6.2. Передача данных по сети. Часть 1

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 6.2. Передача данных по сети. Часть 1

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 19:15

Оглавление

1. 6.2.1 Общие понятия
2. 6.2.2 Клиентское приложение
3. 6.2.3 Сериализация и JSON

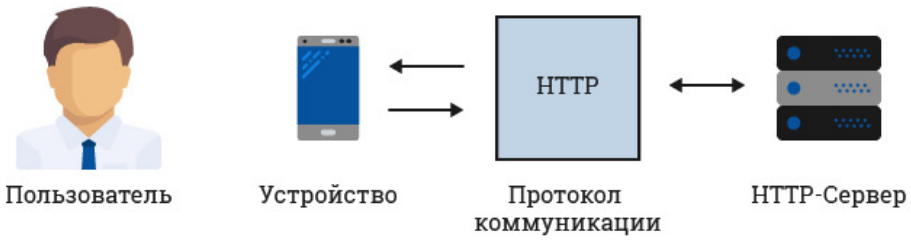
1. 6.2.1 Общие понятия

Архитектура клиент-сервер предполагает наличие трех компонент:

- клиентской части (пользователь);
- серверной части (удаленный сервис);
- среды коммуникации.

В мобильных приложениях клиентская часть располагается на мобильном устройстве и, как правило, предназначена для взаимодействия с пользователем устройства. Серверная часть располагается либо на отдельном компьютере, либо на другом мобильном устройстве. Сервер предоставляет клиенту необходимый сервис (функциональность) и напрямую с пользователем не взаимодействует. Для связи между клиентской и серверной частью используется коммуникационная среда, в случае мобильных приложений, чаще всего, обеспечиваемая протоколами интернет.

Одним из популярных интернет-протоколов является HTTP/HTTPS. Если серверная часть приложения базируется на системе обработки HTTP, то говорят о мобильном HTTP-приложении. В этом случае все коммуникации между клиентом и сервером организуются с использованием стандартных HTTP-запросов и ответов.



Запросы от мобильного клиента к серверу могут быть как синхронными, так и асинхронными. При использовании синхронных сообщений работа приложения блокируется до тех пор, пока не получен ответ от сервера. Если используются асинхронные сообщения, то клиент продолжает работу в штатном режиме, не дожидаясь ответа, а при получении ответа от сервера, происходит вызов функцию колбека (слушателя).

Как было рассмотрено выше клиент серверное взаимодействие осуществляется между клиентом и сервером по протоколу HTTP. В качестве клиента HTTP может выступать:

- браузер - при подключении по URL указывающей конкретный ресурс (URI) на сервере либо веб форма с указанием Action на URI
- другое программное обеспечение, как мобильное так и приложение на ПК.

В качестве HTTP севера обычно выступает специальное серверное ПО, которое устанавливается на компьютер и работает в постоянном режиме принимая на сетевой интерфейс компьютера подключения и отвечая на HTTP запросы. Существует большое количество классических HTTP (веб) серверов. Например:

- Apache web server,
- Nginx,
- Jetty,
- Microsoft internet information server.
- и множество других.

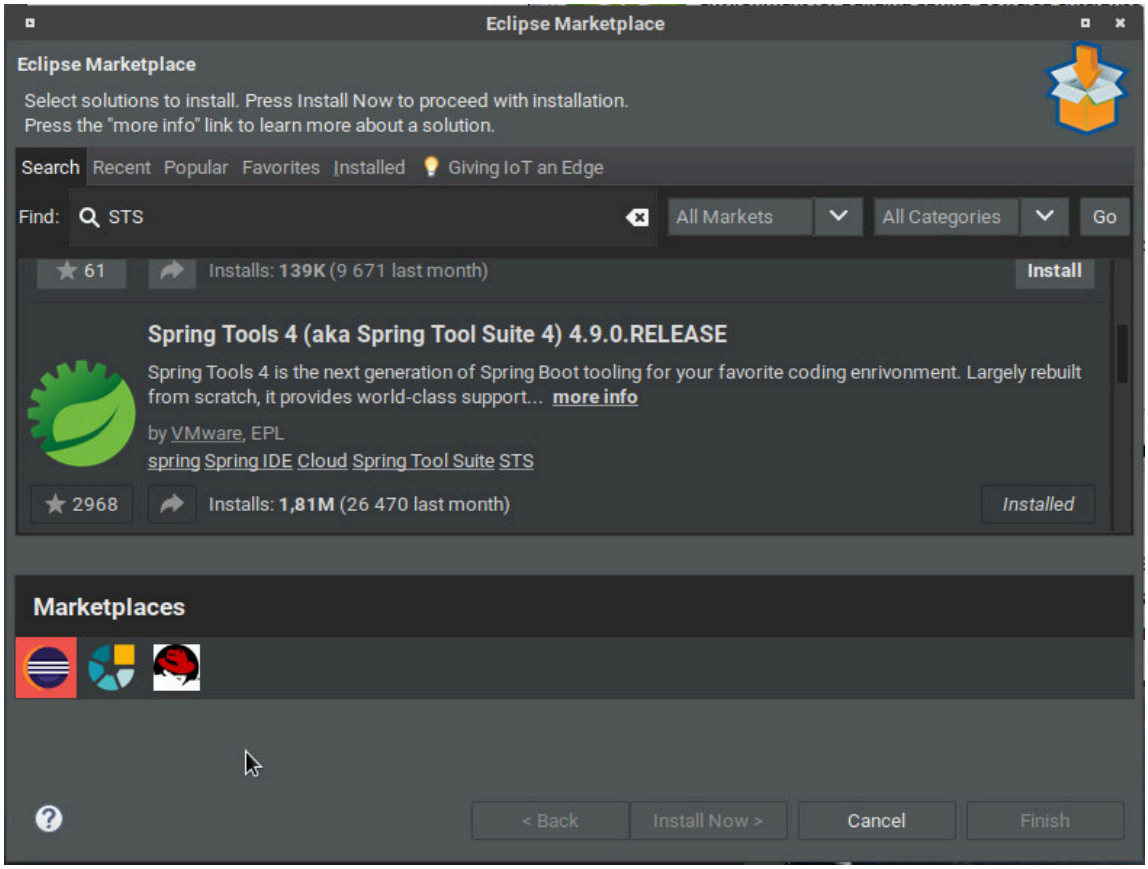
Однако классический веб сервер, который выдает в теле HTTP ответа веб страницу или картинку, которая статически хранится как обычный файл на жестком диске сервера и сопоставлена с конкретным URI не особо интересен в рассматриваемом варианте клиент-серверного взаимодействия. Гораздо интереснее в этом смысле сервера, которые помимо статического контента могут при запросе некоторых URI запускать на сервере программный код, а его результаты возвращать как HTTP ответы. В этой и последующих лекциях будут рассматриваться сервера с возможностью выполнения java кода. Такие сервера называют веб-контейнерами. Это например:

- Apache Tomcat,
- GlassFish,
- WildFly,
- Oracle JBoss,
- IBM WebSphere.

Для выполнения примеров приведенного в лекции Java кода на стороне сервера, использовался Apache Tomcat встроенный в плагин STS 4 (Spring Tools Suite). Однако этот серверный код унифицирован и может выполнятся на многих других серверах контейнерах. Самый примитивный вариант написание серверного функционала - это использовать Java Servlet API (стандарт JSR 369). При его использовании достаточно написать класс сервлет, в котором реализовать функцию doGet или doPost для обработки GET и POST запросов. После создания и компиляции класса его в виде WAR архива копируют в определенную папку на сервере, после чего этот сервлет начинает отвечать на соответствующие HTTP запрос

```
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // программный код обработки GET запроса
        super.doGet(req, resp);
    }
}
```

Однако сегодня мало кто использует Sevlet API, т.к. оно слишком низкоуровневое и негибкое. Более популярны фреймворки надстройки над сервлетами, например Spring, который и будет в дальнейшем использоваться в этом учебнике. Для установки плагина STS (Spring Tools Suite) используемого для разработки серверной части кода в Eclipse нужно кликнуть Help -> Eclipse Marketplace. Далее необходимо набрать в строке поиска STS, далее нажать Install на найденном плагине STS4



Рассмотрим простейший пример клиент-серверного взаимодействия. В качестве клиента будет использоваться следующая вебформа:

```
<html>
<form name='test' method='post' action='http://localhost:8080/hello'>
  <p><b>Фамилия:</b><br>
    <input type='text' name='lastname' size='40'>
  </p>
  <p><b>Имя:</b><br>
    <input type='text' name='firstname' size='40'>
  </p>
  <p>
    <input type='submit' value='Отправить'>
  </p>
</form>
</html>
```

Вышеуказанный HTML документ необходимо открыть в браузере. Открывшаяся форма и будет примитивным клиентом, которая по нажатию кнопки осуществит HTTP GET запрос к северу. Для разработки серверной части, в среде разработки Eclipse с установленным плагином в меню выбрать New -> other Spring Boot-> Spring Starter Project, далее в открывшемся мастере Next и на второй закладке поставить галочку web -> Spring Web. При этом будет создан проект с уже готовой необходимой для запуска структурой. В том числе там будет готовый класс для запуска приложения в контейнере, например - Demo2Application. Единственное что нужно сделать - это сделать класс с подобного вида.

Рассматриваемые далее примеры входят в практическую работу, исходный код которой можно скачать по ссылке

```
@RestController
public class FirstTest {

    @RequestMapping("/hello")
    public String hello(@RequestParam String firstname,@RequestParam String lastname) {
        String rez= "Здравствуйте " + firstname.substring(0,1)+"."+lastname +"!";
        return rez;
    }

}
```

При этом все настройки веб сервера, например привязки URI к методу или параметры запроса к переменным осуществляются при помощи аннотаций. После созданий класса, нужно запустить проект Run-> Run As -> Spring Boot App. При этом плагин STS4 запустит Apache Tomcat с запущенным указанным классом и настроенной реакцией на указанный URI. Т.о. зпри заполненных полях веб формы и нажатии кнопки, клиент отправит данные через HTTP GET запрос на сервер, там на этот URI будет запущена функция hello с переданными параметрами GET запроса, в качестве параметров функции. Функция обработает полученную информацию и вернет строковое значение в теле HTTP ответа. Что и демонстрирует полный цикл клиент серверного взаимодействия через HTTP протокол.

2. 6.2.2 Клиентское приложение

Чтобы отправлять запросы через коммуникационную среду интернет, Android-приложению должен быть разрешен доступ в интернет. Для этого необходимо добавить в манифест строку:

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

Для написания приложения для обмена по HTTP под Android до 22й версии API использовался стандартный клиент Apache HTTP Начиная с версии 23 (Marshmallow) эта библиотека была полностью удалена. На устройствах с большими версиями API обычно используют либо встроенный в Java 8 клиент HttpUrlConnection/HttpsUrlConnection либо внешний библиотеки - например OkHTTP или более сложный фреймворки,такие как Retrofit.

> Следует отметить, что в последнее время большинство производителей ПО считает HTTP небезопасным и рекомендует разработчикам переходить на использование HTTPS. Однако в рассматриваемых примерах все же будет использоваться HTTP, так как создание сертификата для сервера HTTPS это довольно нетривиальная задача. В Android приложении, для того чтобы использовать протокол HTTP, нужно в элемент манифеста application добавить атрибут android:usesCleartextTraffic="true"

Рассмотрим пример создания приложения для клиент - серверного приложения приведенного выше. Для этого разработаем макет со следующими виджетами

- поле ввода URL подключения (R.id.url)
- поле ввода фамилия (R.id.firstname)
- поле ввода имя(R.id.lastname)
- кнопка отправки запроса на сервер (R.id.send)
- текстовое поле для вывода ответа от сервера (R.id.result)

Тогда код активности, осуществляющей обмен с сервером будет следующим:

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val et_url: EditText = findViewById(R.id.url)
        val firstname: EditText = findViewById(R.id.firstname)
        val lastname: EditText = findViewById(R.id.lastnamet)
        val send: Button = findViewById(R.id.send)
        val result: TextView =findViewById(R.id.result)
        send.setOnClickListener(View.OnClickListener {
            val thread:Thread = object:Thread(){
                override fun run() {
                    val url: URL = URL(et_url.text.toString()+"?firstname="+
                            firstname.text+"&lastname="+lastname.text)
                    val urlConnection:HttpURLConnection = url.openConnection() as HttpURLConnection
                    try {
                        val data  = urlConnection.inputStream.bufferedReader().readLine()
                        result.text = data
                    } finally {
                        urlConnection.disconnect()
                    }
                }
            }
            thread.start()
        })
```

> Начиная с API 13 (Android 3.0 Honeycomb) нельзя работать с сетью в основном потоке приложения, что в общем то правильно так как сетевые операции обычно медленные и могут "фризить" пользовательский интерфейс. В связи с чем, в приведенном примере работа с сетью ведется в отдельном потоке на основе класса Thread.

В приведенном примере передача параметров GET запроса осуществляется как часть URL после символа "?". Однако HttpURLConnection может выполнять и POST запросы с передачей параметров в теле запроса.

Для получения сущности URLConnection нужно использовать объект класса **java.net.URL**, его конструктор принимает параметр тип String где помимо всего должен быть указан протокол – в рассматриваемом случае http. После получения объекта URL, необходимо вызывать метод **openConnection()** который возвратит объект типа **HttpUrlConnection**. Если же используется подключение по протокол https, то необходимо перейти на использование типа **HttpsUrlConnection** .

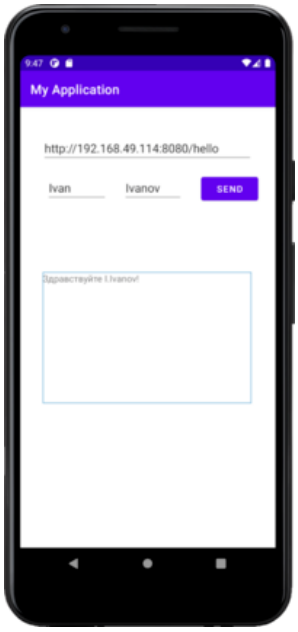
После этого переменная urlConnection будет хранить ссылку на объект HttpUrlConnection. По умолчанию будет формироваться GET-запрос, для того чтобы добавить заголовки, можно воспользоваться методм setRequestProperty(), который принимает key и value. Если используется POST запрос, то можно получить **writer** для формирования тела запроса - например:

```
String body="param=value"
OutputStreamWriter writer = new OutputStreamWriter(urlConnection.getOutputStream());
writer.write(body);
writer.close();
```

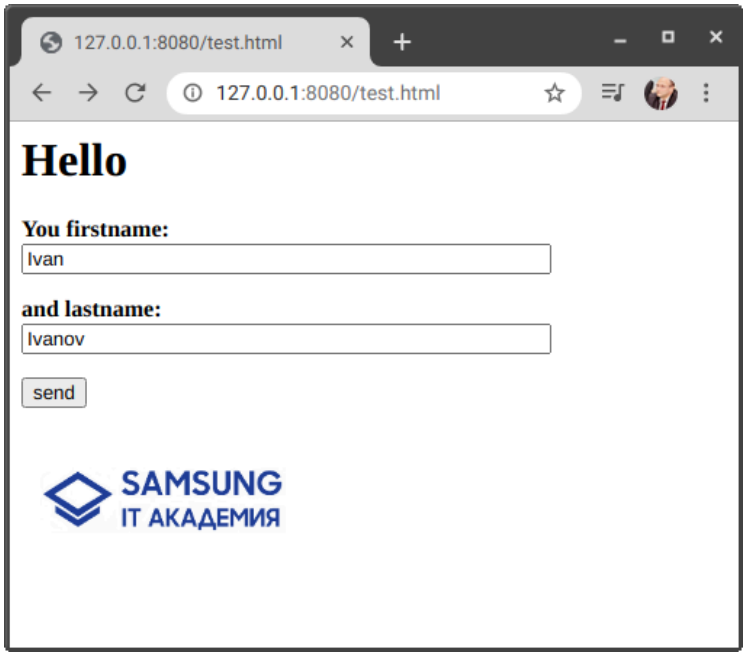
После отправки запроса, тело ответа может быть считано через входной поток, например так:

```
val data  = urlConnection.inputStream.bufferedReader().readLine()
```

После компиляции и запуска приложения при запущенной серверной части, можно наблюдать примерно следующее клиент-серверное взаимодействие.



Что интересно, в серверной части рассматриваемого клиент-серверного приложения, в папку **/src/main/resources/static** можно скопировать любой статический веб контент, такой например как изображения, файлы HTML и т.д. В рассматриваемом примере туда была скопирована страница с веб-формой, которая также как и Android приложение может взаимодействовать с сервером.



Таким образом, в рассмотренный примере представлено целых два клиента. Т.е. по сути представлено кросс-платофрменное клиент-серверное приложение.

3. 6.2.3 Сериализация и JSON

Сериализация

Для передачи структуры данных по коммуникационным протоколам необходимо перед передачей перевести структуру в последовательность битов. Такое преобразование называется сериализацией. Для того чтобы принимающая сторона могла эффективно обработать эти данные, существует обратный процесс, называемый десериализацией. В процессе десериализации происходит восстановление структуры данных в первоначальный вид.

Распространенным видом сериализации является перевод объектов программы в файл. В этом случае объект заполняется нужными данными, потом вызывается функция сериализации, которая преобразует его в файл некоторого формата. Файл передается через коммуникационную среду. Принимающая сторона отправляет файл в функцию десериализации. После обработки получатель располагает исходным объектом, заполненным нужными данными.

Любой из схем сериализации присуще то, что кодирование данных происходит последовательно по определению, и поэтому необходимо считать весь файл и только потом воссоздать исходную структуру данных.

Формат JSON

В качестве формата файла сериализации часто используют XML. Однако возможно использование и других форматов. В частности, при разработке мобильных приложений очень популярен JSON (JavaScript Object Notation) — специальный текстовый формат обмена данными, основанный на JavaScript. Несмотря на то что синтаксис описания данных напоминает JS, формат является независимым от языка и может использоваться практически с любым языком программирования.

За счет лаконичности синтаксиса по сравнению с XML, формат JSON является более подходящим для сериализации сложных структур. Далее представлен пример сериализованного объекта «Пользователь» в обоих форматах.

JSON

```
{
    "firstname": "Александр",
    "lastname": "Викторов",
    "from": {
        "region": "Нижегородская область",
        "town": "Дзержинск",
        "school": 17
    },
    "phone": [
        "312 123-1234",
        "312 123-4567"
    ]
}
```

XML

```
<user>
    <firstname>Александр</firstname>
    <lastname>Викторов</lastname>
    <from>
            <region> Нижегородская область</region>
            <town>Дзержинск</town>
            <school>17</school>
    </from>
    <phone>
        <phonenumber>312 123-1234</phonenumber>
        <phonenumber>312 123-4567</phonenumber>
    </phone>
</user>
```

В JSON-формате предусмотрено использование двух структур:

- набор пар «ключ — значение». Ключом может быть только строка, значением — любая запись. В разных языках программирования ей соответствуют объект, запись, структура, словарь, хеш, именованный список или ассоциативный массив;
- упорядоченный набор значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.

Названные структуры данных удобны: большая часть алгоритмических языков программирования высокого уровня поддерживает их в той или иной форме. Поэтому даже если клиентская и серверная части приложения написаны на разных языках, организовать их взаимодействие не представляет труда.

В качестве значений в JSON-формате возможно использование следующих записей:

- объект — это неупорядоченное множество пар «ключ — значение», заключенное в фигурные скобки { }. Ключ и значение разделяется символом «:». Пары «ключ — значение» разделяются запятыми;
- одномерный массив — это упорядоченное множество значений. Массив заключается в квадратные скобки [ ]. Значения ячеек массива разделяются запятыми;
- простое значение может быть строкой в двойных кавычках, числом или одним из литералов: true, false, null.

Записи могут быть вложены друг в друга. В приведенном ранее примере по ключу from находится объект, состоящий из трех пар «ключ — значение».

```
"from": {
    "region": "Нижегородская область",
    "town": "Дзержинск",
    "school": 17
}
```

Одномерный массив строк задан ключом «phone».

```
"phone": [
    "312 123-1234",
    "312 123-4567"
]
```

Строка в JSON — это строка, состоящая из символов юникода, заключенного в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\», или записаны в кодировке UTF-8. Для записи чисел используется только десятичный формат. Пробелы могут быть вставлены между любыми двумя синтаксическими элементами. Помимо описанного среди основных отличий от XML:

- JSON не поддерживает многострочные тексты — однако можно в рамках одной строки использовать escape-последовательность «\n» вместо переводов строк. Например, { «greetiing» : «Поздравляю с новым годом!\nЖелаю счастья!» };
- Если текст в JSON содержит спецсимвол, то его необходимо вручную экранировать символом escape-последовательности: { «title» : «\«Rock&roll\" = life» };
- В XML нет синтаксического представления массивов и списков, тогда как в JSON есть. Это, конечно, не означает, что в XML нельзя сериализовать список — конечно, можно. В этом случае програмиист просто имеет в виду, что группа узлов с одинаковым названием и есть элементы списка.

Сериализация с помощью класса Gson

> Как и рассмотренный выше пример реализации простейшего клиент-серверного взаимодействия, исходный код примера работы с JSON можно скачать по ссылке

Сериализация в JSON возможна разными способами. Рассмотрим один из простейших способов — с помощью класса Gson из пакета com.google.gson. Для использования библиотеки, ее нужно добавить в проект в файле **build.gradle(app)** в раздел **dependencies**:

```
 implementation 'com.google.code.gson:gson:2.8.5'
```

Предположим, мы хотим сериализовать объект класса User.

```
class User(var firstname: String,
           var lastname: String,
           var school: Int){
}
```

Следующий код сериализует объект класса User в JSON (код для Android).

```
    var user:User = User("Иван","Иванов",7)
    Log.d("Gson test",Gson().toJson(user))
```

После исполнения программы в логах появится строка:

```
{"firstname":"Иван","lastname":"Иванов","school":7}
```

Чтобы производить десериализацию в Gson есть другая функция. Предположим, из JSON-строки jsonText необходимо построить объект для работы в приложении. Тогда код будет выглядеть так:

```
// Строка JSON: {"firstname":"Петр","lastname":"Петров","school":1}"
 val jsonText = "{\"firstname\":\"Петр\",\"lastname\":\"Петров\",\"school\":1}"
 user = Gson().fromJson(jsonText,User::class.java)
 Log.i("Gson test",user.firstname + " " +user.lastname + " " + user.school)
```

После выполнения кода в логах появится строка со значениями из структуры данных User.

Таким образом происходит сериализация/десериализация Java-объектов в JSON-строки. Примерно аналогичным образом выполняется и сериализация/десериализация Java-объектов и в XML-строки.

> Однако рассмотренный способ сериализации — это лишь частный случай более общего процесса сериализации. Следует помнить, что указанные библиотеки — это надстройки над механизмом сериализации в Java. В общем случае сериализация в Java — это возможность представления любого объекта в виде простой последовательности байт («серии»). В этом случае появляется возможность передачи объектов через любые потоки. Больше можно прочесть здесь:
>
> - https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
> - http://info.javarush.ru/translation/2013/09/03/Как-работает-сериализация-в-Java.html
> - http://www.javable.com/tutorials/fesunov/lesson17/

Передача данных по протоколу HTTP на сервер и обратно предполагает текстовый формат. В рассмотренном в первой части этой лекции примере информация, по совпадению, как раз и была строковой. Однако если бы она была иной - например числовой или какие то более сложные структуры данных: как то массивы, списки, объекты собственных классов и агрегации подобных типов данных, то пришлось бы совершить довольно много работы на передающей стороне по переводу этой информации в текстовый вид, а на принимающей стороне обратно из строки в объекты. Исходя из этого становится понятно, что сериализация дает возможность быстро решить проблему конвертации данных из/в строку при передаче через протокол HTTP. Однако на практике библиотеки подобные Gson используют довольно часто, для очень большого класса задач требующих сериализации, например:

- сохранить сложный объект, например коллекцию каких либо объектов в/из файл,
- сохранить объект(ы) в SharedPreferences,
- передать зашифрованный объект как сериализованный байт массив в виде письма,
- и многое другое