Zespół: “U mnie działa”

Aplikacja mobilna „Tiara Przydziału”

Android Dokumentacja Techniczna

Jakub Kubkowski
26.05.2021

Spis treści

1. Wymagania sprzętowe i instalacja aplikacji

Aplikacja można bez problemu uruchomić na androidzie w wersji od 7 do 11. Wersje te obecnie stanowią większość rynku jeśli chodzi o urządzenia mobilne.

Jeśli chodzi o samą instalację aplikacja nie jest obecnie upubliczniona w Play Store (lub innych sklepach), należy skorzystać z dołączonej paczki APK która służy do instalacji aplikacji na telefonie.

Jak zainstalować aplikacją z APK ? (Android do 8 wersji systemu)

- Pobierz plik APK, który chcesz zainstalować.
- Przejdź do menu ustawień telefonu, a następnie do ustawień zabezpieczeń.
- Włącz opcję Zainstaluj z nieznanych źródeł.
- Użyj przeglądarki plików i przejdź do folderu pobierania. Stuknij APK, | aby rozpocząć proces instalacji.
- Aplikacja powinna zostać bezpiecznie zainstalowana.

Jak zainstalować aplikacją z APK ? (Na przykładzie androida 8 i wyżej)

- Pobierz plik APK, który chcesz zainstalować.
- Możesz przejść do folderu pobierania za pomocą aplikacji przeglądarki plików lub po prostu rozpocząć instalację,
- klikając ukończone pobieranie w przeglądarce mobilnej.
- Android poprosi Cię o zezwolenie przeglądarce plików lub przeglądarce internetowej na zainstalowanie aplikacji.
- Udziel pozwolenia, a powinno odesłać Cię z powrotem do ekranu instalacji. Jeśli nie, przejdź z powrotem do folderu pobierania po udzieleniu pozwolenia na ponowną próbę.
- Aplikacja powinna zostać bezpiecznie zainstalowana.

2. Co robi aplikacja

Aplikacja służy jako narzędzie do rozwiazywania quizu, oraz prezentowania jego wyników użytkownikom systemu android.

3. Techniczne aspekty aplikacji

Aplikacja została napisana przy użyciu Kotlina oraz Java. Użyłem tutaj dwóch języków z uwagi na prostotę w napisaniu (nadpisaniu) niektórych metod, niezbędnych przy wyświetlaniu wyników otrzymanych od serwera.

Sama komunikacja z serwerem odbywa się poprzez odpowiednio wystawione endpointy. Serwera wysyła do nas odpowiednio sformatowanego JSONArray, JSONObject lub String. Zadaniem aplikacji jest odpowiednia reprezentacja tych danych oraz przygotowanie odpowiedniej odpowiedzi dla serwera.

Pomocne w tym zadaniu (reprezentacji danych) są Data Class które oferuje Kotlin. Więcej o tym można przeczytać w 3.1 Użyte Data Class

Jeśli chodzi o samą strukturę aplikacji, jest ona niezwykle prosta i powtarzalna. W rozdziale 4 postaram się opowiedzieć o najważniejszych technologiach i przykładowych miejscach ich użycia razem z przykładami z kodu.

4. Najważniejsze technologie

W tym rozdziale opiszę najważniejsze technologie z jakich korzystałem przy pisaniu aplikacji. Jako, że aplikacja zawiera powtarzalną strukturę, skupię się na wykorzystaniu technologii w najważniejszych fragmentach kodu.
Po starannym zapoznaniu się z ich opisem nie powinno być problemu z odniesieniem tego do innych miejsc w których wykorzystana jest podobna struktura.

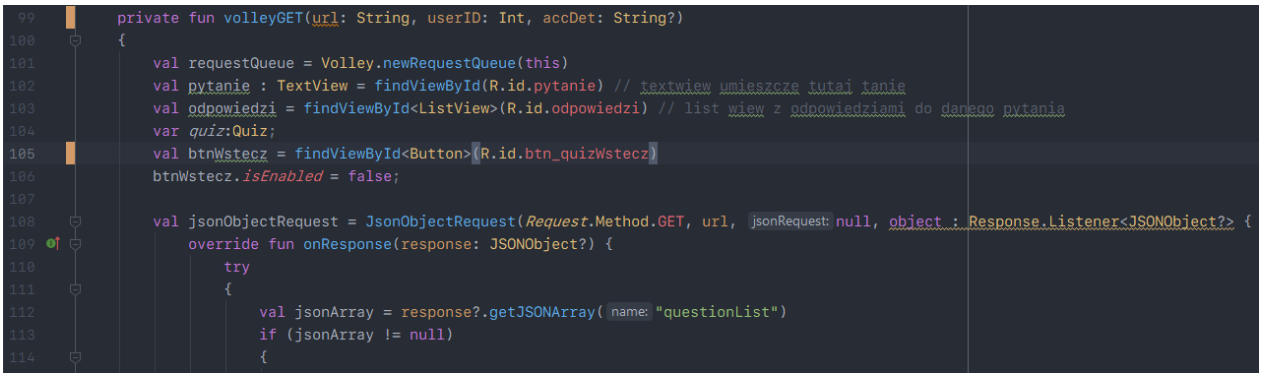
4.1 Komunikacja z backendem

Do komunikacji z serwerem backendowym używam Volley (link do dokumentacji https://developer.android.com/training/volley).

```
implementation 'com.android.volley:volley:1.2.0'
```

Podstawową wykorzystywaną przeze mnie funkcjonalnością jest możliwość wysyłania i dobierania żądanego formatu danych z konkretnego adresu URL.
Przedstawię teraz dwie metody GET i POST których używam w całej aplikacji.

4.1.1 Metoda GET



Pierwszą metodą jaką chciałbym opisać jest metoda GET.
Zacznijmy od składni jaką oferuje nam to konkretne zapytanie:
W linijce 108 znajduje się zmienna w której przechowywane jest całe zapytanie przy użyciu Volley'a
Jak widać żądanym przez nas plikiem jest JSONObject, informuje nas o tym nazwa funkcji (JsobObjectRequest).
Na początku podajemy metodę, później URL na który chcemy daną metodę wysłać, w tym przypadku URL jest zdefiniowany powyżej funkcji i wygląda tak:

```
val url = "https://programowaniezespolowe-app.herokuapp.com/quiz/start"
```

Następnie po podaniu URL jest jsonRequest w tym przypadku jest null-em, ale w przypadku POST jest bardzo istotny, później na końcu podajemy obiekt jakiego się spodziewamy.
W tym konkretnym przypadku żądany obiekt i obiekt wysyłany przez serwer są zgodne (JSONObject -> JSONObject, JSONArray -> JSONArray itd.), jednak pokażę przykład w którym należało stworzyć odpowiednią metodę która wysyła i przejmuje inne obiekty.
Domyślnie Volley nie posiada takiej funkcjonalności.
Od linijki 109 zaczyna się odpowiednie przetwarzanie otrzymanej przez naszą aplikację odpowiedzi.

4.1.2 Metoda POST

```
val jsonObjectRequest = JsonObjectRequest(Request.Method.POST, url, jsonreq, object : Response.Listener<JSONObject?> {
    override fun onResponse(response: JSONObject?) {
        try {
            val jsonArray = response?.getJSONArray( name: "questionList")
            if (jsonArray != null)
            {
                old_quiz = Gson().fromJson(jsonreq.toString(), Quiz::class.java)
                quiz = Gson().fromJson(response.toString(), Quiz::class.java)

                if (quiz.questionList.isEmpty())
                {
                    koniecQuizu(quiz,userID,accDet);
                }
                else
                {
                    val pierwszePytanieQuiz: Question = quiz.questionList[0];
                    var lastClickedPosition: Int = -1;
                    pytanie.setText(pierwszePytanieQuiz.text);

                    odpowiedzi.adapter = MyCustomAdapter( context: this@Start, quiz.questionList[0].answers);
```

Przejdźmy teraz do użycia metody POST.
Składnia pozostaje taka sama jak w GET z tą różnicą, że tutaj podajemy także jsonRequest.
W tym przypadku zmienna jsonreq (JSONObject) jest obiektem Quiz (Data Class), metoda zaprezentowana tutaj wysyła Quiz wraz z zaznaczonymi przez użytkownika odpowiedziami.

4.1.3 Zmodyfikowane metody

Jak wspominałem wcześniej musiałem zmodyfikować (dodać) pewne funkcjonalności do istniejących metod w Volley. Opierając się na dokumentacji poszczególnych metod stworzyłem własne, kody metod wraz z krótkim opisem znajdującym się poniżej.
Jest to jedna z dwóch zmodyfikowanych przeze mnie metod.
W tym przypadku wysyłamy do serwera JSONObject, serwer odpowiada nam odpowiednio zmodyfikowanym Stringiem który później przetwarzamy.

```
    val jsonObjectRequest = JsonArrayRequestMy(Request.Method.POST, url, jsonreq, object : Response.Listener<JSONArray?> {
        override fun onResponse(response: JSONArray?)
        {
            try
            {
                listOfJobs = Gson().fromJson(response.toString(), ProfessionForSearch::class.java)
                listViewJobs.adapter = MyCustomAdapter( context: this@ShowJobsFromCodes, listOfJobs);
            }
            catch (e: JSONException)
            {
                e.printStackTrace()
            }
        }
    },
            object : Response.ErrorListener {
                override fun onErrorResponse(error: VolleyError) {
                    error.printStackTrace()
                    Log.d( tag: "JsonObjectErr", error.toString());
                }
            }
    )
    requestQueue.add(jsonObjectRequest)
}
```

Kolejną metodą którą dodałem jest metoda wysyłająca JSONObject a przyjmująca i przetwarzająca JSONArray.

5. Użyte Data Class

Bardzo ważne w tym projekcie była właściwa prezentacja i przetwarzanie otrzymanych danych. W tym rozdziale skupię się tylko na tym jak dane zostały przechowane (ich reprezentacja). Użyłem dostępnych w Kotlinie Data Class (link do dokumentacji (https://kotlinlang.org/docs/data-classes.html) z uwagi na ich prostotę w użyciu i fakt niepisania nadmiarowych metod, które w Javie nie były domyślne.

```
package com.example.tiara_przydziau

data class Question(
    val answers: List<Answer>,
    val groupCode: String,
    val hasSimilarQuestions: Boolean,
    val id: Int,
    val multipleChoice: Boolean,
    val similarQuestionIds: List<Any>,
    val text: String
)
```

Reprezentacja pytania w Quizie jako data class.

6. Technologie użyte do przetwarzania otrzymanych danych

Nieoceniona w przetwarzaniu obiektów typu JSON na Data Class i odwrotnie okazała się biblioteka:

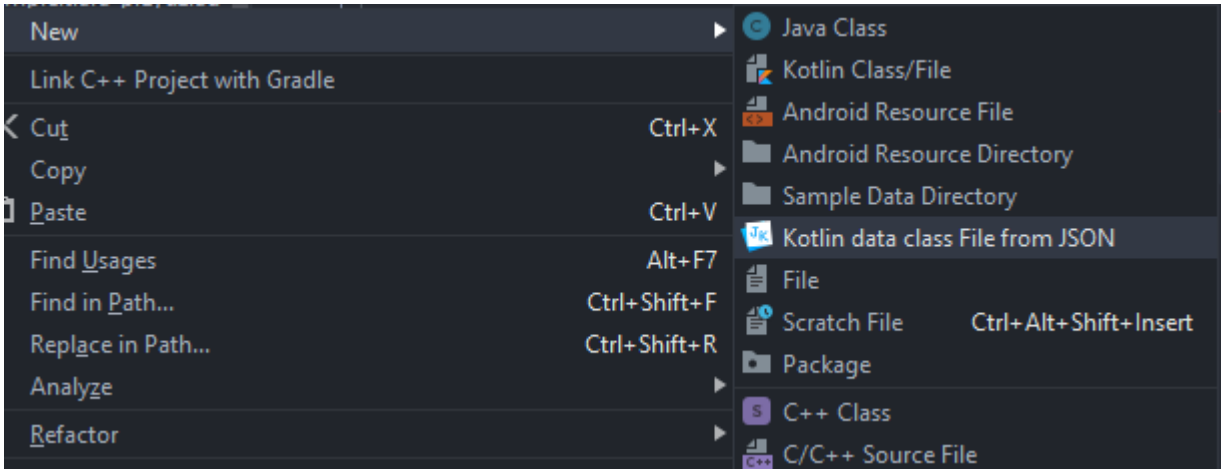
```
implementation 'com.google.code.gson:gson:2.8.6'
```

Umożliwiła ona bardzo łatwą konwersję wcześniej wspomnianych typów danych, użyta została niemal w każdym fragmencie, gdzie wykonywane było zapytanie do serwera i otrzymywaliśmy odpowiedź.
Po więcej odsyłam do github i oficjalnej dokumentacji (https://github.com/google/gson https://sites.google.com/site/gson/gson-user-guide)
Przykłady użycia:

```
listOfJobs = Gson().fromJson(response.toString(), ProfessionForSearch::class.java)
listViewJobs.adapter = MyCustomAdapter( context: this@ShowJobsFromCodes, listOfJobs);
```

Tutaj następuje konwersja uzyskanej odpowiedzi z serwera do żądanej przez nad Data Class.

Na uwagę zasługuje też użycie specjalnej wtyczki przy której pomocy możemy jednym kliknięciem wygenerować odpowiednie Data Classy z danego JSON-a



W ten sposób nie tylko oszczędzamy czas, ale nie narażamy się na błędy.

7. Wizualizacja poszczególnych danych na ekranie

Zajmijmy się teraz strukturą tego co widzi nasz użytkownik.
Jako głównego konstruktora dla poszczególnych okien użyłem

```
implementation "androidx.constraintlayout:constraintlayout:2.0.4"
```

Link do dokumentacji
(https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout)
Po starannej weryfikacji uznałem go za najlepszy wybór, gdyż umożliwia pełna kontrolę nad elementami rozmieszczonymi na danym ekranie.
Większość okien zawiera powtarzalną ilość użytych elementów jak Button TextView, TextEdit.
Przejdźmy do opisu najważniejszej struktury która umożliwia użytkownikowi zobaczenie większości aplikacji.

7.1 Użycie ListView i DropDownList oraz omówienie ich elementów

Jednym z najważniejszych elementów użytych do wyświetlania poszczególnych wyników jest ListView oraz DropDownTextView. Aby skorzystać z DropDownTextView należy zaimplementować odpowiedniego githuba.

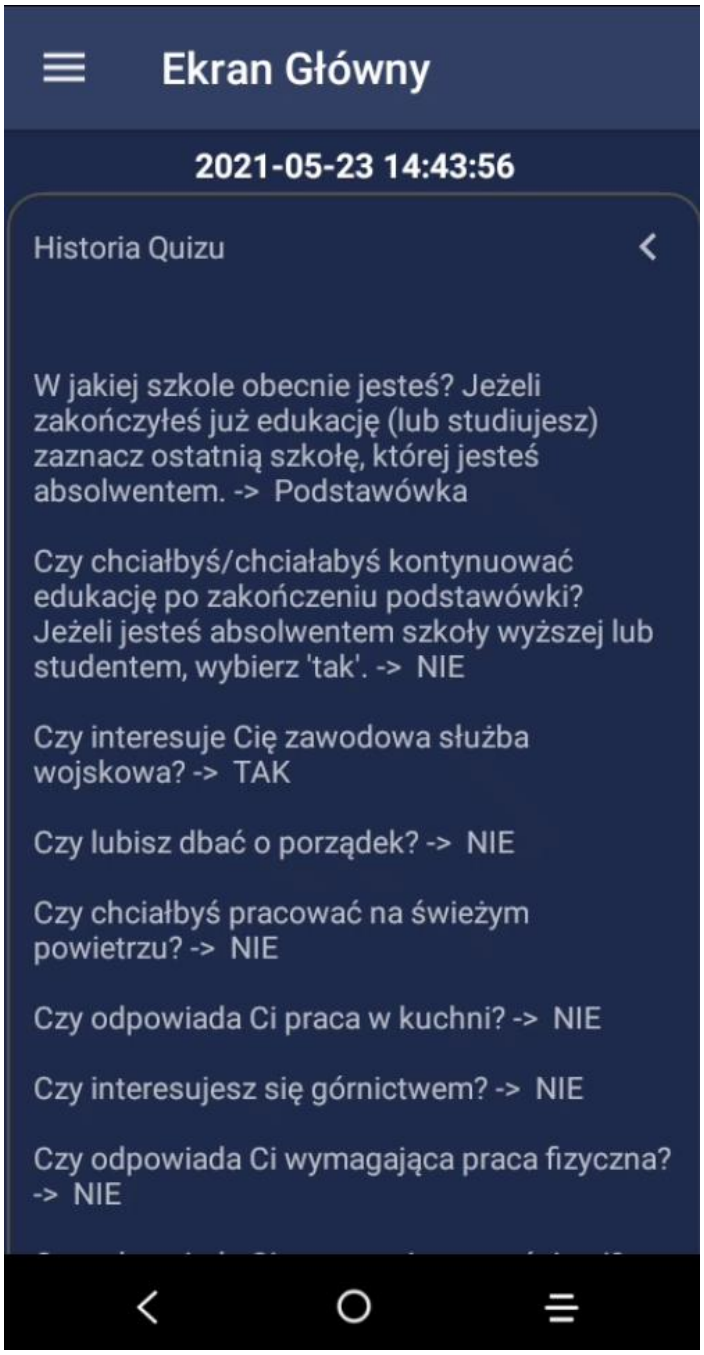
```
implementation 'com.github.hakobast:dropdown-textview:0.1.1'
```

Aby było łatwiej zrozumieć to o czym pisze, posłużę się przykładami z kodu i na ich podstawie postaram się wyjaśnić wszystkie aspekty.

Jako przykład posłużę się oknem z historią wykonanych quizów dla danego użytkownika. Po zalogowaniu i wykonaniu quizu, quiz automatycznie jest zapisywany i jest dostępny (wraz z innymi wykonanymi quizami) do wglądu.

Spójrzmy najpierw jak to wygląda od strony użytkownika aplikacji:
Ekran ten jest opisany szerzej w dokumentacji użytkownika do której odsyłam po więcej informacji.


Ekran Główny
2021-05-23 14:43:56
Historia Quizu
POKAŻ QUIZ
2021-05-24 08:32:08
Historia Quizu
POKAŻ QUIZ
2021-05-24 08:36:34
Historia Quizu
POKAŻ QUIZ
2021-05-24 08:45:31
Historia Quizu
POKAŻ QUIZ
2021-05-24 08:50:16
Historia Quizu



Przeanalizujmy jednak jak on wygląda.
Jak widać sam ekran jest podzielony na kolumny. Za to odpowiada ListView.
Tak wygląda .xml tego ekranu:

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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"
    android:background="@color/tlo_1"
    style="@style/Theme.Design.NoActionBar"
    android:scrollbarStyle="insideInset"
    >


    <ListView
        android:id="@+id/quizHistoryListView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:dividerHeight="5sp"
        android:divider="@android:color/transparent"
        />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Przejdźmy jednak do analizy samego kodu odpowiedzialnego za wyświetlanie danych na ekranie.

Na początku, zaraz po tym jak użytkownik przejdzie do ekranu wykonujemy zapytanie o wykonane przez niego quizy:

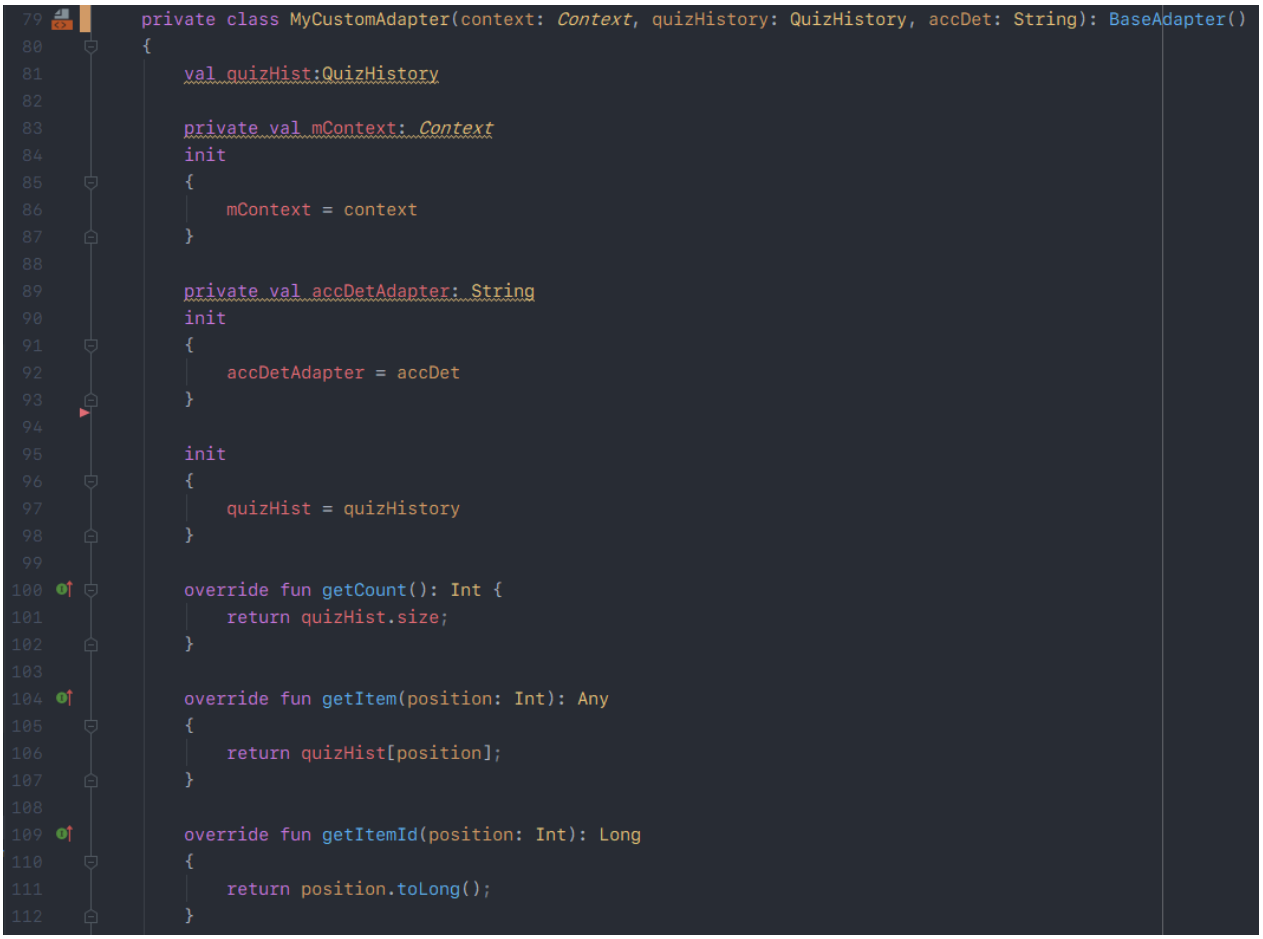
```
private fun volleyQuizHistory(id: Int, username: String, v: View, accDet: String)
{
    val url: String = "https://programowaniezespolowe-app.herokuapp.com/api/quizhistory/getuserhistory"
    var userLogIndata : UserLogInData? = Gson().fromJson(accDet, UserLogInData::class.java)

    val jsonreq = JSONObject()
    jsonreq.put( name: "userId", id)
    jsonreq.put( name: "username", username)

    println(jsonreq)

    val requestQueue = Volley.newRequestQueue(context)
    val jsonObjectRequest = object:JsonArrayRequestMy(
            Request.Method.POST, url, jsonreq,
            Response.Listener<JSONArray?> { response ->
                val quizHist = Gson().fromJson(response.toString(), QuizHistory::class.java)
                val listViewQuizHistory = v.findViewById<ListView>(R.id.quizHistoryListView)
                listViewQuizHistory.adapter = MyCustomAdapter(v.context, quizHist, accDet);
                println("QUESTIONHISRESP: ${response}")
            },
            Response.ErrorListener { error ->
                error(error)
            }
    ){
        override fun getHeaders(): MutableMap<String, String> {
            val headers = HashMap<String, String>()
            headers["Authorization"] = "Bearer ${userLogIndata?.token}"
            return headers
        }
    }
    requestQueue.add(jsonObjectRequest)
}
```

warto tutaj zwrócić uwagę na linijkę:

```
listViewQuizHistory.adapter = MyCustomAdapter(v.context, quizHist, accDet);
```

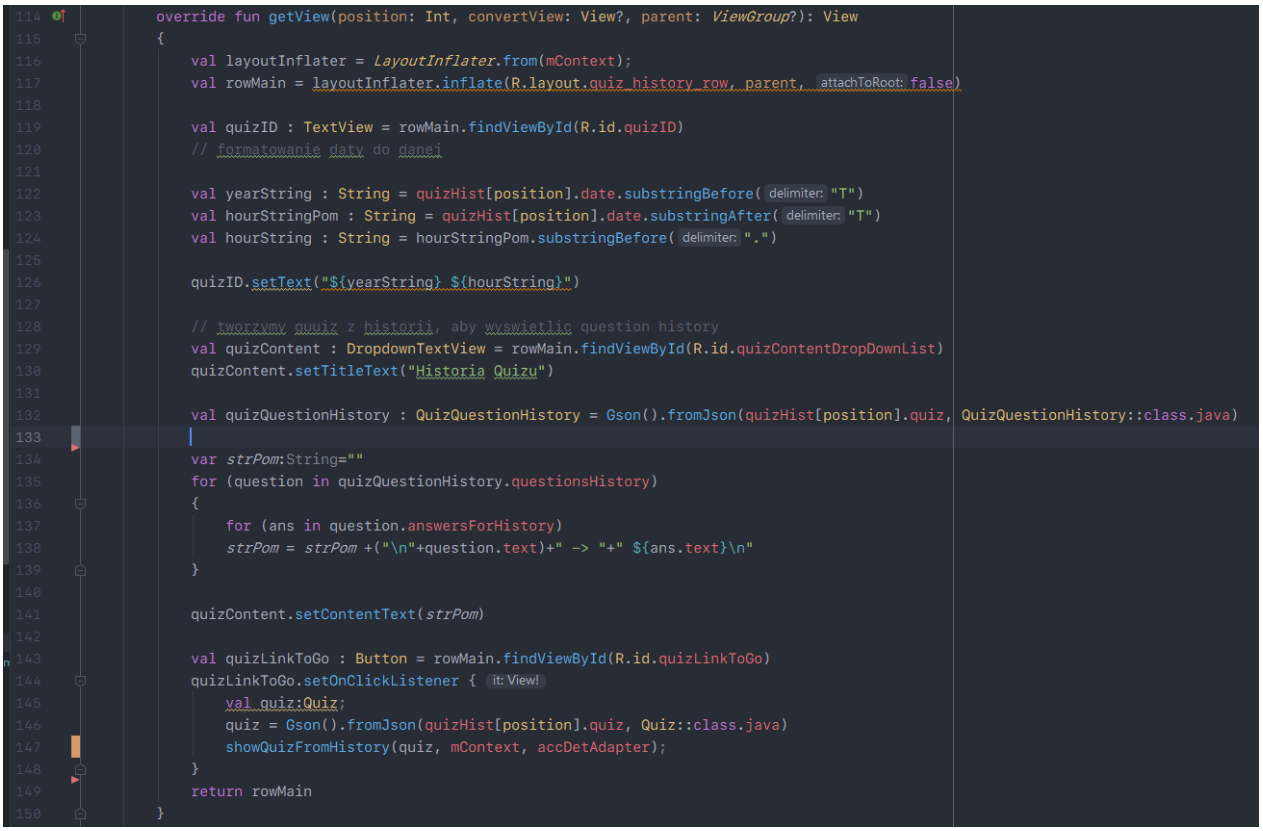
MyCustomAdapter to funkcja reprezentująca BaseAdapter, używany do wyświetlania elementów w ListView.
W naszej aplikacji jest to powszechna struktura, wszystkie okna wyświetlające dane dla użytkownika będą miały taką samą strukturę jak wyżej zaprezentowany przykład.

Spójrzmy na implementację MyCustomAdapter



```
79  private class MyCustomAdapter(context: Context, quizHistory: QuizHistory, accDet: String): BaseAdapter()
80  {
81      val quizHist:QuizHistory
82
83      private val mContext: Context
84      init
85      {
86          mContext = context
87      }
88
89      private val accDetAdapter: String
90      init
91      {
92          accDetAdapter = accDet
93      }
94
95      init
96      {
97          quizHist = quizHistory
98      }
99
100     override fun getCount(): Int {
101         return quizHist.size;
102     }
103
104     override fun getItem(position: Int): Any
105     {
106         return quizHist[position];
107     }
108
109     override fun getItemId(position: Int): Long
110     {
111         return position.toLong();
112     }
```

Na początku musimy zainicjalizować poszczególne wartości, jest to wymagane jeśli chcemy implementować BaseAdapter.



```
114 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View
115 {
116     val layoutInflater = LayoutInflater.from(mContext);
117     val rowMain = layoutInflater.inflate(R.layout.quiz_history_row, parent, attachToRoot: false)
118
119     val quizID : TextView = rowMain.findViewById(R.id.quizID)
120     // formatowanie daty do danej
121
122     val yearString : String = quizHist[position].date.substringBefore( delimiter: "T")
123     val hourStringPom : String = quizHist[position].date.substringAfter( delimiter: "T")
124     val hourString : String = hourStringPom.substringBefore( delimiter: ".")
125
126     quizID.setText("${yearString} ${hourString}")
127
128     // tworzymy quuiz z historii, aby wyswietlic question history
129     val quizContent : DropdownTextView = rowMain.findViewById(R.id.quizContentDropDownList)
130     quizContent.setTitleText("Historia Quizu")
131
132     val quizQuestionHistory : QuizQuestionHistory = Gson().fromJson(quizHist[position].quiz, QuizQuestionHistory::class.java)
133
134     var strPom:String=""
135     for (question in quizQuestionHistory.questionsHistory)
136     {
137         for (ans in question.answersForHistory)
138         strPom = strPom +("\n"+question.text)+" -> "+ ${ans.text}\n"
139     }
140
141     quizContent.setContentText(strPom)
142
143     val quizLinkToGo : Button = rowMain.findViewById(R.id.quizLinkToGo)
144     quizLinkToGo.setOnClickListener { it: View!
145         val quiz:Quiz;
146         quiz = Gson().fromJson(quizHist[position].quiz, Quiz::class.java)
147         showQuizFromHistory(quiz, mContext, accDetAdapter);
148     }
149     return rowMain
150 }
```

W linijce 117 wybieramy plik .xml t którym zawiera się wygląd jaki chcemy aby miał wiersz w naszym LisView (elementy, rozmieszczenie itd.)

Oto kod quiz_history_row

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:gravity="center"
    android:layout_gravity="center"
    android:orientation="vertical">

    <TextView
        android:id="@+id/quizID"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAlignment="center"
        android:textColor="@color/white"
        android:textSize="15sp"
        android:textStyle="bold"
        android:paddingTop="5sp"
        />


    <hakobastvatsatryan.DropdownTextView
        android:id="@+id/quizContentDropDownList"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        app:arrow_drawable="@drawable/ic_keyboard_arrow_down"
        android:scrollbarThumbVertical="@color/buttno_color"
        app:bg_drawable_regular="@drawable/dropdowntext_frame"
        android:layout_marginTop="2sp"
        android:layout_marginBottom="2sp"
        android:layout_marginLeft="2sp"
        android:layout_marginRight="2sp"
        />

```

```
34
35      <androidx.appcompat.widget.AppCompatButton
36          android:id="@+id/quizLinkToGo"
37          android:layout_width="wrap_content"
38          android:layout_height="24sp"
39          android:paddingTop="2sp"
40          android:paddingBottom="2sp"
41          android:background="@drawable/btn_bg"
42          android:textAlignment="center"
43          android:textColor="@color/white"
44          android:text="POKAŻ QUIZ"
45          android:drawableEnd="@drawable/ic_arrow_forward"
46          android:textSize="13sp"
47          android:textStyle="bold"
48          android:onClick="onClick"
49          android:clickable="true"
50          android:paddingRight="5sp"
51          android:paddingLeft="5sp"
52          />
53
54  </LinearLayout>
```

W metodzie getView bierzemy id poszczególnych elementów z quiz_history_row i przypisujemy im odpowiednie wartości.
Zwrócić należy uwagę na to iż tutaj niezbędnym elementem jest quizHist z 81 linijki. Jest to Data Class, której odpowiednie pola należy wyświetlić dla użytkownika.
Jak wspominałem wcześniej wszystkie ekrany które mają ListView stosują taka samą metodologię z adapterem i rozwijaną listą. Jedyna różnica to wyświetlane ilości i inna Data Class której pola mamy wyświetlić.
Data Classy mogą ze sobą współpracować jak w tym przypadku, przykład:

```kotlin
data class QuizQuestionHistory(
    val answerIds: List<Any>,
    val groupCodes: List<String>,
    val questionList: List<Any>,
    val questionsHistory: List<QuestionsHistory>
)
```

```kotlin
data class QuizHistoryItem(
    val date: String,
    val id: Int,
    val quiz: String,
    val userId: Int,
    val uuid: String
)
```

```kotlin
class QuizHistory : ArrayList<QuizHistoryItem>()
```

Wszystko polega tylko na odpowiednim wyświetleniu i formatowaniu poszczególnych danych, co zazwyczaj dokonuje się w sekcji MyCustomAdaptera.

8. Linki do dokumentacji i przydatnych źródeł

- https://material.io/develop/android
- https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout
- https://sites.google.com/site/gson/gson-user-guide
- https://developer.android.com/training/volley
- https://kotlinlang.org/docs/home.html