J2ME clients for JSON services made easy: the complete implementation

This article shows how to create a complete application (client and server) demonstrating how a J2ME application can be used to access web services encoded with JSON.

03 Mar 2013

Introduction

The Internet is plenty of information about anything. Everybody knows it. However, the big problem remains: it is complicated to learn a new subject when it is composed of an intricate myriad of knowledge snippets, it is not easy to create a global view when you have a poor view of the pieces. And you can not extend your knowledge without this global view, innovating and creating new parts.

In this article we want to face this issue in the field of client/server application for mobile phones. Specifically, we want to make clear how a J2ME application can be used to access web services encoded with JSON (JavaScript Object Notation). No piece will be left out: we will talk about the server and we will construct one using node.js. We will discuss how to test the web service and how to create a complete application for Series 40 phones.

We will construct a simple J2ME application for showing a list of names and their phone numbers (contacts). These contacts are stored in our server and they are available through a call to a web service. By principle, we decided to use the minimal amount of external libraries and the smaller possible code. As a consequence, excellent libraries like Tantalum were not used. We strongly recommend it for further developments, after you have acquired the necessary global view. Databases were excluded by the same reasoning.

The server side is initially discussed in our first section. Then, the client side is detailed in the next section. Finally, both sides are put to work together and some final considerations are raised in the last two sections.

Server Side Code

There are numerous languages for constructing the server side: Java, Python, Ruby, PHP or even a bash script. This decision is not easy and highly biased by your knowledge and project requisites. In this article we decided to use node.js as the basic language for our server. The reason is based on the fact that HTML5 is reality and we could use the same language website development and server code. Moreover, node.js is scalable, fast and have many I/O facilities.

Installing Node.js

The *node.js* installation is straightforward. Just open the nodejs.org site and download the installation program. After installing it, it will be possible to run a customized shell where the environment was prepared to recognize *node.js* and its modules. If you are using Windows, a "Node.js Command Prompt" will be created in your application list. Use it as starting point for the *node.js* usage.

Designing our Contacts Service

In this article a simple web service called 'contacts' will be created. After calling it, a list of names and phone numbers will be returned. Suppose our server is running on IP 10.0.0.104 and port 8080, a request to

```
http://10.0.0.104/v1/contacts/
```

will return a structured dictionary like below, very familiar to Javascript programmers:

```
{ list:[ {"name":"Sophia","phone":"123456789"},{"name":"Isabella","phone":"234567891"},
... ] }
```

Answers are encoded using JSON and UTF-8 as character set (charset). JSON is an open standard designed for data interchange but text-based and human-readable. It is highly used for data serialization over network connections and it have been implemented for numerous languages. By its turn, UTF8 is likely the dominant charset of web applications.

Contacts Server Code

(C) Copyright Nokia 2014. All rights reserved.

In order to create such service, a small node.js application need to be written. This application will employ a node.js module called express.js. Express.js is a web application framework, making the construction of web services really simple. In our cases, express.js will be used to gather the calls to the contacts web service (our router) and to provide a consistent response based on JSON.

Express.js is used below to answer to the GET method when requesting `/v1/contacts/`. The syntax is clear, using an express object (`app`), the request method (`get`) and an inner function to create the answer:

```
app.get('/v1/contacts/', function (req, res) {
    var data = { list:contacts };
    res.send(JSON.stringify(data));
});
```

`req` stores incoming parameters and methods related to the current request and `res` has the same role but for output. A dictionary is created (data) and sent to the client in JSON format (`stringify` method of JSON, native in JavaScript). At this point, it is recommended a deeper Internet search about node.js and express.js. For reference, the complete server code is placed here, with comments. Save it as **app.js** inside an empty directory.

```
// Imports the express module and create an instance of it (app)
var express = require('express');
var app = express();

// Our 'embedded' database
var contacts = [
 { name:'Sophia', phone:'123456789' },{ name:'Isabella', phone:'234567891' },
 { name:'Olivia', phone:'345678912' },{ name:'Emily', phone:'456789123' },
 { name:'Elizabeth', phone:'567891234' },{ name:'Natalie', phone:'678912345' },
 { name:'Samantha', phone:'789123456' },{ name:'Victoria', phone:'891234567' },
 { name:'Hannah', phone:'912345678' },{ name:'Charlotte', phone:'012345678' },
 { name:'Layla', phone:'876543210' },
];

// Guarantee that all responses should be encoded using JSON
// since this function is called to set it when a new response starts.
app.use(function(req, res, next) {
 res.contentType('application/json');
 next();
});

// Our web service for contacts list
app.get('/v1/contacts/', function (req, res) {
    var data = { list:contacts };
    res.send(JSON.stringify(data));
});

// Generic answer for other requests
app.get('/*', function(req, res) {
 var data  = { list:[] };
 res.send(JSON.stringify(data));
});

// Start the server at this port
app.listen(8080);
```

Running the Server

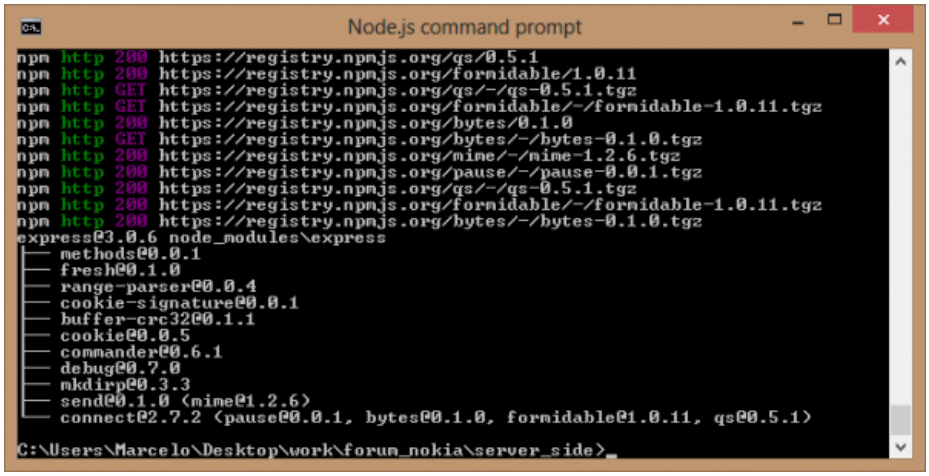
 (C) Copyright Nokia 2014. All rights reserved.

The next step is to deploy our application. In **node.js** this is done by writing a module description called **package.json** where versions and dependencies are listed as shown below. Save the following file together **app.js**.

```
{
    "name": "j2me-json-server",
    "version": "1.0.0",
    "private": true,
    "dependencies": {
        "express": "3.0.x"
    }
}
```

Node.js can read this package and download all dependencies, creating a self contained directory structure. To do so, just open the node.js command prompt, go to package.json directory and type:

```
npm install
```



Finally, you can run your server:

```
node app.js
```

From this point your web service is up and waiting for requests. You can test it using programs like `curl`:

```
C:\Users\Marcelo\bin>curl http://10.0.0.104:8080/v1/friends/
```

```
{C:\Users\Marcelo\bin>curl.exe http://127.0.0.1:8080/v1/contacts/
{"list":[{"name":"Sophia","phone":"123456789"},{"name":"Isabella","phone":"234567891"},
{"name":"Olivia","phone":"345678912"},{"name":"Emily","phone":"456789123"},
{"name":"Elizabeth","phone":"567891234"},{"name":"Natalie","phone":"678912345"},
{"name":"Samantha","phone":"789123456"},{"name":"Victoria","phone":"891234567"},
{"name":"Hannah","phone":"912345678"},{"name":"Charlotte","phone":"012345678"},
{"name":"Layla","phone":"876543210"}]}
```

That is all. Your web service is up and running properly. Time to create the J2ME application.

Client Side Code

We created a simple but easily extensible J2ME application based on the JSON decoder available at json.org. There is a general flow for any API call although only one had been implemented (`getContatcsList`). This flow is described below:

 (C) Copyright Nokia 2014. All rights reserved.

- a call to `getContatcsList()` is done, passing a model as argument to be filled (`ContactModel`). `ContactModel` encapsulates a Vector object for holding the retrieved items and implements the interface `IEntry`. It is from this interface that the JSON decoder is dispatched later, using the method `fromJSONObject()`.

```
// Midlet.java
ContactAPI api = ContactAPI.getInstance();
ContactModel model = new ContactModel();
api.getContatcsList(model)
```

- `getContatcsList()` creates an HTTP request, passing a response handler to it. In our case, the request handler is called `APIResponse`.

```
// ContactAPI.java
public boolean getContatcsList(ContactModel model) {
    APIResponse handler = new APIResponse(model);
    return HttpApiRequest(API_URL, handler);
}
```

- This response handler implements the interface `IHTTPResponseHandler` for creating standard answers for HTTP code 200 (`Response20X()`) or any other code (`ResponseOthers()`). When we do not have errors `Response20X()` is responsible for collecting the answer and for decoding it.

```
// ContactAPI.java
private boolean HttpApiRequest(String url, IHTTPResponseHandler handler) {
// ...
    int code = connection.getResponseCode();
    if (code == HttpConnection.HTTP_OK) {
        result = handler.Response20X(connection);
    } else {
        result = handler.ResponseOthers(connection);
    }
//  ...
```

- The decoding is possible since `APIResponse` has a reference to `IEntry`, implemented by the data model.

```
// APIResponse.java
public boolean Response20X(HttpConnection connection) {
// ...
    try {
        JSONObject jobj = new JSONObject(response);
        result = data.fromJSONObject(jobj); // see ContactModel.java
        result = true;
    } catch (JSONException ex) {
        result = false;
    }
// ...
```

- when decoding the JSON object, `ContactModel` create `ContactEntry` objects and fill them using `IEntry` interface, also implemented by `ContactEntry`.

```
    // ContactModel.java
    data = new Vector();
```

 (C) Copyright Nokia 2014. All rights reserved.

```
    JSONArray friendsList = obj.getJSONArray("list");
    if (friendsList != null) {
        for (int n = 0; n < friendsList.length(); n++) {
            JSONObject entry = friendsList.getJSONObject(n);
            ContactEntry e = new ContactEntry();
            e.fromJSONObject(entry); // see ContactEntry.java
            data.addElement(e);
        }
    }
```

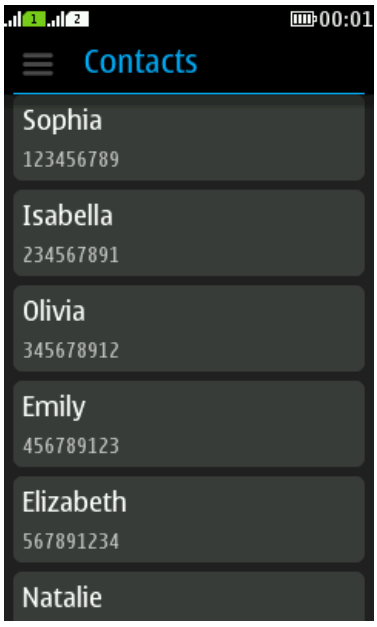
```
    // ContactEntry.java
    public boolean fromJSONObject(JSONObject obj) {
        // ...
            setName(obj.getString("name"));
            setPhone(obj.getString("phone"));
            result = true;
        // ...
    }
```

This way, from the API client point of view (our Midlet), there is only a simple API call returning true or false and filling a model when things go well. All the hard work of connecting, fetching, decoding are kept in other code domain.Thus, the Midlet iterates over the model and creates several StringItems inside its form:

```
    // Midlet.java
    if (api.getContatcsList(model)) {
        for (int n = 0; n < model.size(); n++) {
            ContactEntry e = model.elementAt(n);
            contacts.append(new StringItem(e.getName(),e.getPhone()));
        }
    }
```

Putting all Together

Since your server is running, just compile the available Netbeans projects and run it. It may be necessary to adjust the server IP (see `API_URL` inside `ContactAPI.java`). In this implementation the Java SDK 2.0 was used the but older SDKs should work as well. The complete code is available File:J2ME clients for JSON services.zip for download (server and client).



 (C) Copyright Nokia 2014. All rights reserved.

Enhancements

If you want to implement more web services calls the next steps are required:

- Client:
  1. create a new entry class and implement `IEntry` interface
  2. create a model class for holding a collection these entry objects and implement the `IEntry` interface
  3. create or reuse an `APIResponse` and implement the `IHTTPResponseHandler` interface
  4. create a new method for this service inside `ContactAPI`
- Server
  1. implements the new web service using express.js

And, as expected, many improvements can be done:

- use a real database instead our embedded database
- use Tantalum library for JSON and HTTP requests
- create a view controller besides the model
- implement more response codes for better error handling (30X, for instance)

 (C) Copyright Nokia 2014. All rights reserved.