

IT ШКОЛА SAMSUNG

Модуль 5. Основы разработки серверной части мобильных приложений

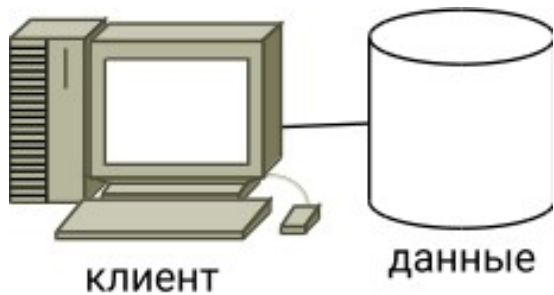
Вебинар. Клиент-серверная архитектура мобильных приложений

Автор: Яценко Д.В. 

SAMSUNG

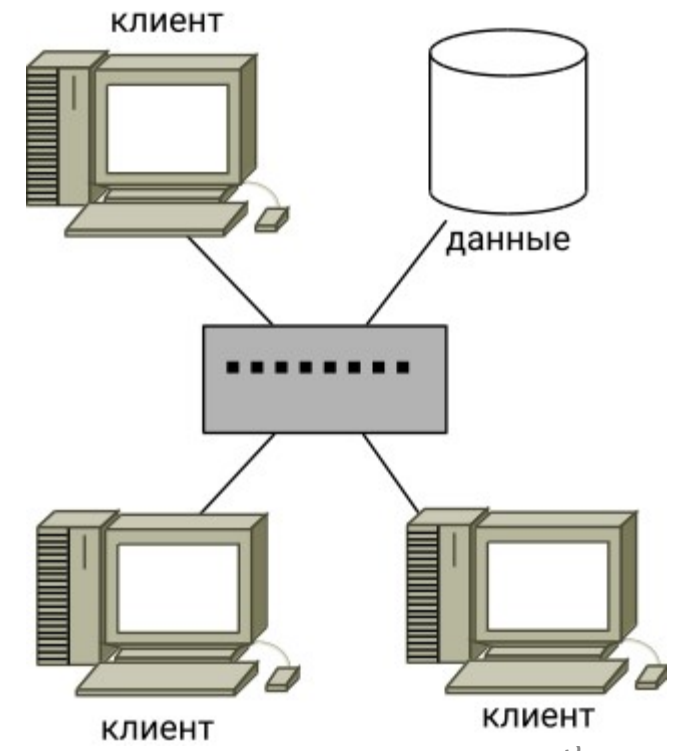
В этом вебинаре мы рассмотрим следующие темы:

- Общее описание клиент-серверной архитектуры, одно-многозвенные архитектуры. СУБД и сервера приложений.
- Развертывание среды разработки.
- Изучение сервера приложений Tomcat/Jboss, история вопроса, понятие сервлета. HTTP, GET, POST.
- Пример написания сервлета. Демонстрация. Понятие сериализации.
- Простое андроид приложение для работы с сервлетом. Сериализация SimpleXML.
- Альтернативные клиенты - HTML формы, JSP, JSTL, JSF.
- Совершенствование нашего облачного приложения.
- Понятие RPC. Понятие J2EE стека.

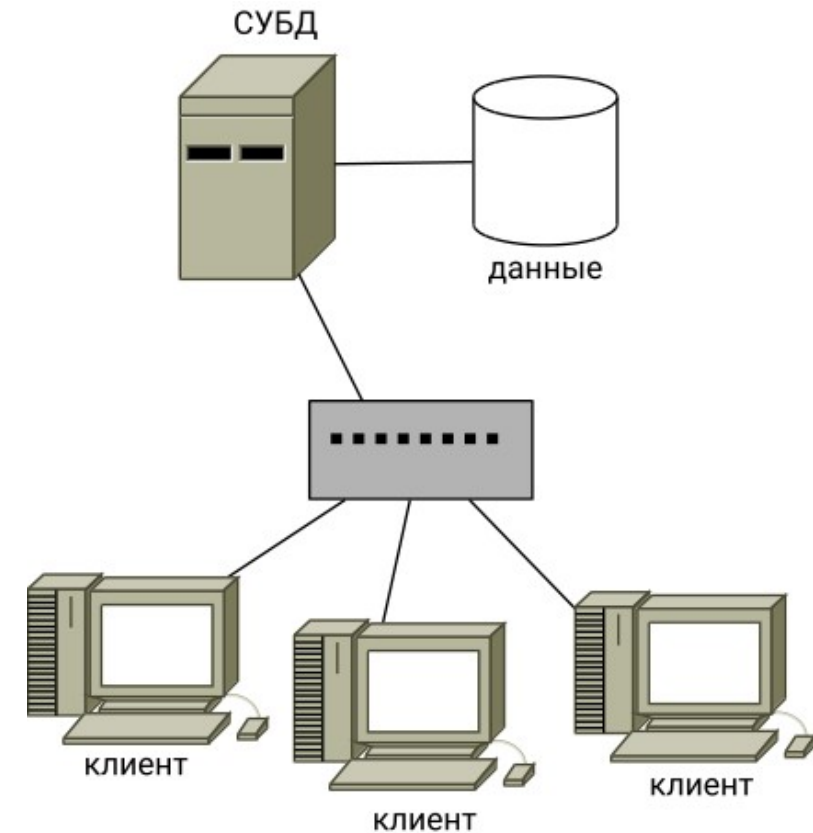


В начале на персональном компьютере данные долговременно сохранялись в виде файла на локальном диске. То есть и программа-потребитель и хранилище данных располагались в рамках одного компьютера. Такая схема называется “одно-звенной”.

Однако, когда широкое развитие получили компьютерные сети, появилась потребность в ПО позволяющем кооперативно работать с данными. Попытки совместного использования файлов данных, расположенных на общих сетевых дисках несло в себе большое количество проблем, таких как: безопасной работы с данными (Data Races), большого объема передаваемой по сети информации, информационной безопасности.



Появление СУБД стало решением проблемы кооперативного доступа к данным. При этом на выделенной ЭВМ работает специальное ПО, которое предоставляет по запросу от клиентов (программ пользователей) наборы нужных данных. При этом вышеперечисленные проблемы решались за счет того, что доступ к данным осуществлялся не независимо, а контролировался СУБД. Именно сервер теперь решает вопросы правомерности доступа к данным, одновременности их использования. Кроме того, так как СУБД выдает клиенту по запросу ограниченный набор данных, а не весь файл данных – это значительно уменьшает нагрузку на сеть.



По мере развития двух-звенных архитектур выявились новые проблемы, а именно:

- На клиентских ЭВМ работает ПО, которое обрабатывает полученные от СУБД данные, что вызывает большую нагрузку на пользовательские ЭВМ.
- Количество отобранных данных для клиентского ПО, хоть и уменьшилось, но все же еще велико.
- Приходится поддерживать актуальную версию ПО на всех клиентских ЭВМ.

Решение этих проблем было аналогично предыдущему. В клиентском ПО выделили интерфейсную часть (ответственную за отображения данных) и прикладную часть (ответственную за преобразование данных, расчеты, аналитику и т.д.), после чего прикладную часть вынесли на отдельный выделенный сервер, названный сервером приложений (Application Server). Таким образом в архитектуре стало три звена.



В течение этого занятия мы разработаем пример простого клиент-серверного приложения. Для этого нам понадобится настроить среду разработки. Рассмотрим создание приложения на условно-бесплатной платформе OpenShift (а позже и на других). Все операции по настройке среды разработки и регистрации приложения занимают всего пару минут. Настройка среды для разработки серверного звена:

- ★ Регистрируемся на <https://www.openshift.com/>
- ★ При регистрации можем сразу создать приложение (у нас conduit) и домен|группу (у нас myitschool) приложений. При создании приложений выбираем сервер приложений tomcat (jboss-7) и картридж postgresql (СУБД). Параметры подключения к постгресс запоминаем (логин, пароль, jdbc строку подключения).
- ★ Заходим в раздел web-console здесь можно создать новое приложение или изменить старое.

OPENSHIFT ONLINE

[Applications](#) [Settings](#) [Help](#) [OpenShift Hub](#)

conduit-myitschool.rhcloud.com [change](#)

Created 1 day ago in domain [myitschool](#) and the [aws-us-east-1](#) region

Idle 1

Cartridges

	Status	Gears	Storage
Tomcat 7 (JBoss EWS 2.0)	Idle	1 small	1 GB
<hr/>			
PostgreSQL 9.2	Database: conduit User: admin Password: show		

Continuous Integration

[Enable Jenkins](#)

Browse the [Marketplace](#), or [see the list of cartridges you can add](#)

Source Code

ssh://56bae05389f5cffeaa000040@conduit-myitschoc

Pass this URL to 'git clone' to copy the repository locally.

Remote Access

[Want to log in to your application?](#)

The command below will open a Secure Shell (SSH) session to your application on most operating systems. See our [SSH help page](#) for information about connecting with Windows, Mac, and Linux computers.

ssh 56bae05389f5cffeaa000040@conduit-myitschoc

Delete this application...

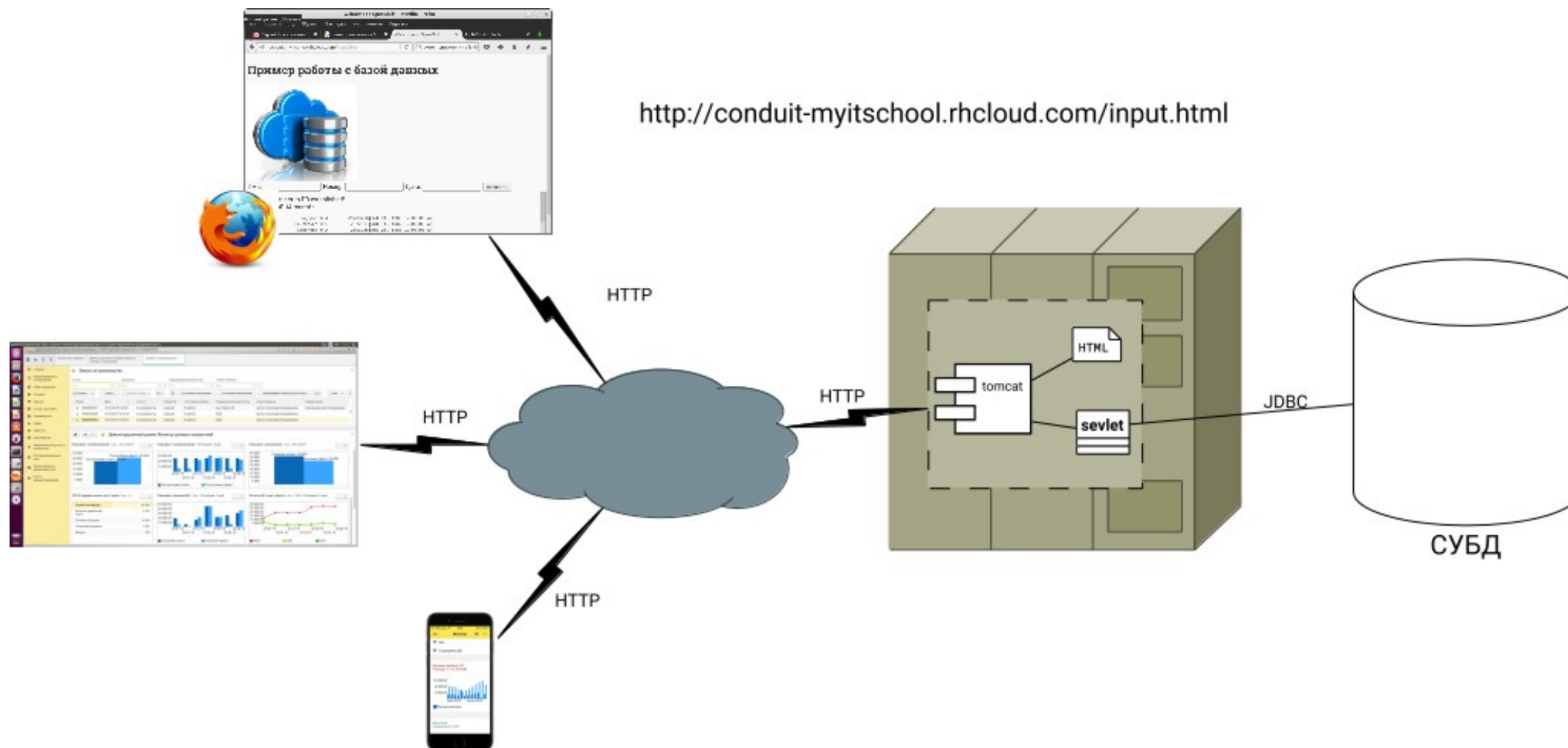
- ★ В эклипсе добавляем плагин JBoss Tools 3.3.0 (отсюда <http://tools.jboss.org/downloads/overview.html>)
- ★ Выбираем **File -> Import -> Existing OpenShift Application**, вводим логин и пароль, указываем путь к закрытому ssh ключу, который был создан при создании домена, указываем импортируемое приложение. Будет предложено указать путь к локальному git-репозиторию.
- ★ В перспективе “JBoss” редактируем нужные html, jsp, сервлеты и web.xml. По окончании в окне servers ПКМ по вашему серверу → publish. При этом приложение загружается (публикуется) на сервер приложений.
- ★ Просмотреть получившееся приложение можете по адресу - <http://<приложение>-<домен>.rhcloud.com/>

Прежде чем приступить к разработке своего приложения, рассмотрим сначала историю создания современных AS и их принципы работы.

До появления серверов приложений существовали подобные технологии WEB-приложений. Классическая платформа LAMP (Linux+Apache+Mysql+PHP) была идентична трех-звенной архитектуре. В качестве первого звена (клиентского приложения) выступал браузер, в качестве второго звена выступал Apache веб сервер с подключаемыми программами на языке PHP и в качестве третьего звена выступал СУБД Mysql, с которым взаимодействовали программы PHP для сохранения данных пользователей.

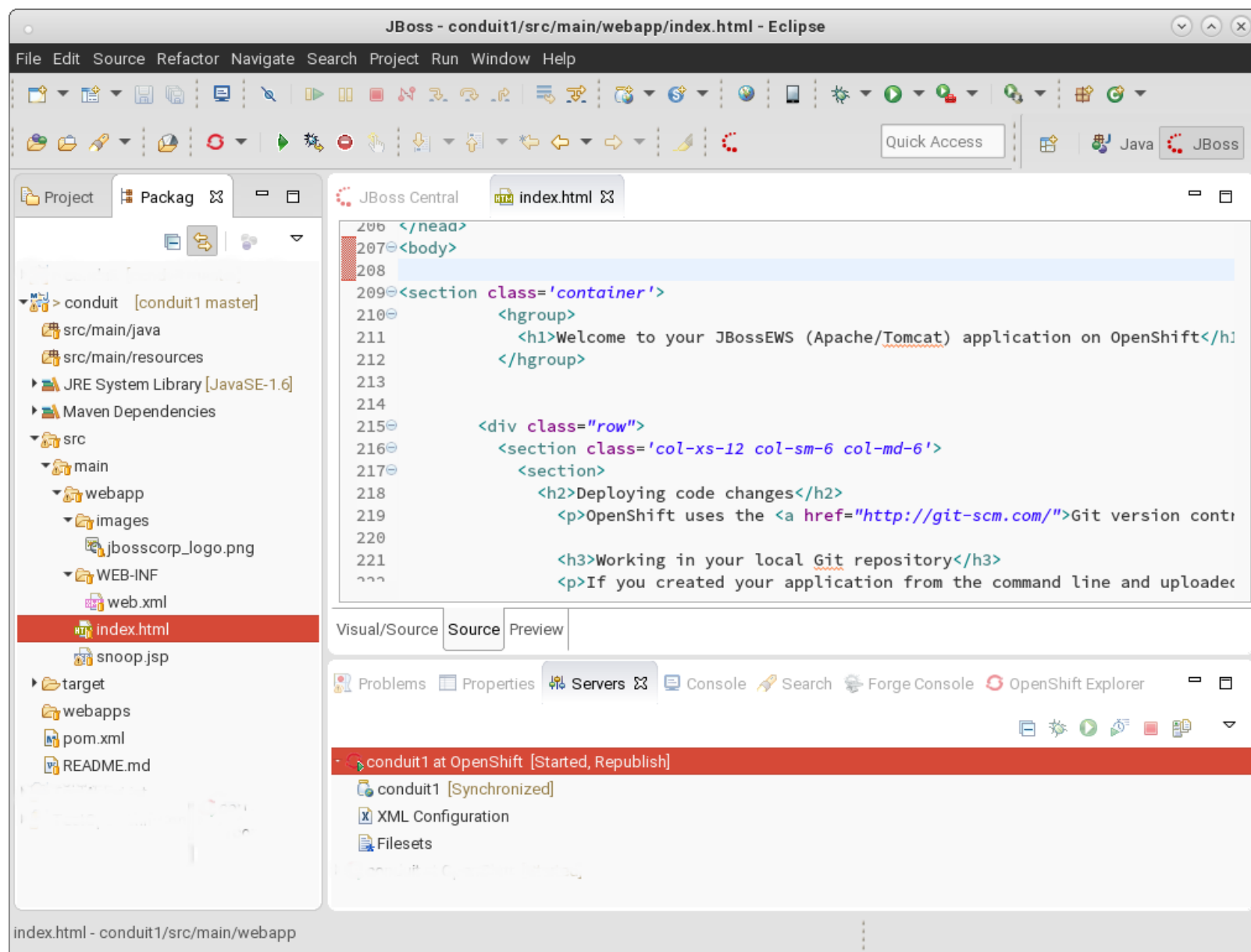
Однако эта платформа имела множество недостатков и в 90х годах прошлого века Apache Software Foundation и Sun Microsystems разрабатывают свои сервера приложений, исполняющие приложения на Java. В 1999г. Sun Microsystems подарил свой Java Web Server ASF. Эти проекты объединили под новым именем Tomcat.

Устройство и принципы работы серверов приложений для Java, реализованные в первом Tomcat не претерпели и сегодня.



Рабочий цикл клиент-серверного приложения следующий.

1. Клиентское ПО открывает сетевое соединение (клиентский сокет) по указанному адресу `conduit-myitschool.rhcloud.com` (IP:54.173.203.216).
2. Установится соединение с сервером Tomcat, слушающим серверный сокет.
3. Клиентское ПО по протоколу HTTP (через запросы GET/POST) передает серверу приложений свои данные и/или запрашивает требуемые данные.
4. Tomcat проанализировав запрос и URL (например `http://conduit-myitschool.rhcloud.com/input.html`) определяет какой ресурс требуется клиенту. Если требуется статический контент (страницы HTML, картинки и т.д.) то он выдается клиенту как есть.
5. Если же сервер определил, что запрошен вызов Java класса (сервлета), то вызывается соответствующий метод `doGet/doPost` соответствующего сервлета.
6. Если сервлету нужно обмениваться данными с имеющейся СУБД, то он делает это при помощи JDBC.



Можно поработать с полученным проектом. Например:

- отредактировать файл `/src/main/webapp/index.html` или,
- создать новый HTML файл в `/src/main/webapp/` или,
- добавить картинку `/src/main/webapp/images`.

Важно не забыть, что если вы добавили новый файл (не важно какого типа), обязательно его добавьте в индекс гита – ПКМ → team → add to index.

Однако работа со статическим контентом не так интересна. Гораздо интереснее добавить именно приложение. Для этого необходимо создать сервлет (Java класс отнаследованный от **`javax.servlet.http.HttpServlet`**). Для этого ПКМ по папке `/src/main/java` → new → other → web → servlet. Указав имя и пакет получаем готовый образец сервлета. В нем нам сразу предложат переопределение методов **`doGet`** и **`doPost`**. Эти методы будут вызываться при получении запросов GET и POST для URL соответствующего нашему сервлету.

Чтобы сервлет был доступен для запросов клиентов его необходимо зарегистрировать его в дескрипторе поставки /src/main/webapp/WEB-INF/web.xml . Если наш сервлет назывался HelloWorld.java , то в web.xml нужно добавить следующее:

```
<servlet>
  <servlet-name>HelloWorld</servlet-name>
  <servlet-class>conduit.HelloWorld</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>HelloWorld</servlet-name>
  <url-pattern>/conduit/HelloWorld</url-pattern>
</servlet-mapping>
```

Здесь мы дали имя сервлету, указали какой это Java класс и указали какой URL будет его вызывать. То есть *http://<addr>/conduit/HelloWorld* . В servlet 3.0 вместо записи в дескрипторе можно добавить аннотацию **@WebServlet("/conduit/HelloWorld")** .

После всех необходимых изменений можно опубликовать наш проект. В окне `servers` ПКМ по нашему проекту → `publish`. При этом произойдет сразу много вещей:

- все наши измененные и добавленные файлы будут отправлены в репозиторий `git` на сервере - откроется доп. окошко, где нужно поставить галочки на файлы, которые должны загрузиться на сервер и добавить комментарий для `git`,
- скомпилированный проект будет скопирован в нужную папку на сервере приложений,
- Север приложений будет перезапущен. Вам откроется окошко с журналом процесса перезагрузки севера и установки(`deployment`) приложения.

Обычно вместе с запуском сервера, запускаются все зарегистрированные сервлеты.

Теперь, когда мы добавили сервлет в проект реализуем метод **doGet**. Его входные параметры – объекты:

- **HttpServletRequest** — содержит входной HTTP запрос. Из него при помощи метода `getParameter` любое значение из запроса (`http://Url?name=Guest`)
- **HttpServletResponse** — содержит HTTP ответ, в том числе в нем можно установить код возврата, и из него можно получить выходной поток, который будет получен клиентом как результирующие данные запроса. Если клиент — браузер, то этот поток будет отображаться как полученная страница.

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType("text/html; charset=UTF-8");
    String name = request.getParameter("name");
    PrintWriter out = response.getWriter();
    out.println("Hello " + name + "!");
    out.flush();
    out.close();
}
```

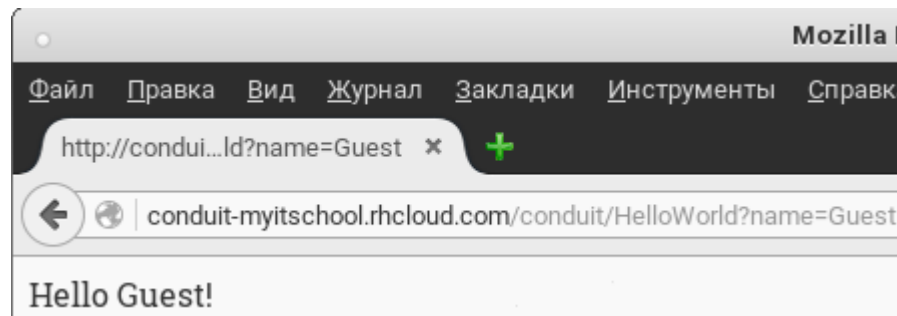
Рассмотрим метод **doPost**. Аналогично прошлому примеру примем один переданный параметр и выведем его в окно браузера. Однако, передать параметр методом POST в отличии от прошлого примера можно из браузера через HTML форму:

```
<form name="add" action="conduit/add" method="post">  
    Имя: <input type="text" name="name" required="true">  
    <input type="submit" value="добавить">  
</form>
```

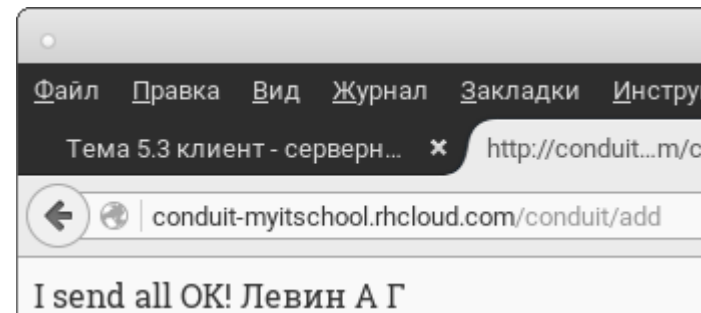
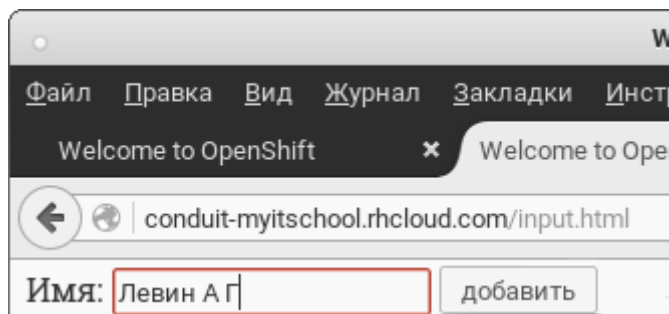
Обратите внимание на поле action – в нем указан URL, которому будет отправлен POST. Теперь сам метод doPost обрабатывающий запрос.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
    request.setCharacterEncoding("UTF-8");  
    String name = request.getParameter("name");  
    response.setContentType("text/html; charset=UTF-8");  
    OutputStreamWriter writer = new OutputStreamWriter(response.getOutputStream());  
    writer.write("I receive all OK! "+name);  
    writer.flush();  
    writer.close();  
}
```

Результат работы **doGet**:



Результат работы **doPost**:



Методы сервлета для остальных методов HTTP запроса – **doDelete**, **doOptions**, **doHead**, **doTrace**, а также методы жизненного цикла сервлета **init**, **destroy** мы рассматривать не будем.

Мы видели, что передаваемые данные в сервлете были получены как строки. Но нам нужно будет передавать не только строковые данные. Зачастую обмен сервера приложений и клиентского ПО производится сложными структурами данных (например списки сложных объектов .)

В общем случае для решения этой проблемы используют механизм сериализации. В самом общем случае – это возможность представить выбранный объект в виде последовательности байт. Соответственно десериализация – это возможность получить из последовательности байт объект идентичный исходному.

Во многих классах (**Integer**, **Date**) есть пара методов – **toString/parse**, позволяющих достаточно полно преобразовать объект в строку и обратно. В сложных объектах такие методы можно реализовать самостоятельно. Собственно в простейших случаях сериализацию делают именно так. Однако этот способ имеет массу недостатков и лучше использовать готовую библиотеку для сериализации. Они зачастую сериализуют данные как отформатированный по определенным правилам текст. Таких библиотек множество. Например XML сериализатор SimpleXML или JSON сериализатор Gson. При дальнейшей разработке мобильного приложения используем SimpleXML.

Теперь мы можем создавать мобильное приложение клиент-серверной архитектуры. В нем мы будем работать с телефонной книгой – списком контактов (имя, номер, дата рождения). Т.е. создадим приложение, которое при помощи протокола HTTP будет обмениваться данными с сервлетом, сериализуя эти данные в виде XML текста. Сервлет будет обрабатывать эти данные на стороне сервера, в том числе сохраняя их в БД.

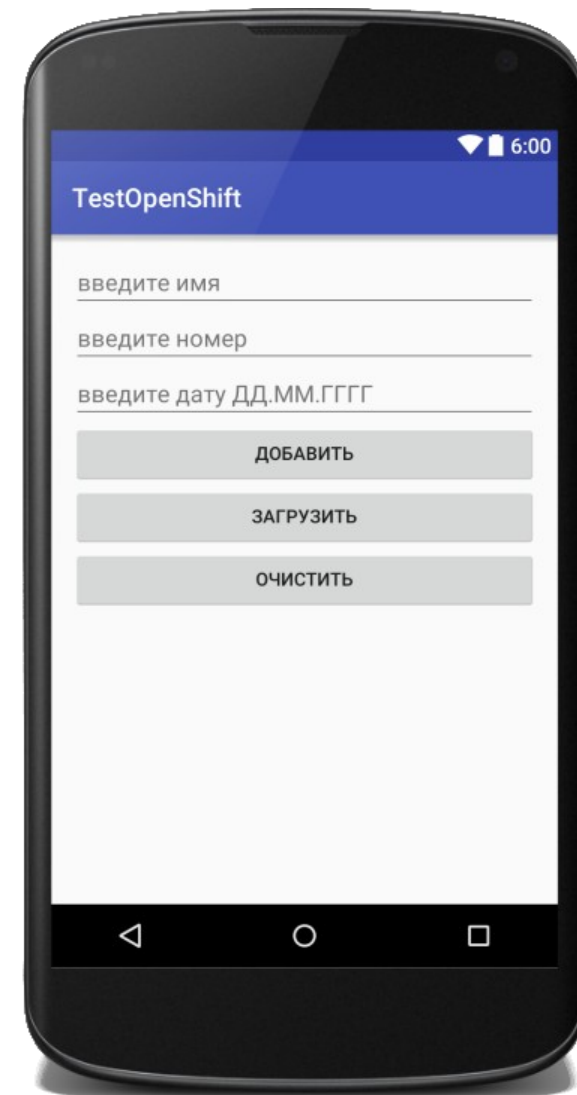
Создадим в андроид студии стандартный проект и добавим в него библиотеку SimpleXML. Для этого скачаем библиотеку `simple-xml-2.7.1.jar` и поместим её в проект в папку `/app/libs` и далее добавим одним из способов:

- ПКМ → Add As Library
- либо как строку в раздел **dependencies** файла `build.gradle` - **compile files('libs/simple-xml-2.7.1.jar')**

Макет интерфейса.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.samsung.myitschool.testsimplexml.MainActivity">
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="введите имя"
        android:inputType="textPersonName" />
    <EditText
        android:id="@+id/number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="введите номер"
        android:inputType="number" />
    <EditText
        android:id="@+id/date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="введите дату ДД.ММ.ГГГГ"
        android:inputType="date" />
```

```
<Button
    android:id="@+id/add"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="add"
    android:text="добавить" />
<Button
    android:id="@+id/load"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="load"
    android:text="загрузить" />
<Button
    android:id="@+id/clear"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="clear"
    android:text="очистить" />
<ScrollView
    android:id="@+id/scrollView"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_weight="1">
    <TextView
        android:id="@+id/out"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="" />
</ScrollView>
</LinearLayout>
```



Обрабатываемые данные будем хранить в таких структурах данных.

```
@Root
public class Contact {
    @Element
    String name;
    @Element(required = false)
    Date birthday;
    @Attribute
    long number;

    public Contact(@Element(name = "name") String name,
                  @Element(name = "birthday") Date birthday,
                  @Attribute(name = "number") long number){
        this.name=name;
        this.birthday=birthday;
        this.number=number;
    }

    public String toString(){
        return String.format("%30s | %15d | %20s",name,number,
                               birthday.toLocaleString());
    }
}
```

```
@Root
public class Phonebook {
    @ElementList
    List<Contact> contacts;

    public Phonebook(){
        contacts=new ArrayList<Contact>();
    }

    public String toString(){
        String str="";
        for (Contact contact:contacts)
            str+=contact+"\n";
        return str;
    }
}
```

```
public class MainActivity extends AppCompatActivity {
    EditText name,number,date;
    TextView out;
    Phonebook phonebook;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        name=(EditText)findViewById(R.id.name);
        number=(EditText)findViewById(R.id.number);
        date=(EditText)findViewById(R.id.date);
        out=(TextView)findViewById(R.id.out);
        phonebook=new Phonebook();
    }
    public void add(View V){
        String in_name=name.getText().toString();
        Date in_date= null;
        try {
            in_date = new SimpleDateFormat("dd.MM.yyyy",new Locale("ru")).parse(date.getText().toString());
        } catch (ParseException e) {
            Toast.makeText(this, "неверный формат даты", Toast.LENGTH_LONG).show();
        }
        long in_number=Long.parseLong(number.getText().toString());
        Contact contact=new Contact(in_name,in_date,in_number);
        phonebook.contacts.add(contact);
        send2Server(contact); // отправить на сервер (см. далее)
        Toast.makeText(this, "Добавлено", Toast.LENGTH_LONG);
    }
    public void clear(View V){
        out.setText("");
    }
    public void load(View V){
        getFromServer(); // получить с сервера (см. далее)
        String str="";
        out.setText(phonebook.toString());
        Toast.makeText(this, "Загружено", Toast.LENGTH_LONG);
    }
}
```

```
private void send2Server(final Contact contact) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                URL url = new URL("https://conduit-myitschool.rhcloud.com/conduit/HelloWorld");
                URLConnection connection = url.openConnection();
                connection.setDoOutput(true);
                BufferedOutputStream out = new BufferedOutputStream(connection
                    .getOutputStream());
                out.write("put\n".getBytes());
                Serializer ser = new Persister();
                ser.write(contact, out);
                out.flush();
                out.close();
                BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
                String returnString = reader.readLine();
                Log.e("NETWORK", "Ответ сервера "+returnString);
                reader.close();
            } catch (Exception e) {
                Log.e("NETWORK", "Ошибка передачи на сервер - " + e.getMessage());
            }
        }
    }).start();
}
```

```
private void getFromServer() {
    new Thread(new Runnable() {
        ArrayList<Contact> rez=new ArrayList<Contact>();
        @Override
        public void run() {
            try {
                URL url = new URL("https://conduit-myitschool.rhcloud.com/conduit/HelloWorld");
                URLConnection connection = url.openConnection();
                connection.setDoOutput(true);
                BufferedOutputStream out = new BufferedOutputStream(connection
                    .getOutputStream());
                out.write("get\n".getBytes());
                out.flush();
                out.close();
                BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
                Serializer ser = new Persister();
                phonebook=ser.read(Phonebook.class, reader);
                String returnString = reader.readLine();
                Log.e("NETWORK", "Ответ сервера "+returnString);
                reader.close();
            } catch (Exception e){
                Log.e("NETWORK", "Ошибка получения с сервера - " + e.getMessage());
            }
        }
    }).start();
}
```

Ну и конечно не забываем добавить в манифест разрешение на доступ в интернет.

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

Мы подготовили приложение, которое при добавлении контакта отправляет на сервер слово **put**, а затем и сам объект **Contact** через **POST** запрос, сериализовав объект в XML.

При нажатии кнопки «загрузить», мы отправляем на сервер **POST** запрос со словом **get**, а в ответ получаем сериализованный объект **phonebook**, который и показываем в поле таблицы контактов.

Теперь нам нужно в соответствующем сервлете написать код, осуществляющий прием и десериализацию полученного контакта и выгрузку и сериализацию объекта **phonebook**. В этом сервлете, кроме обмена данными с клиентским ПО, необходимо написать код, позволяющий сохранять полученные данные в БД и запрашивать эти данные для клиента. И конечно классы **Contact** и **Phonebook** должны быть добавлены и в проект на сервере рядом с сервлетом.

Поскольку весь обмен мобильного приложения с сервлетом происходит посредством POST, то весь нужный код у нас будет в методе doPost:

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    ServletInputStream sin = request.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(sin));
    Connect2db(); // наш метод подключения к БД (см. далее)
    String command = reader.readLine();
    if (command.equals("put")) {
        // Здесь получение объекта и сохранение его в БД
    } else if (command.equals("get")) {
        // Здесь запрос из БД и выдачи phonebook клиенту
    }
}
```

Ниже часть метода **doPost** – получение контакта от клиента

```
Serializer ser = new Persister();
Contact contact=null;
try {
    contact = ser.read(Contact.class, reader); // десериализуем в объект
    put(contact); // наш метод сохранения контакта в БД (см. далее)
    reader.close();
    sin.close();
    response.setStatus(HttpServletResponse.SC_OK);
    OutputStreamWriter writer = new OutputStreamWriter(response.getOutputStream());
    writer.write("I receive all OK!");
    writer.flush();
    writer.close();
} catch (Exception e) {
    System.out.println("ошибка получения объекта "+e.getMessage());
}
```


Ниже часть метода **doPost** – отправление тел. книги клиенту

```
reader.close();
sin.close();
response.setStatus(HttpServletResponse.SC_OK);
Phonebook phonebook=new Phonebook();
phonebook.contacts= getAll();//наш метод получ. списка контактов из БД (далее)
Serializer ser = new Persister();
OutputStreamWriter writer = new OutputStreamWriter(
    response.getOutputStream());

try {
    ser.write(phonebook, writer);
} catch (Exception e) {
    System.out.println("ошибка сериализации списка клиенту - "+e.getMessage());
}
writer.flush();
writer.write("I send all OK!");
writer.flush();
writer.close();
```

Теперь, когда код для взаимодействия с клиентом готов, покажем код для взаимодействия с БД. Метод 1/3 – подключение к БД.

```
public static final String DATABASE_HOST = System.getenv("OPENSIFT_POSTGRESQL_DB_HOST");
public static final String DATABASE_PORT = System.getenv("OPENSIFT_POSTGRESQL_DB_PORT");
public static final String DATABASE_NAME = "conduit"; // параметры подключения
private static final String USER = "admintyvjk9j"; // из web-консоли сервера
private static final String PASS = "tH139Jrvp6US"; // картридж POSTGRESQL
private Connection connection;

private void connect2db() {
    if(connection!=null) return;
    String url = "jdbc:postgresql://" + DATABASE_HOST + ":" + DATABASE_PORT + "/" +
DATABASE_NAME;
    try {
        Class.forName("org.postgresql.Driver");
        connection = DriverManager.getConnection(url, USER, PASS);
        System.out.println("Got Connection");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Метод 2/3 – запись объекта **Contact** в БД.

```
private void put(Contact contact){  
    try{  
        String query="INSERT INTO CONTACTS (CNUMBER, CNAME, CDATE) VALUES (?, ?, ?)";  
        PreparedStatement preparedStatement = connection.prepareStatement(query);  
        preparedStatement.setLong(1, contact.number);  
        preparedStatement.setString(2, contact.name);  
        preparedStatement.setDate(3, new java.sql.Date(contact.birthday.getTime()));  
        preparedStatement.executeUpdate();  
    }catch(Exception e){  
        e.printStackTrace();  
        System.out.println("Table insert error - "+e.getMessage());  
    }  
}
```

Метод 3/3 – запрос тел.книги из БД.

```
private List<Contact> getAll(){
    List<Contact> list;
    String query = "select * from contacts";
    try{
        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        if(rs==null){
            System.out.println("resultset is null");
            return null;
        }
        list=new ArrayList<Contact>();
        while (rs.next()) {
            String name = rs.getString("CNAME");
            long number = rs.getLong("CNUMBER");
            Date date = rs.getDate("CDATE");
            list.add(new Contact(name,date,number));
        }
    }catch(Exception e){
        System.out.println("DB error. Table not created yet? Try create"); // см. далее
        createTable();return null;
    }
    return list;
}
```

И последнее - создание таблицы БД.

```
private void createTable(){
    String query="create table contacts ( cnumber int,
                                   cname varchar(255),cdate date); ";

    try{
        Statement stmt = connection.createStatement();
        stmt.executeUpdate(query);
    }catch(Exception e){
        System.out.println("Table create errorr - "+e.getMessage());
    }
}
```

Сервлет для взаимодействия с мобильным приложением готов. Публикуем и пробуем работать с андроид-приложением. Мы получили работающее трехзвенное приложение. Давайте рассмотрим альтернативные варианты клиентских программ.



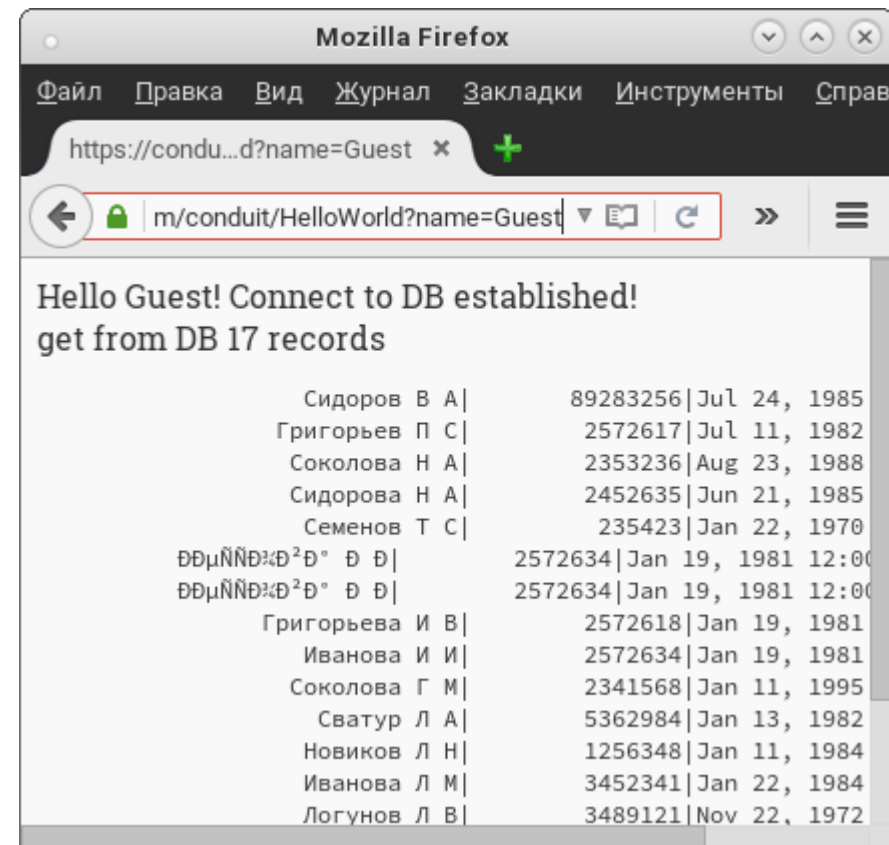
Имя	Номер	Дата
Сидоров В А	89283256	24 Июл 1985 г. 8:00:00
Григорьев П С	2572617	11 Июл 1982 г. 8:00:00
Соколова Н А	2353236	23 Авг 1988 г. 8:00:00
Сидорова Н А	2452635	21 Июнь 1985 г. 8:00:00
Семенов Т С	235423	22 Янв 1970 г. 8:00:00
Д ДμÑ Ñ Д¾Д²Д° Д Д	2572634	19 Янв 1981 г. 8:00:00
Д ДμÑ Ñ Д¾Д²Д° Д Д	2572634	19 Янв 1981 г. 8:00:00
Григорьева И В	2572618	19 Янв

```
public class Connection {
    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        System.out.print("Имя: ");
        String name=sc.nextLine();
        System.out.print("Номер: ");
        long number=sc.nextLong();
        System.out.print("Дата(дд.мм.гггг): ");
        Date date;
        try {
            String sdate=sc.next();
            date=new SimpleDateFormat("dd.MM.yyyy",new Locale("ru")).parse(sdate);
            Contact contact=new Contact(name,date,number);
            URL url = new URL("https://conduit-myitschool.rhcloud.com/conduit/HelloWorld");
            URLConnection connection = url.openConnection();
            connection.setDoOutput(true);
            BufferedOutputStream out = new BufferedOutputStream(connection
                .getOutputStream());
            out.write("put\n".getBytes());
            Serializer ser = new Persister();
            ser.write(contact, out); out.flush(); out.close();
            BufferedReader reader = new BufferedReader(new
                InputStreamReader(connection.getInputStream()));
            String returnString = reader.readLine();
            System.out.println("Ответ сервера "+returnString);
            reader.close();
        }catch(Exception e){
            System.out.println("Ошибка передачи на сервер - " + e.getMessage());
        }
    }
}
```

```
<terminated> Connection [Java Application] /
Имя: Садовников Е О
Номер: 2378695
Дата(дд.мм.гггг): 22.09.1990
Ответ сервера I receive all OK!
|
```

Мы уже обращались из браузера к сервлету (HTTP запрос **GET**) для проверки соединения. Теперь используем этот же запрос для получения данных с сервера приложений в страницу браузера. Для этого в сервлете реализуем метод **doGet**:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html; charset=UTF-8");
    String name = request.getParameter("name");
    PrintWriter out = response.getWriter();
    out.println("Hello " + name + "!");
    out.flush();
    Connect2db(); // см. ранее
    if (connection != null) {
        out.println("Connect to DB established!<br/>");
        out.flush();
    }
    java.util.List<Contact> list = getAll(); // см. ранее
    if (list != null) {
        out.println("get from DB " + list.size() + " records<br/>");
        out.println("<pre>");
        for(Contact contact:list) out.println(contact+"");
        out.println("</pre>");
        out.flush();
    }
    out.close();
}
```



Из браузера из веб страницы также можно отправить **POST** запрос сервлету. Для этого используется механизм web forms. Создадим HTML страницу - /src/main/webapp/input.html

```
<html lang="en">
<head> <meta charset="utf-8"> </head>
<body>
<h1>Пример работы с базой данных</h1>
<form name="add" action="conduit/add" method="post">
  Имя: <input type="text" name="name" required="true">
  Номер: <input type="text" name="number" required="true">
  Дата: <input type="text" name="birthday" required="true">
  <input type="submit" value="добавить">
</form>
<iframe src="/conduit/HelloWorld?name=" height="100%" width="100%"></iframe>
</body>
</html>
```

Обратите внимание на атрибут формы action. В нем мы указываем URL (абсолютный или относительный) для отправки **POST** запроса. Здесь мы указали адрес другого сервлета. Этот новый сервлет и будет обрабатывать **POST** запрос от формы. Сделано в данном случае так, чтобы не пререгружать кодом наш первый сервлет. На след. слайде рассмотрим этот сервлет.

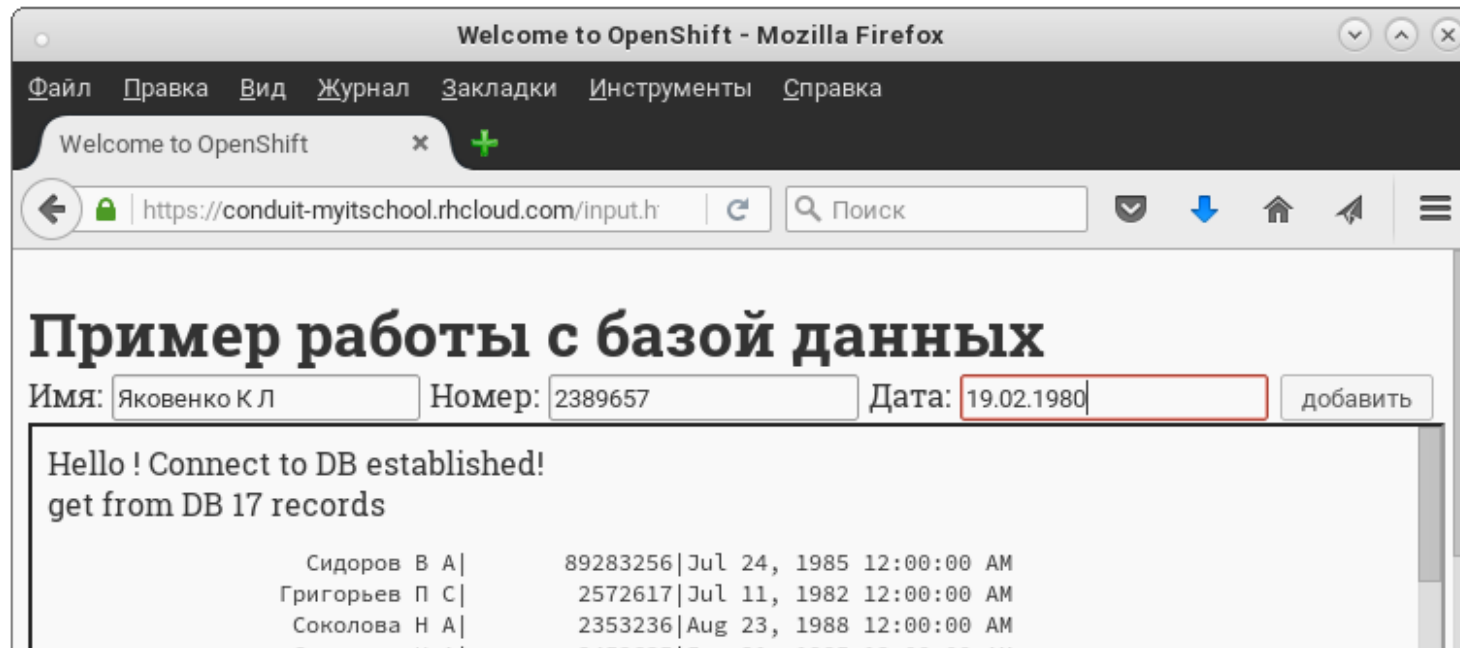
В методе **doPost** мы реализуем только получение данные от веб формы (как параметры) и сохранение их в БД. Методы работы с БД просто копируем с первого сервлета без изменений.

```
public class Form extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        request.setCharacterEncoding("UTF-8");
        String name = request.getParameter("name");
        String number = request.getParameter("number");
        String birthday = request.getParameter("birthday");
        Date in_date= null;
        try {
            System.out.println(name+"--"+birthday);
            in_date = new SimpleDateFormat("dd.MM.yyyy",new Locale("ru")).parse(birthday);
        } catch (ParseException e) {
            System.out.println("Неверный формат даты "+e.getMessage());
        }
        Contact contact=new Contact(name, in_date,Long.parseLong(number));
        connect2db();
        put(contact);
        response.setStatus(HttpServletResponse.SC_OK);
        response.sendRedirect("/input.html");
    }
}
```

Обратите внимание что класс сервлета называется **Form**, а из веб формы мы вызываем **conduit/add** . Это вполне допустимо, просто в дескрипторе развертывания **web.xml** указали следующий маппинг:

```
<servlet>
  <servlet-name>Form</servlet-name>
  <servlet-class>conduit.Form</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Form</servlet-name>
  <url-pattern>/conduit/add</url-pattern>
</servlet-mapping>
```



Рассмотрим еще один вариант взаимодействия с сервером приложений. Это Java Server Pages (**JSP**). Вспомним что происходит при обращении веб браузера по URL, сопоставленному с сервлетом. Так как при этом браузер отправляет HTTP **GET** запрос, сервер приложений вызовет метод **doGet** соответствующего сервлета. В этом сервлете мы можем получить выходной поток, в который будем писать текст, который будет отображаться в качестве выданной веб страницы в браузере. Что то вроде ЭТОГО:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType("text/html; charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html>");    out.println("<body>");
    out.println("<h3>Hello " + name + "!</h3>");
    out.println("get from DB " + list.size() + " records<br/>");
    out.println("<pre>");
    for(Contact contact:list) out.println(contact+"");
    out.println("</pre>");
    out.println("</body>");    out.println("</html>");
    out.flush();
    out.close();
}
```

То есть для того чтобы сформировать даже небольшую веб страницу придётся написать большое количество java кода сотсящего на 90% из **out.println()**. Именно для этих ситуаций разработчики сервлет API предложили альтернативный вариант формирования страницы – по принципу “наоборот”. Все что нужно выводить в HTML вы пишете “как есть”, а то что нужно выполнить как java код отметить специальным тегом **<%...%>**. Исходя из этого наш пример превратится во что то вроде:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <body>
    <h3>Helo <%= name; %></h3>
    <pre>
      <%= list.toString(); %>
    </pre>
  </body>
</html>
```

Собственно так и устроена **JSP** страница. Она очень похожа на HTML со вставками java кода. Такой вариант применяется в основном если необходимо сформировать веб страницы на сервере приложений, содержащие большое количество веб разметки. Да и пишется он практически как HTML страница.

На самом деле за фасадом **JSP** скрывается все тот же сервлет. В момент когда вы впервые обращаетесь к **JSP** странице, сервер приложений компилирует её, автоматически создавая сервлет, в котором в методе **doGet** находится java код, выполняющий нужные операции и выводящий нужную разметку через все тот же **out.println()**.

Рассмотрим пример формирования динамической веб страницы сервером приложений из JSP страницы

```
<%@page import="conduit.Contact"%>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ page import='java.sql.*' %>
<%@ page import='javax.sql.*' %>
<%@ page import='javax.naming.*' %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Пример работы из JSP с базой данных </title>
</head>
<body>
<h1>Пример работы с базой данных</h1>
  
  <pre>
<%
Connection result = null;
try {
    InitialContext ic = new InitialContext();
    Context initialContext = (Context) ic.lookup("java:comp/env");
    DataSource datasource = (DataSource) initialContext.lookup("jdbc/PostgreSQLDS");
    result = datasource.getConnection();
    Statement stmt = result.createStatement() ;
    String query = "select * from contacts;" ;
    ResultSet rs = stmt.executeQuery(query) ;
    while (rs.next()) {
        out.println(new Contact(rs.getString(2),rs.getDate(3),rs.getLong(1)));
    }
} catch (Exception ex) {
    out.println("Exception: " + ex + ex.getMessage());
}
%>
</pre>
</body>
</html>
```



Пример работы из JSP с базой данных - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка

Пример работы из JSP с б... x +

https://conduit-myitschool.rhcloud.com/dbquery.jsp

Пример работы с базой данных



Сидоров В А	89283256 Jul 24, 1985 12:00:00 AM
Григорьев П С	2572617 Jul 11, 1982 12:00:00 AM
Соколова Н А	2353236 Aug 23, 1988 12:00:00 AM
Сидорова Н А	2452635 Jun 21, 1985 12:00:00 AM
Семенов Т С	235423 Jan 22, 1970 12:00:00 AM

JSP очень быстро преобрел популярность, так как позволил заметно снизить объем программного кода при написании динамических веб-страниц. Конечно разработчики стандартов на этом не остановились и было создано расширение к **JSP** - JavaServer Pages Standard Tag Library (**JSTL**). При его использовании вы можете создать свою библиотеку HTML-подобных тегов сопоставленных с определенными функциями. Таким образом разработка динамической страницы вообще переходила от Java-программиста к веб-дизайнеру.

Дальнейшее развитие привело к появлению фреймворка Java Server Faces (**JSF**) – динамические страницы пишутся на XML (отчасти похожем на HTML) связанным напрямую с бизнес-логикой приложений в сервере приложений.

Успех этих архитектур породил целую плеяду реализаций серверов приложений и API и фреймворков для работы с ними. Но все же в основе всех (или почти всех) лежит сервлет.

В данный момент у нас есть реализация прикладной части (бизнес-логики) на сервере приложений, БД, несколько реализаций клиентского ПО, и веб интерфейс к нашему приложению а также сопутствующие материалы. Последний штрих в оформлении нашего приложения – это собрать все ресурсы в единый комплекс. Создадим главную страницу в которой предоставим доступ ко всем разработанным нами ресурсам, оформленным в едином стиле.

Пример работы с базой данных

Сидоров В А | 89283256 | Jul 24, 1985 12:00:00 AM
Григорьев П С | 2572617 | Jul 11, 1982 12:00:00 AM
Соколова Н А | 2352236 | Aug 23, 1988 12:00:00 AM
Сидорова Н А | 2452535 | Jun 21, 1985 12:00:00 AM

Тема 5.3 клиент - серверная архитектура

Здесь приведены различные способы взаимодействия клиентов и сервера.

- [HTML форма ввода с POST отправкой на сервлет](#)
- [JSP страница со встроенным запросом к БД](#)
- [сервлет, обрабатывающий GET и POST запросы](#)
- [JSP страница выдающая статистику по использованию ресурсов сервера](#)
- [Android клиент](#)

Документация

- [Вебинар - Клиент серверная архитектура мобильных приложений](#)
- [Краткая инструкция OpenShift](#)

Проблемы

```
<terminated> Connection [Java Appli
Имя: Новикова А В
Номер: 2896714
Дата (дд.мм.гггг): 25.06.199
Ответ сервера I receive all
```

ШКОЛА SAMSUNG

Модуль 5. Основы разработки серверной части мобильных приложений
Вебинар. Клиент-серверная архитектура мобильных приложений
Автор: Яценко Д.В.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Тема 5.3 клиент - серверная архитектура</title>
</head>
<body>
</img>
<h2>5.3. Клиент-серверная архитектура.</h2>
<h3>Клиентское ПО</h3>
Здесь приведены различные способы взаимодействия клиентов и сервера.
<ul>
<li><a href="/input.html">HTML форма ввода с POST отправкой на сервлет</a></li>
<li><a href="/dbquery.jsp">JSP страница со встроенным запросом к БД</a></li>
<li><a href="/conduit/HelloWorld?name=Guest">сервлет, обрабатывающий GET и POST запросы</a></li>
<li><a href="/snoop.jsp">JSP страница выдающая статистику по использованию ресурсов сервера</a></li>
<li><a href="/files/client.apk">Android клиент</a></li>
<h3>Документация</h3>
<li><a href="/files/webinar.pdf">Вебинар - Клиент серверная архитектура мобильных приложений</a></li>
<li><a href="/files/README_OpenShift.pdf">Краткая инструкция OpenShift</a></li>
</ul>
</body>
</html>
```

Мы создали распределенное приложение, однако можно заметить что у нас достаточно громоздко организовано:

- обмен с сервером приложений,
- взаимодействие с СУБД.

Конечно эти проблемы не остались незамечены, и были созданы множество реализаций технологий **RPC** (Remote Procedure Call) и **ORM** (Object Relative Mapping). Существует большое количество различных реализаций **RPC**, но большинство из них работает следующим образом:

- на клиентском ПО настраивается контекст (задаются параметры сервера приложений),
- после настройки из контекста можно вызывать функции объектов созданных на сервере, как будто вы вызываете их локально,
- при удаленном вызове функций сериализация/десериализация происходит невидимо для программиста,
- сетевое взаимодействие может происходить как поверх HTTP так и на основе собственных протоколов.

Также без внимания разработчиков не остался и достаточно громоздкий процесс взаимодействия с СУБД посредством **JDBC**. На данный момент существует множество реализаций **ORM**, которые прозрачно от программиста позволяют сопоставлять структуры данных и таблицы БД. Т.о. работа с БД превращается в простую работу с Java коллекциями объектов и командами к персистеру на сохранение или извлечение данных.

Контейнер сервлетов, один из ключевых компонентов более крупного набора стандартов, совокупно называемого как Java 2 Enterprise Edition. **J2EE** определяет группу **API** и фреймворков для создания ПО для предприятий.

Вышеупомянутые спецификации это лишь малая часть J2EE. Вот общий состав:

- **EJB** : Enterprise JavaBeans — спецификация технологии серверных компонентов, содержащих бизнес-логику
- **JPA** : Java Persistence API
- **Servlet** : Обслуживание запросов веб-клиентов
- **JSP** : JavaServer Pages — динамическая генерация веб-страниц на стороне сервера
- **JSTL** : JavaServer Pages Standard Tag Library
- **JSF** : JavaServer Faces — компонентный серверный фреймворк для разработки веб-приложений на технологии Java
- **JAX-WS**: Java API for XML Web Services — создание веб-сервисов
- **JAX-RS** : Java API for RESTful Web Services — создание RESTful веб-сервисов
- **JNDI** : Java Naming and Directory Interface — служба каталогов

- **JMS** : Java Message Service — обмен сообщениями
- **JTA** : Java Transaction API
- **JAAS** : Java Authentication and Authorization Service — Java-реализация PAM
- **JavaMail** : Получение и отправка электронной почты
- **JACC** : Java Authorization Contract for Containers
- **JCA** : J2EE Connector Architecture
- **JAF** : JavaBeans Activation Framework
- **StAX** : Streaming API for XML
- **CDI** : Context and Dependency Injection

Сегодня существует множество как платных (например **WebSphere** от IBM) так и бесплатных (например **Jboss**) серверов приложений **J2EE** стека, которые вы можете самостоятельно разворачивать и использовать для установки своих приложений.

Кроме того есть облачные платформы (платно-бесплатные), содержащие отдельные элементы J2EE. Наиболее известные из них это:

- **Heroku**(www.heroku.com)
- **GogleAppEngine** (cloud.google.com/appengine)
- **OpenShift**(www.openshift.com)

Они конечно в условно-бесплатном режиме имеют ряд ограничений.

Давайте теперь рассмотрим альтернативную реализацию нашего мини проекта. Заменим в нем всё — **OpenShift** на **Heroku**, сериализацию **XML** на **JSON**, а в Android приложение добавим библиотеку **Retrofit** (библиотеку для работы с **Rest API**), а на сервере будем использовать фреймворк **Spring**.

Для начала работы с платформой **Heroku** (heroku.com) нам необходимо зарегистрироваться на ней и пройти мини обучающий курс, в ходе которого вам буду продемонстрированы все приёмы работы с платформой и установлено необходимое ПО (**heroku toolbelt**).

Для создания серверного приложения будем использовать фреймворк **Spring**. Для этого нам будет необходим **Eclipse** с установленным фреймворком **STS** (Spring Tool Set).

В эклипсе создаем проект **Spring Starter Project** в диалоге создания используем предложенное имя проекта **demo** и цель — **web**.

Наш проект будет содержать 4е класса:

- **Contact** и **Phonebook** — аналогично прошлому приложению как модель данных (добавлены геттеры),
- **GreetingController** — как замена сервлету,
- **DemoApplication** — очень маленький класс, использующийся спрингом как точка входа.

```
public class Contact {
    String name;
    public String getName() {
        return name;
    }

    public Date getBirthday() {
        return birthday;
    }

    public long getNumber() {
        return number;
    }

    Date birthday;
    long number;

    public Contact(String name, Date birthday, long number) {
        this.name=name;
        this.birthday=birthday;
        this.number=number;
    }

    public String toString(){
        return String.format("%30s|%15d|%20s",name,
                               number,birthday.toLocaleString());
    }
}
```

```
public class Phonebook {
    List<Contact> contacts;

    public Phonebook(){
        contacts=new ArrayList<Contact>();
    }

    public List<Contact> getContacts() {
        return contacts;
    }

    public String toString(){
        String str="";
        for (Contact contact:contacts)
            str+=contact+"\n";
        return str;
    }
}
```

```
@RestController
public class GreetingController {

    @RequestMapping("/phonebook")
    public Phonebook phonebook(@RequestParam(value="name",
        required=false, defaultValue="World") String name) {
        Phonebook pbook=new Phonebook();
        pbook.contacts.add(new Contact("Иванов", new Date(), 2572617));
        pbook.contacts.add(new Contact("Петров", new Date(), 3458642));
        return pbook;
    }
}
```

```
@ComponentScan
@SpringBootApplication
public class DemoApplication {

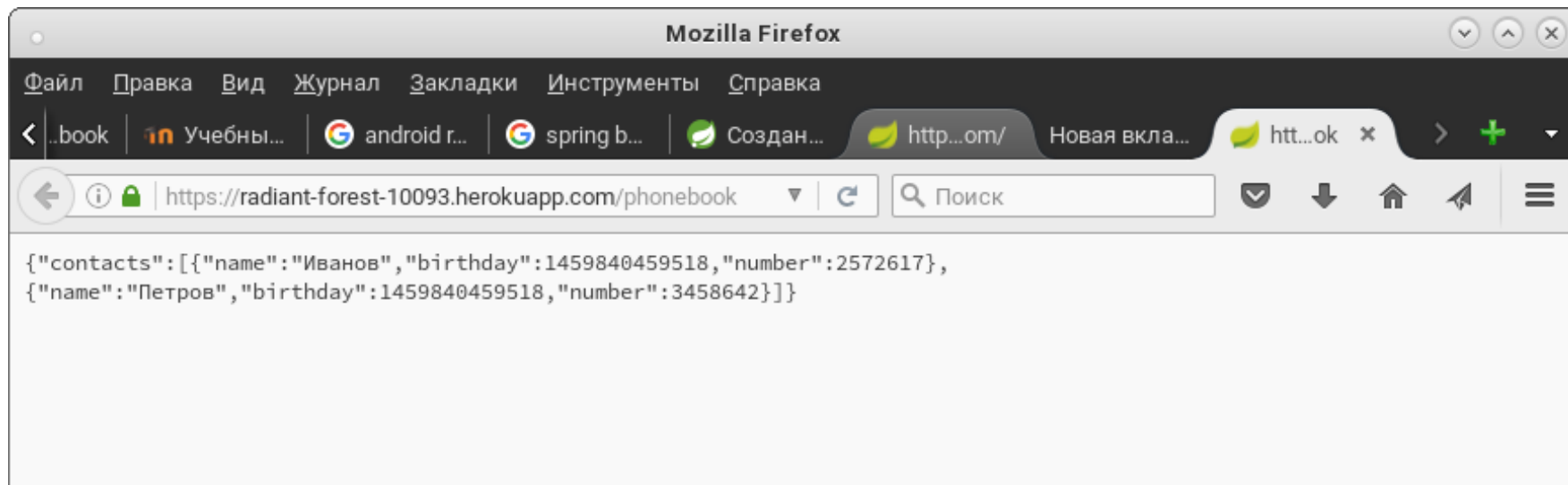
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Конечно приведенный пример упрощен по сравнению с предыдущим — здесь у нас только один метод, выдающий телефонную книгу. Но тем не менее посмотрите насколько проще стал код. Вы описываете просто обычный класс с методом, который выдает нужную информацию. В соответствии с аннотациями методы вашего класса будут сопоставлены с URL сервера. Сериализация/десериализация данных идет в прозрачном режиме и не требует никакой дополнительной настройки.

Теперь осталось загрузить проект на платформу и проверить его работоспособность. Для загрузки проекта нужно сделать немного непривычных действий:

- В консоли выполнить **heroku login** (авторизуемся)
- Переходим в консоли в папку с проектом
- Создаем в папке проекта файл **Procfile** со следующим содержимым
web: java -Dserver.port=\$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
- **heroku create** — создаем проект (вам будет выдана ссылка проекта)
- **git init**
- **git add .**
- **git commit -m "first commit"**
- **git push heroku master**

После этого проект загружен и вы можете проверить его — зайдя браузером например на <https://radiant-forest-10093.herokuapp.com/phonebook>



Здесь вы видите ответ вашего приложения в виде сериализованного в JSON объекта Phonebook состоящего из двух контактов. На этом разработка серверного функционала закончена и мы можем подвести итог. По сравнению с вариантом на основе простого сервлета — этот вариант гораздо проще — вы просто пишете свои классы, с помощью аннотаций делая некоторые методы доступными через HTTP запросы.

Spring Web Service API на самом деле включает в себя сервлет, который называется **DispatcherServlet** и который занимается обработкой HTTP запросов и ответов. Именно он и вызывает методы ваших классов которые аннотированы как **@RestController** (в Spring 4). На методах соответственно должна быть аннотация **@RequestMapping** для сопоставления с URL запросов.

По умолчанию для сериализации используется **JSON**, но при желании можно настроить и **XML** сериализацию.

Разработаем теперь Android приложение для работы с этим сервером. Но в отличии от прошлого примера используем библиотеку **Retrofit** (библиотека для работы с RestAPI). Для этого создадим стандартное приложение Android и добавим в файл **build.gradle** (app)

```
compile 'com.squareup.retrofit2:retrofit:2.0.0-beta3'
```

```
compile 'com.squareup.retrofit2:converter-gson:2.0.0-beta3'
```

И нажмём **sync**, после чего скачаются необходимые библиотеки.

Далее создаем следующий набор классов:

- **Phonebook** и **Contcat** — классы модели,
- **MainActivity** — основной класс приложения,
- **PhonebookInterface** — интерфейс для описания запросов.

Ну и конечно не забываем добавлять в манифест разрешение:

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

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.samsung.myitschool.testtheroku.MainActivity">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="отправить запрос"
        android:onClick="sendPOST"
        android:id="@+id/button" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/result" />
</LinearLayout>
```

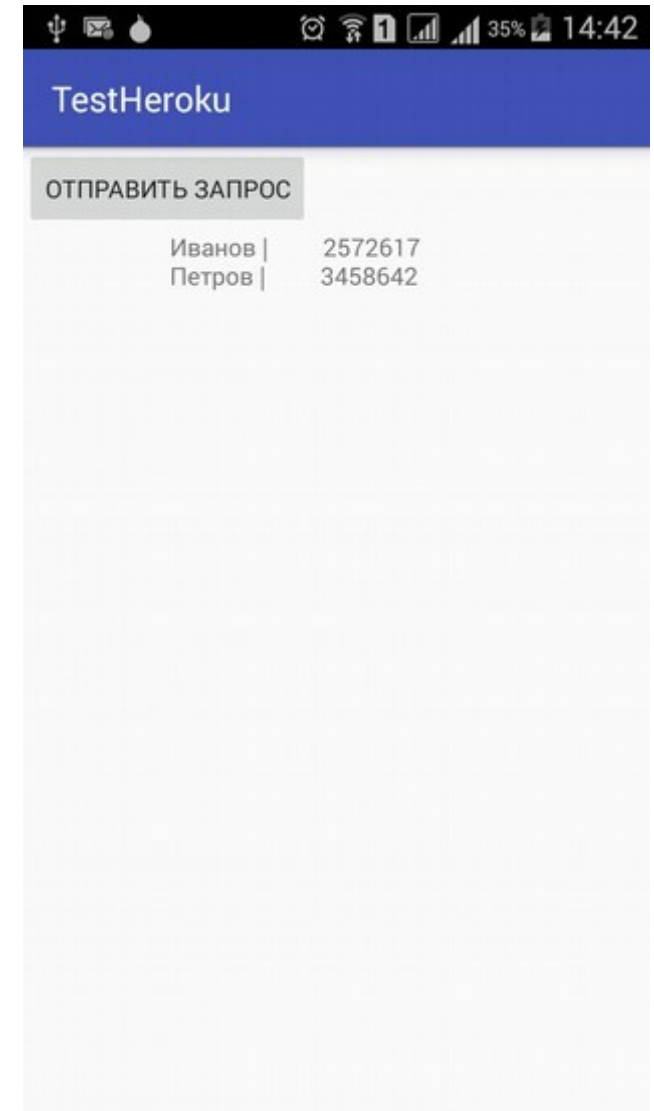
```
public class Contact {  
    String name;  
    long number;  
    public Contact(String name, long number){  
        this.name=name;  
        this.number=number;  
    }  
    public String toString(){  
        return String.format("%30s | %15d", name, number);  
    }  
}
```

```
public class Phonebook {  
    List<Contact> contacts;  
    public Phonebook(){  
        contacts=new ArrayList<Contact>();  
    }  
    public String toString(){  
        String str="";  
        for (Contact contact:contacts)  
            str+=contact+"\n";  
        return str;  
    }  
}
```

```
public interface PhonebookService {  
    @GET("/phonebook")  
    Call<Phonebook> fetchPhonebook();  
}
```

```
public class MainActivity extends AppCompatActivity {
    private static final String baseUrl = "https://radiant-forest-10093.herokuapp.com";
    TextView textView;
    Phonebook serverResults;
    GsonBuilder builder;
    PhonebookService service;
    Call<Phonebook> call;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView) findViewById(R.id.result);
        Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(MainActivity.baseUrl)
            .addConverterFactory(GsonConverterFactory.create())
            .build();
        service = retrofit.create(PhonebookService.class);
        call = service.fetchPhonebook();
    }
    public void sendPOST(View view) {
        new MyAsyncTask().execute("");
    }
    public class MyAsyncTask extends AsyncTask<String, String, String> {
        @Override
        protected String doInBackground(String... params) {
            try {
                Response<Phonebook> userResponse = call.execute();
                serverResults = userResponse.body();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return null;
        }
        @Override
        protected void onPostExecute(String result) {
            super.onPostExecute(result);
            textView.setText(serverResults.toString());
        }
    }
}
```


Обратите внимание, что с применением библиотеки **Retrofit**, процесс отправки запроса и получение ответа сократился до двух строк. Кроме того сериализация/десериализация тоже происходит автоматически.



Во второй части нашего вебинара мы продемонстрировали приложение построенное с помощью фреймворка Spring. Этот фреймворк обеспечивает решение многих задач и может использоваться разработчиками почти во всех типах приложений.

Кроме того он обеспечивает Java Enterprise функционал, то есть его можно использовать для построения сложных систем масштаба предприятия наиболее эффективным и правильным образом.

Именно поэтому сегодня востребованы разработчики знающие и умеющие правильно использовать эти инструменты. Однако количество инструментов растет непрерывно, но важно понимать принципы их работы — только тогда их применение будет действительно эффективно, и при появлении нового инструмента вы сможете быстро его освоить.

В приведенной краткой лекции не ставилось цели дать какие то best practices по разработке клиент-серверных приложений или готовые шаблоны разработки. Скорее целью было разобраться в многообразии архитектур, API и фреймворках применяющихся в настоящее время в отрасли создания распределенных выражениях и самое главное — понять как это все устроено и каковы основные принципы работы и построения многозвенных архитектур.

Спасибо!

