

The background features a large, dark grey arrow pointing from the left towards the center. Behind the arrow, several thin, curved lines in shades of grey and blue radiate outwards, creating a sunburst or gear-like effect.

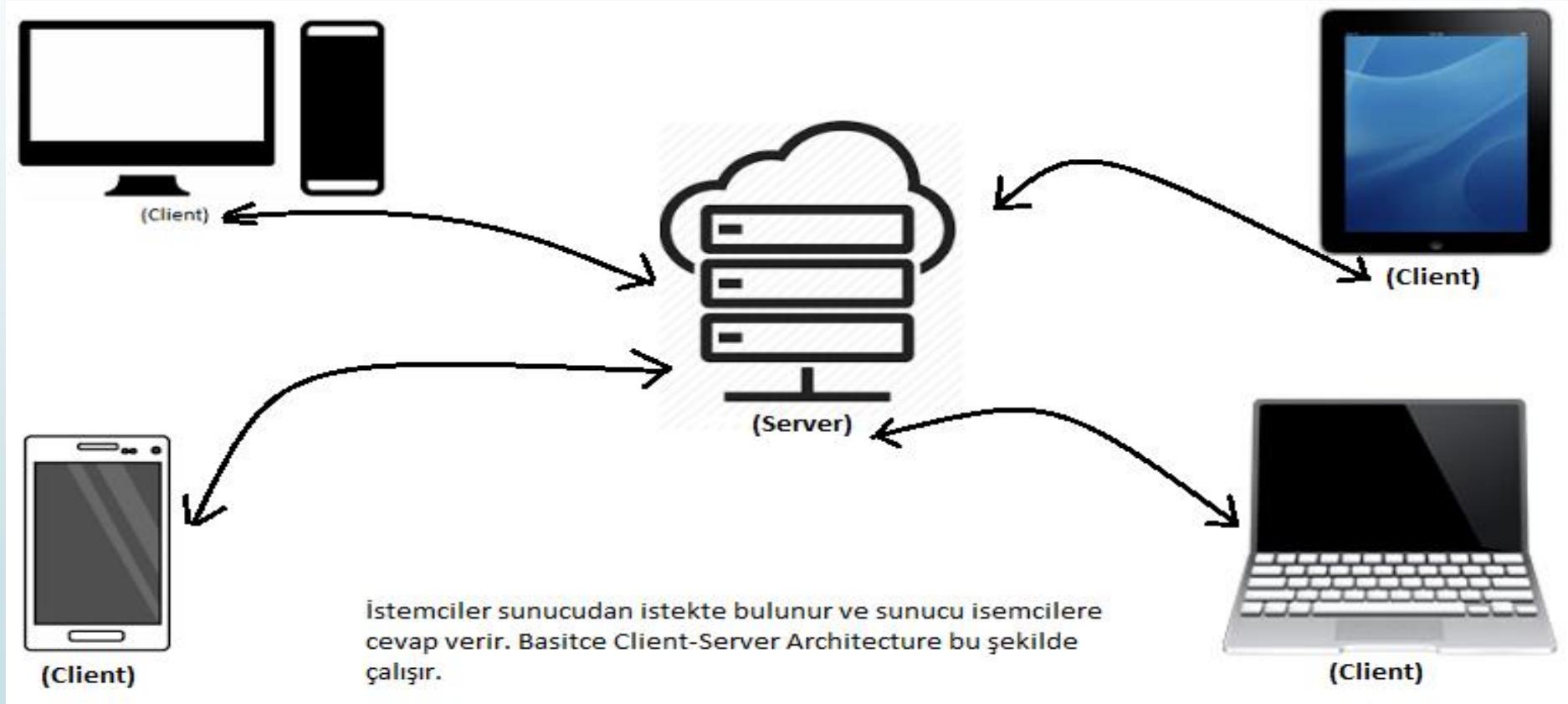
# Angular 16

MUSTAFA ÇAVDAROĞLU

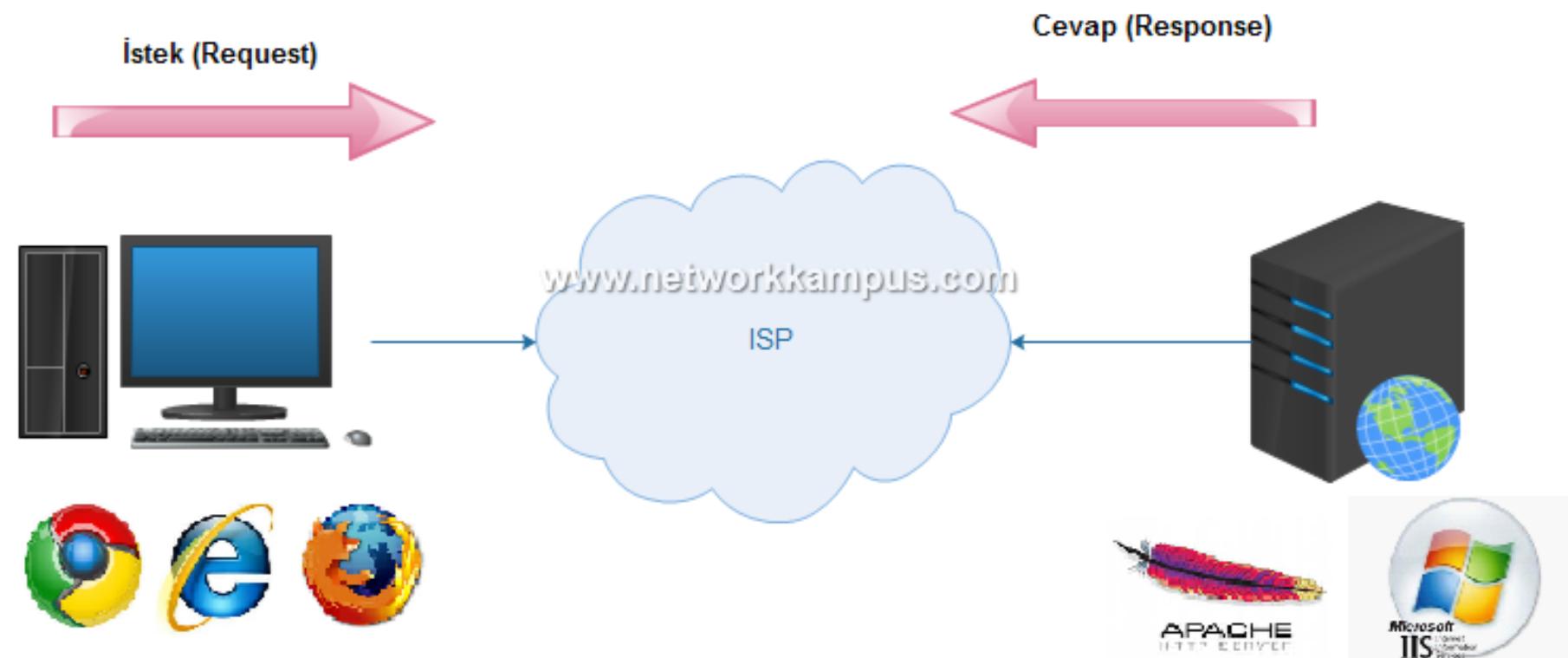
# Konular

- ▶ Middleware
- ▶ Dependency Injection
- ▶ Area'lar
- ▶ Secret Manager Tools
- ▶ Environment
- ▶ ViewModel(vm)&Data Transfer Object(DTO)
- ▶ Çok katmanlı Mimari ve Kurumsal Mimari
  - ▶ BL
  - ▶ Model
  - ▶ Dal
  - ▶ Core
  - ▶ Web.UI

# Client(İstemci) – Server(Sunucu) Mimari

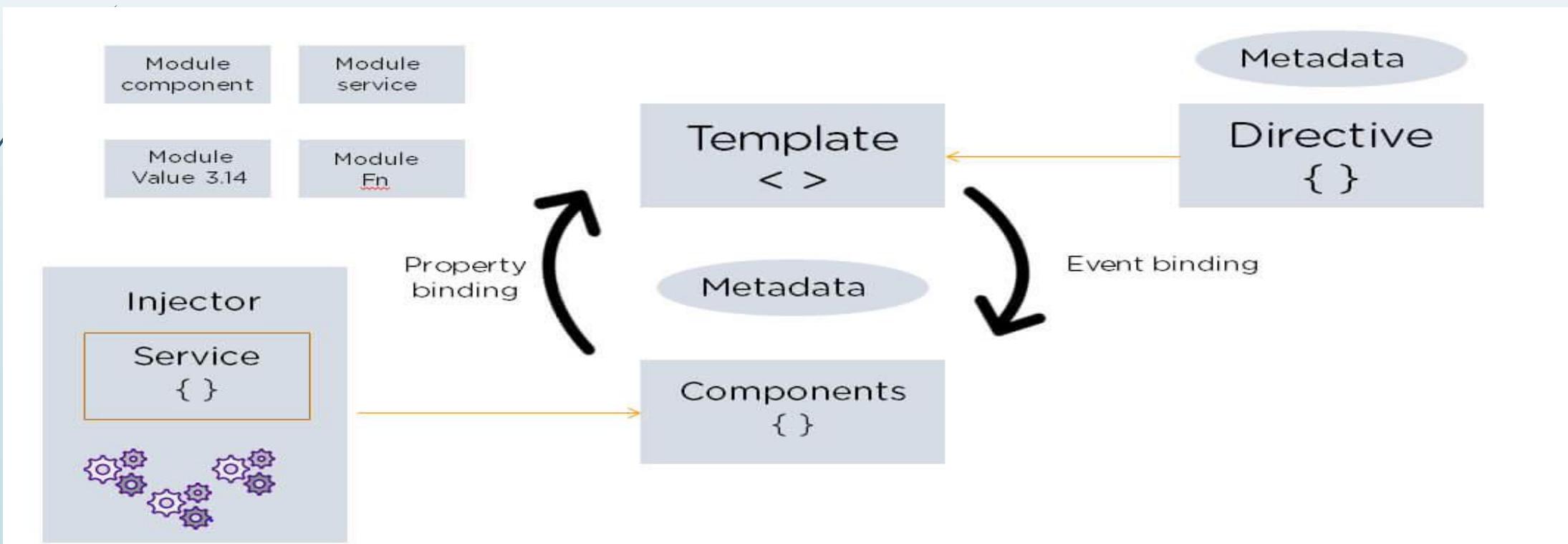


# Client(İstemci) – Server(Sunucu) Mimari



# Angular Nedir

**Angular, TypeScript** ile yazılmış açık kaynaklı bir **JavaScript framework'ü**dür . Angular Google tarafından desteklenir ve birincil amacı tek sayfalık uygulamalar geliştirmektir. Bir framework olarak Angular, geliştiricilerin birlikte çalışması için standart bir yapı sağlarken açık avantajlara sahiptir. Kullanıcıların sürdürülebilir bir şekilde büyük uygulamalar oluşturmasını sağlar.



# Angular Dosya Yapısı

- ▶ **e2e(End to End)** : Unit teslerin ve diğer testlerin konfigürasyonuyla ilgili dosyaları içerir.
- ▶ **node-modules** :Node.js aracılığıyla indirilen paketleri içerir.
- ▶ **src** :Uygulamanın dosyaları bu klasörde mevcuttur. Önemlidir. Bizim çalışmalarımızın %95'i bu klasör içerisinde ceyran edecektir.
  - ▶ **app** :Bu klasörle ilgili bir sonraki yazımızda daha detaylı bir değerlendirmede bulunacağız.
  - ▶ **assets** : Siteyle ilgili tüm varlıklar(image vs.) burada bulunmaktadır.
  - ▶ **environments** : Geliştirme ve yayına alma ortamıyla ilgili konfigürasyonu barındırır.
    - ▶ **environment.ts** :Geliştirme ortamı için konfigürasyonu sağlar.
    - ▶ **environment.prod.ts** :Yayın ortamı için konfigürasyonu sağlar.
  - ▶ **index.html**: Temel geliştirme sayfamızdır. İçinde temel direktifi barındırır.
  - ▶ **main.ts** Uygulamanın giriş noktası. Program.cs'si diyebiliriz. Uygulamanın compiler edilmesinden tutunda hangi modülün ana modül olarak ayarlanacağı bu yapı içerisinde belirtilir.
  - ▶ **polyfills.ts** : Uygulamadan önce tarayıcıya yüklenecek olan "polyfills" listesini tutmaktadır. Uygulamanın farklı tarayıcılarda olası desteklenmemeye durumlarında devreye girmektedir.

# Angular Dosya Yapısı

- ▶ **.editorconfig**

Editörle ilgili temel konfigürasyonları barındırır.

- ▶ **package.json**

NPM ile indirilen paketleri tutar. İçeriğine bakarsak eğer;

- ▶ **.editorconfig**

Editörle ilgili temel konfigürasyonları barındırır.

- ▶ **package.json**

NPM ile indirilen paketleri tutar.

- ▶ **dependencies**

Uygulama için lazım olan, arayüze alakalı paketlerdir.

- ▶ **devDependencies**

Biz geliştiriciler için lazım olan, client ve arayüz ile pekte alakası olmayan paketlerdir.

- ▶ **tsconfig.json**

TypeScript compiler konfigürasyonunu tutar.

# Directives

Angular'da directive'ler DOM nesnelerini manipule etmek için kullanılır. Angular Directive'lerini kullanarak bir DOM nesnesinin görünümü, davranışını veya layout'unu değiştirebilmemiz mümkün. Ayrıca HTML'i genişletmemize de yardımcı olur.

Directive 'ler Angular projesi içinde @Directive decoratore yardımıyla oluşturulan javascript class'larıdır.

Directive 'ler nasıl davradıklarına göre 3 başlıkta sınıflandırılır:

- ▶ Component Directives
- ▶ Structural Directives
- ▶ Attribute Directives

# Component Directives

Angular Component'ler özelleşmiş directive 'lerdir. Directive 'lerin template'li olanlarıdır da diyebiliriz. Component Directive 'ler ana sınıfı oluşturur. Component'in çalışma süresinde nasıl başlatılıp işlenmesi ve nasıl kullanılması gerektiğine yön verir.

# Structure Directives

Structure Directive 'ler HTML nesnesini ve DOM yapısını fizksel olarak manipüle etmek ve değiştirmek için kullanılır. "\*" işaretini ile başlarlar. **Template'in yapsını değiştirirler.**

- Örneğin: \*ngIf directive, \*ngSwitch directive, and \*ngFor directive.
- **\*ngIf Directive:** DOM elementlerini eklemeye silmeye izin verir.
- **\*ngSwitch Directive:** Bu directive de DOM elementlerini eklemeye silmeye izin verir. C# ve benzeri dillerdeki switch kullanımına benzer.

```
<div [ngSwitch]="sayi">
  <div *ngSwitchCase="1">sayı 1</div>
  <div *ngSwitchCase="2">sayı 2</div>
  <div *ngSwitchDefault>hiçbiri</div>
</div>
```

- **\*ngFor Directive:** \*ngFor directive, bir list için HTML Üzerinde yinelenebilir html parçaları eklemeye yarar.
  - index, first, last, even, odd

```
<ul>
  <li *ngFor="let name of names; let index = index;
    let first = first;">{{name}} - {{index}} - {{first}}</li>
</ul>
```

# Attribute Directives

Attribute Directive 'ler DOM nesnesinin görünümünü ve davranışını değiştirmek için kullanılır.

- ▶ Örneğin: ngClass, ngStyle, ngModel
- ▶ **ngClass Directive:** ngClass directive, Html elementlerine css class'ları eklemek silmek için kullanılır.
- ▶ **ngStyle Directive:** ngStyle directive, Html elementlerine style eklemeye, style'larını dinamik değiştirmeye yarar.
- ▶ **ngModel Directive :** Bir veri özelliğini görüntülemek ve kullanıcı değişiklik yaptığında bu özelliği güncellemek için yönergeyi kullanılır. ([FormsModule](#) appModule dosyasına import edilmesi gereklidir)

# Custom Directive

- Selector [] ile kullanıyorsa attribute olarak
- Selector . ile kullanılıyorsa class olarak
- Parametre tanımlamak için @Input field tanımlamak gereklidir.
- @HostListener : oluşturulan directive'in hangi event ile görevlendirileceğini belirlediğimiz bir decoratordur.
- @HostBinding :Directive'in işratlendiği DOM nesnesinin bir özelliğine bind olarak işlemler gerçekleştirilebiliyoruz.

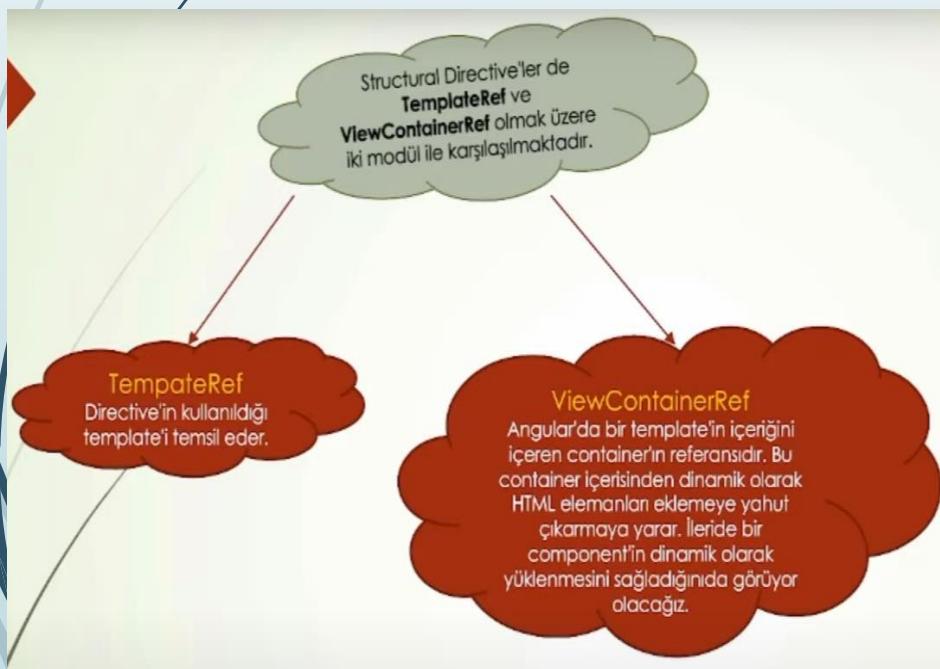
```
@HostBinding("style.color")
writingColor:string = "blue";
}
```

```
//Directive Oluşturma
//selector'ı Attribute Olarak Kullanma
//selector'ı class Olarak Kullanma
//Directive'e Parametre Tanımlama
//HostListener
//HostBinding
```

```
export class ExampleDirective {
  constructor(private element: ElementRef) {
    element.nativeElement.style.backgroundColor="red";
    $(element.nativeElement).fadeOut(2000).fadeIn();
  }
}
```

# Custom Structure Directive

- Selector ile setter property ismi aynı olmak zorundadır.



```
@Directive({
  selector: '[appCustomif]'
})
export class CustomifDirective {

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input() set appCustomif(value: boolean) {
    if (value) {
      this.viewContainerRef.createEmbeddedView(this.templateRef)
    } else {
      this.viewContainerRef.clear();
    }
  }
}
```

# Custom Structure Directive



Eğer ki, directive'i kullanırken değerleri otomatik olarak karşılamak isterseniz \$implicit operatörünü kullanabilirsiniz.

```
this.viewContainerRef.createEmbeddedView(this.templateRef, {  
  $implicit: `$_i`  
});
```

```
<ul>  
  <li *appLoop="2;let i">{{i}}</li>  
</ul>
```

```
@Directive({  
  selector: '[appCustomfor]'  
})  
export class CustomforDirective {  
  
  constructor(  
    private templateRef: TemplateRef<any>,  
    private viewContainerRef: ViewContainerRef  
  ) {}  
  
  @Input() set appCustomfor(value: number) {  
    for (let i = 0; i < value; i++)  
      this.viewContainerRef.createEmbeddedView(this.templateRef,{  
        index : i  
      })  
  }  
}
```

# Pipe

- ▶ “Angular Pipe” özelliğini, kısa ve basitçe, **filtreleme** veya **dönüştürme** olarak tanımlayabiliriz. Bu **Pipe’lama** işlemi, elinizdeki bir değeri veya veriyi **Pipe’lara** gönderip, bu değerlerin işlenip, kendi içindeki dönüşümlere göre yeni bir değer üretmesi olarak tanımlanabilir.

# Pipe

## CurrencyPipe

```
 {{1000 | currency}}
```

Sayısal değerleri  
parasal formata  
dönüştürür.

Default Para Birimi : { provide: DEFAULT\_CURRENCY\_CODE, useValue: 'TL' }

```
 {{1000 | currency :'₺'}}
```

## DatePipe

```
 {{'09.05.1992' | date}} Sep 5, 1992  
 {{'09.05.1992' | date : 'fullDate'}} Saturday, September 5, 1992  
 {{'09.05.1992' | date : 'medium'}} Sep 5, 1992, 12:00:00 AM  
 {{'09.05.1992' | date : 'shortTime'}} 12:00 AM  
 {{'09.05.1992' | date : 'mm:ss'}} 00:00
```

Tarihsel verileri  
biçimlendirir.

## SlicePipe

```
 [[[s, 'u', 'a', 'y', 'i', 'p'] | slice : 2 : 5]] a.y.i
```

Dizilerdeki verilere  
belirli aralıklarla  
ulaşılmasını sağlar.

## JsonPipe

```
 {{ {name:'gençay', surname:'yıldız'} | json}}
```

Bir nesneyi JSON  
formatına  
dönüştürür.

```
 { "name": "gençay", "surname": "yıldız" }
```

## UpperCasePipe

```
 {{'gençay yıldız' | uppercase}}  
 {{'GENÇAY YILDIZ' | lowercase}}
```

Metinsel değerlerin  
tüm harflerini  
büyük harfe  
dönüştürür.

GENÇAY YILDIZ  
gençay yıldız

## LowerCasePipe

Metinsel değerlerin  
tüm harflerini  
küçük harfe  
dönüştürür.

## PercentPipe

```
 {{50 | percent}}
```

Sayısal değerleri  
yüzdelik olarak  
formatlandırr.

## TitleCasePipe

```
 {{'adam sandık fos çıktı' | titlecase}}
```

Adam Sandık Fos Çıktı

Her kelimenin ilk  
harfini büyütür.

## KeyValuePipe

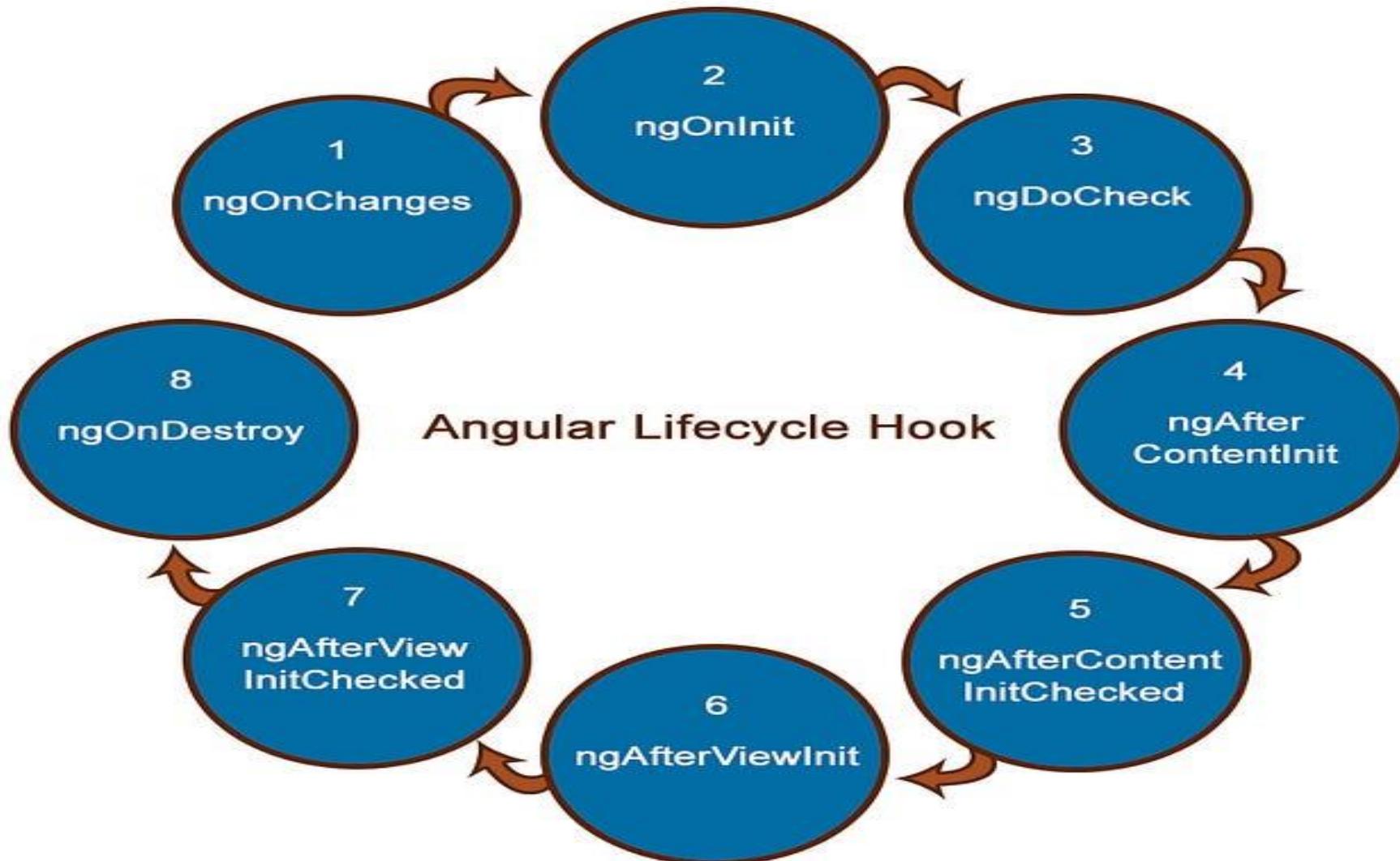
```
 export class AppComponent {  
   names: Map<number, string> = new Map([[1, 'ahmet'], [2, 'hilmi'], [3, 'şuayip']])  
 }  
<ul>  
 <li *ngFor="let name of names | keyvalue">{{name.key}}-{{name.value}}</li>  
</ul>
```

- 1-ahmet
- 2-hilmi
- 3-şuayip

# @Input(), @Output()

- ▶ **@Input() Decorator kullanımı :** Bu decorator'ü parent-componentten child-component'e veri göndermek için kullanırız. Yani ana componentten alt komponente veri göndermek için kullanırız.
- ▶ **@Output Decorator kullanımı :** Burada ise amacımız child-componentten parent componente veri göndermek.

# Component Lifecycle



# Component Lifecycle

- ▶ **ngOnChanges()** : Bu callback fonksiyonu veriye bağlı bir değişiklik olduğunda çağırılır. Bu olayın denetimi component kendi içinde yenilenirken gerçekleşir.

Bu hook sayesinde child bir component ile iletişim için kullanılır. Örneğin bir input değişikliğinde çağrırlabilen. Yaşam döngüsünde ilk çalıştırılan hook'tur

Not: Her değişiklikle beraber çağrılmak üzere performansı olumsuz etkileyeceğini unutmayın!

## Özellikleri

- ▶ Input değişkeni olan her componentte kullanılır.
- ▶ Input değeri her değiştiğinde çağrırlar.
- ▶ **ngOnInit()** : Angular component oluşumunu tamamladıktan sonra (constructordan hemen sonra) bu callback fonksiyonunu çağırır. Dataya bağlı özellikler görüntülenirken başlatılır.

### Özellikler

- ▶ Componentin verilerini çekerken kullanılır.
- ▶ Input verileri set edildikten sonra çağrırlar.
- ▶ Default olarak her componentte Angular CLI tarafından eklenir.
- ▶ Sadece bir kez çağrırlar.

# Component Lifecycle

- ▶ **ngDoCheck()** : Genel olarak bir componentin veya bir directive'in gözden geçirmenin önemli olduğu durumlarda kullanılır. Bu hook aracılığıyla uygulamak istediğiniz logic kontrolleri sağlayabilirsiniz.

NgOnInit'ten hemen sonra çağırılır ve componentte herhangi bir değişiklik olmasa bile işlenir. Herhangi bir input değişikliğini tespit etmek, veriyi logic kontrolünden sonra getirmek istendiğinde kullanılır.

## Özellik

- ▶ Herhangi bir değişikliği tespit etmek istendiğinde çağırılır.

- ▶ **ngAfterContentInit()** : Html içeriğini componentin dışından içeri aktarmak için kullanılır. Child componente contenti aktarılmak istendiğinde kullanılır. Harici child componentler “<ng-content></ng-content>” içerisinde kullanılır.

## Özellikler

- ▶ NgDoCheckten sonra çağırılır.
- ▶ Contenti initialize eder.

- ▶ **ngAfterContentChecked()** : Componente veya directive'e aktarılan contenti kontrol etmek için kullanılır. component içerisinde bir değişiklik kontrolünü sağlar. Componentte herhangi bir değişiklik olmasa da çalışır.

## Özellikler

- ▶ NgAfterContentInit'ten sonra çağırılır.

# Component Lifecycle

- ▶ **ngAfterViewInit()** : NgAfterContentChecked'den hemen sonra çağırılır.NgAfterContentInit'te .ok benzer ancak view yüklenikten sonra çağırılır.

## Özellikler

- ▶ Bir kez çağırılır.
- ▶ Sadece componentlerde kullanılır.

- ▶ **ngAfterViewChecked()** : Bu hook componentin ve child componentin view'i kontrol edilirken çağırılır.

Döngüde hemen NgAfterViewInit'in arkasında gelir.

- ▶ **ngOnDestroy()** : Componentler DOM'dan kaldırılmadan hemen önce çağırılır.

# Template Form & Reactive Form

- ▶ Veri Modeli :
- ▶ Doğrulama :
- ▶ Kontrolün Bağımsızlığı :
- ▶ Test Edilebilirlilik :
- ▶ Karmışıklılık :

# Angular Formlar

## Template-Driven Forms Yaklaşımı Nedir?

- ▶ Angular Forms yapısını oluşturmanın kolay yoludur.
- ▶ Form elemanlarının her birini 'ngModel' direktifi ile işaretleyerek çalışma sergilemektedir.
- ▶ Formun tasarımını ve yapılandırmasını tamamen template üzerinden gerçekleştirir.

## Model- Driven/Reactive Forms Yaklaşımı Nedir?

- ▶ Template-Driven Form'lara nazaran daha karmaşık form işlemleri için uygun olan yaklaşımdır.
- ▶ Bu yaklaşımın da formun temel mantığı component üzerinde bir nesne/object olarak tanımlanır ve bu nesne HTML'de ki ilgili form etiketlerine bind edilir.
- ▶ Model-Driven Form'lar bir yandan da Reactive Forms olarak adlandırılmaktadır. Çünkü, form elemanları ve bu form elemanlarındaki verilerin değişiklik durumları ilgili nesne tarafından reaktif bir şekilde(dinamik) olarak takip edilmektedir.
- ▶ Kodlama açısından Template-Driven Forms'a nazaran daha çok detay gerektireceği için bir nebze daha zorlu bir öğrenme eğrisi mevcuttur.

# Angular Formlar

## Angular Form Konseptleri

Bir Angular Form yapısında, Template-Driven Form yahut Model-Driven Form yaklaşımlarından hangisinin kullanılmış kullanılmadığına bakılmaksızın üç ana yapıtaşı mevcuttur.

Bir form içerisinde bulunan bir grup kontrolü temsil eder. Bir başka deyişle formun kendisini temsil eder. İçerisinde FormControl nesneleri barındır.

**FormGroup**

Form içerisinde dinamik olarak oluşturulan kontrole temsil eden dizisel nesnedir.

**FormArray**

Form içerisinde bulunan, kullanıcıdan veri almamızı sağlayacak olan tek bir kontrolü temsil eder.

**FormControl**

FormGroup, FormControl ve FormArray nesnelerini oluşturmamızı kolaylaşdıracak olan bir servistir. İçerisindeki hazır fonksiyonlar sayesinde formu hızla üretebilmemizi ve konfigür etmemizi sağlar.

**FormBuilder**

# Template-Driven Form

## Template-Driven Forms Yaklaşımında Kullanılan Temel Directive'ler Nelerdir?

ngForm

Formun kurulmasını sağlayan temel directive'dır.

ngModel

Form içerisinde kullanılacak kontrolleri işaretleyecek directive'dır.

Ayrıca ngModel directive'i Two Way Databinding mekanizmasını kullanarak işlem gerçekleştirir.

Bu directive'ler sayesinde Template-Driven Forms yaklaşımıyla geliştirilen formlar hızlı ve basit bir şekilde inşa edilebilirler.

# Template-Driven Form

## Template-Driven Forms Yaklaşımıyla Form Oluşturma Detayları

- Template-Driven Forms yaklaşımıyla form oluşturmak için önceki gerekli form directive'leri ve yapıları barındıran **FormsModule** isimli modülü uygulamada ilgili modüle import ediniz.
- Ardından hangi form'u Template-Driven Forms olarak kullanacaksanız onu **ngForm** directive'ı ile işaretleyiniz. Bu directive ile form nesnesi otomatik olarak algılanıp, bağlanılmaktadır. Dolayısıyla ilgili directive'i ekstradan çağırmak yahut bağlanmak için herhangi bir işlem yapmanız gereklidir.
- Form içerisindeki kullanılacak kontrolleri **ngModel** directive'ı ile işaretleyiniz.
- Ayrıca bilmenizde fayda var ki **ngForm** directive'ı, form nesnesini kendine bağlayacağı gibi bir yandan da arkaplanda bir FormGroup nesnesi oluşturarak bunu temsil edecektr.
- Son olarak form doldurulduğu taktirde kullanıcı tarafından component class'a verilerin gönderilmesi için **ngSubmit** event'inden istifade edebilirsiniz.
- İlgili event'e verilen fonksiyondan form'da ki verileri görseldeki gibi alabilirsiniz.

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

@Component({
  selector: 'app-root',
  template: `
    <form #frm="ngForm" (ngSubmit)="onSubmit(frm.value)">
      <input type="text" name="name" placeholder="Name" ngModel>
      <br>
      <input type="text" name="surname" placeholder="Surname" ngModel>
      <br>
      <select name="job" ngModel>
        <option value="0">Teacher</option>
        <option value="1">Student</option>
        <option value="2">Software Developer</option>
        <option value="3">Chef</option>
      </select>

      <button>Send</button>
    </form>
  `

})
export class AppComponent implements OnInit {
  @ViewChild("frm", { static: true }) frm: NgForm;

  ngOnInit(): void {
    console.log(this.frm);
    console.log(this.frm.form);
    console.log(this.frm.controls);
  }

  onSubmit(data: { name: string, surname: string, job: number, tels: [] }): void {
    console.log(`${data.name} ${data.surname} - ${data.job} - tels : ${data.tels}`);
  }
}
```

# Template-Driven Form

## ngForm Directive Detayları

<b>value</b>	<b>valid</b>	<b>touched</b>	<b>Submitted</b>
FormGroup içerisindeki her FormControl nesnesinin değerini döndürür.	Form geçerli ise true, değilse false değerini döndürür.	Kullanıcı form üzerinde en az bir alana değer girdiyse true değerini döner.	Form submit edildiği taktirde true değerini döndürür.

## FormControl Detayları

<b>value</b>	<b>valid</b>	<b>invalid</b>	<b>touched</b>
İlgili kontrolün değerini döndürür.	Kontrolün değeri geçerli ise true, değilse false değerini döndürür.	Değer geçersizse true, değilse false değerini döndürür.	Öğeye herhangi bir değer girildiği taktirde true döndürür.

```
<input type="text" name="firstname" #fname="ngModel" ngModel>
```

# Template-Driven Form

```
<form #frm="ngForm" (ngSubmit)="onSubmit(frm.value)">
  <input type="text" name="name" placeholder="Name" ngModel> <br>
  <input type="text" name="surname" placeholder="Surname" ngModel> <br>
  <input type="email" name="email" placeholder="Email" ngModel> <br>
  <input type="tel" name="tel" placeholder="Tel"> <br>
  <div ngModelGroup="address"> ←
    <input type="text" name="country" placeholder="Country" ngModel> <br>
    <input type="text" name="city" placeholder="City" ngModel> <br>
    <input type="text" name="address" placeholder="Address" ngModel>
  </div>
  <button>Send</button>
</form>
```

```
▶ {name: "Gencay", surname: "Yildiz", email: "gyildizmail@gmail.com", address: (...)} ●
  ↳ address:
    ↳ address: "Yeminihalle"
    ↳ city: "Ankara"
    ↳ country: "Turkiye"
  ↳ [[Prototype]]: Object
  ↳ email: "gyildizmail@gmail.com"
  ↳ name: "Gencay"
  ↳ surname: "Yildiz"
  ↳ [[Prototype]]: Object
```

```
export class AppComponent implements OnInit {
  @ViewChild("frm", { static: true }) frm: NgForm;

  ngOnInit(): void {
    console.log(this.frm);
    console.log(this.frm.form);
    console.log(this.frm.controls);
  }
}
```

# Template-Driven Form

## Component Class'ında Formu Temsil Etme ve Üzerinde İşlemler Gerçekleştirme

```
@ViewChild("frm", { static: true }) frm: NgForm;
```

► Form'u component class'ında referans etmek için ViewChild ile yukarıdaki gibi temsil etmek yeterlidir.

### Form Kontrollerine İlk Değer Atama

- Form kontrollerine direkt olarak setValue fonksiyonu aracılığıyla aşağıdaki gibi ilk değerleri atanabilmektedir.

```
ngOnInit(): void {
  setTimeout(() => {
    this.frm.setValue({
      name: {
        first: "Gençay",
        middle: "Muhammed",
        last: "Kürşad"
      },
      surname: "Yıldız",
      job: "Software Developer"
    });
  }, 200);
}
```

Burada setTimeout fonksiyonunun kullanılma sebebi OnInit even'i fırlatıldığında form yapılanmasının henüz initialize edilmemesidir.

```
ngOnInit(): void {
  setTimeout(() => {
    this.frm.controls["surname"].setValue("Yıldız");
  }, 200);
}
```

Ya da istenilen form kontrolüne de yukarıdaki gibi direkt değer atanabilir.

### Form'un Bir Kısımına Değer Atama

- Form'da ki bir kaç alan değiştirilmek, ilk değerleri atamak istediği durumlarda aşağıdaki gibi patchValue fonksiyonu kullanılabilmektedir.

```
this.frm.control.patchValue({
  surname: "Yıldız",
  job: "2"
})
```

### Form Değerlerini Sıfırlama

- Form değerleri reset, resetForm veya onReset fonksiyonlarıyla sıfırlanabilir.

```
this.frm.reset();
this.frm.resetForm();
this.frm.onReset();
```

# Model-Driven Forms Yaklaşımı Nedir?

Model-Driven Forms yaklaşımının bir diğer adı Reactive Form'dur.

Component Class'ında formun yapısının nesnel olarak tanımlandığı form yapısıdır.

- ▶ Yani form yapısında kullanılan FormGroup, FormArray ve FormControl gibi tüm nesneleri kendimizin oluşturduğu ve formu konfigüre ettiği bir form yapılanmasıdır.
- ▶ Tüm bunların yanında formun validasyonel kurallarıyla birlikte türlü yapılandırmalarını da bu nesne üzerinden tanımlayabiliyoruz.
- ▶ Template-Driven Forms yaklaşımına nazaran işlemlerin büyük kısmı yoğun olarak component class'ı üzerinde gerçekleştirilmektedir.

# Model-Driven Form (Reactive Form)

formGroup

Formun kurulmasını sağlayan temel directive'dir. Form elemanlarının component class'ındaki model ile senkronize olmasını sağlar.

Bunun için parametre olarak component class'ında tanımlanmış olan FormGroup nesnesini alarak çalışma sergiler.

formControlName

FormGroup nesnesi içerisindeki herhangi bir FormControl'ü form elemanlarından birine bağlamak ve senkronizasyonu sağlamak için kullanılan directive'dir.

İlgili FormControl nesnesine bağlanan form elemanın değeri değişikçe ilgili nesnenin de değeri otomatik olarak güncellenecektir.

```
<form [formGroup]="frm" (ngSubmit)="onSubmit()">
  <input type="text" placeholder="Name" formControlName="name"> <br>
  <input type="text" placeholder="Surname" formControlName="surname"> <br>
  <input type="text" placeholder="Email" formControlName="email"> <br>
  <input type="text" placeholder="Tel" formControlName="tel"> <br>
  <button>Send</button>
</form>
`

}

export class AppComponent {
  frm: FormGroup;
  constructor(private formBuilder: FormBuilder) {
    this.frm = formBuilder.group({
      name: [""],
      surname: [""],
      email: [""],
      tel: [""]
    })
  }
}
```

## formGroupName Directive'i

```
<form [formGroup]="frm" (ngSubmit)="onSubmit(frm.value)">
  <input type="text" placeholder="Name" formControlName="name"> <br>
  <input type="text" placeholder="Surname" formControlName="surname"> <br>
  <input type="text" placeholder="Email" formControlName="email"> <br>
  <input type="text" placeholder="Tel" formControlName="tel"> <br>
<div formGroupName="address">
  <input type="text" placeholder="Country" name="country" formControlName="country"> <br>
  <input type="text" placeholder="City" formControlName="city"> <br>
  <input type="text" placeholder="Address" formControlName="address"> <br>
</div>
<button>Send</button>
</form>
```

this.frm = formBuilder.group({
 name: ["Gençay"],
 surname: ["Yıldız"],
 email: ["gylidizmail@gmail.com"],
 tel: ["123"],
 address: formBuilder.group({
 country: [""],
 city: [""],
 address: [""]
 })
})

Object { name: "Gençay", surname: "Yıldız", email: "gylidizmail@gmail.com", tel: "123", address: (...) }  
address: Object { country: "Türkiye", city: "Ankara", address: "Yenimahalle" }  
email: "gylidizmail@gmail.com"  
name: "Gençay"  
surname: "Yıldız"  
tel: "123"

# Model-Driven Form (Reactive Form)

```
export class AppComponent {  
  frm: FormGroup;  
  constructor(private formBuilder: FormBuilder) {  
    this.frm = formBuilder.group({  
      name: [""],  
      surname: [""],  
      email: [""],  
      tel: [""],  
      address: formBuilder.group({  
        country: [""],  
        city: [""],  
        address: [""]  
      })  
    })  
  }  
}
```

```
<form [formGroup]="frm" (ngSubmit)="onSubmit()">  
  <input type="text" placeholder="Name" formControlName="name"> <br>  
  <input type="text" placeholder="Surname" formControlName="surname"> <br>  
  <input type="text" placeholder="Email" formControlName="email"> <br>  
  <input type="text" placeholder="Tel" formControlName="tel"> <br>  
  <div formGroupName="address">  
    <input type="text" placeholder="Country" formControlName="country"> <br>  
    <input type="text" placeholder="City" formControlName="city"> <br>  
    <input type="text" placeholder="Address" formControlName="address"> <br>  
  </div>  
  <button>Send</button>  
</form>
```

# Model-Driven Form (Reactive Form)

## Validation Tanımlama

```
this frm = formBuilder.group({  
  name: ["", Validators.required],  
  surname: ["", [Validators.required]],  
  email: ["", [  
    Validators.required,  
    Validators.email  
  ]],  
  tel: ["", [  
    Validators.required, Validators.max(100)  
  ]]  
});
```

Göründüğü üzere name alanı required olarak ayarlanmıştır. Yani zorunluluk arz eden bir valdasyona sahiptir. Dolayısıyla bu alan üzerinde yapılacak herhangi bir güncelleme durumu tüm formun kontrol edilmesini sağlayacak ve formun valid özelliğini etkileyecektir.

```
this frm = formBuilder.group({  
  name: ["", Validators.required],  
  surname: [""],  
  email: [""]
```

## onlySelf Özelliği

```
this frm.controls["name"].setValue("Gençay");
```

Şimdi bu şartlarda yukarıdaki kodu tetiklersek eğer tüm form geçerliliği kontrol edilecek, dolayısıyla valid özelliğinin değeri değişecektir.

```
this frm.controls["name"].setValue("Gençay", { onlySelf: true });
```

Angular mimarisinde varsayılan olarak herhangi bir form ögesinin değerinde bir güncelleme olduğunda form yapılanması validasyonel durumların hepsini basamaklı olarak en üst düzeye kadar kontrol edecektir. Bunu engellemek için onlySelf özelliğini kullanabilir ve herhangi bir form kontrolde olacak değişiklikte komple formun değil, sadece ilgili kontrolün sorumluluğu olduğunu ifade edebilirsiniz.

# Model-Driven Form (Reactive Form)

Formdaki kontrolden  
birinin değeri  
değiştiğinde fırlatılır.

Formun geçerlilik  
durumu değiştiğinde  
fırlatılır.

## valueChanges & statusChanges

```
this.frm.valueChanges.subscribe({  
  next: data =>  
    console.log(data);  
})  
  
this.frm.statusChanges.subscribe({  
  next: data =>  
    console.log(data);  
})
```

INVALID

```
Object { name: "G", surname: "Yıldız", email: "", tel: "" }  
  email: ""  
  name: "G"  
  surname: "Yıldız"  
  tel: ""  
  > <prototype>: Object { ... }
```

VALID

```
Object { name: "Ge", surname: "Yıldız", email: "", tel: "" }  
  email: ""  
  name: "Ge"  
  surname: "Yıldız"  
  tel: ""  
  > <prototype>: Object { ... }
```

```
this.frm.get("name").valueChanges.subscribe({  
  next: data =>  
    console.log(data);  
})  
  
this.frm.get("name").statusChanges.subscribe({  
  next: data =>  
    console.log(data);  
})
```

# Built-in Validators

```
constructor(private formBuilder: FormBuilder) {  
    this.frm = formBuilder.group({  
        name: ['', Validators.required],  
        surname: ['', Validators.minLength(2)],  
        email: ['', [Validators.required, Validators.email]],  
        address: formBuilder.group({  
            country: ['', Validators.required],  
            city: ['', Validators.minLength(2)],  
            address: ['', Validators.maxLength(100)]  
        })  
    });  
}
```

- Angular'da ReactiveFormsModule modülü içerisinde kullanıma hazır birkaç built-in validators bulunmaktadır. Bunlar:
  - min : Değerin sağlanan sayıdan büyük veya eşit olmasını gerektiren validator.
  - max : Değerin sağlanan sayıdan küçük veya eşit olmasını gerektiren validator.
  - required : Kontrolün değerinin girilmesini zorunlu kıyan validator.
  - requiredTrue : Kontrolün değerinin true olmasını gerektiren validator. Genellikle checkbox'lar için kullanılır.
  - email : Değerin email formatına uygun olmasını gerektiren validator.
  - minlength & maxlength : Metinsel değerlerde karakter sınırını belirleyen validator'lar.

# Custom Validator Oluşturma ve Kullanma

Angular'da built-in olarak gelen validator'ların dışında özelleştirilmiş validator'lar da oluşturabiliriz.

Bunun için ValidatorFn interface'inden istifade edebiliriz. Bu interface, içerisindeki imza sayesinde bizlere validator oluşturma kurallarını direkt sunmaktadır.

```
import { AbstractControl, ValidationErrors } from "@angular/forms";

export function capitalLetterValidator(control: AbstractControl): ValidationErrors | null {
  const value = control.value;

  const ascii: string[] = [];
  for (let n = 65; n <= 90; n++)
    ascii.push(String.fromCharCode(n));

  if (ascii.indexOf(value[0]) == -1) {
    return { capitalLetter: true }
  }

  return null;
}
```

```
export declare interface ValidatorFn {
  (control: AbstractControl): ValidationErrors | null;
}
```

Gördüğü üzere bir validator tanımlayabilmek için geriye ValidationErrors nesneyi ya da null değer dönen ve parametre olarak AbstractControl türünden değer alan bir fonksiyon tanımlamamız yeterliyim...

# Async Validator

Şu ana kadar gördüğümüz tüm validator'lar, doğrulanmakta ve ardından sonuç döndürmekteydi, yani sync validator davranışını sergilemektedir.

Şimdide async validator'ların nasıl oluşturulduğunu inceleyelim...

Bunun için **AsyncValidatorFn** interface'sinden istifade edeceğiz.

```
export declare interface AsyncValidatorFn {  
  (control: AbstractControl): Promise<ValidationErrors | null> | Observable<ValidationErrors | null>;  
}
```

Görüldüğü üzere async validator tanımlayabilmek için geriye Promise yahut Observable dönen ve AbstractControl türünden parametre alan bir fonksiyon tanımlanması gerekmektedir.

Genellikle bu validator yapısı, doğrulama için gerekli olan verilerin dış bir servisten alındığı uzun soluklu süreçler için kullanılmaktadır.

Bunun için aşağıdaki örneği inceleyiniz...

```
import { AbstractControl, ValidationErrors, AsyncValidatorFn } from '@angular/forms';  
import { concatMap, map, Observable, of, timer } from "rxjs";  
  
export function validateAsyncValidator(): AsyncValidatorFn {  
  return (control: AbstractControl): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> => {  
    const source = of(true);  
    return timer(5000)  
      .pipe(  
        concatMap(() => source),  
        map(() => {  
          console.log("X");  
          if (false)  
            return null;  
          else  
            return { text: "..." };  
        })  
  };  
}
```

# Karşılaştırma Validatörleri

```
export function matchPassword(): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const password: string = control.get("password").value;
    const passwordConfirm: string = control.get("passwordConfirm").value;

    if (password != passwordConfirm)
      return { "noMatch": true };
    return null;
  };
}
```

```
constructor(private formBuilder: FormBuilder) {
  this.frm = formBuilder.group({
    password: ["", Validators.required],
    passwordConfirm: ["", Validators.required]
  }, { validators: [matchPassword()] });
}
```

```
export function matchControl(firstControl: string, secondControl: string): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const control1: string = control.get(firstControl).value;
    const control2: string = control.get(secondControl).value;

    if (control1 != control2)
      return { "noMatch": true };
    return null;
  };
}

this.frm.get("password").updateValueAndValidity();
this.frm.updateValueAndValidity();
```

# Dependency Injection

- ▶ **@Injectable** : Servisleri işaretleyerek bağımlılıklarını incet etmesini sağlıyoruz. Ve bu sınıfın servis olduğunu anlamamızı sağlar.
- ▶ **@Injectable providedIn Parametresi :**
  - ▶ **root** : tüm program için tek bir instance oluşturur (singalten)
  - ▶ **any** : her çağrııldığından yeni bir instance oluşturur
  - ▶ **platform** : Servislerin platforma özgü olduğunu ifade eder.
  - ▶ Sadece ilgili modül için kullanılması isteniyorsa
- ▶ Servisleri iki türlü inject edebiliriz.
  - ▶ constructor
  - ▶ injector()

```
@Injectable({  
    providedIn: AppModule  
})
```

# Angular'da Dependency Injection Aktörleri

Angular'da Dependency Injection için **consumer**, **dependency**, **injection token(DI Token)**, **provider** ve **injector** olmak beş aktör mevcuttur.

## Consumer

Bağımlılığa ihtiyaç duyan sınıfır. (Component, directive, service.. vs)

## Dependency

Consumer'da olması istenilen servistir.

## Injection Token (DI Token)

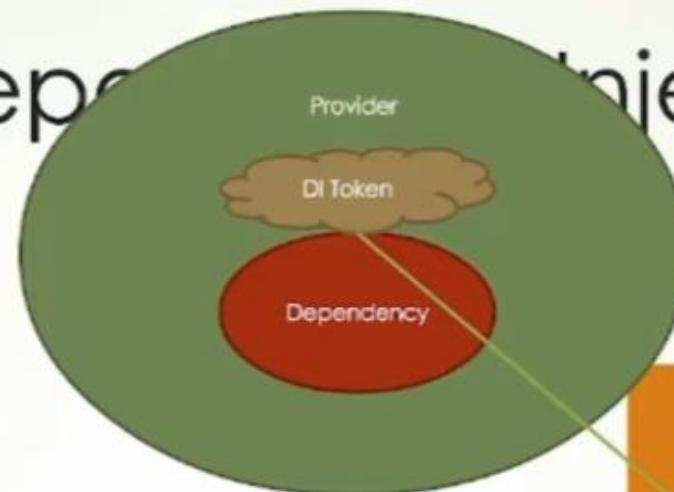
Dependency'leri unique bir şekilde tanımlayan değerdir.

## Provider

Injection Token eşliğinde, dependency'lerin tutulduğu yerdir.

## Injector

Dependency'leri ihtiyaç noktalarında inject etmemizi sağlayan yapılardır. Inject sürecinde DI Token'ı kullanırlar.



# Injector Mekanizması Detayları

Injector, dependency'leri somutlaştıran ve onları ihtiyaç noktalarına enjekte(inject) etmekten sorumlu olan yapılardır.

Injector'lar, provider'da DI Token değerleri ile ilişkilendirilmiş vaziyette depolanan dependency instance'larını DI Token'ları kullanarak aramamızı ve elde etmemizi sağlarlar.

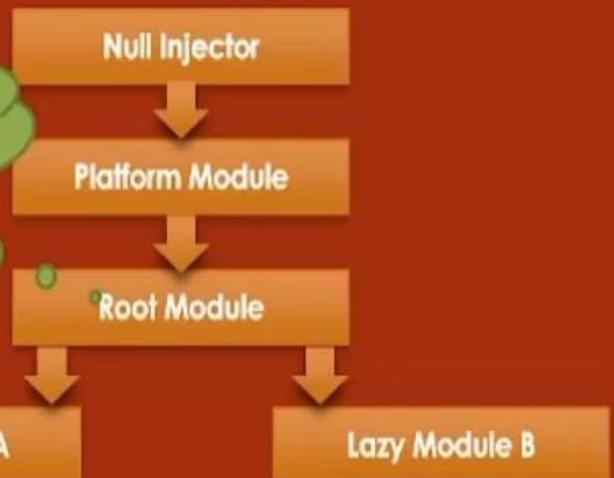
Angular, uygulama tarayıcıya yükleniği takdirde modüller ve componentler/directive'ler için olmak üzere iki injector ağacı oluşturmaktadır.

## Module Injector Tree

Modül seviyesinde provide edilecek servisler için oluşturulan injector ağıacıdır.

Modül seviyesinde `@NgModule` ve `@Injectable` dekoratörleri aracılığıyla iki farklı şekilde servisleri provide edebiliriz.

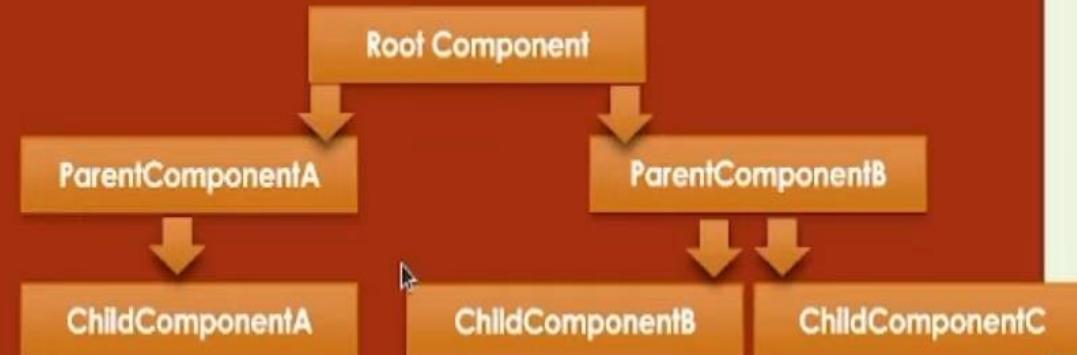
Modül ağacının hiyerarşisi aşağıdaki gibi şekillendirilir.



## Element Injector Tree

Component ve directive'ler seviyesinde provide edilecek servisler için oluşturulan injector ağıacıdır.

Element ağacının hiyerarşisi aşağıdaki gibi şekillendirilir.



# Bir Servisi Provide Etmenin Detayları

Uygulama sürecinde Dependency Injection mekanizmasını kullanabilmek için servisleri provide etmemiz gerektiğini öğrendik. Ve temel seviye de bu provide işleminin nasıl yapılabileceğine dair fikir edinmiş olduk.

Bir servisi provide ederken **type token**, **string token** ve **injection token** olmak üzere üç tür token türünden istifade edebiliriz.

Şimdi provide sürecindeki detayları inceliyor olacağız.

Angular'da bir servisi provide ederken o servise erişim gösterebileceğimiz DI Token ile ilişkilendirerek bu işlemi gerçekleştiririz.

DI Token'a, ilgili servis'in ihtiyaç noktasındaki instance'ını çağırımızı sağlayan aparatır diyebiliriz.

Ayrıca provide sürecinde ilgili dependency'nin nasıl oluşturulacağına dair de konfigürasyonda bulunabiliriz.

Bu konfigürasyonlar da; **class provider(useClass)**, **value provider(useValue)**, **factory provider(useFactory)** ve **aliased class provider(useExisting)** olmak üzere dört tanedir.

```
providers: [{ provide: ProductService, useClass: ProductService }]
```

Şimdi bir servisi provide etmek için ilgili modülün yahut component'in 'providers' özelliğini aşağıdaki gibi kullanabiliriz.

```
providers: [ProductService]
```

Şimdi de DI Token'ın türlerini inceleyelim...

#### Type Token

Herhangi bir ürün token olarak kullanılmasıdır.

```
providers: [{ provide: ProductService, useClass: ProductService }]  
constructor(private productService: ProductService)
```

#### String Token

Herhangi bir metinsel değerin token olarak kullanılmasıdır.

```
providers: [{ provide: "ProductService", useClass: ProductService }]
```

Eğer ki string token kullanılıyorsa ilgili dependency'yi inject ederken aşağıdaki gibi @Inject decorator'ünden istifade etmek zorunludur.

```
constructor(@Inject("ProductService") private productService)
```

Tabii bu şekilde provide edilen değer bir nesneye illaki o nesneyi sanki type tokenmiş gibi parametre referansı üzerinden talep edebilirsiniz.

```
constructor(private productService: ProductService)
```

Evet, bu kullanım da hata vermeyecektir amma velakin string token'da kullanılan değerin türü yoksa yahut metinsel bir değerse o taktirde @Inject decorator'u zorunlidır.

Bunun yanında bir servisi provide ederken aşağıdaki gibi de bir davranış sergilenebilir.

```
providers: [{ provide: ProductService, useClass: ProductService }]
```

#### Provide

Provide edilecek olan servisi temsil edecek ve ihtiyaç dahilinde erişebilmemizi sağlayacak olan Token veya DI Token'dır.

Bu değer, örnekte olduğu gibi herhangi bir referans(Type Token) olabileceği gibi string yahut InjectionToken türünden de tanımlanabilir.

#### Provider/Recipe

Provide edilecek nesnenin tanımlandığı kısımdır. Angular'da ilgili dependency'nin instance'sının nasıl oluşturulacağına karannın verildiği yerdir. Örnekte olduğu gibi 'useClass' kullanılıyorsa eğer bu, provide edilen değerin bir servis olduğunu ifade etmektedir.

#### Injection Token

String token kullanılırken bazen farkında olmaksızın bir geliştiricinin oluşturduğu token'i farklı bir geliştirici tekrardan kullanabilir. Böyle bir durumda aynı isimde tanımlanan token öncekini ezecektir.

İşte bizler bu durumlara istinaden benzersiz token'lar oluşturmak isteyebiliriz. Bunun için injection token'dan istifade edebiliriz.

```
import { InjectionToken } from "@angular/core";  
  
export const productService = new InjectionToken<string>("");  
  
providers: [{ provide: productService, useClass: ProductService }]  
constructor(@Inject(productService)private productService)
```

```
1 import { InjectionToken } from "@angular/core";
2
3 export const productServiceIT : InjectionToken<any> = new InjectionToken("");
```

Dependency Injection

```
providers: [
    // ProductService //DI Token - Default Type Token
    // {provide:ProductService, useClass:ProductService} //Type Token
    // {provide:"productService", useClass:ProductService} //String Token
    {provide: productServiceIT, useClass: ProductService}
    // {provide: "example", useValue:"merhaba"}, //Value Token
    // {provide: "example", useValue: () => {
    //   return "merhaba";
    // }},
    {
        provide: "productService", useFactory: (httpClient: HttpClient) => {
            const obs = httpClient.get("https://jsonplaceholder.typicode.com/posts").
            subscribe({ next: data => console.log(data) });
            return new ProductService();
        },
        deps: [HttpClient]
    }
]
```

# Provider Türleri

## Class Provider useClass

Bir tür/sınıf provide edileceğe  
eğer kullanılan provider'dır.

```
providers: [
  { provide: productService,
    useClass: ProductService }
],
```

Bu şekilde kullanırken  
fonksiyonun komple  
inject edildiğini  
bilmenizde fayda  
vardır.

```
constructor(@Inject("URL") private urlFunction: any)
const url: string = urlFunction();
console.log(url);
```

Dikkat ederseniz useValue ile  
useFactory'de fonksiyon tanımlayarak  
provide işlemlerini gerçekleştiriyoruz. Bu  
durumda ikisinin arasındaki farka  
değinmemiz gerekirse eğer; useValue  
provide edilen fonksiyonu döndürürken,  
useFactory fonksiyon içerisinde return  
edilen result'ı döndürecekler.

## Value Provider useValue

Basit/metinsel değerleri provide  
etmek istediğimizde kullanılan  
provider'dır.

```
providers: [
  { provide: 'URL',
    useValue: 'https://www.ngakademi.com' }
],
```

Ayrıca value provider'da  
aşağıdaki gibi bir fonksiyonuda  
provide edebilirsiniz.

```
providers: [
  { provide: 'URL',
    useValue: () => {
      return "https://www.ngakademi.com";
    }
],
```

## Factory Provider useFactory

Provide edilecek servisi yapılandırdıken  
herhangi bir dış kaynaktan konfigürasyonel  
değer alınacağı yahut farklı bir ihtiyaçtan  
dolayı bir API'a bağlanılacaksa veya  
provide edilecek nesne belirsiz veya  
farklı davranışlar sergileyen vs. bu  
İhtiyaçlı useFactory ile gerçekleştirebiliriz.

```
providers: [
  { provide: LoggerService,
    useClass: LoggerService },
  { provide: "productType",
    useValue: "A" },
  { provide: AProductService,
    useFactory: (productType: "A" ? new AProductService(loggerService) : new BProductService(loggerService))
    , factory: () => {
      if (productType === "A")
        return new AProductService(loggerService);
      else
        return new BProductService(loggerService);
    }
],
```

## Aliased Class Provider useExisting

Provide edilmiş olan bir servisi  
farklı bir referansla temsil etmek  
istediğimiz durumlarda kullanılır.

```
providers: [
  { provide: B, useClass: A },
  { provide: C, useExisting: A }
],
```

İkinci provider için  
instance oluşturur...

İkinci provider için  
instance oluşturur...

c parametresine  
ilk provider'in  
instance'ı inject  
edilir.

# @Self - @SkipSelf - @Optional Decorator'ları

Bu decorator'ler ile DI mekanizmasının dependency'leri nasıl resolve etmeleri gerektiğini belirtebilmekte ve böylece injector'lann davranışlarını değiştirebilmektedir.

Injector'lann davranışlarını değiştirdikleri için bu decorator'lara biryandan da **Resolution Modifiers** adı verilmektedir.

Şimdi konuya girebilmek için aşağıdaki çalışmayı ele alalım.

```
injectable({  
providedIn: 'root'  
}  
.  
.  
.  
import class RandomService {  
private _random = 0;  
constructor() {  
console.log("RandomService c  
this._random = Math.floor(Ma  
}  
get random() {  
return this._random;  
}
```

Yandaki gibi random değer üreten bir servisimiz olduğunu düşünelim...

Ardından yandaki gibi birbirlerini referans eden component'ler de kullandığımızı varsayıyalım...

```
@Component({  
selector: 'app-root',  
template: `  
AppComponent -> {{randomService.random}} <br>  
<app-a></app-a>`,  
});  
export class AppComponent {  
constructor(public randomService: RandomService) {  
}
```

```
@Component({  
selector: 'app-a',  
template: `  
AComponent -> {{randomService.random}} <br>  
<app-b></app-b>`  
});  
export class AComponent {  
constructor(public randomService: RandomService)  
}  
}
```

# ViewProviders

viewProviders özelliği kullanıldığı component'in template'inde referans edilen tüm alt component ve directive'lere servis provide etmemizi sağlayan bir özelliktir.

```
@Component({
  selector: 'app-root',
  template: `
    <app-a></app-a>`,
  viewProviders: [RandomService]
})
export class AppComponent { }
```

# Angular Routing Nedir?

Angular mimarisinde bir view'den diğerine gidebilmek ya da bir component'ten diğerine geçiş yapabilmek için Angular Router modülünden istifade edebiliriz.

Evet, Angular Router bir modüldür ve `@angular/router` dizininde built-in olarak gelmektedir. Bu modül route üzerinden component'ler üzerinde gezinti yapabilmemizi ve bunun için gerekli olan tüm service provider'ları sağlamamaktadır.

Angular Router modülü ile yapılabilecek işlemler aşağıdaki gibidir;

Adres çubuğuna yazıla URL üzerinden belirli bir componente gitme.

Component'e query string değerlerini gönderme.

Browser'ın ileri ve geri düğmelerini aktifleştirme/kullanabilme.

Dinamik olarak view yükleme.

Rotalar üzerinde yetki kontrolü gerçekleştirmeye.

# Angular Router Bileşenleri

Angular Router  
yapılanmasının  
bileşenleri

Activated  
Route

Component class'ı  
Üzerinden etkin URL'i  
elde etmemizi  
sağlayan sınıfır. Bir  
başa deyişle o anki  
etkin route'u temsil  
eder.

RouterState

Route üzerindeki tüm  
bilgileri ve durumları  
iceren bir nesnedir.  
Uygulamanın hangi  
rotada olduğunu ve  
rotanın nasıl  
değiştigini takip  
etmek için kullanılır.

Router

Route

Routes

Angular'da tarayıcı  
Üzerinden geri/ileri  
düğmesini  
aktifleştiren ve  
component'ler  
arasında  
gezilmesini  
sağlayan servistir.

Angular'da  
component'lerin  
rotalarını  
belirlememizi  
sağlayan  
yapılanmadır.

Her bir route: bir path  
ve o path ile eşleşen  
bir component'ten  
oluşur.

Tüm route'ları  
bulunduğu bir dizidir.

Router  
Link  
Active

Tarayıcının adres  
çubuğundaki URL ile  
uyumlu olan route'un  
view'in hangi  
alanında  
gösterileceğini ifade  
eden yapıdır.

Router  
Outlet

RouterLink

HTML öğesi olan a  
tag'ını bir route'a  
bağlayan direktifdir.

RouterLink ile birlik  
kullanılan başka bir  
direktiftir.  
RouterLink'in  
kullanıldığı a tag'ine  
mevcut route  
durumuna dayalı  
olarak aktif bir CSS  
class'ı verir.

# Angular Router Bileşenleri

Angular Router  
yapılanmasının  
bileşenleri

Activated  
Route

Component class'ı  
Üzerinden etkin URL'i  
elde etmemizi  
sağlayan sınıfır. Bir  
başa deyişle o anki  
etkin route'u temsil  
eder.

RouterState

Route üzerindeki tüm  
bilgileri ve durumları  
iceren bir nesnedir.  
Uygulamanın hangi  
rotada olduğunu ve  
rotanın nasıl  
değiştigini takip  
etmek için kullanılır.

Router

Route

Routes

Angular'da tarayıcı  
Üzerinden geri/ileri  
düğmesini  
aktifleştiren ve  
component'ler  
arasında  
gezilmesini  
sağlayan servistir.

Her bir route: bir path  
ve o path ile eşleşen  
bir component'ten  
oluşur.

Tüm route'ları  
bulunduğu bir dizidir.

Router  
Link  
Active

Router  
Outlet

RouterLink

Tarayıcının adres  
çubuğundaki URL ile  
uyumlu olan route'un  
view'in hangi  
alanında  
gösterileceğini ifade  
eden yapıdır.

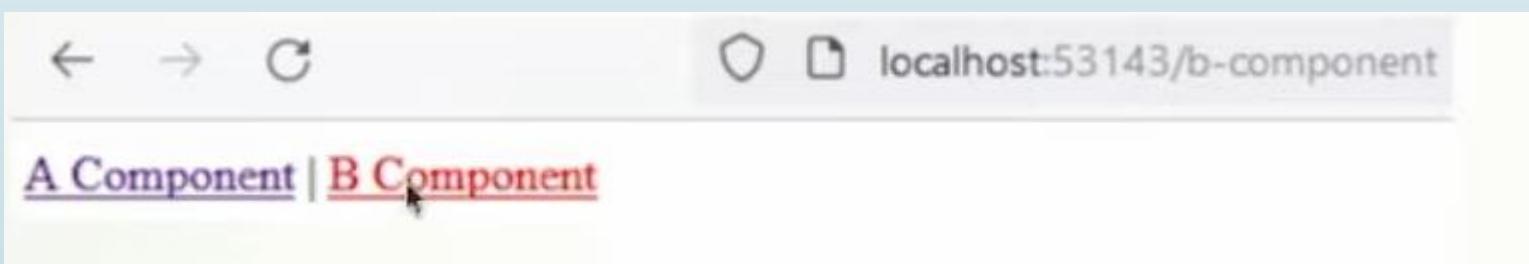
HTML öğesi olan a  
tag'ını bir route'a  
bağlayan direktifdir.

RouterLink ile birlik  
kullanılan başka bir  
direktiftir.  
RouterLink'in  
kullanıldığı a tag'ine  
mevcut route  
durumuna dayalı  
olarak aktif bir CSS  
class'ı verir.

# Angular Routing

- **routerLinkActive** : mevcut rota ile eşleşen link'e dinamik bir şekilde class uygulamamızı sağlayan bir özelliktir.

```
template:  
  <a routerLinkActive="active" routerLink="a-component">A Component</a>  
  <a routerLinkActive="active" routerLink="b-component">B Component</a>  
  <br>  
  <router-outlet></router-outlet>  
,  
styles: [".active{ color:red; }"]
```



# Varsayılan Rota Ayarlama

```
const routes: Routes = [
  { path: "", redirectTo: "/home", pathMatch: "full" },
  { path: "home", component: HomeComponent }]
```

Default route ayarlamak için yukarıdaki gibi boş bir path tanımında bulunmak ve bunu da istenilen rotaya yönlendirmek yeterlidir.

Burada; **pathMatch** alanı router'ın bu url'i nasıl eşleştireceğinin karannı verdirmektedir.

'full' ve 'prefix' olmak üzere iki değer alabilir; 'full', URL'in route tanımındaki yolun tamamıyla eşleşmesini gerektirirken, 'prefix' ise URL'in önekle eşleşmesini yeterli görmektedir.

Örneğin:

Route tanımı '/users' ise;  
'full' sadece '/users' URL'i ile eşleşecekken, 'p'  
'/users' veya '/users/1' gibi URL'lerle eşleşebi

# Wildcard Route

Angular uygulamasında tanımlanmış olan rotalardan herhangi biriyle eşleşmeyen bir URL olduğu takdirde Wildcard Route devreye girmektedir.

Wildcard Route, `**` karakteriyle temsil edilen bir yapıdır. Bu bir yandan da **Joker Karakterli Yol** olarak da tarif edilmektedir. Tarayıcıda URL, sistemde tanımlı olan diğer tüm rotalarla kontrol edildikten sonra hiçbirine eşleşmemesi durumunda Wildcard Route'a yönlendirme gerçekleştirilecektir.

```
const routes: Routes = [
  { path: "", redirectTo: "/home", pathMatch: "full" },
  { path: "home", component: HomeComponent },
  { path: "a-component", component: AComponent },
  { path: "b-component", component: BComponent },

  { path: "**", component: ErrorComponent },
];
```

Wildcard route, anlayacağınız üzere genellikle uygulamanızda sayfaları yönlendirmek veya tanımlanmamış bir URL'i karşılamak için kullanılmaktadır.

Burada dikkat edilmesi gereken husus şudur ki; Wildcard Route'u tanımlarken diğer rota tanımlarının en sonuna tanımlanmalıdır. Çünkü route yapılanması tanımlandıkları sırayla eşleştirilir ve router her zaman ilk eşleşen rotayı döndürecektr.

# Hash Style Routing

Hash Style Routing, bir web uygulamasındaki gezinti işlemlerinin URL'lerini değiştirmeden tek sayfa olarak kalmasına izin veren bir yönlendirme teknigidir.

Hash Style Routing, URL'de # karakterinden sonra gelen bir dize kullanmaktadır. Misal olarak;

localhost:53143/#/a-component  
localhost:53143/#/b-component

Angular uygulamasında url yönlendirmelerinde Hash Style Routing davranışını kullanmak istiyorsanız eğer AppRoutingModule dosyasında aşağıdaki konfigürasyonda bulunabilirsiniz.

```
@NgModule({  
    imports: [RouterModule.forRoot(routes, { enableTracing: true, useHash: true })],  
    exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

# HTML 5 Routing

HTML 5 Routing tarayıcıının geri  
özellikini dahi kullanmamızı sağlayan  
Client side routing yöntemidir.

history.pushState() fonksiyonu  
kullanılarak, programatik olarak  
yönetilebilir ve url konumunu  
değiştirebilirsiniz.

```
var stateObj = { message: "some message" };
history.pushState(stateObj, "title", "/a/b");
```

Şunu unutmamanız gerekmektedir ki,  
tüm tarayıcılar HTML 5 routing'i  
desteklememektedir.

# PathLocationStrategy

## ARTILARI

- http://example.com/foo gibi URL'ler üretir.
- Server-Side Rendering'i destekler.

## EKSİLERİ

- Eski tarayıcılarda desteklenmemektedir.
- Sunucu desteği gereklidir.

## PathLocationStrategy

Normal URL'leri kullanarak yönlendirme yapar.  
Özellikle modern tarayıcılarda tercih edilir.

Angular uygulamasının varsayılan stratejisidir.

Bu stratejinin konfigürasyonu index.html'de ki `<base href>` etiketini yapılandırarak başlamaktadır.

Tarayıcı, sayfada kullanılacak statik kaynaklar(resimler, css, javascript) için ilgili URL'leri oluşturmak üzere bu etikette belirtilen temel adresi kullanacaktır.

Tabii isterseniz `<base href>` etiketini yandaki gibi de 'APP\_BASE\_HREF' injection token'ı üzerinden de konfigüre edebiliyorsınız.

```
providers: [
  { provide: APP_BASE_HREF, useValue: "/my/base" }
],
bootstrap: [AppComponent]
}
export class AppModule { }
```

Bu konfigürasyondan sonra Angular mimarisinin temel routing yapılandırmasını gerçekleştirdiğinizde PathLocationStrategy davranışı devreye girecektir ve rotalar arasında URL tabanlı yönlendirme gerçekleştirilecektir.

# HashLocationStrategy

## ARTILARI

- Tüm tarayıcılar tarafından desteklenmektedir.

## EKSİLERİ

- <http://example.com/#foo> gibi kirli URL'ler üretir.
- Server-Side Rendering'i desteklemez.

## HashLocationStrategy

URL'leri yönetmek için # simbolünü kullanır. Özellikle eski tarayıcılarda, URL değişikliklerinin algılanması ve yönlendirmelerin yapılması için kullanılabilir.

HashLocationStrategy davranışı  
Üzerinden rotalarda yönlendirmeyi  
sağlayabilmek için yandaki gibi  
RouterModule üzerinden forRoot  
fonksiyonunun ikinci parametresindeki  
'useHash' field'ını kullanabilirsiniz.

```
@NgModule({
  imports: [RouterModule.forRoot(routes, { useHash: true })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Ya da isterseniz yandaki gibi  
LocationStrategy'e karşılık değeri  
değiştirebilir ve HashLocationStrategy  
davranışını uygulayabilirsiniz.

```
providers: [
  Location, { provide: LocationStrategy, useClass: HashLocationStrategy }
],
bootstrap: [AppComponent]

import class AppModule { }
```

# Route Parameter Nasıl Tanım

Bir route üzerinde route parameter ile veri taşımak istiyorsanız öncelikle verinin taşınacağı konumda bir parametre tanımlamasında bulunmanız gerekmektedir.

```
{ path: "products/:id", component: ProductComponent }
```

Gördüğü Üzere route üzerinde bir parametere tanımlamak bu kadar kolay.

Ve bir route'ta aşağıdaki gibi istediğiniz kadar parametre tanımlayabilirsiniz.

```
{ path: "products/:id1/:id2/:id3", component: ProductComponent }
```

Tabi route'a parametre tanımlandığı taktirde 'products' URL'ini sade bir şekilde kullanabilmek istiyorsanız onu da ayrıca tanımlamanız gerekmektedir.

<http://localhost:55556/products>  
<http://localhost:55556/products/2>  
<http://localhost:55556/products/2/3/4>

```
{ path: "products", component: ProductComponent },  
{ path: "products/:id", component: ProductComponent },  
{ path: "products/:id1/:id2/:id3", component: ProductComponent }
```

Tanımlanan bu rotalara karşılık a tag'ine karşılık bir link oluşturmak isterseniz yandaki gibi routerLink direktifini kullanabilirsiniz.

```
<a [routerLink]="/products", product.id]>{{product.name}}</a>
```

## ActivatedRoute Nesnesi İle URL'deki Parametre Değerlerini Okuma

”

ActivatedRoute, o anda aktif olan rotaya ilgili işlemler yapmamızı sağlayan bir nesnedir.

Bu nesne sayesinde url'deki route parametrelerini ve query string değerlerini elde edebilir ve amacımız doğrultusunda işleyebiliriz.

ActivatedRoute nesnesini yandaki gibi @angular/router path'inden import edebilir ve kullanabilirsiniz.

```
import { ActivatedRoute } from '@angular/router';  
constructor(private activatedRoute: ActivatedRoute) { }
```

### Route parametre değerlerini nasıl okuyabiliriz?

ActivatedRoute nesnesi, component'in etkinleştirildiği andaki yönlendirme durumunun bir görüntüsü olan **ActivatedRouteSnapshot** nesnesinden oluşur.

```
constructor(private activatedRoute: ActivatedRoute) {  
  (property) ActivatedRoute.snapshot: ActivatedRouteSnapshot  
  The current snapshot of this route  
}  
activatedRoute.snapshot
```

Haliyle bizler sayfaya yapılan yönlendirme neticesinde URL üzerinden gerekli parametre değerlerini okuyabilmek için bu neden istifade edeceğiz.

Şöyledi:

```
constructor(private activatedRoute: ActivatedRoute) {  
  const id: any = activatedRoute.snapshot.paramMap.get("id");  
  const hasId = activatedRoute.snapshot.paramMap.has("id");  
}
```

Gördüğü üzere snapshot üzerinden **paramMap** ile yandaki gibi URL'deki parametre değerlerini okuyabilmekteyiz.

Ya da isterseniz yandaki gibi **params** ile de bu değerleri okuyabilirsiniz.

```
constructor(private activatedRoute: ActivatedRoute) {  
  const id = activatedRoute.snapshot.params["id"];  
}
```

# Observable Kullanarak URL Parametrelerini Okuma

ActivatedRoute Observable bir davranışla route parametrelerini okumamızı sağlayabilmektedir.

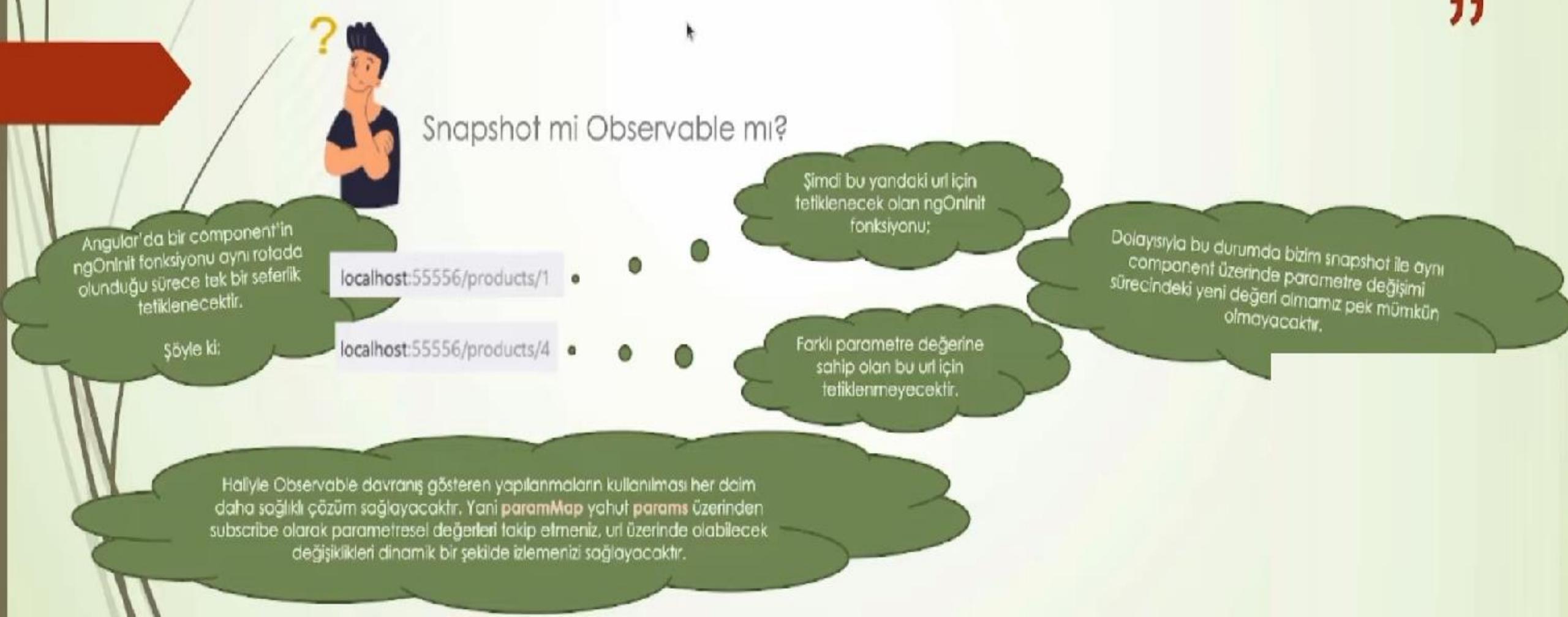
```
constructor(private activatedRoute: ActivatedRoute) {  
  activatedRoute.paramMap.subscribe(  
    next: param => console.log(param.get("id"))  
  );  
}
```

```
constructor(private activatedRoute: ActivatedRoute) {  
  activatedRoute.params.subscribe(  
    next: param => console.log(param["id"])  
  );  
}
```

paramMap üzerinden subscribe olursanız yakalanan değer paramMap türünden olacaktır.

params üzerinden subscribe olursanız yakalanan değer params türünden olacaktır.

# Peki Hangisini Kullanmalıyız?



# RouterLink Directive’inde Query String Kullanma

```
activatedRoute.queryParamMap.subscribe({  
  next: params => console.log(params.get("page"))  
})  
  
activatedRoute.queryParams.subscribe({  
  next: params => console.log(params["page"])  
})
```

”

RouterLink direktifi ile bir bağlantıda query string değeri tanımlayabilmek için [queryParams] direktifini kullanabilirsiniz.

```
<a routerLinkActive="active" routerLink="products" [queryParams]="{page:2, size:10}">Products</a>
```

# queryParamsHandling

**preserve** : Mevcut sayfanın query string değerlerini korur.  
Yönlendirme yapılacak sayfanın query string değerlerini  
getirmez.

localhost:55556/ypage?a=2&b=10

”

Angular mimarisinde her routing işlemi neticesinde default olarak URL'deki query string'ler kaybolacaktır. Bu varsayılan davranışını değiştirebilmek için queryParamsHandling stratejisini aşağıdaki gibi kullanabilirsiniz.

```
<a routerLinkActive="active" routerLink="xpage" [queryParams]="{a:2, b:10}">X Page</a>
<a routerLinkActive="active" routerLink="ypage" [queryParams]="{c:2, d:10}" queryParamsHandling="preserve">Y page</a>
```

```
<a routerLinkActive="active" routerLink="xpage" [queryParams]="{a:2, b:10}">X Page</a>
<a routerLinkActive="active" routerLink="ypage" [queryParams]="{c:2, d:10}" queryParamsHandling="merge">Y page</a>
```

**merge** : Mevcut sayfanın query string değerlerini korur ve  
yönlendirme yapılacak sayfanın query string değerleriyle  
birleştirir.

localhost:55556/ypage?a=2&b=10&c=2&d=10

# Angular Guards Nedir?

Angular'da Guard yapılmaları uygulamadaki rotalara erişim izni vermek ya da reddetmek için kullanılan mekanizmalarıdır.

Genellikle kullanıcıların oturum işlemleri, kimlik doğrulama süreçleri, rota ve sayfalarla erişim yetkileri Guard'lar üzerinden kontrol edilmektedir.

Angular'da CanActivate, CanDeactivate, Resolve, CanLoad(CanMatch) ve CanActivateChild olmak üzere 5 adet guard yapısı mevcuttur.

## CanActivate

Kullanıcı tarafından erişilmek istenen rotanın etkinleştirilip etkinleştirilemeyeceğine veya component'in kullanılıp kullanılmayacağına karar veren guard'dır. Genellikle ilgili kullanıcının hedef component'e erişim istediği durumlarda yetki kontrolü yapmak için kullanılmaktadır.

## CanActivateChild

Bu guard ise bir child route'a erişip erişmeyeceğini karannı vermektedir. CanActivate'in akrabasıdır. CanActivate; ana rotalar için geçerliyken alt rotalarda çalışmamakta. Haliyle bu durumda CanActivateChild devreye girecektir.

## Resolve

Bu guard yapısı bazı görevler tamamlanana kadar rotanın etkinleştirilmesini geciktirecektir. Genellikle, talep edilen component'in işlevsel hale gelebilmesi için bir endpoint üzerinden verilere ihtiyacı olduğu durumlarda, sayfaya yönlendirme yapılmaksızın önce API'dan verilerin çekilip hazırlanması gereklili durumlarda kullanılır.

## CanLoad

Bir modülün kontrollü bir şekilde yüklenmesini sağlar. CanActivate, rotanın erişimini kontrol edip gerekirse engellerken, CanLoad, lazy loading ile yüklenenek modülerin yüklenmesini kontrol edecek ve gerekligi takdirde engellemeyecektir. Haliyle ligili modül içerisinde tüm eylemler korunmuş olacak ve kısmi performansa da kavuşacaktır.

## CanDeactivate

Kullanıcının bulunduğu component'ten ayrılmayacağına karar veren ya da bir başka deyişle geçerli rotadan çıkmak istediği durumu kontrol eden guard'dır. Genellikle kullanıcı tarafından işlem yapılan sayfalarla, kaydedilmemiş verilerin olabileceği durumlarda hayatı öneme sahiptir.

Tüm bunların dışında Angular 14. sürümünde CanMatch guard yapısı da gelmiştir. Bu guard ile CanLoad komple deprecated edilmiştir.

## CanMatch

Bu guard sayesinde aynı rotayı farklı şartlarda erişilebilecek şekilde tanımlamayabilmek mümkünleşmektedir. (Normalde bir path sade ve sadece tek sefer tanımlanır) Misal olarak uygulamanın boş path'ine giriş yaptığı haliyle girişinde gelecek olan sayı içeriğle giriş yapılmaksızın gelecek içerik farklı olacaktır. İşte bu tarz aynı rota için birden fazla şartlı kontrol yapabilmemizi sağlayan bir guard'dır.

```
{
  { path: '', component: LoggedInHomeComponent, canMatch: [isloggedin] },
  { path: '', component: HomeComponent }
}
```

# Guard Nasıl Oluşturulur ve Kullanılır?

Yakın zamana kadar  
Angular'da guard yapılan özel  
interface'ler üzerinden şöyle  
tanımlanmaktaydı.

```
export class CanActivateGuard implements CanActivate
export class CanActivateChildGuard implements CanActivateChild
export class CanDeactivateGuard implements CanDeactivate<unknown>
export class CanLoadGuard implements CanLoad
export class CanMatchGuard implements CanMatch
```

'CanActivate' is deprecated.  
'CanActivateChild' is deprecated.  
'CanDeactivate' is deprecated.  
'CanLoad' is deprecated.  
'CanMatch' is deprecated.

Eee napsaz o  
halde hoca?

Her iki usulü de göreceğiz ☺ Bu  
interface'ler her ne kadar  
deprecated edilmiş olsa da hala  
Angular içerisinde gelmektedirler.

Haliyle önce bu interface'ler  
Üzerinden guard'ların nasıl  
tanımlandığını sizlere gösteriyor  
olacağım.

Sonrasında ise Angular'da ki guard'lar için  
daha da idealleştirilmiş bir yeni davranış  
olan **Functional Router Guard** isimli  
yaklaşımı, nedenleriyle beraber istişare  
ederek inceleyeceğiz.

Ama dikkat ederseniz burada  
kullanılan interface'lerin hepsinin  
Üstü çizili!

Neden mi?

Çünkü decreased edildiler ☺

Şimdi öncelikle bir guard'ı oluşturmak ve kullanabilmek için eski usulü göre incelemeye başlalım. Bunun için aşağıdaki dört adının uygulanması gerekmektedir:

1. İhtiyaç olan guard için bir sınıf oluşturunuz ve hangi guard'ı kullanacağınız onun interface'sinden türetiniz. Ya da Angular CLI üzerinden guard talimatı vererek, uygun türde guard'ı oluşturabilirsiniz.
2. Guard'ın yapacağı işlemi metod içerisinde uygulayınız.
3. Guard'i import ediniz.
4. Guard'ı ilgili route'a uygun property üzerinden ekleyiniz.

Şimdi gelin tüm bu süreci misal olarak CanActivate'e guard'ı üzerinden örnekleyelim.

1. Adım:  
Guard'ı  
oluşturunuz.

Guard'ı oluşturabilmek için bir sınıf oluşturup, CanActivate interface'sini implement edebilir ya da Angular CLI üzerinden aşağıdaki talimatı vererek istediğiniz türden guard'ı hızla oluşturabilirsiniz.

```
PS C:\Users\16534510960\Desktop\ex> ng g g guard/example
> Which type of guard would you like to create? (Press <enter> to proceed)
> (*) CanActivate
```

- ( ) CanActivateChild
- ( ) CanDeactivate
- ( ) CanLoad
- ( ) CanMatch

Bu şekilde guard'ı oluşturduktan sonra yandaki gibi bir sınıf oluşturulacaktır.

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';
```

```
#Injectable({
  providedIn: 'root'
})
export class ExampleGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree | Promise<boolean | UrlTree> | boolean | UrlTree [
      return true;
  ]
}
```

Gördüğü üzere guard özünde bir injectable'dır yani servistir.

2. Adım: İşlevi  
uygulayınız.

Guard'ı oluşturduktan sonra işlevinizi misal aşağıdaki gibi gerçekleştiriniz.

```
canActivate(route: ActivatedRouteSnapshot,
state: RouterStateSnapshot) {
  let userAuthenticated: boolean = true
  if (!userAuthenticated)
    return true
  return false;
}
```

3. Adım:  
Import ediniz.

Hazırlanın guard'ı kullanabilmek için uygun modülde import ediniz. Keza guard yapısı Angular CLI ile oluşturulduğunda, default olarak providedin özelliği 'root' olarak geleceği için bu işlemi bizzat yapmanız gereklidir.

4. Adım:  
Route'a  
ekleyiniz.

Artık bu guard'ın hangi route'ta kullanılmasını istiyorsanız onun tanımına aşağıdaki gibi ekleyiniz.

```
path: "products", component: ProductComponent, canActivate: [ExampleGuard],
```

İşte bu kadar!

Bizler bu örneklendirme de CanActivate guard'ını kullandığımız için rota ile guard ilişkilendirmesini 'canActivate' üzerinden yapmış olduk. Halbuki Angular'da rota tanımda tüm guard'lara uygun diziler aşağıdaki gibi mevcuttur.

```
path: "products", component: ProductComponent, canActivate: [ExampleGuard],
canActivateChild: [],
canDeactivate: [],
canLoad: [],
resolve: [],
canMatch: [],
```

“



Bir kullanıcının ilgili path'e erişim izninin olup olmadığını kontrol etmek için kullanılmaktadır.

# CanActivate - CanActivateFn

## Eski Kullanım - CanActivate

CanActivate guard'ını tanımlayabilmek için eskiden 'CanActivate' interface'inden aşağıdaki gibi istifade ediyorduk.

```
export class ExampleGuard implements CanActivate {
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return true;
  }
}
```

Burada 'canActivate' metodu içerisinde sayfaya erişim gerektiren kontroller yapılmakta ve true değeri döndürüldüğü taktirde erişim gerçekleştirilmektedir.

## Yeni Kullanım - CanActivateFn

CanActivate guard'ını artık 'CanActivateFn' fonksiyonu türünden tanımlayabilmekte ve direkt fonksiyon olarak kullanabilmekteyiz.

```
export const canActivateExample: CanActivateFn = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => {
  return true;
}
```

CanActivate Guard'ını path üzerinde 'canActivate' dizinine tanımlayarak ilgili path ile ilişkilendirebilirsiniz.

{

```
path: "products", component: ProductComponent,
canActivate: [canActivateExample],
```

”

“



Kullanıcının bulunduğu path'ten başka bir path'e geçebilmesinin iznini kontrol etmek için kullanılmaktadır.

# CanDeactivate - CanDeactivateFn

## Eski Kullanım - CanDeactivate

CanDeactivate guard'ını tanımlayabilmek için eskiden 'CanActivate' interface'inden aşağıdaki gibi istifade ediyorduk.

```
export class ExampleGuard<T> implements CanDeactivate<T> {
  canDeactivate(
    component: T,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot) {
      return true;
    }
}
```

Burada 'canDeactivate' metodu içerisinde sayfadan çıkış için gerekli kontroller yapılmakta ve true değeri döndürüldüğü takdirde çıkış gerçekleştirilmektedir.

## Yeni Kullanım - CanDeactivateFn

CanDeactivate guard'ını artık 'CanDeactivateFn' fonksiyonu türünden tanımlayabilmekte ve direkt fonksiyon olarak kullanabilmekteyiz.

```
export const canDeactivateGuard: CanDeactivateFn<unknown> =
  (component: unknown,
   currentRoute: ActivatedRouteSnapshot,
   currentState: RouterStateSnapshot,
   nextState: RouterStateSnapshot) =>
  return true;
```

CanDeactivate Guard'ını path  
Üzerinde 'canDeactivate' dizinine  
tanımlayarak ilgili path ile  
ilişkilendirebilirsınız.

```
{
  path: "products", component: ProductComponent,
  canDeactivate: [canDeactivateGuard],
```

”

Bazen kullanıcıları direkt olarak talep ettikleri sayfalara yönlendirdiğimizde veriler API'lardan gelene kadar boş sayfalarda bekletme durumu söz konusu olabilmektedir.

Bu durumlara karşın veriler gelene kadar yüklediğini ifade eden metin yahut gif dosyalarıyla kullanıcı bilgilendirebilir yahut Resolve guard' ile tam veriler gelene kadar component'i yüklemeden kullanıcıyı bekletebiliriz.

#### Eski Kullanım - Resolve

Resolve guard'ını tanımlayabilmek için eskiden 'Resolve' interface'sinden aşağıdaki gibi istifade ediyorduk.

```
export class ExampleGuard<T> implements Resolve<any> {
  constructor(private httpClient: HttpClient) { }
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.httpClient.get("https://jsonplaceholder.typicode.com/photos");
  }
}
```

Burada 'resolve' metodu içerisinde sayfa için lazım olan API isteği gerçekleştirilmekte ve bu istek tamamlanana kadar sayfanın yüklenmesi geciktirilmektedir.



```
{
  path: "products", component: ProductComponent,
  resolve: { photos: resolveGuard },
```

Kullanıcı tarafından talep edilen rotadaki içeriğin yönlendirme gerçekleştirilmeden önce tam olarak oluşturulmasını beklemek için kullanılmaktadır.

# Resolve' - ResolveFn

#### Yeni Kullanım - ResolveFn

Resolve guard'ını artık 'ResolveFn' fonksiyonu türünden tanımlayabilmekte ve direkt fonksiyon olarak kullanabilmektedir.

```
export const resolveGuard: ResolveFn<any> = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => {
  const httpClient = inject(HttpClient);
  return httpClient.get("https://jsonplaceholder.typicode.com/photos");
};
```

Resolve Guard'ını path üzerinde 'resolve' alanına aşağıdaki gibi tanımlayarak ilgili path ile ilişkilendirebilirsiniz.

```
export class ProductComponent implements OnInit {
  constructor(private router: ActivatedRoute) {}

  datas;
  ngOnInit(): void {
    this.datas = this.router.data.subscribe.photos => {
      this.datas = photos;
    });
  }
}
```

Tabi Resolve guard'ını tanımladıktan sonra da aşağıdaki gibi ActivatedRoute üzerinden kullanabilirsiniz.

”

“



Aynı rotaya farklı şartlarda farklı component'leri yüklememizi sağlamaktadır.

# CanMatch - CanMatchFn

## Eski Kullanım - CanMatch

CanMatch guard'ını tanımlayabilmek için eskiden 'CanMatch' interface'inden aşağıdaki gibi istifade ediyorduk.

```
@Injectable({
  providedIn: 'root'
})
export class IsAdminGuard implements CanMatch {
  canMatch(route: Route, segments: UrlSegment[]) {
    return false;
  }
}

@Injectable({
  providedIn: 'root'
})
export class IsUserGuard implements CanMatch {
  canMatch(route: Route, segments: UrlSegment[]) {
    return true;
  }
}
```

## Yeni Kullanım - CanMatchFn

CanMatch guard'ını artık 'CanMatchFn' fonksiyonu türünden tanımlayabilmekte ve direkt fonksiyon olarak kullanabilmekteyiz.

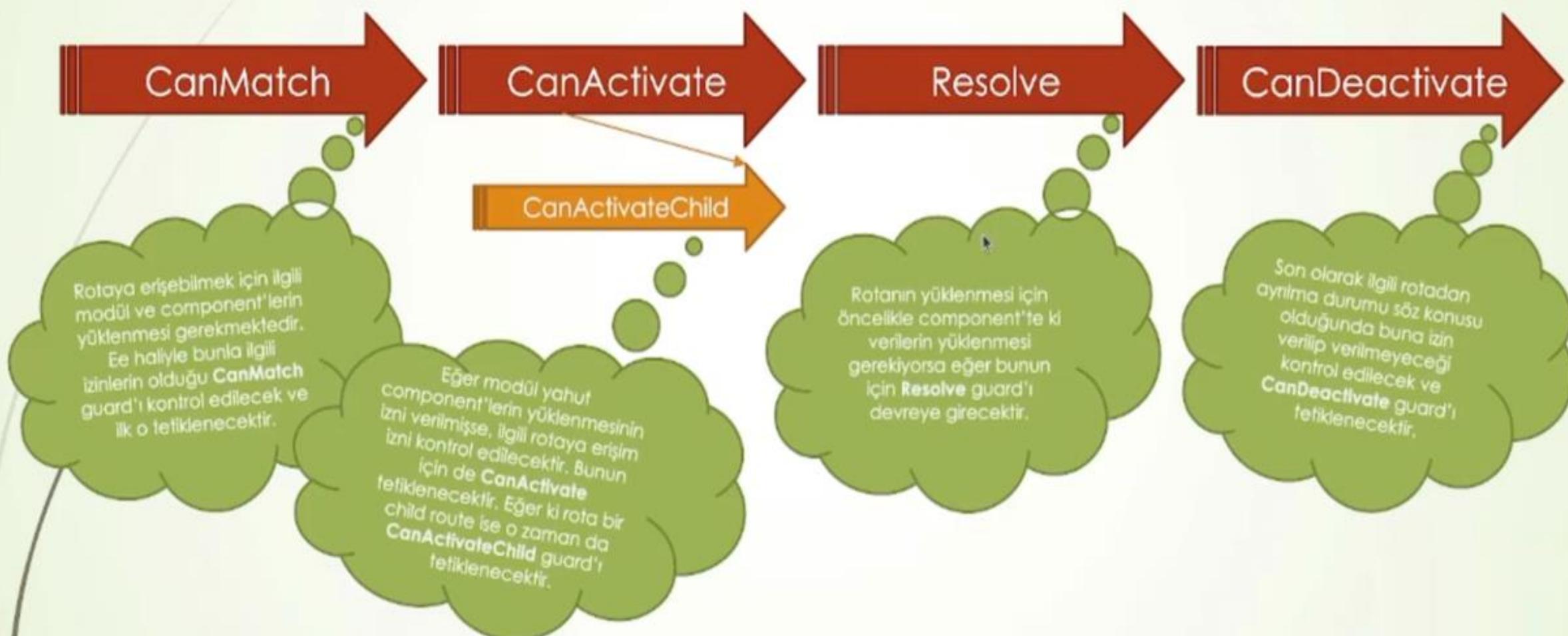
```
export const isAdminGuard: CanMatchFn = (route: Route, segments: UrlSegment[]) => {
  return true;
}

export const isUserGuard: CanMatchFn = (route: Route, segments: UrlSegment[]) => {
  return true;
}
```

CanMatch Guard'ını path üzerinde  
'canMatch' dizinine tanımlayarak ilgili  
path ile ilişkilendirebilirsiniz.

```
]const routes: Routes = [
  { path: "", component: AdminComponent, canMatch: [IsAdminGuard] },
```

# Guard'ların Çalıştırılma(Execution) Sıralaması



# Functional Router Guard Neden Geldi?

Malumunuz ders sürecinde guard yapılanmalarını sizlere tanıtırken hem eski interface üzerinden tanımlarını hem de yeni fonksiyonel hallerini görmüş bulunuyoruz.

Bu fonksiyonel yapılanmanın neden geldiğini ve neden Angular'ın sonraki sürümlerinde interface'lerden kaçınıldığını izah etmemiz gerekirse eğer buna birkaç yorumda bulunabiliriz.

Functional Router Guard yapılanmasına geçsin temelleri arasında sadeleştirilmiş bir JavaScript formatında çalışma arzusu var diyebiliriz.

Normalde eski düzende bir interface üzerinden tanımlanan guard yapılanması esasında sadece bir fonksiyonel görev göreceği halde standartın getirdiği kati ve esneklikten yoksun bir tanıma sahipti.

Functional Router Guard özelliği ile artık guard'lar olması gerekligi esneklige ve sadece fonksiyonel bir tanıma sahiptir. Böylece daha hızlı bir tanımlama ve kullanım söz konusu olmaktadır.

Bunun dışında, Angular 14 ile gelen inject dependency injection'i constructor üzerinden uygulamak mecburiyetinden içerisinde de provider'dan istediğimiz nesneyi talep(inject) edebilmektedir.

Böylece artık bir guard işlemi yapacak fonksiyon için onca class tanımlı, constructor'a vs. ihtiyaç duymaksızın direkt işe odaklanmış salt bir fonksiyon üzerinden ihtiyacımızı giderebilmemekteyiz.

## Functional Router Guard'ın Avantajları

Kısa ve kolay okunabilir, azaltılmış standarta sahip kod

Sadeleştirilmiş mantık

Rota tanımda guard'ın tanımlanabilmesi

Sınıf oluşturma ek yüküne gerek kalmamış iyileştirilmiş performans

# Route Data Passing Nedir?

Route Data Passing, Angular routing mekanizması aracılığıyla sayfalar arasında veri aktarmanın bir yoludur.

Sayfalar arasında veri aktarma deyince aktarma route parameters yahut query string gelmektedir. Bu, route data passing'in onlardan farkı nedir?

Route parameter ve query string yapılanmalari, route data passing ile aynı amaca hizmet etmektedirler. Yani hepsi aynı amaca hizmet eden farklı veri taşıma yöntemleridir.

Tabii ki de aralarında küçükte olsa farklılıklar mevcuttur. Misal olarak route parameters ve query string yapılanmalari: verileri, url'in bir parçası olarak taşıırken, route data passing ise url'den bağımsız olarak veri taşıma işlemini gerçekleştirmektedir.

Yani verileri URL'den ziyade router modülünün 'data' özelliğine aracılığıyla taşımaktadır. Böylece router data passing yöntemi sayesinde url'lerin daha temiz kalması ve okunaklı olması sağlanmaktadır.

Bir diğer fark ise route data passing'in, diğer yöntemlere nazaran verileri güvenli bir şekilde taşımamasına izin vermesidir. Çünkü veriler, URL'in bir parçası olmayacağından daha korunaklı/gizli bir şekilde sayfalar arası transfer gerçekleştirilecektir.

Sonuç olarak, route data passing'in, sayfalar arasında verilerin daha temiz, daha okunaklı ve daha güvenli bir şekilde taşınmasına izin veren çağdaş bir yöntem olduğunu ve bunun yanında route parameter ve query string yöntemlerinin ise URL'in bir parçası olarak veri taşımının geleneksel yöntemleri olduğunu söyleyebiliriz.

Peki bir sayfaya yönlendirme sürecinde route data passing ile herhangi bir veriyi nasıl gönderebiliriz.

Bunun için ilgili sayfanın route tanımında aşağıdaki gibi bir çalışmanın yapılması yeterlidir.

```
{  
  path: "products", component: ProductComponent,  
  data: { value1: "Value1", value2: "Value2" }  
}
```

Burada görüldüğü üzere 'products' path'ine route data passing yöntemiyle bir object tanımlanmıştır. Bu object içerisinde herhangi türden değerler barındırılabilir.

Route'a tanımlanmış bu değeri elde etmek için ise bu rota ile ilişkilendirilmiş component içerisinde yandaki gibi bir çalışma yapılması yeterli olacaktır.

```
@Component({  
  selector: 'app-product',  
  template:  
    <h3>Products</h3>  
    value1 : {{_data.value1}} <br>  
    value2 : {{_data.value2}}  
})  
export class ProductComponent implements OnInit {  
  constructor(private activatedRoute: ActivatedRoute) {}  
  _data: any;  
  ngOnInit(): void {  
    this.activatedRoute.data.subscribe((data: any) => {  
      this._data = data;  
    })  
  }  
}
```

# Route Data Passing Nedir?

Route Data Passing, Angular routing mekanizması aracılığıyla sayfalar arasında veri aktarmanın bir yoludur.

Yani verileri URL'den ziyade router modülünün 'data' özelliğinin aracılığıyla taşımaktadır. Böylece router data passing yöntemi sayesinde url'lerin daha temiz kalması ve okunaklı olması sağlanmaktadır.

Peki bir sayfaya yönlendirme sürecinde route data passing ile herhangi bir veriyi nasıl gönderebiliriz.

```
{  
  path: "products", component: ProductComponent,  
  data: { value1: "Value1", value2: "Value2" }  
}
```

Burada görüldüğü üzere 'products' path'ine route data passing yöntemiyle bir object tanımlanmıştır. Bu object içerisinde herhangi türden değerler barındırılabilir.

Sayfalar arasında veri aktarma devince aktıma route parameters yahut query string gelmemektedir. Bu, route data passing'in onlardan farkı nedir?

Bir diğer fark ise route data passing'in, diğer yöntemlere nazaran veriler güvenli bir şekilde taşımamasına izin vermesidir. Çünkü veriler, URL'in bir parçası olmayacağından daha korunaklı/gizli bir şekilde sayfalar arası transfer gerçekleştirilecektir.

Bunun için ilgili sayfanın route tanımında aşağıdaki gibi bir çalışmanın yapılması yeterlidir.

Route parameter ve query string yapılanmaları, route data passing ile aynı amaca hizmet etmemektedir. Yani hepsi aynı amaca hizmet eden farklı veri taşıma yöntemleridir.

Sonuç olarak, route data passing'in, sayfalar arasında verilerin daha temiz, daha okunaklı ve çağdaş bir yöntem olduğunu ve bunun yanında route parameter ve query string yöntemlerinin ise URL'in bir parçası olarak veri taşımının geleneksel yöntemleri olduğunu söyleyebiliriz.

Tabii ki de aralarında küçüğte olsa farklılıklar mevcuttur. Misal olarak route parameters ve query string yapılanmaları: verileri, url'in bir parçası olarak taşıırken, route data passing ise url'den bağımsız olarak veri taşıma işlemini gerçekleştirmektedir.

```
@Component({  
  selector: 'app-product',  
  template:  
    <h3>Products</h3>  
    value1 : {{_data.value1}} <br>  
    value2 : {{_data.value2}}  
})
```

```
export class ProductComponent implements OnInit {  
  constructor(private activatedRoute: ActivatedRoute) { }  
  _data: any;  
  ngOnInit(): void {  
    this.activatedRoute.data.subscribe((data: any) => {  
      this._data = data;  
    })  
  }  
}
```

Unutmayın ki, getCurrentNavigation fonksiyonuna sadece constructor üzerinden erişim gösterebilirsiniz.

```
<a routerLink="products" [state]="{{ value1: 'Value1', value2: 'Value2' }}>Product</a>  
  
export class ProductComponent {  
  constructor(private router: Router) {  
    this._data = this.router.getCurrentNavigation().extras.state;  
  }  
  _data: any;
```

# Router Event Nedir?

Angular'da Router Event'ler, Angular Router'ın farklı aşamalarındaki değişiklikleri yakalamak için kullanılan olaylardır.

Bu olaylar sayesinde yönlendirme sürecinin ne zaman başladığı, hangi erdiği ile ilgili bilgiler edinilmektedir.

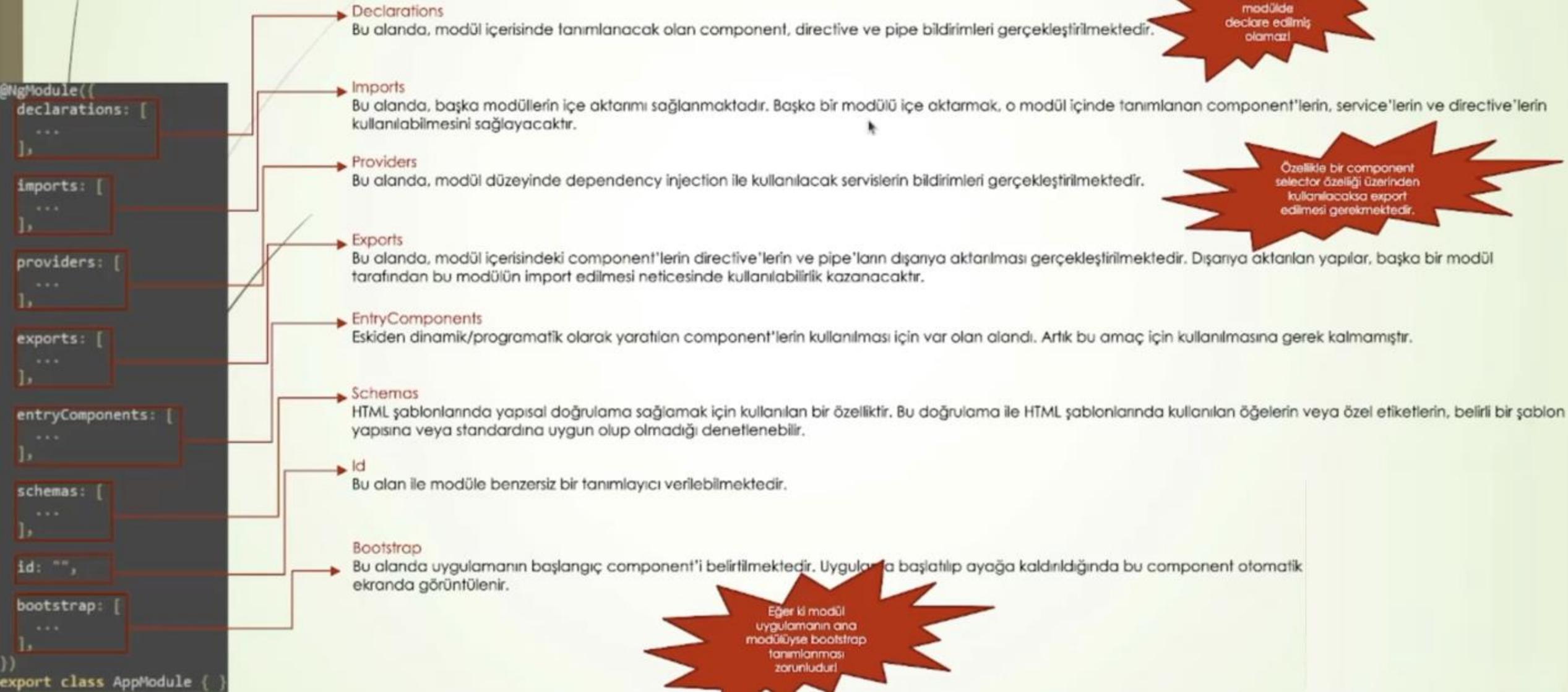
Olaylara '@angular/router' pathi üzerinden yandaki gibi erişebilir ve kullanabilirsiniz.

- NavigationStart  
Yeni bir navigasyon başladığında bu olay tetiklenecektir.
- RoutesRecognized  
Yönlendirmenin gerçekleştirileceği rotalar tanımlandığında bu olay tetiklenir.
- GuardsCheckStart  
Yönlendirmenin yapılacak route'un mevcut guard'ları tetiklediğinde bu olay tetiklenir.
- ChildActivationStart  
Yönlendirilecek rotanın child rotaları etkinleştirildiğinde bu olay tetiklenir.
- ActivationStart  
Rotanın etkinleştirilmesi sırasında bu olay tetiklenir.
- GuardsCheckEnd  
Guard yapılanmanın kontrolü sona erdiğinde bu olay tetiklenir.
- ResolveStart  
Yönlendirme yapılacak sayfanın verileri resolve edilmeye başlandığında bu olay tetiklenir.
- ActivationEnd  
Rotanın etkinleştirilmesi tamamlandıında bu olay tetiklenir.
- ResolveEnd  
Yönlendirme yapılacak sayfanın verileri resolve edildiğinde bu olay tetiklenir.
- NavigationEnd  
Navigasyon tamamlandıında bu olay tetiklenir.
- NavigationCancel  
Navigasyon iptal edildiğinde bu olay tetiklenir.
- NavigationError  
Navigasyon sürecinde bir hata oluşursa bu olay tetiklenir.

```
import {  
  Router,  
  Event,  
  NavigationStart,  
  RoutesRecognized,  
  GuardsCheckStart,  
  ChildActivationStart,  
  ActivationStart,  
  GuardsCheckEnd,  
  ResolveStart,  
  ActivationEnd,  
  ResolveEnd,  
  NavigationEnd,  
  NavigationCancel,  
  NavigationError,  
  Scroll  
} from '@angular/router';
```

```
constructor(private router: Router) {  
}  
ngOnInit(): void {  
  this.router.events.subscribe((event: Event) => {  
    if (event instanceof NavigationStart)  
      console.log('NavigationStart' : ${event})  
    else if (event instanceof RoutesRecognized)  
      console.log('RoutesRecognized' : ${event})  
    else if (event instanceof GuardsCheckStart)  
      console.log('GuardsCheckStart' : ${event})  
    else if (event instanceof ChildActivationStart)  
      console.log('ChildActivationStart' : ${event})  
    else if (event instanceof ActivationStart)  
      console.log('ActivationStart' : ${event})  
    else if (event instanceof GuardsCheckEnd)  
      console.log('GuardsCheckEnd' : ${event})  
    else if (event instanceof ResolveStart)  
      console.log('ResolveStart' : ${event})  
    else if (event instanceof ActivationEnd)  
      console.log('ActivationEnd' : ${event})  
    else if (event instanceof ResolveEnd)  
      console.log('ResolveEnd' : ${event})  
    else if (event instanceof NavigationEnd)  
      console.log('NavigationEnd' : ${event})  
    else if (event instanceof NavigationCancel)  
      console.log('NavigationCancel' : ${event})  
    else if (event instanceof NavigationError)  
      console.log('NavigationError' : ${event})  
  });  
}
```

# NgModule Decorator'ının Anatomisi



```
@NgModule({
  declarations: [
    AddProductComponent,
    DetailProductComponent,
    ListProductComponent
  ],
  imports: [
    CommonModule,
    RouterModule.forChild([
      { path: "add", component: AddProductComponent },
      { path: ":id", component: DetailProductComponent },
      { path: "", component: ListProductComponent }
    ])
  ]
})
export class ProductsModule { }
```

```
@NgModule({
  declarations: [
    AddCustomerComponent,
    DetailCustomerComponent,
    ListCustomerComponent
  ],
  imports: [
    CommonModule,
    RouterModule.forChild([
      { path: "add", component: AddCustomerComponent },
      { path: ":id", component: DetailCustomerComponent },
      { path: "", component: ListCustomerComponent }
    ])
  ]
})
export class CustomersModule { }
```

Normalde bu şekilde tanımlanmış modüllerin uygulamaya import edilmesi beklenir. Lakin bu modüllerin aşağıdaki gibi lazy loading ile sisteme dahil etmemiz bittiğinden boyunca değerlendirilen tüm faydalı sağlıyor olacaktır.

```
const routes: Routes = [
  { path: "", component: HomeComponent },
  { path: "customers", loadChildren: () => import("../app/components/customers/customers.module").then(m => m.CustomersModule) },
  { path: "products", loadChildren: () => import("../app/components/products/products.module").then(m => m.ProductsModule) }
];

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



## Lazy Loading'in Avantajları Nelerdir?

- *İlk yükleme süresini azaltır*

Uygulamanın tüm modüllerini ve bileşenlerini başlangıçta yüklemek yerine, yalnızca kullanıcı tarafından ihtiyaç duyulan anda ilgili modül veya component'in yüklenmesini gerçekleştirir. Bu da uygulama açısından tarayıcıya daha hızlı bir ilk yükleme sağlayacaktır.

- *Performansı arttırmır*

Lazy loading uygulamanın başlangıçta yüklemesi gereken kurucu kaynakların dışındaki modülerin yüklenmesini arındıracağından dolayı uygulama performansını artıracaktır.

- *Client tarafında kaynak kullanımını azaltır*

Uygulamanın tüm kaynaklarını başlangıçta yüklemek yerine, lazy loading kullanarak sadece gerekli(kurucu) kaynakları yüklemek, İstemci tarafından kullanılan kaynak miktarını azaltacaktır. Bu da daha düşük bellek tüketimi ve daha hızlı yanıt süresi sağlayacaktır.

## forRoot ve forChild Farkı

Angular'da **forRoot** ve **forChild** terimleri, işlevsel olarak Angular modüllerinin yönlendirmeyle ilgili yapılandırmalarını gerçekleştirmek için kullanılmaktadır.

**forRoot** metodu, bir modülün ana yönlendirme yapılandırmasını sağlamak için kullanılmaktadır. Ya da bir başka deyişle ana modül için global yönlendirme yapılandırmasını sağlamaktadır.

**forChild** ise, daha da özelleştirilmiş alt modüllerin yönlendirmesini yapılandırmayı sağlamaktadır.

# Angular Preloading Strategy Nedir?

Preloading Strategy, Angular uygulamalarının tarayıcıya yüklenme süresini kısaltmanın bir başka yoludur.

Normal şartlarda Angular mimarisi, tarayıcı üzerinden gelen ilk talep neticesinde tüm modüllerin yüklemeye davranışını sergilemektedir. Bu durum uygulamanın yavaş yüklenmesini sağlayacaktır.

Bu sorunu modüller lazy loading ile yükleyerek çözebileceğimizi biliyoruz. Ayrıca preloading yöntemi ile de uygulamayı daha da optimize ederek yüklenme süresini kısaltabiliyoruz.

Her ne kadar lazy loading modüllerinin dahlilinde yükliy় olsa da, uygulamanın yavaş yüklenen/hantal/ağır/büyük hacimli bir modülüne ihtiyaç duyduğu taktirde yüklenmesi zaman alacak ve ister istemez kullanıcının beklemesine sebebiyet verecektir.

Haliyle bu bekleme süresini de törpüleyebilmek için lazy loading ile yüklenen modüllerin arkapanda asenkron olarak önyüklenmesini gerçekleştirerek yüklenme süresini kısaltabiliyor ve hızımıza hız katabiliyor.

Peki Preloading Strategy'i nasıl kullanacağız?

Preloading Strategy'ı kullanabilmek için yapılması gereken RouterModule'de yandaki konfigürasyonun yapılması yeterlidir.

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

Tabi bunun yapılabilmesi için önceden modüllerin lazy loading ile yüklenmiş/tanımlanmış olması gerekiyor.

Angular'da ön yüklenme stratejisi 'PreloadAllModules' olarak adlandırılır. Bu strateji, uygulamanın başlatılmasıyla birlikte tüm modüllerin arkapanda önyüklenmesini sağlar. Böylece kullanıcı bir sayfaya geçtiğinde tüm modüller hemen kullanılabilir hale gelir.

Bunun yanı sıra 'NoPreloading'ı kullanarak ön yüklenme stratejisini kullanmayarak da çalışmalar devam edebilir. Bu varsayılan davranıştır.

# Custom Preloading Strategy Oluşturma

Preloading Strategy ile tüm modüller ön yüklenenecektir ve bu durum uygulamada yüklenmesi gereken büyük sayıda modül varsa eğer esasında bir darboğaza neden olabilir.

Bu durumda daha iyi bir strateji için şu şekilde bir davranış öneremiz:

- ✓ Başlangıçta gerekli olabilecek ve uygulama açısından her noktada kullanılabilecek tüm çekirdek modüller direkt yükleyin. Misal olarak: kimlik doğrulama modülü.
- ✓ Ardından sık kullanılabilecek modüller yükleyin.
- ✓ Ve geriye kalan modüller de lazy loading yöntemiyle yükleyin.

Özel preloading strategy tanımlayabilmek için bir sınıf oluşturunuz ve bu sınıfa 'PreloadingStrategy' abstract class'ını uygulayınız.

Ardından özel olarak ön yükleme mantığınızı uygulayınız. Aşağıdaki örnekte route data'da ki 'preload' değeri olan modüler 'fn' fonksiyonu çağrılarak ön yüklenmiştir.

Ve son olarak oluşturulan bu custom preloading strategy'i kullanınız.

Burada istenilen modüller yükleyebilmek için özel preloading strategy kullanılması gerekmektedir.

```
import { PreloadingStrategy, Route } from "@angular/router";
import { Observable, of } from "rxjs";

export class CustomStrategy implements PreloadingStrategy {
  preload(route: Route, fn: () => Observable<any>): Observable<any> {
    if (route.data && route.data["preload"]) {
      //Burada özel ön yükleme mantığınızı uygulayabilirsiniz.
      return fn();
    } else {
      //Ön yükleme yapmak istediğiniz modüller için boş bir Observable dönebilirsiniz.
      return of(null);
    }
  }
}
```

# Standalone Components&Directive&Pipe Nedir?

Angular'da bir component'i kullanabilmek için onu oluşturmak yetmemekte, kullanıldığı modüle declare edilmesi gerekmektedir.

Ki bunların yanında istediğiniz kadar tecrübeli biri olun, bir component oluşturulduğu taktirde ihtiyaç noktalarına declare edilmesi yahut export edilme gereği ister istemez unutulmaka ve geliştirme sürecinde de bu unutmalar yavaşlığa sebebiyet verebilmektedir.

Standalone component oluşturabilmek için Angular CLI'dan aşağıdaki talimatı verebilirisiniz.

ng g c ...name.... -standalone

Ya da declare ettiği modülün ihtiyaç olan başka bir modüle import edilmesi gerekmektedir.

Yani anlayacağınız, bir component'in mimarisel olarak oyuna sokulabilmesi için belli bir maliyetin ödemesi gerekmektedir.

Bu maliyet, özellikle Angular'ı yeni öğrenenler için öğrenme hızını oldukça yavaşlatmaktadır.

Çünkü bir component'i kullanabilmek için modüler yapının anlaşılması ve bu yapının gereği olan export, declare vs. gibi davranışların kafada oturtulması gerekmektedir.

```
@Component({
  selector: 'app-standalone2',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './standalone2.component.html',
  styleUrls: ['./standalone2.component.scss']
})
export class StandaloneComponent { }
```

Peki standalone component nasıl kullanılır?

Standalone component, hangi modülde kullanılacaksa orada import edilmelidir.

Evet, yanlış okumadınız... Standalone component'ler kullanıacakları modüllerde declare değil import edilmek zorundadır.

Peki bir standalone component nasıl oluşturulur?



Tabi buradan anlayacağınız bir standalone component oluşturabilmek için illa Angular CLI kullanmanız gerekmemektedir, var olan component'te de bu ayarın yapılması yeterli olmaktadır.

# Standalone Yapılarda Routing ve Lazy Loading

Uygulama standalone altyapısı sayesinde modüler yapılanmadan soyutlandığına göre bu durum routing işlemleri için de aynı bir nitelik kazandırmaktadır.

Nihayetinde lazy loading süreçlerinde modüler yapılanmalar **loadChildren** ile bildiriyorken, standalone component'leri ise modüller olmadığı için **loadComponent** fonksiyonu ile bildirmemiz gerekmektedir.

```
const routes: Routes = [
  { path: "", component: HomeComponent },
  { path: "customers", loadChildren: () => import("../app/components/customers/customers.module").then(m => m.CustomersModule) },
  { path: "products", loadChildren: () => import("../app/components/products/products.module").then(m => m.ProductsModule) },
  { path: "standalone", loadComponent: () => import("../app/standalone/standalone.component").then(m => m.StandaloneComponent) }
];
```

# Standalone Component'leri Kullanarak Uygulamayı Önyükleme

Bir Angular uygulamasında, uygulamanın root component'i olarak 'AppComponent' yerine bir standalone component kullanmak ve bu component'le önyükleme işlemini gerçekleştirmek istiyorsanız bunu root module olan ' AppModule 'de yapamazsınız.

Bu işlemin yapılabilmesi için ' AppModule 'ün kaldırılması ve '**main.ts**' dosyasında **bootstrapApplication** fonksiyonunun yandaki gibi kullanılması gerekmektedir.

```
--main.ts
//platformBrowserDynamic().bootstrapModule(AppModule)
//.catch(err => console.error(err));
bootstrapApplication(StandaloneComponent)
```

Tabi bu işlemden sonra da 'index.html'de root component'in selector'ını değiştirmeyi unutmayın. 

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Ex</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-standalone></app-standalone>
</body>
</html>
```

Bu **bootstrapApplication** fonksiyonu hakkında sonraki dersimizde daha detaylı bir incelemeye bulunacak ve main module olan ' AppModule 'un Angular uygulamalarından kaldırılıp, artık temel konfigürasyonun 'main.ts' üzerinden yapılması hakkında gerekli incelemelerde bulunacağız.

```
import { AppModule } from './app/app.module';
import { bootstrapApplication } from '@angular/platform-browser';
import { StandaloneComponent } from './app/components/standalone/standalone.compon
```

♦

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

**bootstrapApplication(StandaloneComponent)**

```
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-standalone></app-standalone>
</body>
</html>
```

Hatırlarsanız eğer Angular'da built-in olarak gelen \*ngFor, \*ngIf vs. gibi directive'ler ve pipe'lar normalde CommonModule içerisinde gelmektedirler.

CommonModule, kullanmasakta uygulamaya dahil etmektedir.

Halbuki bizler uygulamada built-in olarak gelen bu yapılarından sadece lazım olanları yüklemek isteyebilir ve uygulama boyutunu sınırmek istemeyebiliriz.. İşte bunun için Angular 15'te CommonModule'e bir yenilik getirilmiştir.

```
import { CurrencyPipe, NgFor, NgIf } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    {{1000 | currency}}
    <div *ngIf="0">
      Merhaba
    </div>
  `,
  imports: [NgIf, NgFor, CurrencyPipe]
})
export class AppComponent {
```

Kullanacağınız tüm pipe ve directive'leri yandaki gibi aynı import edebilirsiniz.

Ya da 'ulan tek tek bunlara mi uğraşılır' derseniz CommonModule'ü de yandaki gibi kullanarak tek seferde tümünü yükleyebilirsiniz.

```
import { CommonModule, Curr
import { Component } from 'a

@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    {{1000 | currency}}
    <div *ngIf="0">
      Merhaba
    </div>
  `,
  imports: [CommonModule]
})
export class AppComponent {
```

# CommonModule Yeniliği

# bootstrapApplication NEDİR?

Bir Angular uygulamasını, Standalone Component kullanarak ve herhangi bir uygulamaya bağımlılık olmaksızın önyükleyecek olan fonksiyondur.

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { StandaloneComponent } from './app/standalone/standalone.component';

bootstrapApplication(StandaloneComponent)
```

Bu component'in kesinlikle  
standalone component olması  
gerekmektedir.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-standalone',
  template: 'Standalone Component',
  standalone: true,
  imports: []
})
export class StandaloneComponent { }
```

## Standalone Yapılanmada AppModule'den Kurtulmak

Standalone component'ler de ki amacın mümkün mertebe uygulamadaki modüler bağımlılıkları törpülemek olduğunu söylemişтик.

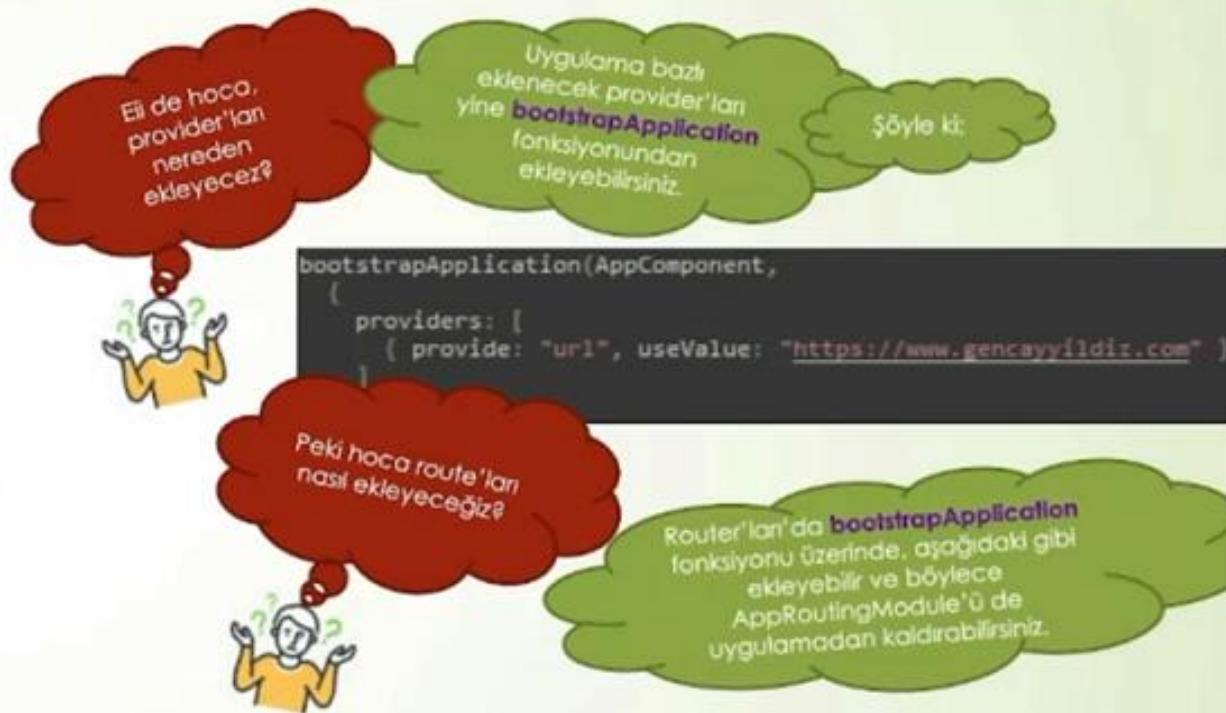
Haliyle bu amaca ilk olarak uygulamadan AppModule'un kaldırılmasıyla yaklaşım sergileyebiliriz.

Dolayısıyla uygulamanın direkt bootstrapApplication fonksiyonu ile bir standalone component üzerinden ayağa kalkması, bu aşamadan sonra AppModule'e olan bağımlılığı ortadan kaldıracak ve bu dosyanın silinebilir olmasını sağlayacaktır.

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { StandaloneComponent } from './app/standalone/standalone.component';

bootstrapApplication(StandaloneComponent)
```



```
bootstrapApplication(AppComponent,
{
  providers: [
    { provide: "url", useValue: "https://www.gencayyildiz.com" },
    provideRouter([
      { path: "", loadComponent: () => import("./app/components/home/home.component").then(s => s.HomeComponent) },
      { path: "about", loadComponent: () => import("./app/components/about/about.component").then(s => s.AboutComponent) },
      { path: "products", loadComponent: () => import("./app/components/product/product.component").then(s => s.ProductComponent) }
    ])
});
```

# Router Inputs

Angular 16 ile component'ler açısından route'dan bilgi alınmasını basitleştirecek ve çok daha kolay hale getirecek olan Router Inputs özelliği gelmiştir.

Gördüğü üzere Router Inputs özelliği; Route & Query Parameters, Route Data Passing ve Resolve Guard gibi tüm yapılarından hızlı ve kolayca veri okuyabilmemizi sağlayan bir nitelik sahiptir.

Tabii bu özelliğin kullanabilmek için uygulamadaki **Component Input Binding** özelliğinin etkinleştirilmesi gerekmektedir.

Haklı bunun için RouterModule'de **bindToComponentInputs** özelliği aktifleştirilmeli ya da standalone bir uygulamada çalışıysa eğer **withComponentInputBinding** konfigürasyonu çağrılmalıdır.

```
bootstrapApplication(AppComponent, {  
  providers: [  
    importProvidersFrom(HttpClientModule),  
    provideRouter([  
      {  
        path: "products/:id",  
        data: {  
          productName: "Corap"  
        },  
        resolve: {  
          products: (route, state) => {  
            const httpClient = inject(HttpClient)  
            return httpClient.get("https://jsonplaceholder.typicode.com/posts");  
          }  
        },  
        component: ProductComponent  
      }  
    ]  
  ]);  
});
```

## Angular 16 - Route Inputs

```
export class ProductComponent {  
  @Input() id: number;  
  @Input() productName: string;  
  @Input() products: any[];  
}
```

Halbuki artık Angular 16 ile gelen Router Inputs özelliği sayesinde route ile ilgili çalışmalarla ActivatedRoute servisini inject etme zahmetinden kurtulmuş bulunuyoruz.

Bunuda component içerisindeki @input değişkenler üzerinden gerçekleştiriliyoruz.

”

Bizler bir route'dan parameter değerini okumak istediğimizde yahut resolve guard'ından gelecek olan veriyi yakalamanak istediğimizde veya hatta route data passing'de tanımlanmış olan bir veriyi elde etmek istediğimizde sürekli ActivatedRoute nesnesini inject ederek çalışmak zorundaydık!

```
bootstrapApplication(AppComponent, {  
  providers: [  
    importProvidersFrom(HttpClientModule),  
    provideRouter([  
      {  
        path: "products/:id",  
        data: {  
          productName: "Corap"  
        },  
        resolve: {  
          products: (route, state) => {  
            const httpClient = inject(HttpClient)  
            return httpClient.get("https://jsonplaceholder.typicode.com/posts");  
          }  
        },  
        component: ProductComponent  
      }  
    ]  
  ]  
});
```

Halbuki artık Angular 16 ile gelen Router Inputs özelliğinin sayesinde route ile ilgili işlemlerde ActivatedRoute servisini inject etme zahmetinden kurtulmuş bulunuyoruz.

Bunuda component içerisindeki @Input değişkenler üzerinden gerçekleştiriliyoruz.

```
export class ProductComponent {  
  constructor(private activatedRoute: ActivatedRoute) {}  
  activatedRoute.paramMap.subscribe(params => console.log(params.get("id")));  
  activatedRoute.data.subscribe(datas => console.log(datas["productName"]));  
  activatedRoute.data.subscribe(datas => console.log(datas["products"]));  
}
```

#### Angular 16 – Route Inputs

```
export class ProductComponent {  
  @Input() id: number;  
  @Input() productName: string;  
  @Input() products: any[];  
}
```

Burada görüldüğü üzere route'da ki tanımlı olan tüm değerler Input değişkenlerle eşleştirilmekte ve elde edilmektedir.

# Router Inputs

Angular 16 ile component'ler açısından route'dan bilgi alınmasını basitleştirecek ve çok daha kolay hale getirecek olan Router Inputs özelliği gelmiştir.

Gördüğü üzere Router Inputs özelliği; Route & Query Parameters, Route Data Passing ve Resolve Guard gibi tüm yapılarından hızlı ve kolayca veri okuyabileceğimizi sağlayan bir niteliğe sahiptir.

Tabii bu özelliği kullanabilmek için uygulamadaki Component Input Binding özelliğinin etkinleştirilmesi gerekmektedir.

Hالىلے bunun için RouterModule'de bindToComponentInputs özelliği aktifleştirilmeli ya da standalone bir uygulamada çalışıysa eğer withComponentInputBinding konfigürasyonu çağrılmalıdır.

Tüm bunların dışında Router Inputs özelliğinin uygulama bazındaki davranışını aşağıdaki öncelikle göre seyrededecektir.



```
RouterModule.forRoot([], {  
  bindToComponentInputs: true  
})
```

- `ng-template` Angular bileşeni içinde birden çok kez yeniden kullanılabilcek bir şablon tanımlamaya yardımcı olur.
- `ng-container` DOM'a ek düğümler eklemeden öğeleri gruplandırmamız gerekiğinde.
- `ng-content` ana bileşenden aktarılan içeriği dinamik olarak oluşturmak için kullanılır.

# ng-content Nedir?

Biliyorsunuz ki, component'ler arası iletişim için @Input ve @Output decorator'larından istifade edebiliyor ve özellikle parent component'ten child component'e veri aktarmak için @Input decorator'ını kullanabiliyoruz.

Fakat bu decorator'lar eşliğinde component'ler arası iletişim sadece verilerle sınırlıdır. Halbuki bizler bazen component'ler arası HTML öğeleri içeren kodları paylaşmak ve edinen component tarafından bu kodları render etmek isteyebiliriz.

İste böyle bir durumda ng-content özelliğinden istifade edebiliriz. ng-content, html içeriklerini parent component'ten child component'e geçirmemizi ve render etmemizi sağlayan, böylece dinamik bir şekilde harici şablonları oluşturmamıza imkan tanıyan bir özelliktir.

İşlevsel olarak child component'in html'inde istenilen noktayı ya da noktaları işaretlememizi sağlayarak parent'tan gelecek olan html öğelerini basmamızı ve yeniden kullanabilir bir component oluşturmamızı mümkün kılmaktadır.

## Parent component

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <app-home>
      <span style="color:red;">Burası ng-content içeriğidir...</span>
    </app-home>
  `,
  imports: [HomeComponent]
})
export class AppComponent { }
```

## Child component

```
@Component({
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  template: `
    Home
    <br>
    <ng-content></ng-content>
  `
})
export class HomeComponent { }
```

## Home

Burası ng-content içeriğidir...

## Multiple ng-content

```
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  template: `
    <ng-content select="header"></ng-content>
    <ng-content select="body"></ng-content>
    <ng-content select="footer"></ng-content>
```

```
export class HomeComponent {
```

```
selector: 'app-root',
standalone: true,
template: `
<app-home>
  <header>Header içeriği...</header>
  <body>Body içeriği...</body>
  <footer>Footer içeriği...</footer>
</app-home>
```

```
imports: [ HomeComponent ]
```

Header içeriği

Body içeriği...

Footer içeriği...

```
<app-root ng-version="15.2.9"
```

→ <app-home>

<header>Header içeriği...</header>

<body>Body içeriği...</body>

<footer>Fo

</app-home

Bu özel isimlerin dışında CSS selector'ı kullanarakta kendinize göre özel isimle kullanabilirsiniz.

```
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  template: `
    <table>
      <tr>
        <td>
          <ng-content select=".solMenu"></ng-content>
        </td>
        <td>
          <ng-content select=".ortaMenu"></ng-content>
        </td>
        <td>
          <ng-content select=".saatMenu"></ng-content>
        </td>
      </tr>
      <tr>
        <td colspan="3">
          <ng-content></ng-content>
        </td>
      </tr>
    </table>
```

```
port class HomeComponent {
```

```
    selector: 'app-root',
    standalone: true,
    template: `
      <app-home>
        <div class="solMenu">Sol menü içeriği...</div>
        <div class="ortaMenu">Orta menü içeriği...</div>
        <div class="sagMenu">Sağ menü içeriği...</div>
        <br/>
        <div style="border: 2px solid red; border-radius: 50%; padding: 5px; text-align: center; width: 20px; margin: 0 auto; margin-top: 10px;>
          Diğer...
        </div>
      </app-home>
    `,
    imports: [ HomeComponent ]
  }

  export class AppComponent {
```

# ng-container Nedir?

ng-container, sayfa üzerinde HTML elementleriyle uğraşmaksızın bir bölüm/alan oluşturamamızı olanak sağlayan, Document Object Model(DOM) içerisinde tanımlanmayan lakin içinde içerik barındırabilen, özi itibariyle ne directive, ne component ne de class yahut interface olan, sade ve sadece bir sözdizimi/syntax ögesinden ibaret olan bir özelliktir.

Peki bunu  
nereerde  
kullanacağız?

ng-container'ı genellikle  
ngIf, ngFor vs, gibi  
direktifler eşliğinde  
kullanabiliyoruz. Misal olarak  
yandaki çalışmaya göz  
atalım;

Yandaki çalışmada belli  
başlı Ürünleri barındıran  
bir liste görüyoruz ve bizler  
bu ürünlerden mevcut  
olanları ekrana listelemek  
istiyoruz.

Bunun için ng-container  
olmaksızın tüm ürünleri tek  
tek span içerisinde ngIf ile  
kontrol ediyor ve uygunsa  
ekrana yazdırıyor.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h1>Başlık</h1>
    <p>İçerik</p>
    <ng-container>
      Container içerik...
    </ng-container>
  `})
  imports: [HomeComponent]
)
export class AppComponent {
```

```
Component({
  selector: 'app-product',
  standalone: true,
  imports: [CommonModule],
  template: `
    <ul>
      <span *ngFor="let product of products">
        <li *ngIf="product.available">{{product.productName}}</li>
      </span>
    </ul>
  `})
  export class ProductComponent {
    products = [
      { productName: "Pencil", available: true },
      { productName: "Notebook", available: true },
      { productName: "Duster", available: false },
      { productName: "Book", available: true },
      { productName: "Table", available: false },
      { productName: "Bin", available: true },
    ];
}
```

Bu örneği incelerseniz  
eğer, bunun render  
edilmiş hali yandaki  
görseldeki gibi olacaktır.

Haliyle bu çalışmayı  
render ettiğimizde  
tarayıcıya aşağıdaki  
kaynakta bir çıktı  
üretiliyor.

```
<app-root ng-version="15.2.9">
  <h1>Başlık</h1>
  <p>İçerik</p>
  Container içerik...
  <!--ng-container-->
</app-root>
<ul>
  <span>
    <li></li>
    <!--bindings:<ng-reflect-ng-if:> "true" -->
  </span>
  <span>
    <li></li>
    <!--bindings:<ng-reflect-ng-if:> "true" -->
  </span>
  <span>
    <li></li>
    <!--bindings:<ng-reflect-ng-if:> "false" -->
  </span>
  <span>
    <li></li>
    <!--bindings:<ng-reflect-ng-if:> "true" -->
  </span>
  <span>
    <li></li>
    <!--bindings:<ng-reflect-ng-if:> "false" -->
  </span>
</ul>
```

ng-container'in render  
edildikten sonra herhangi  
bir HTML elementine  
karşılık gelmediğine  
dikkatinizi çekerim.

Haliyle dikkat ederseniz,  
render neticesinde tüm  
ürünlere karşılık bir span  
basılmıştır. Hatta 'available'  
false olan ürünler için del

Dolayısıyla bu durum  
üretilen HTML dosyası  
açısından kirilik arz  
etmektedir.

Bizler bu HTML dosyalarını temiz tutabilmek ve  
render neticesinde üretilcek kaynak kodu da  
mükemmeli mertebe performanslı olacak şekilde sade  
bir hale getirmek için yaklaşımımız değiştirmeli ve  
span içerisinde çalışma sergilememeliyiz! Bu tarz  
durumlar için ng-container'ı kullanabiliriz.

```
Component({
  selector: 'app-product',
  standalone: true,
  imports: [CommonModule],
  template: `
    <ul>
      <ng-container *ngFor="let product of products">
        <li *ngIf="product.available">{{product.productName}}</li>
      </ng-container>
    </ul>
  `))
export class ProductComponent {
  products = [
    { productName: "Pencil", available: true },
    { productName: "Notebook", available: true },
    { productName: "Duster", available: false },
    { productName: "Book", available: true },
    { productName: "Table", available: false },
    { productName: "Bin", available: true },
  ];
}
```

```
<ul>
  <li></li>
  <!--bindings={ "ng-reflect-ng-if": "true" }-->
  <!--ng-container-->
  <li></li>
  <!--bindings={ "ng-reflect-ng-if": "true" }-->
  <!--ng-container-->
  <!--bindings={ "ng-reflect-ng-if": "false" }-->
  <!--ng-container-->
  <li></li>
  <!--bindings={ "ng-reflect-ng-if": "true" }-->
  <!--ng-container-->
  <!--bindings={ "ng-reflect-ng-if": "false" }-->
  <!--ng-container-->
  <li></li>
  <!--bindings={ "ng-reflect-ng-if": "true" }-->
  <!--ng-container-->
  <!--bindings={ "ng-reflect-ng-for-of": "[object Object],[object Object]" }-->
</ul>
```

Böylece olmayan ürünler için lüzumsuz yere bir span nesnesi üretilmeyeceği için maliyet ve performans açısından lehimize bir davranış serglemiş oluyoruz.

# ng-template Nedir?

ng-template, ng-container gibi sayfa üzerinde HTML elementleriyle uğraşmaksızın bir bölüm/alan oluşturamamıza olanak sağlayan ve Document Object Model(DOM) içerisinde tanımlanmayan bir özelliktir.

ng-container'dan temel farkı, render edildiği taktirde dahi ayn apartlar aracılığıyla gösterilmmediği sürece içerisindeki HTML elementlerin DOM üzerine işlenmemesidir.

## ngTemplateOutlet Directive'i ile ng-template'i Görüntüleme

Bu yöntemde ng-template içeriğini bir ng-container içerisine yükleyerek görüntülemeyi sağlıyoruz.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <ng-container *ngTemplateOutlet="templateIcerik">
      Container içerik...
    </ng-container>
    <ng-template #templateIcerik>
      Template içerik...
    </ng-template>
  `,
  imports: [NgTemplateOutlet]
})
export class AppComponent {
```

YandaKİ örneği incelersek eğer ilgili ng-template'e bir referans atıyoruz ve bu referansı ng-container üzerinde 'ngTemplateOutlet' direktifi eşliğinde çağırıyoruz.

Yapılan render neticesine göz atarsak eğer ng-template'in ng-container'ın içeriğini ezdiğini görüyoruz.

Template içerik...  
!--bindings={ "ng-reflect-ng-template-outlet": "[object Object]" }-->  
!--container-->

Misal olara aşağıdaki örnek çalışmayı incelersek;

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <ng-container>
      Container içerik...
    </ng-container>
    <ng-template>
      Template içerik...
    </ng-template>
  `,
  imports: []
})
export class AppComponent {
```

Bu kod render edildiğinde ng-container'ın çıktısının DOM'a eklendiğini lakin ng-template'in çıktısının ise eklenmediğini göreceğiz.

Container içerik...  
!--ng-container-->  
!--container-->

Buradan anlıyoruz ki, ng-template'in nerede ve ne zaman görüntüleneceğini bildirmek bizim işimiz. Yani iradeli bir şekilde ng-template'i görüntülememiz gerekmektedir.

Bunun için de ngTemplateOutlet direktifi ve TemplateRef & ViewContainerRef öğeleri eşliğinde olmak üzere iki farklı şekilde ng-template'i görüntüleyebiliriz.

## TemplateRef & ViewContainerRef ile ng-template'i Görüntüleme

```
@Component({  
  selector: 'app-root',  
  standalone: true,  
  template:  
    <ng-template #templateIcerik>  
      Template içerik...  
    </ng-template>  
  })  
  
export class AppComponent {  
  @ViewChild("templateIcerik", { read: TemplateRef }) templateIcerik: TemplateRef<any>;  
  constructor(private viewContainerRef: ViewContainerRef) {}  
  
  ngAfterViewInit() {  
    this.viewContainerRef.createEmbeddedView(this.templateIcerik);  
  }  
}
```

Bu yöntemde ise HTML kısmında tanımlı olan **ng-template** nesnesini component class'ı Üzerinden referans ederek görüntülenmesi sağlanmaya çalışılmaktadır.

**TemplateRef**, **ng-template**'i component class'ı üzerinde temsil etmemizi sağlayan bir sınıfken, **ViewContainerRef** ise **ng-template**'in nerede oluşturulacağını belirtmemizi sağlayan başka bir sınıftır.

Yandaki görseli incelersek eğer yine HTML'deki **ng-template**'e bir referans verilmekte ve bu referans component class'ında **@ViewChild** eşliğinde **TemplateRef** türünde elde edilmektedir.

Aydın丹 dependency injection ile inject edilen **ViewContainerRef** nesni üzerinden **createEmbeddedView** metodu kullanılarak **ng-template** sayfada görüntülenmektedir.

# ng-template Hangi Durumlarda Tercih Edilmeli?

bu durumlara göre şablon gösterimlerinde  
template biçilmiş kaftandır.

```
uent({
  ctor: 'app-root',
  standalone: true,
  late: '',
  'ngIf="value; then template1 else template2"',
  >
  template #template1>
  blon 1
  -template>
  template #template2>
  blon 2
  -template>
  rts: [NgIf]
}
class AppComponent {
  : boolean = true;
```

İndirilen çalışmayı ayrıca aşağıdaki gibi  
[ngIf], [ngIfThen] ve [ngIfElse] direktifleri eşliğinde  
geliştirebiliriz.

```
uent({
  ctor: 'app-root',
  standalone: true,
  late: '',
  template [ngIf]="value" [ngIfThen]="template1" [ngIfElse]="template2",
  >
  template #template1>
  on 1
  -template>
  template #template2>
  on 2
  -template>
  s: [NgIf]
```

Listeleme süreçlerinde de repeat edilecek yapıyı  
ng-template ile oluşturabilirsiniz.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <ul>
      <li *ngFor="let product of products;
        trackBy: trackByProduct">{{product}}</li>
    </ul>
  `,
  imports: [NgFor]
})
export class AppComponent {
  products: string[] = ['pencil', 'mouse', 'duster'];
  trackByProduct = (index, product: string) => product;
}
```

Bu çalışmayı da aşağıdaki gibi ngFor, [ngForOf]  
ve [ngForTrackBy] direktifleri ile birlikte inşa  
edebilirsiniz.

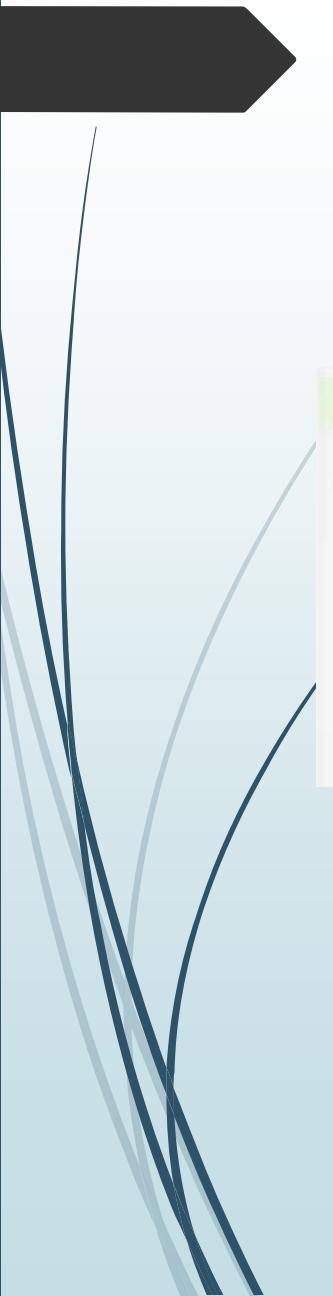
```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <ul>
      <ng-template ngFor let-product
        [ngForOf]="products"
        [ngForTrackBy]="trackByProduct">
        <li>{{product}}</li>
      </ng-template>
    </ul>
  `,
  imports: [NgFor]
```

switch-case yapısının da kullanılacağı  
senaryolarda ng-template tercih edilebilir.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <div [ngSwitch]="value">
      <div *ngSwitchCase="1">Bir</div>
      <div *ngSwitchCase="2">İki</div>
      <div *ngSwitchCase="3">Üç</div>
      <div *ngSwitchCase="4">Dört</div>
    </div>
  `,
  imports: [NgSwitch, NgSwitchCase]
})
export class AppComponent {
  value: number = 3;
}
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <div [ngSwitch]="value">
      <ng-template [ngSwitchCase]="1">Bir</ng-t
      <ng-template [ngSwitchCase]="2">İki</ng-t
      <ng-template [ngSwitchCase]="3">Üç</ng-te
      <ng-template [ngSwitchCase]="4">Dört</ng-
    </div>
  `,
  imports: [NgSwitch, NgSwitchCase]
})
```





```
<ng-container [ngTemplateOutlet]="t" [ngTemplateOutletContext]="{data: 123}">
  Ng Container içeriği...
</ng-container>
<ng-template #t let-data="data">
  Ng Template içeriği...
  <br>
  {{data}}
</ng-template>
```

# ngTemplateOutlet Direktifi Nedir? Nasıl Kullanılır?

ng-container ile ayrılmış çeşitli bölgelere ng-template'in barındırdığı içerikleri paylaşmak için kullanılan structural bir direktiftir.

Eğer ngTemplateOutletContext ile bir çalışma yapılacaksa ngTemplateOutlet'i de bu şekilde kullanmalıyız!

```
int{
  or: 'app-root',
  lone: true,
  te:
  container *ngTemplateOutlet="contentTemplate">
    -container>
      template #contentTemplate>
        intent...
        -template>
        s: [NgTemplateOutlet]
  class AppComponent {
    number = 3;
```

Eğer birden fazla parametrik değer göndermek istiyorsanız yandaki gibi de çalışabilirsiniz...

```
<ng-container *ngTemplateOutlet="contentTemplate"
  [ngTemplateOutletContext]="{
    data1:{value:15},
    data2:{value:16},
    data3:{value:17}
  }">
</ng-container>
<ng-template #contentTemplate
  let-data1="data1"
  let-data2="data2"
  let-data3="data3">
  Content... <br>
  {{data1.value}}
  {{data2.value}}
  {{data3.value}}
```

ya da

```
<ng-container *ngTemplateOutlet="contentTemplate;
  context:{>
    data1:{value:15},
    data2:{value:16},
    data3:{value:17}}">
</ng-container>
<ng-template #contentTemplate
  let-data1="data1"
  let-data2="data2"
  let-data3="data3">
  Content... <br>
  {{data1.value}}
  {{data2.value}}
  {{data3.value}}
```

```
<ng-container *ngTemplateOutlet="contentTemplate"
  [ngTemplateOutletContext]="{data:{value:15}}">
</ng-container>
<ng-template #contentTemplate let-data="data">
  Content... <br>
  {{data.value}}
</ng-template>
```

Tabi tüm bu yapılanma alternatif olarak aşağıdaki gibi de tanımlanabilir.

ng-template'ten parametrik veri gönderebilmek için bu şekilde let-data keyword'u eşliğinde bir tanımda bulunulmalı ve içerisinde parametre adı belirtilmeli.

let-data içerisinde verilen değer her ne ise ngTemplateOutletContext için bu parametre adı olarak değerlendirilmekte ve JSON datasının ilk field'ı olarak yorumlanmaktadır.



“

\$implicit keyword'ü sayesinde tüm parametreler için default değer tanımlayabilirsiniz.

```
</ng-container>
<ng-template #contentTemplate let-data let-default let-hilmi>
  Content...
  {{data}} <br>
  {{default}} <br>
  {{hilmi}}
</ng-template>
```

Göründüğü üzere bu parametrelerin hiçbirinde `let-x='...'` şeklinde bir tanımlamada bulunulmamıştır.

Dolayısıyla bu parametrelerle \$implicit keyword'ü ile tanımlanmış değer set edilecektir.

# \$implicit Kullanımı

Bizler \$implicit keyword'ünü genellikle tekil parametrelerin söz konusu olduğu durumlar için pratik olduğu düşüncesiyle tercih ediyoruz. Şöyle ki:

```
t({
  r: 'app-root',
  one: true,
  e:
  container [ngTemplateOutlet]="personTemplate"
  [ngTemplateOutletContext]="{ $implicit:persons }"
)
  template let-persons #personTemplate>
    li *ngFor="let person of persons"
      {{person.name}} {{person.surname}}
    /li
  template>
  : [NgTemplateOutlet, NgFor]
)
ass AppComponent {
  e: { name: string, surname: string }[] = [
    {e: "Gençay", surname: "Yıldız"}, {e: "Nevin", surname: "Yıldız"}, {e: "Gülşah", surname: "Yıldız"}, {e: "Elif", surname: "Yıldız"}]
```

Dikkat ederseniz 'persons' isimli bir parametre tanımlanmış ve bu parametreden gelen değer üzerinde işlem gerçekleştirılmıştır.

Halbuki 'persons' parametresine bir tanımlamada bulunulmadığı için buraya değer `ng-container`'da ki \$implicit'ten gönderilmiştir.

Ki \$implicit'in değeride 'persons' koleksiyonu olarak tanımlandığı için dolaylı yoldan bu koleksiyon ren...

Yani anlayacağınız, bu tarz bir manevratik kullanım amacıyla \$implicit keyword'ünü tercih edebiliyoruz.

Content...  
default value  
default value  
default value



“  
ngTemplateOutlet’ı kullanarak parent component’ten child component’e template’leri geçirilebilir ve istediğimiz yerlerde kullanabiliriz.

## Child Component’e Template Geçirmek

```
Component({  
  selector: 'app-child',  
  standalone: true,  
  template:  
    <div>  
      i component content  
    </div>  
    <div>  
      <ng-templateOutlet *ngTemplateOutlet="childTemplate"></ng-templateOutlet>  
    </div>  
  ,  
  imports: [NgTemplateOutlet]  
}  
  
class ChildComponent {  
  @Input() childTemplate: TemplateRef<HTMLElement>;  
}
```

Gördüğü üzere child component’te TemplateRef türünden bir @Input değişken tanımlanmış ve bu değişken ng-container’da ngTemplateOutlet olarak kullanılmaktadır.

Haliyle bu child component’in kullanılacağı parent component’te bu @Input değişkene yandaki gibi ng-template’i gönderebilir ve kullanabiliriz.

```
Component({  
  selector: 'app-root',  
  standalone: true,  
  template:  
    <ng-template #parentTemplate>  
      Parent component template content  
    </ng-template>  
    <div>  
      <app-child [childTemplate]="parentTemplate"></app-child>  
    </div>  
  ,  
  imports: [NgTemplateOutlet, ChildComponent]  
})  
export class AppComponent {  
}
```

# ViewChild – Read Option

ViewChild ile referans edilen öğeler birden fazla türle ilişkilendirilerek daha efektif bir şekilde yönetilebilmektedir.



Angular'da directive'ler işaretledikleri nesneleri kendi türleriyle kapsülemeektedir!

```
<input type="text" #txtInput appExample [(ngModel)]="data">
```

Yukarıdaki input'u ele alırsak eğer variable', verilmiş ve bunun yanında appExample ve ngModel direktifleri ile işaretlenmiştir.

Bu durumda Angular çalışmalarında şöyle düşünebilirsiniz. Bu input nesnesini ViewChild ile referans etmek istediğimiz vakit, bu nesnenin özyi itibariyle bir ElementRef olmasından dolayı ElementRef referansıyla, ngModel direktifi ile işaretlendiği için NgModel referansıyla, benzer şekilde appExample direktifi ile işaretlendiği için ExampleDirective referansıyla ve tüm bunların dışında bu nesne ne de olsa bir view container olduğu için de ViewContainerRef referansıyla temsil edilebilir...

```
@Component({
  selector: 'app-root',
  standalone: true,
  template:
    <input type="text" #txtInput appExample [(ngModel)]="data">

  imports: [ExampleDirective, FormsModule]
})
export class AppComponent {
  data: any;

  // ElementRef
  @ViewChild("txtInput", { static: true }) _txtInput_Default;
  // ElementRef
  @ViewChild("txtInput", { static: true, read: ElementRef }) _txtInput_ElementRef;
  // NgModel
  @ViewChild("txtInput", { static: true, read: NgModel }) _txtInput_NgModel;
  // ViewContainerRef
  @ViewChild("txtInput", { static: true, read: ViewContainerRef }) _txtInput_ViewContainerRef;
  // ExampleDirective
  @ViewChild("txtInput", { static: true, read: ExampleDirective }) _txtInput_ExampleDirective;

  ngAfterViewInit() {
    console.log(this._txtInput_Default);
    console.log(this._txtInput_ElementRef);
    console.log(this._txtInput_NgModel);
    console.log(this._txtInput_ViewContainerRef);
    console.log(this._txtInput_ExampleDirective);
  }
}
```

`read` parametresinin bir diğer işlevi ise child component'te provide edilmiş bir değer inject etmemizi sağlamasıdır.

```
@Component({
  selector: 'app-child',
  standalone: true,
  template: `
    <h3>Child Component</h3>
  `,
  providers: [{ provide: 'Example Provider', useValue: 'Example Value' }]
})
export class ChildComponent {
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <app-child #child></app-child>
  `,
  imports: [ChildComponent]
})
export class AppComponent {
  @ViewChild("child", { static: true, read: 'Example Provider' }) _exampleProvider: string;

  ngOnInit() {
    console.log(this._exampleProvider);
  }
}
```

# ViewChildren Nedir? Nasıl Kullanılır?

ViewChild sayfada ilgili öğeden kaç tane olursa olsun eşleşecek olan ilk nesneyi tek olarak referans etmektedi. ViewChildren ise verilen öğenin tümünü QueryList olarak elde etmemizi sağlayan bir decorator'dur.

ViewChild ile referans edilen öğeler birden fazla türle ilişkilendirilerek daha efektif bir şekilde yönetilebilmektedir.

## ViewChildren – Read Option

Nasıl ki ViewChild dekoratöründe read option'ı child component'in provider'ını inject etmemizi sağlamaktır, benzer mantıkla ViewChildren dekoratörü de child component'lerin provider'larını inject ederek kümülatif olarak elde etmemize imkan tanımaktadır.

```
template:  
  <h3>Child1 Component</h3>  
  
providers: [{ provide: 'Example Provider', useValue: 'Example Value - Child 1' }]  
  
import class Child1Component {  
  
template:  
  <h3>Child2 Component</h3>  
  
providers: [{ provide: 'Example Provider', useValue: 'Example Value - Child 2' }]  
  
import class Child2Component {  
  
template:  
  <h3>Child3 Component</h3>  
  
providers: [{ provide: 'Example Provider', useValue: 'Example Value - Child 3' }]  
  
import class Child3Component {
```

```
@Component({  
  selector: 'app-root',  
  standalone: true,  
  template: `  
    <div #div>1</div>  
    <div #div>2</div>  
    <div #div>3</div>  
  `,  
  imports: []  
)  
  
export class AppComponent {  
  @ViewChildren("div") _div: QueryList<ElementRef>;  
  
  ngAfterViewInit() {  
    console.log(this._div);  
  }  
}
```

```
▼ Object { _emitDistinctChangesOnly: true, dirty: false, _results: (3)  
[-], _changesDetected: true, _changes: null, length: 3, first: {}  
last: {} }  
  _changes: null  
  _changesDetected: true  
  _emitDistinctChangesOnly: true  
  ▶ _results: Array(3) [ {}, {}, {} ]  
  dirty: false  
  ▶ first: Object { nativeElement: div 0 }  
  ▶ last: Object { nativeElement: div 0 }  
  length: 3
```

ViewChildren'da static parametresi olmadığı için referans ettiği öğelere ngOninit event'i içerisinde erişemeyeceğini ifade etmeye yardımcı var.

# QueryList – changes Property'si

`ViewChildren` dekoratörü neticesinde `QueryList` türünden elde edilen öğelerin sayfaya eklendiğine dair bilgi edinebilme için `changes` property'sine subscribe olarak süreci takip edebiliriz.

```
Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h3 #element>A</h3>
    <h3 #element *ngIf="_visible">B</h3>
    <h3 #element *ngIf="_visible">C</h3>

    <button (click)="visible()">Show/Hide</button>
  `,
  imports: [NgIf]
})
export class AppComponent {
  _visible: boolean;
  visible() {
    this._visible = !this._visible;
  }

  @ViewChildren("element") elements: QueryList<ElementRef>;
  ngAfterViewInit() {
    this.elements.changes.subscribe({
      next: data => console.log(data)
    })
  }
}
```

Misal olarak aşağıdaki kodu incelerseniz `ngIf` direktifi ile html öğelerinin visible özelliğine müdahale edilmekte ve her değişiklik durumu için `change` property'si takip edilmektedir.

Son olarak;

Bir component yahut herhangi bir direktifle işaretlenmiş HTML nesnesi direkt olarak component ya da direktif class'ı tarafından aşağıdaki gibi referans edilebilmektedir.

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <app-child1></app-child1>
    <h3 appExample>merhaba</h3>
  `,
  imports: [Child1Component, ExampleDirective]
})
export class AppComponent {
  @ViewChild(Child1Component, { static: true }) _child1Component: Child1Component;
  @ViewChild(ExampleDirective, { static: true }) _appExample: ExampleDirective;

  ngAfterViewInit() {
    console.log(this._child1Component);
    console.log(this._appExample);
  }
}
```

# Renderer2 Nedir?

Angular'da Renderer2, HTML elementlerini manipüle etmek için kullanılabilir bir sınıfır.

Renderer2, özellikle Angular uygulamalarının platformlar arasında taşınabilir olmasını sağlayan bir sınıfır.

Bu sınıf, DOM manipülasyonunu yönetmek için standartlaşmış bir API sağlar ve böylece uygulamaların tarayıcıda, Web Worker'lar da veya başka platformlarda çalışmasını kolaylaştırır.

Yani Renderer2 kullanarak, tarayıcı dışında çalışan bir Angular uygulaması üzerinde DOM manipülasyonunu kolayca yönetebilirsiniz.

Peki Renderer2'yi nasıl kullanabiliriz?

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <div #exampleDiv> </div>
  `,
  imports: []
})
export class AppComponent {
  constructor(private renderer: Renderer2) { }
  @ViewChild('exampleDiv', { static: true }) _exampleDiv: ElementRef;
  ngOnInit() {
    this.renderer.setProperty(this._exampleDiv.nativeElement,
      'innerHTML',
      'Merhaba');
    this.renderer.setStyle(this._exampleDiv.nativeElement,
      'color',
      'green');
  }
}
```

Peki Renderer2 ile yeni bir element nasıl oluşturabiliriz?

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

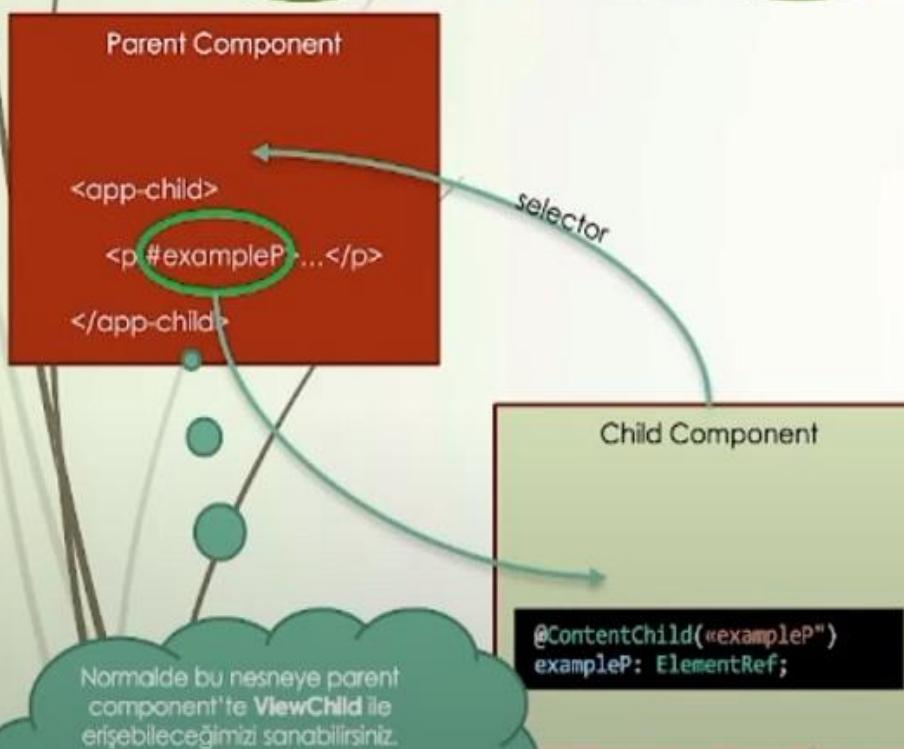
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <div #exampleDiv> </div>
  `,
  imports: []
})
export class AppComponent {
  constructor(private renderer: Renderer2, private element: ElementRef) { }
  @ViewChild('exampleDiv', { static: true }) _exampleDiv: ElementRef;
  ngOnInit() {
    const div = this.renderer.createElement("div");
    const text = this.renderer.createText("Merhaba");
    this.renderer.appendChild(div, text);
    this.renderer.appendChild(this._exampleDiv.nativeElement, div);
  }
}
```

# ContentChild & ContentChildren Nedir?

ContentChild ve ContentChildren decorator'ları ViewChild ve ViewChildren decorator'larine çok benzemektedir.

Component içerisindeki herhangi bir DOM nesnesini component class'ında referans alabilmek için ViewChild veya ViewChildren'i kullanıyoruz.

ContentChild ve ContentChildren decorator'ları ise ng-content içerisindeki herhangi bir nesneyi 'child' component class'ında referans etmek için kullandığımız decorator'lardır.



```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h3>App Component</h3>
    <app-child>
      <input type="text" #exampleInput>
    </app-child>
  `,
  imports: [Child1Component]
})
export class AppComponent { }
```

```
@Component({
  selector: 'app-child1',
  standalone: true,
  template: `
    <h3>Child1 Component</h3>
    <ng-content></ng-content>
  `,
  providers: [{ provide: 'Example Provider', useValue: 'Example Value - Child 1' }]
})
export class Child1Component {
  @ContentChild('exampleInput') exampleInput: ElementRef;
  ngAfterContentInit() {
    this._exampleInput.nativeElement.value = "merhaba"
  }
}
```

# Angular Global CSS Styles

Her web uygulamasında olduğu gibi Angular ile geliştirilmiş uygulamalarda da CSS ile çalışma gerçekleştirilmemesi elzemdir diyebiliriz. Haliyle uygulama genelinde global olarak kullanmak isteyeceğimiz harici CSS dosyaları söz konusu olabilmektedir.

Bu dosyalan uygulamaya yükleyebilmek için iki farklı yöntemden istifade edebiliriz. Şimdi gelin bu yöntemleri inceleyelim:

angular.json Dosyası Üzerinden CSS Yükleme

```
{
  "architect": {
    "build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
        "outputPath": "dist/ex",
        "index": "src/index.html",
        "main": "src/main.ts",
        "polyfills": [
          "zone.js"
        ],
        "tsConfig": "tsconfig.app.json",
        "inlineStyleLanguage": "scss",
        "assets": [
          "src/favicon.ico",
          "src/assets"
        ],
        "styles": [
          "src/styles.scss"
        ],
        "scripts": []
      }
    }
  }
}
```

index.html Dosyası Üzerinden CSS Yükleme

```
<!doctype html>
<html lang="en">

  <head>
    <meta charset="utf-8">
    <title>Example</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css">
  </head>

  <body>
    <app-root></app-root>
  </body>

</html>
```

# View Encapsulation

View Encapsulation, template'te tanımlanmış olan style'ların uygulanmanın diğer bölgelerini nasıl etkileyeceğini ayarlamamızı sağlamaktadır.

Bir component'te tanımlanmış olan bir CSS kurallarının başka bir component'te ki kuralları geçersiz kılmasını sağlamak oldukça önem arz etmektedir.

Angular bunu View Encapsulation stratejileri ile kontrol altına almaktadır.

Yani anlayacağınız View Encapsulation; uygulamanın herhangi bir component'inde tanımlanan style'ları diğer bölgelere karşı etkisini belirttiğimiz stratejilerin isimsel karşılığıdır.

View Encapsulation'in ViewEncapsulation.None, ViewEncapsulation.Emulated ve ViewEncapsulation.ShadowDOM olmak üzere üç farklı stratejisi mevcuttur.

Şimdi gelin bu stratejileri incelemeye başlayalım;

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-component1',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      Component 1 içerik
    </p>
  `,
  styles: ["p {color:red}"],
})
export class Component1Component { }
```

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-component2',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      Component 2 içerik
    </p>
  `,
})
export class Component2Component { }
```

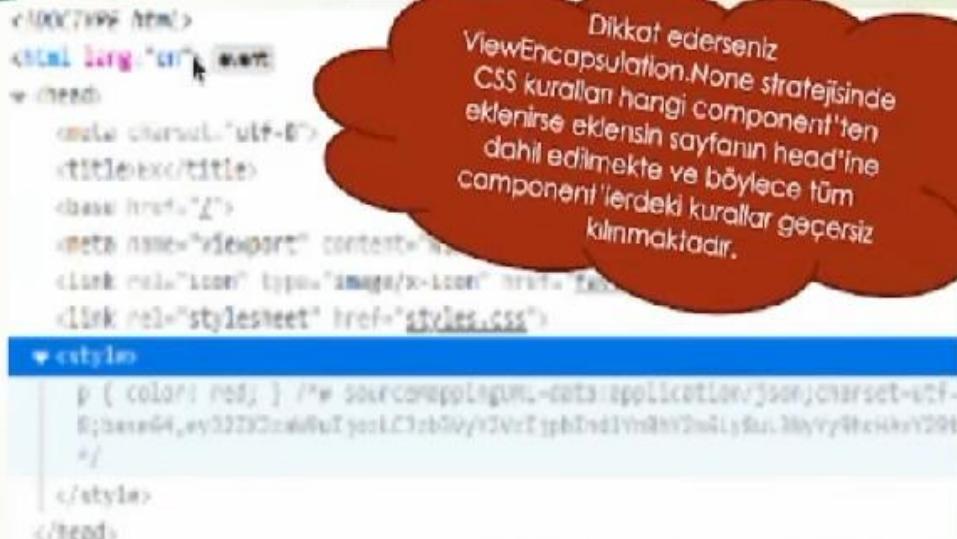
```
<app-component1></app-component1>
<app-component2></app-component2>
```

## ViewEncapsulation.None

Herhangi bir kapsülleme istenmediği durumlarda kullanılır. Böylece tek bir component'te kullanılan CSS kuralı sayfada yükü olan diğer component'leri de etkileyecektir.

```
@Component({
  selector: 'app-component1',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      Component 1 içerik
    </p>
  `,
  styles: ["p {color:red}"],
  encapsulation: ViewEncapsulation.None
})
```

Dikkat ederseniz ViewEncapsulation.None stratejisinde CSS kuralları hangi component'ten eklenirse eklensin sayfanın head'ine dahil edilmekte ve böylece tüm component'lerdeki kurallar geçersiz kılınmaktadır.



### **ViewEncapsulation.Emulated**

Bu strateji ile component'teki CSS kurallarını kapsüleme için bir işaretleme yöntemi uygulanır.

```
@Component({
  selector: 'app-component1',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      Component 1 içerik
    </p>
    ,
    styles: ["p {color:red}"],
    encapsulation: ViewEncapsulation.Emulated
  `)
```

ViewEncapsulation.Emulated neticesinde artık bir component'te uygulanan CSS kuralının diğer component'lere yayılmadığını görebilirsiniz.

### **ViewEncapsulation.ShadowDOM**

Bu stratejide, tarayıcı tarafından uygulamanın DOM nesnesinin dışında aynı bir Shadow DOM nesnesi oluşturulur. Bu Shadow DOM nesnesi, ana DOM nesnesinden etkilenmez ve tüm özellikleri, state'i ve CSS kuralları gizli kalır. Haliyle bu strateji kullanıldığı taktirde şöyle bir görüntüyle karşılaşılmaktadır.

```
@Component({
  selector: 'app-component1',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      Component 1 içerik
    </p>
    ,
    styles: ["p {color:red}"],
    encapsulation: ViewEncapsulation.ShadowDom
  `)
```

```
<html lang="en"> <head>
  <meta charset="utf-8">
```

# Style Binding

ar mimarisinde, style özelliklerini binding mekanizmasıyla yapılandırılmaktadır.

```
template:  
  <p [style.color] = "color"  
    Component 1 içerik  
  </p>
```

Yandaki örnekte görüldüğü üzere style binding syntax'ı özetlenmiştir. Dikkat ederseniz style binding işlemini gerçekleştirmek için köşeli parantez kullanılmakta ve hangi özelliğe değer atanacağına önce style keyword'ü vererek ardından özellik(color) belirtilmektedir ve son olarak bu özelliğe gelecek değeri taşıyan değişken tırnak içerisinde atanmaktadır.

Burada color yerine hangi style özelliği yapılandırılacaksa onu yazmanız yeterli olacaktır.

Misal:

```
template:  
  <p [style.backgroundColor]  
    Component 1 içerik  
  </p>
```

gibi...

## Koşullu Style Binding

ama sürecinde bazen koşullu şekilde style'ları uygulamak gerektiğinde kalabiliriz. Angular, bu davranışını style binding ile desteklemektedir.

```
@Component({  
  selector: 'app-component1',  
  standalone: true,  
  imports: [CommonModule],  
  template:  
    <p [style.backgroundColor] = "status == 'error' ? 'red' : 'blue'">  
      Component 1 içerik  
    </p>  
})  
export class Component1Component {  
  status: string = "error";  
}
```

Örnekte görüldüğü üzere koşulu ilgili elementte direkt uygulayabilir ya da yandaki gibi fonksiyon tabanlıda çalışma gerçekleştirebilirsiniz.

```
@Component({  
  selector: 'app-component1',  
  standalone: true,  
  imports: [CommonModule],  
  template:  
    <p [style.backgroundColor] = "getColor()">  
      Component 1 içerik  
    </p>  
})  
export class Component1Component {  
  status: string = "error";  
  
  getColor(): string {  
    return this.status == 'error' ? 'red' : 'blue';  
  }  
}
```

Tabii bu style'ların adlarını ömeklerde gösterildiği gibi camelCase olarak yazmak yerine dash-case olarak da belirtebilirsiniz.

## Birim Ayarlama

```
template:  
  <button [style.fontSize.px] = "48px">
```

# Class Binding

Angular mimarisinde, style özelliğinde olduğu gibi class'larda da binding mekanizması kullanılabilir.

Class binding ile HTML öğelerine class ekleyip, kaldırabilir ya da bunları koşullu olarak yönetebilirsiniz.

Angular, HTML öğelerine sınıf eklemek veya kaldırmak için üç farklı yol sunmaktadır. Şimdi gelin bu yöntemleri inceleyelim...

Tüm bunların dışında önceden direktiflerde incelediğimiz `ngClass` direktifi ile de class'ları dinamik bir şekilde uygulayabilirsiniz.

## ClassName İle Class Binding

```
template:  
  <button [className]="'button'">  
    Button  
  </button>  
  
,  
styles: [".button{color:red}"]
```

## Şartlı Olarak Class Binding

```
@Component({  
  selector: 'app-component1',  
  standalone: true,  
  imports: [CommonModule],  
  template:  
    <button [className]="status == 'error' ? 'errorButton' : 'successButton'">  
      Button  
    </button>  
  ,  
  styles: [".errorButton{color:red}", ".successButton{color:green}"]  
)  
export class Component1Component {  
  status: string = "error";  
}
```

## HTML Class İle Class Binding

```
template:  
  <button class="button">  
    Button  
  </button>  
  
,  
styles: [".button{color:red}"]
```

Ayrıca şartlı class uygulamasını şu şekilde de gerçekleştirebilirsiniz.

```
template:  
  <button [class.errorButton]="true">  
    Button  
  </button>  
  ,  
  styles: [".errorButton{color:red}", ".successButton{color:green}"]
```

Bir Angular uygulamasına bootstrap yükleyebilmek için öncelikle bootstrap kütüphanesinin npm tarafından uygulamaya çekilmesi gerekmektedir.

Devamında ise dist içerisindeki css ve js dosyalarının angular.json dosyasında uygun yerlere referans edilmesi gerekmektedir.

Ardından yüklenen kütüphanenin node\_modules dosyasından fiziksel dosyalarına erişimi gerekmektedir.

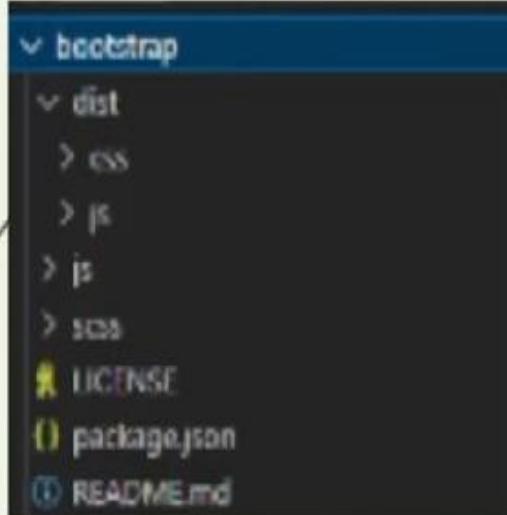
Bu aşamalar neticesinde artık Angular uygulamasında bootstrap kullanılabilir vaziyette olacaktır.

Adm I

**npm i bootstrap**

## Adım 2

Adım 3



Not !

angular.js dosyasında  
yapılan değişikliklerin  
uygulamaya yansımıası için  
tekrardan serve etmeniz  
gerekliğiniz unutmayın!

```
 30           "src/assets"
 31     ],
 32   "styles": [
 33     "node_modules/bootstrap/dist/css/bootstrap.min.css"
 34   ],
 35   "scripts": [
 36     "node_modules/bootstrap/dist/js/bootstrap.min.js"
 37   ]
 38 }
```

## Angular Uygulamasına Bootstrap Nasıl Yüklenir?

# APP\_INITIALIZER Nedir?

APP\_INITIALIZER, Angular tarafından sağlanan built-in bir injection token'dır.

Bu token sayesinde uygulama çalışmaya başlamadan önce geliştiriciler tarafından bazı kodlar yürütülebilimtedir.

Bu kodlar; kimlik doğrulaması, environment yapılandırması yahut öncül veri yüklemeleri vs. gibi davranışlar olabilir.

Angular, APP\_INITIALIZER tarafından sağlanan tüm işlevleri çalıştırıp sonuçlanana kadar uygulama başlangıcı bekleyecektir.

```
import { APP_INITIALIZER, importProvidersFrom } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { HttpClientModule, HttpClient } from "@angular/common/http";

bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule),
    {
      provide: APP_INITIALIZER,
      useFactory: (httpClient: HttpClient) => {
        return httpClient.get("https://jsonplaceholder.typicode.com/posts")
      },
      deps: [HttpClient]
    }
  ]
})
```

# Angular Runtime Configuration

Biliyorsunuz ki çoğu uygulama, uygulama ayağa kaldırılırken başlangıçta yüklenmesi gereken runtime yapılandırma bilgilerine ihtiyaç duyabilmektedir.

Angular'da, yapılandırma verilerini tutabilmek için environment variable'lar mevcuttur.

Örneğin; src/app/assets/config dizininde yapılandırma dosyasını tutabilir ve bu dosyayı JSON ya da XML formatında, yandaki gibi oluşturabiliriz.

Peki yapılandırmalar  
ne zaman  
okunmalıdır?

Bazı yapılandırma bilgilerini uygulamanın ilk sayfası yüklenmeden önce okumak gerekmektedir. Bunun için biraz önce gördüğümüz APP\_INITIALIZER token'ından aşağıdaki gibi faydalanabiliriz.

Misal olarak; uygulamanın veri ihtiyacı olduğunda kullandığı endpoint'ler test, pre-prod ve prod ortamlarında farklılık arz edebilmektedir.

İşte bu farklı duruma istinaden gerekli olan yapılandırmayı, runtime'da nasıl konfigüre edebileceğimizi inceliyor olacağız.

```
{  
  "appTitle": "Test App",  
  "apiEndpoint": {  
    "endpoint1": "https://jsonplaceholder.typicode.com/posts",  
    "endpoint2": "https://jsonplaceholder.typicode.com/comments"  
  },  
  "appSettings": {  
    "config1": "value1",  
    "config2": "value2"  
  }  
}
```

```
bootstrapApplication(AppComponent, {  
  providers: [  
    importProvidersFrom(HttpClientModule),  
    {  
      provide: APP_INITIALIZER,  
      useFactory: (httpClient: HttpClient) => {  
        httpClient.get("./assets/config/appConfig").  
          subscribe(configs => console.log(configs));  
      },  
      deps: [HttpClient]  
    }  
  ]  
})
```



# Örnek Runtime Configuration

```
{  
  "appTitle": "Test App",  
  "api1Endpoint": {  
    "endpoint1": "https://jsonplaceholder.typicode.com/posts",  
    "endpoint2": "https://jsonplaceholder.typicode.com/comments"  
  },  
  "appSettings": {  
    "config1": "value1",  
    "config2": "value2"  
  }  
}
```

Bunu modelleyen bir arayüz  
ya da model oluşturuyoruz.

```
export interface IAppConfig {  
  appTitle: string;  
  api1Endpoint: {  
    endpoint1: string,  
    endpoint2: string,  
  };  
  appSettings: {  
    config1: string,  
    config2: string  
  }  
}
```

```
@Component({  
  selector: 'app-root',  
  template: '<button (click)="btnClick()">Click</button>',  
  standalone: true  
)  
export class AppComponent {  
  btnClick() {  
    console.log(AppConfigService.settings);  
  }  
}
```

```
bootstrapApplication(AppComponent, {  
  providers: [  
    importProvidersFrom(HttpClientModule),  
    {  
      provide: APP_INITIALIZER,  
      useFactory: (appConfigService: AppConfigService) => {  
        return appConfigService.loadConfigurations();  
      },  
      deps: [AppConfigService]  
    }  
  ]  
})
```

İhtiyaç noktalarında  
yapılandırmalar  
okuyoruz.

APP\_INITIALIZER token'i  
ekliyoruz.

```
@Injectable({  
  providedIn: 'root'  
)  
export class AppConfigService {  
  static settings: IAppConfig;  
  constructor(private httpClient: HttpClient) {}  
  
  loadConfigurations() {  
    const configFile = 'assets/config/appConfig.json';  
  
    return new Promise<void>((resolve, reject) => {  
      this.httpClient.get(configFile).subscribe({  
        next: configs => {  
          AppConfigService.settings = <IAppConfig>configs;  
          console.log(AppConfigService.settings);  
          resolve();  
        }  
      })  
    })  
  }  
}
```

Konfigürasyonu okuyan  
servis oluşturuyoruz

# Angular Environment Variables

Çoğu uygulama production ortamına geçmeden önce developer, test, pre-prod ve production ortamlarından geçmektedir.

Yazılımsal açıdan tüm bu ortamlar farklı bir çevredir(environment) ve her bir çevre farklı kurulum konfigürasyonları ve yapılandırmalar gerektirebilmektedir.

Environment variable'lar; web uygulamalının bulunduğu ortama göre değeri değiştirebilen değişkenleridir. Bu değişkenler sayesinde uygulamanın ortamına göre bazı davranışlarını değiştirebilmekte veya yönetebilmektedir.

Genellikle developer, test ve production ortamlarındaki endpoint'ler farklı olacağından dolayı bunların ortamına göre yönetimini environment'lar üzerinden gerçekleştirilebilir.

## Environment Variable'lar Nerede?

Angular 15'e kadar environment variable'ları tanımlayamamak için hali hazırda bir environment.ts dosyası geliyordu. Versiyon 15'ten sonra artık bu dosya default olarak gelmemekte ve ihtiyaç doğrultusunda aşağıdaki talimat neticesinde manuel oluşturulmaktadır.

ng environments

```
PS C:\Users\1885451880\Desktop\plex> ng g environments
CREATE src/environments/environment.ts (31 bytes)
CREATE src/environments/environment.development.ts (31 bytes)
UPDATE angular.json (3118 bytes)
```

angular.json dosyasında 'fileReplacements' özelliğine ilgili environment dosyaları eklenmiştir.

```
"fileReplacements": [
  {
    "replace": "src/environments/environment.ts",
    "with": "src/environments/environment.development.ts"
  }
]
```

Böylece environment variable'larınn dosyalarını oluşturmuş oluyoruz.

## Environment Variable'lar Nasıl Oluşturulur?

Yeni bir environment variable oluşturmak oldukça basittir. Aşağıdaki gibi herhangi bir değişkeni tüm environment dosyalara eklemeniz yeterli olacaktır.

environment.development.ts

```
export const environment = {
  production: false,
  apiEndPoint : "https://dev.filanca.com"
};
```

## Environment Variable'lar Nasıl Okunur?

Environment variable'ları okuyabilmek için sade ve sadece environment.ts dosyasını (environment.development.ts gibi başka ortam dosyalarının değil) import etmeniz yeterlidir.

```
import { environment } from 'src/environments/environment';

@Component({
  selector: 'app-root',
  template: `<button (click)="btnClick()">Click</button>`,
  standalone: true
})
export class AppComponent {
  btnClick() {
    console.log(environment.apiEndPoint);
  }
}
```

## Environment Variable'lar Nasıl Test Edilir?

Normal şartlarda **ng serve** uygulamayı varsayılan olarak production ortamında ayağa kaldıracaktır. Uygulamayı farklı bir ortamda ayağa kaldırabilmek için aşağıdaki talimatı verebilirsınız.

```
ng serve --configuration='production'
```

# HttpClient Kütüphanesini Kullanırken Yardımcı HttpParams Metotları

## Set Fonksiyonu

İstek gönderilecek URL'e parametre eklememizi sağlar.

```
getPosts() {  
  const params = new HttpHeaders()  
    .set("id", 1)  
    .set("sort", "asc");  
  
  this.httpClient.get("https://jsonplaceholder.typicode.com/posts", { params  
});  
}
```

HttpParams nesnesi immutable bir nesnedir. Yani değeri değişmez bir nesnedir. Bu sebepten dolayı aşağıdaki gibi bir kullanım geçerli olmayacağındır.

```
const params = new HttpHeaders();  
params.set("id", 1)  
params.set("sort", "asc");
```

Burada set fonksiyonuyla yapılan her bir tetikleme mevcut instance üzerinde bir degersel değişiklik yaratmaz, bilakis yeni instance üreterek onu döndürür.

Haliyle bu parametrelerin nesneye eklenmesini istiyorsanız eğer ya yukarıdaki gibi nesne new operatörüyle oluşturulduğu anda set fonksiyonu çağrılmalı ya da aşağıdaki gibi bir kullanım gerçekleştirilmelidir.

```
let params = new HttpHeaders();  
params = params.set("id", 1)  
params = params.set("sort", "asc");
```

## Append Fonksiyonu

URL'e birden fazla aynı isimde parametre eklenmesini sağlar.

```
let params = new HttpHeaders()  
  .set("id", 1)  
  .append("id", 2)  
  .set("sort", "asc");
```

## Has, Get ve GetAll Fonksiyonları

Has : Belirtilen parametre adı karşılığında değer olup olmadığını söyler.  
Get : Verilen key'e karşılık ilk parametre değerini getirir.  
GetAll : Verilen key'e karşılık tüm parametre değerlerini getirir.

```
let params = new HttpHeaders()  
  .set("id", 1)  
  .set("sort", "asc");  
  
if (params.has("id"))  
  params = params.append("id", 2)  
  
console.log(params.get("id"));  
console.log(params.getAll("id"));
```

# HttpParams'ın fromString ve fromObject Özellikleri

fromString

URL parametrelerini string olarak iletmeyenin pratik yoludur.

fromObject

URL parametrelerini object olarak iletmeyenin pratik yoludur.

```
let params = new HttpParams({fromString: `id=1&sort=asc`})
```

```
let params = new HttpParams({ fromObject: { id: 1, sort: "asc" } });
```

Tabi tüm bunların dışında HttpParams kullanmaksızın da URL parametrelerini aşağıdaki gibi kullanabilirsiniz.

```
this.httpClient.get("https://jsonplaceholder.typicode.com/posts?id=1&sort=asc")
```

# HttpHeaders

HTTP protokolü üzerinden yapılan iletişim sürecinde client ve server birbirlerine HTTP Header üzerinden ek bilgi paylaşabilmektedirler.

Genellikle yetkilendirme gerektiren endpoint'lere erişim sürecinde, yetkiyi temsil eden token değeri, server'a bu Headers kısmından iletilmektedir.

Tabi bunun dışında istenilen veriyi Headers üzerinden server'a ya da server'dan da client'a iletebilirsiniz.

```
const headers = new HttpHeaders()
  .set("name", "gencay")
  .set("community", "ngakademi");

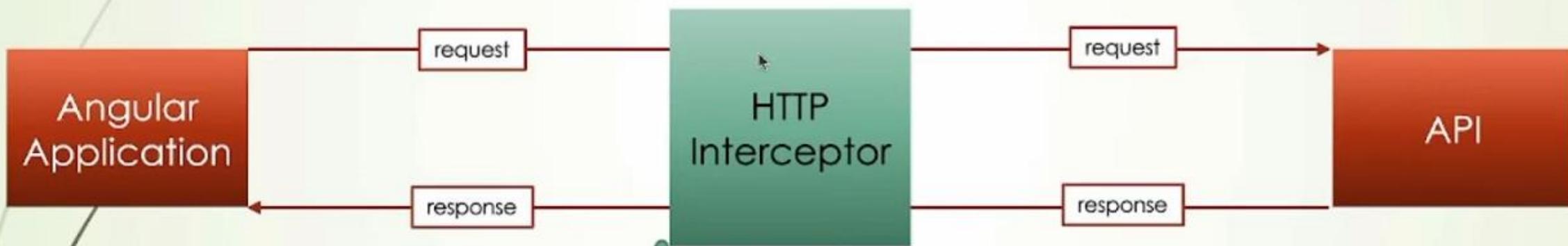
this.httpClient.get("https://localhost:53066/api/ozet/x", { headers: headers })
  .subscribe({
    next: (data: any) => {
      console.log(data.headers.keys());
    },
    error: error => {
      console.log(error);
    }
});
```

# Http Interceptor Nedir?

HTTP Interceptor, Angular'da yapılan HTTP isteklerinde merkezi bir nokta üzerinden işlemler yapmamızı sağlayan bir mekanizmadır.

Bu mekanizma sayesinde, uygulama bazı yapılan HTTP request'leri yakalanarak yapılacak request'e dair müdahalede bulunulabilmektedir.

Ayrıca bir tek request'e değil, bu request neticesinde gelecek olan response'a da merkezi olarak müdahale etme şansı tanımaktadır.



İşte bu interceptor sayesinde, uygulama seviyesinde olan herhangi bir HTTP request yakalanacak ve merkezi olarak manipüle edilme şansı söz konusu olacaktır.

İyi de hoca, bu interceptor'a nerde, hangi durumlarda ihtiyacımız oluyor ki?

HTTP Interceptor'a birçok nedenden dolayı ihtiyacımız olabilir. Lakin bu nedenlerden öyle biri vardır ki en çok bu ihtiyaç için interceptor biçimliş kaftanıdır diyebiliriz.

Böylece kodu, her istek sürecinde, request'e access token'ı dahil etme maliyetinden törpülemiş olacak ve bu işlem için merkezi bir nokta sağlayan interceptor'dan istifade etmiş olacağız.

Bu neden de, yetkilendirme gerektiren API'lara istek süreçlerinde kullanılacak Access Token değerinin ilgili request'in header'na eklenmesi gereklidir.

# Http Interceptor Oluşturma

HTTP Interceptor oluşturabilmek için Angular CLI Üzerinden aşağıdaki talimatları verebilirsiniz.

```
ng g interceptor [name]
```

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class MyInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```

Gördüğü üzere bu fonksiyonun 'request' parametresi yakaladığı 'request'i bizlere getirmektedir.

Bir sınıfın interceptor olabilmesi için **HttpInterceptor** arayüzünden implemente edilmesi gerekmektedir.

Bu arayüz request ve respons'ları yakalayıp, işlem yapmamızı sağlayacak olan intercept fonksiyonunu uygulatmaktadır.

next parametresi ise request'in devam etmesini sağlayacak olan bir handler görevi görmektedir. Yani araya girip, yapacaklarınıza yaptıktan sonra, next parametresi ile isteği salar, devam etmesini sağlaz.

# Http Interceptor Kullanımı

Oluşturulan HTTP Interceptor'ı kullanabilmek için uygulama seviyesinde `HTTP_INTERCEPTORS` injection token'ına karşılık provide edilmesi yeterlidir.

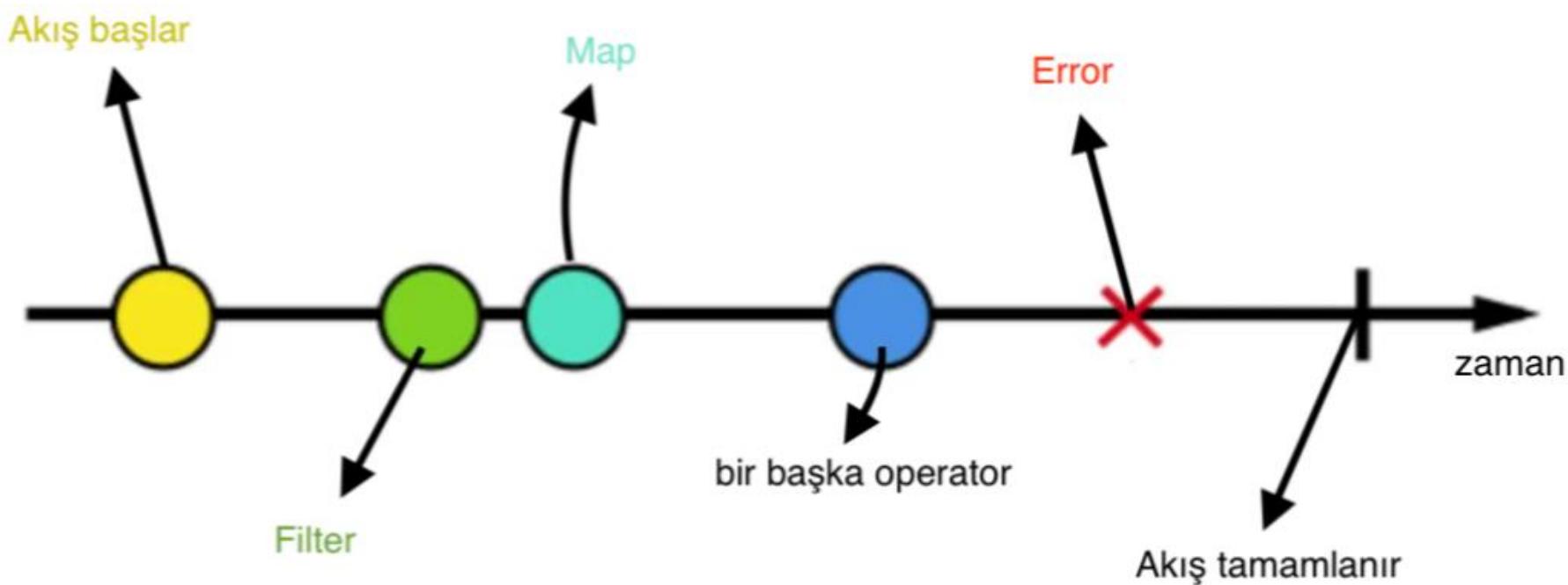
```
bootstrapApplication(AppComponent, {  
  providers: [  
    importProvidersFrom(HttpClientModule),  
    { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true }  
  ]  
})
```

```
export const httpInterceptorProviders = [  
{  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthExpiredInterceptor,  
  multi: true,  
},  
{  
  provide: HTTP_INTERCEPTORS,  
  useClass: ErrorHandlerInterceptor,  
  multi: true,  
},  
{  
  provide: HTTP_INTERCEPTORS,  
  useClass: AppHttpInterceptor,  
  multi: true,  
}  
];
```

```
providers: [  
  { provide: ErrorHandler, useClass: AppErrorHandler }, httpInterceptorProviders,
```

# Observables

- inlenilen obje'ye Subject ve dinleyenlerede Observer dersek. Observer subject'e abone olabilir(subscribe). Bir subject'e birden fazla observer abone olabilir. Bu da one-to-many relationship yapıya örnek olur. Subject'de herhangi bir durum değişikliği tüm abonelerine bildirilir.

























```

//services.AddScoped<ConsoleLog>();
//services.AddScoped<ConsoleLog>(p => new ConsoleLog(5));

//services.AddTransient<ConsoleLog>();
//services.AddTransient<ConsoleLog>(p => new ConsoleLog(5));

//services.AddScoped<ILog>(p => new TextLog());
services.AddScoped<ILog, ConsoleLog>(p => new ConsoleLog(5));

services.AddControllersWithViews();
}

```

```

0 references
public class HomeController : Controller
{
    //readonly ILog _log;

    //public HomeController(ILog log)
    //{
    //    _log = log;
    //}

    0 references
public IActionResult Index([FromServices]ILog log)
{
    log.Log();
    return View();
}

```

The screenshot shows a code editor with several tabs: Index.cshtml\*, ConsoleLog.cs, Startup.cs, HomeController.cs\*, and TextLog.cs. The HomeController.cs\* tab is active, displaying the following code:

```

1 @inject dependency_injection.Services.Interfaces.ILog log
2
3

```

Built-in IoC Container, içerisinde koymak istediği/nesneleri üç farklı davranışla alabilmektedir.

Singleton	Scoped	Transient
Uygulama bazlı tekil nesne oluşturur. Tüm taleplere o nesneyi gönderir.	Her request başına bir nesne üretir ve o request pipeline'nında olan tüm isteklere o nesneyi gönderir.	Her request'in her talebine karşılık bir nesne üretir ve gönderir.

# Validation

- ▶ DataAnnotations
- ▶ MetadataTypes
- ▶ FluentValidation (Kütüphanesi ile)

# Validation (Server Taraflı)

- ▶ DataAnnotations
- ▶ MetadataTypes
- ▶ FluentValidation (Kütüphanesi ile)

# Fluent Validation

- Solid prensiplerinden Single Responsibility Principle(Tek sorumluluk ilkesine) dayalı olabilmesi için ;
  - Entitiy yada dto larda kullanılmaz.

```
✓ references
public class Startup
{
    ✓ references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews().AddFluentValidation(x => x.RegisterValidatorsFromAssemblyContaining<Startup>
        ());
    }
}
```

- Burada fluent validation eklenirken Startup Assembly de bulunan bütün validationları eklemesi gerektiğini bildiriyoruz.

# Fluent Validation

```
using FluentValidation;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace OrnekUygulama.Models.Validators
{
    public class ProductValidator : AbstractValidator<Product>
    {
        public ProductValidator()
        {
            RuleFor(x => x.Email).NotNull().WithMessage("Email boş olamaz!");
            RuleFor(x => x.Email).EmailAddress().WithMessage("Lütfen doğru bir email giriniz.");

            RuleFor(x => x.ProductName).NotNull().NotEmpty().WithMessage("Lütfen product name'i boş geçmeyiniz.");
        }
    }
}
```

# Validation (Client Tabanlı)

- ▶ Jquery
- ▶ JqueryValidate
- ▶ JqueryValidationUnobtrusive

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model OrnekUygulama.Models.Product
    <script src="~/jquery/jquery.min.js"></script>
    <script src="~/jquery-validate/jquery.validate.min.js"></script>
    <script src="~/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"></script>

    <div asp-validation-summary="All">
    </div>

    <form asp-action="CreateProduct" asp-controller="Product" method="post">
        <input type="text" asp-for="ProductName" placeholder="Product name" />
        <span asp-validation-for="ProductName"></span> <br />

        <input type="number" asp-for="Quantity" placeholder="Quantity" />
        <span asp-validation-for="Quantity"></span><br />

        <input type="email" asp-for="Email" placeholder="Email" />
        <span asp-validation-for="Email"></span> <br />
        <button>Gönder</button>
    </form>
```