BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 8

Karim Karimov
Lecturer

What is HTTP?

The Hypertext Transfer Protocol (HTTP) is the foundation of the World Wide Web, and is used to load webpages using hypertext links. HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. A typical flow over HTTP involves a client machine making a request to a server, which then sends a response message.

What is in an HTTP request?

An HTTP request is the way Internet communications platforms such as web browsers ask for the information they need to load a website.
Each HTTP request made across the Internet carries with it a series of encoded data that carries different types of information. A typical HTTP request contains:

1. HTTP version type
2. a URL
3. an HTTP method
4. HTTP request headers
5. Optional HTTP body.

HTTP method

An HTTP method, sometimes referred to as an HTTP verb, indicates the action that the HTTP request expects from the queried server. For example, two of the most common HTTP methods are ‘**GET**’ and ‘**POST**’; a ‘GET’ request expects information back in return (usually in the form of a website), while a ‘POST’ request typically indicates that the client is submitting information to the web server (such as form information, e.g. a submitted ***username*** and ***password***).

HTTP request headers

HTTP headers contain text information stored in key-value pairs, and they are included in every HTTP request (and response). These headers communicate core information, such as what browser the client is using and what data is being requested.
Example of HTTP request headers from Google Chrome's network tab:

HTTP response

An HTTP response is what web clients receive from an Internet server in answer to an HTTP request. These responses communicate valuable information based on what was asked for in the HTTP request.

A typical HTTP response contains:

1. an HTTP status code
2. HTTP response headers
3. optional HTTP body

HTTP response

HTTP status codes are 3-digit codes most often used to indicate whether an HTTP request has been successfully completed. Status codes are broken into the following 5 blocks:

- **1xx Informational**
- **2xx Success**
- **3xx Redirection**
- **4xx Client Error**
- **5xx Server Error**

The “xx” refers to different numbers between 00 and 99.
Status codes starting with the number ‘2’ indicate a success. For example, after a client requests a webpage, the most commonly seen responses have a status code of ‘200 OK’, indicating that the request was properly completed.

HTTP response headers

Much like an HTTP request, an HTTP response comes with headers that convey important information such as the language and format of the data being sent in the response body.
Example of HTTP response headers from Google Chrome's network tab:

HTTP response body

Successful HTTP responses to ‘GET’ requests generally have a body which contains the requested information. In most web requests, this is HTML data that a web browser will translate into a webpage.

Network access in Android

To perform network operations in your application, your manifest must include the following permissions:

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

Best practices for security

Before you add networking functionality to your app, you need to ensure that data and information within your app stays safe when transmitting it over a network. To do so, follow these networking security best practices:

- Minimize the amount of sensitive or personal **user data** that you transmit over the network.
- Send all network traffic from your app over **SSL**.
- Consider creating a network security configuration, which lets your app trust custom **certificate authorities (CAs)** or restrict the set of system CAs that it trusts for secure communication.

Choose an HTTP client

Most network-connected apps use HTTP to send and receive data. The Android platform includes the HttpsURLConnection client, which supports TLS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. In this topic, we use the **Retrofit HTTP** client library, which lets you create an HTTP client declaratively. Retrofit also supports automatic serialization of request bodies and deserialization of response bodies.

Repository

To simplify the process of performing network operations and reduce code duplication in various parts of your app, you can use the repository design pattern. A repository is a class that handles data operations and provides a clean API abstraction over some specific data or resource.
You can use Retrofit to declare an interface that specifies the HTTP method, URL, arguments, and response type for network operations, as in the following example:

```
interface UserService {
    @GET("/users/{id}")
    suspend fun getUser(@Path("id") id: String): User
}
```

Repository

Within a repository class, functions can encapsulate network operations and expose their results. This encapsulation ensures that the components that call the repository don't need to know how the data is stored. Any future changes to how the data is stored are isolated to the repository class as well.

```
class UserRepository constructor(
    private val userService: UserService
) {
    suspend fun getUserById(id: String): User {
        return userService.getUser(id)
    }
}
```

Call in background thread

To avoid creating an unresponsive UI, don't perform network operations on the main thread. By default, Android requires you to perform network operations on a thread other than the main UI thread; if you don't, a **NetworkOnMainThreadException** is thrown.

Survive configuration changes

When a configuration change occurs, such as a screen rotation, your fragment or activity is destroyed and recreated. Any data that isn’t saved in the instance state for your fragment or activity, which should hold only small amounts of data, is lost and you might need to make your network requests again.
You can use a ViewModel to ensure that your data survives configuration changes. A ViewModel is a component that's designed to store and manage UI-related data in a lifecycle-conscious way. Using the UserRepository created above, the ViewModel can make the necessary network requests and provide the result to your fragment or activity using LiveData.

Survive configuration changes

```
class MainViewModel constructor(
    savedStateHandle: SavedStateHandle,
    userRepository: UserRepository
) : ViewModel() {
    private val userId: String = savedStateHandle["uid"] ?:
        throw IllegalArgumentException("Missing user ID")

    private val _user = MutableLiveData<User>()
    val user = _user as LiveData<User>

    init {
        viewModelScope.launch {
            try {
                // Calling the repository is safe as it will move execution off
                // the main thread
                val user = userRepository.getUserById(userId)
                _user.value = user
            } catch (error: Exception) {
                // show error message to user
            }

        }
    }
}
```

Headers in Retrofit

Static headers

```
@Headers("Cache-Control: max-age=640000")
@GET("widget/list")
Call<List<Widget>> widgetList();
```

```
@Headers({
        "Accept: application/vnd.github.v3.full+json",
        "User-Agent: Retrofit-Sample-App"})
@GET("users/{username}")
Call<User> getUser(@Path("username") String username);
```

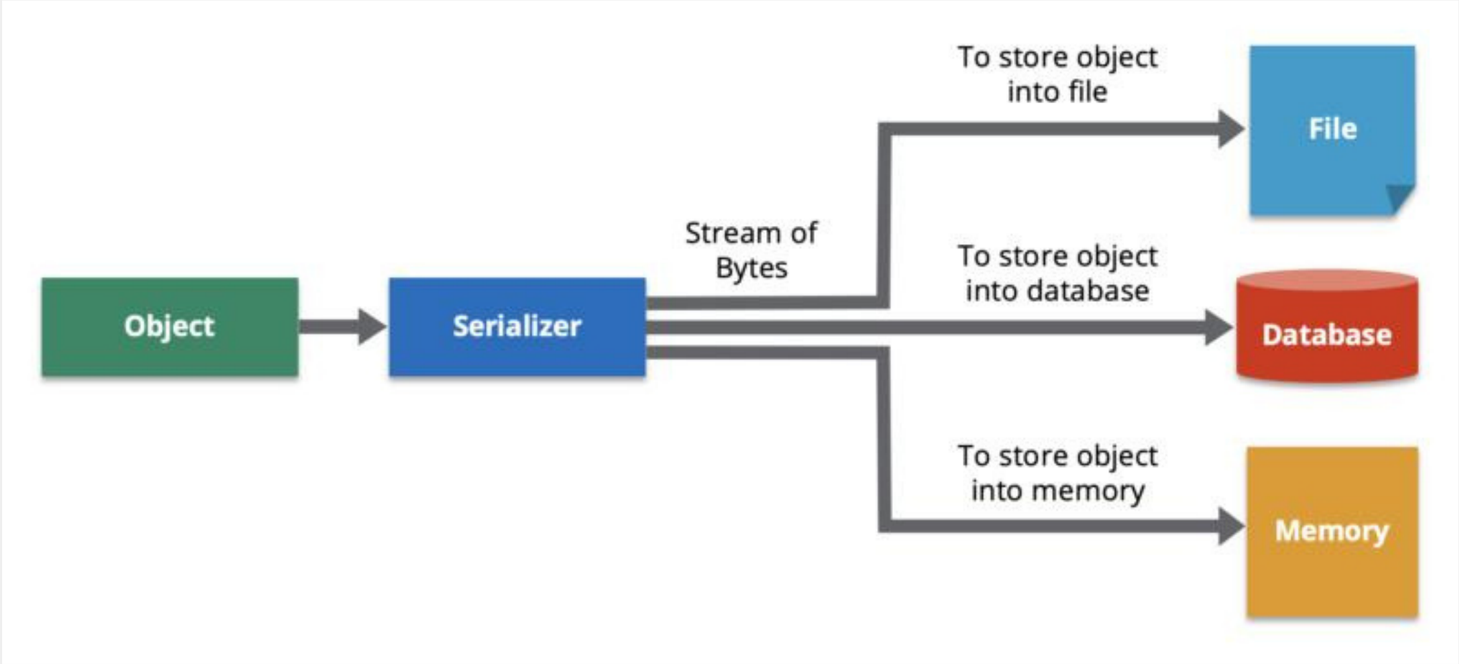
Dynamic headers

```
@GET("user")
Call<User> getUser(@Header("Authorization") String auth)
```

```
@GET("user")
Call<User> getUser(@HeaderMap Map<String, String> headers)
```

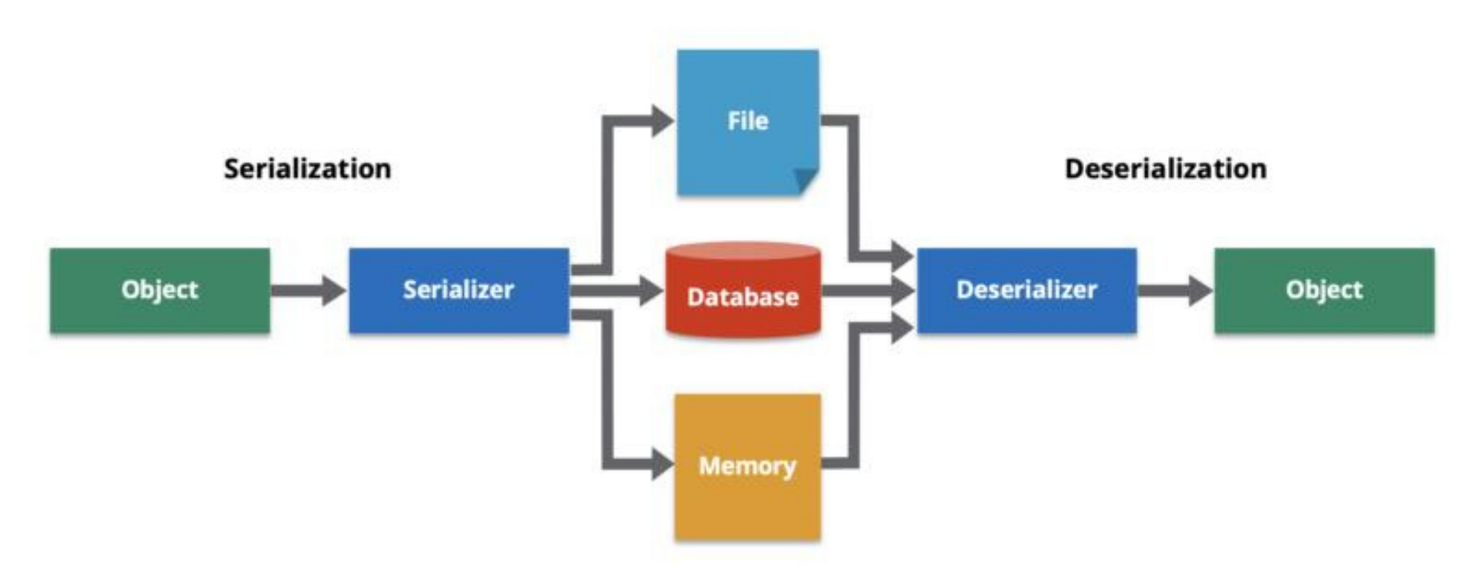
Serialization

Serialization is the process of converting a data object—a combination of code and data represented within a region of data storage—into a series of bytes that saves the state of the object in an easily transmittable form.



Deserialization

The reverse process—constructing a data structure or object from a series of bytes—is **deserialization**. The deserialization process recreates the object, thus making the data easier to read and modify as a native structure in a programming language.



HTTP interceptors

Interceptors are a powerful mechanism that can monitor, rewrite, and retry calls. Here's a simple interceptor that logs the outgoing request and the incoming response.

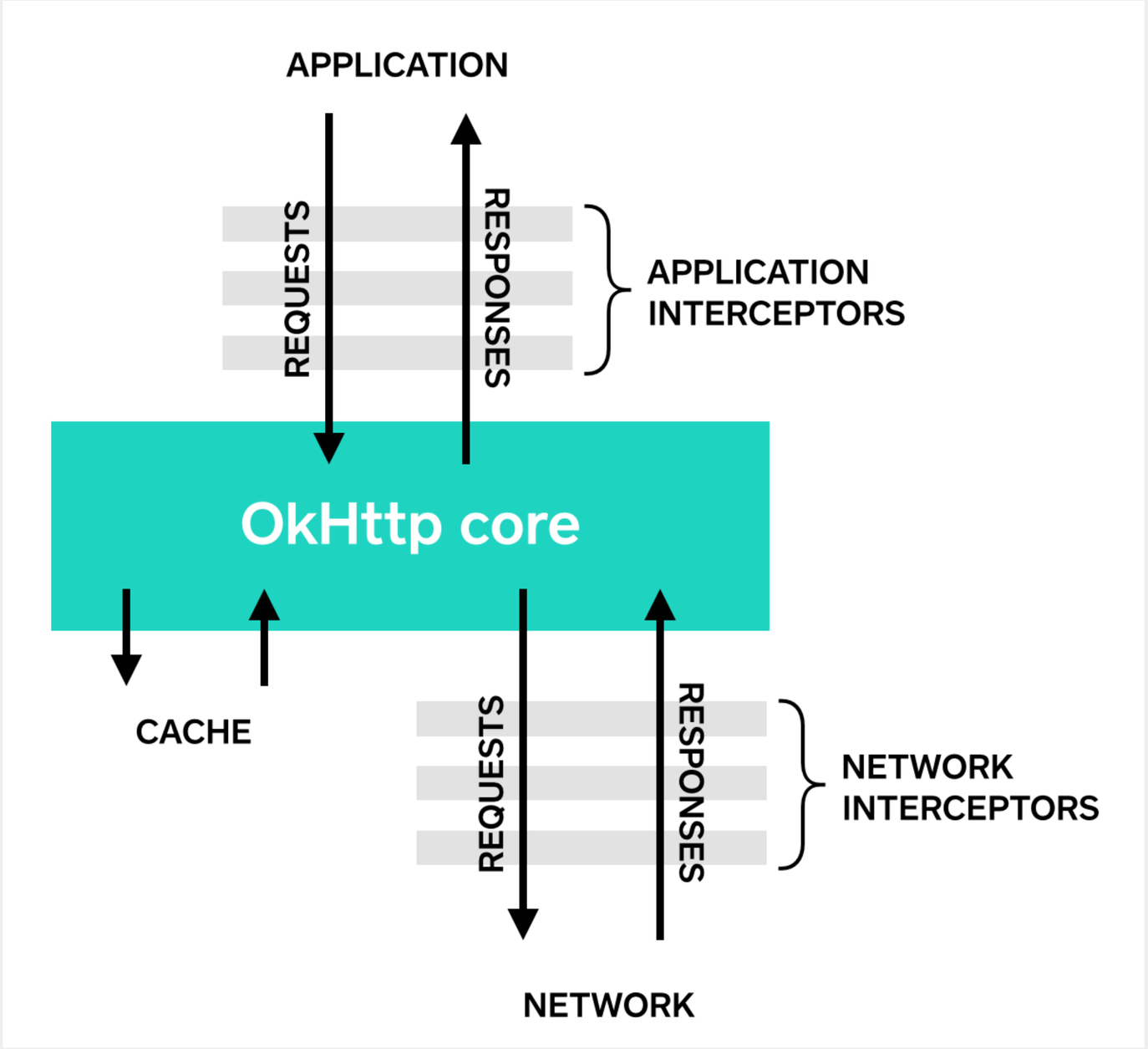
```
class LoggingInterceptor implements Interceptor {
  @Override public Response intercept(Interceptor.Chain chain) throws IOException
    Request request = chain.request();

    long t1 = System.nanoTime();
    logger.info(String.format("Sending request %s on %s%n%s",
        request.url(), chain.connection(), request.headers()));

    Response response = chain.proceed(request);

    long t2 = System.nanoTime();
    logger.info(String.format("Received response for %s in %.1fms%n%s",
        response.request().url(), (t2 - t1) / 1e6d, response.headers()));

    return response;
  }
}
```

HTTP interceptors

A call to ***chain.proceed(request)*** is a critical part of each interceptor's implementation. This simple-looking method is where all the HTTP work happens, producing a response to satisfy the request. If ***chain.proceed(request)*** is being called more than once previous response bodies must be closed.

Interceptors can be chained. Suppose you have both a compressing interceptor and a checksumming interceptor: you'll need to decide whether data is compressed and then checksummed, or checksummed and then compressed. OkHttp uses lists to track interceptors, and interceptors are called in order.

HTTP interceptors



Application interceptors

The URL **http://www.publicobject.com/helloworld.txt** redirects to **https://publicobject.com/helloworld.txt**, and OkHttp follows this redirect automatically. We can see that we were redirected because ***response.request().url()*** is different from ***request.url()***. The two log statements log two different URLs.

```
OkHttpClient client = new OkHttpClient.Builder()
    .addInterceptor(new LoggingInterceptor())
    .build();

Request request = new Request.Builder()
    .url("http://www.publicobject.com/helloworld.txt")
    .header("User-Agent", "OkHttp Example")
    .build();

Response response = client.newCall(request).execute();
response.body().close();
```

```
INFO: Sending request http://www.publicobject.com/helloworld.txt on null
User-Agent: OkHttp Example

INFO: Received response for https://publicobject.com/helloworld.txt in 1179.7ms
Server: nginx/1.4.6 (Ubuntu)
Content-Type: text/plain
Content-Length: 1759
Connection: keep-alive
```

Network interceptors

When we run this code, the interceptor runs **twice**. Once for the initial request to **http://www.publicobject.com/helloworld.txt**, and another for the redirect to **https://publicobject.com/helloworld.txt**.

```
OkHttpClient client = new OkHttpClient.Builder()
    .addNetworkInterceptor(new LoggingInterceptor())
    .build();

Request request = new Request.Builder()
    .url("http://www.publicobject.com/helloworld.txt")
    .header("User-Agent", "OkHttp Example")
    .build();

Response response = client.newCall(request).execute();
response.body().close();
```

```
INFO: Sending request http://www.publicobject.com/helloworld.txt on Connection
User-Agent: OkHttp Example
Host: www.publicobject.com
Connection: Keep-Alive
Accept-Encoding: gzip

INFO: Received response for http://www.publicobject.com/helloworld.txt in 115.6
Server: nginx/1.4.6 (Ubuntu)
Content-Type: text/html
Content-Length: 193
Connection: keep-alive
Location: https://publicobject.com/helloworld.txt

INFO: Sending request https://publicobject.com/helloworld.txt on Connection{pub
User-Agent: OkHttp Example
Host: publicobject.com
Connection: Keep-Alive
Accept-Encoding: gzip

INFO: Received response for https://publicobject.com/helloworld.txt in 80.9ms
Server: nginx/1.4.6 (Ubuntu)
Content-Type: text/plain
Content-Length: 1759
Connection: keep-alive
```

References

- **HTTP overview** - https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http
- **Connectivity android** - https://developer.android.com/training/basics/network-ops/connecting
- **Retrofit HTTP client** - https://square.github.io/retrofit
- **Serialization** - https://hazelcast.com/glossary/serialization
- **OkHTTP interceptors** - https://square.github.io/okhttp/features/interceptors

End of the session