1. How Angular App Starts and Works

Biz biliyoruz ki angular app tarayıcı üzerinde çalışıyor ve single page olarak çalışıyor. Sayfanın kaynağını görüntüleyince body içerisinde sadece bir app-root etiketi olduğunu görüyoruz ancak gerçekte sayfada bir çok eleman var.

```
</head>
<body>
  <app-root></app-root>
<script type="text/javascript" src="runtime.js"></script><script type="text/javascript" src="es2015-polyfills.js" nomodule></script><script type="text/javascript" src="polyfills.js"></script><script type="text/javascript" src="styles.js"></script><script type="text/javascript" src="vendor.js">
</script><script type="text/javascript" src="main.js"></script></body>
</html>
```

Bunu sağlayan şey sayfada import edilen js dosyaları böylece sadece tek bir index html yüklenmesine rağmen aynı index html üzerinde sayfa refresh edilmeden bir çok işlem yürütebiliyoruz.

Yani index.html'in contenti sayfa yüklendikten sonra javascript kullanılarak oluşturuluyor

Peki flow nasıl yani angular app içerisinde bir çok parça var bu parçalar nasıl yükleniyor olay nasıl oluyor?

- Aslında server'dan bilgisayarımıza yüklenen sayfa index.html sayfası. Bizim projemizdeki index.html içerisinde script importları bile yok bu importlar uygulama build edilirken cli tarafından otomatik olarak eklenir.
- Main.ts dosyası starting file'dır. Angular app başladığında ilk çalışacak file budur.

```
TS main.ts
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynam
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.log(err));
13
```

-
- Buradaki komutlarla angular kendi kendini çalışmaya başlatır. Burada root module olarak AppModule kullanılıyor ve parametre olarak geçiriliyor.
- ApModule'ü uygulamanın tüm parçalarının tanımlandığı ve birleştiği bir file olarak düşünebiliriz.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

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

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

- Body'si olmayan bir AppModule class'I görüyoruz, ancak bunun üzerinde bir @NgModule decorator'ı yer alıyor.
- Decorator bir typescript feature'ıdır ve arkaplanda ait olduğu class'ı manipüle eder.
- Sonuçta özel NgModule decorator'I kullanılarak normal şekilde tanımlanan AppModule class'ı angular tarafından yorumlanacak bazı metadata'ları olan bir module'e dönüştürür.
- Bir module'ün genelde yukarıda görülen 3 definition'ı olur.
    - **Declarations:** declares all the components we are using in our project.
    - **Imports:** Diğer module'leri import eder. Böylece ilgili module'e farklı functionalities kazandırmış oluruz. Fonksiyondan kastımız bazı hazır bundled components olabileceği gibi directives de olabilir. Örneğin yukarıda import edilen BrowserModule sayesinde app module içerisinde ngFor directive'ini kullanabilir hale geliyoruz.
    - **Providers:** servisleri provide etmek için kullanılır.
    - **Bootstrap:** root module'üne özel bir property'dir ve burada uygulamamızın main component'i belirtilir.

2. Components

Uygulamamızın bir çok component'ten oluştuğunu biliyoruz, örneğin root component olan app.component.ts'in içine bakalım:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Refresh';
}
```

- Component gördüğümüz gibi bir class'tan ve bu class'a ait bir @Component decorator'ından oluşur.
- Component decorator'ı sayesinde AppComponent class'ı sıradan bir class olmaktan öteye geçer ve eklenen metadata sayesinde angular bu class'I artık bir component olarak render eder.

Index html içerisinde çağırılabilen tek component root component'i olan app.component'tir, diğer componentler kendi içlerinde veya app.componentin view'i içerisinde çağırılır.

Uygulamada kullanılacak product yukarıdaki mantıkla yaratılır sonra da app module içerisinde import edilir ki uygulamamız component'i kullanabilsin. Elbette component'i terminalden yaratırsak işimiz daha rahat olur.

3. Template Syntax

Bildiğimiz gibi component'lerin html sayfaları var ve component class'ları ile view'leri bağlantılı. Bu sayede dinamik html sayfaları oluşturabiliyoruz.

```
<h1>My Products</h1>
<input type="text" [(ngModel)]="productName">
<button (click)="productName = 'A Tree'" [disabled]="isDisabled">Chang
<div>{{ productName }}</div>
```

Örneğin yukarıda ilk eleman statik bir h1 elemanıdır.

Buna karşın input elemanında ngModel directive'i ile two way binding yapılmış, artık input içerisinde yazılan değer anlık olarak productName değişkeninin içeriğini değiştiriyor benze şekiklde productName içeriği bir şekilde değişirse input'un içeriği de hemen etkileniyor.

Bu productName değişkeni aşağıda görüldüğü gibi component class'ı içerisinde tanımlanmış.

```
export class ProductsComponent {
  productName = 'A Book';
  isDisabled = true;

  constructor() {
    setTimeout(() => {
      // this.productName = 'A Tree';
      this.isDisabled = false;
    }, 3000);
  }
}
```

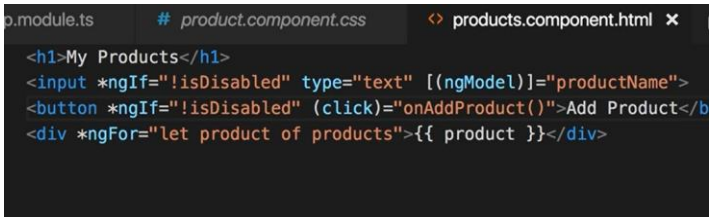
Benzer şekilde yukarıdak event binding de yapılmış, buton elmanına tıklandığında bir typescript kodu çalıştırılmış, burada tıklandığında bir fonksiyon da çağırılabilirdi.

Benzer şekilde property binding ile dom elemanlarının property'leri yine class'a bağlanmıştır. Bu konunun detayından zaten türkçe kursun ders notlarında bahsedilmişti.

Son olarak string interpolation ile direkt olarak class içeriisndeki değişkenlere ulaşılabileceğini de biliyoruz.

4. Custom Property & Event Binding

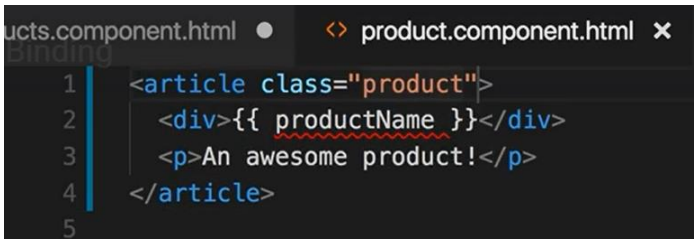
Bir **products** component'i oluşturmuştuk içerisinde bazı binding yöntemlerini ve directive'leri kullanmıştık, products view'I aşağıdaki gibiydi.



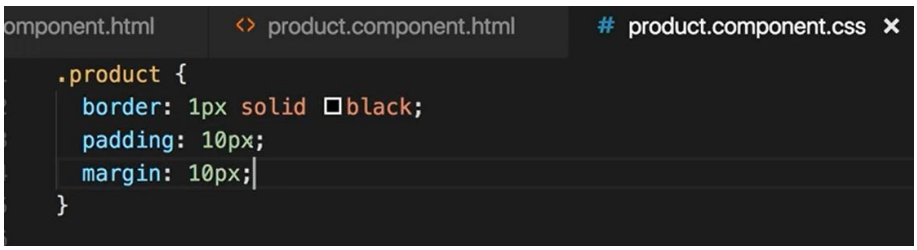
Burada ngFor ile bir products listesindeki product'ları tek tek yazdırmıştık, products ile temelde product isimlerinin yazılı olduğu bir bir string list olarak component içerisinde tanımlanmıştı.

Ancak ben product'ın basit bir stringden fazlasını içermesini isteyebilirim, product da kendi başına bir component olsun isteyebilirim.

Bu amaçla yeni bir **product** component'i yaratalım ve view'ini aşağıdaki gibi dolduralım:



Ayrıca bu component'in view'ine style da eklemek isteyebilirim bunun için de product componetinin css file'ını kullanırım:



Artık product component'inin view'ini ve style sayfasını doldurduk.

Ancak bir problemimiz var şuanda productName değişkeninin tanımlı olduğu component **product** component'i değil, tamamen bağımsız bir component olan **products** componenti, bu yüzden product view'i içerisinde productName değişkenine ulaşamıyoruz.

Product view içerisinde productName değişkenine ulaşabilmek için productName değişkenini component içerisinde tanımlamamız gerek ancak product componenti içerisinde tanımlanan bu property'e veri dışarıdan, products component'inden gelecek. Bu yüzden product component'i içerisinde aşağıdaki gibi bir tanımlama yapıyoruz, yani productName property'si bu component'e input olarak dışarıdan verilecek dedik.

Şimdi products.component.ts içini tekrar hatırlayalım, burada ürün isimlerini içeren bir products listesi var:

```
export class ProductsComponent implements OnInit {

  products = ['Book','Computer','Phone'];

  constructor() { }

  ngOnInit() {
  }

}
```

Bu liste products.component.html içerisinden ulaşılabilir:

Products view'i içerisinde aşağıdaki gibi bir yapı kullanarak, listediki her ürün için product component'ini çağırdım ve çağırırken de product component'i içerisindeki productName ismindeki @input dekoratörlü field'e products listesinden alınan değeri geçirdim:

```
<h1>My Products</h1>
<app-product *ngFor="let product of products" [productName]="product"></app-
product>
```

Burada yapılan olaya CUSTOM PROPERTY BINDING denir böylece bir component içerisinde diğer component'i çağırırken component'ler arası bir bilgi geçişi sağladık, yani products component'i product component'ini kendi verdiği productName değişkeni ile çağırıyor.

Burada products component'ini parent diğerini ise child component olarak düşünürsek, parent child'I kendi view'inde çağırdı ve bu sırada parent'tan child'a bir bilgi geçişi oldu.

Custom Event Binding

Şimdi bir diğer yöntem olan child component'ten event ile parent component'e bilgi göndermeyi göreceğiz.

Öncelikle child component içerisinde bild click event'i tanımlıyorum ve bir elemana tıklanınca onClicked() methodu çalışsın istiyorum:

```
<article (click)="onClicked()" class="product">
  <div>{{productName}}</div>
  <p>Awesome product!</p>
</article>
```

Commented [S1]: Özetle custom property binding şu: Bir componentin içerisine dışarıdan veri almak istersek, component içerisinde input decorator'I ile bir property tanımlıyoruz. Daha sonra bu component'in tag'ini nerede kullanıyorsak, bu tag içerisinde önceden tanımlanmış olan property'i sanki ilgili html elemanının bir property'si gibi kullanabiliyoruz.
[productName]="product" dediğimizde yapılan şey, products componentinde tanımlı product değerininin yerni tanımlanan product componentin içerisindeki productName propertysine atanmasıdır.

Yani componente, component tag'inin kullanıldığı yerden atama yapılabilecek bir property atamış oluyoruz. Böylece ilgili componente data girişi mümkün oluyor.

Custom property binding ile bir component'e tag'in kullanıldığı yerden bilgi getirebilirken, custom event binding ile ise bir componentten dışarı yani tag'in kullanıldığı yere bilgi taşımamız mümkün oluyor.

Output decorator'I ile kullanılan eventemitter, istenilen event gerçekleştiğinde dışarıya emit ediliyor, emit edilen event dışarıda yani tag'in kullanıldığı yerde, aynı (click)="" event'i gibi kullanılabiliyor, yani custom bir event kullanmış oluyoruz, componentten emit edilen event içerisinde taşınan bilgiye de $event ile ulaşabiliyoruz.

onClicked methodu da child component class'I içerisinde aşağıdaki gibi tanımlanıyor bu tanım için EventEmitter class'ını ve Output decorator'ını kullanıyoruz.

```
export class ProductComponent implements OnInit {
  @Input() productName: string;
  @Output() productClicked =  new EventEmitter();

  constructor() { }

  ngOnInit() {
  }

  onClicked(){
    this.productClicked.emit();
  }
```

Bu event child component içerisinde dinleniliyor ancak ben istiyorum ki child component içerisinde gerçekleşen event dışarıya yani parent component'ine taşınsın.

```
<h1>My Products</h1>
<app-
product (productClicked)="clkd(product)" *ngFor="let product of products" [pro
ductName]="product"></app-product>
```

Bunun için yukarıdaki gibi parent view'i içerisinde productClicked event'i tanımlıyorum ki bu event aslında child component içerisindeki elemana tıklanınca çağırılacak bir eventtir. Yani event'i dışarıya emit etmiş olduk.

Son olarak parent component içerisinde ilgili event gerçekleşince çağırılacak methodu yazıyorum:

```
})
export class ProductsComponent implements OnInit {

  products = ['Book','Computer','Phone'];

  clkd(prdct){
    console.log('Clicked on '+prdct);
  }
```

Sonuçta böylece parent view (products) üzerinde child view gösteriliyor ve bu sırada da bazı veriler parent component'ten child component'e @input decorator'I yardımıyla aktarıldı.

Aynı anda child component içerisinde tanımlanan bir click event'i parent component'e emit edildi. Ne zaman ki child component'in article elemanına tıklanıldı o anda sırasıyla parent component'ine ait bir clkd() methodu çalıştırılıyor.

5. Forms

Şimdi formlara bakacağız, zaten önceki kursta template-driven-forms başlığı altında formlarla ilgili bir çok işlemi yapmıştık buradaki mantık da benzer ama bana daha pratik geldi.

Aşağıdaki gibi bir form oluşturduk bir input var bir de button var, button'un tipi submit normalde butona basılınca server'a bir request gönderilir ve form submit edilir ancak biz burada bunu istemiyoruz.

App.module içerisinde FormsModule import edildiği için angular form element ile behind the scenes ilgilenecek ve submission'ı engelleyecek bunu sağlayan event de ngSubmit event'i. Sonuçta submit butonuna basılınca form'un ngSubmit event'i triggered edilmiş olacak.

Sonuçta angular yardımı ile form'un validity'sini vesaire öğrenebileceğiz. Bunu zaten template-driven-forms kısmında da ngModel ile yapmıştık.

```
<h1>My Products</h1>
<form (ngSubmit)="addProduct(f)" #f="ngForm">
  <input type="text" ngModel name="productName" required>
  <button type="submit">Add Product</button>
</form>
<app-
product (productClicked)="clkd(product)" *ngFor="let product of products" [pro
ductName]="product"></app-product>
```

ngModel diyerek input'u angular'a bir control olarak register etmiş oluyoruz, yani angular'a diyoruz ki: bu input'u ve onun value'sunu vesaire takip et.

Daha sonra aynı input elemanına bir name ekledik ve bir de required validator'ı ekledik ki bu html özelliği idi, farklı validators'ı da kullanabileceğimizi türkçe kurstan hatırlıyoruz.

Sonuçta angular ilgili input ile ilgili bir javascript object yaratacak ve bu input hakkında bilgileri tutacak, bu objeye component içerisinde ulaşmak için bir local reference tanımlıyoruz.

Sonuçta bu reference'a ngForm objesini verip addProduct methodu içinde bunu yollayarak component içerisinde ilgili input'un form control objesine ulaşabiliyorum.

Component içinde aşağıdaki gibi bir method tanımlarsam, form submit edildiğinde console'da form control objesi yazdırılacaktır.

```
addProduct(form){
  console.log(form)
}
```

Bu obje de aşağıdaki gibi olacak, buradan input'un durumu hakkında bilgiler edinip ona göre kontroller gerçekleştirebiliriz.

1. *NgForm {submitted: true, _directives: Array(1), ngSubmit: EventEmitter, form: FormGroup}*
    1. control: (...)
    2. controls: (...)
    3. dirty: (...)
    4. disabled: (...)
    5. enabled: (...)
    6. errors: (...)
    7. form: FormGroup {validator: null, asyncValidator: null, pristine: true, touched: true, _onCollectionChange: ƒ, …}
    8. formDirective: (...)
    9. invalid: (...)
    10. ngSubmit: EventEmitter {_isScalar: false, observers: Array(1), closed: false, isStopped: false, hasError: false, …}
    11. path: (...)
    12. pending: (...)
    13. pristine: (...)
    14. status: (...)
    15. statusChanges: (...)
    16. submitted: true
    17. touched: (...)
    18. untouched: (...)
    19. valid: (...)
    20. value: (...)
    21. valueChanges: (...)
    22. _directives: [NgModel]
    23. __proto__: ControlContainer

Bu noktadan sonra bazı style vesaire işlerini önceki kursta yapmıştık. Basit bir örnek olması açısından:

```
addProduct(form){
  if(form.valid) {
    this.products.push(form.value.productName)
  }
}
```

Dersek, eğer submit edildiğinde form valid ise component'in products listesine ilgili form'un içindeki value eklenecektir.

Bu değişiklik de ngFor ile hemen yakalanacak ve sayfaya yansıyacaktır.

6. Services & Dependency Injection

Services are typescript classes which can be reached from any component which to be precise can be injected into any component or actually also into other services.

Yeni bir **products.service.ts** dosyası yaratarak bir products service'i yaratalım:

```
export class ProductService {
    private products = ['A Book'];

    addProduct(productName: string){
        this.products.push(productName);
    }

    getProducts(){
        return [...this.products];
    }
}
```

Service'ler genellikle farklı component'lerle paylaşmak isteyebileceğimiz bazı functionalities barındırır. Bu diğer componentler tarafından kullanılmak istenen data olabileceği gibi, utility functions like logging something or sending a http request de olabilir.

Burada service bir products array'i tutsun istedik. Bu array ve methodlar diğer component'ler ile paylaşılacak.

addProduct basitçe içine girilen parametre ile yeni bir product oluşturuyor. Buna karşın getProducts methodu products arrayinin kopyasını return ediyor. Bunun sebebi şu products arrayi bir pointer olduğu için eğer getProducts ile bu array'i return edersek aslında ilgili pointer'ı return etmiş olacağız, o zaman o pointer üzerinde yapılan değişiklik orjinal products'ı da etkileyecek o zaman da products'ı private yapmamızın bir anlamı kalmayacak.

Sonuçta service'imizi yukarıdaki gibi oluşturduk. Artık bu service'i bir component'de veya bir başka service'te inject ederek kullanabiliriz.

Örneğin biraz evvel products component'i içinde aşağıdaki gibi bir addProduct methodu kullanmıştık, biz bu method içerisinde işimizi service'le halletmek istiyor olalım.

```
  addProduct(form){
    if(form.valid) {
      this.products.push(form.value.productName)
    }
  }
```

Bunu yapmak için oluşturulan ProductService'i component'e inject etmeliyiz, ancak bundan önce service'i **app.module.ts** içerisinde provide etmeliyiz:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { ProductsComponent } from './products/products.component';
import { ProductComponent } from './product/product.component';
import { FormsModule }   from '@angular/forms';
import { ProductService } from './products.service';

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

Şimdi ise service'i **products.component.ts** içerisinde inject edebiliriz:

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

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

  products = [];

  constructor(private productService: ProductService) {
    this.products = productService.getProducts();
  }

  ngOnInit() {
  }

  clkd(prdct){
    console.log('Clicked on '+prdct);
  }

  addProduct(form){
    if(form.valid) {
      this.productService.addProduct(form.value.productName);
    }
  }
}
```

Yukarıda kırmızı ile gösterilen kısımlar ProductService'i nasıl inject edeceğimizi gösteriyor. Inject edildikten sonra constructor içerisinde service'in getProducts methodu kullanılıyor böylece service'in products listesi component içerisine çekiliyor.

Benzer şekilde yine component içerisinde addProduct methodu kullanılarak component'ten service'e bilgi geçişi oluyor.

Burada yapılan işlem aynı component içinde service'den hem bilgi almak hem bilgi vermek oldu, belki bu pek mantıklı değil çünkü aynı işlemi o zaman component içinde bir products listesi ile de halledebilirdik ama service'in asıl güzelliği component'ler arası bilgi alışverişine izin vermesi, yani service tüm componentlerin ulaşabileceği ve bilgi alıpverebileceği bir yapı olarak görülebilir.

Aslında component içerisindeki products listesinin service kullanılarak initialize edileceği daha iyi bir yer var, constructor yerine onInit() methodu kullanılabilir. Bu methodu kullanabilmek için component'in OnInit interface'ini implement ediyor olması gerek.

```
export class ProductsComponent implements OnInit {

  products = [];

  constructor(private productService: ProductService) {
  }

  ngOnInit() {
    this.products = this.productService.getProducts();
  }
```

Products listesi component constructor'ı yerine burada intialize edilince component yaratıldıktan ve constructor çalıştıktan hemen sonra bunu çalıştıracaktır.

Sonuçta component oluştuğunda service'ten products listesi çekilecek ve component çalışırken yeni eleman eklendiğinde service'e yeni elemanlar eklenecektir.

Ancak problem şu ki, service'ten products listesini çekmek şuan yalnız bir kere ve component'in oluştuğu anda yapılıyor, o halde products listesi yenilendiğinde component view'i üzerine bu değişiklik yansımayacak. İşte bu sebeple bizim service'in products listesi her güncellendiğinde, component içerisinde de

```
this.products = this.productService.getProducts();
```

methodunu çalıştırmamız gerek ki products listemiz anlık olarak güncel görünsün.

Şimdi bunu yapalım:

Öncelikle service içerisine bir subject import edip yaratılan subject objesinin next methodunu her addProduct methodu çağırıldığında çalıştrdık.

```
import {Subject} from 'rxjs';

export class ProductService {
    private products = ['A Book'];
    productsUpdated = new Subject();

    addProduct(productName: string){
        this.products.push(productName);
        this.productsUpdated.next();
    }

    getProducts(){
        return [...this.products];
    }
}
```

Böylece service'e her eleman eklendiğinde bu method çalışacak.

Şimdi products component'inde bu methodun çalışmasını yakalayıp ona göre tekrar service'den products listesini çekmemiz gerek:

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

export class ProductsComponent implements OnInit, OnDestroy {

  private productsSubscription: Subscription;

  constructor(private productService: ProductService) {
  }

  ngOnInit() {
    this.products = this.productService.getProducts();
    this.productsSubscription = this.productService.productsUpdated.subscribe(
 ()=> {
      this.products = this.productService.getProducts();
    });
  }


  addProduct(form){
    if(form.valid) {
      this.productService.addProduct(form.value.productName);
    }
  }
```

```
  ngOnDestroy(){
    this.productsSubscription.unsubscribe();
  }
}
```

onInit içerisnde this.productService diye başlayan kod temelde ne zaman ki service'in productsUpdated.next() methodu çalıştı, o zaman çalışacak ve çalışınca da component'in products listesi yenilecek.

Ancak bunun yanında good practice olarak subscription'a ihtiyaç duymadığımızda ubsubscibe etmeliyiz. İşte bunun için satır kısımlar yazıldı.

Sonuçta component destroy edildiğinde unsubscribe mehodu ile sanıyorum service üzerind eyapılan değişiklikleri dinlemeyi bırakmış oluyoruz.

7. Angular Routing

Son olarak routing'e bakacağız.

Routing means that our angular app can have multiple pages event though only one page is sent from the server the other pages are really just simulated. Angular re-renders the entire page to give the illusion of having loaded a new page even though everything was done by javascript which is of course faster than fetching e new page.

Yani özetle aslında olan şey, yapay olarak sanki yeni bir html sayfasını fetch ediyormuşuz gibi bir görüntü yaratmak aslında bu noktada servere giden bir request falan olmaz, sadece javascript ile zaten çoktan yüklenmiş olan component view'leri index.html içinde ekrana getirilir.

Routing'i anlamak için öncelikle bir **home component** yarattık. Ardından diğer componentler gibi bu component'i de **app.module.ts** içerisinde import ettik.

```
import { HomeComponent } from './home/home.component';

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

Şimdi ise **app-routing.module.ts** ismiyle bir routing module'ü yaratıyoruz. Routing'I kendi module'ü içerisine koymak common convention'dır. Aslında şart değildir ama burada da öyle yapacağız:

```
import { Routes, RouterModule } from '@angular/router';
import { ProductsComponent } from './products/products.component';
import { HomeComponent } from './home/home.component';


const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'products', component: ProductsComponent},
];

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

Yukarıda aslında yeni bir module tanımlanıyor, adı da AppRoutingModule, angular routing işlemini bu module aracılığı ile yapacak.

Burada routes isminde bir constant array tanımlıyoruz, array'in her elamanı bir js object ve her js object bir route configuration'I temsil ediyor.

Dedik ki root route yani '' için gösterilecek component home component olsun. Bunun yanında /products route'I için gösterilecek component de products component olsun.

Ardından modüle içerisinde RouterModule.forRoot(routes)'ı import ediyorum ve RouterModule'ü **AppModule**'de kullanmak üzere export ediyorum.

```
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule

  ],
})
export class AppModule { }
```

Şimdi routing configürasyon işlemlerini halletmiş olduk, root routing için home component'i, /products routing'I için products component'i görünsün istiyorduk ancak tam olarak view'in neresinde bunlar görünecek vesaire bunu halletmek için, root component'in view'ine (app.component.html) geçiyoruz:

```
<a routerLink="/">Home</a>
<a routerLink="/products">Products</a>
<app-products></app-products>
<router-outlet></router-outlet>
```

Sonuçta burada <router-outlet> etiketi routing'e tabiidir! Bunun dışındaki kısımlar sabit olarak app component üzerinde gösterilecektir, ancak eğer routing /products denilirse <router-outlet> etiketi yerine sanki products component etiketi gelmiş gibi olacak, benzer şekilde eğer home route'ındaysak da sanki <router-outlet> etiketi yerine home component etiketi gelmiş gibi olacaktır.

Yukarıya eklenen linklerle de navigate edebiliriz, böylece ilk girişte çalışan app component üzerinde hem products componen görüntülenecek hem de home component görüntülenecek ancak eğer /products linkine gidersek o halde sanki alt alta iki tane

```
<app-products></app-products>
<app-products></app-products>
```

Varmış gibi olacak. Olay bundan ibaret!