
STOCARD

Networking

What do we need?

- Retrieve data
- Parse data
- Background thread
- Error handling

OkHttp

```
// build this once and reuse
val client = OkHttpClient.Builder().build()

// specify a request
val request = Request.Builder()
        .get()
        .url("https://android-hackschool.herokuapp.com")
        .build()
```

OkHttp

```
// execute async
client.newCall(request).enqueue(object : Callback {
    override fun onResponse(call: Call, response: Response) { ... }
    override fun onFailure(call: Call, e: IOException) { ... }
})

// execute sync
try {
    val response = client.newCall(request).execute()
    ...
} catch (ex: IOException) { ... }
```

What do we need? - OkHttp

- Retrieve data ✔
- Parse data ✖
  - Use Gson or Moshi if JSON data
- Background thread ✔
- Error handling ✔

→ square.github.io/okhttp/

Retrofit

```
interface BackendInterface {

    @GET("/")
    fun fetchMessages(): Call<List<Message>>

    @POST("/message")
    fun postMessage(@Body chatMessage: Message): Call<Message>

}
```

Retrofit

```
val retrofit = Retrofit.Builder()
        .client(okHttpClient) // ← we know this guy already
        .baseUrl("https://android-hackschool.herokuapp.com")
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
// now we have an implementation of our Interface!
val backend = retrofit.create(BackendInterface::class.java)

backend.fetchMessages().enqueue(object : Callback<List<Message>> {
    override fun onFailure(call: Call<>, t: Throwable) { ... }
    override fun onResponse(call: Call<>, response: Response<>) { ... }
})
```

What do we need? - Retrofit

- Retrieve data ✔
- Parse data ✔
  - Converters for Gson, Moshi, Jackson, Protobuf, XML, Custom
- Background thread ✔
- Error handling ✔

→ square.github.io/retrofit/

One more thing

```
// Retrofit
public interface Call<T> extends Cloneable {
  void enqueue(Callback<T> callback);
}
public interface Callback<T> {
  void onResponse(Call<T> call, Response<T> response);
  void onFailure(Call<T> call, Throwable t);
}
// makes us write
backend.fetchMessages().enqueue(object : Callback<List<Message>> {
    override fun onFailure(call: Call<>, t: Throwable) { ... }
    override fun onResponse(call: Call<>, response: Response<>) { ... }
})
```

One more thing

```
fun <T> Call<T>.enqueue( // how we would like it in Kotlin
        onFailure: (t: Throwable) -> Unit, // Kotlin knows functions
        onResponse: (response: Response<T>) -> Unit
)
```

One more thing - Kotlin Extension Functions

```
fun <T> Call<T>.enqueue( // how we would like it in Kotlin
        onFailure: (t: Throwable) -> Unit, // Kotlin knows functions
        onResponse: (response: Response<T>) -> Unit
) {
    this.enqueue(object : Callback<T> { // ← this is the Callback we know
        override fun onFailure(call: Call<T>, t: Throwable) {
            onFailure(t)
        }
        override fun onResponse(call: Call<T>, response: Response<T>) {
            onResponse(response)
        }
    })
}
```

One more thing - Kotlin Extension Functions

```
// now we can write
backend.fetchMessages().enqueue(
    onFailure = { error -> ... },
    onResponse = { response -> ... }
)

// or
backend.fetchMessages().enqueue(
    { error -> ... },
    { response -> ... }
)
```