RETROFIT

Retrofit, Android uygulamalarda RESTful API'lar ile iletişim kurmayı kolaylaştıran bir HTTP kütüphanesidir. HTTP istekleri göndermek ve dönen sonuçları işlemek için kullanılır.

Retrofit'i kullanabilmek için öncelikle gerekli implementasyonları yapmak gerekiyor. Retrofit kütüphanesini ve dönen sonuçları işlemek için kullanılacak "converter" kütüphanesini projeye eklemek gerekiyor. Burada tercihi Gson Converter'den yana kullandık.

```
implementation 'com.squareup.retrofit2:retrofit:(insert latest version)'
implementation 'com.squareup.retrofit2:converter-gson:(insert latest version)'
```

HTTP isteklerini yapabilmek için internet iznini de projeye eklemek gerekiyor.

```
<uses-permission android:name="android.permission.INTERNET"/>
```

RETROFIT KULLANIMI

Retrofit'i en basit şekilde kullanmak için bir Retrofit nesnesine, bir de isteklerin yapılacağı API interface'sine ihtiyaç vardır. En basit şekilde Retrofit nesnesi aşağıdaki şekilde oluşturulabilir:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Daha sonra API interface'sini referans göstererek Retrofit create edilir. Oluşturulan nesne üzerinden tüm API interface methodlarına erişilebilir.

```
val service = retrofit.create(ApiService::class.java)
val call = service.getUser(1)
```

Buradan sonrası API interface'sinin içine yazılacak HTTP isteklerini çağırmaktan ibarettir. Şimdi API interface'sinde yazılabilecek istek türlerine bakalım.

TEMEL HTTP YÖNTEMLERİ

@GET:

Sunucudan veri almak için kullanılan HTTP yöntemidir. İstek yapılırken veri gönderilmez. Belirli bir URI'dan veri almak için kullanılır.

Kullanım Alanları: Bir kullanıcının bilgilerini çekmek, tüm kullanıcıları çekmek gibi örnekler verilebilir.

```
@GET("users/{id}")
fun getUser(@Path("id")id : String) : Call<UserResponse>
```

@POST:

Sunucuya veri göndermek için kullanılan HTTP yöntemidir. İstek yapılırken veri gönderilir. Kaynak oluşturmak, güncellemek, silmek için kullanılabilir.

Kullanım Alanları: Sunucuda yeni bir kullanıcı oluşturmak, yeni bilgi girişi, yeni kaynak oluşturulması için kullanılabilir. Uygulama tarafında oluşturulan bir kullanıcı nesnesi @Body annotation'u ile sunucuya gönderilebilir. Burada gönderilen nesne Gson Converter ile Json nesnesine dönüştürülür. Veri Body'de güvenli bir şekilde gönderilir.

```
@POST("users/add")
fun addUser(@Body user: User) : Call<UserResponse>
```

Ayrıca post yöntemi ile form verileri de gönderilebilir. @FormUrlEncoded annotation'u eklenen bir metot ve @Field annotation'ı ile eklenen parametreler ile form verileri de gönderilebilir.

```
@FormUrlEncoded
@POST("users/add")
fun addUser(@Field("username") username: String,
    @Field("email") email: String,
    @Field("password") password: String) : Call<UserResponse>
```

@PUT:

Sunucuya veri göndererek yeni bir kaynak göndermek veya var olan bir kaynağı değiştirmek için kullanılan HTTP yöntemidir.

Kullanım Alanları: Daha önceden var olan bir kullanıcının verilerini güncelleme, var olan kaynağın belli bölümlerini veya tamamını değiştirme gibi işlemler için kullanılabilir.

```
@PUT("users/update")
fun updateUser(@Body user: User): Call<UserResponse>
```

@DELETE:

Sunucudaki bir kaynağı silmek için kullanılan HTTP yöntemidir.

Kullanım Alanları: Var olan bir kaynağı, bir kullanıcıyı, bir veriyi silmek için kullanılabilir.

```
@DELETE("users/delete")
fun deleteUser(@Body user: User): Call<UserResponse>
```

DİĞER ÇOK KULLANILAN ANNOTATIONLAR

@Path: Dinamik url yolları oluşturmak için kullanılır. Aşağıdaki örnekte id değeri @Path annotation'u kullanılarak dinamik bir şekilde eklenmiştir.

```
@GET("users/{id}")
fun getUser(@Path("id")id : String) : Call<UserResponse>
```

@Query: Sorgu parametrelerini eklemek için kullanılır. Aşağıdaki örnekte belirli şehir ve ilçedeki kullanıcıları aramak için @Query annotation'u kullanılmıştır.

```
@GET("users")
fun getUsers(@Query("ilce")ilce : String, @Query("il")il: String) : Call<UsersResponse>
```

@Body: İsteğin gövdesini temsil eder. Genellikle Json nesnesi yollamak için kullanılır. Xml, file, text türünde veriler göndermek için de kullanılabilir.

```
@PUT("users/update")
fun updateUser(@Body user: User): Call<UserResponse>
```

@Header: İsteğin başlık kısmını temsil eder. Burada API'nın gerek duyduğu bilgiler gönderilebilir. API key, content type, token gibi gönderilmesi gereken doğrulama verileri burada girilir.

```
@GET("user")
  fun getUser(@Header("Authorization") token: String?): Call<User>
```

@Headers: Birden fazla header ekleneceğinde tercih edilebilecek grup başlık yöntemidir. @Header'ın aksine fonksiyonun üzerinde yazılır.

```
@Headers("content-type: application/json",
    "authorization: ${API_KEY}")
@GET("users/{id}")
fun getUser(@Path("id")id : String) : Call<UserResponse>
```

@FormUrlEncoded: POST isteklerinde girilen form verilerini kodlamak için kullanılır.

@Field: Form verilerinde sunucu ve uygulamadan gelen verileri eşleştirmek için kullanılır.

```
@FormUrlEncoded
@POST(“users/add”)
fun addUser(@Field("username") username: String,
    @Field("email") email: String,
    @Field("password") password: String) : Call<UserResponse>
```

@Multipart: POST isteklerinde birden fazla veri parçası olduğunda kullanılır. Birden fazla farklı türde veri gönderileceğinde tercih edilebilir. Metin, dosya, resim gibi verilerden birden fazlasını göndermek gerektiğinde kullanılabilir.

@Part: Mutlipart işlemlerde farklı partları belirtmek için kullanılır.

```
@Multipart
@POST("upload")
fun uploadFile(
    @Part filePart: MultipartBody.Part,
    @Part("description") description: RequestBody
): Call<UploadResponse>
```

@Streaming: Büyük dosyaları indirirken kullanılır. Büyük dosyalar indirilip belleğe yüklenirken kullanıcı arayüzünün donmaması veya bellek sorunu oluşmaması için kullanılır.

```
@Streaming
@GET("file/download")
fun downloadFile(): Call<ResponseBody>
```

@SerializedName: Bir sınıf property’sinin Json’da farklı bir adda temsil edildiğini belirtmek için kullanılır.

@Expose: Bazı karakterlerin desteklenmesi için kullanılır. Türkçe dil desteği için kullanılabilir.

```
data class User(
  @Expose
  @SerializedName("user_id") val userId: String,
   @Expose
  @SerializedName("user_name") val userName: String)
```

HTTP DURUM KODLARI

200 OK: İstek başarılı. İstenen kaynak yanıt olarak döndü.

201 Created: Yeni kaynak başarıyla oluşturuldu. Genelde POST isteklerinde karşılaşılır.

204 No Content: İstek başarılı fakat yanıt olarak hiçbir şey dönmedi. Genelde PUT ve DELETE isteklerinde karşılaşılır.

400 Bad Request: İstek sunucu tarafında anlaşılamadı veya işlenemedi. Hatalı veri gönderilme durumlarında karşılaşılır.

401 Unauthorized: İsteğin yapılabilmesi için kimlik doğrulaması gerekiyor, yetkisiz kullanıcı.

403 Forbidden: İstek yapılan kaynağa erişim izni yok. Kullanıcı doğrulanmış bile olsa erişim izni yok.

404 Not Found: İstek yapılan kaynak bulunamadı. URL geçersiz veya sunucuda mevcut değil.

405 Method Not Allowed: İstek yapılan URL üzerinde yanlış method kullanılmış. Örneğin, sadece POST isteğini destekleyen URL'e GET isteği yapılması gibi.

500 Internal Server Error: Sunucu bir iç hata ile karşılaştı ve istek işlenemedi. Genellikle sunucu taraflı bir hata olduğunda karşılaşılır.

503 Service Unavailable: Sunucu geçici olarak hizmet veremiyor. Genellikle sunucunun bakımda veya yoğun olması sebebiyle karşılaşılır.

Son olarak da gönderilen isteklerin sonuçlarının nasıl yakalanacağına bakalım. Servisten çağırılan fonksiyonu .enqueue ile çağırarak bir Callback objesi oluştururuz. Bu obje bizden iki fonksiyonu override etmemizi ister. Bunlar onResponse ve onFailure'dir. Eğer sonuç döndürürse onResponse çalışır ve orada veri işlenir. Eğer istek başarılı olmazsa onFailure çalışır ve hata mesajı döndürülebilir.

```
val call = service.getUser(1)
call.enqueue(object : Callback<Data> {
   override fun onResponse(call: Call<Data>, response: Response<Data>) {
      if (response.isSuccessful) {
         val data = response.body()
      }
   }
   override fun onFailure(call: Call<Data>, t: Throwable) {
      // İstek başarısız oldu
   }
})
```