Android : Using Firebase with Kotlin

In this lesson, we will make use of one of the most powerful real-time database API Google and Android provides to us, Firebase.

Firebase is a real-time database which provide immense features to manage a mobile app. Actually, not even a mobile app, complete web apps and enterprise applications can make use of Firebase with virtually unlimited performance.

This lesson will throw a very deep light on Firebase features and its usage, prominently, using a simple app in Android, a To Do app powered by Firebase Database API. Let’s get started.

Firebase

Firebase offers many features with its strong API and unlimited performance. We will point out some of those here:

- No infrastructure needs to be managed by developers or business holders as it managed by Google itself. Based on needs and access, Firebase can scale automatically.
- Not just the database API, it offers many solutions like Analytics, Notifications, Ad campaigns and much more. Best thing is, all these products work seamlessly together and allows you to focus on Business needs rather than searching for different APIs to work.
- There are free plans across all Firebase products. Even with Database, there is a Spark plan which manages your development

needs and you do not have to worry about paying anything unless you go live and is ready for a broader audience.

- Offline, yes, Firebase works offline. Caching mechanism with Firebase database is just excellent. Firebase Real Time Database SDK persists the app data to disk.

New Project in Android

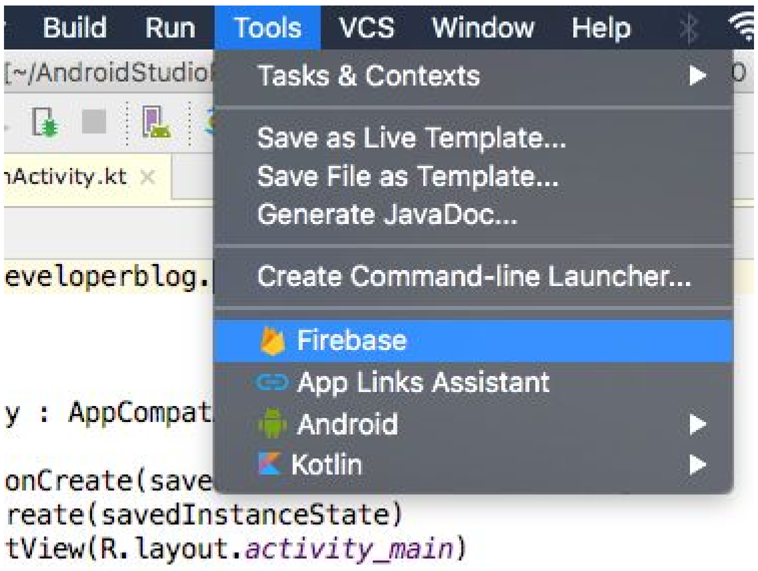
We will be creating a new project in Android to demonstrate a simple app as a To do List app in Firebase Database with completely functional CRUD example.

Creating a project in Android Studio

Let us quickly setup a project in Android Studio with name as *FirebaseToDo*. Our base package will be *com.appsdeveloperblog*. Remember to check the “Include Kotlin support” while setting up the project.

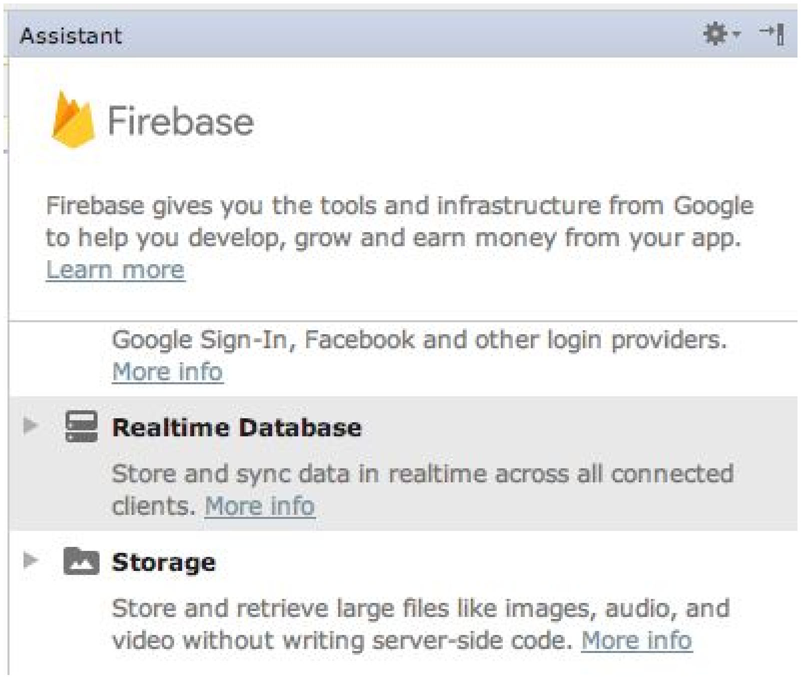
Adding Firebase support

Next, we add Firebase support in our application. This can be done easily from inside the Android Studio itself. Click **Tools** > **Firebase** to open the Assistant window.



```
eveloperblog.

y : AppCompat

onCreate(save
reate(savedInstanceState)
tView(R.layout.activity_main)
```

Once that is done, the Firebase assistant will appear presenting many Firebase services, we need to select the `Realtime Database` feature:



Firebase gives you the tools and infrastructure from Google to help you develop, grow and earn money from your app. Learn more

Google Sign-In, Facebook and other login providers. More info

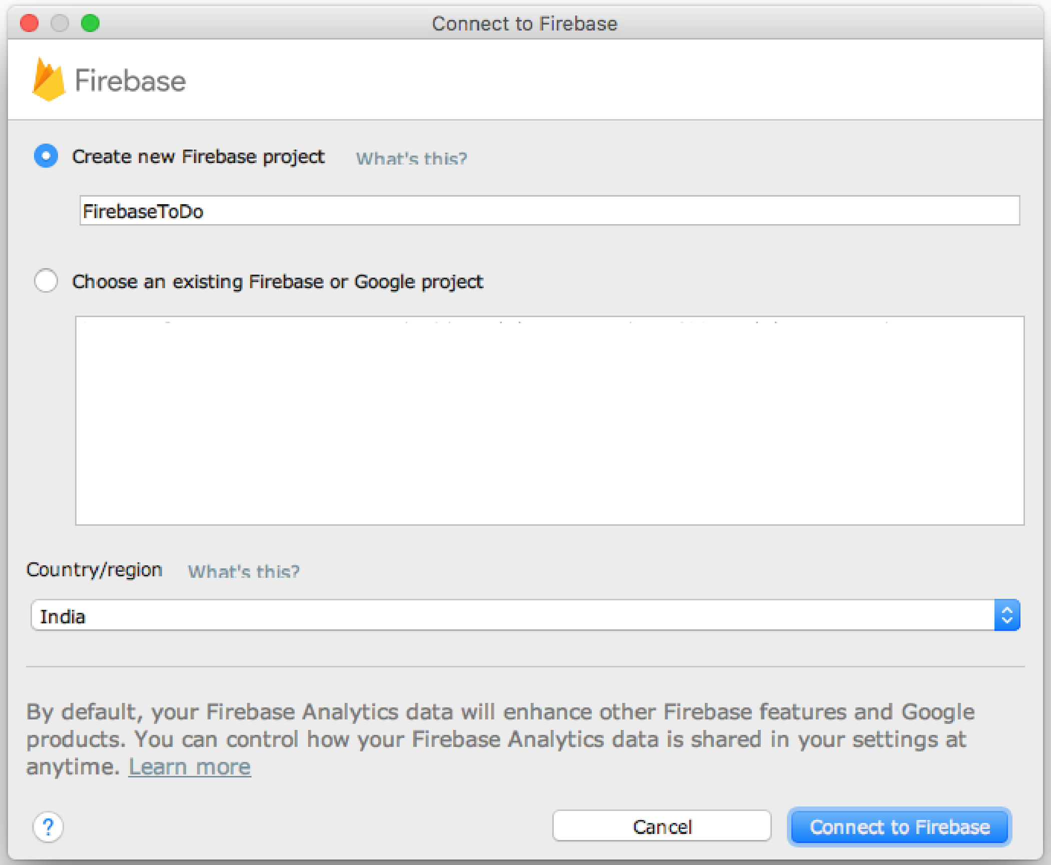
Realtime Database

Store and sync data in realtime across all connected clients. More info

Storage

Store and retrieve large files like images, audio, and video without writing server-side code. More info

Once you click 'Setup Firebase Realtime Database', following dialog will appear to confirm setting up the project:


Connect to Firebase

Firebase

Create new Firebase project   What's this?

FirebaseToDo

Choose an existing Firebase or Google project

Country/region   What's this?

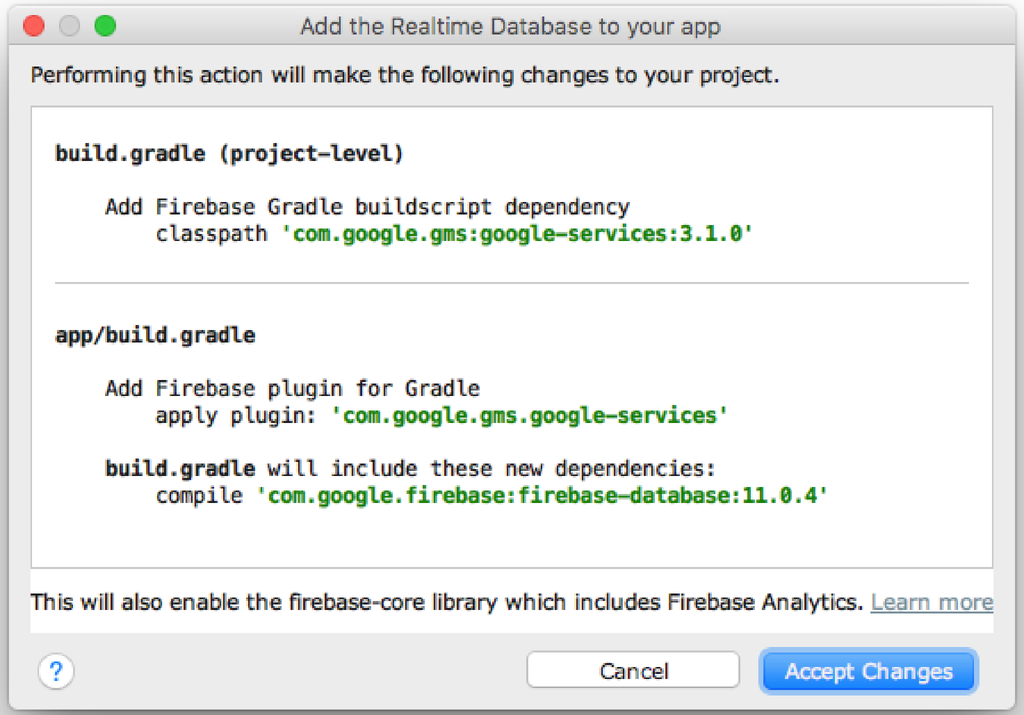
India

By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more

Next, Android Studio will offer to add relevant code snippets to our app like Gradle dependency:

```
implementation 'com.google.firebase:firebase-database:11.0.4'
...
//Added as last line
apply plugin: 'com.google.gms.google-services'
```

With other changes, it is clearly shown in the dialog Android Studio presented us:



Performing this action will make the following changes to your project.

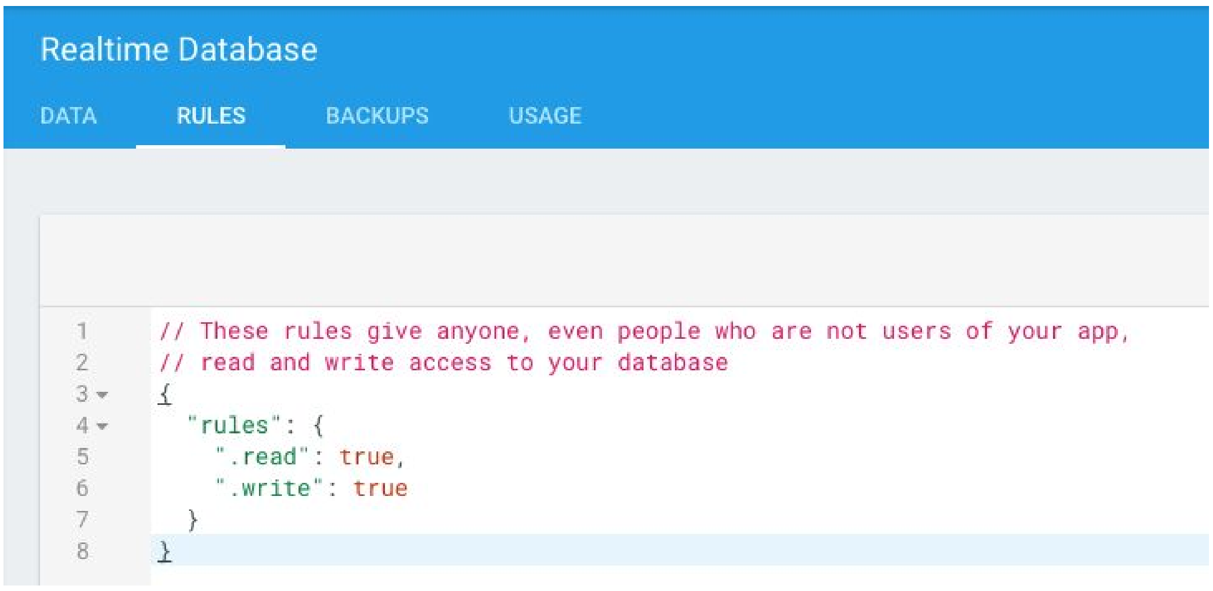
```
build.gradle (project-level)

    Add Firebase Gradle buildscript dependency
        classpath 'com.google.gms:google-services:3.1.0'

app/build.gradle

    Add Firebase plugin for Gradle
        apply plugin: 'com.google.gms.google-services'

    build.gradle will include these new dependencies:
        compile 'com.google.firebase:firebase-database:11.0.4'
```

This will also enable the firebase-core library which includes Firebase Analytics. Learn more

Now that the dependencies are set up correctly, we can start to work on our exciting app which will communicate to the firebase server. Make sure you have a project made in Firebase Console. You will be needing its ID later to be configured in the project too.

Also, please note that this lessons is not meant for authentication purposes. So, the to do items we add in our app will be accessible to all the users using the same app as we’re accessing the same database irrespective of the user identity. So, to make our database public, we will add the following script in our Firebase project access rules:

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```



```
// These rules give anyone, even people who are not users of your app,
// read and write access to your database
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

In the Data tab above, you will find a URL like:

```
https://fir-todo-something.firebaseio.com/
```

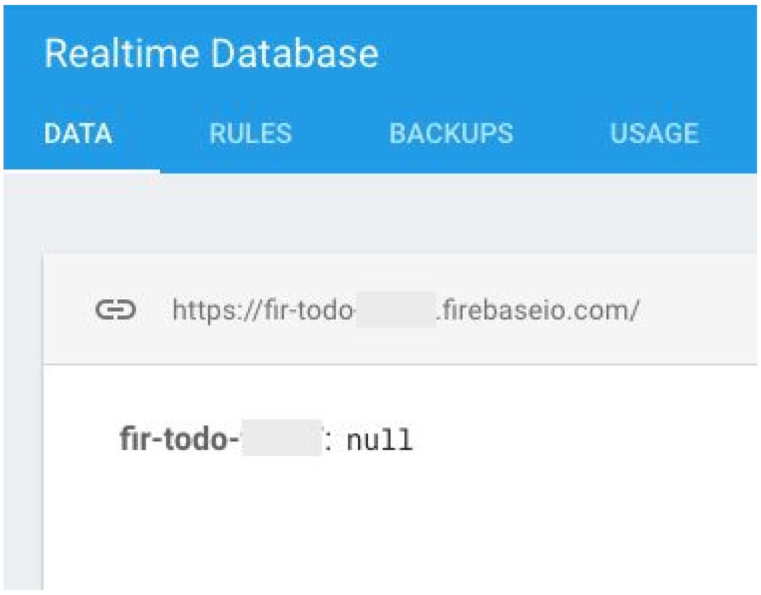
Though this URL is not important as our app will access the Firebase over the network and Firebase will identify which database to access via the app's package name itself and the *google-services.json* file added automatically by Firebase during its setup.

We will add our Database collection name in a new File in Kotlin whose content will be as follows:

```
package com.appsdeveloperblog.firebasetodo

object Constants {
   @JvmStatic val FIREBASE_ITEM: String = "todo_item"
}
```

Our collection name will be *todo_item.* We will see where will this appear as soon as we save first item in Firebase.

As of now, our Database on Firebase looks like:



fir-todo- : null

Adding To Do Item model

Here, we will design a model object which will be used to hold data for our To Do list item data. We will intentionally keep this very simple for demonstration purposes.

```
package com.appsdeveloperblog.firebasetodo

class ToDoItem {

    companion object Factory {
        fun create(): ToDoItem = ToDoItem()
    }

    var objectId: String? = null
    var itemText: String? = null
    var done: Boolean? = false
}
```

Let us understand what we’ve done in this simple model:

- There are three properties in the model.
    - *objectId* will hold the Firebase generated ID in the model This will be used to uniquely identify our item when we show it in a List

    - *itemText* will contain the To Do item text
    - *done* will contain a boolean value, if the listed to-do item is completed or not
- Before defining three properties, we have also defined companion factory declaration. This makes sure that our model can be instantiated.

Designing the app

Now that the complete configuration is done and we’ve finalised the data model, we can start working on the simple but intuitive design.

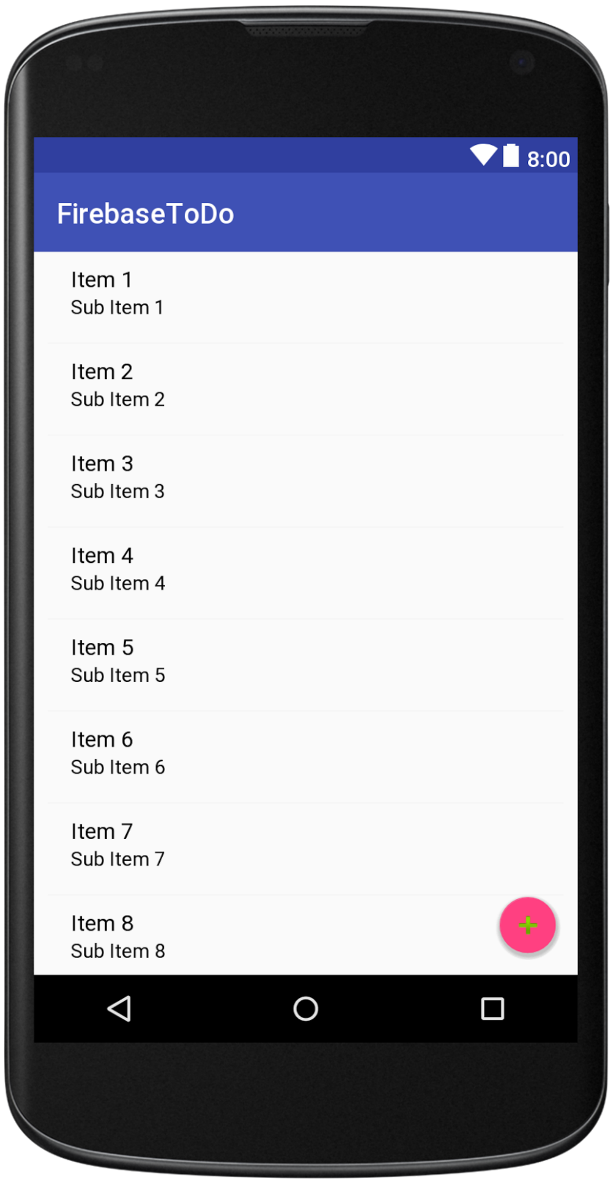
We need following functionalities in our app:

- Have a button to show a dialog where we can enter simple text and submit, so that a new item is added
- The newly added item should be shown in a listview with a checkbox on each row
- When this checkbox is clicked, it should mark this item as done
- When we load data from firebase, it will also tell us if the item is done, in which case, we will just show the checkbox as checked
- When we long press the list item, we will see a context menu with ‘Update’ and ‘Delete’ options
- On Update option selected, dialog will again appear with text already filled in and we can update the text and submit it
- On Delete option selected, the item will be deleted from the app and the Firebase database

To start, we need following elements:

- A ListView where To Do items can be shown
- A FloatingActionButton where a user can click and add new To Do item

Our initial UI will look like this:



To make this simple UI, we kept these properties in XML in *res > layout > activity_main.xml* file as:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"

tools:context="com.appsdeveloperblog.firebasetodo.MainActivity">
```

```
    <ListView
        android:id="@+id/items_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:scrollbars="none" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="15dp"
        android:src="@android:drawable/ic_input_add"
        app:elevation="6dp"
        app:pressedTranslationZ="12dp" />

</android.support.design.widget.CoordinatorLayout>
```

When this FloatingActionButton is clicked, we need to show a dialog with an EditText where user can enter new item and save it with Database.

Showing a Dialog on FloatingActionButton click

To do this, we must do following steps:

- Add a click listener on FloatingActionButton
- Show dialog when FloatingActionButton is clicked

Let’s make changes to our Activity now. We first define a FloatingActionButton in *onCreate(...)* method which points to the FloatingActionButton in our XML above.

```
val fab = findViewById<View>(R.id.fab) as FloatingActionButton
```

Then we set a click listener on it.

```
//Adding click listener for FAB
```

```
fab.setOnClickListener { view ->
   //Show Dialog here to add new Item
   addNewItemDialog()
}
```

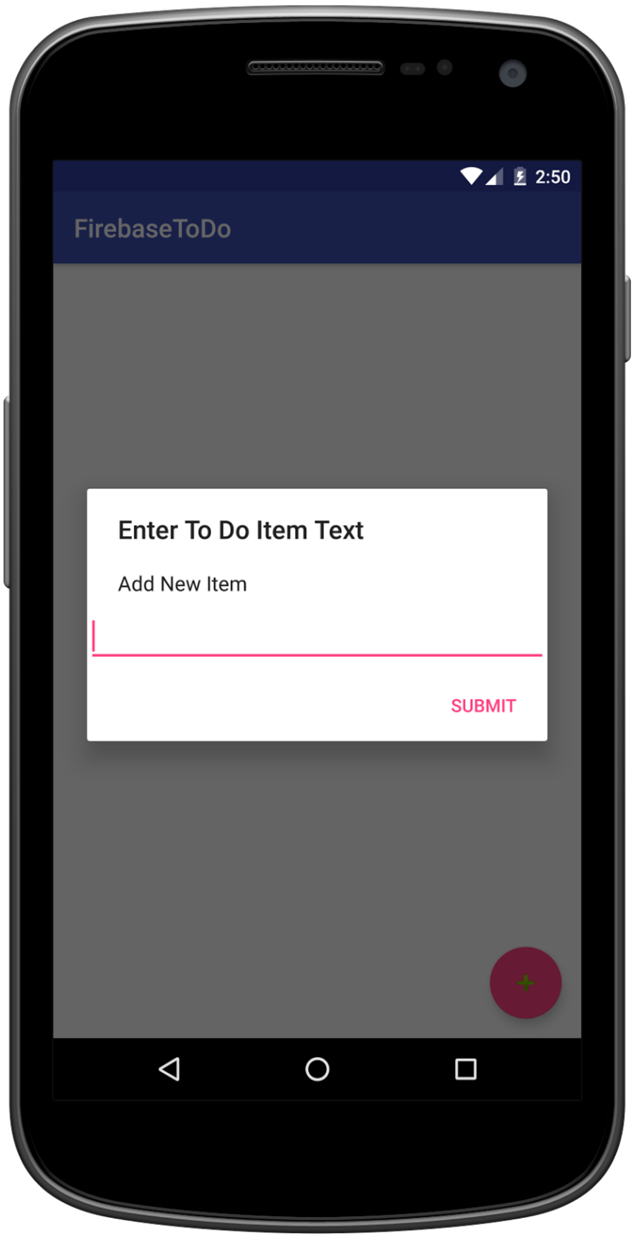
In *addNewItemDialog* function, we will show a dialog. As of now, our complete *onCreate(...)* method looks like:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)

   //reference for FAB
   val fab = findViewById<View>(R.id.fab) as FloatingActionButton

   //Adding click listener for FAB
   fab.setOnClickListener { view ->
       //Show Dialog here to add new Item
       addNewItemDialog()
   }
}
```

Let’s add the method to show the dialog next:

```
/**
* This method will show a dialog bix where user can enter new item
* to be added
*/
private fun addNewItemDialog() {
   val alert = AlertDialog.Builder(this)

   val itemEditText = EditText(this)
   alert.setMessage("Add New Item")
   alert.setTitle("Enter To Do Item Text")

   alert.setView(itemEditText)

   alert.setPositiveButton("Submit") { dialog, positiveButton ->   }

   alert.show()
}
```

Once we add the above code, alert dialog will appear once we hit the FloatingActionButton:



Creating entry in Firebase

This is where our real journey starts. In this section, we will start inserting data into the Firebase. We have already shown the current state of the database. Now, let’s modify the code of positive button listener of the dialog as:

```
alert.setPositiveButton("Submit") { dialog, positiveButton ->

   val todoItem = ToDoItem.create()
   todoItem.itemText = itemEditText.text.toString()
   todoItem.done = false

   //We first make a push so that a new item is made with a unique
ID
   val newItem = mDatabase.child(Constants.FIREBASE_ITEM).push()
   todoItem.objectId = newItem.key

   //then, we used the reference to set the value on that ID
   newItem.setValue(todoItem)

   dialog.dismiss()
   Toast.makeText(this, "Item saved with ID " + todoItem.objectId,
Toast.LENGTH_SHORT).show()
}
```

Let us see what we did above:

- Created a new *todoItem* instance and initialised it with default values.
- Using *push()* method, we obtain a new ID from firebase which we can provide to our To Do item.
- Once we do *setValue(...)*, the todoItem will be saved in firebase database.

We’ve used *mDatabase* in above code. What is it?

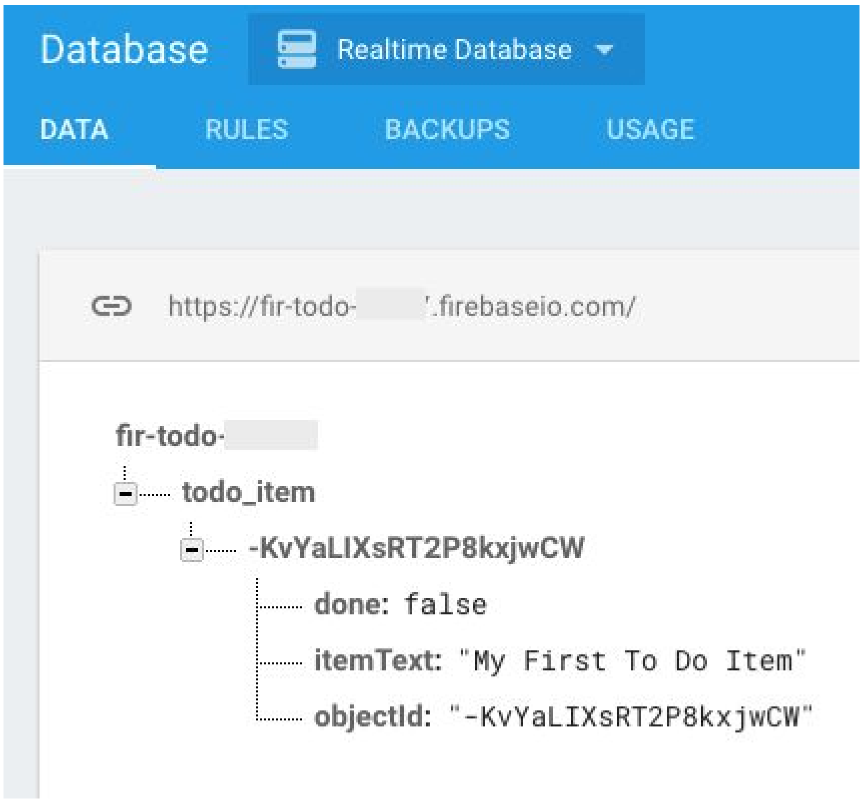
mDatabase is the Firebase database reference. Let’s create it as a Global variable in our Activity.

```
//Get Access to Firebase database, no need of any URL, Firebase
//identifies the connection via the package name of the app
lateinit var mDatabase: DatabaseReference
```

And initialise it in our *onCreate(...)* method:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    ...

    mDatabase = FirebaseDatabase.getInstance().reference
...
}
```

As simple as that. Now, run the app once again, enter some value and hit 'Submit'. The item will appear in the Firebase Database as:



```
fir-todo-[redacted]
  todo_item
    -KvYaLIXsRT2P8kxjwCW
      done: false
      itemText: "My First To Do Item"
      objectId: "-KvYaLIXsRT2P8kxjwCW"
```

Note here that our database name is the top element. Next element is our collection named as *todo_item*. Followed by it are the list of To Do items.

Reading Firebase data in a List

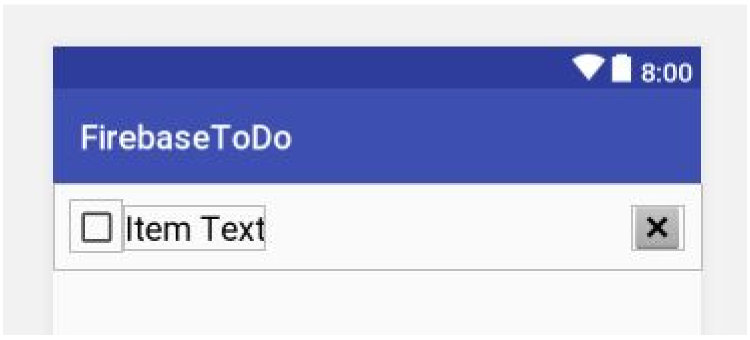
What is the use of the data until and unless we’re able to read the data. In the previous section, we designed a simple UI with a ListView. Now, we will enhance it with custom UI for its row.

Make a new file in res > layout folder and name it as *row_items.xml*. Fill it with following design:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal"
   android:padding="10dp">

   <CheckBox
       android:id="@+id/cb_item_is_done"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content" />

   <TextView
       android:id="@+id/tv_item_text"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_centerVertical="true"
       android:layout_toEndOf="@+id/cb_item_is_done"
       android:layout_toRightOf="@+id/cb_item_is_done"
       android:text="Item Text"
       android:textColor="#000000"
       android:textSize="20sp" />

   <ImageButton
       android:id="@+id/iv_cross"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
```

```
        android:layout_alignBottom="@+id/tv_item_text"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:background="@android:color/transparent"
        android:gravity="end"
        app:srcCompat="@android:drawable/btn_dialog" />

</RelativeLayout>
```

This is a simple UI which will create following row design:



Clearly, this row is enough to update item state as done using the CheckBox and to delete the row via the ImageButton.

To populate the ListView, we must create a new Adapter which will complete this responsibility.

```
package com.appsdeveloperblog.firebasetodo

import android.content.Context
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.BaseAdapter
import android.widget.CheckBox
import android.widget.ImageButton
import android.widget.TextView

class ToDoItemAdapter(context: Context, toDoItemList:
```

```
MutableList<ToDoItem>) : BaseAdapter() {

   private val mInflater: LayoutInflater =
LayoutInflater.from(context)
   private var itemList = toDoItemList

   override fun getView(position: Int, convertView: View?, parent:
ViewGroup?): View {

       val objectId: String = itemList.get(position).objectId as
String
       val itemText: String = itemList.get(position).itemText as
String
       val done: Boolean = itemList.get(position).done as Boolean

       val view: View
       val vh: ListRowHolder
       if (convertView == null) {
           view = mInflater.inflate(R.layout.row_items, parent,
false)
           vh = ListRowHolder(view)
           view.tag = vh
       } else {
           view = convertView
           vh = view.tag as ListRowHolder
       }

       vh.label.text = itemText
       vh.isDone.isChecked = done

       return view
   }

   override fun getItem(index: Int): Any {
       return itemList.get(index)
   }

   override fun getItemId(index: Int): Long {
       return index.toLong()
   }
   override fun getCount(): Int {
       return itemList.size
   }

   private class ListRowHolder(row: View?) {
       val label: TextView =
row!!.findViewById<TextView>(R.id.tv_item_text) as TextView
       val isDone: CheckBox =
row!!.findViewById<CheckBox>(R.id.cb_item_is_done) as CheckBox
       val ibDeleteObject: ImageButton =
row!!.findViewById<ImageButton>(R.id.iv_cross) as ImageButton
   }
}
```

In above adapter, we:

- Defined a *ListRowHolder* to contain items for our UI
- Populated list items with appropriate data

Now, we must use this Adapter in our Activity. Also, we also have to fetch data from Firebase.

Fetching data from Firebase

Getting data from Firebase is actually easy. Just do this in *onCreate(...)* method:

```
mDatabase.orderByKey().addListenerForSingleValueEvent(itemListener
)
```

We will add item listener next. We just add a listener for current database which will fetch and alert the function when the data has been fetched.

We first define some global variables, as:

```
var toDoItemList: MutableList<ToDoItem>? = null
lateinit var adapter: ToDoItemAdapter
private var listViewItems: ListView? = null
```

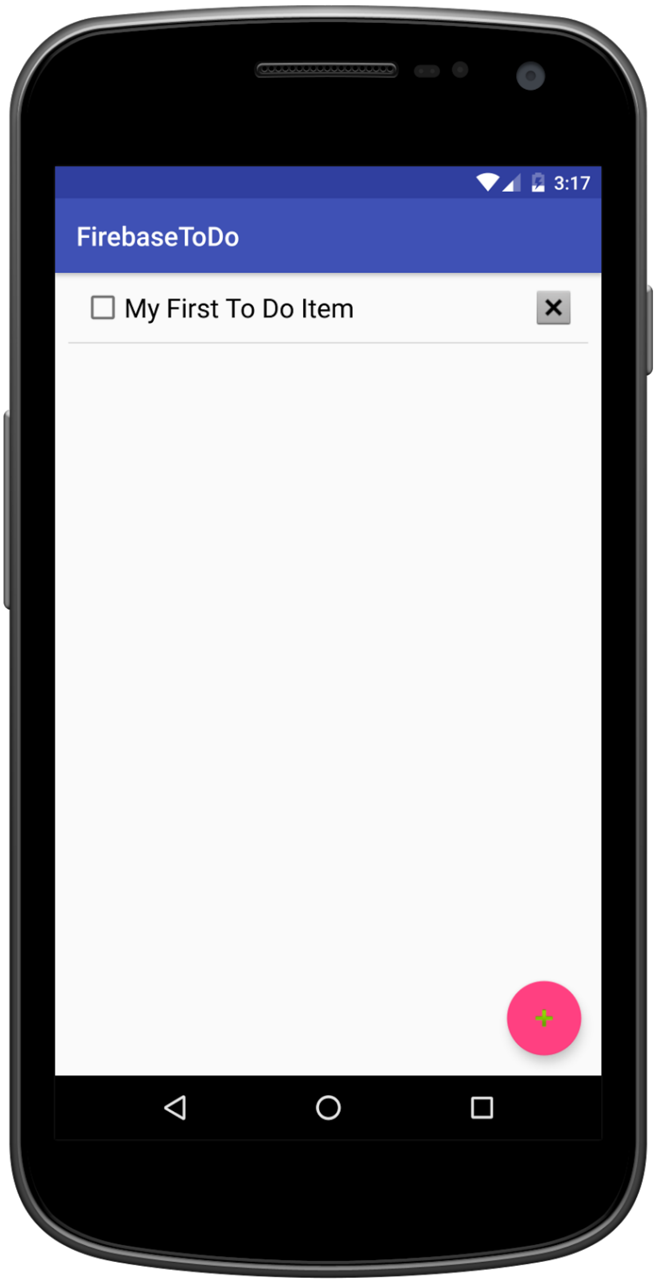
We will initialise each of them in our *onCreate(...)* method. Once all this is done, our method will look like:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)

   //reference for FAB
   val fab = findViewById<View>(R.id.fab) as FloatingActionButton
   listViewItems = findViewById<View>(R.id.items_list) as ListView

   //Adding click listener for FAB
   fab.setOnClickListener { view ->
       //Show Dialog here to add new Item
       addNewItemDialog()
   }

   mDatabase = FirebaseDatabase.getInstance().reference
   toDoItemList = mutableListOf<ToDoItem>()
   adapter = ToDoItemAdapter(this, toDoItemList!!)
   listViewItems!!.setAdapter(adapter)

mDatabase.orderByKey().addListenerForSingleValueEvent(itemListener
)
}
```

Finally, our listener looks like:

```
var itemListener: ValueEventListener = object : ValueEventListener
{
        override fun onDataChange(dataSnapshot: DataSnapshot) {
            // Get Post object and use the values to update the UI
            addDataToList(dataSnapshot)
        }

        override fun onCancelled(databaseError: DatabaseError) {
            // Getting Item failed, log a message
            Log.w("MainActivity", "loadItem:onCancelled",
databaseError.toException())
        }
    }
    private fun addDataToList(dataSnapshot: DataSnapshot) {

        val items = dataSnapshot.children.iterator()
        //Check if current database contains any collection
        if (items.hasNext()) {
            val toDoListindex = items.next()
            val itemsIterator = toDoListindex.children.iterator()

            //check if the collection has any to do items or not
            while (itemsIterator.hasNext()) {

                //get current item
                val currentItem = itemsIterator.next()
                val todoItem = ToDoItem.create()

                //get current data in a map
                val map = currentItem.getValue() as
HashMap<String, Any>

                //key will return Firebase ID
                todoItem.objectId = currentItem.key
                todoItem.done = map.get("done") as Boolean?
                todoItem.itemText = map.get("itemText") as String?

                toDoItemList!!.add(todoItem);
            }
        }

        //alert adapter that has changed
        adapter.notifyDataSetChanged()
    }
```

What each line does is explained in above code itself. Now, we're ready to show the data in our ListView. Once we run our app now, we can see the data as:



That’s great !! Our data is clearly visible now. As of now, we only can create and read our data.

We are left with two operations. Updating the item as done or undone by checking that CheckBox. Second by clicking on that cross image to delete this data.

Update and Delete the items

Now, to perform these operations. We must know when each of our row is clicked. Also, we need to get the ObjectID of the row that was clicked. This can be obtained from the Adapter.

We will implement **Observer pattern** in this scenario which is an ideal implementation about how to **interact between an adapter and an activity**.

Let's start by creating an Kotlin Interface file as:

```
package com.appsdeveloperblog.firebasetodo

interface ItemRowListener {

   fun modifyItemState(itemObjectId: String, isDone: Boolean)
   fun onItemDelete(itemObjectId: String)
}
```

Clearly, we want to provide two operations.

Next, we will make our activity implement this interface by doing:

```
class MainActivity : AppCompatActivity(), ItemRowListener {
```

As we've implemented an interface, we must implement its methods as well:

```
override fun modifyItemState(itemObjectId: String, isDone:
Boolean) {
   val itemReference =
mDatabase.child(Constants.FIREBASE_ITEM).child(itemObjectId)
   itemReference.child("done").setValue(isDone);
}
```

```
//delete an item
override fun onItemDelete(itemObjectId: String) {

   //get child reference in database via the ObjectID
   val itemReference =
mDatabase.child(Constants.FIREBASE_ITEM).child(itemObjectId)
   //deletion can be done via removeValue() method
   itemReference.removeValue()
}
```

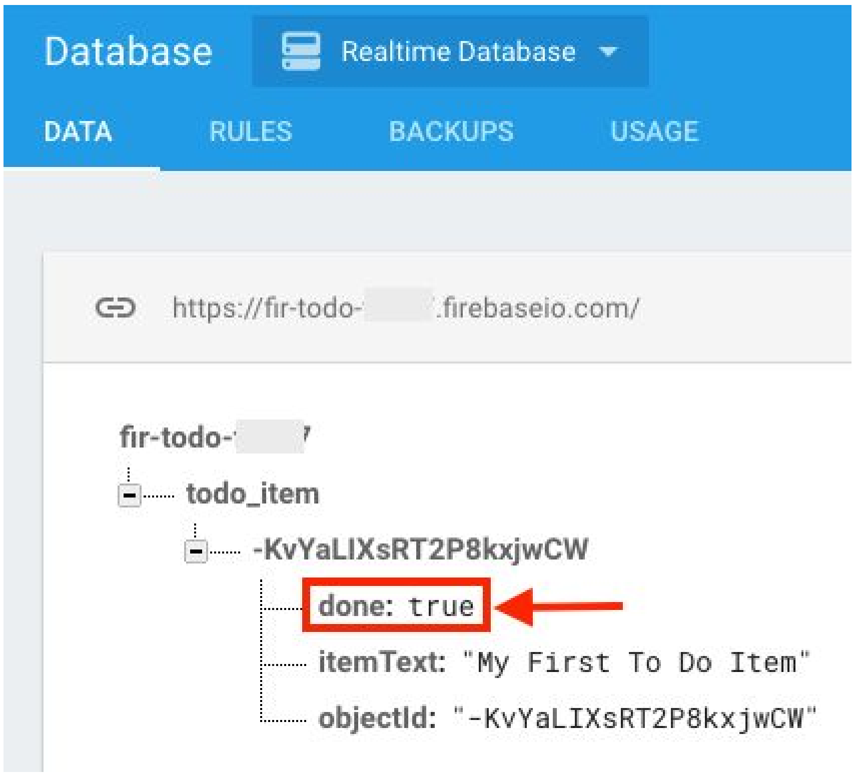
We need to call these methods via our adapter. In our *getView(...)* method, we can do as:

```
vh.isDone.setOnClickListener {
    rowListener.modifyItemState(objectId, !done) }

vh.ibDeleteObject.setOnClickListener {
    rowListener.onItemDelete(objectId) }
```

This can only be done if we define *rowListener* in our Adapter class:

```
private var rowListener: ItemRowListener = context as
ItemRowListener
```

Now, once we click on checkbox as done, our database will change as:



```
fir-todo-[redacted]
  todo_item
    -KvYaLIXsRT2P8kxjwCW
      done: true
      itemText: "My First To Do Item"
      objectId: "-KvYaLIXsRT2P8kxjwCW"
```

Value has been changed.

Extras : Persisting Firebase data offline

Firebase is excellent in caching data. Once you run the app and get your data, Firebase will cache to so that it can be used offline as well. Once your app is back online, it will update the data.

The caching behaviour is **off by default.** Let’s modify this behavior.

We will create an Application class now:

```
package com.appsdeveloperblog.firebasetodo

import android.app.Application
import com.google.firebase.database.FirebaseDatabase

class ThisApplication: Application() {
   override fun onCreate() {
       super.onCreate()

       //Enable Firebase persistence for offline access
       FirebaseDatabase.getInstance().setPersistenceEnabled(true)
   }
}
```

That’s it! Just a single line of code and caching is enabled. Of course, you need to add this Application class in your Manifest file:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:name=".ThisApplication"
    android:theme="@style/AppTheme">

   ....
</application>
```

Our full fledged To Do Item application is ready.

Conclusion

In this lesson, we looked at strong CRUD properties of Firebase and how fast is it to sync with web server. We completed CRUD operations in our elegant app and enjoyed it as well.

Firebase also has strong caching techniques through which it promises strong user experience by presenting them data even when there is no network. We can a lot more with Firebase like saving images etc. Let’s save it for another time.