

About OpenTDB .....	2
Fetch data dynamically .....	2
Not every successful query return questions.....	2
Getting category list from opentdb.com/api_category.php .....	3
Avoid searching too frequently .....	3
About Firebase.....	4
Firebase operates asynchronously(IMPORTANT!) .....	4
Abundant error handling.....	5
Pass the database URL to the getInstance() method .....	5
Use HashMap instead of ArrayList.....	5
Use addListenerForSingleValueEvent instead of addValueEventListener.....	6
Create empty constructor for every class representing data .....	6
Firebase doesn't support multi-field queries .....	7
About Assignment.....	8
Multiple choices or Boolean? .....	8
Allocate at least 7days of work.....	8

# About OpenTDB

## Fetch data dynamically

The default API URL (<https://opentdb.com/api.php?amount=10>) only returns questions with random types, categories, and difficulties. However, what we need is to filter questions based on specific criteria, we can do so by adjusting the URL parameters.

For instance, such URL (<https://opentdb.com/api.php?amount=10&category=20&difficulty=medium&type=multiple>) will return 10 questions of medium difficulty, multiple-choice type, and from the Mythology category.

Retrofit's **Query annotation** allows us generate query parameters dynamically like this.

```
2 usages
public interface QuizService {
    1 usage
    @GET("api.php")
    Call<Question> getQuestions(@Query("amount") int amount, @Query("category") int category,
                               @Query("difficulty") String difficulty, @Query("type") String type);
}
```

Go back to MainActivity, we can query questions with any criteria by adjusting the parameters.

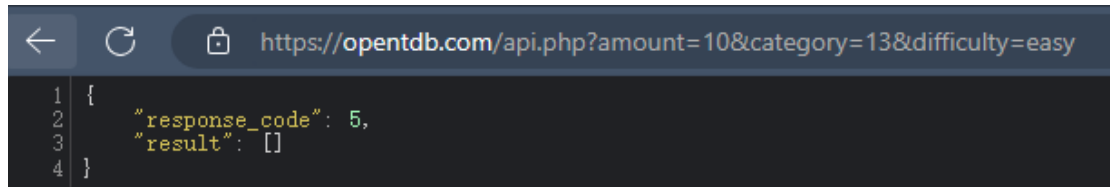
```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://opentdb.com/")
    .addConverterFactory(GsonConverterFactory.create())//配置URL的基地址
    .build();
QuizService quizService = retrofit.create(QuizService.class);
Call<Question> questions = quizService.getQuestions( amount: 15, category: 9, difficulty: "easy", type: "multiple");
```

If we prefer random categories, types, or difficulties, we can achieve this by setting **integer** parameters as **0** and **string** parameters as **null**. For example, to get 15 random questions without any restrictions, we can code like this.

```
Call<Question> questions = quizService.getQuestions( amount: 15, category: 0, difficulty: null, type: null);
```

## Not every successful query return questions

URL address like such returns no questions. The query is indeed successful, but no questions are included. This is not our issue, as the website doesn't have a question bank corresponding to the current type of parameter. **Check** whether the returned data is **empty or null** in advance to prevent unexpected crash down.



```
1 {  
2   "response_code": 5,  
3   "result": []  
4 }
```

## Getting category list from [opentdb.com/api\\_category.php](https://opentdb.com/api_category.php)

[opentdb.com/api\\_category.php](https://opentdb.com/api_category.php) will return the **name** of each category and the corresponding **id**, which will be used as a parameter in querying. The format is also JSON. You can create a class to store such data, or simply hardcode.



```
"trivia_categories": [  
  {  
    "id": 9,  
    "name": "General Knowledge"  
  },  
  {  
    "id": 10,  
    "name": "Entertainment: Books"  
  },  
  {  
    "id": 11,  
    "name": "Entertainment: Film"  
  },  
  {  
    "id": 12,  
    "name": "Entertainment: Music"  
  },  
  {  
    "id": 13,  
    "name": "Entertainment: Musicals & Theatricals"  
  },  
  {  
    "id": 14,  
    "name": "Entertainment: Television"  
  },  
  {  
    "id": 15,  
    "name": "Entertainment: Video Games"  
  }  
]
```

## Avoid searching too frequently

Each IP can only access the API **once every 5 seconds**. Consider this in the code in advance or simply avoid searching so frequently when displaying the assignment.

# About Firebase

## Firebase operates asynchronously(IMPORTANT!)

All our previous tasks (at least mine) have been executed on the **main thread**, but services like Firebase operate **asynchronously** in the **background**. The advantage is that it doesn't block the main thread due to network delays, but the challenge lies in considering the sequence of code execution.

Look at a simple example. We have 2 methods: **step\_1\_getData()** and **step\_2()**. The first one will render a message of “**Step\_1**” when successfully connected to the firebase database, and the second one renders “**Step\_2**”. We want “**Step\_1**” displayed before “**Step\_2**”, but in reality, they always follow the opposite order.

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_page_sign_up);

    Button btn = findViewById(R.id.btn);
    btn.setOnClickListener(v -> {

        step_1_getData();
        step_2();

    });
}

1 usage
private void step_1_getData() {
    myref.addValueEventListener(new ValueEventListener() {

        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {

            Toast.makeText(context: PageSignUp.this, text: "Step_1", Toast.LENGTH_SHORT).show();

        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }

    });
}

1 usage
private void step_2() {

    Toast.makeText(context: this, text: "Step_2", Toast.LENGTH_SHORT).show();

}
```

This is because firebase executes in the background asynchronously. After executing the **step\_1\_getData()** method, the system doesn't wait for firebase to return data but directly proceeds to execute the **step\_2()** method.

Consider the scenario where we need to check if an email address exists in the database during user registration. If it does, we need to display an error message; if not, we proceed with successful registration. Following the previously mentioned method, even if the email address already exists,

the user would still be successfully registered, and the error message would only be received afterwards. To address this issue, you might need a **callback function** to handle it. Solve this problem, then there won't be any major challenges left for this assignment.

## Abundant error handling

```
java.lang.NullPointerException: Attempt to invoke virtual method 'java.lang.String com.example.assignment_2.User.getName()' on a null object reference
```

If you encounter errors like a **null object reference**, or **index bound 0 of 0**, it's likely due to asynchronous execution. Using **Toast** or **Log** statements more frequently can help us quickly identify the location of the error. Additionally, it's important to anticipate where data might be **null** or **empty** and implement error handling in advance. Due to lack of experience, this part has taken up at least 20% of my workload. Now I've placed the code like this everywhere.

```
@Override
public void onResponse(Call<RawOpenTDBDataReturned> call, Response<RawOpenTDBDataReturned> response) {

    if (response.body() == null) {

        Toast.makeText(context: PageCreateQuiz.this, text: "Unable to retrieve data", Toast.LENGTH_SHORT).show();
        return;
    }
    List<RawOpenTDBDataReturned.ResultsBean> DataReturned = response.body().getResults();

    if (DataReturned.isEmpty()) {

        Toast.makeText(context: PageCreateQuiz.this, text: "No Data Available, choose Another Option", Toast.LENGTH_SHORT).show();
    } else {
```

## Pass the database URL to the getInstance() method

If your database node is **not** in the **US** region, you need to pass the current database URL as a parameter to the **getInstance()** method as shown in the image below.

```
FirebaseDatabase.getInstance(url: "https://assignment2-fd51e-default-rtdb.asia-southeast1.firebaseio.com/")
```

## Use HashMap instead of ArrayList

Firebase is **more friendly** towards data structures like **HashMap** than **ArrayList**. A typical firebase data structure is Hashmap where keys are auto generated, and values are inputted by us. With keys and values, we can query and modify corresponding data. If you have nested data types, such as **incorrect answers**, or **every quiz a user has participated in**, it's better to store and upload them to the

database in **HashMap** format rather than **ArrayList**. After completing the assignment, I tried to change all HashMap data to ArrayList storage, and I regretted it. Although it looks a little bit more elegant (because you don't have to meddle with those strange keys), I encountered some problems when retrieving data, which was not as convenient as directly using HashMap. So please get used to and enjoy these strange keys. By the way, the key exists as a String, like "-Nx5GcN2W1WrG6RZnGn3". Don't forget the "-" at the beginning; it's also part of it.

## Use **addListenerForSingleValueEvent** instead of **addValueEventListener**

Firebase highly recommends using the **addValueEventListener** method to dynamically monitor data. It's convenient as it calls whenever any value of a node we set changes, which might be why firebase is adopted by many real-time communication applications. However, in this assignment, we don't really need this feature. On the contrary, since **addValueEventListener** method also executes asynchronously, it might lead to unexpected errors while we are still unfamiliar with it. Therefore, I prefer **addListenerForSingleValueEvent**, which executes only once when called, and that's enough.

Once, I used a **push()** method inside the **addValueEventListener** method, and the program crashed. This happened because each **push()** method causes a change in value, which will create a loop within the **addValueEventListener**. Eventually, I had to force stop the program and clear the database. You might not make such a rookie mistake, but still, be cautious. I only use the **addValueEventListener** where there are **no database modifications involved** (for example, in places where I display the current quiz to provide real-time updates on changes made in other activities).

## Create empty constructor for every class representing data

Classes representing data (such as quiz, user, etc.) need an empty constructor, even if it's not used in our own code at all.

```
//constructor
public Quiz() {
    this.name = "";
    this.difficulty = "";
    this.category = "";
    this.startDate = new Date();
    this.endDate = new Date();
    this.likes = 0;
    this.questions = new ArrayList<>();
}
```

This is because when we query data, if we want the database directly return **an instance** of that class as picture below, the database will default to calling this empty constructor. Otherwise, an error will

occur.

```
@Override
public void onDataChange(@NonNull DataSnapshot dataSnapshot) {

    HashMap<String, Quiz> quizList = new HashMap<>();

    for (DataSnapshot quiz : dataSnapshot.getChildren()) {
        Quiz quizData = quiz.getValue(Quiz.class);
        quizList.put(quiz.getKey(), quizData);
    }
}
```

## Firestore doesn't support multi-field queries

Firestore offers some great Query methods, like below one that can query all data where endDate (a field in the Quiz class) is later than the current time.

```
Query myQuery = myRef.orderByChild( path: "endDate").startAt(new Date().getTime());
myQuery.addListenerForSingleValueEvent(new ValueEventListener() {
```

Unfortunately, it doesn't support querying **multiple fields simultaneously**. I hope to return **ongoing quizzes** using this approach, but it will result in an error of “**you can't combine multiple orderBy calls**”.

```
Query myQuery = myRef.orderByChild( path: "endDate").startAt(new Date().getTime()).
    orderByChild( path: "startDate").endAt(new Date().getTime());
```

```
FATAL EXCEPTION: main
Process: com.example.assignment_2, PID: 16784
java.lang.IllegalArgumentException: You can't combine multiple orderBy calls!
```

Apart from downloading the entire quiz instance and then querying within the class like this, I haven't found a better solution. If you have one, please enlighten me.

```
HashMap<String, Quiz> quizListOngoing = new HashMap<>();
for (Map.Entry<String, Quiz> entry : quizList.entrySet()) {
    if (!quizKeyList.getKeyList().contains(entry.getKey()) && entry.getValue().getEndDate().after(new Date()) &&
        entry.getValue().getStartDate().before(new Date())) {
        quizListOngoing.put(entry.getKey(), entry.getValue());
    }
}
```

# About Assignment

## Multiple choices or Boolean?

There are two types of questions, and their data structures are the same, except that multiple choice questions have three incorrect answers, while Boolean questions have only one (but also in the form of an ArrayList). Tariq suggests that we only need to **choose one type** of question. Boolean seems simpler because we don't have to worry about adjusting the order of four answer options when displaying the question (You don't want all correct answers to be A, right?).

## Allocate at least 7days of work

If you, like me, have no prior experience manipulating firebase databases, it's best to reserve at least 7 days to complete the whole assignment\_2, with 30% of that time potentially dedicated to debugging and fine-tuning data communication with Firebase.

Good Luck!