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

**TITLE**:

**APPLICATION MARKETPLACE PORTAL**

**PURPOSE**:

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

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of APPLICATION MARKETPLACE PORTAL.

# **FIREBASE PROJECTS**

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.

1. 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.

1. Limits for Firebase projects, apps, and sites:

Number of projects per account

Spark pricing plan — Project-creation quota is limited to a lower count of projects (usually around 5-10).

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.

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. Read more about best practices for multi-tenancy below.

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 websites 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, but the iOS and Android versions of that label can be in the same project. Each independently labeled app shouldn't (for privacy reasons) share data with the others

**2. Firebase Realtime Database**

Store and sync data with our NoSQL cloud database. Data is synced across all clients in real time, and remains available when your app goes offline.

The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and synchronized in realtime to every connected client. When you build cross-platform apps with our iOS, Android, and JavaScript SDKs, all of your clients share one Realtime Database instance and automatically receive updates with the newest data.

1. Installation & Setup on Android

Create a Database

1. Navigate to the **Realtime Database** section of the [Firebase console](https://console.firebase.google.com/project/_/database?authuser=1). You'll be prompted to select an existing Firebase project. Follow the database creation workflow.
2. Select a starting mode for your Firebase Security Rules:  
   **Test mode**  
   Good for getting started with the mobile and web client libraries, but allows anyone to read and overwrite your data. After testing, **make sure to review the** [**Understand Firebase Realtime Database Rules**](https://firebase.google.com/docs/database/security?authuser=1) **section.**

**Locked Mode**

Denies all reads and writes from mobile and web clients. Your authenticated application servers can still access your database.

1. Choose a region for the database. Depending on your choice of region, the database namespace will be of the form <databaseName>.firebaseio.com or <databaseName>.<region>.firebasedatabase.app. For more information, see [select locations for your project](https://firebase.google.com/docs/projects/locations?authuser=1#rtdb-locations).
2. Click Done.

b. Add the Realtime Database SDK to your app

Using the Firebase Android BoM, declare the dependency for the Realtime Database Android library in your module (app-level) Gradle file (usually app/build.gradle).

dependencies {

// Import the BoM for the Firebase platform

implementation platform('com.google.firebase:firebase-bom:28.2.1')

// Declare the dependency for the Realtime Database library

// When using the BoM, you don't specify versions in Firebase library dependencies

implementation 'com.google.firebase:firebase-database'

}

c. Configure Realtime Database Rules

The Realtime Database provides a declarative rules language that allows you to define how your data should be structured, how it should be indexed, and when your data can be read from and written to.

1. Write to your database

Retrieve an instance of your database using getInstance() and reference the location you want to write to.

// Write a message to the database

FirebaseDatabase database = FirebaseDatabase.getInstance();

DatabaseReference myRef = database.getReference("message");

myRef.setValue("Hello, World!");

You can save a range of data types to the database this way, including Java objects. When you save an object the responses from any getters will be saved as children of this location.

1. Read from your database

To make your app data update in realtime, you should add a ValueEventListener to the reference you just created.

The onDataChange() method in this class is triggered once when the listener is attached and again every time the data changes, including the children.

// Read from the database

myRef.addValueEventListener(new ValueEventListener() {

@Override

public void onDataChange(DataSnapshot dataSnapshot) {

// This method is called once with the initial value and again

// whenever data at this location is updated.

String value = dataSnapshot.getValue(String.class);

Log.d(TAG, "Value is: " + value);

}

@Override

public void onCancelled(DatabaseError error) {

// Failed to read value

Log.w(TAG, "Failed to read value.", error.toException());

}

});

c. Configure ProGuard

When using Firebase Realtime Database in your app along with ProGuard, you need to consider how your model objects will be serialized and deserialized after obfuscation. If you use DataSnapshot.getValue(Class) or DatabaseReference.setValue(Object) to read and write data, you will need to add rules to the proguard-rules.profile:

# Add this global rule

-keepattributes Signature

# This rule will properly ProGuard all the model classes in

# the package com.yourcompany.models.

# Modify this rule to fit the structure of your app.

-keepclassmembers class com.yourcompany.models.\*\* {

\*;

}

**d. Structure Your Database**

This guide covers some of the key concepts in data architecture and best practices for structuring the JSON data in your Firebase Realtime Database.

Building a properly structured database requires quite a bit of forethought. Most importantly, you need to plan for how data is going to be saved and later retrieved to make that process as easy as possible.

*How data is structured: it's a JSON tree*

All Firebase Realtime Database data is stored as JSON objects. You can think of the database as a cloud-hosted JSON tree. Unlike a SQL database, there are no tables or records. When you add data to the JSON tree, it becomes a node in the existing JSON structure with an associated key. You can provide your own keys, such as user IDs or semantic names, or they can be provided for you using push().

For example, consider a chat application that allows users to store a basic profile and contact list. A typical user profile is located at a path, such as /users/$uid. The user alovelace might have a database entry that looks something like this:

{

"users": {

"alovelace": {

"name": "Ada Lovelace",

"contacts": { "ghopper": true },

},

"ghopper": { ... },

"eclarke": { ... }

}

}

Although the database uses a JSON tree, data stored in the database can be represented as certain native types that correspond to available JSON types to help you write more maintainable code.

* Avoid nesting data

Because the Firebase Realtime Database allows nesting data up to 32 levels deep, you might be tempted to think that this should be the default structure. However, when you fetch data at a location in your database, you also retrieve all of its child nodes. In addition, when you grant someone read or write access at a node in your database, you also grant them access to all data under that node. Therefore, in practice, it's best to keep your data structure as flat as possible.

For an example of why nested data is bad, consider the following multiply-nested structure:

{

// This is a poorly nested data architecture, because iterating the children

// of the "chats" node to get a list of conversation titles requires

// potentially downloading hundreds of megabytes of messages

"chats": {

"one": {

"title": "Historical Tech Pioneers",

"messages": {

"m1": { "sender": "ghopper", "message": "Relay malfunction found. Cause: moth." },

"m2": { ... },

// a very long list of messages

}

},

"two": { ... }

}

}

With this nested design, iterating through the data becomes problematic. For example, listing the titles of chat conversations requires the entire chats tree, including all members and messages, to be downloaded to the client.

* Flatten data structures

If the data is instead split into separate paths, also called denormalization, it can be efficiently downloaded in separate calls, as it is needed. Consider this flattened structure:

{

// Chats contains only meta info about each conversation

// stored under the chats's unique ID

"chats": {

"one": {

"title": "Historical Tech Pioneers",

"lastMessage": "ghopper: Relay malfunction found. Cause: moth.",

"timestamp": 1459361875666

},

"two": { ... },

"three": { ... }

},

// Conversation members are easily accessible

// and stored by chat conversation ID

"members": {

// we'll talk about indices like this below

"one": {

"ghopper": true,

"alovelace": true,

"eclarke": true

},

"two": { ... },

"three": { ... }

},

// Messages are separate from data we may want to iterate quickly

// but still easily paginated and queried, and organized by chat

// conversation ID

"messages": {

"one": {

"m1": {

"name": "eclarke",

"message": "The relay seems to be malfunctioning.",

"timestamp": 1459361875337

},

"m2": { ... },

"m3": { ... }

},

"two": { ... },

"three": { ... }

}

}

It's now possible to iterate through the list of rooms by downloading only a few bytes per conversation, quickly fetching metadata for listing or displaying rooms in a UI. Messages can be fetched separately and displayed as they arrive, allowing the UI to stay responsive and fast.

* Create data that scales

When building apps, it's often better to download a subset of a list. This is particularly common if the list contains thousands of records. When this relationship is static and one-directional, you can simply nest the child objects under the parent.

Sometimes, this relationship is more dynamic, or it may be necessary to denormalize this data. Many times you can denormalize the data by using a query to retrieve a subset of the data, as discussed in Retrieve Data.

But even this may be insufficient. Consider, for example, a two-way relationship between users and groups. Users can belong to a group, and groups comprise a list of users. When it comes time to decide which groups a user belongs to, things get complicated.

What's needed is an elegant way to list the groups a user belongs to and fetch only data for those groups. An index of groups can help a great deal here:

// An index to track Ada's memberships

{

"users": {

"alovelace": {

"name": "Ada Lovelace",

// Index Ada's groups in her profile

"groups": {

// the value here doesn't matter, just that the key exists

"techpioneers": true,

"womentechmakers": true

}

},

...

},

"groups": {

"techpioneers": {

"name": "Historical Tech Pioneers",

"members": {

"alovelace": true,

"ghopper": true,

"eclarke": true

}

},

...

}

}

You might notice that this duplicates some data by storing the relationship under both Ada's record and under the group. Now alovelace is indexed under a group, and techpioneers is listed in Ada's profile. So to delete Ada from the group, it has to be updated in two places.

This is a necessary redundancy for two-way relationships. It allows you to quickly and efficiently fetch Ada's memberships, even when the list of users or groups scales into the millions or when Realtime Database security rules prevent access to some of the records.

This approach, inverting the data by listing the IDs as keys and setting the value to true, makes checking for a key as simple as reading /users/$uid/groups/$group\_id and checking if it is null. The index is faster and a good deal more efficient than querying or scanning the data.

### e. Basic write operations

For basic write operations, you can use setValue() to save data to a specified reference, replacing any existing data at that path. You can use this method to:

* Pass types that correspond to the available JSON types as follows:
  + String
  + Long
  + Double
  + Boolean
  + Map<String, Object>
  + List<Object>
* Pass a custom Java object, if the class that defines it has a default constructor that takes no arguments and has public getters for the properties to be assigned.

If you use a Java object, the contents of your object are automatically mapped to child locations in a nested fashion. Using a Java object also typically makes your code more readable and easier to maintain. For example, if you have an app with a basic user profile, your User object might look as follows:

@IgnoreExtraProperties

public class User {

public String username;

public String email;

public User() {

// Default constructor required for calls to DataSnapshot.getValue(User.class)

}

public User(String username, String email) {

this.username = username;

this.email = email;

}

}

You can add a user with setValue() as follows:

public void writeNewUser(String userId, String name, String email) {

User user = new User(name, email);

mDatabase.child("users").child(userId).setValue(user);

}

Using setValue() in this way overwrites data at the specified location, including any child nodes. However, you can still update a child without rewriting the entire object. If you want to allow users to update their profiles you could update the username as follows:

mDatabase.child("users").child(userId).child("username").setValue(name);

### f. Read data with persistent listeners

To read data at a path and listen for changes, use the addValueEventListener() method to add a ValueEventListener to a DatabaseReference. You can use the onDataChange() method to read a static snapshot of the contents at a given path, as they existed at the time of the event. This method is triggered once when the listener is attached and again every time the data, including children, changes. The event callback is passed a snapshot containing all data at that location, including child data. If there is no data, the snapshot will return false when you call exists() and null when you call getValue() on it.

The following example demonstrates a social blogging application retrieving the details of a post from the database:

ValueEventListener postListener = new ValueEventListener() {

@Override

public void onDataChange(DataSnapshot dataSnapshot) {

// Get Post object and use the values to update the UI

Post post = dataSnapshot.getValue(Post.class);

// ..

}

@Override

public void onCancelled(DatabaseError databaseError) {

// Getting Post failed, log a message

Log.w(TAG, "loadPost:onCancelled", databaseError.toException());

}

};

mPostReference.addValueEventListener(postListener);

The listener receives a DataSnapshot that contains the data at the specified location in the database at the time of the event. Calling getValue() on a snapshot returns the Java object representation of the data. If no data exists at the location, calling getValue() returns null.

In this example, ValueEventListener also defines the onCancelled() method that is called if the read is cancelled. For example, a read can be cancelled if the client doesn't have permission to read from a Firebase database location. This method is passed a DatabaseError object indicating why the failure occurred.

## Work with data offline

If a client loses its network connection, your app will continue functioning correctly.

Every client connected to a Firebase database maintains its own internal version of any data on which listeners are being used or which is flagged to be kept in sync with the server. When data is read or written, this local version of the data is used first. The Firebase client then synchronizes that data with the remote database servers and with other clients on a "best-effort" basis.

As a result, all writes to the database trigger local events immediately, before any interaction with the server. This means your app remains responsive regardless of network latency or connectivity.

Once connectivity is reestablished, your app receives the appropriate set of events so that the client syncs with the current server state, without having to write any custom code.

**QUESTIONS:**

1. Sparkfun package for back end solutions comes with a limited amount of supported projects. Explain the number of projects supported for the spark pricing plan.

Answer:

Spark pricing plan — Project-creation quota is limited to a lower count of projects (usually around 5-10).

**REFERENCE:**

* <https://support.google.com/googleplay/android-developer/answer/9859348#rollout&zippy=%2Capp-signing-by-google-play%2Cwhats-new-in-this-release>
* <https://developer.android.com/studio/publish/upload-bundle>
* <https://developer.android.com/distribute/best-practices/launch/launch-checklist>
* [**https://medium.com/googleplaydev/a-guide-to-the-google-play-console-1bdc79ca956f**](https://medium.com/googleplaydev/a-guide-to-the-google-play-console-1bdc79ca956f)