlemlerini yapıyor olmasıydı daha sonra component içerisinden bu service'e ulaşıyorduk.

Eğer database işlemleri için http request'leri ile falan uğraşıyor olsaydık bir önceki kısımdaki gibi observable'larla falan uğraşırdık ancak burada uygulama belleğinde tutulan bir liste veritabanı olarak kullanılacağı için gerek kalmayacak.

Şimdi bu LoggingService class'ını gerektiği yerde bir loglama işlemi yapmak için instance edeceğiz. Örneğin daha önce yaratılan MovieService içerisinde filmler database'den çekilince, LoggingService aracılığı ile bir loglama işlemi yapılsın:

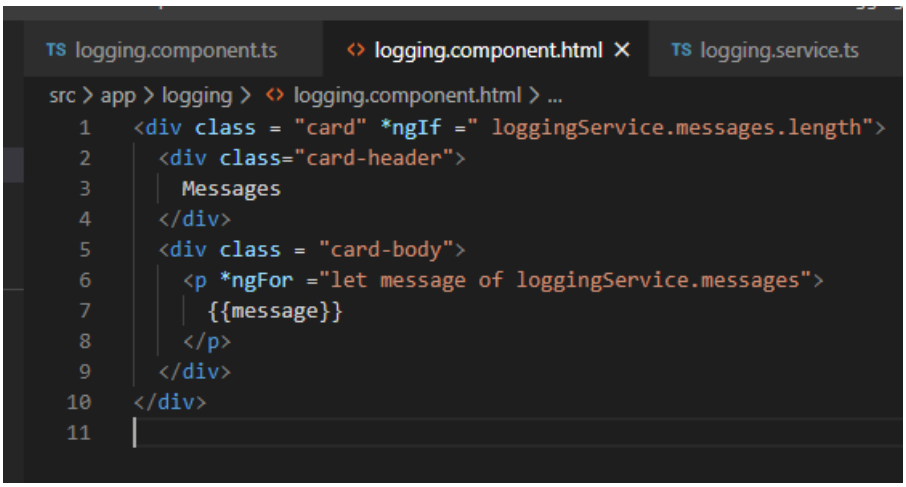


Göründüğü gibi öncelikle MovieService objesi üretildiği anda bir de loggingService property'si üretilsin dedik, daha sonra ne zaman ki getMovies methodu ile filmler alındı, o zaman loggingService aracılığı ile loglama yapıldı.

Yukarıda yapılan şey, MovieService ile database'den filmler çekildiğinde, loggingService.add'i kullanarak bir mesajı eklemekti. Şimdi bu mesajı logging.component.html içerisinden görüntüleyelim:

Öncelikle logging.component.ts içerisinden bir loggingService tanımlıyorum, ki html içerisinden bu loggingService objesine ulaşabileyim.

Daha sonra logging.component.html içerisinde bu loggingService objesine ulaşabiliyorum ve aşağıdaki gibi eğer içeriğinde mesaj varsa buna ulaşıyorum.



???ANLAMADIĞIM YER ŞU Kİ MOVIE.SERVICE.TS VE LOGGING.COMPONENT.TS ICERISINDE URETILEN loggingService'ler nasıl oluyor da aynı içeriğe sahip oluyor bunların iki ayrı obje olması gerekmiyor mu? Acaba injectable olduğu için mi böyle şeyler oluyor?

SANIRIM SERVICE OLUNCA EKLEDİĞIMIZ DECORATOR'LAR YUZUNDEN, MODULE'UN HERYERİNDE BİREBİR AYNI OBJECT'E ULAŞMIŞ OLUYORUZ!

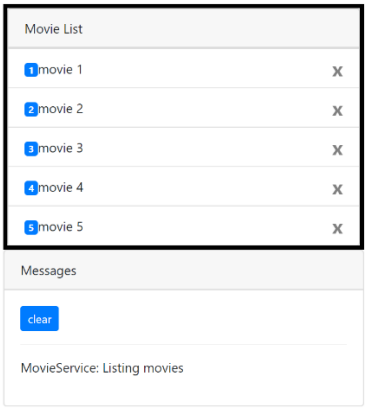
Son olarak logging.component.html üzerine eklenen yeni bir buton üzerinden loggingService objesinin clear() methodunu çağıralım ve böylece loglar temizlensin:

Sonuçta benim buradan anladığım şey şu: Service class'ını normal şekilde kullanıyoruz ancak, bu class'tan üretilen tüm objeler aslında aynı objeyi gösteriyor bu şekilde root module'ün heryerinde aynı service'e ulaşılmış oluyor!

PART 11: Routing

Bu kısımda uygulamaya routing ekleyeceğiz.

Uygulama localhost4200 altında çalıyor ve şuan anasayfa'ya ulaşında aşağıdaki görüntü karşımıza çıkıyor çünkü, index html app.component.html'i çağırıyor ve app.component.html içinde de movies ve logging componentlerinin template'leri gösteriliyor.

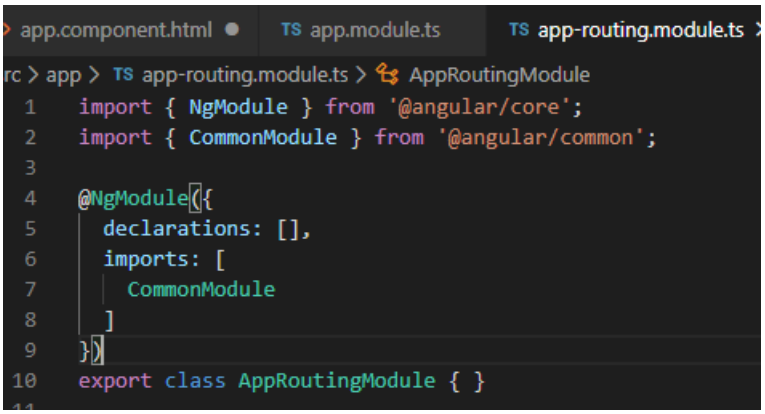


Benim şuan istediğim ise localhost4200/movies dediğimde bu movies componentini görmek.

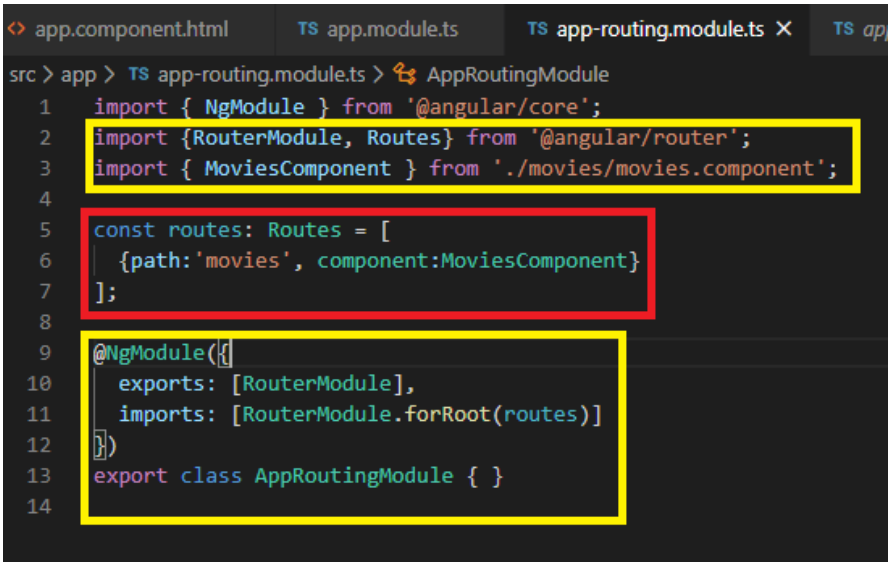
Yani routing bilgileri ile componentler ilişkilendirilecek böylece ugulamamız single page olmasına rağmen sanki çok sayfada çalışıyormuş izlenimi elde edilecek.

Uygulamamıza routing eklemek için öncelikle yeni bir module ekleyeceğiz:

ng g app-routing --flat --module=app diyerek hem yeni app-routing module'ünü ekliyorum hem de app.module içerisinde app-routing import edildi böyle routing'i app.module component'leri içerisinde kullanabileceğiz. Flat diyerek de şuan ihtiyacımız olmayan diğer dosyaların eklenmesinden kurtuluyoruz sanırım.

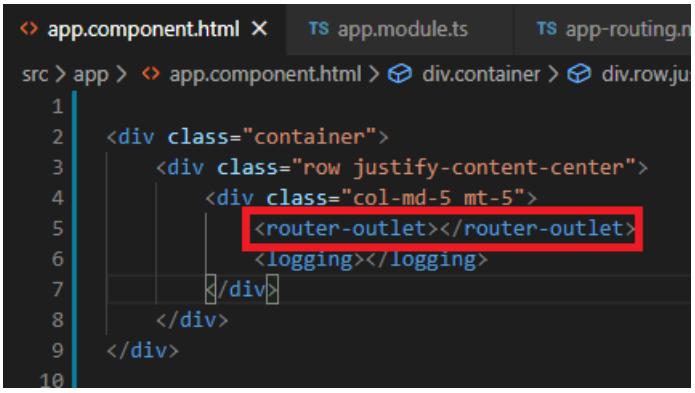


Burada app-routing.module.ts yukarıdaki gibi initialize edilecek. Declarations kısmında component'ler tanımlanıyordu, biz bu module için bir component tanımlamayacağız bu kısmı silebiliriz. Benzer şekilde CommonModule import'larını da silebiliriz.



Sonuçta app-routing.module.ts içerisinde routes constant'ı içerisinde bir link tanımladım bu link /movies linki ve bu link ile bir MoviesComponent'i bağladım.

Şimdi app.component.html'e gidiyorum, buraya aşağıdaki gibi <router-outlet> etiketini eklediğim zaman, bu şu demek oluyor:



- Eğer localhost:4200 sayfasında yani anasayfadaysan bu kısımı gösterme hatta çalıştırma ancak bu etiket dışında kalan diğer kısımlar gösterilecek! Tabi logging zaten movies component'inin çalışmasına bağlı olduğu için şu durumda o component'de görünmeyecek.
- Eğer localhost:4200/movies dersek bu kez <router-outlet> etiketi yerinde movies etiketi varmış gibi sayfa çalışacak.

PART 12: Dashboard

Şimdi bazı yeni componentler oluşturacağız ve anasayfaya yeni bir görünüm vereceğiz. Bunun için bazı film fotoğrafları indirdik.

İlk olarak **ng g c navbar** ile bir navbar componenti oluşturuyoruz.

Daha sonra **ng g c dashboard** ile bir dashboard componenti oluşturuyoruz.

Şimdi navbar.component.html'in içini dolduralım, burada href yerine routerLink ile link verdiğimize dikkat et, bunu yapınca sayfa yenilenmemiş oluyor anladığım kadarıyla.

navbar.component.html | app.component.html

src > app > navbar > navbar.component.html > div.navbar.navbar-expand-sm.bg-prima

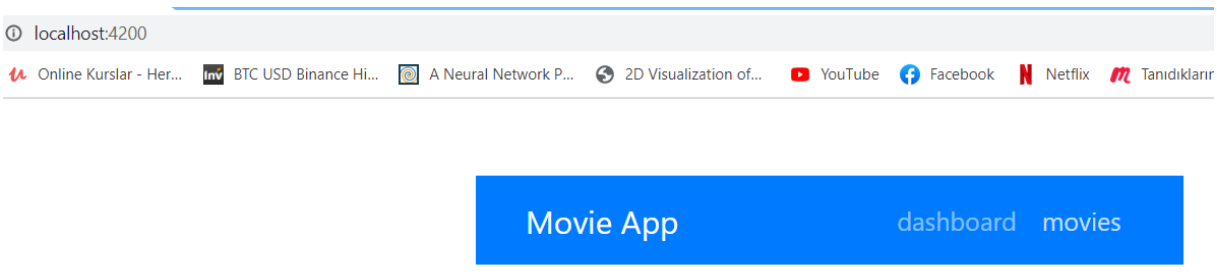
```
<div class="navbar navbar-expand-sm bg-primary navbar-dark">
  <div class="container">
    <a routerLink="/">Movie App</a>
    <ul class="navbar-nav">
      <li class="nav-item">
      <a routerLink="/dashboard" class="nav-link">dashboard</a>
      </li>
      <li class="nav-item">
      <a routerLink="/movies" class="nav-link">movies</a>
      </li>
    </ul>
  </div>
</div>
```

Daha sonra da bu navbar template'ini anasayfada görmek için app.component.html'in içinde <navbar> etiketi ile çağırırız.

navbar.component.html | app.component.html

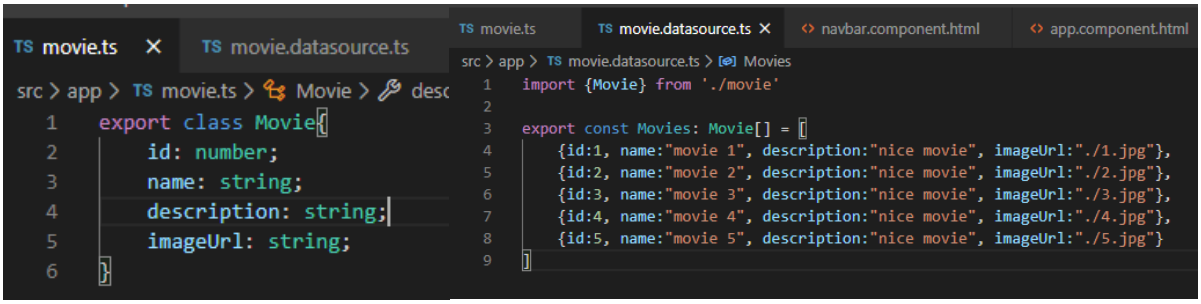
src > app > app.component.html > div.container > div.row.ju

```
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-5 mt-5">
            <navbar></navbar>
            <router-outlet></router-outlet>
            <logging></logging>
        </div>
    </div>
</div>
```

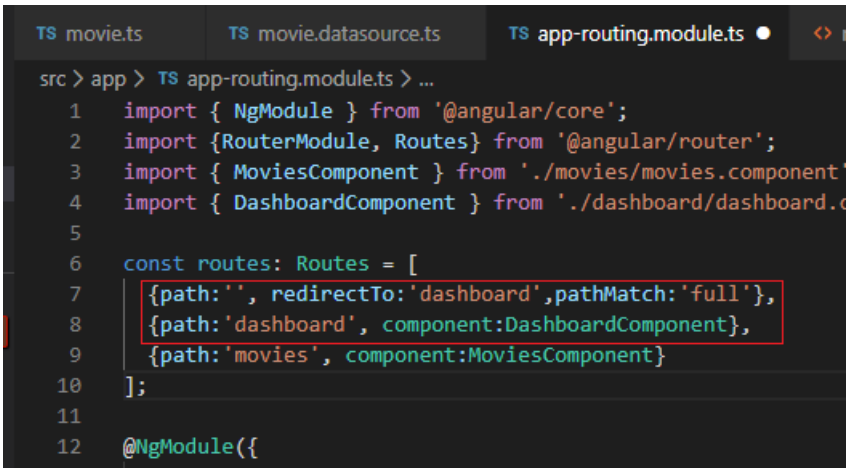


Burada artık movies'e tıklarsak /movies'e gideriz ve movies component'i görünür olur aynı şekilde Movie App'e tıklarsak ana sayfaya geri döneriz, fakat dashboard şuan routelink olarak tanımlanmadığı için buna tıkladığımızda hiçbir şey olmaz.

Dashboard linkini oluşturup component'i doldurmadan önce bir şey değiştirelim moive objesini biraz detaylandıralım ve description ile imageUrl özellikleri ekleyelim daha sonra da datasource'u buna göre değiştirelim:



Şimdi /dashboard için ve hiçbirşey girilmemesi durumu için iki route daha tanımlıyorum.



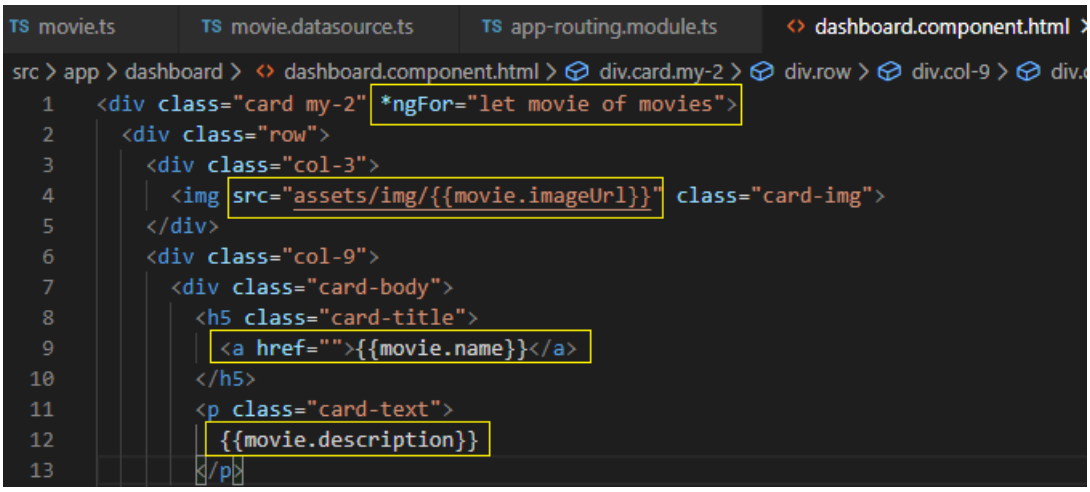
Burada 3 tane path belirtmiş olduk, ilki zaten homepage durumunda da /dashboard'a yönlendir diyor.

Artık biz app.component.html içinde <router-outlet> linkini çağırdığımızda eğer url / ise veya /dashboard ise <router-outlet> yerine <dashboard> yazmışız gibi olacak yani orada dashboard componenti çağırılacak.

Yok eğer url /movies ise o halde de aynı <router-outlet> yerine <movies> yazılmış gibi olacak yani movies componenti çalıştırılıp çağırılacak.

Böylece routing yapmış olacağız.

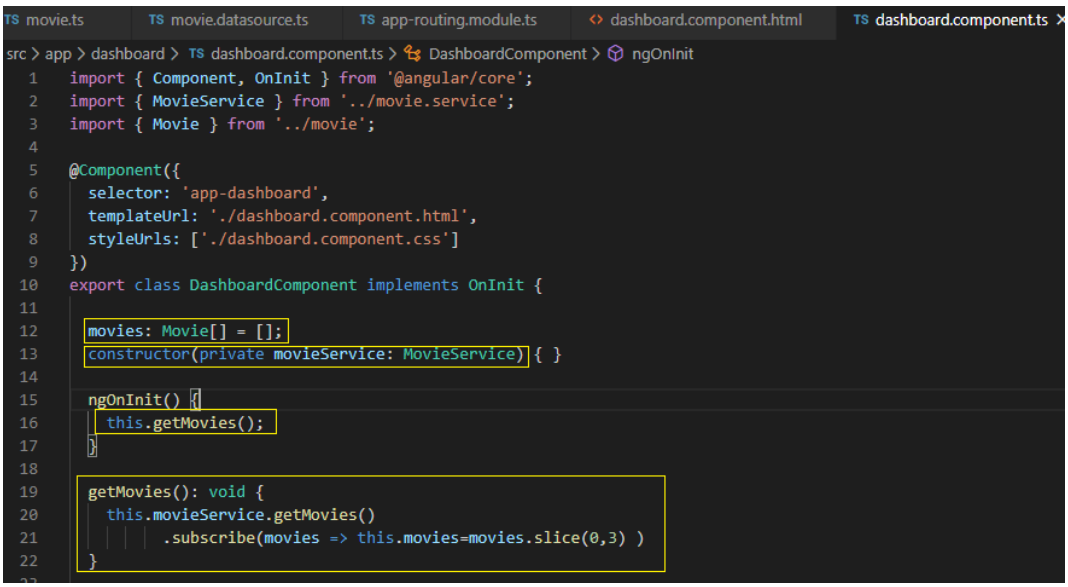
Şimdi gidelim ve dashboard.component.html içeriğini dolduralım, burada 5 adet filmi fotoğrafları ile birlikte ekrana getirelim.



Yukarıda görüldüğü gibi dashboard template'i , dashboard component'i içerisinden bir movies listesini alır ve bu listenin her bir elemanı için image, name ve description'ı gösterir.

Burada {{}} sembolünün src="" içerisinde dahi kullanılabildiğine dikkat et.

Elbette bu dashboard template'in çalışması için, dashboard component içerisinden movies listesi hazırlanmalı:

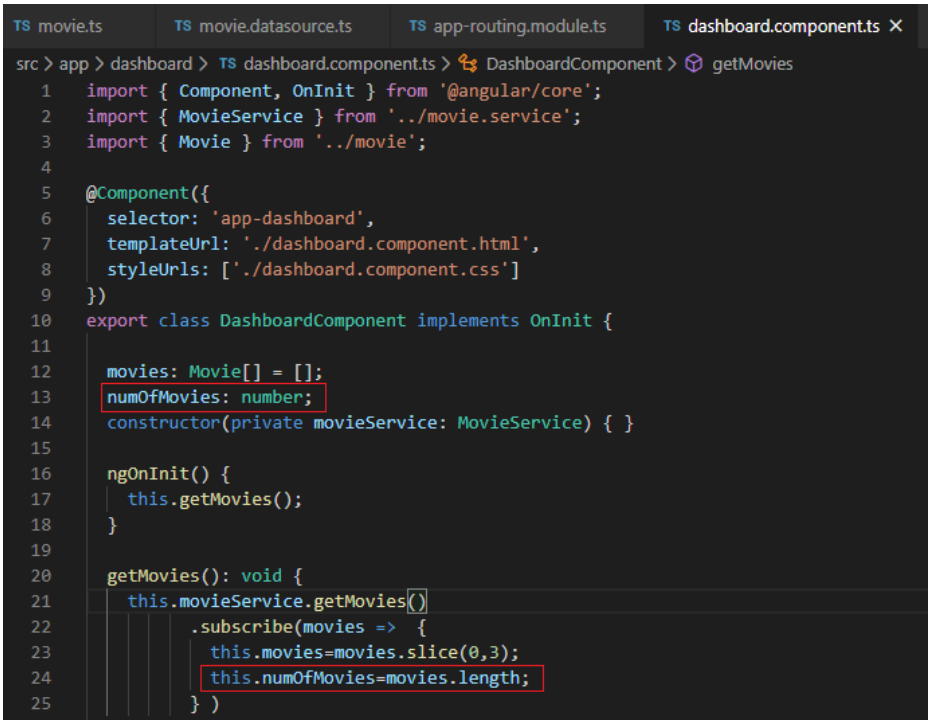


Görüldüğü üzere burada önce movieService yaratılıyor ve service kullanılarak tüm filmler getMovies() ile elde ediliyor daha sonra bunların ilk 3'ü component'in movies'i içine atılıyor.

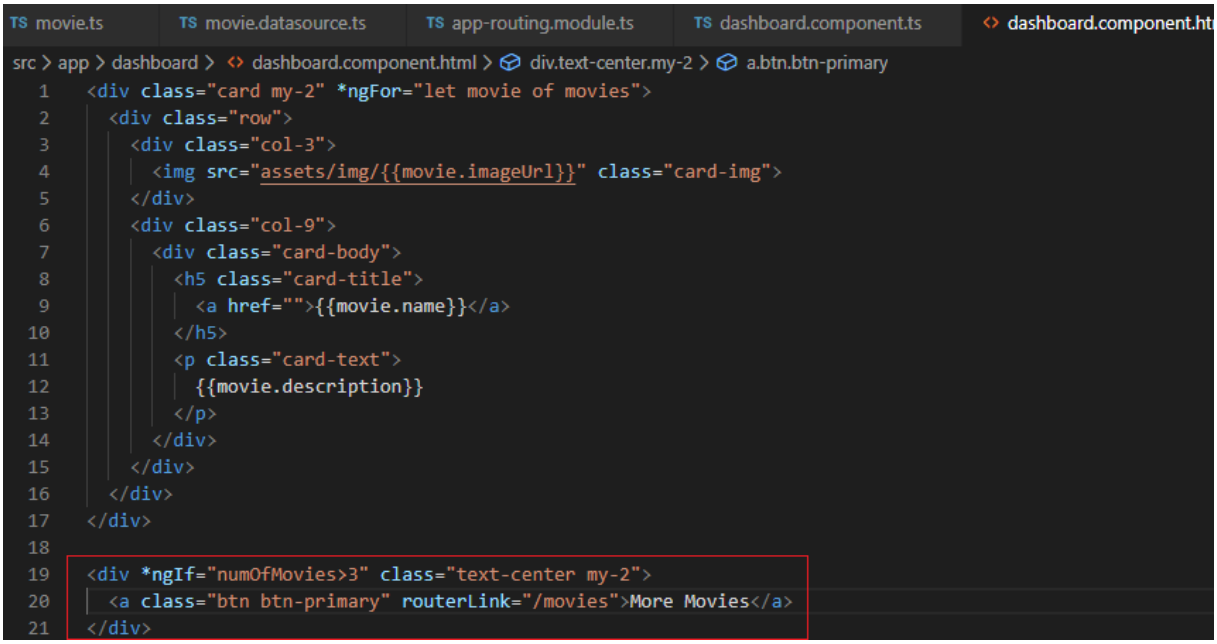
Artık dashboard template'i movies'i kullanılabilecek durumda.

Son olarak dashboard componentine bir buton ekleyelim ve bu buton bizi bütün filmlere götürsün.

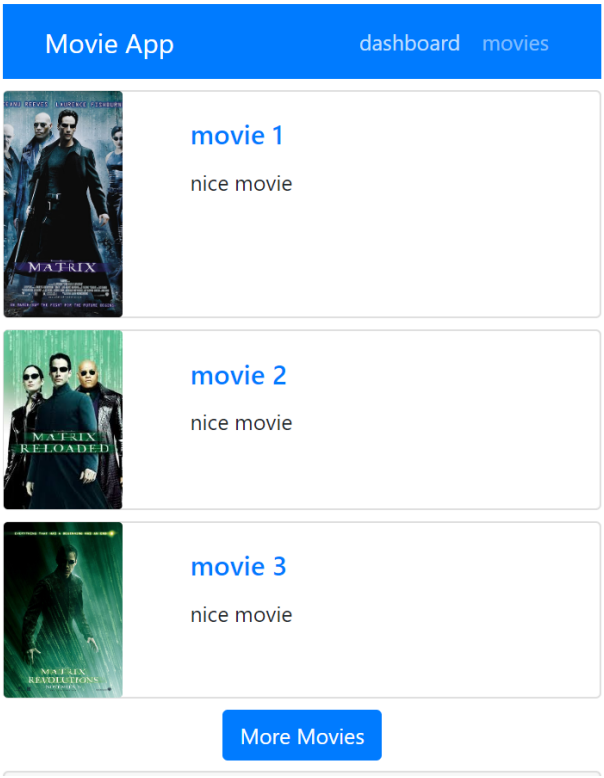
Öncelikle html içerisinde kullanmak için movieservice ile alınan movies listesinde kaç film olduğunu numOfMovies değişkeninde tutuyoruz.



Şimdi de dashboard içerisinde yeni bir div oluşturuyoruz bu div ancak eğer dashboard'da gösterilen film sayısı (bunu 3 olarak ayarlamıştık) numOfMovies'ten küçükse görünür olsun, eğer değilse div görünmez olsun ve butona tıklanırsa bizi /movies'e yönlendirsin dedik.



Sonuçta aşağıdakine benzer bir app'imiz oldu, anasayfada navigation, dashboard ve logging componenti görünüyor eğer movies'e tıklarsak dashboard yerine movies componenti görünür oluyor diğer componentler olduğu gibi kalıyor.

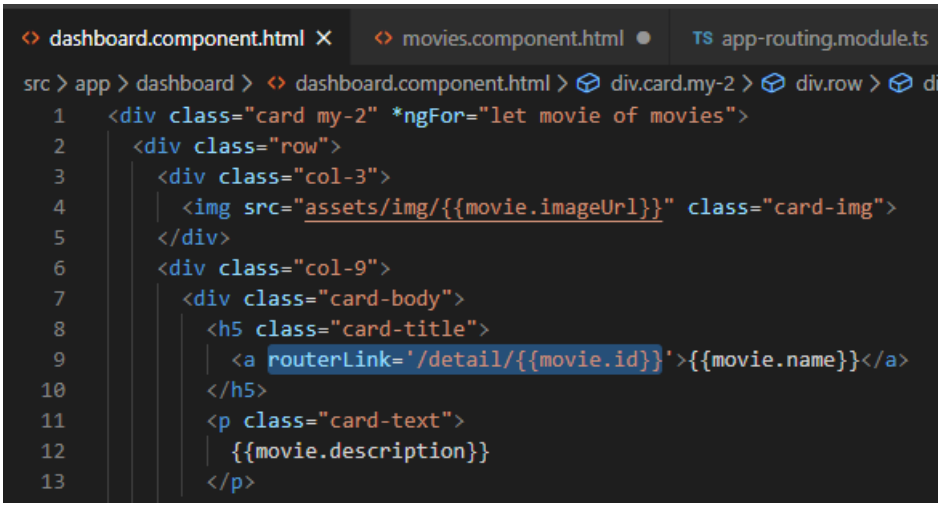


PART 13: Movie Detail Gösterimi

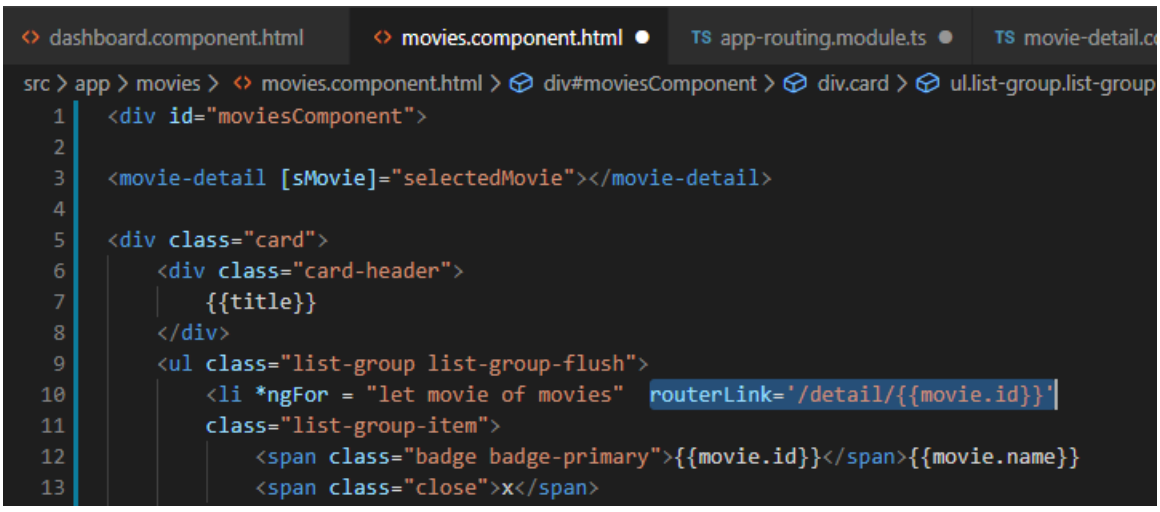
Şimdiye kadarki kısımda movies componenti içerisinde movie detail componentini kullanarak küçük bir kısımda movie detayını gösteriyorduk.

Ancak istiyoruz ki routing ile örneğin /detail/2 dediğimizde id'si 2 olan filmin detail sayfasını görüntüleyelim. Bu kısımda bununla uğraşacağız.

Öncelikle dashboard'a gidiyorum ve dashboard üzerindeki movie isimlerine tıklanınca gidilmesi gereken routing'i belirtiyorum.

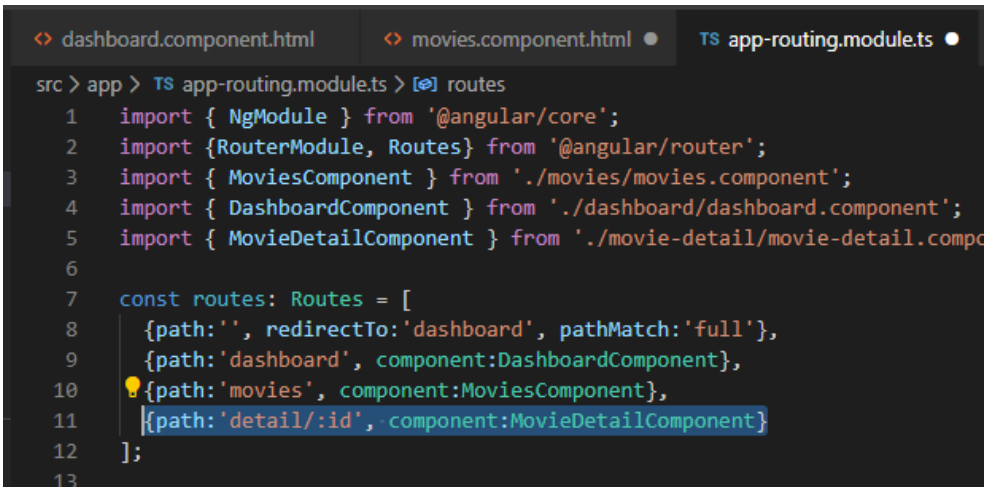


Benzer şekilde movies template'i içindeki fiml isimlerine de routerLink atıyorum. Bir filme tıklanınca /detail/id'ye gidilecek.

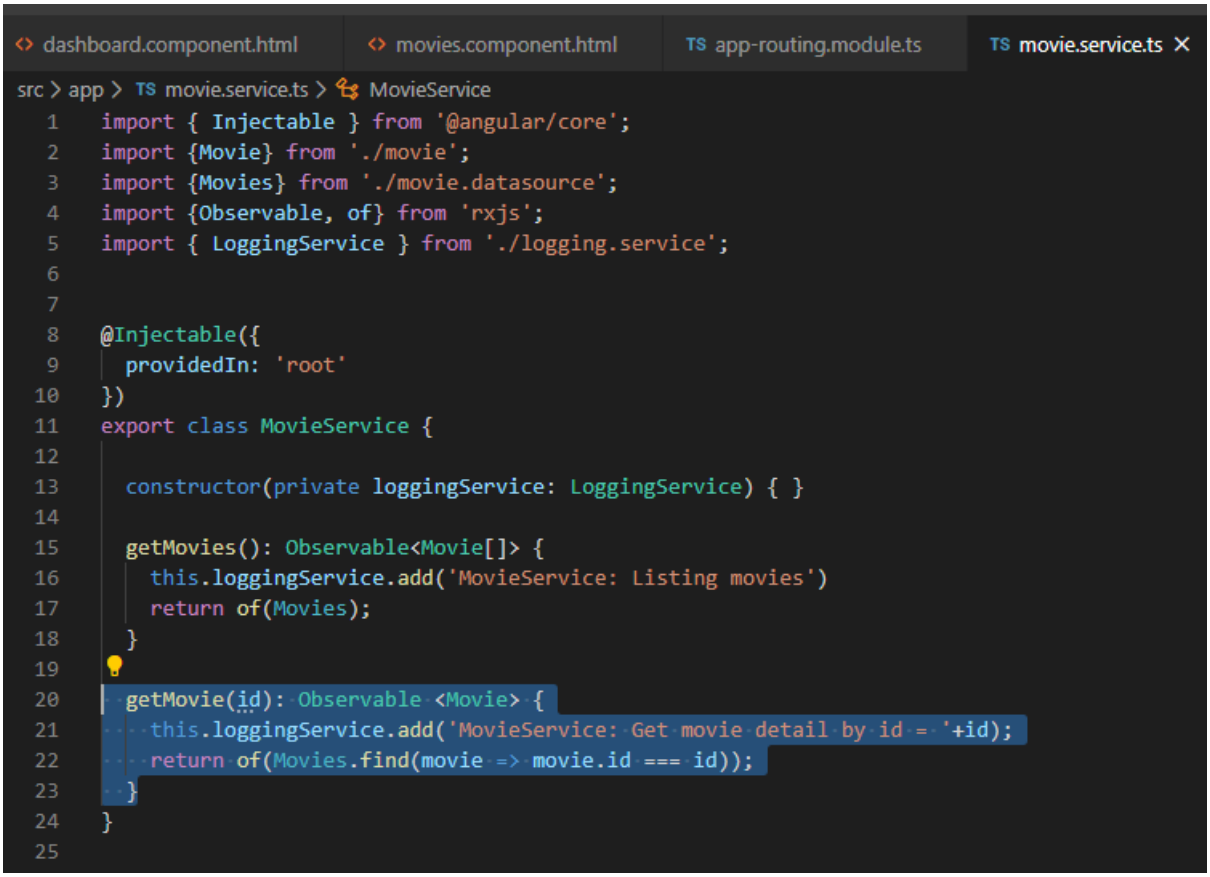


Ancak henüz bu route app-routing.module.ts içerisinde belirtilmedi, o yüzden çalışmayacaktır, routing'i belirttiğimizde hangi module ile route'un pair olacağını belirtiriz. Ayrıca bu sayfadan <movie-detail> tag'ini artık silelim. Çünkü artık detail'i movies componenti altında değil, router ile app.componentde göreceğiz.

Dikkat edersen buradaki routing biraz farklı belirtildi, denildi ki /detail/id verilirse o zaman <router-outlet> tag'inin olduğu yere MovieDetailComponent'i çağır.



Önce gidip movie service içerisinde bir getMovie() methodu tanımlıyorum ki bu method içine bir id alıp, movie'leri arasından ilgili id'li olanı döndürüyor. Bu sırada observable kullanılıyor çünkü bu bilginin database'den check edildiğini düşünüyoruz sanırım.



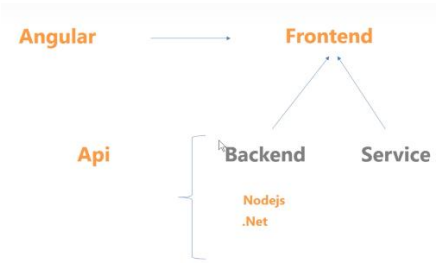
Şimdi detail component'inin içine bakıyoruz:

- Uygulama üzerinde detail/id şeklinde bir route girildiğinde ilgili component aktif olacak, routing'de id'yi :id şeklinde vermiştik, bu yüzden burada bu id'yi yakalamamız gerek.
- İd'yi yakalamak için route: ActivatedRoute şeklinde bir değişken tanımlıyoruz bunun dışında, detail component'inin içinde seçilen id'li movie'yi almak için service'i kullanıyoruz. Böylece tüm movie'leri almadan sadece service'ten ilgili elemanı alabiliyoruz.
- Bunun için görüşdüğü gibi öncelikli id'yi alıyoruz ve daha sonra movieService içinde tanımlanan getMovie methodu ile ilgil movie'yi alırız. Burada alınacak değer observable olacağı için subscrive ile alıyoruz.

Burada moviedetail component'in movie property'si eskiden kaldı hala @input decorator'ı ile duruyor çünkü önceden bu component içine movie'yi sMovie ismiyle pass ediyorduk.

PART 14: Http Servisleri

Bildiğimiz gibi Angular bir javascript uygulaması ve tamamen tarayıcı üzerinde çalışıyor. Yani angular frontend için kullanılıyor. Bir backend dili ile ise de database hazırlayabiliriz daha sonra yine backend tarafında Nodejs veya .Net gibi dillerle hazırlanan service'ler angular tarafında kullanılabilir böylece veri alınabilir.



Bunun dışında backend ile uğraşmayıp hazır olarak sağlanan service'leri kullanabiliriz. Anladığım kadarıyla bunlar api'ler oluyor, deneme amaçlı kullanabileceğimiz api'ler var. Bize verilen key ile request gönderip json tipinde veri elde edebiliyoruz.

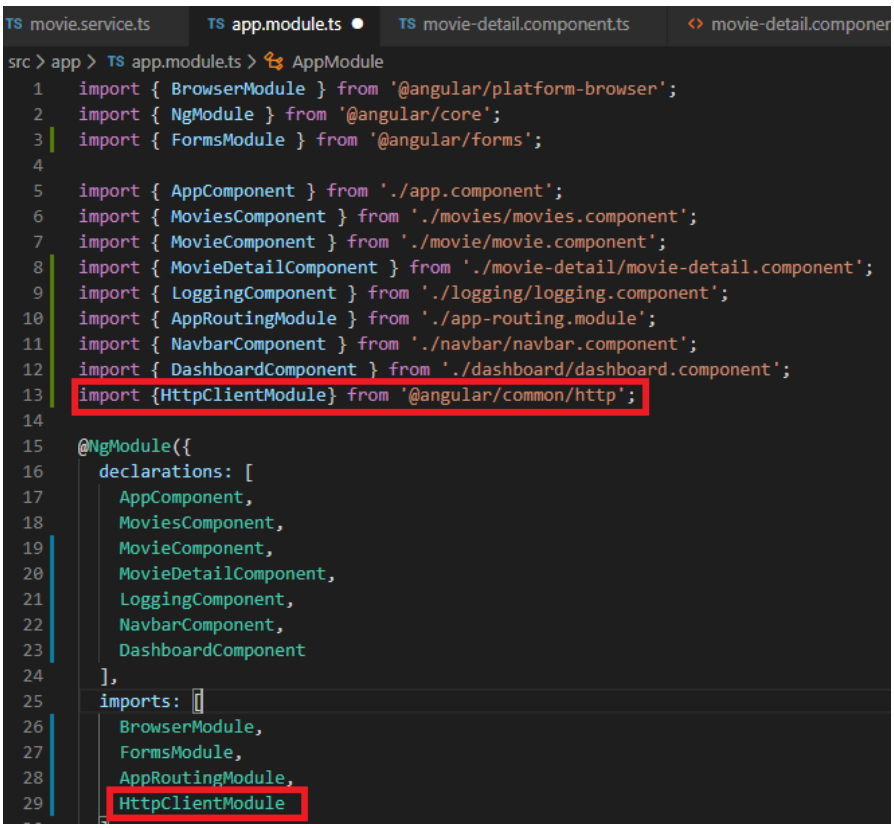
Bu dersteki amacımız bir API'a request yollayabilmek örneğin api.abc'ye bir request yollarız ve talep sonucunda elde edeceğimiz json tipindeki veriyi angular uygulamasında gösteririz. Bunu yapalım.

Daha önceden datayı elde etmek için movie.datasource dosyasını kullanıyorduk. Service içinde Movies'i import edip bir sorgu yapıyorduk ve sonucu observable olarak return ediyorduk.

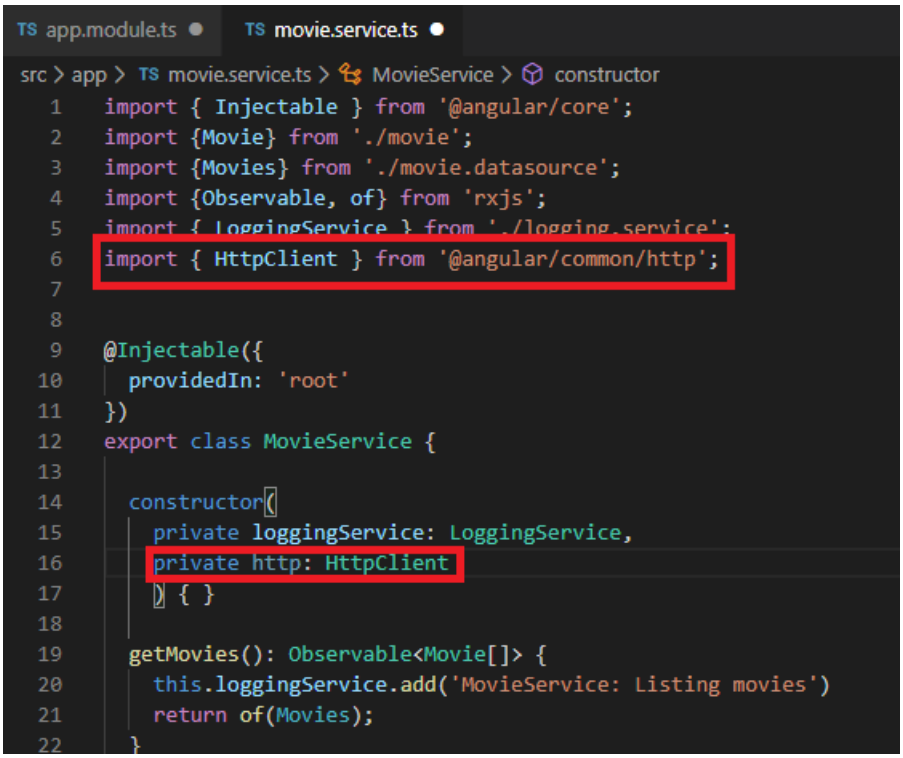
```
movie.service.ts | movie-detail.component.ts | movie-detail.component.html | dashbo
src > app > movie.service.ts > MovieService > getMovie
1   import { Injectable } from '@angular/core';
2   import {Movie} from './movie';
3   import {Movies} from './movie.datasource';
4   import {Observable, of} from 'rxjs';
5   import { LoggingService } from './logging.service';
6
7
8   @Injectable({
9     providedIn: 'root'
10  })
11  export class MovieService {
12
13    constructor(private loggingService: LoggingService) { }
14
15    getMovies(): Observable<Movie[]> {
16      this.loggingService.add('MovieService: Listing movies')
17      return of(Movies);
18    }
19
20    getMovie(id): Observable <Movie> {
21      this.loggingService.add('MovieService: Get movie detail by id='+id);
22      return of(Movies.find(movie => movie.id === id));
23    }
24  }
25
```

Şimdi ise bu bilgilere httpclient aracılığı ile erişmek istiyoruz, sanki bilgileri API'dan alıyormuşuz gibi yapacağız. Angular kütüphanesinin içinde API'ı taklit etmesi için hazır bir yapı var. Biz bu yapının içine daha önce kullandığımız datasource file'ının içindeki movie listesini aktaracağım ve daha sonra bu bilgilere httprequest'leri ile ulaşacağım. Böylece bir API kullanınca nasıl bilgi alacağımızı da anlamış olacağız.

1. Uygulamada http servislerini kullanabilmek için yapmamız gereken ilk şey HttpClientModule'ünü appModule içerisine eklemek.



2. Şimdi bir service içerisine HttpClient'ı inject ederek service üzerinden bir talepte bulunabiliriz. Burada movie service içinde http ismiyle inject edildi.

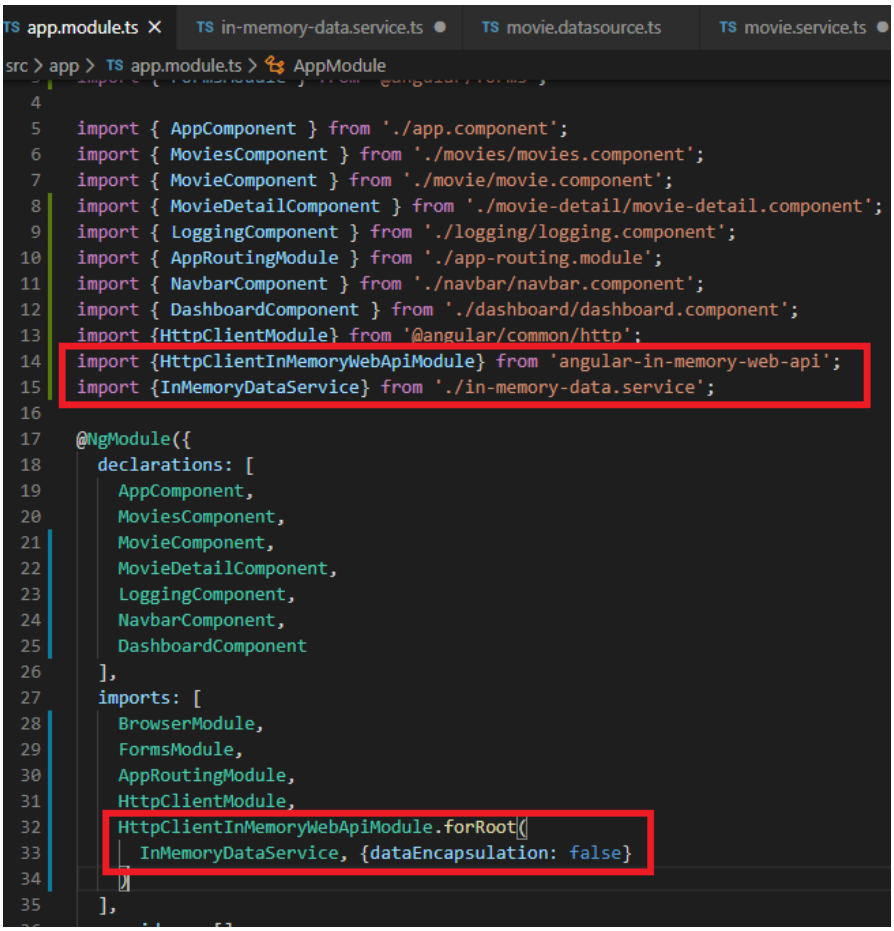


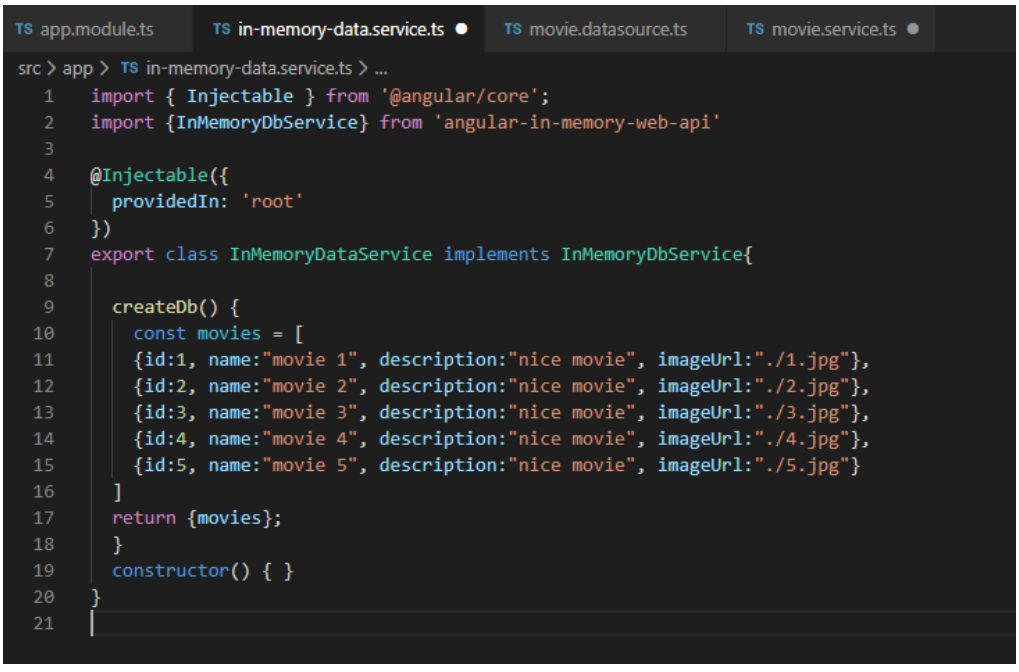
- Artık örneğin getMovies() gibi bir methodun altında **http.get(apiUrl)** formatı ile bir api üzerinden veri alabiliriz.
- Ancak biz şuan böyle yapmayacağız çünkü bir API kullanmak yerine, uygulamamız içerisinde bir web API taklitçisi kuracağız, işlemlerimizi de bunun üzerinden yapacağız.

3. Öncelikle **npm install angular-in-memory-web-api –save** diyerek bu web api taklitçisini kullanmak için gerekli dependency'leri kuruyoruz.

4. Daha sonra movie listesini direkt datasource.ts dosyası şeklinde değil de bir servis aracılığı ile dışarıya açmak için yeni bir servis oluşturuyoruz. **ng g s InMemoryData**

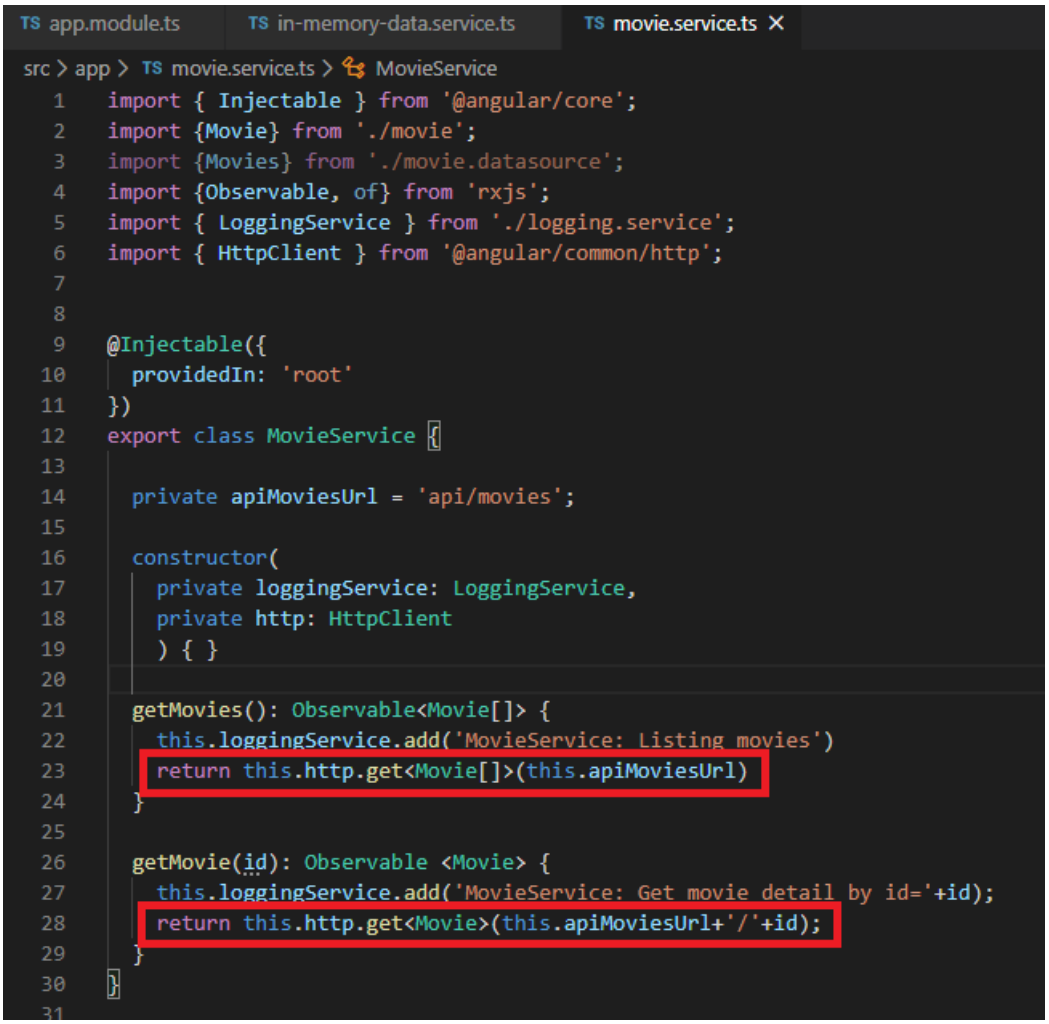
5. Şimdi ise taklit bir api kullandığımız için oluşturulan InMemoryData servisi ile biraz evvelki HttpClientMemoryWebApi module'ünün bağlantısını yapacağız. Aşağıda yapılan işlemi gerçek API'lerle uğraşırken yapmayız.





Yukarıdaki satırlarla artık in-memory-data.service.ts içerisinde createDb() içerisindeki movies listesini sanki **api/movies** gibi bir url ile ulaşabildiğimiz bir api servisi gibi kullanıyoruz.

Artık movie.service.ts içerisinde tanımlanan httpClient objesi üzerinden get dedikten sonra return tipini <> içine belirtip, url'yi veridğimizde veriyi elde edebiliyoruz.



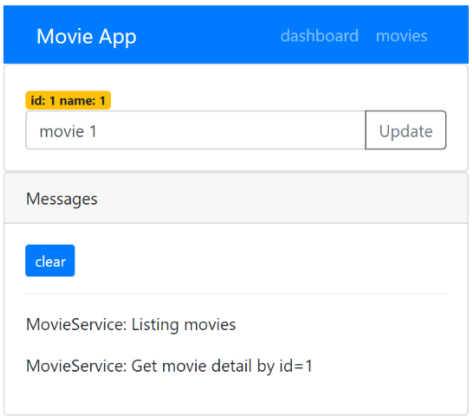
api/movies ile ulaşabilmemizin sebebi, createDb içinde movie değişkeninir return etmemiz.

Eğer id'ile selection yapmak istersek, **api/movies/id** şeklinde bir url ile get methodunu çalıştırmamız yeterli olacaktır.

Unutma gerçek bir api kullansaydık, önceki sayfada yapılanlar anlamsız olacaktı.

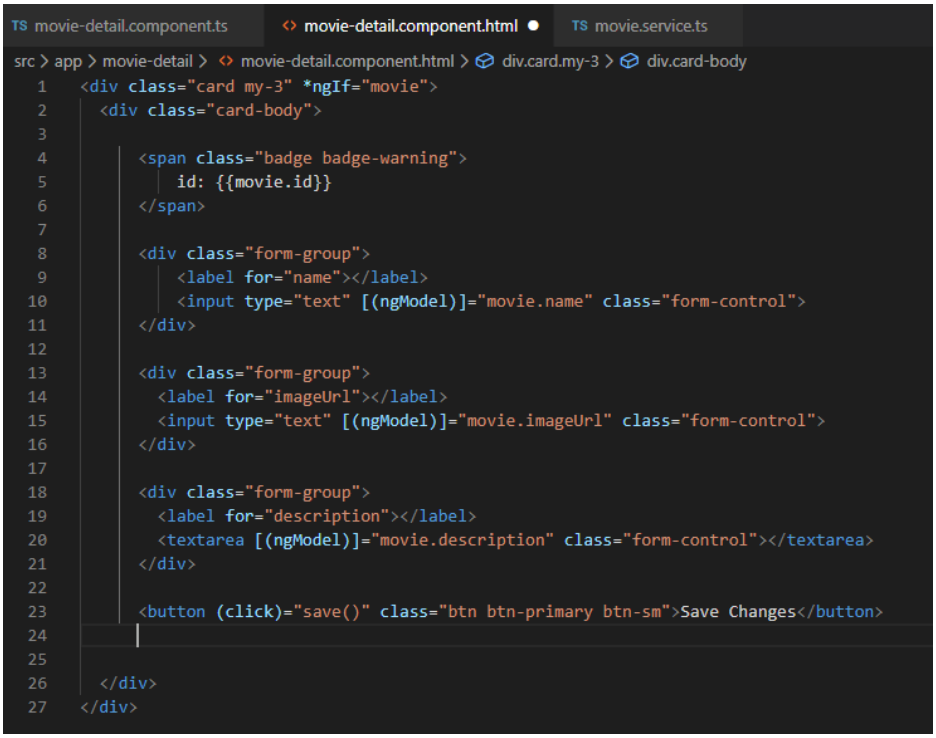
PART 15: Movie Kaydını Güncelleme

Uygulamanın şuanki halinde bir movie'ye tıkladığımızda movie-detail component'i aracılığı ile detay kısmını aşağıdaki gibi görebiliyoruz, burada id, name görünüyor ve bir Update methodu yer alıyor.

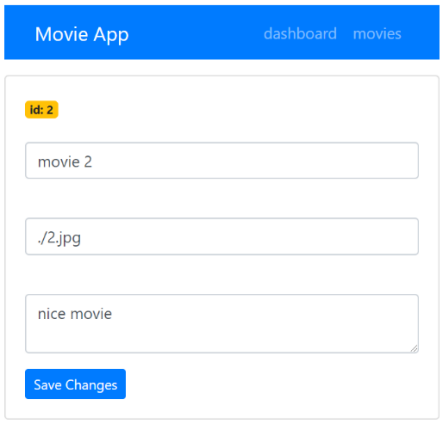


Amacımız şu, burada movie'nin description ve imageUrl'si de görünsün ayrıca da Update butonunu canlandıralım, böylece movie kaydını güncelleyebilelim.

Öncelikle detail template'ini güncelliyoruz, movie name, url ve description kısmını input olarak veriyoruz ve bunları ngmodel ile componente double bind ediyoruz. Eski update butonu yerine vir Save Changes butonu ekliyoruz ve tıklanınca component içindeki save() methodu çağrılsın diyoruz.



Sonuçta detay sayfası aşağıdaki gibi görünsün istiyoruz:



Şimdi template'te belirtilen save methodunu burada dolduruyoruz. Save methodu ile yapılmak istenen database'i update etmek. Daha sonra da bir önceki sayfaya geri dönmek.

```
movie-detail.component.ts ×   movie-detail.component.html ●   TS movie.service.ts
rc > app > movie-detail > TS movie-detail.component.ts > MovieDetailComponent > ngOnInit
1   import { Component, OnInit, Input } from '@angular/core';
2   import { Movie } from '../movie';
3   import { MovieService } from '../movie.service';
4   import { ActivatedRoute } from '@angular/router';
5   import { Location } from '@angular/common';
6
7   @Component({
8     selector: 'movie-detail',
9     templateUrl: './movie-detail.component.html',
10    styleUrls: ['./movie-detail.component.css']
11  })
12  export class MovieDetailComponent implements OnInit {
13
14    @Input() movie:Movie
15
16    constructor(
17      private movieService: MovieService,
18      private route: ActivatedRoute,
19      private location: Location
20
21    ) { }
22
23    ngOnInit() {
24      this.getMovie()
25    }
26
27    getMovie(): void{
28      const id = +this.route.snapshot.paramMap.get('id');
29      this.movieService.getMovie(id)
30            .subscribe(movie => this.movie = movie);
31    }
32
33    save(): void {
34      this.movieService.update(this.movie)
35          .subscribe(()=>{
36            this.location.back();
37          });
38    }
39  }
40
```

Database'i update etmek amacıyla movies service'i içerisinde bir update methodu tanımlayacağız component içinden bu methodu çağıracağız. Bir önceki sayfaya geri dönmek için ise location class'ını import edeceğiz ve bu

sayede yukarıda göründüğü gibi, update işleminden sonra bir önceki sayfaya geçeceğiz.

Burada önemli nokalardan biri şu, save içindeki update methoduna this.movie gönderiliyor, bu update edilmiş movie oluyor çünkü ngModel ile double side bind işlemi yaptık. Bu sayede içerik değişince component içindeki movie de değişir.

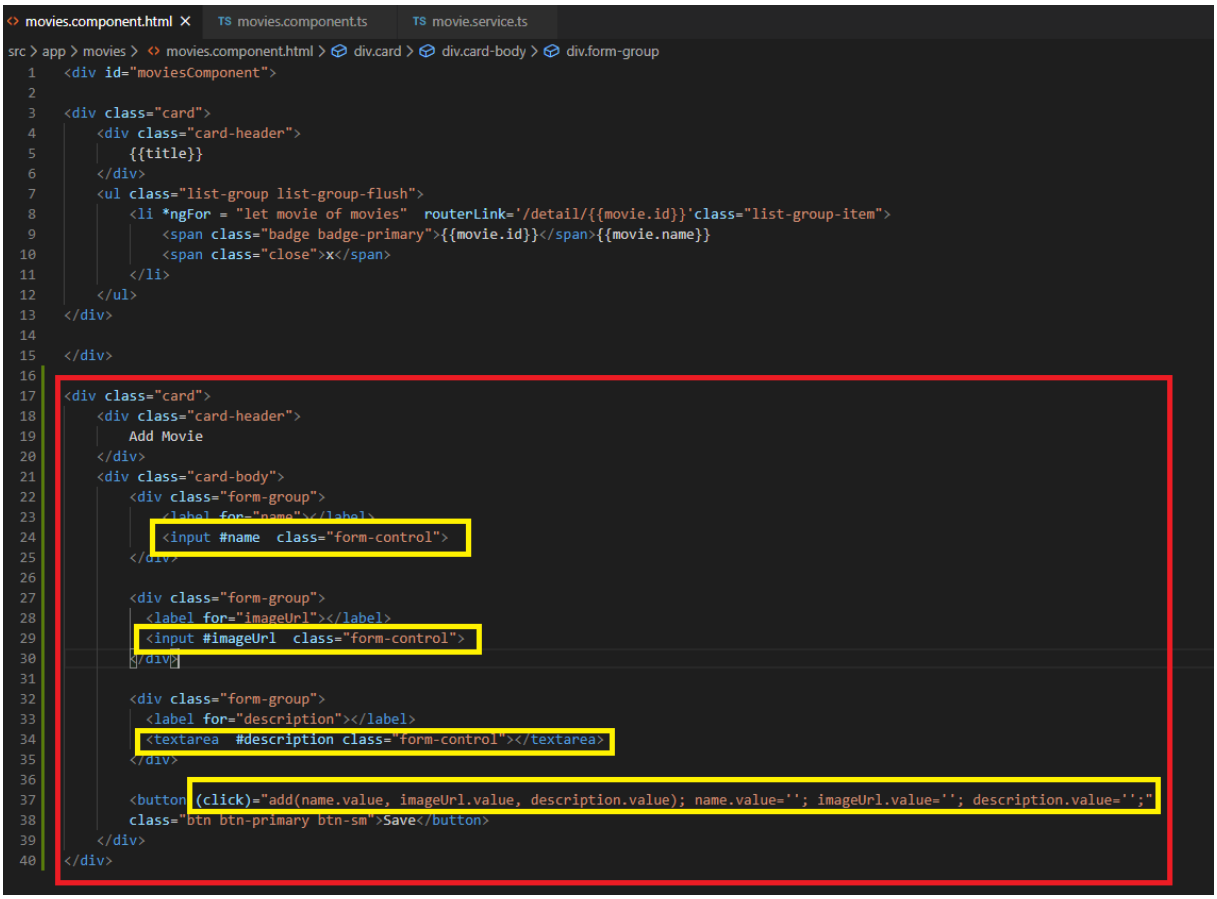
() => {statements} denmesinin sebebi update methodunun bir şey return etmemesi o yüzden input olarak () alıyoruz. {} içinde ise yapılmak istenen yapılıyor.

Şimdi update methodunun service içinde nasıl tanımlandığına bakalım:

http.put ile api üzerinden update işlemi yapılıyor, url veriliyor ve ilgili movie veriliyor. Bu sayede ilgili movie id'sine göre yeni özelliklerini alıyor.

PART 16: Yeni Movie Kaydı Ekleme

Şimdi öncelikle, movies template'ine yeni bir kısım ekliyorum, burada input'lar olsun ve bir save butonu olsun böylece yeni bir movie girebilelim.



Yukarıda görüldüğü gibi input'lar içindeki bilgiyi component'e gönderebilmek için #name, #imageUrl, #description gibi id'ler kullandık böylece click eventi içinde bu elemanların value'sını name.value, imageUrl.value veya description.value şeklinde componente add methodu içinde gönderbiliyoruz.

Add methodu çalıştıktan sonra da input alanları boşalsın diye name.value='' şeklinde ifadeler yer alıyor.

Şimdi de component içinde ilgili movie bilgilerinin nasıl alınıp, yeni bir movie kaydı oluşturulduğuna bakalım:

Görüldüğü gibi, add methodu içinde alınan movie bilgileri ile yeni bir movie objesi oluşturuluyor ve bu obje componentin içindeki movieService objesi aracılığı ile add yapılıyor.

Daha sonra movieservice'in add methodunun döndürdüğü movie değeri alınıp, push ile sayfadaki movies'e ekleniyor, böylece anlık olarak sayfada yeni

movie'yi görebiliriz. Bunu yapmasak da movie eklenecekti ama görmek için tekrar movies linkine tıklamamız gerekecekti.

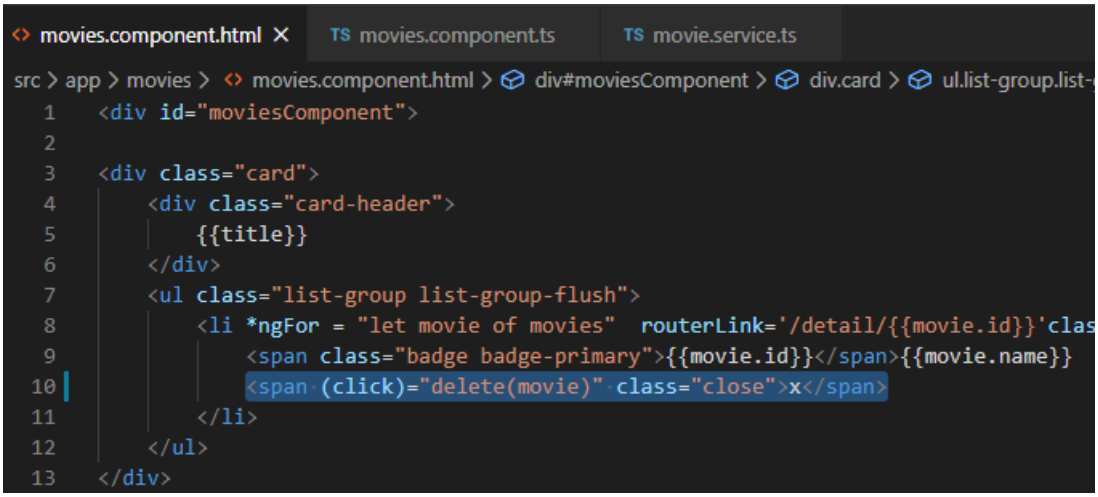
Son olarak da movie service içinde tanımlanan, database'e yeni elemanı ekleme işini yapan kısma bakalım. Görüldüğü gibi buradaki add methodu bir input olarak verilen movie objesini post request'i ile database'e ekliyor ve sonucunda bir observable movie döndürüyor.

PART 17: Movie Kaydını Silme

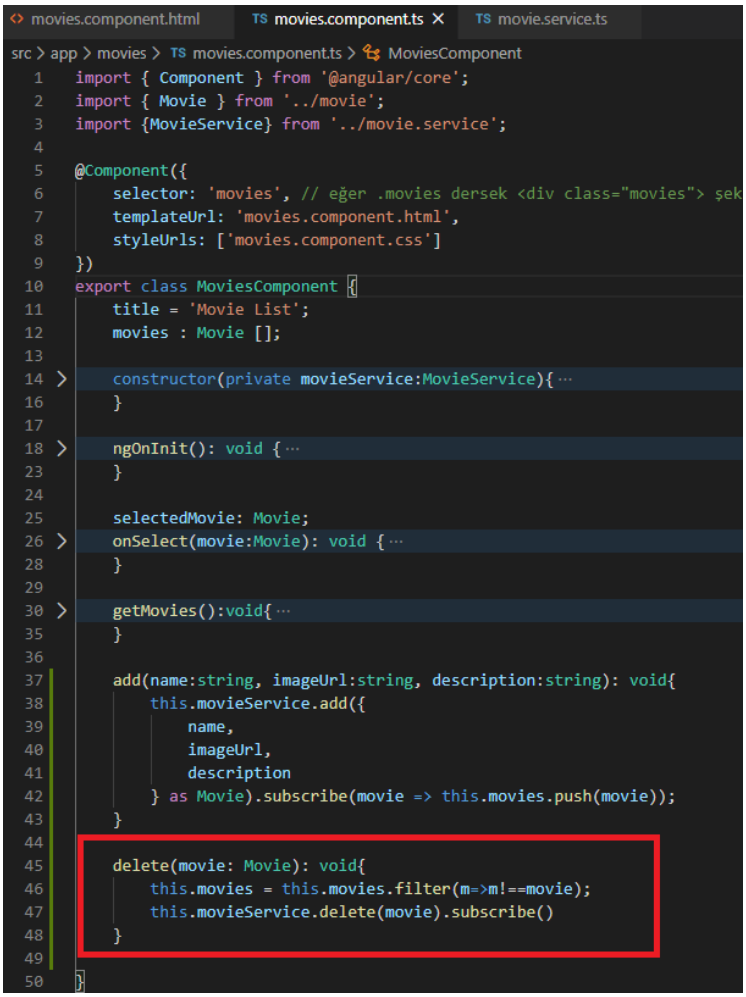
Bu uygulamada yapacağımız son şey de, movies template'inde görünen çarpılara basınca ilgili filmi silmek olacak. Hem anlık görünmesi için component'in listesinden sileriz, hem de service üzerinden http kullanarak, database'den bu kaydı sileriz.

Bunu yapalım:

Öncelikle template içerisinde çarpı işaretine bir onclick eventi ekleyelim ve component içindeki delete methodu ilgili movie'yi input alarak çalışsın diyelim.



Template'de çağırılan delete methodunu componentte dolduralım, burada iki farklı işlem yapılacak. İlgili movie hem component'in movies listesinden silinecek hem de service ile database'den silinecek.



Yukarıdaki şekilde filter ile this.movies'den movie siliniyor. Yani ilgili movie'ye eşit olmayan tüm movie'ler this.movies'e atılıyor.

Daha sonra'da service'den delete methodu çağırılıyor ve return değeri subscribe ile alınıyor.

Son olarak service içinde delete methodunu dolduralım. Göründüğü gibi delete methodunun içinde http.delete'i kullanıyoruz buna bir url veriyoruz, mesela api/movies/2 diyoruz ve id'si 2 olan movie siliniyor sonucunda da silinen movie observable olarak return ediliyor.