Модуль 5. Основы разработки серверной части мобильных приложений

Тема 5.3. Клиент-серверная архитектура мобильных приложений

4 часа

Оглавление

5.3. Клиент-серверная архитектура мобильных приложений 2
5.3.1. Общие понятия 2
5.3.2. Отправка запросов из Android приложений 3
Упражнение 5.3.1 4
5.3.3. Форматы JSON и XML. Сериализация 7
Формат JSON 8
Сериализация с помощью класса Gson 10
Здание 5.3.1 10
5.3.4. Библиотека Retrofit 11
Упражнение 5.3.2 13
Задание 5.3.2....16
Благодарности 16

5.3. Клиент-серверная архитектура мобильных приложений

5.3.1. Общие понятия

Архитектура клиент-сервер предполагает наличие трех компонент:

- клиентской части (пользователь);
- серверной части (удаленный сервис);
- среда 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 некий сервис 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 Hyper Text Transmission Protocol (HTTP). 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, рассмотренных в главе 5.2.

Рисунок 5.3.1

Запросы от мобильного клиента к серверу могут быть как синхронными, так и асинхронными. Понятия синхронных и асинхронных сообщений по своей сути сходны с понятиями синхронных и асинхронных процессов, рассмотренных в модуле 3. 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.

5.3.2. Отправка запросов из Android приложений

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 HTTP-приложений под Android существует ряд классов, которые позволяют быстро реализовывать базовую функциональность. Они содержатся в пакете org.apache.http и его подпакетах. Интерфейс клиента называется HttpClient. Экземпляр класса с этим интерфейсом легче всего создать, используя класс DefaultHttpClient. В случае вызова new DefaultHttpClient() создается стандартный клиент.

```
HttpClient httpclient = new DefaultHttpClient();
```

Чтобы отправлять и получать данные клиент использует объекты, имплементирующие интерфейс HttpUriRequest. Для Android разработаны классы, приспособленные для частных видов запросов. Например, для отправки данных на сервер используется HttpPost, для получения данных с сервера – HttpGet.

Объект HttpPost можно создать следующим образом:

```
HttpPost httppost = new HttpPost("http://192.168.72.3:8080");
```

В конструктор передается URI сервера (в виде объекта класса URI или класса String). Это может быть как IP адрес, с указанием порта или без него, так и символьное доменное имя.

Существуют и другие виды HTTP запросов, для которых разработаны специальные классы – HttpDelete, HttpPut, HttpOptions и т.д. Их особенности и варианты использования описаны в документации к пакету org.apache.http.client.methods.

Выполнение запроса осуществляется с помощью метода execute() интерфейса HttpClient. Существуют различные имплементации метода, с различными входными параметрами. Если в качестве аргумента используется объект одного из классов HTTP запросов.

Метод может бросать исключения IOException и ClientProtocolException. Первое вырабатывается в случае, если возникли проблемы с соединением, второе – если вернулась ошибка HTTP протокола.

Метод возвращает ответ сервера в виде объекта, имплементирующего интерфейс HttpResponse. Поэтому конструкция по отсылке запроса на сервер и получения ответа должна быть такой:

```
HttpResponse response = httpclient.execute(httppost);
```

Для извлечения полезной информации из объекта, имплементирующего HttpResponse, используются интерфейс HttpEntity и класс EntityUtils. Первый из них описывает блок пользовательских данных, содержащийся в HTTP сообщении. Используя метод интерфейса HttpResponse getEntity(), можно получить эту часть сообщения.

```
HttpEntity entity = response.getEntity();
```

Класс EntityUtils – это небольшая библиотека статических методов для обработки блоков пользовательских данных. Там есть функции преобразования данных в строки и массив байтов. Вызов метода из этого класса таков:

```
String answerHTTP =EntityUtils.toString(entity);
```

Рассмотрим использование названных интерфейсов и классов на практическом примере.

Упражнение 5.3.1

Разработаем программу для Android, которая реализует задание аналогичное Упражнению 5.2.2.

При реализации асинхронного клиент-серверного приложения важно все коммуникационные операции инкапсулировать в класс, расширяющий класс AsyncTask, как это было показано в модуле 3. В упражнении 5.3.1 достаточно переопределить методы DoInBackground () и onPostExecute(), но для решения других задач можно использовать весь арсенал средств, предоставляемых AsyncTask.

Итак, имплементация клиентской части требуемого приложения будет выглядеть следующим образом.

```
package com.samsung.itschool.app5_3_1;

import android.os.AsyncTask;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;


import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.NameValuePair;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.util.EntityUtils;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class MainActivity extends AppCompatActivity {
  TextView lastnameF;
  String answerHTTP;
  String lastnameS, firstnameS;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
```

```
    lastnameF = (TextView) findViewById(R.id.lastnameF);
  }

   public void sendPOST(View view) {
    EditText lastname = (EditText) findViewById(R.id.lastname);
    EditText firstname = (EditText) findViewById(R.id.firstname);
    lastnameS = lastname.getText().toString();
    firstnameS = firstname.getText().toString();
    new MyAsyncTask().execute("");
  }

class MyAsyncTask extends AsyncTask<String, String, String> {

   @Override
   protected String doInBackground(String... params) {

     // Create a new HttpClient and Post Header
     HttpClient httpclient = new DefaultHttpClient();
     HttpPost httppost = new HttpPost("http://192.168.72.3:8080");

     try {

         List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(2);

         nameValuePairs.add(new BasicNameValuePair("lastname", lastnameS));
         nameValuePairs.add(new BasicNameValuePair("firstname", firstnameS));
         httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs, "UTF-8"));

         // Execute HTTP Post Request
         HttpResponse response = httpclient.execute(httppost);
         if (response.getStatusLine().getStatusCode() == 200) {
           HttpEntity entity = response.getEntity();
           answerHTTP = EntityUtils.toString(entity);
         }

     } catch (ClientProtocolException e) {
         // TODO Auto-generated catch block
     } catch (IOException e) {
         // TODO Auto-generated catch block
     }
     return null;
   }

   @Override
   protected void onPostExecute(String result) {
        super.onPostExecute(result);
        lastnameF.setText(answerHTTP);
   }
 }
}
```

Пользовательский интерфейс должен включать:

- два поля текстового редактирования lastname, firstname
- кнопку “Отправить” с обработчиком под именем sendPOST.

Для корректной работы серверная часть должна быть несколько иной, чем в Упражнении 5.2.2.

Класс HttpControllerREST изменится:

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HttpControllerREST extends HttpServlet {

  @RequestMapping("/")
  public String index(HttpServletRequest request, HttpServletResponse response) {
        if (request.getParameter("lastname") != null ||
request.getParameter("firstname") != null)
        if (!request.getParameter("lastname").equals("")
        && !request.getParameter("firstname").equals("")) {
        String lastname = request.getParameter("lastname");
        String firstname = request.getParameter("firstname");
        firstname = firstname.substring(0, 1);
        return lastname + " " + firstname + ".";
        } else
        return "No POST data lastname or firstname";

        return "<h1>Greetings from Spring Boot!</h1>"
        + "<form name='test' method='post' action=''>"
        + "<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>";
  }

}
```

Обратите внимание, что посылаемая через интернет пользовательская информация разделена на пары ключ-значение. Ключи в упражнении - firstname и lastname. Они одинаковы как для клиентской части приложения (хранится в nameValuePairs), так и для серверной части. По этим ключам происходит присвоение и извлечение соответствующих значений, которые в дальнейшем используются в программе.

Класс Application останется прежним:

```
import java.util.Arrays;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class Application {
```

```
        public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class,
args);

        System.out.println("Let's inspect the beans provided by Spring
Boot:");
        // Логирование последовательности загрузки сервера и библиотек
         String[] beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
                System.out.println(beanName);
        }
    }
}
```

* Аналогичная реализациея сервера на PHP приведена ниже:

```
<?php
if (isset($_POST["lastname"]) || isset($_POST["firstname"]))
    if ($_POST["lastname"] != "" && $_POST["firstname"] != "") {
        $lastname = $_POST["lastname"];
        $firstname = $_POST["firstname"];
        $firstname = iconv('UTF-8', 'windows-1251', $firstname); // перевод
кодировки нужен для того, чтобы корректно русские буквы отображались
        $firstname = substr($firstname, 0, 1);
        $firstname = iconv('windows-1251', 'UTF-8', $firstname); // перевод
кодировки нужен для того, чтобы корректно русские буквы отображались
        echo $lastname . " " . $firstname . ".";
    } else
        echo "No POST data lastname or firstname";
else
    echo "
<form name='test' method='post' action=''>
<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>";
?>
```

5.3.3. Форматы JSON и XML. 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 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 клиент-сервер очень популярен **JSON** (JavaScript Object Notation) — специальный текстовый формат обмена данными, основанный на JavaScript. Несмотря на происхождение, формат является независимым от языка и может использоваться практически с любым языком программирования.

За счёт лаконичности синтаксиса по сравнению с XML, формат JSON является более подходящим для сериализации сложных структур. Далее представлен пример сериализованного объекта Пользователь в том и другом 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>
  </address>
  <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, хэш, 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, заключённое в фигурные скобки «{ }». Между ключом и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми.
- Одномерный массив — это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми.
- Простое значение может быть строкой в двойных кавычках, числом, одним из литералов: true, false или 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 находится под ключом “phone”.

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

Строка в JSON — это упорядоченное множество из нуля или более символов юникода, заключенное 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 — они всегда представляются в виде одной строки, со специальной escape-последовательностью вместо переводов строк.
  { "description" : "Hello, Alice!\nHow do you do?" }
- Если текст в JSON содержит «специальные символы», то их приходится вручную экранировать escape-последовательностями.
  { "title" : "\"Rock&roll\" = life" }

- В XML произвольные дочерние узлы могут быть только у элементов — это вынуждает использовать их вместо, например, атрибутов, для большей гибкости и единообразия. JSON для создания иерархий можно использовать, например, списки или массивы.

Сериализация с помощью класса Gson

Сериализация в JSON возможна разными способами. Разберем один из простейших способов - с помощью класса Gson из пакета com.google.gson. Предположим, мы хотим сериализовать объект класса User.

```
public class User {

    public String firstname; // имя
    public String lastname; // фамилия
    public int school; // номер школы
}
```

Следующий код сериализует объект student в JSON.

```
User student = new User();
student.firstname = "Александр";
student.lastname = "Викторов";
student.school = 17;

Gson gson = new Gson();

Log.i("GSON", gson.toJson(student));
```

После исполнения программы в логах появится строка:

{"fistname":"Александр","lastname": “Викторов”,"school":17}

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:

```
String jsonText =
{\"fistname\":\"Alexandr\",\"lastname\":\”Viktorov\”,\"school\":17}";

Gson gson = new Gson();
User user = gson.fromJson(jsonText, User.class);
Log.i("GSON", "Name: " + user.firstname + "\nLast Name: " + user.lastname +
"\nSchool: " + user.school);
```

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

Здание 5.3.1

Измените расмотренный пример сериализации с помощью класса Gson таким образом, чтобы результат записывался и считывался из собственного локального файла.

Java интерфейс Serializable

Если рассмотренный выше способ сериализации не подходит, и в целом для понимания механизма сериализации в Java полезно изучить интерфейс `java.io.Serializable`*. Указание "implements Serializable" в объявлении класса и выполнение ряда простых требований позволяет сохранять/восстанавливать объекты этого класса стандартным образом:*

- *для сохранения объектов применяется класс* `ObjectOutputStream`
- *для восстановления* `ObjectInputStream`.

Описание этих классов и примеры использования можно посмотреть тут http://www.javable.com/tutorials/fesunov/lesson17/

5.3.4. Библиотека Retrofit

Для реализации клиент-серверных приложений, коммуницирующих по протоколу HTTP, удобно использовать специально разработанные для таких целей пакеты классов, или как их часто называют библиотеки. Одной из наиболее популярных библиотек является Retrofit. К её достоинствам относят:

1) Нет нужды делать запросы к HTTP API в отдельном потоке;
2) Сокращается длина кода и, соответственно, ускоряется разработка;
3) Возможно подключение стандартных пакетов для конвертации JSON в объекты и обратно(например, пакета Gson);
4) Динамическое построение запросов;
5) Обработка ошибок;
6) Упрощенная передача файлов.

Логика работы библиотеки завязана на аннотациях. Благодаря ним можно создавать динамические запросы на сервер.

Для описания запросов к серверу необходимо создать интерфейс, который впоследствии будет использоваться при генерации запросов. Над каждым методом должна стоять аннотация, с помощью которой Retrofit определяет, какого типа запрос обрабатывается данным методом. Также с помощью аннотаций можно указывать параметры запроса.

Вот так, например, выглядит описание GET-запроса:

```
import retrofit.client.Response;
import retrofit.http.GET;

public interface UserService {
   @GET("/greeting/user")
   Response fetchUser();
}
```

Адрес сайта, который отправляется запрос, не указывается. Он будет передан на этапе создания соединения клиент-сервер. Аннотация содержит лишь путь к PHP файлу на сервере. В классе Response содержится информация о статусе запроса и ответ от сервера.

POST-запрос выглядит схоже с GET:

```
import retrofit.client.Response;
import retrofit.http.POST;

public interface UserService{
   @POST("/greeting/registration")
   Response registerUser();
}
```

Можно изменять путь к файлу динамически:

```
@GET("/greeting/{firstName}/{lastName}")
Response fetchUser(@Path("firstName") String firstName, @Path("lastName")
String lastName);
```

Retrofit заменит слова «{firstName}» и «{lastName}» на те, которые будут переданы методу при вызове. Сам аргумент должен быть аннотирован словом Path, а в скобках должно заключаться ключевое слово.

Для того, чтобы задать запросу параметры используется аннотация @Query. Слово указанное в скобах рядом с аннотацией будет ключом, а аннотированый аргумент значением.

```
@GET("/greeting/user")
Response fetchUser(@Query("name") String name);
```

Создание соединения клиент-сервер выглядит, как серия вызовов методов библиотеки Retrofit. Например, так:

```
Retrofit retrofit = new Retrofit.Builder()
                    .baseUrl("http://192.168.72.3")
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();

UserService service = retrofit.create(UserService.class);
```

Через параметр метода baseUrl() передается адрес сервера. GsonConverterFactory – это класс для конвертации объектов в JSON. В нем используется уже рассмотренный нами класс Gson. UserService – интерфейс с запросами HTTP API.

После того, как соединение создано, можно его эксплуатировать, вызывая методы, привязанные к HTTP API.

```
Response resp = service.fetchUser(firstname, lastname);
```

Если ожидается ответ в виде специально созданной структуры данных, то используется конструкция Call<T>, где T – тип возвращаемой структуры данных. Например, интерфейс и структура могут выглядеть так:

```
public interface UserService {
   @GET("/greeting/user")
   Call<User> fetchUser();
}
public class User {
    public String firstName;
    public String lastName;
    public String fullName;
}
```

Call – это класс, который знает, что такое десериализация и может работать с HTTP запросами. Один экземпляр этого класса – это одна пара запрос-ответ. Экземпляр может быть использован лишь единожды. Чтобы использовать ту же пару повторно, необходимо использовать метод clone().

Вот как выглядит запрос, в ответ на который ожидается получить специальную структуру данных.

```
Call<User> call = service.fetchUser(firstname, lastname);
            try {
                Response<User> userResponse = call.execute();
                userFromServer = userResponse.body();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

Как видно, сначала создается вызов запроса, и лишь потом он выполняется методом execute(). Это дает возможность передавать созданный запрос в другие классы и выполнять его в любом месте программы.

Если попытаться выполнить вызов запроса дважды, то будет брошено исключение IllegalStateException.

```
userResponse = call.execute();

// This will throw IllegalStateException:
userResponse = call.execute();

Call<User> call2 = call.clone();
// This will not throw:
userResponse = call2.execute();
```

Если вызов клонировать и выполнить уже объект-клон, то программа сработает без ошибок.

Упражнение 5.3.2

Модифицируем программу из упражнения 5.3.1 так, чтобы она использовала JSON и библиотеку Retrofit. Полученный код клиентской части приведен ниже.

Импорт используемых классов:

```
import android.os.AsyncTask;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

import com.samsung.itschool.app5_3_2.R;
import com.samsung.itschool.app5_3_2.api.UserService;
import com.samsung.itschool.app5_3_2.model.User;

import java.io.IOException;
import retrofit.Call;
import retrofit.GsonConverterFactory;
```

```
import retrofit.Response;
import retrofit.Retrofit;
```

Базовая активность приложения:

```
public class MainActivity extends AppCompatActivity {
      private static String LOG_TAG = "MainActivity";
      private final String baseUrl = "http://192.168.72.3:8080";
      private TextView lastnameF;
      private String answerHTTP;
      private String lastnameS, firstnameS;
      private User userFromServer;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      lastnameF = (TextView) findViewById(R.id.lastnameF);

      }


      public void sendPOST(View view) {
      EditText lastname = (EditText) findViewById(R.id.lastname);
       EditText firstname = (EditText) findViewById(R.id.firstname);

      lastnameS = lastname.getText().toString();
      firstnameS = firstname.getText().toString();

      new MyAsyncTask().execute("");
      }
```

Асинхронный поток, осуществляющий клиент-серверный обмен информацией:

```
class MyAsyncTask extends AsyncTask<String, String, String> {

        @Override
        protected String doInBackground(String... params) {
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(baseUrl)
                .addConverterFactory(GsonConverterFactory.create())
                    .build();
        UserService service = retrofit.create(UserService.class);
        Call<User> call = service.fetchUser(firstnameS, lastnameS);
        try {
                Response<User> userResponse = call.execute();
                userFromServer = userResponse.body();
                Log.d(LOG_TAG, userFromServer.fullName);
        } catch (IOException e) {
                e.printStackTrace();
        }
        return null;
        }

        @Override
        protected void onPostExecute(String result) {
        super.onPostExecute(result);
        lastnameF.setText(userFromServer.fullName);
        }
}
```

Интерфейс клиент-серверного взаимодействия UserService:

```
import com.samsung.itschool.app5_3_2.model.User;
import retrofit.Call;
import retrofit.http.GET;
import retrofit.http.Path;

public interface UserService {
      @GET("/greeting/{firstName}/{lastName}")
      Call<User>      fetchUser(@Path("firstName")      String      firstName,
@Path("lastName") String lastName);
}
```

Класс передаваемой структуры данных User:

```
public class User {
      public String firstName;
      public String lastName;
      public String fullName;
}
```

На сервере получаем путь запроса с параметрами GET и возвращаем сериализованный объект User:

```
package ru.itschool.controller;

import ru.itschool.model.User;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class UserController {
      @RequestMapping(path    =    "/greeting/{firstName}/{lastName}",    method    =
RequestMethod.GET)
      public    User    greeting(@PathVariable("firstName")    String    firstName,
@PathVariable("lastName") String lastName) {
      return new User(firstName, lastName);
    }
}
```

Класс User на Java сервере:

```
package ru.itschool.model;

public class User {
      public String firstName;
      public String lastName;

      public User(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
      }

      public String getFullName() {
      return firstName + " " + lastName;
      }
}
```

*Аналогичная функциональность на PHP сервере будет выглядеть следующим образом:

```php
<?php

class User
{
	public $lastname = "";
	public $firstname = "";

	function __construct($firstname, $lastname)
	{
	$this->firstname = $this->getShortFirstname($firstname);
	$this->lastname = $lastname;
	}

	public function getShortFirstname($firstname)
	{
	$firstname = iconv('UTF-8', 'windows-1251', $firstname); // перевод кодировки нужен для того, чтобы корректно русские буквы отображались
	$firstname = substr($firstname, 0, 1);
	$firstname = iconv('windows-1251', 'UTF-8', $firstname); // перевод кодировки нужен для того, чтобы корректно русские буквы отображались
	return $firstname;
	}
}

if (isset($_GET["lastname"]) || isset($_GET["firstname"]))
	if ($_GET["lastname"] != "" && $_GET["firstname"] != "") {
	$user = new User($_GET["firstname"], $_GET["lastname"]);
	echo json_encode($user);
	} else
	echo "No GET data lastname or firstname";
else
	echo "No data";

?>
```

Задание 5.3.2

Модифицируйте программу из Упражнения 5.3.2 таким образом, чтобы к серверу по нажатию кнопки “Получить” посылался HTTP запрос GET, а по нажатию кнопки “Отправить” - POST.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Крылову Сергею Владимировичу.