|  |  |  |
| --- | --- | --- |
|  | **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/IS(8/15) | Page: 1 of |

**TITLE**:

**SERVER RESPONSE MAPPING**

**PURPOSE**:

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

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of server response mapping.

# **HANDLING NETWORK HTTP RESPONSE OF SUCCESS DATA AND FAILURE**

Requesting data to a remote server and handling responses is always one of the important issues in Android development. When we are building our project, even a small-sized demo project, we have to spend time constructing the REST service structure. Using the modern Android development tool like Retrofit2, OkHttp, LiveData, or RxJava we can implement the network interfaces easily and request asynchronously. But designing standardized response models/interfaces and handling success data and errors are also important things.



Figure 1: Requesting Data

We’re going to find out how to build a network structure and handle success, failure data from the responses using the Sandwich library.

## Sandwich library

Including

We should import the dependency of the library to our build.gradle file.

dependencies {

implementation "com.github.skydoves:sandwich:1.0.1"

}

1. ApiResponse

ApiResponse is an interface of the retrofit response for handling data and error response with useful transformation extensions. We can get ApiResponse using the scope extension request from the Call.

Let’s see how to get and use the ApiResponse from the DisneyService.

interface DisneyService {

@GET("/")

fun fetchDisneyPosterList(): Call<List<Poster>>

}

val disneyService = retrofit.create(DisneyService::class.java)

// Asynchronously request REST call and get an ApiResponse model.

disneyService.fetchDisneyPosterList().request { response ->

when (response) {

is ApiResponse.Success -> {

// stub success case

livedata.post(response.data)

}

is ApiResponse.Failure.Error -> {

// stub error case

}

is ApiResponse.Failure.Exception -> {

// stub exception case

}

}

}

After getting an instance of the disneyService from the retrofit.create, we can call the the fetchDisneyPosterList().reqeust. The reqeust method receives a lambda function and we can get an instance of theApiResponse via reqeust lambda. There are three types of ApiResponse.

1. ApiResponse.Success

API success response class from the Retrofit service.

We can get the response body data, StatusCode, Headers and etc from the ApiResponse.Success. If we received 2xx status code from the remote server, we can get the data from the ApiResponse.Success.

val data: List<Poster>? = response.data

val statusCode: StatusCode = response.statusCode

val headers: Headers = response.headers

1. ApiResponse.Failure.Error

API format does not match, or applications need to handle errors.e.g. Internal server error (4xx, 5xx server error).

val errorBody: ResponseBody? = response.errorBody

val statusCode: StatusCode = response.statusCode

val headers: Headers = response.headers

1. ApiResponse.Failure.Exception

An unexpected exception occurs while creating the request, processing the response, or parsing error in the client. The problem is the client, so we can’t get an error body or status code. Instead, we can get an exception.

e.g. Network connection error.

1. ApiResponse Extensions

We can handle response cases handier using extensions using onSuccess, onError, onException. We can use these scope functions to the ApiResponse, it reduces the usage of the if/when clause.

disneyService.fetchDisneyPosterList().request { response ->

response.onSuccess {

// stub success case

livedata.post(data)

}.onError {

// stub error case

}.onException {

// stub exception case

}

}

1. ApiErrorModelMapper

Sometimes we have to get our own error response model from the error response. We can map ApiResponse.Failure.Error model to our customized error model using the mapper. Below example shows how to transform ApiResponse.Failure.Error to ErrorEnvelope via ErrorEnvelopeMapper.

data class ErrorEnvelope(

val code: Int,

val message: String

)

object ErrorEnvelopeMapper : ApiErrorModelMapper<ErrorEnvelope> {

override fun map(apiErrorResponse: ApiResponse.Failure.Error<\*>): ErrorEnvelope {

return ErrorEnvelope(apiErrorResponse.statusCode.code, apiErrorResponse.message())

}

}

// hadling the error response case.

response.onError {

// map the ApiResponse.Failure.Error to a customized error model using the mapper.

map(ErrorEnvelopeMapper) {

val code = this.code

val message = this.message

}

}

1. ResponseDataSource

ResponseDataSource is an implementation of the DataSource interface. We can bind an instance of the Call to the ResponseDataSource. The ResponseDataSource requests asynchronously, and it holds data if the response success. we can observe every response. Also, we can retry fetching data (re-request) if our request got failure.

1. Asynchronously send requests.
2. A response data holder from the REST API call for caching data on memory.
3. Observable for every response.
4. Retry fetching data when the request gets failure.
5. Concat another DataSource and request sequentially.
6. Combine

Combine a Call and lambda scope for constructing the DataSource.

val disneyService = retrofit.create(DisneyService::class.java)

val dataSource = ResponseDataSource<List<Poster>>()

dataSource.combine(disneyService.fetchDisneyPosterList()) { response ->

// stubs

}

1. Request

Request API network call asynchronously.

If the request is successful, this data source will hold the success response model. In the next request after the success, request() returns the cached API response. If we need to fetch a new response data or refresh, we can use invalidate().

dataSource.request()

1. Retry

Retry requesting API call when the request gets failure.

dataSource.retry(3, 5000L)

1. ObserveResponse

Observes every response data ApiResponse from the API call request.

dataSource.observeResponse {

Timber.d("observeResponse: $it")

}

1. Invalidate

Invalidate a cached (holding) data and re-fetching the API request.

dataSource.invalidate()

1. Concat

Concat another DataSource and request API call sequentially if the API call getting successful.

val dataSource2 = ResponseDataSource<List<PosterDetails>>()

dataSource2.retry(3, 5000L).combine(disneyService.fetchDetails()) {

// stubs handling dataSource2 response

}

dataSource1

.request() // request() must be called before concat.

.concat(dataSource2) // request dataSource2's API call after the success of the dataSource1.

.concat(dataSource3) // request dataSource3's API call after the success of the dataSource2.

1. Here is the example of the ResponseDataSource in the MainViewModel.

class MainViewModel constructor(

private val disneyService: DisneyService

) : ViewModel() {

// request API call Asynchronously and holding successful response data.

private val dataSource = ResponseDataSource<List<Poster>>()

val posterListLiveData = MutableLiveData<List<Poster>>()

val toastLiveData = MutableLiveData<String>()

/\*\* fetch poster list data from the network. \*/

fun fetchDisneyPosters() {

dataSource

// retry fetching data 3 times with 5000 time interval when the request gets failure.

.retry(3, 5000L)

// combine network service to the data source.

.combine(disneyService.fetchDisneyPosterList()) { response ->

// handle the case when the API request gets a success response.

response.onSuccess {

Timber.d("$data")

posterListLiveData.postValue(data)

}

// handle the case when the API request gets a error response.

// e.g. internal server error.

.onError {

Timber.d(message())

// handling error based on status code.

when (statusCode) {

StatusCode.InternalServerError -> toastLiveData.postValue("InternalServerError")

StatusCode.BadGateway -> toastLiveData.postValue("BadGateway")

else -> toastLiveData.postValue("$statusCode(${statusCode.code}): ${message()}")

}

// map the ApiResponse.Failure.Error to a customized error model using the mapper.

map(ErrorEnvelopeMapper) {

Timber.d(this.toString())

}

}

// handle the case when the API request gets a exception response.

// e.g. network connection error.

.onException {

Timber.d(message())

toastLiveData.postValue(message())

}

}

// observe every API request responses.

.observeResponse {

Timber.d("observeResponse: $it")

}

// request API network call asynchronously.

// if the request is successful, the data source will hold the success data.

// in the next request after success, returns the cached API response.

// if you want to fetch a new response data, use invalidate().

.request()

}

}

1. DataSourceCallAdapterFactory

We can get the DataSource directly from the Retrofit service.

Add a call adapter factory DataSourceCallAdapterFactory to your Retrofit builder. And change the return type of your service Call to DataSource

Retrofit.Builder()

...

.addCallAdapterFactory(DataSourceCallAdapterFactory())

.build()

interface DisneyService {

@GET("DisneyPosters.json")

fun fetchDisneyPosterList(): DataSource<List<Poster>>

}

Here is the example of the DataSource in the MainViewModel.

class MainViewModel constructor(disneyService: DisneyService) : ViewModel() {

// request API call Asynchronously and holding successful response data.

private val dataSource: DataSource<List<Poster>>

init {

Timber.d("initialized MainViewModel.")

dataSource = disneyService.fetchDisneyPosterList()

// retry fetching data 3 times with 5000L interval when the request gets failure.

.retry(3, 5000L)

.observeResponse(object : ResponseObserver<List<Poster>> {

override fun observe(response: ApiResponse<List<Poster>>) {

// handle the case when the API request gets a success response.

response.onSuccess {

Timber.d("$data")

posterListLiveData.postValue(data)

}

}

})

.request() // must call request()

# **FIREBASE PROJECTS**

This page offers brief overviews of several important concepts about Firebase projects. When available, follow the links to find more detailed information about features, services, and even other platforms. At the bottom of this page, find a listing of general best practices for Firebase projects.

## Relationship between Firebase projects, apps, and products

A Firebase project is the top-level entity for Firebase. In a project, you create Firebase apps by registering your iOS, Android, or web apps. After you register your apps with Firebase, you can add the Firebase SDKs for any number of Firebase products, like Analytics, Cloud Firestore, Performance Monitoring, or Remote Config.

Learn more detailed information about this process in the Getting Started guides (iOS | Android | web | Unity | C++).

## Relationship between Firebase projects and Google Cloud

When you create a new Firebase project in the Firebase console, you're actually creating a Google Cloud project behind the scenes. You can think of a Google Cloud project as a virtual container for data, code, configuration, and services. A Firebase project is a Google Cloud project that has additional Firebase-specific configurations and services. You can even create a Google Cloud project first, then add Firebase to the project later.

Since a Firebase project is a Google Cloud project:

1. Projects that appear in the Firebase console also appear in the Google Cloud Console and Google APIs console.
2. Billing and permissions for projects are shared across Firebase and Google Cloud.
3. Unique identifiers for a project (like the project number and the project ID) are shared across Firebase and Google Cloud.
4. You can use products and APIs from both Firebase and Google Cloud in a project.
5. Deleting a project deletes it across Firebase and Google Cloud.

## Setting up a Firebase project and registering apps

You can set up a Firebase project and register apps in the Firebase console (or, for advanced use cases, via the Firebase Management REST API or the Firebase CLI). When you set up a project and register apps, you need to make some organizational decisions and add Firebase-specific configuration information to your local projects.

Make sure to review some general project-level best practices (at the bottom of this page) before setting up a project and registering apps.

1. The project name

When you create a project, you provide a project name. This identifier is the internal-only name for a project in the Firebase console, the Google Cloud Console, and the Firebase CLI. The project name is not exposed in any publicly visible Firebase or Google Cloud product, service, or resource; it simply serves to help you more easily distinguish among multiple projects.

You can edit a project name at any time in the settings Project settings of the Firebase console. The project name is displayed in the top pane.

1. The project number

A Firebase project (and its associated Google Cloud project) has a project number. This is the Google-assigned globally unique canonical identifier for the project. Use this identifier when configuring integrations and/or making API calls to Firebase, Google, or third-party services.

1. API calls and the project number

For many API calls, you need to include a unique identifier for a project. Although many APIs accept the project ID, it's recommended that you use the project number for making API calls to Firebase, Google, or third-party services.

Learn more about using project identifiers, especially the project number, in Google's AIP 2510 standard.

1. Find the project number

* Firebase console: Click settings Project settings. The project number is displayed in the top pane.
* Firebase CLI: Run firebase projects:list. The project number is displayed along with all the Firebase projects associated with your account.
* Firebase Management REST API: Call projects.list. The response body contains the project number in the FirebaseProject object.

1. The project ID

A Firebase project (and its associated Google Cloud project) has a project ID. This is a user-defined unique identifier for the project across all of Firebase and Google Cloud. When you create a Firebase project, Firebase automatically assigns a unique ID to the project, but you can edit it during project setup. This identifier should generally be treated as a convenience alias to reference the project.

If you delete a project, the project ID is also deleted and can never be used again by any other project.

1. Firebase resources and the project ID

The project ID displays in publicly visible Firebase resources, for example:

* Default Hosting subdomain — PROJECT\_ID.web.app and PROJECT\_ID.firebaseapp.com
* Default Realtime Database URL — PROJECT\_ID-default-rtdb.firebaseio.com or PROJECT\_ID-default-rtdb.REGION\_CODE.firebasedatabase.app
* Default Cloud Storage bucket name — PROJECT\_ID.appspot.com

For all of the aforementioned resources, you can create non-default instances. The publicly visible names of non-defaults are fully customizable. You can connect custom domains to a Firebase-hosted site, shard the Realtime Database, and create multiple Cloud Storage buckets (visit the platform-specific Get Started page).

1. The Firebase CLI and the project ID

For some use cases, you might have multiple Firebase projects associated with the same local app directory. In these situations, when you use the Firebase CLI, you need to pass the --project flag with the firebase commands to communicate which Firebase project you want to interact with.

You can also set up a project alias for each Firebase project so that you don't have to remember project IDs.

1. API calls and the project ID

For many API calls, you need to include a unique identifier for a project. Although many APIs accept the project ID, it's recommended that you use the project number for making API calls to Firebase, Google, or third-party services.

Learn more about using project identifiers, especially the project number, in Google's AIP 2510 standard.

1. Find the project ID
2. Firebase console: Click settings Project settings. The project ID is displayed in the top pane.
3. Firebase CLI: Run firebase projects:list. The project ID is displayed along with all the Firebase projects associated with your account.
4. Firebase Management REST API: Call projects.list. The response body contains the project ID in the FirebaseProject object.
5. Firebase config files and objects

When you register an app with a Firebase project, the Firebase console provides a Firebase configuration file (iOS/Android apps) or a configuration object (web apps) that you add directly to your local app directory.

1. For iOS apps, you add a GoogleService-Info.plist configuration file.
2. For Android apps, you add a google-services.json configuration file.
3. For web apps, you add a Firebase configuration object.

At any time, you can obtain an app's Firebase config file or object.

A Firebase config file or object associates an app with a specific Firebase project and its resources (databases, storage buckets, etc.). The configuration includes "Firebase options", which are parameters required by Firebase and Google services to communicate with Firebase server APIs and to associate client data with the Firebase project and Firebase app. Here are the required, minimum "Firebase options":

1. API key: a simple encrypted string used when calling certain APIs that don't need to access private user data (example value: AIzaSyDOCAbC123dEf456GhI789jKl012-MnO)
2. Project ID: a user-defined unique identifier for the project across all of Firebase and Google Cloud. This identifier may appear in URLs or names for some Firebase resources, but it should generally be treated as a convenience alias to reference the project. (example value: myapp-project-123)
3. Application ID ("AppID"): the unique identifier for the Firebase app across all of Firebase with a platform-specific format:

* Firebase iOS apps: GOOGLE\_APP\_ID (example value: 1:1234567890:ios:321abc456def7890).This is not an Apple bundle ID.
* Firebase Android apps: mobilesdk\_app\_id (example value: 1:1234567890:android:321abc456def7890).This is not an Android package name or Android application ID.
* Firebase Web apps: appId (example value: 1:65211879909:web:3ae38ef1cdcb2e01fe5f0c)

The content of the Firebase config file or object is considered public, including the app's platform-specific ID (iOS bundle ID or Android package name) and the Firebase project-specific values, like the API Key, project ID, Realtime Database URL, and Cloud Storage bucket name. Given this, use Firebase Security Rules to protect your data and files in Realtime Database, Cloud Firestore, and Cloud Storage.

For open source projects, we generally do not recommend including the app's Firebase config file or object in source control because, in most cases, your users should create their own Firebase projects and point their apps to their own Firebase resources (via their own Firebase config file or object).

## Managing a Firebase project

Make sure to review the general project-level best practices for considerations that might affect how you manage a Firebase project.

1. Tools to manage a project

* Firebase console

The Firebase console offers the richest environment for managing Firebase products, apps, and project-level settings.

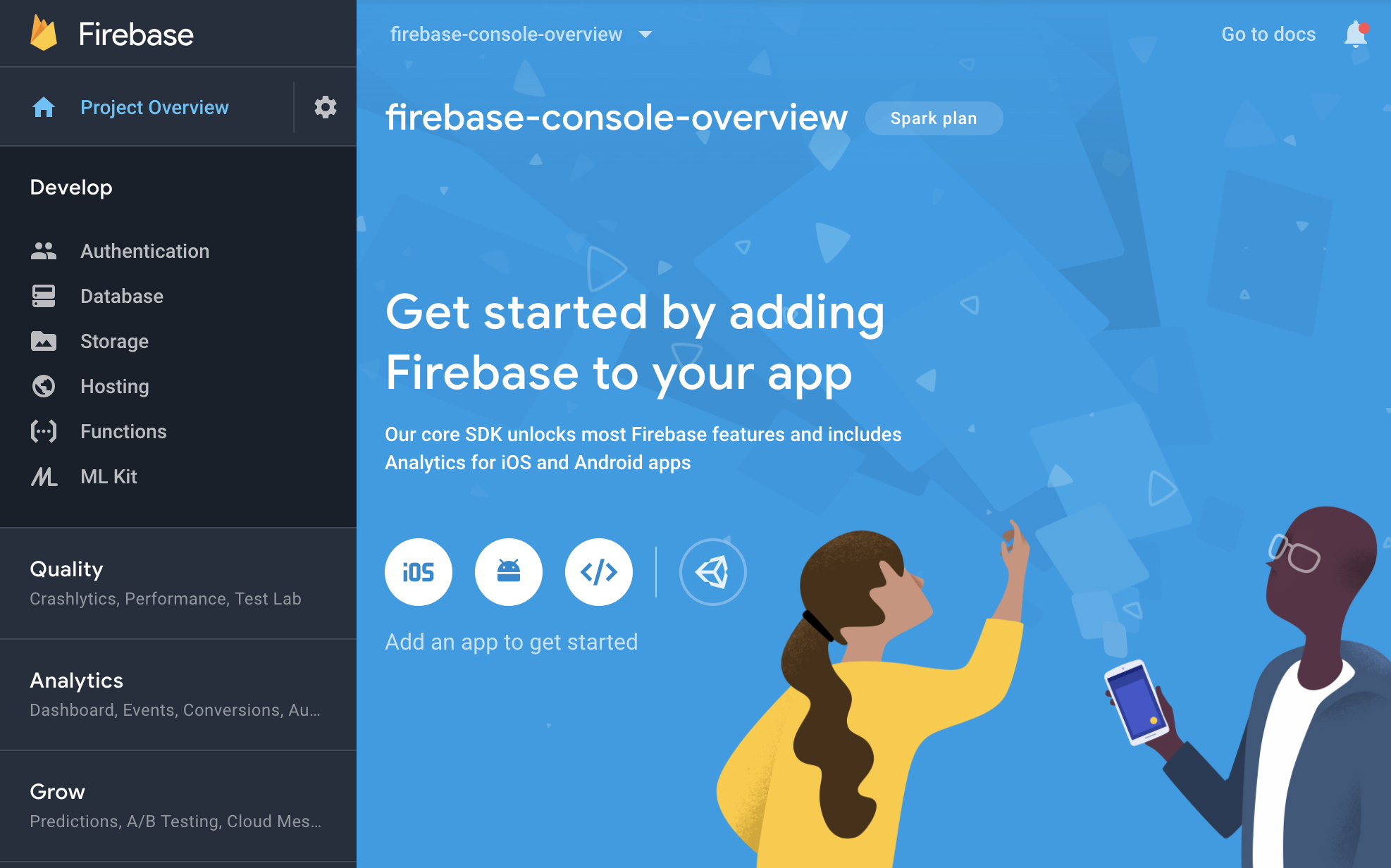


Figure 2: Firebase Console Overview

The left-side panel of the console lists the Firebase products, organized by top-level categories. At the top of the left-side panel, access a project's settings by clicking settings. A project's settings include integrations, access permissions, and billing.

The middle of the console displays buttons that launch setup workflows to register various types of apps. After you start using Firebase, the main area of the console changes into a dashboard that displays stats on the products you use.

* Firebase CLI (a command line tool)

Firebase also offers the Firebase CLI for configuring and managing specific Firebase products, like Firebase Hosting and Cloud Functions for Firebase.

After installing the CLI, you have access to the global firebase command. Use the CLI to link your local app directory to a Firebase project, then deploy new versions of Firebase-hosted content or updates to functions.

## Firebase Management REST API

Using the Firebase Management REST API, you can programmatically manage a Firebase project. For example, you can programmatically register an app with a project or list the apps that are already registered (iOS | Android | web).

1. General best practices

Adding apps to a project

Ensure that all apps within a project are platform variants of the same application from an end-user perspective. It's advisable to register the iOS, Android, and web versions of the same app or game with the same Firebase project. All the apps in a project generally share the same Firebase resources (database, storage buckets, etc.).

If you have multiple build variants with different iOS bundle IDs or Android package names defined, you can register each variant with a separate Firebase project. However, if you have variants that share the same Firebase resources, register them with the same Firebase project.

Here are some general limits for Firebase projects, apps, and sites:

1. Number of projects per account
2. Spark pricing plan — Project-creation quota is limited to a lower count of projects (usually around 5-10).
3. Blaze pricing plan — Project-creation quota per account increases substantially as long as the associated Cloud Billing account is in good standing.

The limit on project-creation quota is rarely a concern for most developers, but if needed, you can request an increase in project quota.

Be aware that the complete deletion of a project requires 30 days and counts toward project quota until the project is fully deleted.

1. Number of apps per project

Firebase restricts the total number of Firebase Apps within a Firebase project to 30.

You should ensure that all Firebase Apps within a single Firebase project are platform variants of the same application from an end-user perspective. For example, if you develop a white label application, each independently labelled app should have its own Firebase project.

1. Number of Hosting sites per project

The Firebase Hosting multisite feature supports a maximum of 36 sites per project.

1. Multi-tenancy

Connecting several different logically independent apps and/or web sites to a single Firebase project (often called "multi-tenancy") is not recommended. Multi-tenancy can lead to serious configuration and data privacy concerns problems, including unintended issues with analytics aggregation, shared authentication, overly-complex database structures, and difficulties with security rules.

Generally, if a set of apps don't share the same data and configurations, strongly consider registering each app with a different Firebase project.

For example, if you develop a white label application, each independently labelled app should have its own Firebase project. Each app doesn't and shouldn't (for privacy reasons) share data with the others.

**QUESTIONS:**

1. Explain the function of the server response.

**Answer:**

* 1. **implement the network interfaces easily and request asynchronously**

**REFERENCE:**

1. https://proandroiddev.com/handling-network-http-response-of-success-data-and-failure-for-your-android-project-using-sandwich-36db824bd82d
2. <https://firebase.google.com/docs/projects/learn-more>