Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of theoretical computer science

Master's thesis

# Admission procedure Automatic processing of applications for master's study program

***Bc. Ján Ondrušek***

Supervisor: Ing. Tomáš Kadlec

8th June 2012

# Acknowledgements

I would like to thank my family and friends for support during writing this thesis.

# Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act 60 no. 121/2000 (copyright law), and with the rights connected with the copyright act included the changes in the act.

In Prague 8th June 2012 .....

Czech Technical University in Prague
Faculty of Information Technology


## Citation of this thesis

Ján Ondrušek. *Admission procedure Automatic processing of applications for master's study program: Master's thesis.* Czech Republic: Czech Technical University in Prague, Faculty of Information Technology, 2012.

# Abstract

Primary aim of this thesis is to analyse Conditions for admission and Dean's directive for admission process to master's study programme at CTU FIT. Implement RESTful API, which exposes backend functionality for admission processing using Business Process Management.

**Keywords** Admission procedure, RESTful API, BPM, jBPM, Spring, Spring Roo

# Abstrakt

Primárnym cieľom tejto diplomovej práce je analyzovať Řád přijímacího řízení ČVUT a Směrnici děkana pro přijímací řízení na ČVUT Fakultě informačních technologií. Implementovať RESTful API, ktoré vystaví funkcionalitu backendu pre prijímací proces s použitím Business Process Management stroja.

**Klíčová slova** Spracovanie prihlášok, RESTful API, BPM, jBPM, Spring, Spring Roo

# Contents

# List of Figures

# List of Tables

# Prologue

## Motivation and objectives

Every year, hundreds of high school graduates apply for studies at Czech Technical University, Faculty of Informatics. This raises certain requirements, including managing, storing, analysing and processing of all these applications. Each application has its own life cycle, which begins with filling out an on-line form and continues through various steps which an applicant has to pass. The life cycle ends when a decision of acceptance is delivered to the applicant and he either enrolls in the studies or not.

We live in the world of new era of the Internet. Everything goes on-line, web and the latest trend - everything goes mobile. People want things to happen very quickly. They want to access all the information fast, now.

Students and applicants are no different. They expect from this prestigious University, especially from Faculty of Informatics, the most modern and useful gadgets when it comes to software and web.

## How do things work now

Currently all applications are processed rather manually. Many man days of administrative work are consumed during the process. Although an electronic form is filled in and submitted by an applicant, the rest of actions almost exclusively fall into the hands of Study Department staff. Some of the work is handled by simple scripts or other utilities. The question is: Why don't we do most of the work automatically?

This work is monotonous and can even lead to men's frustration.

## What should be achieved - the goals

Courses at Faculty of Informatics teach its students to handle various programming languages, web technologies and techniques. We all know what to expect from a working web application and good looking one is a bonus. This is why knowledge of faculty's students should be used for good of their suc-

cessors. Fast, reliable, informative and functional system will make them feel more comfortable and perhaps could even save some precious time.

Ideal state would be to accept on-line applications and automatically generate invitations for applicants, that should attend a test. After the test, process all results and generate a decision of acceptance letter for all who passed the test or are accepted without it. The only manual interventions that will remain is to accept apology, appeal and insert the letters into the envelopes.

Pragmatically goals of this thesis could be summarized as follows:

- familiarise with RESTful best practices, patterns and anti-patterns
- familiarise with Business process management (BPM) with main focus on jBPM
- implement RESTful Application Programming Interface (API) (back-end) according to functionality requested by Android and Web UI teams
- implement admission processing using Java and jBPM processing machine
- explore new and modern Java (JEE) technologies
- follow modern development methodologies
- perform tests during and after development
- use exclusively Open Source software and tools

## Let's make things better

Taking the above written into account, this might be a good idea for a master's or bachelor's thesis. However if we want to use all available technologies that have become popular in past years and automatize the majority of admission processing, it turns out to be a very complex project. So why not to create several teams and split necessary work into multiple, both bachelor's and master's, thesis?

This is how project Přiříz was born. It includes web interface for both students and Study Department staff, native Android application and RESTful API with BPM processing machine, which is the subject of my master's thesis.

## Structure of this work

Basically, I could divide my work into these main parts, which are then further split into chapters:

- Theoretical introduction is covered by chapters 1 and 2. Following parts will largely draw on the information contained here.
- Analytical part talks about architecture of the whole ecosystem with main focus on RESTful API. Describes technologies used, methodologies applied and tools commonly used during development and testing phases.
- Implementation and unit testing
- Integration and regression testing
- Results and conclusion

Appendices at the end of the document are referred directly from the text within the chapters. Smaller figures, tables or other objects are put directly into the content.

CHAPTER 1

# RESTful API with JAX-RS

Nowadays, Internet consumers demand fast growth of various services and integration of their favourite ones. As en example I can point out synchronization of contact list between very popular social networks, e-mail providers and phone contact lists.

Other example may be growing amount of `mashups`[1] and uncountable number of `startups`[2], who often provide RESTful or different type of public API.

## 1.1 Talking about REST, what is it?

REpresentational State Transfer (REST) or RESTful programming is not defined by any official standard and there are no official guidelines or rules for it. So what is it then? It is and architectural and programming style for Web, where a set of constraints is defined. Lots of text has been written about it during past years and describing the whole idea of REST is out of scope of this master's thesis. I can however try to point out the most significant, important and basically, what I personally managed to adopt.

### 1.1.1 Main principles of REST, RESTful web service

There are several architectural principles that one should keep in mind when thinking of REST [1, p. 3]:

[1] Applications that are created via combination of multiple different services. Such application, almost exclusively web based, can be created very quickly by consuming several APIs. Not necessarily from the same provider.

[2] Constantly rising amount of web applications that focus on fast growth of attracted users. They offer various services, which are often very innovative and experimental. One successful example is popular social network and my favorite information channel - Twitter.

- **Addressable resources** The key abstraction of information and data in REST is a resource, and each resource must be addressable via a Uniform Resource Identifier (URI).
- **A uniform, constrained interface** Use a small set of well-defined methods to manipulate your resources.
- **Representation-oriented** You interact with services using representations of that service. A resource referenced by one URI can have different formats. Different platforms need different formats. For example, browsers need HTML, JavaScript needs JSON (JavaScript Object Notation), and a Java application may need XML.
- **Communicate statelessly** Stateless applications are easier to scale.
- **Hypermedia As The Engine Of Application State (HATEOAS)** Let your data formats drive state transitions in your applications.

HATEOAS is often understood as a core principle of REST. It carries an idea of resource representation via links and stateless implementation of services.

RESTful web services are the result of applying these constraints to services that utilize web standards such as URIs, Hypertext Transfer Protocol (HTTP), Extensible Markup Language (XML), and JavaScript Object Notation (JSON).

### 1.1.2 Back to the roots, HTTP is reborn

Service-oriented architecture (SOA) has been in this world for a long time. Many different approaches and technologies exist to implement it. From those worth to mention: DCE, CORBA, Java RMI, ... They offer robust standards, one can build large, complex and scalable systems on top of it, but there is a cost. They often bring huge complexity and maintenance requirements into place.

Currently, when one says SOA, it often evokes Simple Object Access Protocol (SOAP) in a mind that spent several years using technologies mentioned above. This however is not a bad thing. SOAP is used very widely and is perfectly suitable for developing services and APIs. But it is definitely not a lightweight technology and it is not ideal for everything. Its most common use case is for server-server communication in enterprise systems.

Nowadays, we need something quickly adoptable, widely spreadable, platform and technology independent and client oriented. This needs a completely different approach and new way of thinking when it comes to SOA. It is about Web, so why not to start with something that is Web, as we see it today, based on? Yes, it is HTTP.

Although REST is not protocol specific, when saying REST it usually automatically means REST + HTTP. No wonder. HTTP is perfectly suitable for client-server SOA, it is just about the way of thinking. It offers transport layer, request-response mechanism, descriptive responses, caching mechanism and many more. It is true that in past years, when various types of web applications started to appear, many web developers limited their thinking and use of HTTP to two basic cases:

- GET a page with URI, perhaps containing a few query parameters
- POST a form

**Code 1** HTTP GET request/response example of a standard web page

```
GET /index.html HTTP/1.1
User-Agent: curl/7.24.0 (i686-pc-cygwin) ...
Host: www.google.sk
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: sk-sk,en;q=0.5

HTTP/1.1 200 OK
Date: Thu, 07 Jun 2012 11:25:15 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-2
Set-Cookie: ... expires= ...; path=/; domain=.google.sk
Set-Cookie: ... expires= ...; path=/; domain=.google.sk; HttpOnly
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked

<!doctype html><html ...><head> ... <body> ...
```

The example above 1 shows most common HTTP request and response, when browsing the web via standard web browser. It requests object **/index.html** using **GET** method placed on host **www.google.sk**. My client also put several HTTP headers into the request:

- **Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
- **Accept-Language:** sk-sk,en;q=0.5

Also the request does not contain any request body, as it is GETting information from the server.

The response of the message received is 200, which means OK - success. An overview of all available HTTP response codes can be found on-line at [5]. **Content-type** header of the response message says that the body received is of type HTML.

RESTful web service needs more than that and luckily HTTP offers much more. It will be better to point out its features in a relationship to each of REST's architectural principles.

### 1.1.3 Addressable resources, URIs and links

Each **resource** in a system should be reachable through a **unique identifier**. When reflected to the idea of REST and HTTP, URIs will automatically come to mind. The format of a URI looks like this [6]:

```
<scheme>://<authority><path>?<query>
```

The above is a citation directly from RFC, but for purposes of this master's thesis should be rewritten into more detailed form:

```
<scheme>://<host>:<port>/<path>?<queryString>#<fragment>
```

Where in the RESTful world these parts usually mean:

- **scheme** typically **http** or **https**
- **host** aka server name, e.g. **fit.cvut.cz** and **port**
- **path** to the resource on server, e.g. **/admission/123-456-01**
- **queryString** after the **?** is typically used for a set of resources and can be a page number, number of items in the set, or a filter definition and many more, e.g. **?page=1&limit=10**
- **fragment** after the # usually points to a certain place in a document

An example of such URI pointing to a set of resources may be:

```
http://pririz.is.fit.cvut.cz:9090/admission/services/admission
?page=2&count=20
```

An example of URI pointing to a concrete single resource:

```
http://pririz.is.fit.cvut.cz:9090/admission/services/admission
/123-456-01
```

Characters allowed in URI string are all alphanumeric, comma, dash, asterisk and underscore. Space is converted into plus and other characters are encoded using specific schema into a two digit hexadecimal number, which is appended to the % character.

### 1.1.4 The uniform, constrained interface, HTTP methods

This is probably the prettiest part of a relationship between REST and HTTP. It may be a bit difficult to adopt this principle for a person, who spent a couple of years developing CORBA or SOAP services. There is a finite set of HTTP methods and all REST operations have to stick to it. All other parameters describing operations must be omitted from URI.

Let's see what HTTP offers:

| Method | `Idempotent`[a] | `Safe`[b] | Operation(s) |
|---|---|---|---|
| GET | yes | yes | read - query information from a server |
| POST | no | no | write, update - both can change a server state in a unique way each time executed |
| PUT | yes | no | update for the known resource, updating the same resource more than once does not effect it |
| DELETE | yes | no | delete - removes resource |
| HEAD | yes | yes | read without response body, returns only response headers |
| OPTIONS | yes | yes | information about communication options with server |

[a]Idempotent means that no matter how many times you apply the operation, the result is always the same.

[b]Safe means that invoking an operation does not change the state of the server at all. This means that, other than request load, the operation will not affect the server.

Table 1.1: An overview of HTTP methods and their roles in a RESTful service

HTTP contains a few other methods (TRACE, CONNECT), which are unimportant for purposes of RESTful services.

What is more interesting, nowadays a couple of non-HTTP methods have appeared, which may be good for RESTful service design in the future. Namely PATCH (very similar to the PATCH method found in `WebDAV`[3]) and MERGE. According to various sources I have found on the web, namely [3], [4] or [2], they may appear in further HTTP specifications. They are not a part of current HTTP 1.1.

PATCH and MERGE are used for partial update of known resources that contain large amount od data and updating the whole object would be a

[3]WebDAV stands for Web-based Distributed Authoring and Versioning. It is a set of extensions to the HTTP protocol which allows users to collaboratively edit and manage files on remote web servers.

lengthy and ineffective operation. This however can be simulated using POST and specifying detailed path of the resource via URI.

### 1.1.5 Representation-oriented

I already described that each resource has its own URI and client-server principle using HTTP. Its methods allow the client to receive current representation via GET method, remove it from server via DELETE or change the representation via POST and PUT methods. Concrete representation can be received in JSON, XML, YAML or any other format one can imagine.

Representation format is agreed between client and server in a RESTful system interaction. HTTP offers such feature by specifying **Content-Type** header. Its value string is represented by Multipurpose Internet Mail Extensions (MIME) format:

```
<type>/<subtype>[;name=value;name=value...]
```

An example may be:

```
text/html; charset=utf-8
text/xml
text/json
application/xml
application/json
```

To choose preferred format(s), client can specify **Accept** HTTP header in a request. Now it becomes more clear how REST and HTTP perfectly fit each other. Together they offer addresability, method choice and object representation format.

### 1.1.6 Communicate statelessly, (no)sessions

HTTP offers powerful client session management, which is commonly used when browsing the web via traditional web browser. It stores so called **Cookies** when a server asks for it in response headers, which are then sent back to the server with subsequent requests. This is how the server handles stateful interaction with the client over HTTP.

Stateless communication in REST means that there is no client session data stored on a server. In other words, none of the above is performed. It does not mean that RESTful application cannot be stateful, though.

Reason for this is simplicity, which further leads to easily scalable RESTful service. It is generally much less difficult to build a cluster of stateless applications than to handle session replication and possibly another service layer.

### 1.1.7 HATEOAS

[1, p. 11] Hypermedia is a document-centric approach with the added support for embedding links to other services and information within that document format.

There are several ways how to understand any apply this RESTful architectural principle. One use case is to add hyperlinks when composing complex and large objects. This avoids unexpected server load, delay in response to the client and helps to reference dependent or embedded objects without bloating the response.

An example of server response without any hyperlinks:

```
<terms>
  <term>
    ...
    <registrations>
      <registration>
        <admission>
          <code>96858805</code>
          <type>P</type>
          <accepted>false</accepted>
          ... some huge object containing
          a lot of information
        </admission>
        <term>
          <dateOfTerm>2012-05-10...</dateOfTerm>
          <room>BS</room>
          <capacity>1500</capacity>
          <registerFrom>2012-05-03...</registerFrom>
          <registerTo>2012-05-08...</registerTo>
          <apologyTo>2012-05-08...</apologyTo>
          ... another huge object containing
          a lot of information, graph can be even circular
        </term>
      </registration>
      ...
    </registrations>
    ...
  </term>
  ...
</terms>
```

Let's apply embedded hyperlinks on the document above:

```
<terms>
  <term>
```

```
    ...
    <registrations>
      <registration>
        <admission>
          <link href="http://.../admission/96858805"
          method="GET" rel="admission" />
        </admission>
        <term>
          <link href="http://.../term
          /dateOfTerm:2012-05-10T14:22:00/room:BS"
            method="GET" rel="term" />
        </term>
      </registration>
      ...
    </registrations>
    ...
  </term>
  ...
</terms>
```

This concept of HATEOAS is called aggregation. But it isn't everything. In a case that the server would return thousands of term objects and each of them would contain thousands of registrations including admissions, the response would be, again, very large. However, there is one even more interesting part of HATEOAS - the „engine".

Core of the engine principle is not to return the whole set of object available, but just a subset of it and to tell the client, where to find the rest:

```
<terms>
  <count>5</count>
  <totalCount>123</totalCount>
  <link href="http://.../term?page=3&count=5" method="GET"
    rel="next" />
  <link href="http://.../term?page=1&count=5" method="GET"
    rel="previous" />
  <term>
    ...
    <registrations>
      <registration>
        <admission>
          <accepted>false</accepted>
          <link href="http://.../admission/96858805"
            method="GET" rel="admission" />
        </admission>
        <term>
```

```
          <link href="http://.../term
          /dateOfTerm:2012-05-10T14:22:00/room:BS"
            method="GET" rel="term" />
        </term>
      </registration>
      ...
    </registrations>
    ...
  </term>
  ...
</terms>
```

This approach saves a lot of server resources and prevents client from unexpected delays due to large responses. Such response should be always returned in constant time, because the request query defines its maximum size - number of objects.

## 1.2 REST, Java and JAX-RS

CHAPTER 2

# BPM and jBPM

CHAPTER 3

# Chosen technologies

## 3.1 Why not SOAP?

## 3.2 REST vs. SOAP

# Bibliography


[1] Burke, B.: *Restful Java with Jax-RS*. O'Reilly Media, 2009.

[2] Compaq/W3C; W3C/MIT; Xerox; etc.: Request for Comments: 2616. June 1999. Available at WWW: <http://www.ietf.org/rfc/rfc2616.txt>

[3] Lab, L.: PATCH Method for HTTP draft-dusseault-http-patch-16. November 2009. Available at WWW: <http://tools.ietf.org/html/draft-dusseault-http-patch-16>

[4] Microsoft: MSDN 2.2.4.1 PATCH/MERGE. Available at WWW: <http://msdn.microsoft.com/en-us/library/dd541276(v=prot.10).aspx>

[5] RFC 2616 Fielding, e. a.: part of Hypertext Transfer Protocol – HTTP/1.1. Available at WWW: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

[6] Xerox: Request for Comments: 2396. August 1998. Available at WWW: <http://www.ietf.org/rfc/rfc2396.txt>

APPENDIX A

# Content of CD

| | |
|---|---|
| readme.txt | - the file with CD content description |
| | |
| data/ | - the data files directory |
| graphs/ | - the directory of graphs of experiments |
| *.eps | - the B/W graphs in PS format |
| *.png | - the color graphs in PNG format |
| *.dat | - the graphs data files |
| | |
| exe/ | - the directory with executable WBDCM program |
| wbdcm | - the WBDCM program executable (UNIX) |
| wbdcm.exe | - the WBDCM program executable (MS Windows) |
| | |
| src/ | - the directory of source codes |
| wbdcm/ | - the directory of WBDCM program |
| Makefile | - the makefile of WBDCM program (UNIX) |
| thesis/ | - the directory of LaTeX source codes of the thesis |
| figures/ | - the thesis figures directory |
| *.eps | - the figures in PS format |
| *.pdf | - the figures in PDF format |
| *.tex | - the LaTeX source code files of the thesis |
| | |
| text/ | - the thesis directory |
| thesis.pdf | - the Diploma thesis in PDF format |
| thesis.ps | - the Diploma thesis in PS format |