GLOBAL SMART TECHNOLOGIES

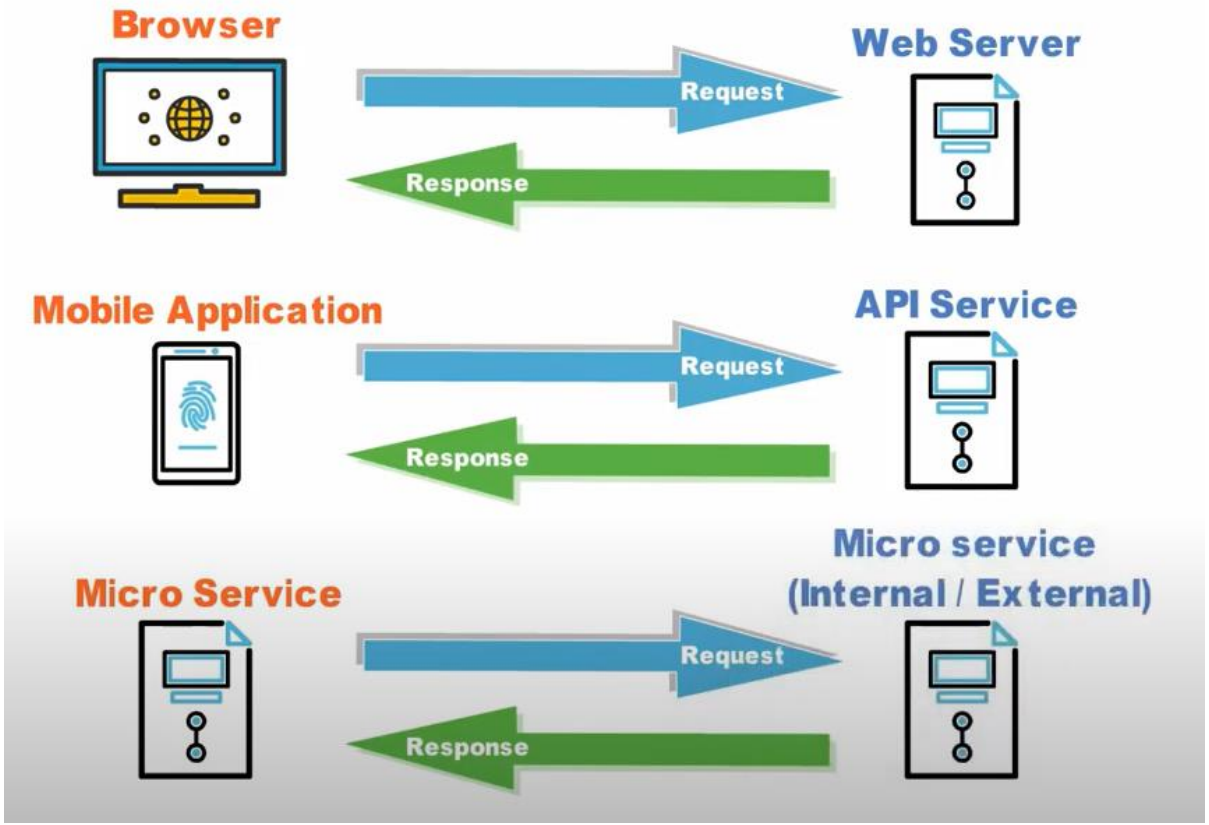
Network Request Xml/Json Parser

Created By: **DoanPT1**

Content

- Communication between devices
- HTTP and HTTPS
- Http Request and Http Response
- HttpURLConnection
- Required Permissions and check the network availability
- XML format and XmlPullParser
- Json format and Json Parser
- Gson
- Retrofit
- Retrofit with OkHttp
- Retrofit with coroutines
- Retrofit with RxJava3
- Add Headers to requests
- Logging with Retrofit

Communication

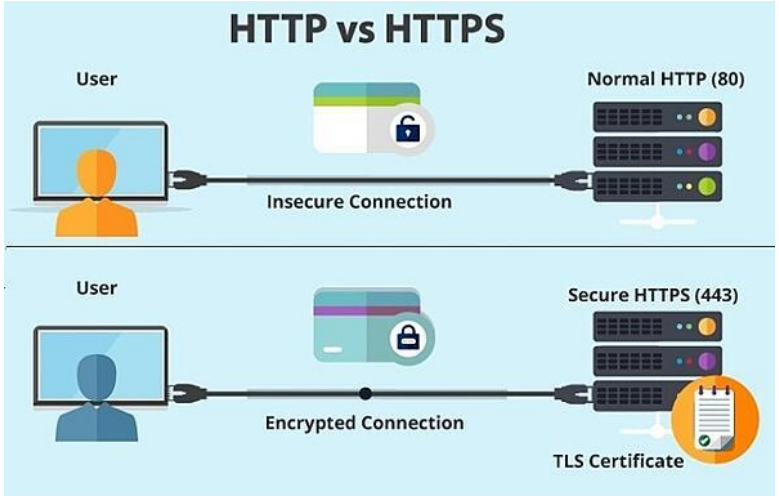


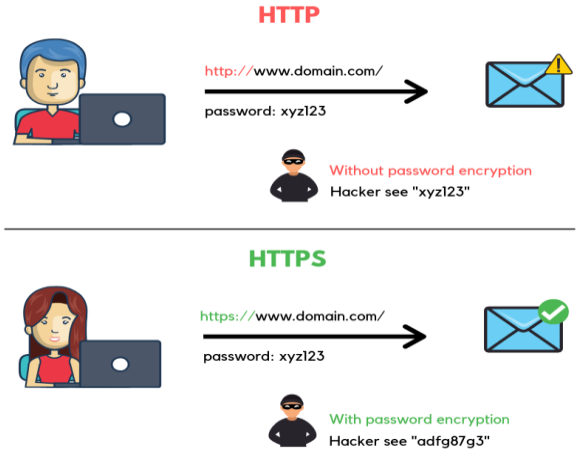
HTTP

HTTPS

What is HTTP and HTTPS

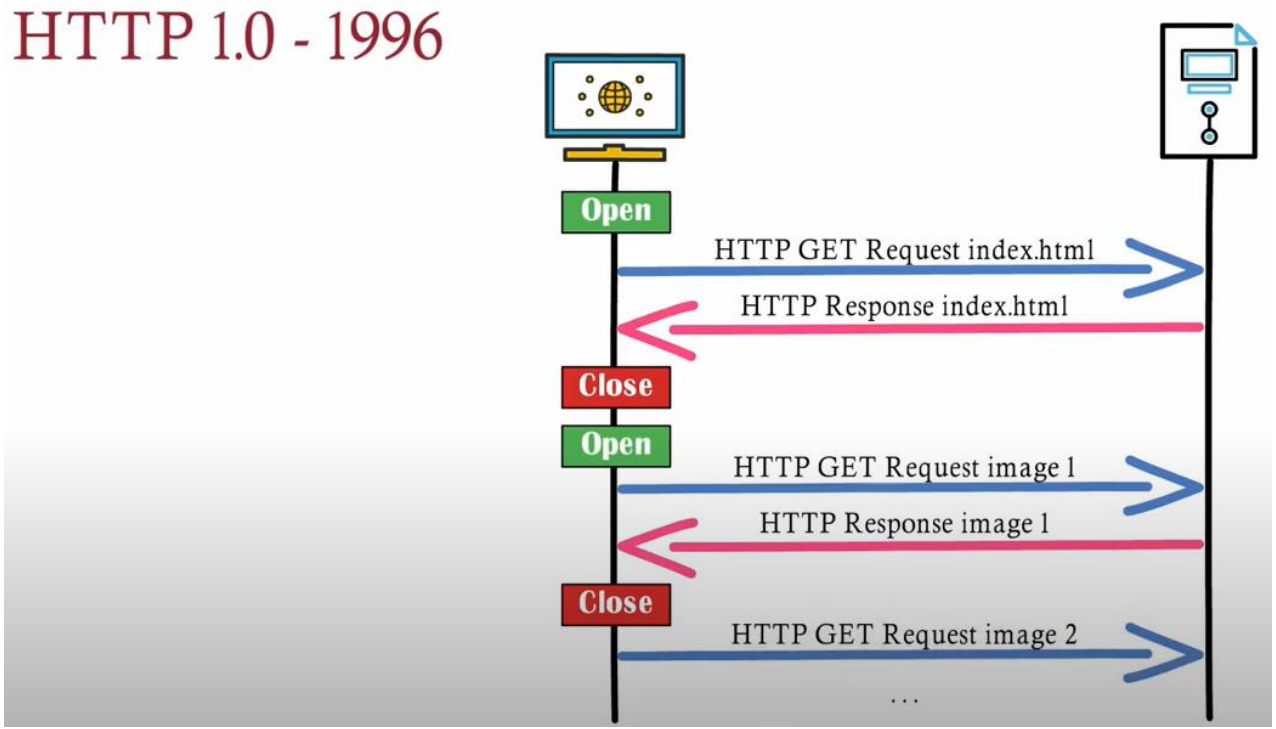
- HTTP is stand for Hyper Text Transfer Protocol. Communication between client computers and web servers is done by sending **HTTP Requests** and receiving **HTTP Responses**
- HTTPS(Hypertext Transfer Protocol Secure) is the secure version of HTTP, which is the primary protocol used to send data between a web browser and a website. HTTPS is encrypted in order to increase security of data transfer





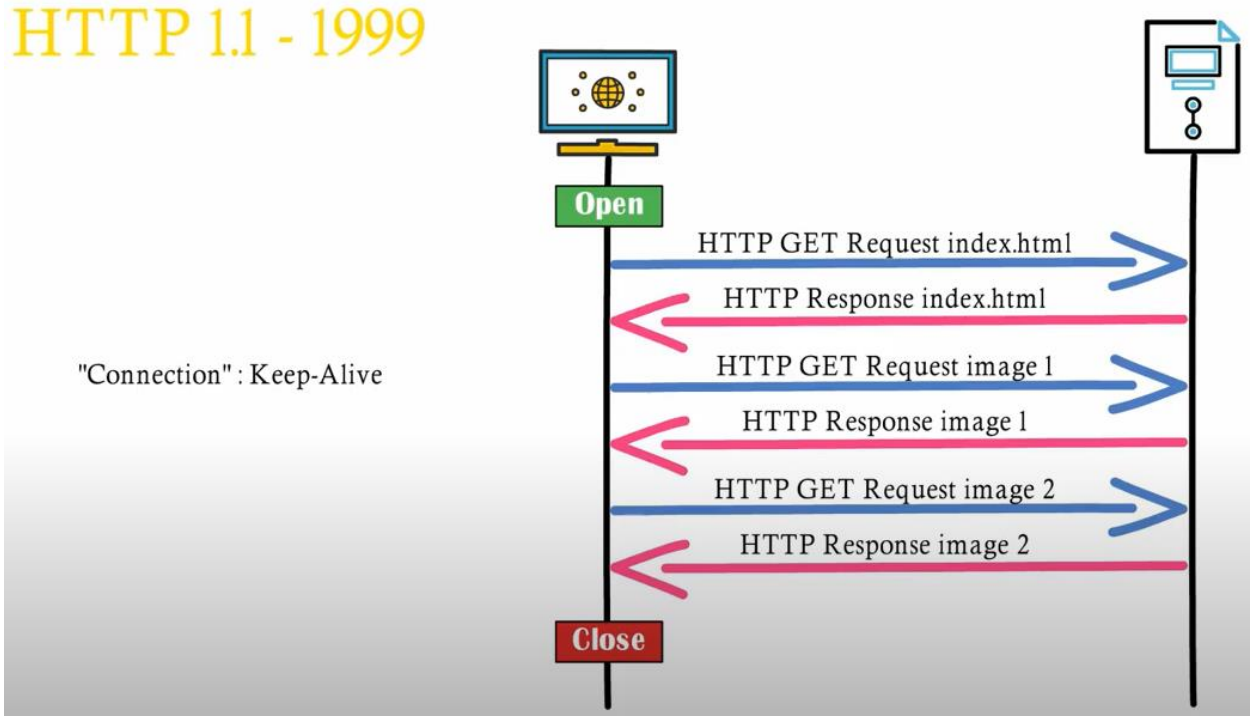
HTTP 1.0

With Http 1.0, for each request/response client must create a connection and close it after done the request

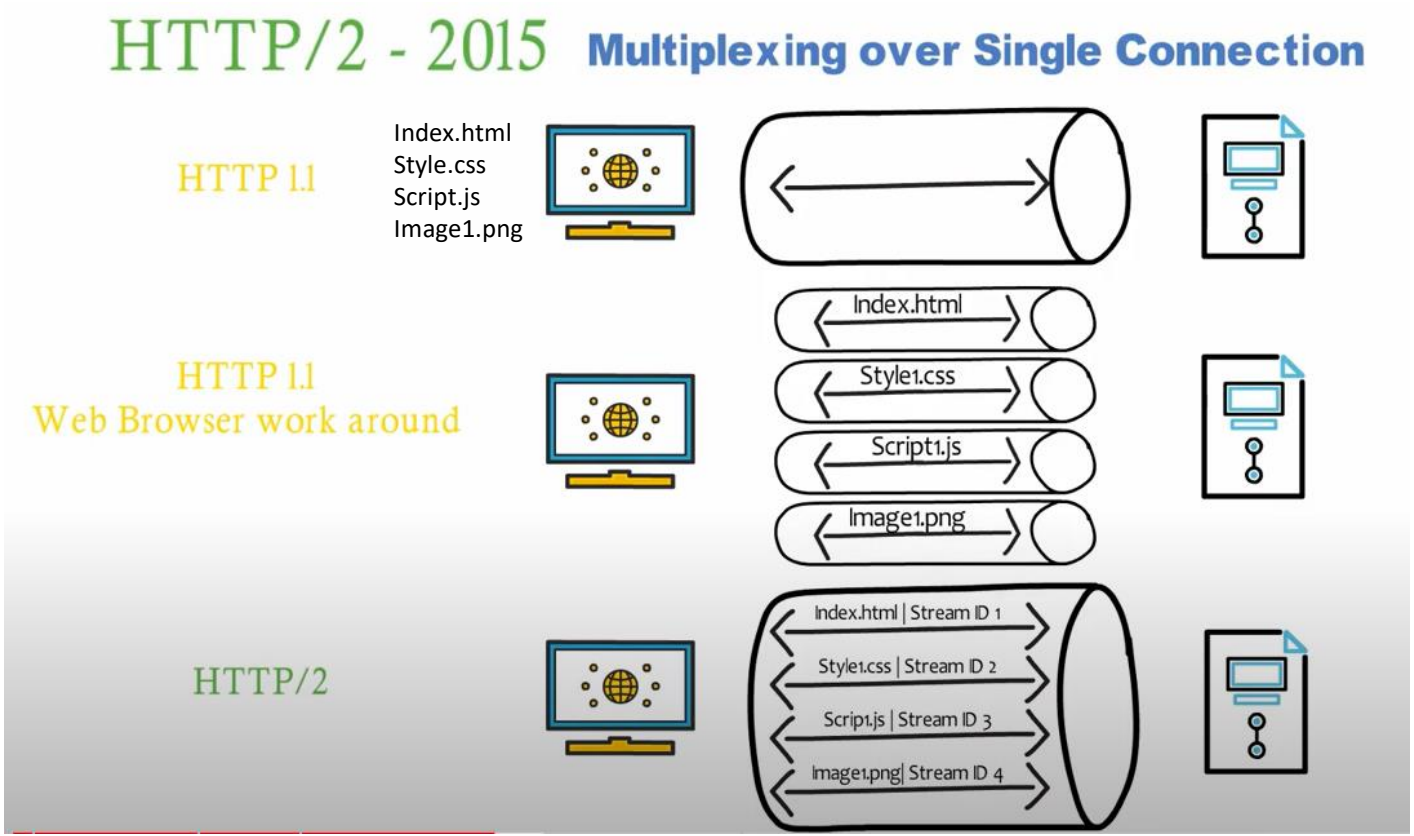


HTTP 1.1

With Http 1.1, the connection speed is faster because with one connection, server and client can do many request/response via “Connection:Keep-Alive” in header



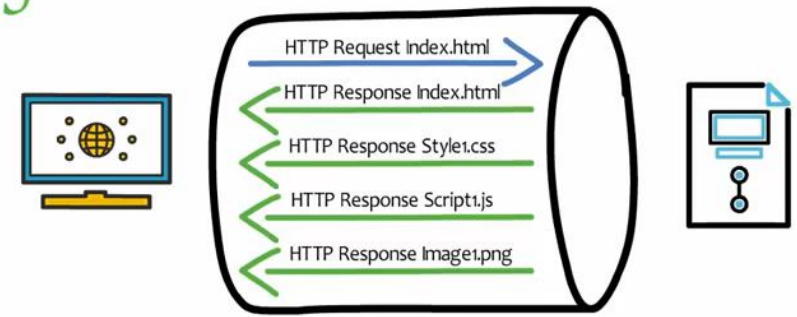
HTTP2


HTTP/2 - 2015
Multiplexing over Single Connection
HTTP 1.1
Index.html
Style.css
Script.js
Image1.png
HTTP 1.1
Web Browser work around
Index.html
Style1.css
Script1.js
Image1.png
HTTP/2
Index.html | Stream ID 1
Style1.css | Stream ID 2
Script1.js | Stream ID 3
Image1.png| Stream ID 4

HTTP2

HTTP/2 - 2015

Server Push



Compression (Headers and Data)

Secure by default

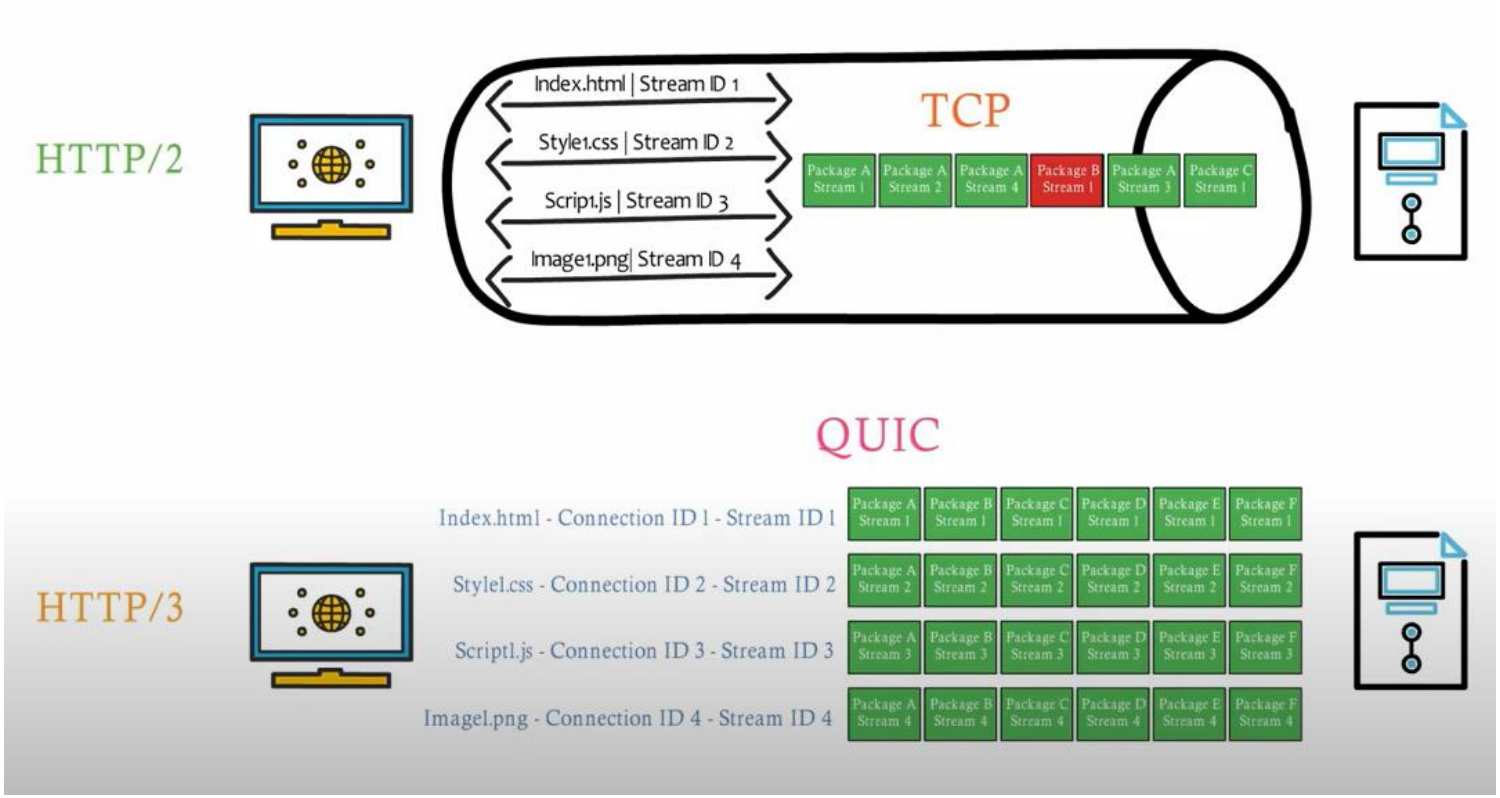
Protocol Negotiation during TLS

Server Push can be abused when configured incorrectly

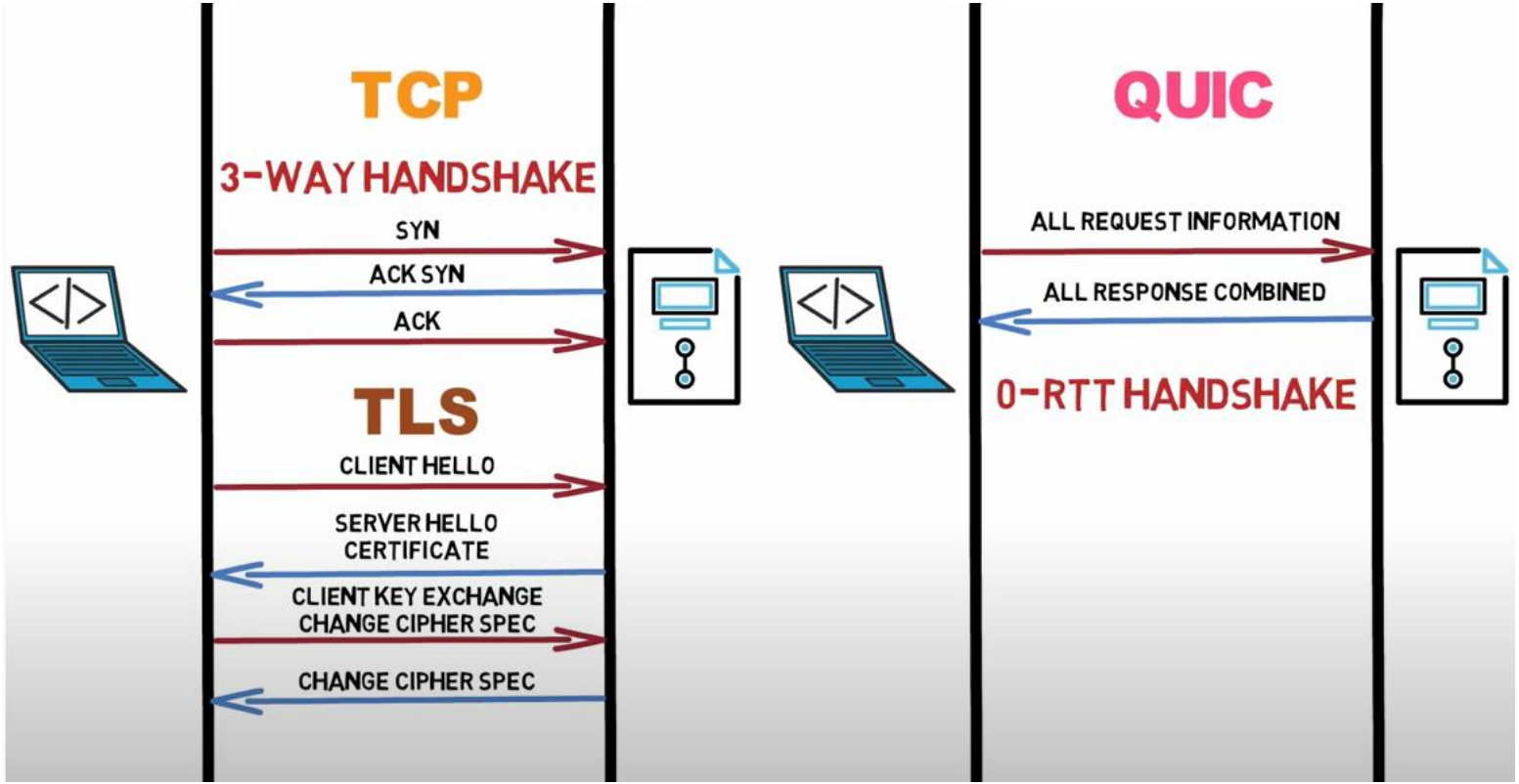
Can be slower when in mixed mode (H1 and H2)

Compare speed between HTTP1 and HTTP2: https://http2.golang.org/gophertiles

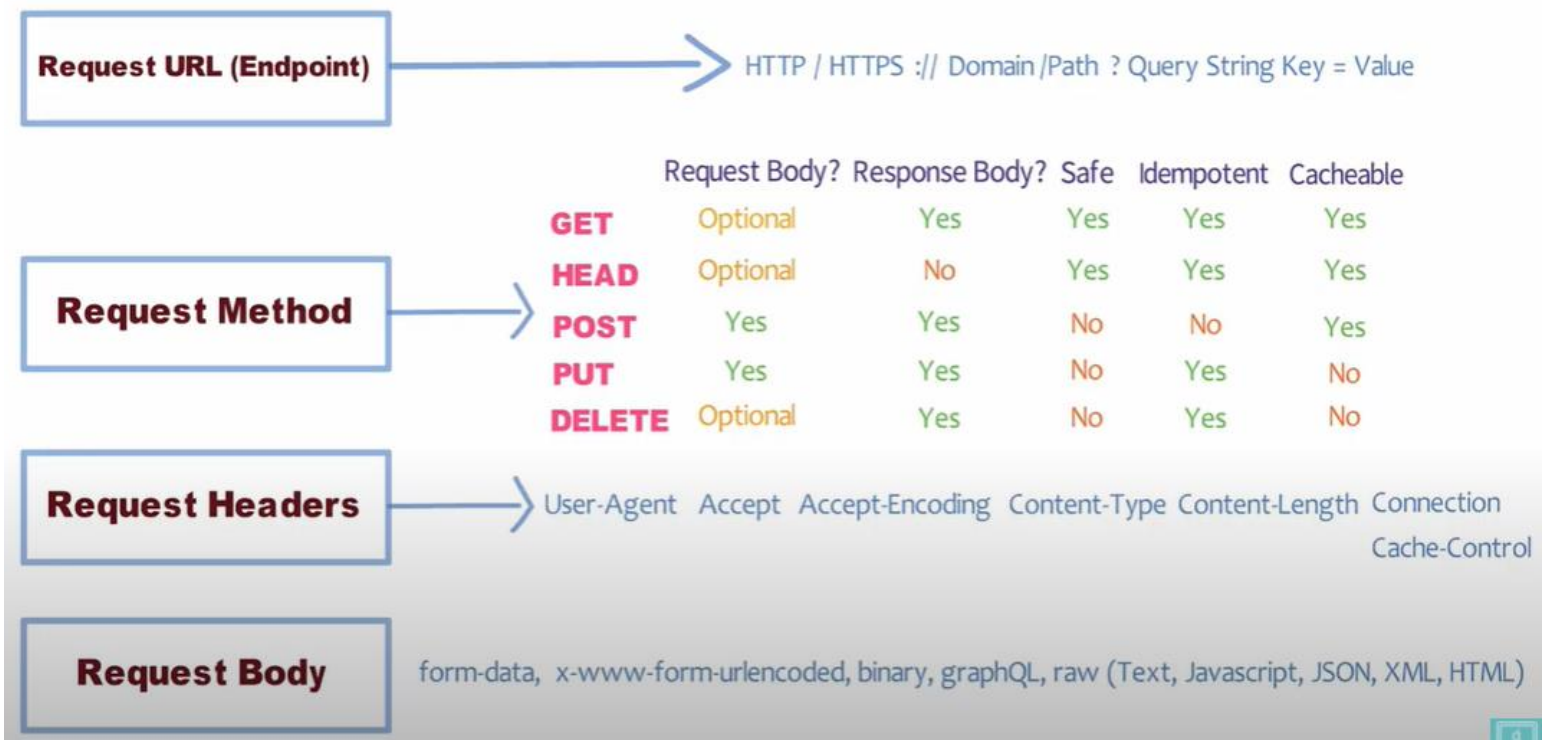
HTTP3 – Cải tiến việc truyền dữ liệu



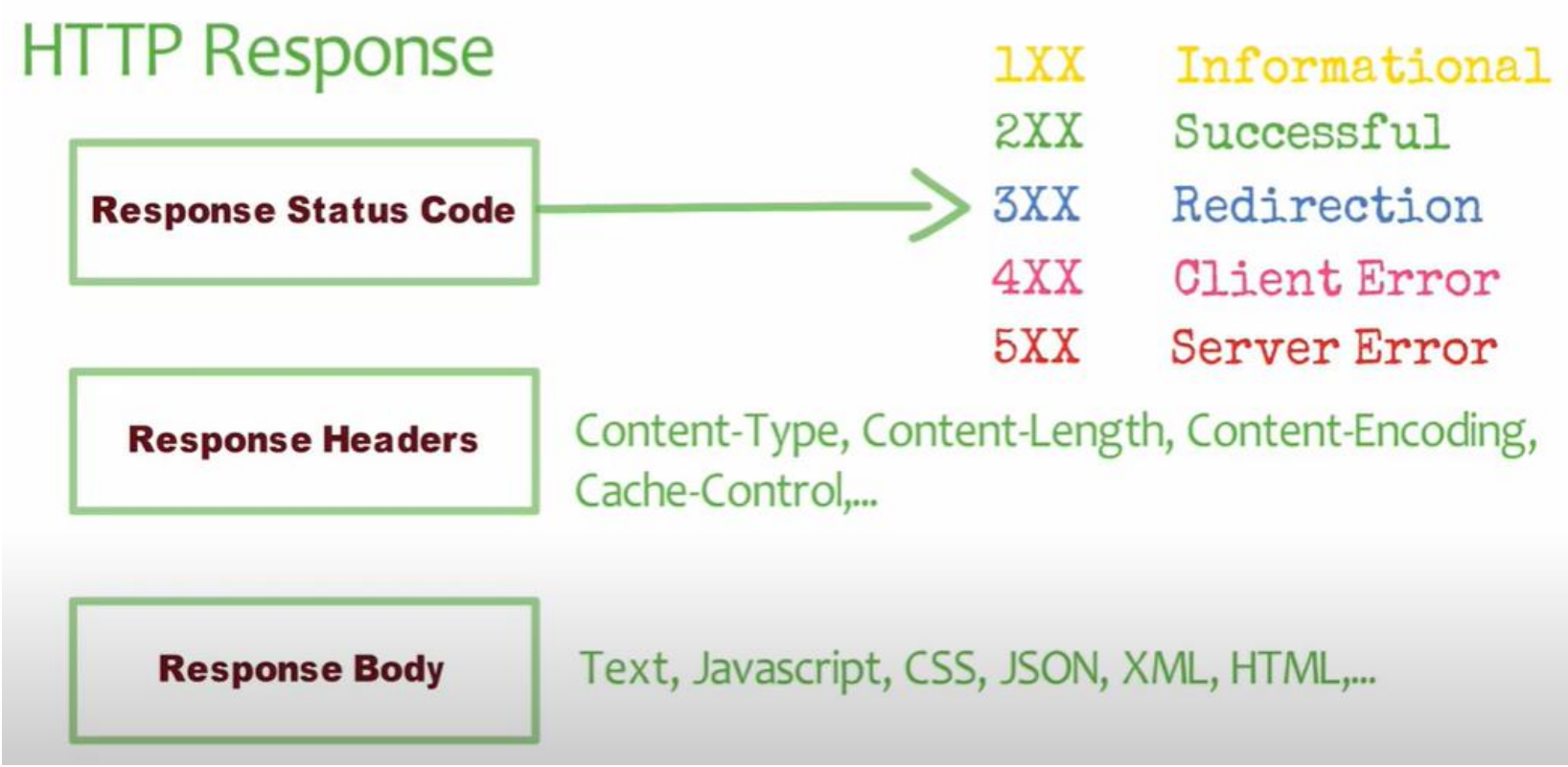
HTTP3 – Tăng tốc quá trình bắt tay



HTTP Request



HTTP Response



HttpUrlConnection

- HttpURLConnection is the standard HTTP client for Android, used to send and receive data over the web. It is a concrete implementation of URLConnection for HTTP (RFC 2616).
- Using HttpURLConnection

1. Create URL object and pass a link as a parameter
2. Obtain a new HttpURLConnection by calling url.openConnection() and casting the result to HttpURLConnection
3. Prepare the request. The primary property of a request is its URI. Request headers may also include metadata such as credentials, preferred content types, and session cookies.
4. Optionally upload a request body. Instances must be configured with setDoOutput(true) if they include a request body. Transmit data by writing to the stream returned by URLConnection.getOutputStream().
5. Read the response. Response headers typically include metadata such as the response body's content type and length, modified dates and session cookies. The response body may be read from the stream returned by URLConnection.getInputStream(). If the response has no body, that method returns an empty stream.
6. Disconnect. Once the response body has been read, the HttpURLConnection should be closed by calling disconnect(). Disconnecting releases the resources held by a connection so they may be closed or reused.

HttpURLConnection Example

```
14      fun getListPokemon(link: String): List<Pokemon> {
15          val url = URL(link)
16          val http = url.openConnection()
17          val br = BufferedReader(InputStreamReader(http.getInputStream()))
18          val data = StringBuilder()
19          var line = br.readLine()
20          while (line != null) {
21              data.append(line)
22              line = br.readLine()
23          }
24          val rootObj = JSONObject(data.toString())
25          val pokeList: JSONArray = rootObj.getJSONArray("results")
26          val results = mutableListOf<Pokemon>()
27          for (i in 0 until pokeList.length()) {
28              val pokemonObj: JSONObject = pokeList[i] as JSONObject
29              results.add(Pokemon(i, pokemonObj.getString("name"), pokemonObj.getString("url")))
30          }
31          return results.toList()
32      }
33  }
```

Require Permission

- To access the Internet your application requires the android.permission.INTERNET permission.

- To check the network state your application requires the android.permission.ACCESS_NETWORK_STATE permission.

Network operations can involve unpredictable delays. To prevent this from causing a poor user experience, always perform network operations on a separate thread from the UI. Using Thread/AsyncTask/Coroutines/RxJava… to create worker thread for http request

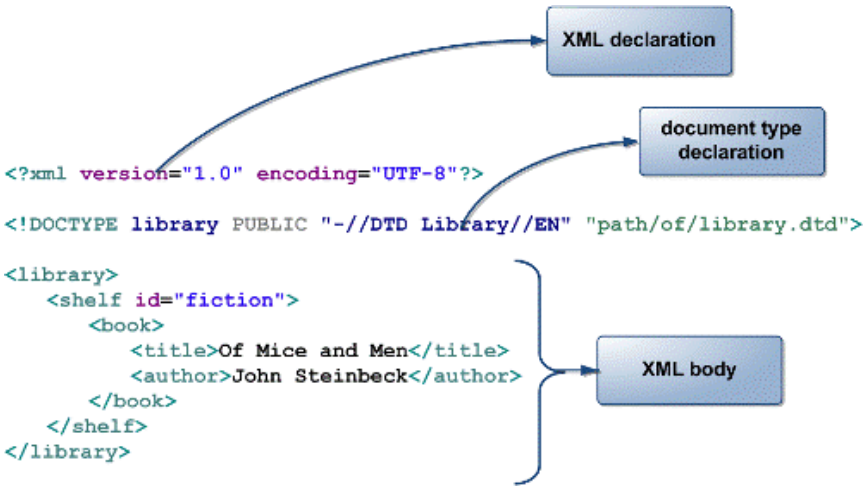
Check the network availability

- Obviously the network on an Android device is not always available. You can check the network is currently available via the following code.

```
370    fun isNetWorkAvailable(): Boolean {
371        val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
372        val capabilities =
373            connectivityManager.getNetworkCapabilities(connectivityManager.activeNetwork)
374        if (capabilities != null) {
375            when {
376                capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> {
377                    Log.i( tag: "Internet",  msg: "NetworkCapabilities.TRANSPORT_CELLULAR")
378                    return true
379                }
380                capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) -> {
381                    Log.i( tag: "Internet",  msg: "NetworkCapabilities.TRANSPORT_WIFI")
382                    return true
383                }
384                capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET) -> {
385                    Log.i( tag: "Internet",  msg: "NetworkCapabilities.TRANSPORT_ETHERNET")
386                    return true
387                }
388            }
389        }
390        return false
391    }
392 }
```

What is XML?

- XML stands for Extensible Markup Language.XML is a very popular format and commonly used for sharing data on the internet
- Websites that frequently update their content, such as news sites or blogs, often provide an XML feed so that external programs can keep abreast of content changes

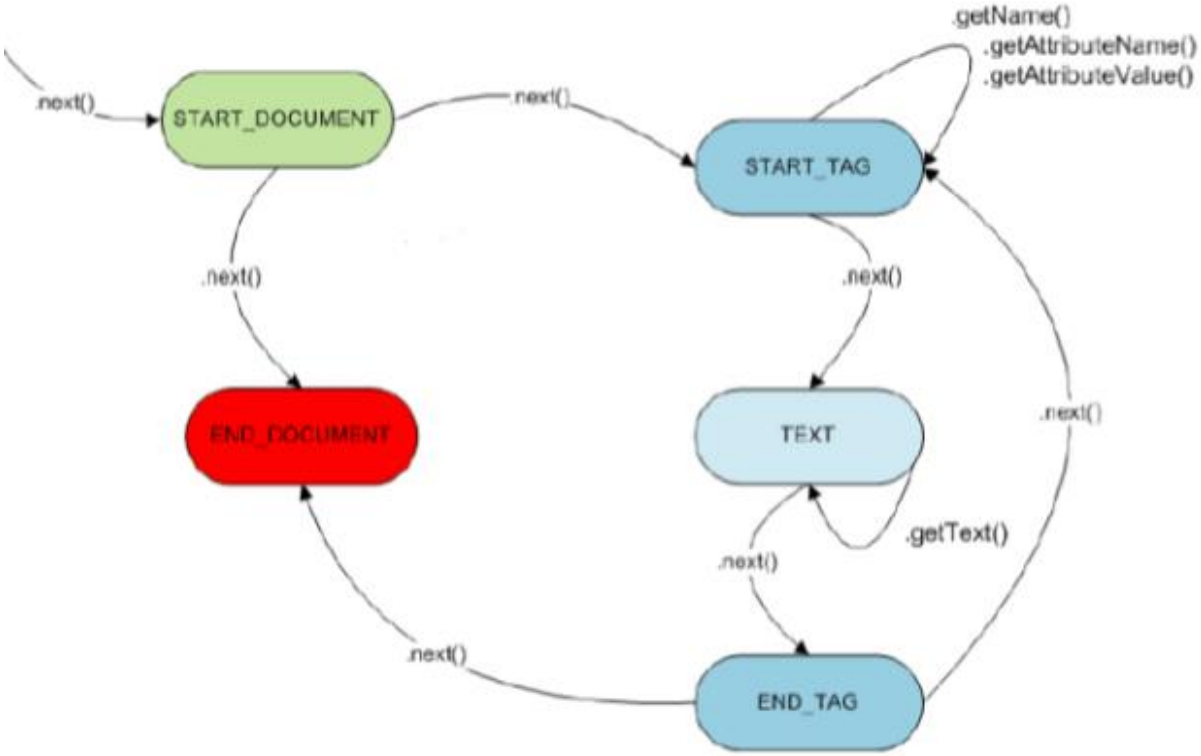


Android XML Parser

- Android Provide three types of XML Parser
  - ✓ **XmlPullParser**
  - ✓ DOM
  - ✓ SAX
- Android recommend using XmlPullParser because It's efficient, easy to use and maintainable
- Refer: Comparing methods of XML parsing in Android

- In Android, JSOUP is popular library for html parsing using DOM

XML Pull Parser Flow



XML Pull Parser Example

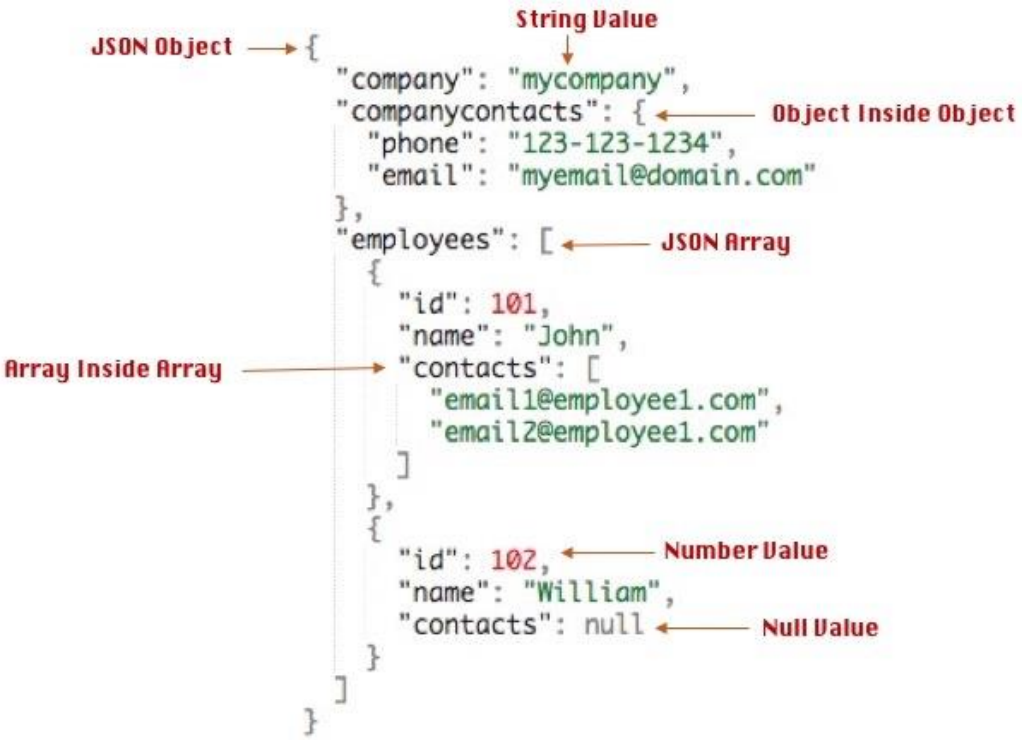
```
object StackOverFlowParser {

    fun parserLink(link: String): List<Question> {
        val url = URL(link)
        val http = url.openConnection()
        val parserFactory = XmlPullParserFactory.newInstance()
        val xmlPullParser = parserFactory.newPullParser()
        xmlPullParser.setInput(http.getInputStream(), "utf-8")

        var text = ""
        var eventType = xmlPullParser.eventType
        var question: Question? = null
        val questions = mutableListOf<Question>()
        while (eventType != XmlPullParser.END_DOCUMENT) {
            when (eventType) {
                XmlPullParser.START_TAG -> {
                    val startTag = xmlPullParser.name
                    if (startTag == "entry") {
                        question = Question()
                    }
                }
                XmlPullParser.END_TAG -> {
                    val endTag = xmlPullParser.name
                    if (endTag == "id") {
                        question?.url = text
                    }
                    ...
                    }else if (endTag == "entry") {
                        questions.add(question!!)
                    }
                }
                XmlPullParser.TEXT -> {
                    text = xmlPullParser.text
                }
            }
            eventType = xmlPullParser.next()
        }
        return questions
    }
}
```

What is JSON

- JSON stands for JavaScript Object Notation
- JSON is very light weight, structured, easy to parse and much human readable.
- JSON is best alternative to XML when your android app needs to interchange data with your server



Json Parser

- Android provide include Json parser in native sdk.
- There are 2 Object are: JSONObject and JSONArray

✓ JSONObject: we can get attribute of object by getXXX(XXX is value type)

✓ JSONArray: we can use as an Array and get element by index

https://pokeapi.co/api/v2/ pokemon Submit

Need a hint? Try pokemon/ditto, pokemon/1, type/3, ability/4, or pokemon?limit=100&offset=200.

Direct link to results: https://pokeapi.co/api/v2/pokemon

Resource for pokemon


```
count: 1118
next: "https://pokeapi.co/api/v2/pokemon?offset=20&limit=20"
previous: null
▼ results: [] 20 items
  ▼ 0: {} 2 keys
      name: "bulbasaur"
      url: "https://pokeapi.co/api/v2/pokemon/1/"
  ▼ 1: {} 2 keys
      name: "ivysaur"
      url: "https://pokeapi.co/api/v2/pokemon/2/"
  ▼ 2: {} 2 keys
      name: "venusaur"
      url: "https://pokeapi.co/api/v2/pokemon/3/"
  ▼ 3: {} 2 keys
      name: "charmander"
      url: "https://pokeapi.co/api/v2/pokemon/4/"
  ▼ 4: {} 2 keys
      name: "charmeleon"
      url: "https://pokeapi.co/api/v2/pokemon/5/"
  ▼ 5: {} 2 keys
      name: "charizard"
      url: "https://pokeapi.co/api/v2/pokemon/6/"
  ▼ 6: {} 2 keys
      name: "squirtle"
      url: "https://pokeapi.co/api/v2/pokemon/7/"
```


View raw JSON (1.956 kB, 87 lines)

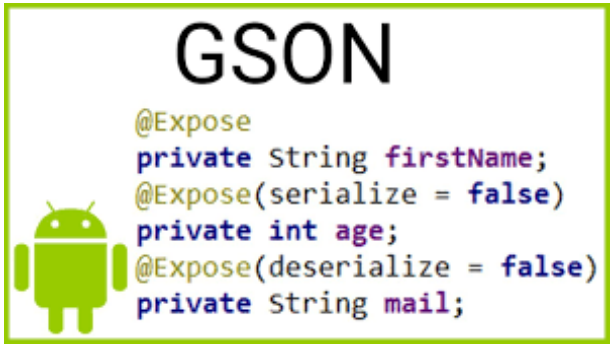
JSON Parser Example

```
object PokemonParser {

    fun getListPokemon(link: String): List<Pokemon> {
        val url = URL(link)
        val http = url.openConnection()
        val br = BufferedReader(InputStreamReader(http.getInputStream()))
        val data = StringBuilder()
        var line = br.readLine()
        while (line != null) {
            data.append(line)
            line = br.readLine()
        }
        val rootObj = JSONObject(data.toString())
        val pokeList: JSONArray = rootObj.getJSONArray("results")
        val results = mutableListOf<Pokemon>()
        for (i in 0 until pokeList.length()) {
            val pokemonObj: JSONObject = pokeList[i] as JSONObject
            results.add(Pokemon(i, pokemonObj.getString("name"), pokemonObj.getString("url")))
        }
        return results.toList()
    }
}
```

GSON

- Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of




```
Gson gson = new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();
Employee employee = new Employee("John", 30, "john@mail.com");
String jsonResult = gson.toJson(employee);

String json = "{\"age\":30,\"firstName\":\"John\",\"mail\":\"john@mail.com\",\"password\":\"fdfarg2\"}";
Employee employee1 = gson.fromJson(json, Employee.class);
```

Retrofit

- Retrofit turns your Rest API into a Interface
- Simplifies HTTP communication by turning remote APIs into declarative, type-safe interfaces
- A type-safe HTTP client for Android and Java
- Created by Square(Jake Wharton)

- Refer: Homepage and git repo

Retrofit work flow

1. Create an interface and define APIs
2. Create Retrofit instance using Retrofit.Builder
3. Let Retrofit class generates an implementation of Interface
4. Make a Synchronous or Asynchronous HTTP request

Retrofit work flow

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

The `Retrofit` class generates an implementation of the `GitHubService` interface.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

Each `Call` from the created `GitHubService` can make a synchronous or asynchronous HTTP request to the remote webserver.

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

Retrofit converter

- Retrofit is the class through which your API interfaces are turned into callable objects. By default, Retrofit will give you sane defaults for your platform but it allows for customization.
- By default, Retrofit can only deserialize HTTP bodies into OkHttp's ResponseBody type and it can only accept its RequestBody type for @Body.
- Converters can be added to support other types. Six sibling modules adapt popular serialization libraries for your convenience

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- JAXB: `com.squareup.retrofit2:converter-jaxb`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

Retrofit Call

- After have class generates implementation of Retrofit, you can make synchronous request by calling **execute** method, and asynchronous request by calling **enqueue** method

```
UserService service = ServiceGenerator.createService(UserService.class);

// 1. Calling '/api/users/2' - synchronously

Call<UserApiResponse> callSync = service.getUser(2);

try
{
    Response<UserApiResponse> response = callSync.execute();
    UserApiResponse apiResponse = response.body();

    //API response
    System.out.println(apiResponse);
}
catch (Exception ex)
{
    ex.printStackTrace();
}
```

```
UserService service = ServiceGenerator.createService(UserService.class);

// 2. Calling '/api/users/2' - asynchronously

Call<UserApiResponse> callAsync = service.getUser(2);

callAsync.enqueue(new Callback<UserApiResponse>()
{

    @Override
    public void onResponse(Call<UserApiResponse> call, Response<UserApiResponse> response)
    {
        if (response.isSuccessful())
        {
            UserApiResponse apiResponse = response.body();

            //API response
            System.out.println(apiResponse);
        }
        else
        {
            System.out.println("Request Error :: " + response.errorBody());
        }
    }

    @Override
    public void onFailure(Call<UserApiResponse> call, Throwable t)
    {
        System.out.println("Network Error :: " + t.getLocalizedMessage());
    }
});
```

Retrofit with coroutines

- We can add suspend keyword to api declaration then make retrofit api call in coroutines scope easily

```
interface PokemonService {
    @GET("pokemon")
    suspend fun getAllPokemons(): PokemonResponse
}
```

```
class PokemonRetrofitFragment : Fragment() {

    private val job = Job()
    private val mainScope = CoroutineScope(Dispatchers.IO + job)

    private lateinit var binding: FragmentDataBinding
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentDataBinding.inflate(inflater)
        mainScope.launch {
            val pokemons = PokemonClient.retrofitService.getAllPokemons().results
            withContext(Dispatchers.Main) {
                binding.rvData.adapter = PokemonAdapter(pokemons)
            }
        }
        return binding.root
    }
}
```

Retrofit with RxJava

- Retrofit also support RxJava call adapter by add a bit to the retrofit config

✓ Add rxjava-retrofit-adapter to build.gradle

✓ Return Observable instead of Call in Retrofit Interface

✓ Add RxJava as Call Adapter Factory

✓ Use Retrofit like Rx call

Retrofit with RxJava

1. Add library

```
//RxJava
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'

// Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation "com.github.akarnokd:rxjava3-retrofit-adapter:3.0.0"
```

2. Return Observable in Interface

```
interface PokemonService {
    @GET("pokemon")
    fun getAllPokemons(): Observable<PokemonResponse>
}
```

3. Add RxJava as Call Adapter

```
private fun retrofit(): Retrofit {
    return Retrofit.Builder()
        .addConverterFactory(GsonConverterFactory.create())
        .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
        .baseUrl(Const.BASE_URL)
        .build()
}
```

4. Using Retrofit like a Rx call

```
PokemonClient.retrofitService.getAllPokemons()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeOn(Schedulers.io())
    .subscribe {
        Log.d("doanpt", "retrofit get ${it.results.size}")
        binding.rvData.adapter = PokemonAdapter(it.results)
    }
```

Retrofit *API Declaration*

- Annotations on the interface methods and its parameters indicate how a request will be handled

@GET	@FormUrlEncoded	@Query
@POST	@Multipart	@QueryMap
@PUT	@Headers	@Body
@PATCH	@Header	@Path
@DELETE	@HeaderMap	@ Url

URL Manipulation

Sample GET annotation with @Path, @Query, @QueryMap and @Url

```
interface JsonService {
    @GET("posts")
    suspend fun getPosts(): List<Post>

    @GET("posts")
    suspend fun getPosts(
        @Query("userId") userId: Array<Int>,
        @Query("_sort") sort: String,
        @Query("_order") order: String
    ): List<Post>

    @GET("posts")
    suspend fun getPosts(@QueryMap parameters: Map<String, String>): List<Post>

    @GET("posts/{id}/comments")
    suspend fun getComments(@Path("id") postId: Int): List<Comment>

    @GET
    suspend fun getComments(@Url url: String): List<Comment>
}
```

➔ http://jsonplaceholder.typicode.com/posts

➔ http://jsonplaceholder.typicode.com/posts?userId=5&_sort=id&_order=desc

➔ Add parameter as map values passed same with Query annotation above

➔ http://jsonplaceholder.typicode.com/posts/50/comments

➔ Do get action as url parameter passed

POST annotation

Post annotation used to create/add action

```
//Param will be passed via body
@POST("posts")
suspend fun createPost(@Body post: Post): Post


//Param will be passed via url, ex: http://xxx.com/post?userId=123&title="New%20Text&body=abc
@FormUrlEncoded
@POST("posts")
suspend fun createPost(
    @Field("userId") userId: Int,
    @Field("title") title: String,
    @Field("body") text: String
): Post

//Param will be passed via url, ex: http://xxx.com/post?userId=123&title="New%20Text&body=abc
@FormUrlEncoded
@POST("posts")
suspend fun createPost(@FieldMap fields: Map<String, String>): Post
```

Update and Delete action

- We can use PUT, PATCH, DELETE to perform update actions
  - ✓ PUT annotation will replace an existing object by another
  - ✓ PATCH annotation will update a field
  - ✓ DELETE annotation used to delete action

```
@PUT("posts/{id}")
suspend fun putPost(@Path("id") id: Int, @Body post: Post): Post

@PATCH("posts/{id}")
suspend fun patchPost(@Path("id") id: Int, @Body post: Post): Post

@DELETE("posts/{id}")
suspend fun deletePost(@Path("id") id: Int): Response<Unit>
```

Add Header to request

- We can add Headers to request by using Header annotation

```
@Headers("Static-Header1: 123", "Static-Header2: 456")
@PUT("posts/{id}")
suspend fun putPost(
    @Header("Dynamic-Header") header: String,
    @Path("id") id: Int,
    @Body post: Post
): Post

@PATCH("posts/{id}")
suspend fun patchPost(
    @HeaderMap headers: Map<String, String>,
    @Path("id") id: Int,
    @Body post: Post
): Post
```

Add Header to request

- If you want to add Header to every request, you can use Interceptor then add it to okHttp and Retrofit instance

```
val headerInterceptor = Interceptor {
    val originalRequest: Request = it.request()
    val newRequest: Request = originalRequest.newBuilder()
        .header("Interceptor-Header", "headerInterceptor")
        .build()
    it.proceed(newRequest)
}

val okHttpClient = OkHttpClient.Builder()
    .addInterceptor(headerInterceptor)
    .addInterceptor(loggingInterceptor)
    .build()

return Retrofit.Builder()
    .addConverterFactory(GsonConverterFactory.create())
    .client(okHttpClient)
    .baseUrl(Const.BASE_JSON_API_URL)
    .build()
```

Logging with Retrofit

- To print request log with retrofit, we use logging interceptor of ohHttp library
- There are some levels of log: **BASIC**, **BODY**, **HEADER**, **NONE**

```
private fun retrofit(): Retrofit {
    val loggingInterceptor = HttpLoggingInterceptor()
    loggingInterceptor.level = HttpLoggingInterceptor.Level.BASIC

    val okHttpClient = OkHttpClient.Builder()
        .addInterceptor(loggingInterceptor)
        .build()

    return Retrofit.Builder()
        .addConverterFactory(GsonConverterFactory.create())
        .client(okHttpClient)
        .baseUrl(Const.BASE_JSON_API_URL)
        .build()
}
```

References

1. https://github.com/doanpt/android-learing/tree/pokedex
2. https://square.github.io/retrofit/
3. https://codinginflow.com/tutorials/android/retrofit/part-1-simple-get-request
4. https://codinginflow.com/tutorials/android/gson/part-1-simple-serialization-deserialization
5. https://developer.android.com/reference/java/net/HttpURLConnection
6. https://developer.android.com/training/basics/network-ops/xml
7. https://o7planning.org/10459/android-json-parser

Thank you