



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическое занятие № 7/2ч.

Разработка мобильных компонент анализа безопасности
информационно-аналитических систем

	<i>(наименование дисциплины (модуля) в соответствии с учебным планом)</i>
Уровень	бакалавриат
	<i>(бакалавриат, магистратура, специалитет)</i>
Форма обучения	очная
	<i>(очная, очно-заочная, заочная)</i>
Направление(-я) подготовки	09.03.02 «Информационные системы и технологии»
	<i>(код(-ы) и наименование(-я))</i>
Институт	комплексной безопасности и специального приборостроения ИКБСП
	<i>(полное и краткое наименование)</i>
Кафедра	КБ-3 «Безопасность программных решений»
	<i>(полное и краткое наименование кафедры, реализующей дисциплину (модуль))</i>
Используются в данной редакции с учебного года	2022/23
	<i>(учебный год цифрами)</i>
Проверено и согласовано « ____ » _____ 20 ____ г.	
	<i>(подпись директора Института/Филиала с расшифровкой)</i>

Москва 2023 г.

ОГЛАВЛЕНИЕ

1	СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ В OS ANDROID.....	3
1.1	Реализация HttpURLConnection и HttpClient	5
1.2	Формат обмена данными	8
1.3	Задание. Socket.....	11
1.4	Задание. HttpURLConnection	13
2	СТОРОННИЕ БИБЛИОТЕКИ	18
2.1	Retrofit.....	18
2.2	FIREBASE.	18
2.3	Firebase Authentication	20
2.4	Задание.....	20
3	КОНТРОЛЬНОЕ ЗАДАНИЕ	30

1 СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ В OS ANDROID

Реализация сетевого взаимодействия с удаленной инфраструктурой требует учитывать следующие особенности:

1. Количество трафика может быть ограничено оператором связи.
2. Ограниченный заряд батареи устройства.
3. Защищенный канал связи между клиентом и серверной частью.

Существуют несколько подходов реализации сетевого взаимодействия между клиент-серверными приложениями. Первый подход реализован на основе применения сокетов (работа непосредственно с «*Socket API*», рисунок 1.1). Наиболее часто используется в приложениях, где наиболее важными требованиями являются скорость доставки сообщений, порядок доставки сообщений и необходимость поддержки стабильного соединения с сервером. Примером таких приложений являются мессенджеры, играх, такси и т.д.

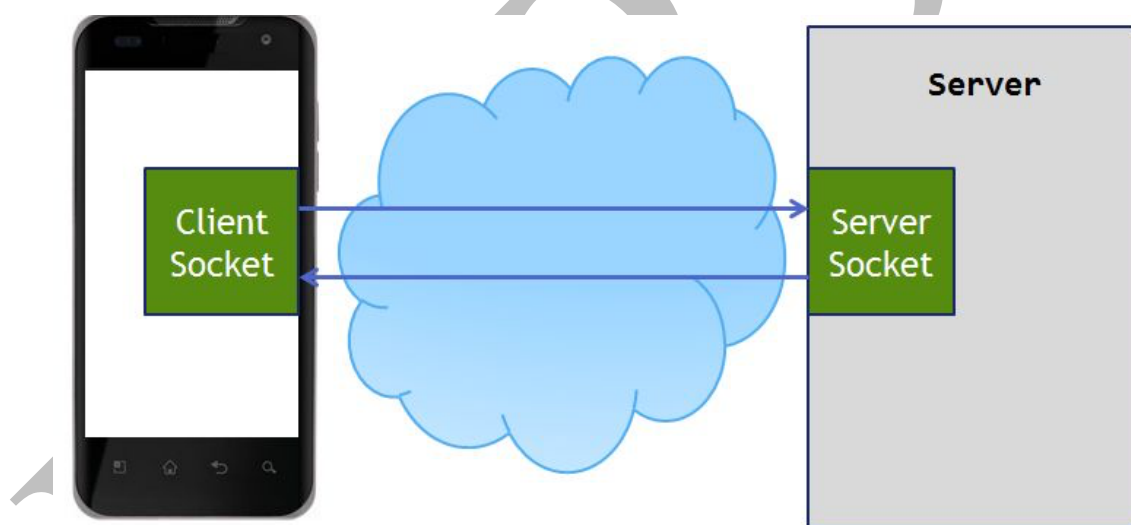


Рисунок 1.1 – Последовательность взаимодействия при Socket API

Вторым подходом взаимодействия являются «частые опросы» (англ. «*polling*»): клиент формирует запрос получения новых данных на сервере, на что сервер в ответе передает все, что у него скопилось к этому моменту (рисунок 1.2). Недостатком данного подхода является то, что клиент не обладает знаниями о появлении новой информации на сервере, т.е. присутствует избыточный трафик, в первую очередь из-за частых установок соединений с сервером.

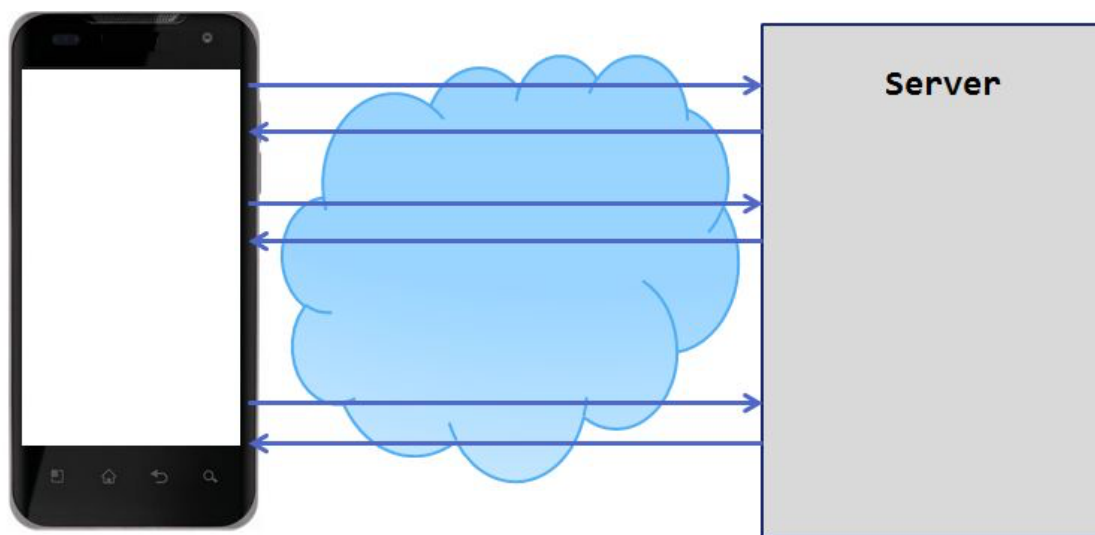


Рисунок 1.2 – Последовательность взаимодействия при частых опросах

Третьим подходом организации сетевого обмена данными являются «длинные опросы» (англ. «*long polling*»), заключающиеся в том, что клиент отправляет «ожидающий» запрос на сервер. Сервер выполняет поиск новой информации для клиента, и в случае их отсутствия, держит соединение до тех пор, пока они не появятся (рисунок 1.3). В случае появления сведений для клиента они передаются в качестве ответа на запрос. Клиент, получив данные от сервера, тут же посылает следующий «ожидающий» запрос и т.д.

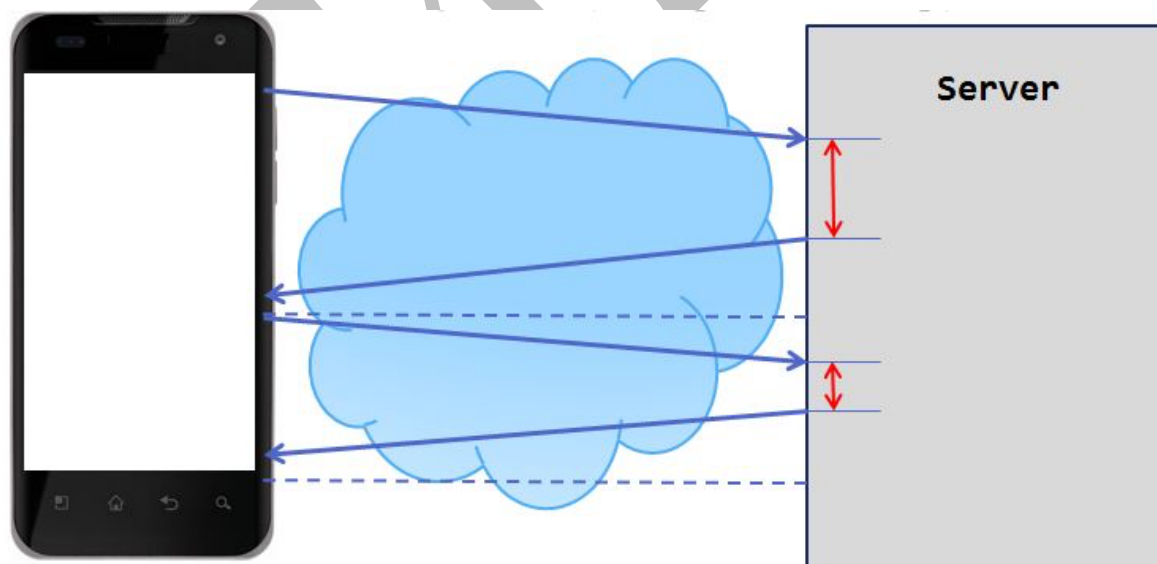


Рисунок 1.3 – Последовательность взаимодействия с длинными запросами

Реализация данного подхода имеет недостаток на мобильных устройствах в первую очередь из-за нестабильности мобильного соединения – требуется постоянная проверка причины разрыва и организации нового соединения. Однако,

при данном подходе расходуется меньше трафика, чем при обычном «частом опросе», т.к. сокращается количество установок соединений с сервером.

Механизм «длинных опросов» или пуш-уведомлений (англ. «*push notifications*»), реализован с помощью сервиса «*Firebase*» в ОС «*Android*». Использование данного сервиса является наиболее оптимальным для большинства задач уведомлений. Стоит отметить, что взаимодействие между сервером и клиентом организуется с помощью инфраструктуры «*Firebase*», который реализует всю логику доставки данных до потребителя. Достоинством данного подхода является устранение необходимости частых установок соединения с сервером за счет того, что разработчик точно знает, что данные появились, и об этом вас оповещает сервис.

Наиболее популярными подходами, используемыми при разработке приложений, являются пуш-уведомления и частые опросы. Далее будут рассмотрены инструменты, которые имеются в наличии у разработчика для работы в рамках протоколов HTTP/HTTPS.

1.1 Реализация `HttpURLConnection` и `HttpClient`

Основными библиотеками, используемыми для работы с протоколами HTTP/HTTPS и включенными в *Android SDK* являются «*java.net.HttpURLConnection*» и «*org.apache.http.client.HttpClient*».

Класс «*java.net.HttpURLConnection*» является дочерним классом «*java.net.URLConnection*» и позволяет реализовать работу по отправке и получению данных по протоколу HTTP (рисунок 1.4). Родительский класс «*URLConnection*» предназначен для реализации канала связи с удаленной инфраструктурой не только по HTTP-протоколу, а также ftp, smtp, POP3 и т.д.

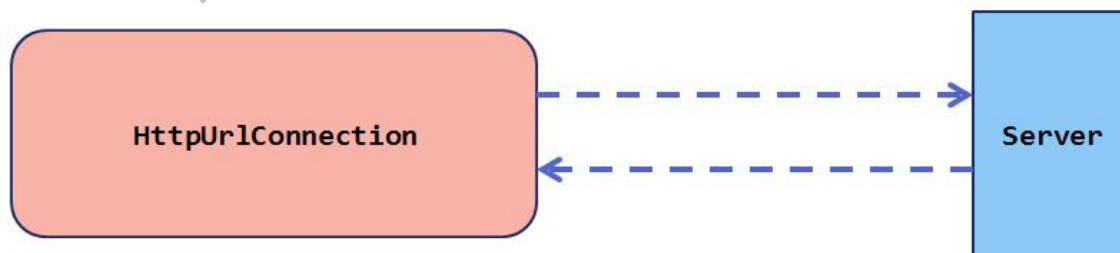


Рисунок 1.4 – Схема взаимодействия сервера с `HttpURLConnection`

Реализация класса «*HttpClient*» содержит несколько абстракций. Базовое применение подразумевает использование пяти различных интерфейсов: «*HttpRequest*», «*HttpResponse*», «*HttpEntity*» и «*HttpContext*» (рисунок 1.5).

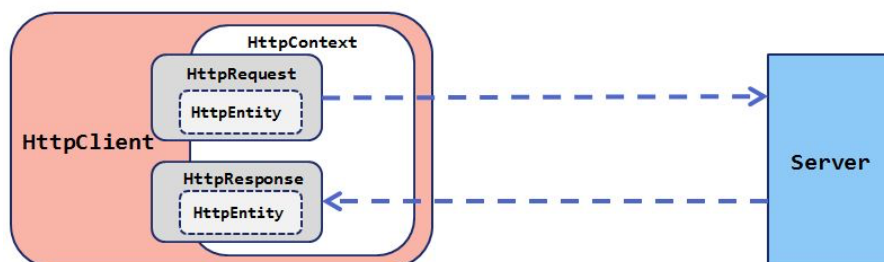


Рисунок 1.5 – Схема взаимодействия сервера с «*HttpClient*»

Как правило, при реализации приложений применяется один экземпляр класса «*HttpClient*» в связи с его количеством запрашиваемых ресурсов (рисунок 1.6). Для реализации данного подхода используется паттерн «*Singleton*». Возможно, к примеру, хранить экземпляр HTTP-клиента в наследнике класса «*Application*».

В случае «*HttpURLConnection*» следует создавать на каждый запрос новый экземпляр клиента (рисунок 1.6).

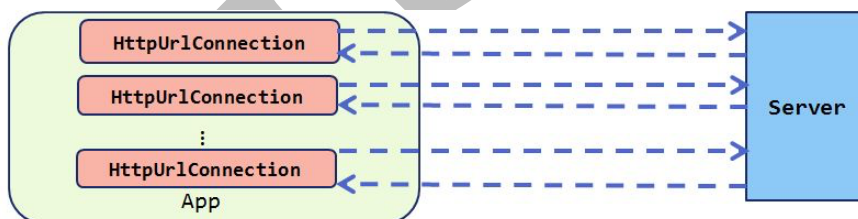


Рисунок 1.6 – Схема взаимодействия сервера с *HttpClient*

Реализация типовой схемы обмена данными приложениями с удаленной инфраструктурой представлена на рисунке 1.7.

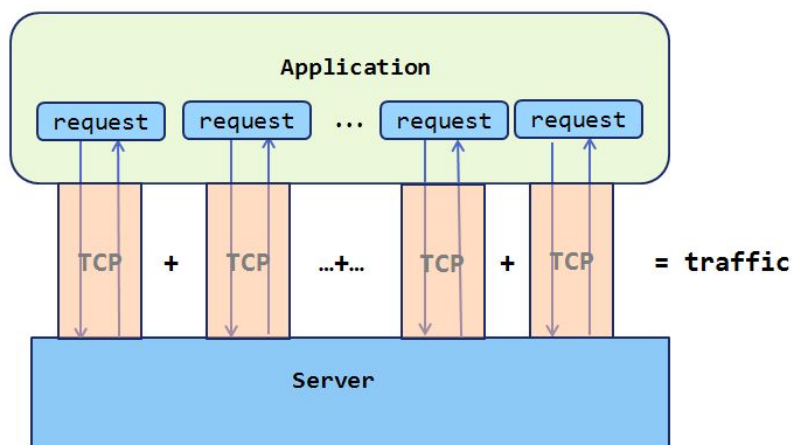


Рисунок 1.7 – Типовая схема функционирования клиент-серверного приложения

Количество и частота запросов зависят от используемого подхода реализации приложения. Каждый запрос требует установки TCP-соединения с серверной частью. В приведенном примере объем переданного трафика равняется сумме трафика установки соединений и переданных данных. Снижение расхода трафика возможно за счет применения долгоживущего соединения (англ. «*keep alive*»), приведенной на рисунке 1.8.

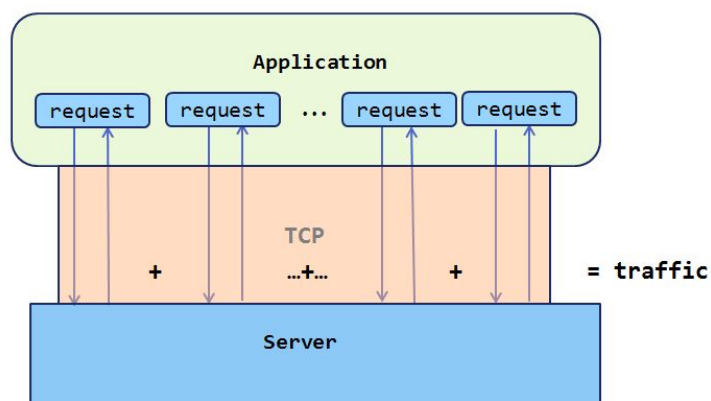


Рисунок 1.8 – Типовая схема функционирования клиент-серверного приложения с использованием параметра «*keep-alive*»

Применение долгоживущего соединения заключается в использовании одного и того же TCP-соединения для отправки и приема HTTP-запросов. Главным преимуществом является снижение объема передаваемого трафика и времени выполнения запроса. Далее приведен пример создания 10 тестовых запросов с различным значением параметра «*keepalive*». Данные представлены в таблице 1.1.

Таблица 1.1 Временные характеристики при использовании KeepAlive

Номер запроса	Время (мс.) при использовании параметра KeepAlive=false	Время (мс.) при использовании параметра KeepAlive=true
1.	2098	2023
2.	2157	1604
3.	2037	1698
4.	2096	1774
5.	1944	1173
6.	2055	1573
7.	1865	1683
8.	2119	1670
9.	1986	1666
10.	1965	1541
	≈2032,2	≈1700,5

Применение параметра «*KeepAlive*» позволяет снизить временные затраты с двух секунд до 1,7 секунды (на 16% быстрее). Данный показатель обуславливается тем, что устраняется необходимость частой установки соединения с серверной частью. В условиях применения HTTPS-соединения разница будет значительней, т.к. процедура «*SSL Handshake*» является более затратной чем «*TCP Handshake*».

Важным параметром долгоживущего соединения является «*keep alive duration*». Данный параметр предназначен для установки временного интервала существования соединения. В случае осуществления нескольких HTTP-запросов в пределах одного интервала времени, будет использоваться установленное TCP-соединение. На рисунке 1.9 представлен пример превышения значения параметра «*keep alive duration*» между третьим и четвертым запросами с дополнительным созданием нового TCP-соединения с сервером.

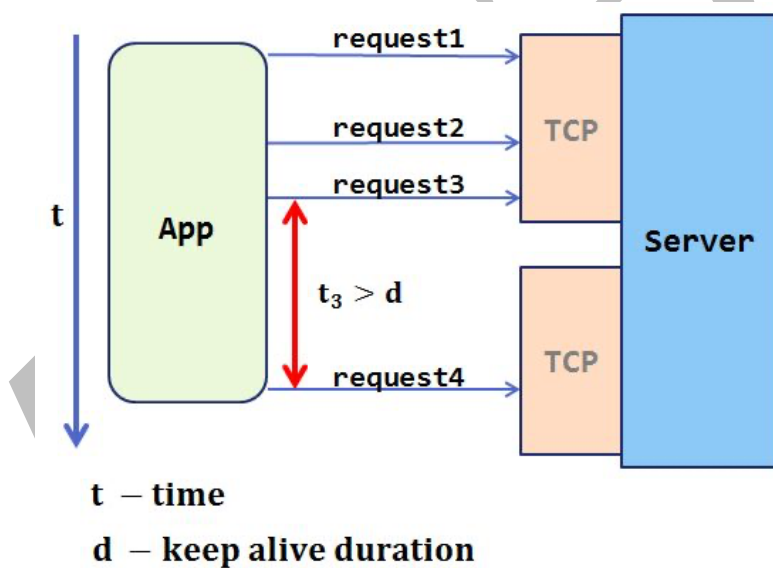


Рисунок 1.9 – Схема функционирования клиент-серверного приложения при установленном параметре «*keep alive duration*»

1.2 Формат обмена данными

В настоящее время в качестве формата данных, предназначенных для взаимодействия между клиентами и серверной частью, широко используются два формата: XML и JSON. Формат обмена данными JSON является средством кодирования объектов «*JavaScript*» в строковое. Структуру данных в JSON возможно представить как комбинацию:

- JSON-объектов: «{ключ: значение, ...}» (рисунок 1.10);

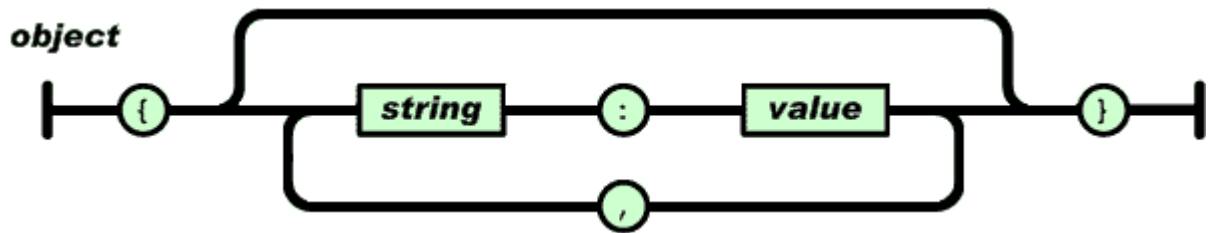


Рисунок 1.10 Структура JSON-объекта

- JSON-массивов: «[value,value]» (рисунок 1.11);

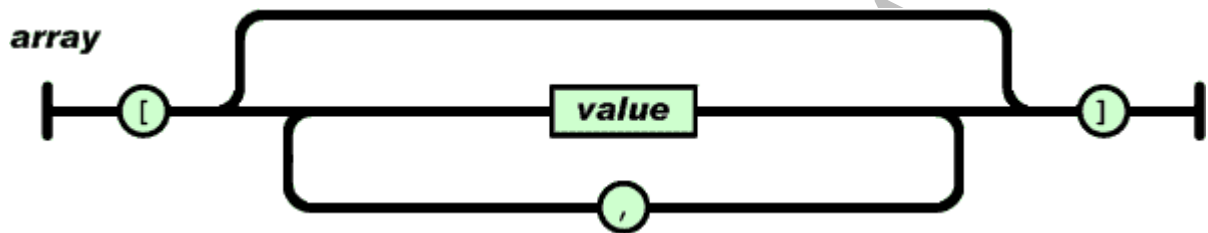


Рисунок 1.11 Структура JSON-массива

- «JSONStringer» предназначен для кодирования примитивных типов, объектов или массивов: «string, number, object, array, true, false, null» (рисунок 1.12).

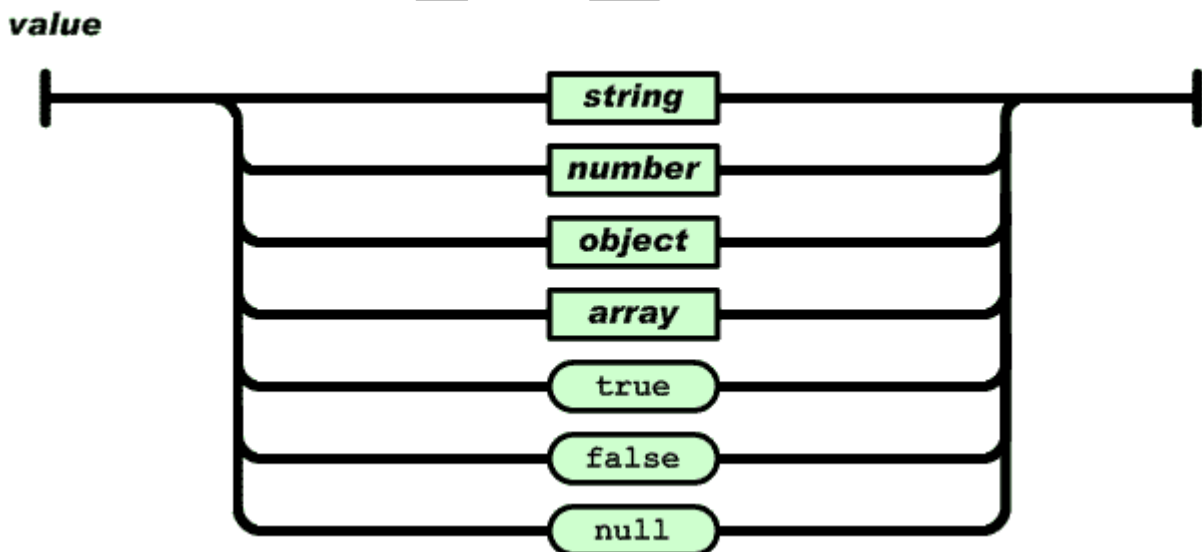


Рисунок 1.12 Структура JSONStringer

Для работы с JSON в пакете «*json.org*» используются следующие классы: «JSONArray», «JSONObject», «JSONStringer», «JSONTokenizer», «JSONException».

Пример записи JSON-объекта:

```
{
  "name": "mirea",
  "course": 4
}
```

Пример записи JSON-массива:

```
{
  "name": "mirea",
  "course": 4,
  "address": {
    "postal": 111111,
    "country": "Russia",
    "city": "Moscow",
    "building": "Stromynka,20"
  },
  "phoneNumbers": [
    {
      "type": "03",
      "number": 111
    },
    {
      "type": "Стромынка",
      "number": 222
    },
    {
      "type": "Виртуальность",
      "number": 333
    }
  ]
}
```

В представленном примере объект *«address»* содержит несколько полей с различными значениями, а поле *«phoneNumbers»* является массивом. Создание объектов производится следующим образом:

```
JSONObject jsonObject = new JSONObject();
```

Запись значений в объект производится вызовом метода *«put»* и установкой ключей и значений в качестве аргументов:

```
jsonObject.put("id", 10);
jsonObject.put("name", "mirea");
```

Создание массива подразумевает использование класса *«JSONArray»* с передачей в метод *«put»* либо примитивов, либо JSON-объектов:

```
JSONArray jsonArrayMessages = new JSONArray();
JSONObject jsonMessage = new JSONObject();
jsonMessage.put("id_department", 1);
jsonMessage.put("value", "ИКБСП");
jsonArrayMessages.put(jsonMessage);
```

Также, возможно добавление массива в JSON-объект:

```
jsonObject.put("departments", jsonArrayMessages);
```

Возможно создание JSON-объекта из строки:

```
JSONObject jsonObject = new JSONObject(result);
```

Получение значения осуществляется с помощью вызова метода *«get*»*

с указанием ключа из объекта JSON:

```
String name = jsonObject.getString("name");
```

Получение нужного массива выполняется с помощью вызова метода «*getJSONArray*» с указанием ключа:

```
JSONArray jArray = jsonObject.getJSONArray("departments");
```

Полученный массив также позволяет вернуть JSON-объект:

```
JSONObject msg = jArray.getJSONObject(1);  
int id_message = msg.getInt("id_department");
```

1.3 Задание. Socket

Требуется создать новый проект «ru.mirea.«фамилия».Lesson7». В меню «*File | New Project | Empty Views Activity*». Название модуля «*TimeService*».

Для организации соединения с сервером с помощью сокетов необходимо создание экземпляра класса «*Socket*» с указанием ip-адреса и порта. После получения данных требуется закрыть сокет. В случаях, когда указан недоступный адрес, вернётся исключение «*UnknownHostException*».

В открытом доступе существуют различные веб-серверы, предоставляющие возможность подключения к ним используя сокет. Например, сервера времени возвращают информацию о текущем времени. Рассмотрим работу с сервисом «*time.nist.gov*», возвращающий ответ в виде двух строк. В первую очередь требуется создать отдельный класс с методом для чтения поступающих данных.

```
public class SocketUtils {  
    /**  
     * BufferedReader для получения входящих данных  
     */  
    public static BufferedReader getReader(Socket s) throws IOException {  
        return (new BufferedReader(new InputStreamReader(s.getInputStream())));  
    }  
  
    /**  
     * Makes a PrintWriter to send outgoing data. This PrintWriter will  
     * automatically flush stream when println is called.  
     * В примере не используется  
     */  
    public static PrintWriter getWriter(Socket s) throws IOException {  
        // Second argument of true means autoflush.  
        return (new PrintWriter(s.getOutputStream(), true));  
    }  
}
```

Внешний вид экрана представлен на рисунке 1.13 и содержит одно текстовое поле и кнопку:

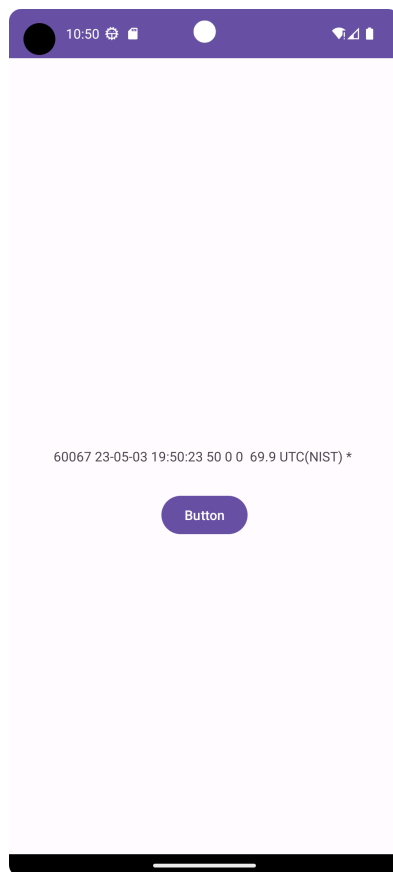


Рисунок 1.13 – Внешний вид приложения Socket

Сервис времени размещается по адресу «*time.nist.gov*» на порту «13». Для обеспечения доступа к интернету необходимо соответствующее разрешение в манифест-файле.

```
<uses-permission android:name="android.permission.INTERNET" />
```

В данном примере для выполнения задачи в другом потоке будет использоваться класс «*AsyncTask*». С версии API 30 данный класс является устаревшим. В данном механизме основным методом является «*doInBackground*» – в котором выполняются основные вычисления и который возвращает результат вычислений в метод «*onPostExecute*».

```

public class MainActivity extends AppCompatActivity {
    private ActivityMainBinding binding;
    private final String host = "time.nist.gov"; // или time-a.nist.gov
    private final int port = 13;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        binding.button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                GetTimeTask timeTask = new GetTimeTask();
                timeTask.execute();
            }
        });
    }

    private class GetTimeTask extends AsyncTask {
        @Override
        protected String doInBackground(Void... params) {
            String timeResult = "";
            try {
                Socket socket = new Socket(host, port);
                BufferedReader reader = SocketUtils.getReader(socket);
                reader.readLine(); // игнорируем первую строку
                timeResult = reader.readLine(); // считываем вторую строку
                Log.d(TAG, timeResult);
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return timeResult;
        }

        @Override
        protected void onPostExecute(String result) {
            super.onPostExecute(result);
            binding.textView.setText(result);
        }
    }
}

```

Для **ВЫПОЛНЕНИЯ ЗАДАНИЯ** требуется разобрать полученную строку и создать экран отображения времени и даты.

1.4 Задание. HttpURLConnection

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Название модуля «*HttpURLConnection*». Требуется определить внешний IP-адрес устройства.

Класс «*java.net.HttpURLConnection*» является дочерним классом «*java.net.URLConnection*» и позволяет реализовать работу по отправке и получению данных из сети по протоколу HTTP. Данный класс следует использовать для

отправки и получения потоковых данных, размеры которых нельзя заранее определить. Алгоритм использования механизма:

- получение и инициализация объекта «*HttpURLConnection*» с помощью вызова метода «*URL.openConnection*»;

- подготовка необходимого запроса с указанием сетевого адреса сервиса. Также в запросе возможно указать различные метаданные: учётные данные, тип контента, «*cookies*» сессии и т.п.;

- формирование тела запроса (опционально). В данном случае используется метод «*setDoOutput(true)*». Передача данных, записанных в поток, возвращается через метод «*getOutputStream*»;

- обработка ответа. Заголовок ответа, как правило, включает такие метаданные как тип и длина контента, дата изменения, идентификаторы сессии. Чтение данных из потока осуществляется с помощью вызова метода «*getInputStream*».

- разъединение соединения после обработки ответа сервера осуществляется с помощью вызова метода «*disconnect*» экземпляра класса «*HttpURLConnection*».

По умолчанию «*HttpURLConnection*» используется метод «*GET*». Использование метода «*POST*» достигается вызовом функции «*setDoOutput(true)*» с последующей передачей данных в исходящий поток «*openOutputStream*». Настройка других HTTP-методов (OPTIONS, HEAD, PUT, DELETE and TRACE) осуществляется с помощью методов «*setRequestMethod(String)*».

Возможно использование прокси-сервера при создании соединения с помощью «*URL.openConnection(Proxy)*». Каждый экземпляр «*HttpURLConnection*» может использоваться только для одной пары запроса/ответа. Операции с соединениями следует проводить в отдельном потоке.

Определение выходного ip-адреса предполагает наличие доступа в интернет. Необходимо установить требуемые разрешения для работы с интернетом и сетью в манифест-файле:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
```

На первом этапе проводится проверка наличия подключения к интернету

с помощью «*ConnectivityManager*».

```
public void onClick(View view) {
    ConnectivityManager connectivityManager =
        (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkinfo = null;
    if (connectivityManager != null) {
        networkinfo = connectivityManager.getActiveNetworkInfo();
    }

    if (networkinfo != null && networkinfo.isConnected()) {
        new DownloadPageTask().execute("https://ipinfo.io/json"); // запуск нового по-
    } else {
        Toast.makeText(this, "Нет интернета", Toast.LENGTH_SHORT).show();
    }
}
```

Взаимодействие с удаленным сервером, в случае наличия интернета, осуществляется в отдельном потоке с помощью «*AsyncTask*». В приведенном примере ниже на первом этапе в методе «*doInBackground*» вызывается «*downloadIpInfo*», который возвращает текст с указанной страницы. Вначале указывается нужный адрес для загрузки в переменной типа URL. Данный адрес передается в метод «*doInBackground*», а затем в метод *downloadOneUrl*. На втором шаге внутри метода создается экземпляр класса «*HttpURLConnection*» через вызов *URL.openConnection()*. Далее проверяется статус код запроса и в случае успешного получения кода 200 (константа *HttpURLConnection.HTTP_OK*) производится обработка входящего потока данных. Существуют различные способы чтения потоков данных. В данном примере используется класс «*ByteArrayOutputStream*». В методе *onPostExecute* представлен пример, как из ответа формируется JSON объект и извлекается значение ip адреса. Итоговый текст попадает в переменную «*data*» и выводится в «*TextView*» для отображения результата.

```

private class DownloadPageTask extends AsyncTask<String, Void, String> {
    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        binding.textView.setText("Загружаем...");
    }
    @Override
    protected String doInBackground(String... urls) {
        try {
            return downloadIpInfo(urls[0]);
        } catch (IOException e) {
            e.printStackTrace();
            return "error";
        }
    }
    @Override
    protected void onPostExecute(String result) {
        binding.textView.setText(result);
        Log.d(MainActivity.class.getSimpleName(), result);
        try {
            JSONObject responseJson = new JSONObject(result);
            Log.d(MainActivity.class.getSimpleName(), "Response: " + responseJson);
            String ip = responseJson.getString("ip");
            Log.d(MainActivity.class.getSimpleName(), "IP: " + ip);
        } catch (JSONException e) {
            e.printStackTrace();
        }
        super.onPostExecute(result);
    }
}

private String downloadIpInfo(String address) throws IOException {
    InputStream inputStream = null;
    String data = "";
    try {
        URL url = new URL(address);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setReadTimeout(100000);
        connection.setConnectTimeout(100000);
        connection.setRequestMethod("GET");
        connection.setInstanceFollowRedirects(true);
        connection.setUseCaches(false);
        connection.setDoInput(true);
        int responseCode = connection.getResponseCode();

        if (responseCode == HttpURLConnection.HTTP_OK) { // 200 OK
            inputStream = connection.getInputStream();
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            int read = 0;
            while ((read = inputStream.read()) != -1) {
                bos.write(read);
            }
            bos.close();
            data = bos.toString();
        } else {
            data = connection.getResponseMessage() + ". Error Code: " + responseCode;
        }
        connection.disconnect();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
    }
    return data;
}

```


Для **ВЫПОЛНЕНИЯ ЗАДАНИЯ** требуется изменить внешний вид экрана для вывода в отдельные текстовые поля значения полей: город, регион и т.д. В примере был представлена механизм извлечения значения ip – адреса из объекта JSON. Полученные координаты требуется передать в сервис погоды «https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41¤t_weather=true» и отобразить погоду на экране.

2 СТОРОННИЕ БИБЛИОТЕКИ

2.1 Retrofit

«*Retrofit*» – это библиотека для android, предназначенная для организации сетевого взаимодействия приложения с внешними ресурсами. Авторами библиотеки являются разработчики из компании «*Square*». Библиотека предназначена для получения и разбора различного вида структурированных данных от сервисов, использующие REST API. В «*Retrofit*» для (де)сериализации данных используются конвертеры, которые необходимо указывать вручную. Поддерживаются следующие конвертеры:

- Gson: com.squareup.retrofit2:converter-gson;
- Jackson: com.squareup.retrofit2:converter-jackson;
- Moshi: com.squareup.retrofit2:converter-moshi;
- Protobuf: com.squareup.retrofit2:converter-protobuf;
- Wire: com.squareup.retrofit2:converter-wire;
- Simple XML: com.squareup.retrofit2:converter-simplexml;
- JAXB: com.squareup.retrofit2:converter-jaxb;
- Scalars (primitives, boxed, and String): com.squareup.retrofit2:converter-scalars.

Для работы с «*Retrofit*» требуются следующие три класса:

- класс модель, используемый в качестве модели JSON;
- интерфейсы, определяющие возможные HTTP операции;
- экземпляр класса Retrofit.Builder, использующий интерфейсы и API Builder,

для определения конечной точки URL.

Подробная документация по библиотеке размещена по адресу: <https://square.github.io/retrofit/>.

2.2 FIREBASE.



На конференции Google I/O 2015 была представлена облачная база данных на основе NoSQL с названием «*Firebase*». В мае 2016 компания «Google» существенно модернизировала платформу, позволив выполнять построение Android, iOS и мобильных веб-приложений за счет реализованного функционала.

База данных позволяет работать с данными, хранящимся в виде JSON, синхронизировать их в реальном времени и кэшировать при отсутствии интернета. «*Firebase*» поддерживает аутентификацию по связке электронная «почта+пароль», «*Facebook*», «*Twitter*», «*Git*Hub», «*Google*» и другие сервисы, предоставляющие возможность аутентификации. Кроме базы данных «*Firebase*» предлагает хостинг статичных файлов для веб-сайта.

Firebase содержит компоненты, представленные в таблице 2.1.

Таблица 2.1 – Возможности сервиса Firebase

	База данных в реальном времени Хранение и синхронизация данных
	Аутентификация Аутентификация пользователей через облачный сервис
	Облачное хранилище Хранение и получение файлов с облачного хранилища
	Тестовая лаборатория для Android Облачная инфраструктура тестирования приложений, позволяющая выполнять тестирование приложений на различных типах устройств
	Отчеты о сбоях Обнаружение ошибок и их исправление. Сбор и отправка отладочной информации, позволяющей выполнить поиск проблем iOS/Android-приложений
	Облачные функции Реализация кода JavaScript в облачной среде Node.js.
	Хостинг Предназначен для размещения статических веб-сайтов
	Мониторинг данных о производительности Отслеживание ключевых показателей быстродействия приложений, просмотр трассировки данных
	Google Analytics Формирование отчетов о вовлеченности пользователей и производительности приложений
	AdMob Интеграция рекламы в приложениях
	Облачные сообщения Доставки push-уведомлений из облака на устройства
	Удаленное конфигурирование Позволяет подстраивать и обновлять элементы приложения во время выполнения кода без необходимости обновления пакета приложения. Возможность включения и выключения определённых элементов приложения, распространение обновлений на конкретные Аудитории пользователей.

	App Indexing Привлечение поискового трафика в приложении
	AdWords Рекламная платформа для анализа рекламных кампаний

Основная документация по сервисам размещена на сайте <https://firebase.google.com>. По данному адресу возможно перейти в консоль управления проектами. В консоли имеется возможность создания новых проектов, просмотра данных пользователей, управление файлами и базой данных и т.д. Большая часть продуктов, включая «*Realtime Analytics*», «*Crashlitics*», «*Remote Config*», «*In-App Messaging*», и «*Dynamic Links*» являются бесплатными и не имеют каких-либо ограничений. Платные сервисы типа «*Test Lab*», «*Storage*», «*Hosting*» и других имеют гибкую ценовую сетку. Бесплатный тариф «*SPARK*» имеет некоторые ограничения и предназначен для создания прототипа приложения, дипломной работы или стартапа. Для использования сервисов требуется иметь учетную запись «*Google*».

2.3 Firebase Authentication

Firebase имеет несколько способов для проведения аутентификации: по электронному адресу и паролю, учётным записям Facebook, Twitter, GitHub, анонимно и другие возможности.

2.4 Задание

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Название модуля «*FirebaseAuth*».

Возможность идентификации пользователей необходима в большинстве современных приложений. Данный функционал позволяет разделять доступ к данным, хранить личные данные пользователей в облаке и обеспечить персонализированный контент на всех устройствах пользователя.

На первом шаге требуется выполнить [регистрацию](#) в облачном сервисе (необходим «*Google*»-аккаунт). В среде разработчик «*AndroidStudio*» необходимо выбрать «*Tools|Firebase|Authentication|Authentication using a custom authentication system*»|*Connect to Firebase*». Далее следует выбрать модуль и выполнится переход

в консоль разработчика для создания нового проекта с подробной инструкцией действий. В результате подключения среды разработки и сервиса «*firebase*» может отобразиться экран, представленный на рисунке 2.1.

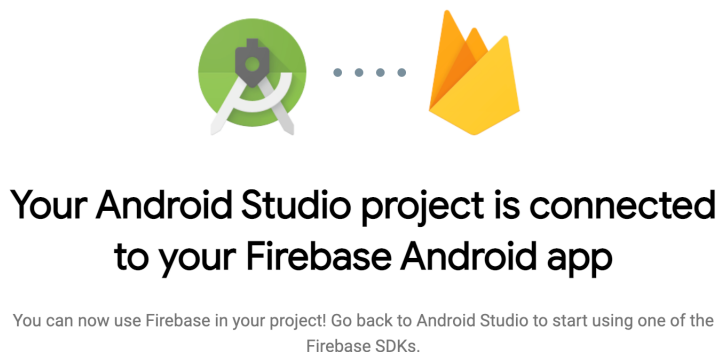


Рисунок 2.1 – Подключение модуля к «*Firebase*»

После выполнения первого шага в ассистенте должно состояние измениться на «*connected*» (рисунок 2.2), а в корне модуля появиться файл «*google-services.json*».

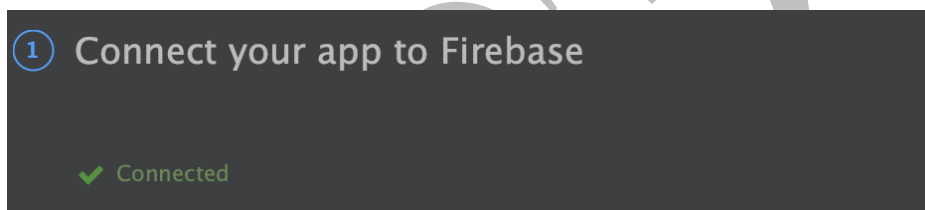


Рисунок 2.2 – Подключение модуля к «*Firebase*»

Далее требуется добавить зависимости в проект «Add the Firebase Authentication SDK to your app» (рисунок 2.3).

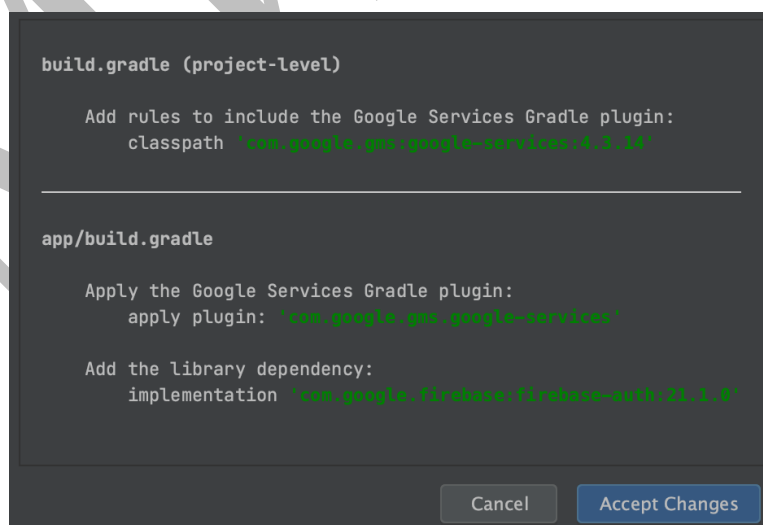


Рисунок 2.3 – Экран добавления требуемых зависимостей

Для использования других функциональных модулей требуется добавление

дополнительных библиотек. Сервис аутентификации тесно интегрируется с другими сервисами «*Firebase*», использует отраслевые стандарты, такие как «*OAuth 2.0*» и «*OpenID Connect*», так что значительно упрощает организацию взаимосвязи с собственной реализацией серверной части. Далее рассматривается метод аутентификации пользователей с помощью адреса электронной почты и пароля.

Требуется перейти в настройки созданного проекта в консоли разработчика и выбрать в меню «*Build/Authentication*» (рисунок 2.4).

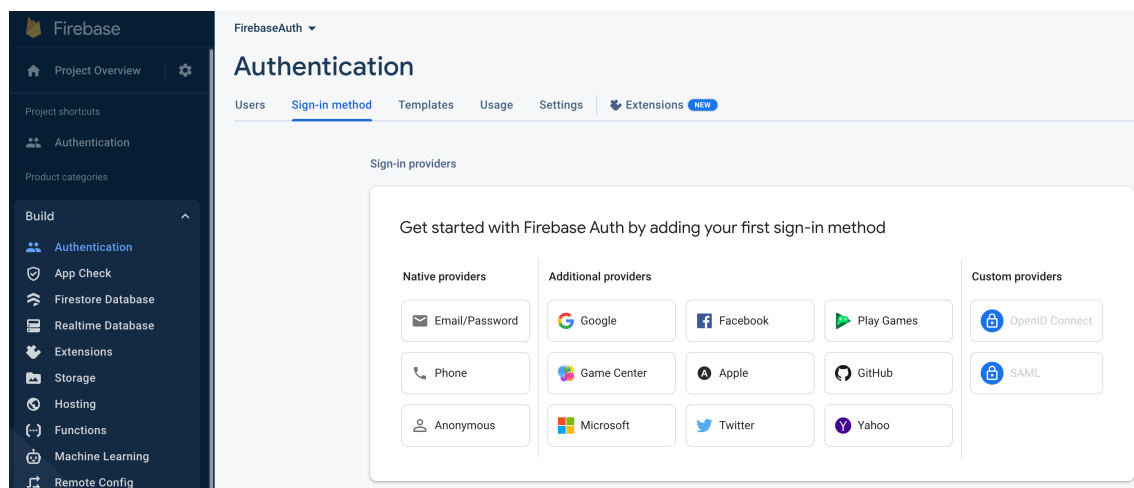


Рисунок 2.4 – Настройка способа аутентификации проекта «*Firebase*»
Далее требуется активировать авторизации по почте и паролю (рисунок 2.5).

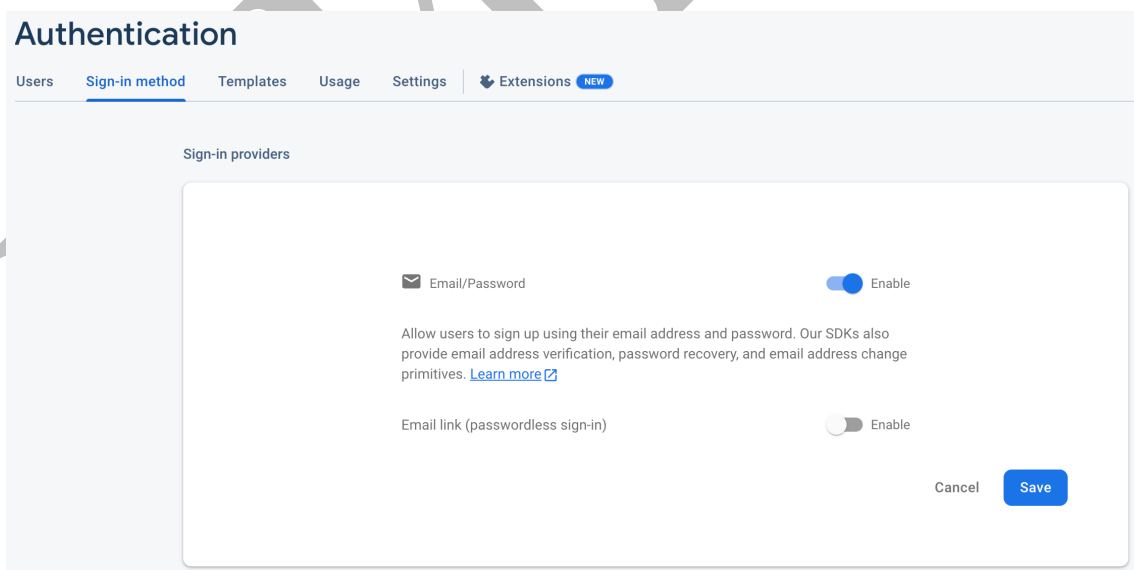


Рисунок 2.5 – Активация способа авторизации по почте и паролю в консоли разработчика «*Firebase*»

В процессе регистрации пользователей в приложении информация о них будет

появляться на вкладке «Пользователи». Здесь возможно управлять пользователями — например, добавить пользователя в базу приложения, а также отключить или удалить пользователя.

Требуется создать аналогичный экран аутентификации, представленный на рисунке 2.6, позволяющий производить аутентификацию пользователей и создание аккаунтов.

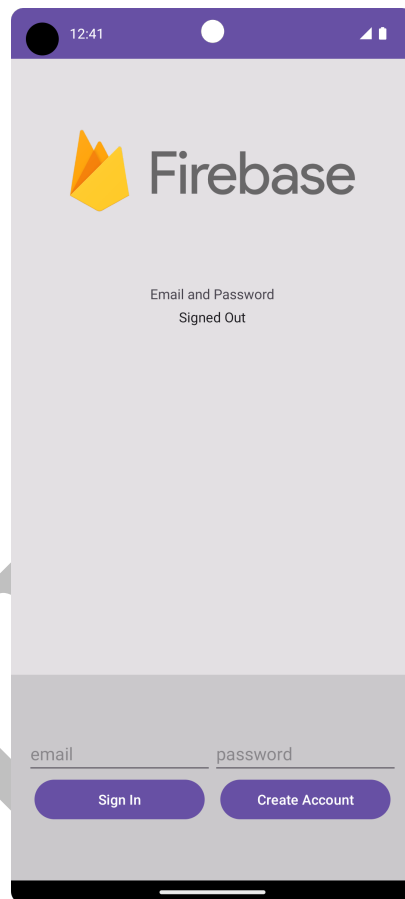


Рисунок 2.6 – Экран приложения «FirebaseAuth»

В файл `«res|values|strings.xml»` необходимо добавить значения строковых констант:

```

<resources>
    <string name="app_name"> FirebaseAuth </string>

    <string name="label_emailpassword">Email/Password Authentication</string>
    <string name="desc_emailpassword">Use an email and password to authenticate
with Firebase.</string>
    <string name="hint_user_id">User ID</string>
    <string name="sign_in">Sign In</string>
    <string name="create_account">Create Account</string>
    <string name="sign_out">Sign Out</string>
    <string name="verify_email">Verify Email</string>
    <string name="signed_in">Signed In</string>
    <string name="signed_out">Signed Out</string>
    <string name="auth_failed">Authentication failed</string>
    <string name="firebase_status_fmt">Firebase UID: %s</string>
    <string name="firebase_user_management">Firebase User Management</string>
    <string name="emailpassword_status_fmt">Email User: %1$s (verified:
%2$b)</string>
    <string name="emailpassword_title_text">Email and Password</string>
    <string name="error_sign_in_failed">Sign in failed, see logs for de-
tails.</string>
</resources>

```

В классе «*MainActivity*» требуется инициализировать элементы пользовательского интерфейса, а также объекта класса «*FirebaseAuth*» и его слушателя «*AuthStateListener*». Класс «*FirebaseAuth*» является точкой входа для взаимодействия с «*Firebase Authentication SDK*». Интерфейс «*FirebaseAuth.AuthStateListener*» вызывается, когда происходит изменение в состоянии аутентификации.

В методе «*onCreate*» требуется произвести инициализацию элементов экрана и установить слушатели нажатия на кнопки. В методах «*onStart*» проверяется состояние аутентификации пользователя и вызывается метод обновления экрана.


```

public class MainActivity extends AppCompatActivity{
    private static final String TAG = MainActivity.class.getSimpleName();
    private MainActivityBinding binding;
    // START declare_auth
    private FirebaseAuth mAuth;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Initialization views
        binding = ...
        setContentView(binding.getRoot());
        // [START initialize_auth] Initialize Firebase Auth
        mAuth = FirebaseAuth.getInstance();
        // [END initialize_auth]
    }
    // [START on_start_check_user]
    @Override
    public void onStart() {
        super.onStart();
        // Check if user is signed in (non-null) and update UI accordingly.
        FirebaseUser currentUser = mAuth.getCurrentUser();
        updateUI(currentUser);
    }
    // [END on_start_check_user]
    private void updateUI(FirebaseUser user) {
        if (user != null) {
            binding.statusTextView.setText(getString(R.string.emailpassword_status_fmt,
                                                    user.getEmail(), user.isEmailVerified()));
            binding.detailTextView.setText(getString(R.string.firebase_status_fmt, user.getUid()));
            binding.emailPasswordButtons.setVisibility(View.GONE);
            binding.emailPasswordFields.setVisibility(View.GONE);
            binding.signedInButtons.setVisibility(View.VISIBLE);
            binding.verifyEmailButton.setEnabled(!user.isEmailVerified());
        } else {
            binding.statusTextView.setText(R.string.signed_out);
            binding.detailTextView.setText(null);
            binding.emailPasswordButtons.setVisibility(View.VISIBLE);
            binding.emailPasswordFields.setVisibility(View.VISIBLE);
            binding.signedInButtons.setVisibility(View.GONE);
        }
    }
}

```

Экземпляр класса «*FirebaseUser*» предоставляет сведения о профиле пользователя в базе данных пользователей проекта «*Firebase*». Он также содержит вспомогательные методы для изменения или получения информации о профиле, а также для управления состоянием аутентификации пользователя. Экземпляр данного класса передаётся в метод «*updateUI*», принимающий экземпляр текущего пользователя. В этом методе происходит вывод информации о текущем пользователе в текстовые поля на экране приложения, а также регулируется видимость элементов пользовательского интерфейса в зависимости от того, авторизован пользователь или нет.

Основной функционал экрана заключается в создании учетной записи пользователя, верификации почтового ящика пользователя, авторизации и выхода из учетной записи.

Ниже представлен пример реализации создания аккаунта пользователя в методе «*createAccount*». В облачной базе данных проекта «Firebase» почтовый адрес является уникальным идентификатором, а также используется для отправки письма сброса пароля. Методом, отвечающим за создание учетной записи в классе «*FirebaseAuth*», является «*createUserWithEmailAndPassword*».

```
private void createAccount(String email, String password) {
    Log.d(TAG, "createAccount:" + email);
    if (!validateForm()) {
        return;
    }
    // [START create_user_with_email]
    mAuth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's
information
                    Log.d(TAG, "createUserWithEmail:success");
                    FirebaseUser user = mAuth.getCurrentUser();
                    updateUI(user);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "createUserWithEmail:failure",
task.getException());
                    Toast.makeText(MainActivity.this, "Authentication
failed.",
                                Toast.LENGTH_SHORT).show();
                    updateUI(null);
                }
            }
        });
    // [END create_user_with_email]
}
```

В случае успешного создания учетной записи пользователя, будет автоматически выполнен вход данного пользователя в приложении. Однако, создание учетной записи может завершиться ошибкой в случаях, когда учетная запись существует, почтовый адрес является некорректным или пароль недостаточно сложным. Список возможных исключений:

- «*FirebaseAuthWeakPasswordException*» – пароль не является достаточно сложным;
- «*FirebaseAuthInvalidCredentialsException*» – почтовый адрес имеет

неправильный формат;

- «*FirebaseAuthUserCollisionException*» – существует учетная запись с таким почтовым адресом.

После создания учётной записи возможно пройти аутентификацию. Метод «*signIn*» в качестве аргументов принимает почтовый адрес и пароль и выполняет попытку авторизации пользователя с помощью метода «*signInWithEmailAndPassword*».

```
private void signIn(String email, String password) {
    Log.d(TAG, "signIn:" + email);
    // [START sign_in_with_email]
    mAuth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                // Sign in success, update UI with the signed-in user's
information
                Log.d(TAG, "signInWithEmail:success");
                FirebaseUser user = mAuth.getCurrentUser();
                updateUI(user);
            } else {
                // If sign in fails, display a message to the user.
                Log.w(TAG, "signInWithEmail:failure", task.getExcep-
tion());
                Toast.makeText(MainActivity.this, "Authentication
failed.",
                    Toast.LENGTH_SHORT).show();
                updateUI(null);
            }

            // [START_EXCLUDE]
            if (!task.isSuccessful()) {
                binding.statusTextView.setText(R.string.auth_failed);
            }
            // [END_EXCLUDE]
        }
    });
    // [END sign_in_with_email]
}
```

При авторизации пользователем возможны следующие исключения:

- «*FirebaseAuthInvalidUserException*» – учетная запись пользователя с таким почтовым адресом не существует или отключена;
- «*FirebaseAuthInvalidCredentialsException*» – введен неправильный пароль.

Пользователь может осуществить выход из своей учетной записи в приложении с помощью вызова метода «*signOut*» в классе «*FirebaseAuth*».

```
private void signOut() {
    mAuth.signOut();
    updateUI(null);
}
```

Последней функциональной возможностью экрана является выполнение верификации почтового ящика пользователя с помощью метода «*sendEmailVerification*» в классе «*FirebaseAuth*».

```
private void sendEmailVerification() {
    // Disable button
    binding.verifyEmailButton.setEnabled(false);

    // Send verification email
    // [START send_email_verification]
    final FirebaseUser user = mAuth.getCurrentUser();
    Objects.requireNonNull(user).sendEmailVerification()
        .addOnCompleteListener(this, new OnCompleteListener() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                // [START_EXCLUDE]
                // Re-enable button
                binding.verifyEmailButton.setEnabled(true);

                if (task.isSuccessful()) {
                    Toast.makeText(MainActivity.this,
                        "Verification email sent to " + user.getEmail(),
                        Toast.LENGTH_SHORT).show();
                } else {
                    Log.e(TAG, "sendEmailVerification", task.getException());

                    Toast.makeText(MainActivity.this,
                        "Failed to send verification email.",
                        Toast.LENGTH_SHORT).show();
                }
                // [END_EXCLUDE]
            }
        });
    // [END send_email_verification]
}
```

Для **ВЫПОЛНЕНИЯ ЗАДАНИЯ** требуется реализовать изменение экрана в зависимости от состояния авторизации пользователя в соответствии с рисунком 2.6, а также осуществить вызов методов регистрации, верификации почтового ящика, авторизации и выход из учетной записи по действиям пользователя.

Запустить проект и создать пользователя. Необходимо сделать скриншот зарегистрированного пользователя и консоли разработчика с отображением идентификатора ящика (как на рисунке 2.6). Создать директорию «raw» по адресу «app|src|main|res» и разместить полученный материал.

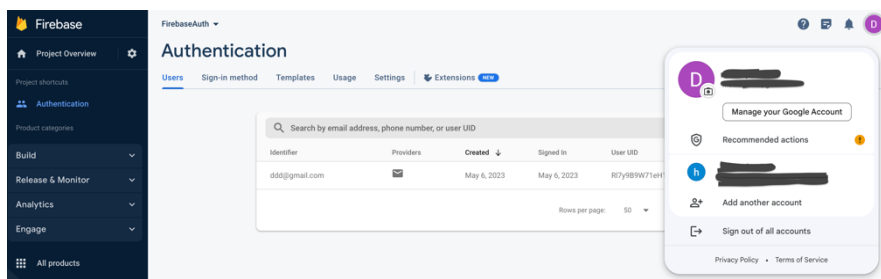
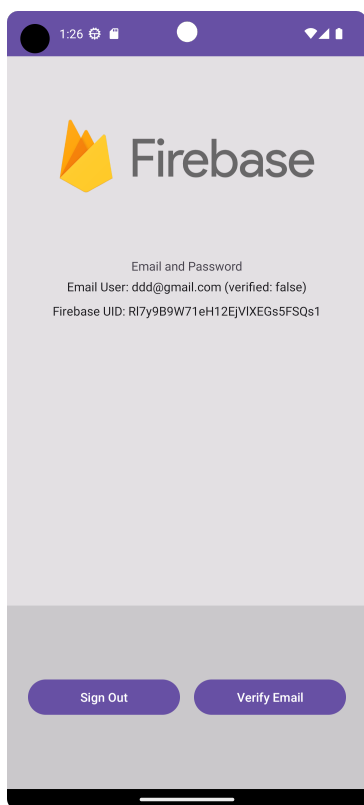


Рисунок 2.7 – Экран приложения «*FirebaseAuth*» и консоли разработчика с авторизованным пользователем

3 КОНТРОЛЬНОЕ ЗАДАНИЕ

В проекте «*MireaProject*». Добавить экран входа в приложение с помощью «*Firebase*» любым способом.

Наиболее простым способом решения является создание нового «*activity*» и изменении записи точки входа в `manifest`-файле.

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />

  <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

После успешной авторизации произвести переход на главный экран. Добавить фрагмент, отображающий любую информацию из сетевого ресурса. Возможно использование различных способов сетевого взаимодействия (напр. «*Retrofit*»).