PROG7314

HYPERTEXT TRANSFER PROTOCOL (HTTP) CONNECTION

An HTTP connection is the fundamental mechanism by which a client (your app) and a server exchange data over the web. At its core:

Protocol layer: HTTP (Hypertext Transfer Protocol) sits at the application layer, riding over TCP/IP.

Request – response model: The client opens a connection to the server's URL, sends an HTTP request (method, headers, optional body), and then reads back the server's HTTP response (status code, headers, body).

Statelessness: Each request is independent; any context (authentication tokens, session IDs) must be resent on every call.

HYPERTEXT TRANSFER PROTOCOL (HTTP) CONNECTION

Uses of HTTP connections in Kotlin
Whether you’re writing a JVM application or an Android app, HTTP lets you:

1. Consume RESTful APIs

Fetch JSON or XML data from web services (e.g. weather, maps, social feeds)
Submit form data or JSON payloads via POST/PUT

2. Upload and download files

Download images, videos, documents at runtime
Upload user-generated content (photos, logs) to a server

RESTFUL APIS +

```
private fun fetchJoke() {
    CoroutineScope(Dispatchers.IO).launch {
```

Coroutines:

- CoroutineScope(Dispatchers.IO).launch { ... } starts a background coroutine on the IO dispatcher (optimised for network/file I/O).

```
val url = URL( spec: "https://v2.jokeapi.dev/joke/Programming")
val connection = (url.openConnection() as HttpURLConnection).apply {
    requestMethod = "GET"
    connectTimeout = 10_000
    readTimeout = 10_000
}
```

- URL(...): wraps the endpoint string into a URL object.
- openConnection(): returns a generic URLConnection; casting to HttpURLConnection lets you set HTTP-specific options.
- requestMethod: "GET" means “retrieve data.”
- Timeouts: ensure your coroutine won’t hang indefinitely.

+

```
val jokeText = if (connection.responseCode == HttpURLConnection.HTTP_OK) {
    val response = connection.inputStream.bufferedReader().use { it.readText() }
    connection.disconnect()
```

responseCode check

- Only proceed if it’s 200 OK.

Reading

- inputStream.bufferedReader().use { it.readText() } reads the entire JSON payload as a string, then closes the stream.

Disconnect

- Immediately call disconnect() to free resources.

```
// Parse according to 'type'
val json = JSONObject(response)
when (json.getString( name: "type")) {
    "single"  -> json.getString( name: "joke")
    "twopart" -> "${json.getString( name: "setup")}\n\n${json.getString( name: "delivery")}"
    else      -> "🤔 Unexpected joke format"
}
```

JSON parsing

- Wrap the raw string in JSONObject.
- getString("type") tells you if it’s a "single" or "twopart" joke.
- Branch on that to extract the right fields ("joke" vs. "setup" + "delivery").

Error path

- For non-200 responses, return a simple error message.

- Uses /joke/{category} (I picked “Programming”).
- Returns either:
  - { "type": "single", "joke": "..." }
  - { "type": "twopart", "setup": "...", "delivery": "..." }
  - So you had to update:

1. The URL string itself

2. The JSON parsing: switch from "value" → "joke" or "setup"/"delivery" based on "type".

If I find a new URL, what do I change?

URL construction

- Update the string (or better, a constant) that you pass to URL(...) (or to your Uri.Builder if you're building it).

Query parameters (optional)

- If the new endpoint requires parameters, append them with Uri.Builder or string interpolation.

JSON structure/parsing

- Check the API's docs or hit the URL in your browser/Postman to see the keys and nesting.
- Adjust your JSONObject(...).getString("...") calls (or use a data class + a library like Gson/Retrofit to map the JSON automatically).

View update

- If you extracted the URL into a constant or method parameter, you usually only need to change it in one place.
- My parsing/UI-update code stays the same unless the JSON format itself has changed.