|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. **INTEGRATE MOBILE APP WITH DATA SOURCE.** 5. VERIFY SUCCESSFUL API INTEGRATION 6. VERIFY DEVELOPED MOBILE APP. 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/I(6/15) | Page: 1 of 18 |

**TITLE**:

**APPLICATION PROGRAMMING INTERFACE (API)**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of API.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of API.

# **API**

An Application Programming Interface (API) is a particular set of rules (‘code’) and specifications that programs can follow to communicate with each other. APIs are growing exponentially every year.

The world of software moves fast. In earlier days data entered and processed in the same system, but now the origin of the data and processing place is entirely different. We should be able to access the data from anywhere at any time, that’s why we store this data in cloud storage.

For sending and receiving data from/to the server, we want a middleman who is platform-independent. That middleman handles the requests and serves the response to the user. The below diagram illustrates this better than words.

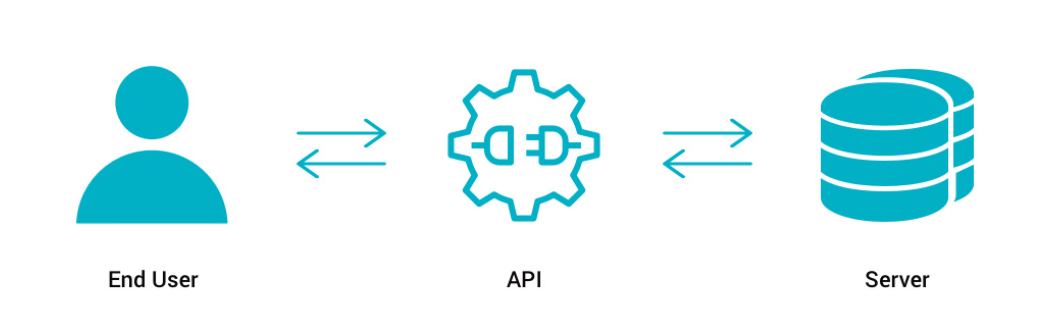


Figure 1: Illustration of API in The Simplest Form

The End user sends a request, API executes the instruction then get the data from the server and respond to the user.

Think about it this way:

You are at a restaurant, and you want to eat something, place your order to the waiter he conveys the order to the kitchen and serves it to you once it ready. Here the waiter is ACT as an API. Customer may change but the waiter serves them in the same manner. The following diagram explains it simply.

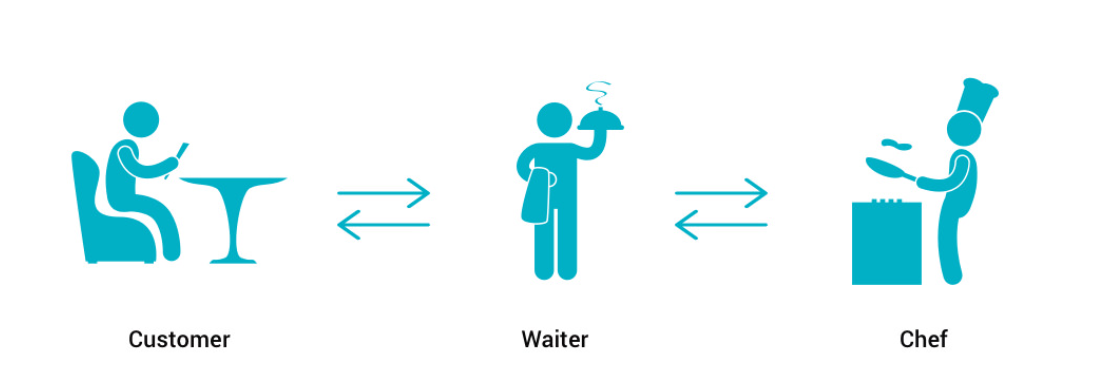


Figure 2: Illustration of API in Layman Term

## Types of API

1. SOAP API (Simple Object Access Protocol)
2. REST API (Representational state transfer)

## What are web APIs?

It is used for exchanging information with a website, either by receiving or by sending data. A web API typically consists of multiple publicly exposed endpoints that accept HTTP requests and respond with the requested data, typically in the form of JSON or XML. Some well-known web API’s: Google, Facebook, Flickr, Twitter.

## API call in Android?

Performing API call in Android is a little bit difficult because first, you need to check the internet connection. If it’s available, then only you can make a request. And you need to handle the cache also. In the android setup, we will focus on the method of Volley to call API in the mobile app.

# **VOLLEY OVERVIEW**

Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster. Volley is available on GitHub. Volley offers the following benefits:

* Automatic scheduling of network requests.
* Multiple concurrent network connections.
* Transparent disk and memory response caching with standard HTTP cache coherence.
* Support for request prioritization.
* Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
* Ease of customization, for example, for retry and backoff.
* Strong ordering makes it easy to correctly populate your UI with data fetched asynchronously from the network.
* Debugging and tracing tools.

Volley excels at RPC-type operations used to populate a UI, such as fetching a page of search results as structured data. It integrates easily with any protocol and comes out of the box with support for raw strings, images, and JSON. By providing built-in support for the features you need, Volley frees you from writing boilerplate code and allows you to concentrate on the logic that is specific to your app.

Volley is not suitable for large download or streaming operations, since Volley holds all responses in memory during parsing. For large download operations, consider using an alternative like DownloadManager.

The core Volley library is developed on GitHub and contains the main request dispatch pipeline as well as a set of commonly applicable utilities, available in the Volley "toolbox." The easiest way to add Volley to your project is to add the following dependency to your app's build.gradle file:

dependencies {

...

implementation("com.android.volley:volley:1.1.1")

}

You can also clone the Volley repository and set it as a library project:

* Git clone the repository by typing the following at the command line:

git clone https://github.com/google/volley

* Import the downloaded source into your app project as an Android library module as described in Create an Android Library.

## Sending a simple request

At a high level, you use Volley by creating a RequestQueue and passing it Request objects. The RequestQueue manages worker threads for running the network operations, reading from and writing to the cache, and parsing responses. Requests do the parsing of raw responses and Volley takes care of dispatching the parsed response back to the main thread for delivery.

This lesson describes how to send a request using the Volley.newRequestQueue convenience method, which sets up a RequestQueue for you.

Add the INTERNET permission

To use Volley, you must add the android.permission.INTERNET permission to your app's manifest. Without this, your app won't be able to connect to the network.

1. Use newRequestQueue

Volley provides a convenience method Volley.newRequestQueue that sets up a RequestQueue for you, using default values, and starts the queue. For example:

val textView = findViewById<TextView>(R.id.text)

// Instantiate the RequestQueue.

val queue = Volley.newRequestQueue(this)

val url = "https://www.google.com"

// Request a string response from the provided URL.

val stringRequest = StringRequest(Request.Method.GET, url,

Response.Listener<String> { response ->

// Display the first 500 characters of the response string.

textView.text = "Response is: ${response.substring(0, 500)}"

},

Response.ErrorListener { textView.text = "That didn't work!" })

// Add the request to the RequestQueue.

queue.add(stringRequest)

Volley always delivers parsed responses on the main thread. Running on the main thread is convenient for populating UI controls with received data, as you can freely modify UI controls directly from your response handler, but it's especially critical to many of the important semantics provided by the library, particularly related to canceling requests.

See Setting Up a RequestQueue for a description of how to set up a RequestQueue yourself, instead of using the Volley.newRequestQueue convenience method.

1. Send a request

To send a request, you simply construct one and add it to the RequestQueue with add(), as shown above. Once you add the request it moves through the pipeline, gets serviced, and has its raw response parsed and delivered.

When you call add(), Volley runs one cache processing thread and a pool of network dispatch threads. When you add a request to the queue, it is picked up by the cache thread and triaged: if the request can be serviced from cache, the cached response is parsed on the cache thread and the parsed response is delivered on the main thread. If the request cannot be serviced from cache, it is placed on the network queue. The first available network thread takes the request from the queue, performs the HTTP transaction, parses the response on the worker thread, writes the response to cache, and posts the parsed response back to the main thread for delivery.

Note that expensive operations like blocking I/O and parsing/decoding are done on worker threads. You can add a request from any thread, but responses are always delivered on the main thread.

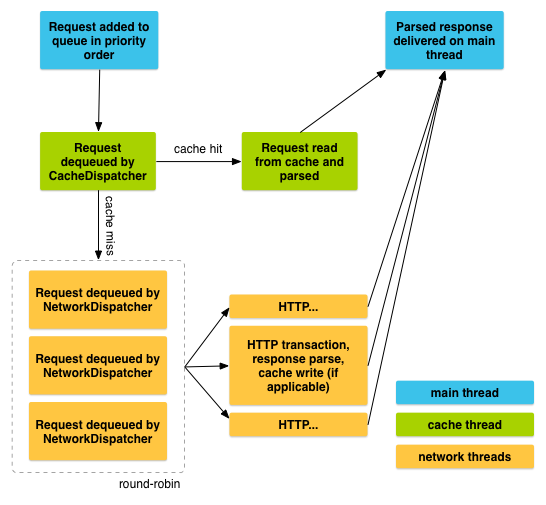


Figure 1: Life of a Request

1. Cancel a request

To cancel a request, call cancel() on your Request object. Once cancelled, Volley guarantees that your response handler will never be called. What this means in practice is that you can cancel all of your pending requests in your activity's onStop() method and you don't have to litter your response handlers with checks for getActivity() == null, whether onSaveInstanceState() has been called already, or other defensive boilerplate.

To take advantage of this behavior, you would typically have to track all in-flight requests in order to be able to cancel them at the appropriate time. There is an easier way: you can associate a tag object with each request. You can then use this tag to provide a scope of requests to cancel. For example, you can tag all of your requests with the Activity they are being made on behalf of, and call requestQueue.cancelAll(this) from onStop(). Similarly, you could tag all thumbnail image requests in a ViewPager tab with their respective tabs and cancel on swipe to make sure that the new tab isn't being held up by requests from another one.

Here is an example that uses a string value for the tag:

1. Define your tag and add it to your requests.

val TAG = "MyTag"

val stringRequest: StringRequest // Assume this exists.

val requestQueue: RequestQueue? // Assume this exists.

// Set the tag on the request.

stringRequest.tag = TAG

// Add the request to the RequestQueue.

requestQueue?.add(stringRequest)

1. In your activity's onStop() method, cancel all requests that have this tag.

protected fun onStop() {

super.onStop()

requestQueue?.cancelAll(TAG)

}

Take care when canceling requests. If you are depending on your response handler to advance a state or kick off another process, you need to account for this. Again, the response handler will not be called.

https://developer.android.com/training/volley/simple

## Setting up RequestQueue

The previous lesson showed you how to use the convenience method Volley.newRequestQueue to set up a RequestQueue, taking advantage of Volley's default behaviors. This lesson walks you through the explicit steps of creating a RequestQueue, to allow you to supply your own custom behavior.

This lesson also describes the recommended practice of creating a RequestQueue as a singleton, which makes the RequestQueue last the lifetime of your app.

1. Set up a network and cache

A RequestQueue needs two things to do its job: a network to perform transport of the requests, and a cache to handle caching. There are standard implementations of these available in the Volley toolbox: DiskBasedCache provides a one-file-per-response cache with an in-memory index, and BasicNetwork provides a network transport based on your preferred HTTP client.

BasicNetwork is Volley's default network implementation. A BasicNetwork must be initialized with the HTTP client your app is using to connect to the network. Typically this is an HttpURLConnection.

This snippet shows you the steps involved in setting up a RequestQueue:

// Instantiate the cache

val cache = DiskBasedCache(cacheDir, 1024 \* 1024) // 1MB cap

// Set up the network to use HttpURLConnection as the HTTP client.

val network = BasicNetwork(HurlStack())

// Instantiate the RequestQueue with the cache and network. Start the queue.

val requestQueue = RequestQueue(cache, network).apply {

start()

}

val url = "http://www.example.com"

// Formulate the request and handle the response.

val stringRequest = StringRequest(Request.Method.GET, url,

Response.Listener<String> { response ->

// Do something with the response

},

Response.ErrorListener { error ->

// Handle error

textView.text = "ERROR: %s".format(error.toString())

})

// Add the request to the RequestQueue.

requestQueue.add(stringRequest)

If you just need to make a one-time request and don't want to leave the thread pool around, you can create the RequestQueue wherever you need it and call stop() on the RequestQueue once your response or error has come back, using the Volley.newRequestQueue() method described in Sending a Simple Request. But the more common use case is to create the RequestQueue as a singleton to keep it running for the lifetime of your app, as described in the next section.

1. Use a singleton pattern

If your application makes constant use of the network, it's probably most efficient to set up a single instance of RequestQueue that will last the lifetime of your app. You can achieve this in various ways. The recommended approach is to implement a singleton class that encapsulates RequestQueue and other Volley functionality. Another approach is to subclass Application and set up the RequestQueue in Application.onCreate(). But this approach is discouraged; a static singleton can provide the same functionality in a more modular way.

A key concept is that the RequestQueue must be instantiated with the Application context, not an Activity context. This ensures that the RequestQueue will last for the lifetime of your app, instead of being recreated every time the activity is recreated (for example, when the user rotates the device).

Here is an example of a singleton class that provides RequestQueue and ImageLoader functionality:

class MySingleton constructor(context: Context) {

companion object {

@Volatile

private var INSTANCE: MySingleton? = null

fun getInstance(context: Context) =

INSTANCE ?: synchronized(this) {

INSTANCE ?: MySingleton(context).also {

INSTANCE = it

}

}

}

val imageLoader: ImageLoader by lazy {

ImageLoader(requestQueue,

object : ImageLoader.ImageCache {

private val cache = LruCache<String, Bitmap>(20)

override fun getBitmap(url: String): Bitmap {

return cache.get(url)

}

override fun putBitmap(url: String, bitmap: Bitmap) {

cache.put(url, bitmap)

}

})

}

val requestQueue: RequestQueue by lazy {

// applicationContext is key, it keeps you from leaking the

// Activity or BroadcastReceiver if someone passes one in.

Volley.newRequestQueue(context.applicationContext)

}

fun <T> addToRequestQueue(req: Request<T>) {

requestQueue.add(req)

}

}

Here are some examples of performing RequestQueue operations using the singleton class:

// Get a RequestQueue

val queue = MySingleton.getInstance(this.applicationContext).requestQueue

// ...

// Add a request (in this example, called stringRequest) to your RequestQueue.

MySingleton.getInstance(this).addToRequestQueue(stringRequest)

https://developer.android.com/training/volley/requestqueue

## Making a standard request

This lesson describes how to use the common request types that Volley supports:

1. StringRequest. Specify a URL and receive a raw string in response. See Setting Up a Request Queue for an example.
2. JsonObjectRequest and JsonArrayRequest (both subclasses of JsonRequest). Specify a URL and get a JSON object or array (respectively) in response.

If your expected response is one of these types, you probably don't have to implement a custom request. This lesson describes how to use these standard request types. For information on how to implement your own custom request, see Implementing a Custom Request.

1. Request JSON

Volley provides the following classes for JSON requests:

1. JsonArrayRequest—A request for retrieving a JSONArray response body at a given URL.
2. JsonObjectRequest—A request for retrieving a JSONObject response body at a given URL, allowing for an optional JSONObject to be passed in as part of the request body.

Both classes are based on the common base class JsonRequest. You use them following the same basic pattern you use for other types of requests. For example, this snippet fetches a JSON feed and displays it as text in the UI:

val url = "http://my-json-feed"

val jsonObjectRequest = JsonObjectRequest(Request.Method.GET, url, null,

Response.Listener { response ->

textView.text = "Response: %s".format(response.toString())

},

Response.ErrorListener { error ->

// TODO: Handle error

}

)

// Access the RequestQueue through your singleton class.

MySingleton.getInstance(this).addToRequestQueue(jsonObjectRequest)

For an example of implementing a custom JSON request based on Gson, see the next lesson, Implement a custom request.

https://developer.android.com/training/volley/request

# **IMPLEMENTATION OF CUSTOM REQUEST**

## Writing a custom request

Most requests have ready-to-use implementations in the toolbox; if your response is a string, image, or JSON, you probably won't need to implement a custom Request.

For cases where you do need to implement a custom request, this is all you need to do:

* Extend the Request<T> class, where <T> represents the type of parsed response the request expects. So if your parsed response is a string, for example, create your custom request by extending Request<String>. See the Volley toolbox classes StringRequest and ImageRequest for examples of extending Request<T>.
* Implement the abstract methods parseNetworkResponse() and deliverResponse(), described in more detail below.

## parseNetworkResponse

A Response encapsulates a parsed response for delivery, for a given type (such as string, image, or JSON). Here is a sample implementation of parseNetworkResponse():

override fun parseNetworkResponse(response: NetworkResponse?): Response<T> {

return try {

val json = String(

response?.data ?: ByteArray(0),

Charset.forName(HttpHeaderParser.parseCharset(response?.headers)))

Response.success(

gson.fromJson(json, clazz),

HttpHeaderParser.parseCacheHeaders(response))

}

// handle errors

// ...

}

Note the following:

* parseNetworkResponse() takes as its parameter a NetworkResponse, which contains the response payload as a byte[], HTTP status code, and response headers.
* Your implementation must return a Response<T>, which contains your typed response object and cache metadata or an error, such as in the case of a parse failure.

If your protocol has non-standard cache semantics, you can build a Cache.Entry yourself, but most requests are fine with something like this:

return Response.success(myDecodedObject,

HttpHeaderParser.parseCacheHeaders(response))

Volley calls parseNetworkResponse() from a worker thread. This ensures that expensive parsing operations, such as decoding a JPEG into a Bitmap, don't block the UI thread.

1. deliverResponse

Volley calls you back on the main thread with the object you returned in parseNetworkResponse(). Most requests invoke a callback interface here, for example:

override fun deliverResponse(response: T) = listener.onResponse(response)

1. Example: GsonRequest

Gson is a library for converting Java objects to and from JSON using reflection. You can define Java objects that have the same names as their corresponding JSON keys, pass Gson the class object, and Gson will fill in the fields for you. Here's a complete implementation of a Volley request that uses Gson for parsing:

/\*\*

\* Make a GET request and return a parsed object from JSON.

\*

\* @param url URL of the request to make

\* @param clazz Relevant class object, for Gson's reflection

\* @param headers Map of request headers

\*/

class GsonRequest<T>(

url: String,

private val clazz: Class<T>,

private val headers: MutableMap<String, String>?,

private val listener: Response.Listener<T>,

errorListener: Response.ErrorListener

) : Request<T>(Method.GET, url, errorListener) {

private val gson = Gson()

override fun getHeaders(): MutableMap<String, String> = headers ?: super.getHeaders()

override fun deliverResponse(response: T) = listener.onResponse(response)

override fun parseNetworkResponse(response: NetworkResponse?): Response<T> {

return try {

val json = String(

response?.data ?: ByteArray(0),

Charset.forName(HttpHeaderParser.parseCharset(response?.headers)))

Response.success(

gson.fromJson(json, clazz),

HttpHeaderParser.parseCacheHeaders(response))

} catch (e: UnsupportedEncodingException) {

Response.error(ParseError(e))

} catch (e: JsonSyntaxException) {

Response.error(ParseError(e))

}

}

}

**QUESTIONS:**

1. Explain the type of API available in the market.

**Answer:**

* + 1. SOAP API (Simple Object Access Protocol)
    2. REST API (Representational state transfer)

(2 Marks)

**REFERENCE:**

1. https://developer.android.com/training/volley/request-custom