ÖDEV 6

1. Öncelikle Retrofit'in tanımını yapmak gerekirse, Retrofit restful servislerle çalışan bir network kütüphanesidir. Sunucuya HTTP istekleri yapmamıza ve çeşitli yanıtlar almamıza imkan tanır. Bu istekleri yaparken ise çeşitli metotlar kullanılmaktadır. Bunlardan bazıları : GET, POST, PUT, DELETE, PATCH, OPTIONS ve HEAD'dir. Aşağıda bunların içerisinde en çok kullanılan GET, POST, PUT ve DELETE metotlarının kullanım alanları ve açıklamaları verilmektedir.

- **GET :** Get metodunu, sunucudan belirli bilgileri çekmek için kullanırız. Örneğin :

  Base url'den sonra gelen bir endpoint görülmektedir . Aşağıda belirtilen endpoint bağlantılarına bir istek yapılır ve eğer isteğimiz başarılıysa bize buradaki veriler dönmelidir. Almak istediğimiz veriyi 2. Örnekte verildiği gibi parametrelerle özelleştirebiliriz. Bunu yapmanın farklı bir yolu da Annotations'lardır. Annotationslar bu sorunun konusunda olmadığı için burada bahsedilmemiştir.

```
1- @GET("users/list")

2- @GET("users/list?sort=desc")
```

- **POST:** Sunucuda yeni bir obje oluşturmak istediğimizde kullanılır. Burada sunucuya gönderdiğimiz bilgiler body içerisine yazılarak gönderilmelidir. Bu da @BODY annotations ile olur. Body ile gönderilecek veriler aynı zamanda bir data class tarafından tutulmalıdır. Aşağıdaki kod örneğinde, **"baseurl+posts"** endpointine bir POST isteği yapılmıştır. Info adındaki data class'daki veriler POST metodu ile sunucuya gönderilmiştir. Sunucudan geri dönen model ise ApiCallResponse tarzında olmalıdır.

```
1  interface ApiCall {
2      @POST("posts")
3      fun makePost(@Body info: Info) : Call<ApiCallResponse>
4  }
```

- **PUT :** Put metodu, sunucudaki veride bir değişiklik ya da güncelleme yapmak istediğimizde kullanılır. Verinin tümünü güncellediği için benzeri bir metodu olan PATCH ile ayrılmaktadır. PATCH nesnenin sadece istenilen özelliklerini değiştirir. Bu açıdan PUT, büyük verilerle çalışırken zorlayıcı olabilir.

- **DELETE :** Bir objeyi silmek için kullanılır. Aşağıdaki örnekte User tipinde bir body gönderilerek bir obje silinmiştir.

```
1.  @DELETE("users")
2.  fun failingDeleteUser(@Body user: User): Call<User>
```

2. Annotationlar, gönderdiğimiz isteği çeşitli şekillerde farklılaştırıp özelleştirmemize yarar. Örneğin yukarıda bahsedilen @GET, @POST, @PUT ve @DELETE bir annotationdur. Bu metotların ne işe yaradığına yukarıda bahsedilmiştir. Bunlar dışında kullanılan farklı annotation'lar da bulunmaktadır.

- **QUERY :** Kullanıcılar bir yere istekte bulunurken farklı parametre değerleri gönderebilmektedirler. Örneğin : https://reqres.in/api/users?page=2 adresine istek yollanmak istiyor. Burada görüldüğü gibi, endpoint'Den sonra ? ile ayrılmış bir alan vardır. Buraya, almak istediğimiz veriyi özelleştirmek adına bazı parametreler ekleriz. Bunlar sorgu parametreleridir. Örneğin bu istekte page değeri 2 olan dataları bize getirmelidir. Bu URL'i statik olarak vermek yerine, @QUERY annotation'ı ile dinamikleştirmek mümkündür.

```
1    @GET("users")
2    fun callQueryDynamic(@Query("page") n: Int) : Call<ResponseClass>
```

Yukarıda, users endpointinden sonra, @QUERY annotation kullandığımızda bundan sonraki değerler "?" ile ayrılacak ve parametre olarak değerlendirilecektir. callQueryDynamic() fonksiyonuna Int değerinde bir parametre gönderilince(örneğin "callQueryDynamic(2)" şeklinde), bu gönderilen parametre "?page=2" şeklinde gelecektir.

- **URL :** API istekleri yaparken, url kısmını @GET,@POST vb. gibi annotation metotlarının içerisinde değil de, bir parametre ile dinamik olarak göndermek istiyorsak kullanışlıdır.

```
1    @GET
2    fun callUrlDynamic(@Url url: String): Call<ResponseClass>
```

Yukarıdaki örnekte, bu servisi kullanırken callUrlDynamic(https://dummyjson.com/docs/auth) gibi bir çağrı yapabiliriz. Bu sayede gidilmek istenen url kullanım anında belirtilmiş olur.

- **PATH :** @Path annotation'ı ile, url'i ve içerisindeki değerleri dinamikleştirebiliriz.

```
1    @GET("{endpoint}")
2    fun callReplacement(@Path("endpoint") endpoint : String, @Query("page") n: Int) : Call<Respon
```

Yukarıdaki örnekte {endpoint} ile süslü parantezler içerisinde belirtilen yapıyı fonksiyon çağırırken belirleyebiliriz. Bu yapılar string olmalıdır. Ayrıca süslü parantez içerisindeki isim ile annotation'a verdiğimiz paratez içerisindeki isim aynı olmalıdır. Burada kullanıcıdan callReplacement fonksiyonu ile hem bir @PATH, hem de @QUERY ile bir sorgu parametresi istenmiştir. Bu servisi "callReplacement("users",1)" ile çağırdığımız zaman bunnun anlamı şudur : "baseurl+user?page=1".

Başka bir örnek vermek gerekirse, url'i dinamik vermek istediğimiz durumlarda da bu yapıyı kullanabiliriz.Süslü parantezlerin içerisine {url} yazıp, aynı parametreyi @PATH ile vermek yeterli olmaktadır.

- **BODY :** API isteklerinde(özellikle POST methodunda), body içerisinde bir nesne gönderilmesi istendiyse bu annotation'ı kullanmalıyız. Body ile gönderilecek tip ya bir nesne olmalıdır ya da bir data class(kotlin özelinde). Aşağıda örnek bir kod verilmiştir.

```
interface ApiCall {
    @POST("posts")
    fun makePost(@Body info: Info) : Call<ApiCallResponse>
}
```

Burada post isteği yaparken body kısmında Info sınıfından bir data class gönderilmiştir.

3. HTTP Status Code'lar, internette gezinirken ya da bir API'ye istek atarken vb. gibi durumlarda yaptığımız istekler ile alakalı olarak bize bir geri bildirim sunan kodlardır. İsteğimiz başarılı mı oldu, başarısız mı oldu ya da sunucudan kaynaklı bir hata mı var gibi konularda bize ipuçları sunmaktadırlar. Aşağıda en sık karşılaşılan bazı status code'lar verilmiştir.

- **404 :** Yazılım dünyasına çok aşina olmayan kişilerin dahi bildiği bu status code, istek yaptığımız sayfanın bulunamadığını bize bildirir. Sayfa silinmiş olabilir ya da şu anda girilen URL geçersiz olup sayfanın URL bilgileri değişmiş olabilir.

- **200 :** Sunucunun isteği başarılı bir şekilde aldığı ve geriye de başarılı bir şekilde yanıt döndürdüğünü söyleyen durum kodudur. Sayfanın çalıştığının bir göstergesidir.

- **403 :** İstemcinin ulaşmak istediği adrese yetkisi olmadığını gösterir. Sunucu tarafında, adrese ulaşmaya çalışan istemciye izin verilmediği durumlarda oluşur. Sunucu istemciye, "seni tanıyorum ama burayı görüntülemeye iznin yok" der.

- **401 :** Sunucuya istek atan istemcinin kimlik bilgilerinin bulunamadığı durumlarda ortaya çıkar. Sunucu istemciye bir nevi "seni tanımıyorum" der. Örneğin istek atılan API'ye yanlış token gönderilmiş olunabilir.

- **500 :** Sunucu taraflı bir hata olduğunu bize bildirir.