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

28 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 28 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, 2013.

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 consists of chapters 3 and 4. 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. Theoretical and analytical parts together partially form Feasibility Study.
- Implementation and testing (unit, integration and regression) is covered by chapters 5 and 6. They form so called Detailed Design.
- Results and conclusion is a final chapter of this work 7.

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 and 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 [3, 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 [19]. **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 [22]:

```
<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 [8], [12] or [4], 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 of data and updating the whole object would be a lengthy

[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.

and ineffective operation. This, however, can be simulated using POST and specifying a 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 addressability, 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

[3, 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 and 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

[3, p. xiii] The Java API for RESTful Web Services (JAX-RS) is a new API that aims to make development of RESTful web services in Java simple and intuitive. The initial impetus for the API came from the observation that existing Java Web APIs were generally either:

- Very low-level, leaving the developer to do a lot of repetitive and error-prone work such as URI parsing and content negotiation, or
- Rather high-level and proscriptive, making it easy to build services that conform to a particular pattern but lacking the necessary flexibility to tackle more general problems.

JAX-RS is one of the latest generations of Java APIs that make use of Java annotations to reduce the need for standard base classes, implementing required interfaces, and out-of-band configuration files. Annotations are used to route client requests to matching Java class methods and declaratively map request data to the parameters of those methods. Annotations are also used to provide static metadata to create responses.

JAX-RS also provides more traditional classes and interfaces for dynamic access to request data and for customizing responses.

This is a brief description of what JAX-RS is. Its usage and development approach will be demonstrated in following parts of this master's thesis.

CHAPTER 2

BPM and jBPM

[6] Business process modeling (BPM), sometimes called business process management, refers to the design and execution of business processes.

It does not have to be necessarily used in a context of Information Technology (IT) and software development. Primary field of BPM falls into management, though. Before Information and communication technologies (ICT) was widely spread and automatic software processing was a dream, BPM was manual and paper driven.

But yes, BPM is also closely aligned with the notion of SOA, particularly the emerging W3C web services stack. Whereas the traditional use of a workflow was about the movement of work from person to person within an organization, contemporary BPM processes are built to interact as services with other systems, or even to orchestrate or choreograph other systems, including the business processes of other companies.

2.1 Business Process

A business process is a service, one intended to be called by other systems, and these calls drive its execution. Realizing this fact is one of the first big steps in understanding BPM.

Being algorithmic, a process can potentially be run by some sort of process engine. As long as the process can be expressed in a form that is syntactically and semantically unambiguous, that is, in a programming language or other interpretable form, the engine can accept it as input, set it in motion, and drive its flow of control. To be precise, the engine creates and runs instances of a given process definition. The steps of the process are called activities or tasks.

2.2 Important process modelling terms

- **Process definition** The basic algorithm or behavior of the process.
- **Process instance** An occurrence of a process for specific input. Each instance of the travel reservation process, for example, is tied to a specific customer's itinerary.
- **Activity or task** A step in a process, such as sending a flight request to the airline.
- **Automated activity or automated task** A step in a process that is performed directly by the execution engine.
- **Manual activity or manual task** A step in a process that is meant to be performed by a human process participant.

The distinction between manual and automated activities is extremely important. At one time, before the reign of software, a business process was completely manual and paper-driven: paper was passed from person to person, and was often misplaced or delayed along the way. Now, much of the process runs on autopilot.

Automated activities generally fall into two categories:

- **Interactions with external systems** e.g., sending a booking request to an airline
- **Arbitrary programmatic logic** e.g., calculating the priority of a manual task

2.3 BPM application

BPM is suited only for applications with an essential sense of state or process, that is, applications that are process-oriented. Typical characteristics of a process-oriented application are:

- **Long-running** From start to finish, the process spans hours, days, weeks, months, or more.
- **Persisted state** Because the process is long-lived, its state is persisted to a database so that it outlasts the server hosting it.
- **Bursty, sleeps most of the time** The process spends most of its time asleep, waiting for the next triggering event to occur, at which point it wakes up and performs a flurry of activities.
- **Orchestration of system or human communications** The process is responsible for managing and coordinating the communications of various system or human actors.

2.4 BPM standards

BPM is not that difficult. The business analyst designs the process, the process is run by an engine, and the engine has EAI and human interaction capabilities. Questions appear when it comes to selection of a right solution, design and modelling tools, runtime engine, ...

It is good to look for some well known and widely accepted approach. BPM does not lack standards. The adoption of Business Process Execution Language (BPEL) and Business Process Model and Notation (BPMN) is a recipe for success. According to [6, Ch. 1.3.1], the important BPM standards are:

- Business Process Execution Language for Web Services (BPEL4WS), sometimes shortened to **BPEL**

  A BPEL process is a web service with an associated process definition defined in an XML-based language. The behavior of a BPEL process is to act on, and be acted on by, other processes; put differently, a BPEL process can invoke another web service or be invoked as a web service.

- **Business Process Modeling Language (BPML)**

  From the Business Process Modeling Initiative (BPMI) organization, which is an XML-based process definition language similar to BPEL. BPMN, another specification from BPMI, is a sophisticated graphical notation language for processes. Significantly, the BPMN specification includes a mapping to BPML-rival BPEL, which facilitates the execution of BPMN-designed processes on BPEL engine.

- **Web services choreography**

  Choreography describes, from a global point of view, how web services are arranged in a control view spanning multiple participants. Choreography's global view is contrasted with the local view of process orchestration in languages such as BPEL; a BPEL process is the process of a single participant, and a choreography is the interaction model for a group of participants. Web Services Choreography Description Language (WS-CDL) is the W3C's recommended choreography standard.

- Workflow Management Coalition (WfMC) has published a BPM reference model, as well as a set of interfaces for various parts of the BPM architecture. Though WfMC does not specify a standard graphical process notation, it does provide an exportable XML format called XML Process Definition Language (XPDL); processes built in an XPDL-compliant design tool can run on a WfMC enactment engine.

- **Object Management Group (OMG)** did not aim to build a new process language or interface but abstract BPM models conforming to its Model-Driven Architecture (MDA).

- **Business Process Specification System (BPSS)** from OASIS group

  A choreography language, but is built for business-to-business collaborations. In a typical exchange between a buyer and seller, for example, the buyer sends a request to the seller, to which the seller responds immediately with consecutive acknowledgements of receipt and acceptance. When the seller has finished processing the request, it sends an indication of this to the buyer, and the buyer in turn sends an acknowledgement of the indication to the seller.

From the above are only two standards worth considering. BPEL and BPMN, which I am going to describe in analytical part of this work.

2.5 Execution engine - jBPM

[21] jBPM is a flexible BPM Suite. It's light-weight, fully open-source (distributed under Apache license) and written in Java. It allows to model, execute and monitor business processes, throughout their life cycle.

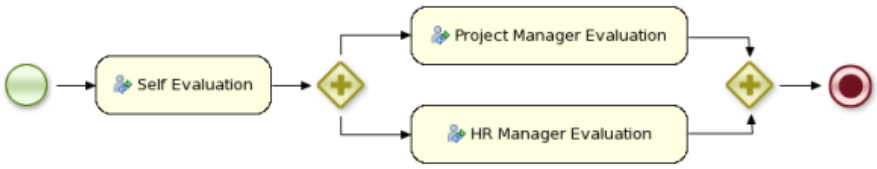

Figure 2.1: Example jBPM process as shown in [21]

A business process allows to model various business goals by describing the steps that need to be executed to achieve that goal and the order, using a flow chart. This greatly improves the visibility and agility of a business logic. jBPM focuses on executable business process, which are business processes that contain enough detail so they can actually be executed on a BPM engine. Executable business processes bridge the gap between business users and developers, as they are higher-level and use domain-specific concepts that are understood by business users but can also be executed directly.

The core of jBPM is a light-weight, extensible workflow engine written in pure Java that allows to execute business processes using the latest BPMN 2.0 specification. It can run in any Java environment, embedded in an application or as a service.

BPM makes the bridge between business analysts, developers and end users, by offering process management features and tools in a way that both

business users and developers like. Domain-specific nodes can be plugged into the palette, making the processes more easily understood by business users.

jBPM supports adaptive and dynamic processes that require flexibility to model complex, real-life situations that cannot easily be described using a rigid process. jBPM is also not just an isolated process engine. Complex business logic can be modeled as a combination of business processes with business rules and complex event processing. jBPM can be combined with the Drools project to support one unified environment that integrates these paradigms where business logic can be modelled as a combination of processes, rules and events.

Apart from the core engine itself, there are a few optional components that can be used. Eclipse-based or web-based business process designer and a management console for the execution engine.

2.5.1 jBPM Core Engine

The core jBPM engine is the heart of the jBPM project. It's a light-weight workflow engine that executes business processes. It can be embedded as a part of an application or deployed as a service. It's most important features are:

- Solid, stable core engine for executing process instances
- Native support for the latest BPMN 2.0 specification for modeling and executing business processes
- Strong focus on performance and scalability
- Light-weight (can be deployed on almost any device that supports a simple Java Runtime Environment, does not require any web container at all)
- (Optional) pluggable persistence with a default JPA implementation
- Pluggable transaction support with a default JTA implementation
- Implemented as a generic process engine, so it can be extended to support new node types or other process languages
- Listeners to be notified of various events
- Ability to migrate running process instances to a new version of their process definition

The core engine can also be integrated with a few other (independent) core services:

- The human task service can be used to manage human tasks when human actors need to participate in the process. It is fully pluggable and the default implementation is based on the WS-HumanTask specification and manages the life cycle of the tasks, task lists, task forms and some more advanced features like escalation, delegation, rule-based assignments, etc.

- The history log can store all information about the execution of all the processes on the engine. This is necessary if access to historic information is needed, as runtime persistence only stores the current state of all active process instances. The history log can be used to store all current and historic state of active and completed process instances.

  It can be used to query for any information related to the execution of process instances, for monitoring, analysis, etc.

CHAPTER 3

Přiříz architecture and requirements

Previous chapters should provide enough information to understand basic principles of a RESTful service and benefits of using BPM. Discussion about the two mentioned will become more concrete from now on, because admission process will be taken into account.

As a result of this master's thesis a working application is implemented. Further referred as **RESTful API** and it integrates both technologies.

This chapter describes requirements for application functionality and its role in the architecture of the whole project currently running at CTU FIT.

3.1 Faculty Information System

The Faculty Information System (FIS) project covers activities related to information system development CTU. It is quite complicated and long term project involving more than dozen people in various roles. RESTful API with two other applications directly interconnected is just a part of it and together they form **Přiříz** component 3.1.

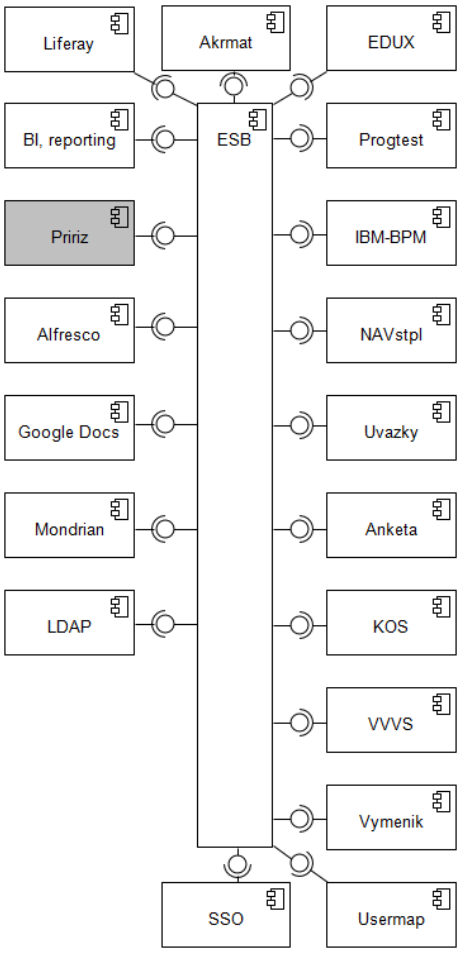

Figure 3.1: FIS architecture

3.2 Catalogue of requirements

This section lists requirements for the RESTful API.

3.2.1 Functional requirements

Functional requirements define which services should system provide.

- F00 Admission import
  data import from e-admission, on-line CTU form
- F01 Admission evidence
  evidence of valid admissions

- F01.1 Admission detail

  detailed information about admission and admissioner

- F01.2 Admission edit

  update of allowed admission data

- F01.3 Password reset

  allow password reset/recovery to a valid user account related to the admission

- F02 Term evidence

  entrance exam term evidence

- F02.1 Term management

  entrance exam term management

- F02.2 Enrollment term management

  enrollment term of accepted admissioners management

- F02.3 Term registrations evidence

  Entrance exam or enrollment term registrations of admissionsers evidence

- F03 Statistics

  various statistics of this year's admission process

- F04 User management

  password change

- F05 Admission state

  view current state of admission

- F05.1 Entrance exam registration

  allow an admissioner to book entrance exam term

- F05.2 Entrance exam apology

  allow an admissioner to apologise from the entrance exam registration

- F05.3 Enrollment registration

  allow an admissioner to book enrollment term

- F05.4 Enrollment apology

  allow an admissioner to apologise from the enrollment registration

- F06 Admission process management

  allow management of an admission during admission process

- F06.1 Send e-mails

  allow sending of informative e-mail

- F06.2 User action processing

  allow processing of user actions during admission process

- F07 File number administration

  allow file number assignment to documents communicated with an admissioner

- F08 Document creation

  allow document creation of various types for an admissioner

3.2.2 Non-functional requirements

Non-functional requirements do not directly define system functionality, but describe constraints or general properties.

- NF01 User action processing

  jBPM engine requires local TaskService server to handle user actions in process

- NF02 E-mail server configuration

  jBPM engine requires properly configured mail sender to be able to use CTU's mail server

- NF03 Technologies

  use the following for implementation: JEE, Apache Maven, jBPM, Spring Framework, Spring Roo, JPA

- NF04 System and web service security

  system and web services will be secured

- NF04.1 User roles and permissions

  security will be handled by user roles and permissions

- NF05 MySQL database

  primary DBMS will be MySQL

- NF06 Server Tomcat

  RESTful API will be able run on Apache Tomcat servlet container

- NF07 Performance under load

  system must handle 250 concurrent users and 2500 total users

- NF07.1 Scaling

  system must be able to split load among multiple instances

During the development process several changes were made to the catalogue of requirements. RESTful API team and UI team switched responsibilities for F07, F08. F03 was descoped but can be implemented with relatively small effort when needed.

NF07.1 is a matter of infrastructure. While this work was being created, only one virtual server was available for the whole Přiříz project and makes no sense to configure multiple instances on a single machine.

3.3 Who and how will use RESTful API?

I already mentioned that RESTful API will be consumed by two different teams 3.3. This is their simplified requirements catalogue - functional requirements only:

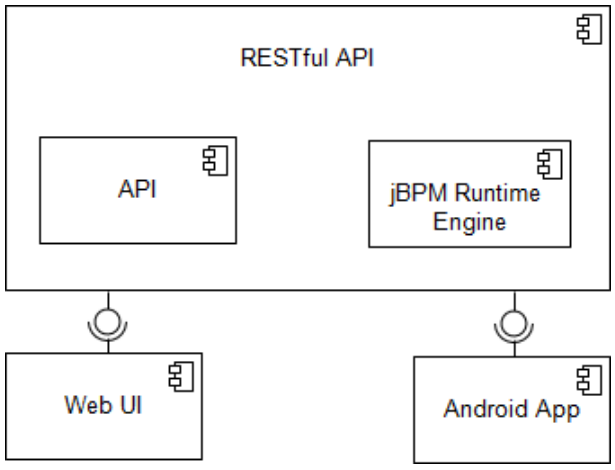

Figure 3.2: Přiříz project components

- Android team

  Responsible for native Android application development. Its tasks are:

  - allow to log-in in various user roles
  - log-out
  - barcode identification

  - view admission/admissioner information
  - save admission result - entrance exam score
  - take a picture of a document and upload it to server

- UI team

  Implements web interface. This allows admissioners and CTU FIT's employees to interact with admissions and manage them during admission process.

  - Admission/Admissioner
    * log-in/log-out
    * view personal information
    * book a term (entrance exam, enrollment)
    * apologise from a term registration
    * change/reset password
  - Employee (Study Department staff, sub dean, Faculty departments staff)
    * log-in/log-out
    * list admissions, filter
    * view admission detail
    * edit/delete admission
    * reset user's password by admission
    * view terms (entrance exam, enrollment)
    * create/edit/delete terms
    * import admission data from KOmponenta Studium, Study Information System at CTU (KOS)
    * view study programs
    * edit/delete study programme
    * confirm admissioner's attendance at entrance exam or enrollment
    * accept or decline admissioner's apology
    * edit properties of an admission process
    * view statistics
    * change password

Although UI team has quite a long list of task when compared to Android's, they practically overlap. This is why both (and for future all) teams are going to share the same API and RESTful API will handle all consumers the same way.

All requirements from the list above, but those platform specific, will use RESTful API as a backend and should not store any persistent data on their own. Platform specific means that they must store runtime data on their own. An example may be HTTP session or other temporaries.

The task for RESTful API team is to expose a public API, which will satisfy requirements from the list above.

CHAPTER 4

Chosen technologies

This chapter describes pros and cons of available tools, frameworks and architectural patterns, that may be used and explains why and which technologies I finally selected.

One of the non-functional requirements from the previous chapter 3.2.2 talks about technologies, which should be used. The truth is that NF03 was back-added after discussion described in this chapter.

4.1 REST vs. SOAP

Although implementing a RESTful API is one of the main topics of this master's thesis, it is good to compare other possibilities too. Currently the most commonly used technology to build SOA in addition to REST is SOAP. So what are the differences between them? Why should I not use SOAP, when thousands of enterprise systems are using it?

Sometimes people talk about SOAP like it was something that is deprecated, or even dead and REST is its successor and is much better and modern. This is not true and not even part of it. REST is no revolution in SOA, but rather an evolution. SOAP has its place when it comes to a question of implementing services or APIs and so does REST. The main difference is: **SOAP** is aimed for **server-server** communication and **REST** is more suitable for **client-server** communication.

Let's forget about the RESTful API requirement and make it just an API. What are the pros and cons of REST, resp. SOAP if I could choose one or the another on my own? The only things that I have to keep in mind: I'm implementing an API for admission processing and I have two different API consumers using two various platforms 3.3.

	REST	SOAP
Data Format	XML, JSON, YAML, ...	XML
Transport	agnostic, but very tightly coupled with HTTP, unlikely to use anything else	agnostic
Error handling	implementation specific	built in
Primary use	client-server directly	server-server, possibly via mediators
CRUD[a]	HTTP methods	implementation specific
Interface description	text description, XSD for XML representation	standardized WSDL, optional XSD
Tools availability	lacking, partial	outstanding
Security	HTTPS, implementation specific	built in, WS-Security
Pros	Lightweight Space saving formats Easy to learn	Standards Extensible Tool support Type checking Rigid
Cons	HTTP dependency Lacking standards Lacking tools Assumes point-to-point use	Bloating data format Complexity and learning curve

[a]Create Read Update Delete

Table 4.1: REST vs. SOAP properties

Does not look very good for REST from the overview above. Using simple math, it has less pros and more cons. But does it mean that SOAP should be chosen for API that Přiříz project needs?

Android team uses a mobile phone device, which is definitely a client. What about the Web interface? I am not sure about the implementation, but from my point of view, it should be a thin client, which stores only necessary runtime data for its own needs. No need to synchronize any other data, just consume and present. This means **two clients** will be using the API.

Web interface will be deployed on its own application server somewhere inside the CTU's infrastructure and will use fast network connection. On the other hand, Android device will be able to use some wireless fast network -

best case. But what if the mobile application will be forced to rely on a slower carrier network? EDGE - worst case. The API should then try to save as much traffic as it can. XML full of namespaces that **SOAP** happily uses is **not** the **right** way, though.

HTTP dependency is not a problem, I would use SOAP over HTTP anyway. Lack of standards is a matter of good design. This is why I have to be very careful when exposing new functions or data models. It does not mean that this issue would not effect SOAP design.

Lack of tools is a thing that bothers me a bit. If I agree on standardized approach with other teams and we involve a bit of communication, hopefully we can get over this one.

So far for the cons of REST. Which pros of SOAP will I miss, if I decided to use REST? Just the tool support.

To sum up, RESTful API would win and therefore I am happy with the original requirement.

4.2 BPEL vs. BPMN

When I started to collect information about BPM and its standards, I found many discussions about older BPEL vs. relatively new BPMN, currently available in BPMN 2.0 specification.

[6] says that the only standard worth considering is BPEL. This information is a few years old and by the time a another player has shown up. BPMN 2.0 became a solid competitor to BPEL.

[20] BPEL and BPMN are both „languages“ or „notations“ for describing and executing business processes. Both are open standards. Most business process engines will support one or the other of these languages.

It turns out that BPEL is really well suited to modeling some kinds of processes and BPMN is really well suited to modeling other kinds of processes. Of course there is a pretty significant overlap where either will do a great job.

[5] BPMN 2.0 is now a business model that can be executed after implementation details are added. BPMN favors the business user, even though a developer can „refine with execution semantics“ to make it executable. It is graph based, and incorporates user swim lanes, which makes it effective for modelling end to end business processes.

BPMN 2.0 introduces a standardized file format (previously is was proprietary or converted to XPDL). BPMN looks like a version of BPEL where the assigns are tucked away into other activities to clean up the diagram.

BPEL's nature is still service orchestration, and will be great for building composite services and integrating with applications. BPEL will still probably be the choice for developers, where BPMN will be good for the pure decision layer and **Human Task interaction**.

[15, p. 1] The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation. Another goal is to ensure that XML languages designed for the execution of **business processes**, such as Web Services Business Process Execution Language (WSBPEL), **can be visualized** with a business-oriented notation.

Human Task interaction is exactly what admission processing is looking for and visualization of the process is a must. It is expected that further modifications of business process model will be performed by non-developers. Therefore an user friendly visual modelling tool is required. That is why BPMN 2.0 looks like a good choice. Luckily jBPM, which was chosen as a primary technology for this master's thesis, fully implements BPMN 2.0 standard.

4.3 Dependency management

One of the myths about Java development that is well established among people talks about some kind of JAR hell. Basically it is a nickname for class loading problem, which used to be an issue some time ago. Nowadays there are tons of tools available that effectively solve such issues, one just needs to keep following the progress.

A solution is to use standardized Java project structure, right build tools and dependency management tools. There are dozens of such utilities but the two mainstream ones are:

- Apache Ant + Apache Ivy

  [2] Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications.

  Ant is extremely flexible and does not impose coding conventions or directory layouts to the Java projects which adopt it as a build tool.

  Software development projects looking for a solution combining build tool and dependency management can use Ant in combination with Apache Ivy.

  [7] Apache Ivy is a dependency manager oriented toward Java dependency management, although it can be used to manage dependencies of

any kind. Apache Ivy is integrated with Apache Ant, the most popular Java build management system, so Apache Ivy follows Apache Ant design principles.

- Apache Maven

  [18] Apache Maven is a software project management and comprehension tool. Based on the concept of a Project Object Model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

  Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

  - Making the build process easy
  - Providing a uniform build system
  - Providing quality project information
  - Providing guidelines for best practices development
  - Allowing transparent migration to new features

Combination of Ant and Ivy is suitable for older, already existing Java projects that use Ant scripts for build process. Ivy will allow to remove hard copied JAR files and enable dependency management using remote JAR repositories.

Maven 2 on the other hand is currently the most commonly used project management and comprehension tool. Dependency management and project build are just a small part of its capabilities. It offers easy plugin support, from which direct deployment to an application server, code coverage and code quality reports might be interesting for this project.

Another very nice effect of using is that it leads developers to use standardized project structure. I decided to use Maven for RESTful API implementation.

4.4 JAX-RS implementation

In theoretical part 1.2 of this work I briefly described JAX-RS. Standard JVM contains only the JAX-RS API, but there is no implementation out of box. Luckily, Java developers community is one of the biggest among available enterprise solutions and probably the biggest Open-source software (OSS) enterprise community. One of the advantages of such a number of developers is that there is an endless choice of frameworks, utilities and other libraries. When it comes to popular JAX-RS implementations these are worth noticing:

- Apache CXF

  Very complex, popular and pleasant to use.

- Jersey

  Oracle's reference implementation.

- RESTeasy

  From JBoss Community.

Personally I tried Apache CXF and RESTeasy. Although any of the above would work well for RESTful API, I have best experience with Apache CXF. It offers much more than just plain JAX-RS implementation. Spring framework integration is available, error handling is very neat and makes RESTful API server or client development a breeze.

Moreover, it also contains Java API for XML Web Services (JAX-WS) implementation, which might be useful for future, when SOAP integration is required. This will keep REST and SOAP parts consistent across the whole source code.

4.5 Spring Core, MVC, Security, ...

When creating a new Java project from scratch, it is good to take a while and think about frameworks, tools and possibly other utilities that might simplify development process.

I know I wanted something widely spread because of quick community support and fair documentation state. Utilities like Apache Commons, Collection Utils, ...is a must. But which framework to choose if there are so many? Well from all the available, I am going to look at the most popular ones. Good place to look or ask is Stack Overflow and this is what community recommends:

- Struts 2
- Apache Wicket
- Play
- Spring 3
- pure JEE

The list is surprisingly short. Let's list my requirements too, because this should help me to apply method of exclusion:

- fully OSS

- easily configurable
- extensible and plugin friendly

  No framework contains all one ever needs, therefore it must be easily extensible with custom components.
- suitable for integration with other frameworks

  I need to integrate JAX-RS with the rest.
- large community
- good documentation
- REST friendly

  RESTful application is considered to be a web application, but with one significant difference. When MVC patterrn is applied, we can actually skip the View part, because RESTful API generates output based on client-server agreement (the Accept HTTP header) and does not need any templates (JSP, FM, Velocity, ...). Therefore all JSF based frameworks do not fit well, as JAX-RS handles the response and rendering.

After the first round Struts 2 and Apache Wicket have to go. Struts is not complex enough and I did not find any examples of integrating JAX-RS with Struts. Although there is a REST plugin for Struts, it is not JAX-RS implementation.

Apache Wicket might be suitable for creating rich web applications, but again, no example of JAX-RS integration with itself. I found a few tutorials for RESTful application with Wicket, unfortunately this is not what I was looking for.

Play 2 framework seems very interesting. It is becoming quite popular and has an emerging community. I also found a couple of discussions about integrating it with JAX-RS, but they were linking to already non-existing sources and so I started to be rather pessimistic about using it for RESTful API. I would definitely give it a try when creating a traditional web application.

4.6 Spring 3 vs. JEE 6

I decided to leave a duel of Spring 3 and pure JEE 6 framework at the end. Nowadays one can find dozens of discussions about why JEE 6 is better than Spring 3 and why one group of developers claim that they are going to use purely JEE and the other wants to stick with Spring 3.

Both specifications came out in about the same time, late 2009. Both will do practically the same for an enterprise application, but their philosophy is quite different. While JEE 6 wraps and defines several standards, technologies

and APIs e.g. Servlet 3.0, EJB 3.1, JAX-RS, JAX-WS, JAXB, JSF, ... - their list is quite long [17]. It is application server's task to implement JEE 6 standard. Because JEE 6 is not simple, certified application servers tend to be quite big. Such application servers are JBoss AS 7, Glassfish 3.1, ...

On the other hand there is Spring 3.0. It aims to promote modularity, extensibility and portability across Java EE servlet containers - Tomcat, Jetty and others. One can integrate almost everything from the JEE standard with Spring and lightweight servlet container, because Spring itself acts like an application server. Almost means, that not everything is possible. EJB for instance will not work with plain servlet container.

So what are the benefits of using one or another? If it is not necessary to use EJB and perhaps a few other technologies exclusively provided by JEE 6, what makes JEE 6 or Spring a favorite?

	Spring 3	JEE 6
Pros	Modular Flexible Deployable to a simple servlet container Faster availability of new technologies `Faster development`[b] and deployment	Standards Usually more `complete implementation`[a]
Cons	Subset of JEE 6 stack Not based on standards - but allows flexibility	Standards - not meant to be flexible Slower release cycle Dependency on bigger and slower application servers `Uneasy to migrate`[c] an application from one AS to another

[a] Each JEE standard comes with many new features. They often bring several inventions and better and more comeplete implementation of what is already available somewhere else. An example may be Spring DI and CDI, where CDI came out later than Spring's DI, but CDI is a superset of what Spring DI offers.

[b] Especially in combination with very lightweight Jetty, development and deployment on a local machine is a matter of seconds when compared to a full application server and JEE

[c] JEE standards become deprecated after a while, this is a case of JEE 5 and JEE 6 will be no different. If one needs to upgrade an application from older to a newer standard, application server needs to be upgraded too. This brings several problems including purchase of new licences, especially when commerce AS like WebSphere or WebLogic is used.

Table 4.2: Spring 3 vs. JEE 6

After a closer look at what JEE 6 would offer at a price of slower development and dependency on AS, I can only see everything I already know from Spring (I spent several years using it). Even if it would be sufficient to use JEE Web Profile and not Full JEE Profile, which means simpler AS like TomEE.

One of the goals of this master's thesis is to explore new and modern technologies. I do not see it in JEE 6. What attracts me more is quite a new project - Spring Roo.

4.7 Spring Roo

[9] Spring Roo is a dynamic, domain-driven development framework from SpringSource. The Spring framework simplifies and expedites application development through a three-pronged approach:

- enables services on on POJOs - declaratively and transparently through dependency injection and aspect-oriented programming
- where functionality can't be achieved effectively through those channels alone provides the simplest, cleanest abstractions and APIs under the sun to solve problems
- simplify existing, often verbose APIs

Isn't Roo redundant then? Spring is one of the most proficient ways to work with Java, but the current thinking strongly supports the conclusion that the next barrier to enhancing productivity on the JVM is the Java language itself.

Spring Roo is built using standard Java and uses standard Java and Spring. During development, Roo Shell watches the developer while helping him out as much as possible and required. It is almost like a pair programming buddy or a very advanced code completion tool.

Let's suppose an example of editing a JPA entity in Spring Roo project. I want to add a field **surname** to a **Person** entity. As soon as I add the field, Spring Roo automatically jumps in and adds a corresponding accessor and mutator pair for that field to a `shadow class definition`[4] in the background. Similarly, it will implement a **toString()** definition (reflecting the fields added) if one does not already exist, and it will implement an **equals()** and **hashCode()** methods following the same criteria. Updating the field triggers the same action.

[4] AspectJ Inter Type Declaration (ITD) that Spring Roo maintains in the background. When an application compiles, the ITD is merged with the Java code, creating one class that has both the field I typed in, as well as the automatically generated accessor and mutator pair, correct equals() and hashCode() implementations, and a correct toString() implementation. One should never need to updat these ITD definitions

If I add an equals method to the JPA entity, the shadow definition is removed, delegating to my implementation instead. This shadow class definition is kept in sync, responding to the changes, but it does not get in the way.

An example **Person.java** entity:

```
@RooJavaBean
@RooToString
@RooEquals(excludeFields = { "personId" })
@RooJpaActiveRecord(finders = { "findPeopleByEmailEquals" })
@XmlAccessorType(XmlAccessType.FIELD)
public class Person {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long personId;
}
```

The shadow classes created are:

- Person_Roo_Configurable.aj

  Enables Spring driven configuration.

- Person_Roo_Equals.aj

  Adds **equals()** and **hashCode()** methods implementation.

  ```
  privileged aspect Person_Roo_Equals {

    public boolean Person.equals(Object obj) {
    ...
    }

    public int Person.hashCode() {
    ...
    }
  }
  ```

- Person_Roo_Finder.aj

  Creates custom declared **@RooJpaActiveRecord(finders = ...)** JPA query methods.

- Person_Roo_JavaBean.aj

  Accessors and mutators for Entity's fields.

  ```
  privileged aspect Person_Roo_JavaBean {

  public Long Person.getPersonId() {
  ```

```
    return this.personId;
}

public void Person.setPersonId(Long personId) {
    this.personId = personId;
}
```

- Person_Roo_Jpa_ActiveRecord.aj

  Standard Roo JPA methods. Default finders (by primary key, ...) and CRUD enablers.

- Person_Roo_Jpa_Entity.aj

  Enables JPA entity.

- Person_Roo_ToString.aj

  Adds **toString()** method implementation.

Of course, Spring Roo does not do anything unless it is asked for it. POJO needs to be properly annotated using Roo Java annotations, otherwise no shadow class definition is created.

Moreover it nicely integrates with popular IDEs like Eclipse and IDEA and works perfectly with Maven. Developer does not have to care about the **.aj** files at all. They are hidden by default from project's view and the **.java** implementation acts like all shadow class's methods were already merged.

These are practically all Spring Roo features that RESTful API will use. Of course, it offers much more:

- Database Reverse Engineering
- NOSQL support
- Extension of Spring MVC
- Scaffolding
- Spring Web Flow integration
- Integration with other, non-Spring, frameworks (GWT, Vaadin, ...)
- Logging, Security, **Testing**, ...

RESTful API uses Spring Roo's automatically generated Unit and Integration tests for all domain models and active record JPA methods.

CHAPTER 5

Implementation

Main focus of this chapter is on development and its progress. It describes application structure, its layers, used principles, design patterns and conventions taking the analytical part into account.

5.1 Environment and new Spring Roo project setup

5.1.1 Development platform

It is not a secret that developing applications for Java platform is resource consuming and requires solid performance for effective work. Luckily today's computers provide good computing power since mid-level price range and purchasing multi-core CPU and several GB of RAM is a standard.

My configuration during development and writing this master's thesis was:

- Dell Latitude E6410 laptop
- CPU Intel Core i5 Quad Core @ 2.5GHz
- 4GB RAM DDR3
- 120GB SSD MLC hard drive

What is more important these days is developer's software equipment. Starting with OS, 64bit is a must. It depends on a personal preference to use IDE or not and then of course various support tools like console or term emulator, monitoring tools, ...

This is what I have been using:

- Windows 7 64bit, openSUSE 12.2 64bit

- Java 7 runtime and compile (Oracle's JDK 1.7.0_10)
- Maven 3.0.3
- Cygwin 1.7.10 with Bash as a primary shell in Console2
- Spring Roo 1.2.2 with Java 1.7 support
- Eclipse IDE, 3.7.2 Indigo and 4.2.1 Juno
  - Spring STS
    Spring Roo's installation path needs to be set up.
  - JBoss Tools
    Maven tool should be switched from Eclipse's embedded implementation to the system wide installed one.

To make all this work, it is required to set up system environment variables properly. Especially:

- JAVA_HOME
  Pointing to the JDK 7 installation directory.
- MAVEN_HOME
  Pointing to the Maven 3 installation directory.
- ROO_HOME
  Pointing to the Spring Roo 1.2 installation directory.
- append all the above concatenated with **/bin** to the **PATH** variable

After all these steps are done, nothing should stand in the way anymore and development can start straight ahead.

5.1.2 Configuring the new project

RESTful API is using Maven and Spring Roo, which play very nicely together. I could either use basic Maven's `archetype`[5] and configure the project for Spring Roo or let it create standard Spring Roo project and tweak it, as I do not want a standard web project, but Spring Roo without Spring MVC's View integrated with Apache CXF.

In the end, I decided to combine both approaches. Because basic Maven project is nothing else but a standard directory structure with JEE web application and Maven configuration files, this gives me a clean start. On the other hand, Spring Roo will create a sample web project (also Maven project),

[5] A template used by Maven to create a new project.

from which I will simply take what I need in my clean Maven project to make it play with Spring Roo.

Let's create basic Maven project first via Eclipse IDE - no archetype:

```
.
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   `-- resources
|   `-- test
|       |-- java
|       `-- resources
`-- target
    |-- classes
    |   `-- META-INF
    |       `-- MANIFEST.MF
    `-- test-classes

11 directories, 2 files
```

Explanation of the above follows:

- **pom.xml**

  Maven's configuration file for this project. Contains project's metadata (name, description, identification, ...) and what is important - dependencies, plugins, reports and their various settings.

- **src/main/java**

  Standard place for Java source code.

- **src/main/resources**

  A place for various resources like metadata, configuration files, ... This directory will be in classpath at runtime.

- **src/test/java**

  Standard place for test Java source code. Unit and integration tests are placed here.

- **src/test/resources**

  Its role is the same as of **src/main/resources** directory, but this one is in classpath only during test phase.

- **target**

  Build directory and temporary directory. This is where all Maven outputs including deployable archive or report results are written. During the development process can be simply ignored.

How different from basic Maven project Spring Roo's project is? Let's create it by using Roo Shell take a look:

```
$ mkdir MI-MPR-DIP-Admission-Roo
$ cd MI-MPR-DIP-Admission-Roo
$ roo.sh
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.1.RELEASE [rev 6eae723]


Welcome to Spring Roo. For assistance press TAB or
type "hint" then hit ENTER.
roo> project --topLevelPackage cz.cvut.fit.mi_mpr_dip.admission
Created ROOT\pom.xml
Created SRC_MAIN_RESOURCES
Created SRC_MAIN_RESOURCES\log4j.properties
Created SPRING_CONFIG_ROOT
Created SPRING_CONFIG_ROOT\applicationContext.xml
roo>
```

Which will create a new Roo project with this structure:

```
.
|-- log.roo
|-- pom.xml
`-- src
    `-- main
        `-- resources
            |-- META-INF
            |   `-- spring
            |       `-- applicationContext.xml
            `-- log4j.properties

5 directories, 4 files
```

It created basic Maven project structure, with basic Spring context setup, `log4j`[6] configuration and pom.xml contains all necessary dependencies.

Let's use some more of Roo's features:

```
roo> jpa setup --provider HIBERNATE \
--database HYPERSONIC_IN_MEMORY
Created SPRING_CONFIG_ROOT\database.properties
Updated SPRING_CONFIG_ROOT\applicationContext.xml
Created SRC_MAIN_RESOURCES\META-INF\persistence.xml
Updated ROOT\pom.xml
...
roo>
```

The above just added persistence capabilities to the project via JPA. I know that root if the entire RESTful API is an admission, which will also serve as a key domain object - a Person is 1:1 related to the Admission. Thus I will create Admission domain object via Roo Shell:

```
roo> entity jpa --class ~.domain.Admission --testAutomatically
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\Admission.java
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\AdmissionDataOnDemand.java
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\AdmissionIntegrationTest.java
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\Admission_Roo_Configurable.aj
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\Admission_Roo_Jpa_Entity.aj
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\Admission_Roo_Jpa_ActiveRecord.aj
Created SRC_MAIN_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\Admission_Roo_ToString.aj
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\AdmissionDataOnDemand_Roo_Configurable.aj
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
```

[6]Very popular and still widely used logging library, but there are already better implementations available. Log4j is quite an old library, development is discontinued and using facade SLF4J with Logback as an implementation is a standard nowadays. This is why I am going to replace it in RESTful API.

```
\domain\AdmissionDataOnDemand_Roo_DataOnDemand.aj
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\AdmissionIntegrationTest_Roo_Configurable.aj
Created SRC_TEST_JAVA\cz\cvut\fit\mi_mpr_dip\admission
\domain\AdmissionIntegrationTest_Roo_IntegrationTest.aj
~.domain.Admission roo>
```

Argument –**testAutomatically** tells Roo Shell to generate Unit and Integration tests too. And so it did:

```
.
|-- log.roo
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   | `-- cz
|   |   |  `-- cvut
|   |   |   `-- fit
|   |   |    `-- mi_mpr_dip
|   |   |     `-- admission
|   |   |      `-- domain
|   |   |       |-- Admission.java
|   |   |       |-- Admission_Roo_Configurable.aj
|   |   |       |-- Admission_Roo_Jpa_ActiveRecord.aj
|   |   |       |-- Admission_Roo_Jpa_Entity.aj
|   |   |       `-- Admission_Roo_ToString.aj
|   |   `-- resources
|   |       |-- META-INF
|   |       |   |-- persistence.xml
|   |       |   `-- spring
|   |       |       |-- applicationContext.xml
|   |       |       `-- database.properties
|   |       `-- log4j.properties
|   `-- test
|    `-- java
|     `-- cz
|      `-- cvut
|       `-- fit
|        `-- mi_mpr_dip
|         `-- admission
|          `-- domain
|           |-- AdmissionDataOnDemand.java
|           |-- AdmissionDataOnDemand_Roo_Configurable.aj
```

```
|           |-- AdmissionDataOnDemand_Roo_DataOnDemand.aj
|           |-- AdmissionIntegrationTest.java
|           |-- AdmissionIntegrationTest_Roo_Configurable.aj
|           `-- AdmissionIntegrationTest_Roo_IntegrationTest.aj
`-- target
...
```

Let's use Maven to run the tests now:

```
roo> perform test
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
roo>
Results :
roo>
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0
roo>
[INFO] ------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
[INFO] Total time: 2:04.484s
[INFO] Finished at: Thu Jun 14 18:23:38 CEST 2012
[INFO] Final Memory: 6M/23M
[INFO] ------------------------------------------------
roo>
```

I can see that Roo has created 9 Unit and Integration tests (via JUnit) for a single domain model. The report says that all 9 tests passed.

This is just an example of how to work with Roo and Roo Shell. It really saves a lot of work and lots of typing. In following sections of this chapter I am going to describe RESTful API's design and the final structure of project.

5.2 Application architecture and layers

What I wanted to avoid in RESTful API is to use Spring MVC and Web MVC, because they are targeted for a traditional web application. Each web application after its deployment and server start should be able to tell if it is running and everything looks normal. For API like applications it becomes a bit difficult, as it does not contain anything like a home page.

This is why I decided to add something home page like just to be able to tell if RESTful API is healthy at any time - so called `Smoke test`[7] resource. Smoke test can be done by simply opening RESTful API static resource as a web page in a browser. For this and only this purpose Spring MVC is used in RESTful API.

This requirement leads to dual servlet configuration. One for pure Spring MVC and the other for Apache CXF. Of course, both will be Spring context aware, as I want it to manage my application.

Each web application deployed to a servlet container starts looking for a `web.xml`[8] file. Description of RESTful API's layers should therefore start here.

To enable Spring Application Context and to tell Spring that I want it to manage my I need to add a context parameter and a listener:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:META-INF/spring/applicationContext.xml
  </param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

This is what actually Spring Roo generated project does by default when one creates a web project via its Shell, which was not my case.

The next step is to define two new servlets and configure them. In **web.xml** file:

```
<servlet>
  <servlet-name>cxf</servlet-name>
  <servlet-class>
    org.apache.cxf.transport.servlet.CXFServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
```

[7] Usually a first test made after system restart, re-deployment or any other action that might require further testing. Smoke test is then used to tell if system does not crash immediately.

[8] The web.xml file provides configuration and deployment information for the Web components that comprise a Web application. The Java Servlet 2.5 specification defines the web.xml deployment descriptor file in terms of an XML schema document.

```
    <servlet-name>cxf</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>admin</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>admin</servlet-name>
    <url-pattern>/admin/*</url-pattern>
  </servlet-mapping>
```

This means that Spring MVC servlet is mapped to **/admin/*** URI pattern and CXF to **/services/*** pattern. The rest of the URI path is managed by annotation in MVC's Controller annotated POJOs or by CXF managed POJOs - mappers. In RESTful API named **endpoints** and **error mappers**.

For each servlet, corresponding configuration file needs to be added to context path. In this case:

- src/main/webapp/WEB-INF/**admin-servlet.xml**

  Defines URL mapping and View Resolver. Everything routes to a single **AdminController** annotated with **@Controller** annotation. FreeMarker templates are used, and so **FreeMarkerViewResolver** must be configured. It looks for **.ftl** templates in src/main/webapp/**WEB-INF/freemarker/** directory.

- src/main/webapp/WEB-INF/**cxf-servlet.xml**

  Enables Spring annotations, sets up JAX-RS REST server, its listeners (endpoints and error mappers) and content providers (XML is enabled by default via JAX-RS annotations, JSON has some specific properties and so it as to be explicitly configured).

After adding all necessary configuration files and refactoring the default Spring Roo's structure, the project looks like this:

```
.
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
```

```
|   |   |   |-- cz
|   |   |   |   `-- cvut
|   |   |   |       `-- fit
|   |   |   |           `-- mi_mpr_dip
|   |   |   |               `-- admission
|   |   |   |                   |-- adapter
|   |   |   |                   |-- authentication
|   |   |   |                   |-- builder
|   |   |   |                   |-- comparator
|   |   |   |                   |-- controller
|   |   |   |                   |-- dao
|   |   |   |                   |-- domain
|   |   |   |                   |-- endpoint
|   |   |   |                   |   |-- helper
|   |   |   |                   |   `-- mapper
|   |   |   |                   |-- exception
|   |   |   |                   |-- jbpm
|   |   |   |                   |   `-- eval
|   |   |   |                   |-- service
|   |   |   |                   |   |-- auth
|   |   |   |                   |   |-- deduplication
|   |   |   |                   |   |-- initializing
|   |   |   |                   |   |-- logging
|   |   |   |                   |   |-- mail
|   |   |   |                   |   `-- user
|   |   |   |                   |-- util
|   |   |   |                   |-- validation
|   |   |   |                   `-- web
|   |   |   `-- org
|   |   |       `-- springframework
|   |   |-- resources
|   |   |   |-- META-INF
|   |   |   |   |-- deployment.prod.properties
|   |   |   |   |-- deployment.properties
|   |   |   |   |-- jbpm-process.properties
|   |   |   |   |-- persistence.xml
|   |   |   |   |-- security.properties
|   |   |   |   |-- spring
|   |   |   |   |   |-- applicationContext.xml
                        ...
|   |   |   |-- bpm
|   |   |   |-- bsp
|   |   |   |-- msp
|   |   |   `-- process
```

```
|   |   `-- webapp
|   |       |-- WEB-INF
|   |       |   |-- admin-servlet.xml
|   |       |   |-- cxf-servlet.xml
|   |       |   |-- freemarker
|   |       |   `-- web.xml
|   |       `-- error
|   `-- test
|       |-- java
|       |   `-- cz
|       |       `-- cvut
|       |           `-- fit
|       |               `-- mi_mpr_dip
|       |                   `-- admission
|       `-- resources
|           |-- META-INF
|           `-- test

98 directories, 877 files
```

The above is final RESTful API's structure. I left out unimportant verbose information, like **.java** and **.aj** files, subdirectories, ...

Basically, **src/main/webapp** directory is added, which contains web application configuration files. Spring context configuration is split into multiple files, where the original **applicationContext.xml** server is just as a parent and imports child configurations.

Log4j has been completely removed and excpluded in **pom.xml** from all dependencies. SLF4j with Logback replaced it.

src/main/resources/META-INF contains several **.properties** files:

- **deployment.properties** for development
- **deployment.prod.properties** for production

  General application properties including database configuration, services properties, ...
- **jbpm-process.properties**

  jBPM context properties.
- **security.properties**

  Contains **Roles** and **Permissions** definition. RESTful API's security is explained in 5.3.

They serve as placeholders for for Spring context, where each variable marked as **${key}** is replaced during Spring initialization by value found in either of listed property files.

Interesting is to see how application layers are situated and how they communicate with each other:

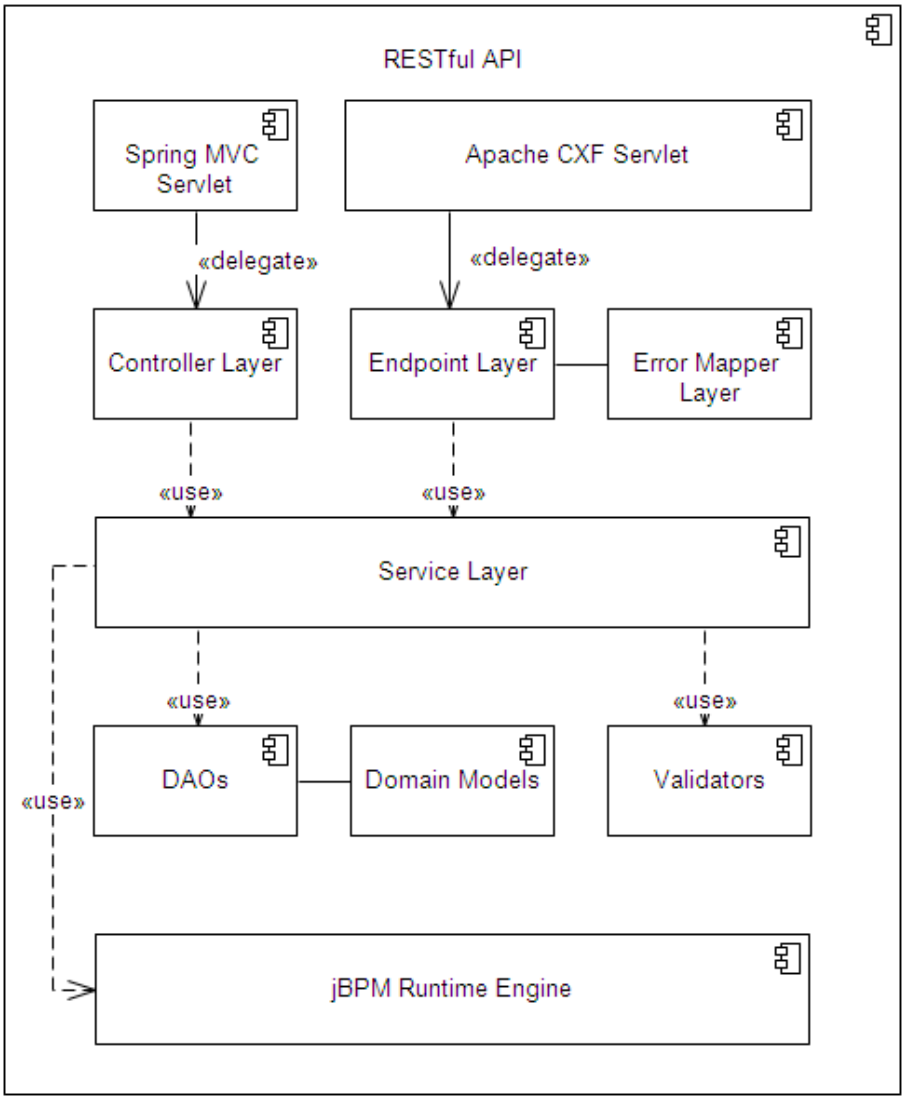

Figure 5.1: RESTful API internal architecture

Having two different servlets defined does not mean that RESTful API has duplicities. It was necessary to enable various capabilities and each required different approach. Both are then allowed to use the same application layers - Model in MVC pattern.

5.3 Security

RESTful API is tightly coupled with Spring ecosystem in general and heavily uses its features. Often even extends its implementations with custom ones. Spring framework is nicely modular and Maven enables to add only those dependencies that are really needed. Spring Security library is one of them.

It is an authentication and access-control framework, which became standard for securing Spring-based applications. Enabling Spring Security for Spring application is just a matter of adding a simple filter definition to a **web.xml** descriptor file:

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

5.3.1 Authentication

Further configuration of Spring Security is exclusively defined via Spring context configuration. In RESTful API a file found in META-INF/spring/**spring-security.xml**. Each CXF endpoint has a **filter chain** defined. Such definition example may be:

```
<bean id="springSecurityFilterChain"
    class="org.springframework.security.web.FilterChainProxy">
  <constructor-arg>
    <list>
      <security:filter-chain pattern="/services/user/identity"
        filters="sessionAuthenticationFilter,\
        basicAuthenticationFilter,anonymousAuthFilter,\
        exceptionTranslationFilter" />
      <security:filter-chain pattern="/services/admission"
        filters="sessionAuthenticationFilter,\
        anonymousAuthFilter,exceptionTranslationFilter" />
    </list>
  </constructor-arg>
</bean>
```

The above shows two different filter chains that can be found in RESTful API. The first one is used in only one case - a UserIdentity was requested via credentials:

- **sessionAuthenticationFilter**

  RESTful API's custom authentication filter. It looks for **X-CTU-FIT-Admission-Session** HTTP request header, where a **session identifier** from **/user/identity** RESTful API's call is provided. This carries information about **UserIdentity**, its **Roles**, session validity, ...

- **basicAuthenticationFilter**

  Default Basic HTTP Authentication implementation from Spring Security. Verifies **username** and **password** against Faculty's Lightweight Directory Access Protocol (LDAP) for employees or RESTful API's database for imported **Admissions**. During the import, **UserIdentity** is created for each **Admission**.

- **anonymousAuthFilter**

  A fallback filter, which sets **Anonymous Identity** into Security Context, when no **Identity** has been set by previous members of the filter chain.

- **exceptionTranslationFilter**

  Translates an Exception thrown during security filter chain processing into HTTP response if possible.

The later one is a standard security filter chain used for every other use case in RESTful API:

- **sessionAuthenticationFilter**
- **anonymousAuthFilter**
- **exceptionTranslationFilter**

Basically, it is missing **basicAuthenticationFilter** and therefore it requires RESTful API's session identifier to successfully authenticate **UserIdentity**.

5.3.2 Authorization

To be authenticated is not enough to use any of RESTful API's endpoints. Successful authentication prevents one from receiving 401 HTTP response. Each endpoint's call requires a specific **Permission**. Relationship between

Role and **Permission** is M:N. Relationship between **Role** and **UserIdentity** is M:N.

Without necessary **Roles**, authenticated **UserIdentity** lacks **Permissions** and such call will be rejected with HTTP 403 response.

Endpoint's implementation methods are annotated with **@Secured** annotations, where required **Permissions** are specified:

```
@Secured("PERM_WRITE_ADMISSION")
@Consumes({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML })
@POST
@Override
public Response addAdmission(Admission admission) {
  validateAndDeduplicateAndStore(admission);
  return Response.created(...).build();
}
```

The example above shows the **/admission** POST method. It is able to consume both, JSON and XML request body, expects **Admission** serialized object in it and requires the **PERM_WRITE_ADMISSION Permission**.

5.4 jBPM

jBPM processing machine luckily integrates with Spring quite well. In RESTful API is configured directly via Spring Context and its configuration is stored in **src/main/resources/META-INF/spring/jbpm.xml**.

Admission business processes and its Use Cases can be simply modelled in BPMN 2.0 using a graphic tool. JBoss Community offers such application as Eclipse IDE plugin. All **.bpmn** processes in RESTful API have been created this way.

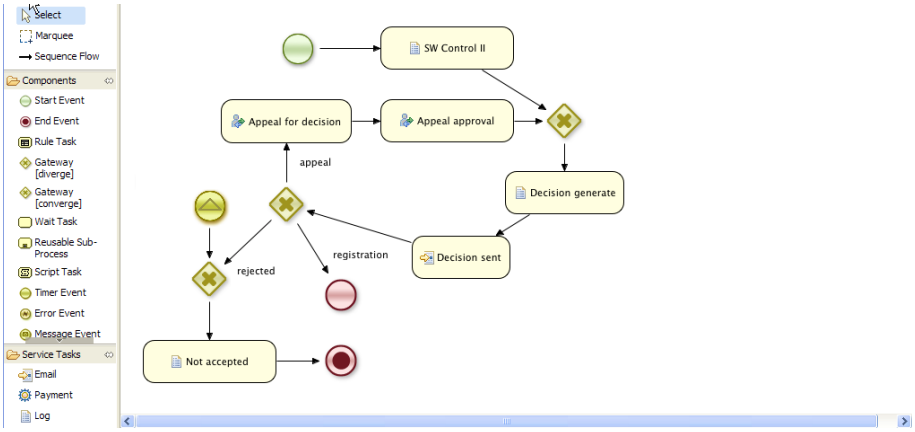

Figure 5.2: Eclipse Drools BPMN 2.0

It is needed to differentiate between **bachelor's** and **master's** Admission processes. These are further split into smaller parts like registration, apology, enrollment, ...- also BPMN 2.0 processes. Their **.bpmn** representations are located in **src/main/resources/bsp** respectively **src/main/resources/msp** directories. Common processes are then stored in **src/main/resources/process** project directory. Their graphical forms can be found in Appendices section D.

Each RESTful API call, which has some impact on Admission process should invoke appropriate task in jBPM runtime engine. Such action is allowed by using Spring managed **@Service** annotated implementations from **cz.cvut.fit.mi_mpr_dip.admission.jbpm** package. After RESTful API's Service and DAO layers finish their job, Admission is then further passed to jBPM runtime engine for evaluation. This guarantees correct state in Admission process.

In Admission object representation, which RESTful API provides to its consumers, a state is represented by AdmissionState object. Modification of the state is jBPM runtime engine's responsibility. Further it handles a few other tasks including notification via e-mail and writing an information about changes made to the Admission object in logs.

5.5 Error handling

JAX-RS itself defines error mapping interfaces and a framework for capturing exceptional states. Apache CXF, used as implementation of JAX-RS in RESTful API, handles several common errors such as authentication, authorization and bad request body format. Returns appropriate response, when error mapper is registered. CXF servlet has to be configured to do so:

```
<jaxrs:server id="restContainer" address="/">
```

```
    ...
    <jaxrs:providers>
      ...
      <ref bean="accessDeniedExceptionMapper" />
      <ref bean="businessExceptionMapper" />
      <ref bean="technicalExceptionMapper" />
      <ref bean="jaxbExceptionMapper" />
      <ref bean="webApplicationExceptionMapper" />
      <ref bean="throwableExceptionMapper" />
    </jaxrs:providers>
  </jaxrs:server>
```

All **ExceptionMapper** Spring managed beans are implementations of **javax.ws.rs.ext.ExceptionMapper<E>** interface, where **E** is a subclass of **java.lang.Throwable** being captured by this Exception Mapper. Besides special types of Exceptions defined by various frameworks:

- **AccessDeniedException**

  Typically produces HTTP 401 or 403 response

- **JaxbException**

  Produces HTTP 400 response

- **WebApplicationException**

  Various 4xx and in some special cases 5xx HTTP response

RESTful API defines three „levels“ of custom Exceptions, which are either thrown by Validation layer or some library used by RESTful API:

- **BusinessException**

  Exclusively thrown in Validation layer. This means an error on Client side. Request constrains are violated, Client is not authorized, ... Produces 4xx HTTP response and results in `OK`[9] RESTful API status.

- **TechnicalException**

  A known and recognized Exception, which is usually a wrapper for the original Exception thrown by a backend or database. In this case a Client cannot fix the problem on his side, but he can retry later - a cause of the problem is shown in the response message. Usually returns HTTP 500 response.

[9] RESTful API did not produce any errors on server side. Client has supplied invalid request and typically by fixing problem on his side, request can be accepted and could result in 2xx or 3xx HTTP response.

- **Throwable**

  An unexpected and unknown error. Most probably indicates a bug in RESTful API. Results in HTTP error 500 without further description as it may put some Java specific information into the message.

  Both TechnicalException and Throwable cause `ERROR`[10] RESTful API response status.

A standardized Error Response body is used for all Exceptional states of RESTful API:

```
<errorResponse>
  <message>Access is denied</message>
  <internalRequestId>
    33d91c25-9d42-4291-9af7-a387dbb49ffe
  </internalRequestId>
</errorResponse>
```

From which:

- **message**

  Should tell the Client what happened, if possible.

- **internalRequestId**

  A value generated for each RESTful API request and is its unique identifier, which is further appended to each row in the server log and so progress can be easily traced in it. Non ERROR responses contain this value in HTTP response headers.

5.6 Profiling, Logging

An important part of business logic of each application should be its ability to inform system administrator, operation or even developer, what is happening and in a case of failure, why it was not able to process the task.

RESTful API does not have many possibilities to choose from. Writing a system log seems to be a simple and usable approach.

As I already mentioned, SLF4j logging API and Logback as its implementation is used. To utilize their capabilities, RESTful API defines and implements **LoggingService** interface. This service contains:

- **logRequest(BufferedRequestWrapper httpRequest)**

  All information from HTTP request including headers

[10] An error has occurred on Server side and Client cannot fix it by changing his request. ERROR status forces RESTful API's Logging service to be more verbose than OK status. It writes full Exception chain stack trace into server log with ERROR priority.

- **logRequestBody(BufferedRequestWrapper httpRequest)**

  Copies HTTP request body InputStream and sends to Logger

- **logResponse(BufferedResponseWrapper httpResponse)**

  All information from HTTP response including headers and **status** - OK, request processing duration, HTTP response code

- **logResponseBody(BufferedResponseWrapper httpResponse)**

  Copies HTTP response body OutputStream and sends to Logger

- **logErrorResponse(BusinessException exception)**

  The same as **logResponse**

- **logErrorResponse(TechnicalException exception)**

  The same as **logResponse**, but the **status** is ERROR and Exception's stack trace is sent to the Logger

- **logErrorResponse(Throwable throwable, Integer httpResponseCode)**

  The same as logErrorResponse(TechnicalException exception), but **UnexpectedError** message is put into the response body

LoggingService methods are executed from **LoggingFilter**, which is an implementation of a standard **javax.servlet.Filter**. A single exception is executing **logErrorResponse** methods, which are invoked from JAX-RS Exception Mappers.

To put it all together one example RESTful API successful call results in these log entries:

```
12-06-26 11:14:19.618 INFO  991a0741-d86b-4334-8134-876c5ad25967
admission-request
call-identifier=/admission/services/user/identity_GET
Connection=keep-alive Authorization=Basic_bGVzczpsZXNz==
Accept=application/xml User-Agent=Java/1.7.0_05
Host=localhost:9090 query=null
12-06-26 11:14:19.633 INFO  991a0741-d86b-4334-8134-876c5ad25967
admission-request-body
12-06-26 11:14:19.846 INFO  991a0741-d86b-4334-8134-876c5ad25967
c.c.f.m.a.a.UserIdentityAuthenticationProvider
Found authenticationService
[c.c.f.m.a.s.a.LdapAuthenticationService] for authentication
[LDAP]
12-06-26 11:14:19.846 INFO  991a0741-d86b-4334-8134-876c5ad25967
c.c.f.m.a.a.UserIdentityAuthenticationProvider
```

```
Successfuly authentified [less]
12-06-26 11:14:20.638 INFO  991a0741-d86b-4334-8134-876c5ad25967
admission-response
call-identifier=/admission/services/user/identity_GET code=200
duration=1024 status=OK
12-06-26 11:14:20.638 INFO  991a0741-d86b-4334-8134-876c5ad25967
admission-response-body <?xml version="1.0" ...
```

Each log entry contains a timestamp, **internalRequestId**, Logger name (admission-request, admission-request-body, ...) followed by information being logged, which can contain more parts separated by a single white space.

It is a custom that application server with JEE application deployed is „hidden“ behind AJP or HTTP proxy, e.g. Apache httpd or nginx. These applications also provide some logging and profiling functionality, but to enable RESTful API's profiling capabilities, **duration** in ms is also sent to Logger.

Logback is further configurable and allows many different log formats, outputs, log rotating etc. File log appender is used in production environment. Console output is configured for development.

One important information is that the entire logging and file writing process is asynchronous and so it does not block RESTful API from completing its primary task. This is why under a high load with many concurrent users log entries might appear in different order than expected. However, it is very common in all enterprise production environments.

CHAPTER 6

Testing

6.1 Unit Testing

[14] The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as expected. Each unit is tested separately before integrating them into modules to test the interfaces between modules.

In other words, unit tests test class by class, method by method. If a tested class or method has a dependency, the dependency has to be stubbed, mocked, faked, dummied or spied. The five just named together form a set of test doubles.

6.1.1 Test-driven Development (TDD)

[10] TDD is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working. This technique is heavily emphasized by those using Agile development methodologies.

These steps should be followed when doing TDD right:

- Understand the requirements of the story, work item, or feature that you are working on
- **Red** Create a test and make it fail
  - Imagine how the new code should be called and write the test as if the code already existed.
  - Create the new production code stub. Write just enough code so that it compiles.

  - Run the test. It should fail. This is a calibration measure to ensure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail.

- **Green** Make the test pass by any means necessary
  - Write the production code to make the test pass. Keep it simple.
  - Some advocate the hard-coding of the expected return value first to verify that the test correctly detects success. This varies from practitioner to practitioner.
  - If you've written the code so that the test passes as intended, you are finished. You do not have to write more code speculatively. The test is the objective definition of „done." The phrase You Ain't Gonna Need It (YAGNI) is often used to veto unnecessary work. If new functionality is still needed, then another test is needed. Make this one test pass and continue.
  - When the test passes, you might want to run all tests up to this point to build confidence that everything else is still working.

- **Refactor** Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass
  - Remove duplication caused by the addition of the new functionality.
  - Make design changes to improve the overall solution.
  - After each refactoring, rerun all the tests to ensure that they all still pass.

- Repeat the cycle. Each cycle should be very short, and a typical hour should contain many Red/Green/Refactor cycles

Personally, I adapted to work like this very quickly. It, however, has a drawback. One usually has to write two and more times more code than he would without testing. Each refactoring, that does not involve method renaming and extraction only, requires even more effort to update unit tests. This is why I used this approach only to test it within RESTful API, when developing import of admissions.

Due to lack of time, I continued without implementing unit tests any further. On the other hand, I encourage everyone to adopt this technique, as it adds much more confidence and trust in developer's work.

Frameworks that help to make unit testing a joy are e.g. JUnit and TestNG.

There are several ways how to verify that production code is covered by unit tests. Very helpful tools, which are also available as Maven report plugins, are Cobertura, JaCoCo (continuation of EMMA as a forked project) or

JMockit code coverage. If one is willing to pay for a commercial solution, Clover may be worth looking. These metrics should become a standard, when developing an important project. Therefore, they should be set to 100% code coverage by unit tests, both line and branch.

Another very helpful tool and Maven plugin is Findbugs. It compares the source code with a database of known best practices and warns the developer, if there are some places that need fixing or attention.

6.2 Integration Testing

[11] Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing identifies problems that occur when units are combined. By using a test plan that requires you to test each unit and ensure the viability of each before combining units, you know that any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

6.2.1 Arquillian

It is a testing platform for the JVM that enables developers to easily create automated integration, functional and acceptance tests for Java middleware.

Arquillian handles all the plumbing of container management, deployment and framework initialization. Moreover it covers all aspects of test execution, which entails:

- Managing the lifecycle of the container
- Bundling the test case, dependent classes and resources into a Shrink-Wrap archive
- Deploying the archive to the container
- Enriching the test case by providing dependency injection and other declarative services
- Executing the tests inside the container

- Capturing the results and returning them to the test runner for reporting
- Integrates with familiar testing frameworks (e.g., JUnit, TestNG), allowing tests to be launched using existing IDE, Ant and Maven test plugins

RESTful API uses Spring Roo generated integration tests all for JPA Entities. Each domain model is tested via Spring Roo unit and integration tests.

Due to lack of time I did not implement any integration tests using Arquillian, but if RESTful API should be further developed, this is a good framework to work with.

6.3 Acceptance Testing

A common form of acceptance testing consists of tests that exercise a consumer scenario. These tests emulate the application interface interactions. Important forms of acceptance testing follow.

6.3.1 Verification and Regression Testing

Verification testing is a regular check of application functionality. It is done from a perspective of a consumer. For RESTful API this means the client.

Regression testing means testing that the software did not stop working. In other words: functionality that was working yesterday, is still working today. This can be understood as a set of Verification tests collected over time.

[13] Any time implementation within a program is modified, regression testing should be done. It usually means rerunning existing tests against the modified code to determine whether the changes break anything that worked prior to the change and by writing new tests where necessary. Adequate coverage without wasting time should be a primary consideration when conducting regression tests.

Some strategies and factors to consider during this process include the following:

- Test fixed bugs promptly. The programmer might have handled the symptoms but not have gotten to the underlying cause.
- Watch for side effects of fixes. The bug itself might be fixed but the fix might create other bugs.
- Write a regression test for each bug fixed.
- If two or more tests are similar, determine which is less effective and get rid of it.
- Identify tests that the program consistently passes and archive them.

- Focus on functional issues, not those related to design.
- Make changes (small and large) to data and find any resulting corruption.
- Trace the effects of the changes on program memory.

6.3.2 How to perform Regression Testing

First thing that should be kept in mind is that verification or regression should test the whole application in an environment similar to production. This is why it should be properly deployed and running.

The only change between test and production environment should be testing data and testing without any other system dependencies. E.g. if a middleware is tested, it should be running in standalone mode - I do not want to test third party's application. This can influence two sides of the application. Input (North) or Backend interaction (South) or both.

What does this mean for RESTful API? Because it is directly consumed by clients, I do not need to care about the Northern part. Tests will just act like clients. More interesting is the Southern part. RESTful API communicates with Faculty's LDAP server and mail server. During Verification or Regression testing none of them should be contacted.

Each RESTful API's service, that has a possibility to interact with a backend, contains an adapter implementation. Adapter is a Dependency Injection (DI), which is injected during server startup and its implementation is configurable via **deployment.properties** file:

```
# DUMMY | PROD
adapter.ldap=DUMMY
adapter.pwd=PROD
# DUMMY | UUID
generator.string=UUID
# EMAIL
# true | false
mail.disable=false
```

A test suite for RESTful API is available. It is split into Regression pack and Load testing pack. The first one contains multiple test scenarios, which were collected during the development. They verify RESTful API in a matter of its basic functionality. `Apache Jmeter`[11] has been used for this purpose. The **.jmx** testing project is available as a part of RESTful API's source code

[11] [1] The Apache JMeter™ desktop application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.

jmeter/admission_tests.jmx. It contains „profiles“ for various test cases, which can be enabled or disabled by selecting desired user variables set via JMeter GUI.

6.3.3 Performance Testing

3.2.2 NF07 talks about number of total and concurrent users that RESTful API should be able to handle. To verify this requirement I ran a simple performance test.

Performance test should probe the application in an environment similar to production. It strictly focuses on the application that the test scope is about. This means, there should be no delays due to the backend communication, e-mail sending or similar action. The goal is to go through all the application layers from the top to the bottom and back: request - processing - response. It can also be used to find memory leaks in the application, but this is out of scope of my work.

Load testing pack of my JMeter test suite was created specifically for this task. A fact is that `Rainy day`[12] scenarios are left out from Load testing pack, which makes it different from the Regression pack. Simply because they usually do not use all application layers and the request is usually refused during validation. This would give me distorted results and also would make error rate detection difficult under high load.

In other words, the Load testing pack is a subset of Regression test pack, where only positive `Sunny day`[13] scenarios are included.

RESTful API should be performance tested in three scenarios:

- **Admissions import**

  This is one time task, which imports large number of new data through a single RESTful API call. The performance test should verify that RESTful API does not slow down during the import - each Admission should be processed in constant time.

- **Entrance exam**

  Short term status, when large number of Admission results are put through RESTful API with many photos. Basically hits two endpoints. Should again verify that RESTful API does not slow down due to high data load.

- **Casual use**

  Test should simulate standard behavior of Web UI interface with focus on Registrations, session creation, Admission read, ... This is where the

[12] Non standard sequence of actions. Errors, exceptions, and less typical usage paths
[13] The typical sequence of actions and system responses

original NF07 is applicable and result of this scenario should reflect it in results.

Admissions import and Entrance exam scenarios have practically been tested during the development phase, when Web UI and Android teams tested their applications against RESTful API. No issues have been discovered.

More important is to test, whether the application can handle number of concurrent users described in NF07. No server errors should appear in logs and because performance test **.jmx** suite contains Sunny day scenarios only, no business errors should appear either. This is not completely true. Because there is only a single User Identity used for all operations, concurrent access will mess up session removal. For this and only this single case 401 and 404 HTTP error responses are not considered to be an error.

Number of total users is not that interesting, because it is then just a matter of time to process them.

6.3.3.1 Infrastructure

When load testing an application a separate environment should be set up for both, the application being tested and the load injecting application. Luckily Faculty of Information Technology (FIT) provides quite powerful infrastructure, and so RESTful API runs on its own virtual machine, which contains:

- AMD quad core CPU @ 2.0GHz
- 6GB RAM

This configuration should be sufficient. For production use it should, however, be duplicated and load balanced.

6.3.3.2 Scenarios and results

For quick performance evaluation I decided to create statistics of all calls from RESTful API logs found on shared virtual server. These were collected for several days. This is what I declared to be a casual use. Load test is then set up to simulate behavior of Web UI and Android teams during their development.

Load test was then performed in two phases:

- 50% load, i.e. 125 concurrent threads ($\approx$ users)
- 100% load, i.e. 250 concurrent threads ($\approx$ users)

Tests were run for approximately 15 minutes in each setup. During the test I found out, that the most resource consuming operation is the import of a new Admission. This implies creating all its child entities or assigning

Identifier	Ratio	Description
throughput.viewproperties	5	Static page, lists all server properties
throughput.admissions	20	Retrieve list of admissions, paginated
throughput.admission.crud	5	Admission CRUD scenario
throughput.user.userIdentity	60	Obtain all user information (session, roles, ...)
throughput.programs	20	Retrieve list of programs, paginated
throughput.programme.crud	10	Programme CRUD scenario
throughput.terms	20	Retrieve list of terms
throughput.term.crud	10	Term CRUD scenario

Table 6.1: Load distribution among calls

the existing ones (RESTful API does not create duplicates for entities, that can be shared, e.g. Country, City, ...). Out of the interest I did one Load test round without Admission import. This implies more GET than PUT or POST operations.

Detailed result analysis can be found in Appendix C. They prove that RESTful API is capable of handling the request amount of concurrent users. Error ratio is zero, which means that all requests have been successfully processed. Although it lacks responsibility a bit. This leaves space for further optimisation. Places worth to look at involve:

- JPA and Hibernate tuning
- Second level database cache for entities
- MySQL, JVM, Tomcat and virtual machine tuning
- Multiple Tomcat nodes with load balancing

This list is not even half complete, but may be considered as a hint at the beginning. Each of the items listed above can possibly lead to new discoveries and possible enhancements.

6.3.4 User Acceptance Testing

This is a special case of Verification testing, when application is working exactly like in production environment, put together with Northern and Southern dependencies.

The only difference when compared to real production use is that non-production data should be provided as an input during the test. Typically test users and some randomly generated data.

For such scenario, a small subset of Regression test pack can be selected, built on top of Sunny day scenarios. UAT should run very quickly. A common approach for RESTful API might be that each endpoint is hit with a single Sunny day request. Regression Test pack could be reused again or a quick automatic or manual test via Web UI is applicable too.

CHAPTER 7

Conclusion

Purpose of this chapter is to summarize everything that has been done in a manner of this master's thesis and to describe results of RESTful API team's work.

Results are evaluated against primary tasks and Catalogue of requirements 3.2.

7.1 RESTful API and its application

Functional requirements and special requirements of the other two teams practically describe public interface of RESTful API. Positive result is that all points of Functional requirements are fulfilled.

What I was unable to achieve is to implement Admission filters. Lower priority of this special requirement by Web UI team caused its last place in the implementation plan, which unfortunately was not completely finished. This however has no critical impact on RESTful API's functionality.

7.2 jBPM

The BPM part of this project was something very new to me and I had to dive into business processes in general first. jBPM as a processing machine was studied afterwards. It indeed is a technology, which is worth working with and it found its use case in RESTful API application. I will consider jBPM in my further projects, where it may be helpful, as well.

We managed to create business processes for all the Conditions for admission and Dean's directive for admission process. Although they work very well when tested isolated, due to lack of time we did not manage to hook BPM processes with RESTful API's service layers. They both work with the same database and data, unfortunately do not play well together yet.

7.3 Open questions

What shall be done with jBPM part? If RESTful API should be deployed into production, code change and some development is required. This should be a simple task, because both API and jBPM work correctly when isolated.

If Conditions for admission or Dean's directive change, how does it effect BPM models? It is quite simple to perform model changes using GUI tools. They however have to be deployed into RESTful API. Currently BPM models are stored as plain files. An enhancement would be to store them encoded in the database and swap them via some simple GUI, e.g. custom web application.

7.4 Lessons learned

Each project of such scale should teach all involved people a lesson and all of them should gain valuable experience. RESTful API is not an exception.

The main problem was time and underestimating of the whole project. It turns out, that the required amount of time was 120 man days for two skilled developers. This is three months in two people working on the project full time!

Another problem was lack of experience in selected technologies. None of the two working on RESTful API was experienced in BPM and jBPM, and David was lacking Java, Spring and JPA skills as well. This, of course, slowed down the development.

Assigning one more person to a project of such scale should be considered next time.

7.5 Contribution to the community

During the development of RESTful API and possibly some other school projects, I discovered, that all of my Maven based applications share lots of common dependencies, plugins and settings. This is why I stared a tiny project simply called *parent*. It contains lots of version properties and common settings. Can be simply used as a Maven project parent.

Another one is a tiny set of JUnit and EasyMock testing templates. One has to follow a pattern, where a single class is tested and all of its dependencies can be injected and are mocked. This is practically what a developer has to do manually for each test class. My project called *test-templates* does this automatically just by using annotations. This saves lots of typing.

Both projects can be found on my GitHub [16] and are public.

Bibliography

[1] Apache JMeter. 1999-2012. Available at WWW: <http://jmeter.apache.org/>

[2] Ant, A.: Apache Ant. 2000-2012. Available at WWW: <http://ant.apache.org/>

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

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

[5] CONSULTING, M.: BPMN 2.0 vs BPEL (Oracle BPM VS SOA Suite). September 2010. Available at WWW: <http://www.mandsconsulting.com/bpmn-20-vs-bpel-oracle-bpm-vs-soa-suite>

[6] Havey, M.: *Essential Business Process Modeling*. O'Reilly Media, 2005.

[7] Ivy, A.: Apache Ivy. 2012. Available at WWW: <http://ant.apache.org/ivy>

[8] 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>

[9] Long, J.; Mayzak, S.: *Getting Started with Roo*. O'Reilly Media, 2011.

[10] Microsoft: Guidelines for Test-Driven Development. Available at WWW: <http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx>

[11] Microsoft: Integration Testing. Available at WWW: <http://msdn.microsoft.com/en-us/library/aa292128(v=vs.71).aspx>

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

[13] Microsoft: Regression Testing. Available at WWW: <http://msdn.microsoft.com/en-us/library/aa292167(v=vs.71)>

[14] Microsoft: Unit Testing. Available at WWW: <http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71)>

[15] OMG: Business Process Model and Notation (BPMN). January 2011. Available at WWW: <http://www.omg.org/spec/BPMN/2.0>

[16] Ondrušek, J.: GitHub. 2011-2012. Available at WWW: <https://github.com/JeanVEGA>

[17] Oracle: Java EE 6 Technologies. 2009. Available at WWW: <http://www.oracle.com/technetwork/java/javaee/tech/index.html>

[18] Porter, B.; van Zyl, J.: Apache Maven Project. 2002-2012. Available at WWW: <http://maven.apache.org/>

[19] 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>

[20] Shepherd, P.: Architecture Standards – BPMN vs. BPEL for Business Process Management. March 2011. Available at WWW: <https://blogs.oracle.com/enterprisearchitecture/entry/architecture_standards_bpmn_vs>

[21] jBPM team, T.: jBPM User Guide. May 2012. Available at WWW: <http://docs.jboss.org/jbpm/v5.3/userguide/>

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

APPENDIX A

Acronyms

Acronyms

API Application Programming Interface.

BPEL Business Process Execution Language.

BPEL4WS Business Process Execution Language for Web Services.

BPM Business process management.

BPMI Business Process Modeling Initiative.

BPML Business Process Modeling Language.

BPMN Business Process Model and Notation.

BPSS Business Process Specification System.

DI Dependency Injection.

FIS Faculty Information System.

FIT Faculty of Information Technology.

HATEOAS Hypermedia As The Engine Of Application State.

HTTP Hypertext Transfer Protocol.

ICT Information and communication technologies.

IT Information Technology.

ITD Inter Type Declaration.

JAX-RS Java API for RESTful Web Services.

JAX-WS Java API for XML Web Services.

JSON JavaScript Object Notation.

KOS KOmponenta Studium, Study Information System at CTU.

LDAP Lightweight Directory Access Protocol.

MDA Model-Driven Architecture.

MIME Multipurpose Internet Mail Extensions.

OMG Object Management Group.

OSS Open-source software.

POM Project Object Model.

REST REpresentational State Transfer.

SOA Service-oriented architecture.

SOAP Simple Object Access Protocol.

TDD Test-driven Development.

URI Uniform Resource Identifier.

WfMC Workflow Management Coalition.

WS-CDL Web Services Choreography Description Language.

WSBPEL Web Services Business Process Execution Language.

XML Extensible Markup Language.

YAGNI You Ain't Gonna Need It.

APPENDIX B

Build and Deploy

One benefit of using open source software and projects is, that one can benefit from continuous development of community and modern trends without any need to write extensive amount of code. Man just has to keep up and follow, what is happening and what is new.

I consider Maven to be one of the best things that could ever happen to the Java development. A few years ago, when I was writing my bachelor's thesis, I remember setting up set of nested Ant *build.xml* files, tons of properties and each modification took hours of studying the custom build process. Maven on the other hand follows the 'convention instead of configuration' approach. This means, if not explicitly specified, all Maven based projects keep the same structure and share the same lifecycle, where build phases follow:

- **validate** the project is correct and all necessary information is available
- **compile** the source code of the project
- **test** the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** takes the compiled code and packages it in its distributable format, such as a JAR, WAR, EAR
- **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run
- **verify** - run any checks to verify the package is valid and meets quality criteria
- **install** - install the package into the local repository, for use as a dependency in other projects locally

- **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

B.1 Web application

RESTful API requires environment set up according to 5.1.1. Then follow database installation B.2. After that, *parent* and *test-templates* need to be installed in a local Maven repository. To do so, checkout the two from my GitHub repositories [16] or use snapshots from the attached CD E:

```
parent > mvn install
test-templates > mvn install
```

When RESTful API is checkout fresh from its repository, then nothing should stop the packaging process:

```
MI-MPR-DIP > mvn package
```

To make sure nothing is left from some older build:

```
MI-MPR-DIP > mvn clean package
```

This will compile all sources, execute both unit and integration tests and create deployable WAR package. This can be found in **target/admission.war**. This file can be deployed in any servlet container, such as Jetty or Tomcat. Application server is not required.

RESTful API is also configured for simple deployment to Tomcat 7 container. In the **pom.xml** file, several properties should be edited:

```
<maven.tomcat7.url>
  http://tomcat7-host:port/manager/html
</maven.tomcat7.url>
<maven.tomcat7.username>USERNAME</maven.tomcat7.username>
<maven.tomcat7.password>PASSWORD</maven.tomcat7.password>
```

Now Maven is ready to deploy RESTful API into Tomcat 7:

```
MI-MPR-DIP > mvn clean package tomcat7:deploy
```

B.2 Database

Because no application logic is written using SQL, i.e. no stored procedures nor triggers, database machine can be as simple and resource saving as possible. MySQL with InnoDB engine suits well.

RESTful API source code is shipped with **.sql** schema installation file. This can be found in **src/main/resources/META-INF/sql/admission.sql**. Installation in MySQL follows:

```
~ > mysql
mysql > create database admission character set utf8 collate utf8_bin;
mysql > quit
~ > mysql admission < src/main/resources/META-INF/sql/admission.sql
```

The last step is to point the RESTful API to the MySQL server. This is done via **.property** file, which can be found in **src/main/resources/META-INF/deployment.properties**. Concrete keys are:

```
database.url=jdbc:mysql://mysql-host:port/admission
database.username=USERNAME
database.password=PASSWORD
```

APPENDIX C

Performance tests

C.1 50% load simulating 125 concurrent users

Figure C.1: Hits per second without Admission import

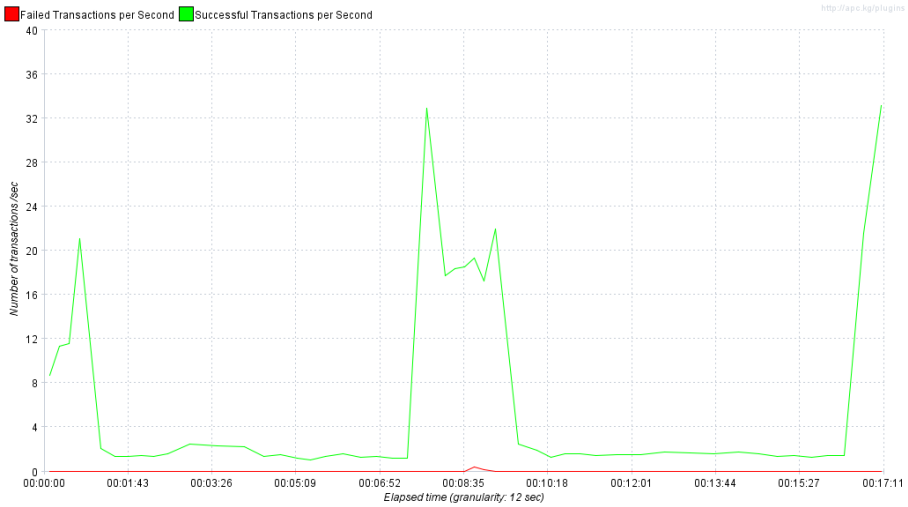

Figure C.2: TPS without Admission import

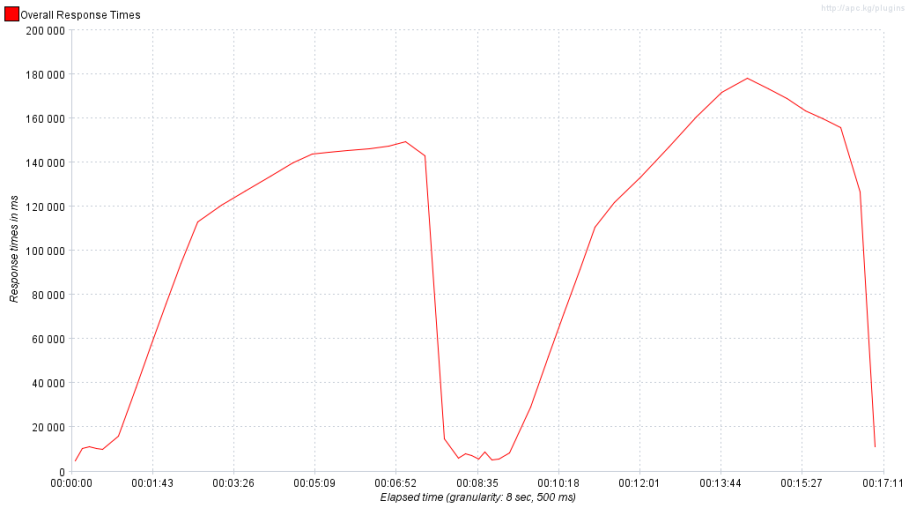

Figure C.3: Response times without Admission import

C.2 100% load simulating 250 concurrent users

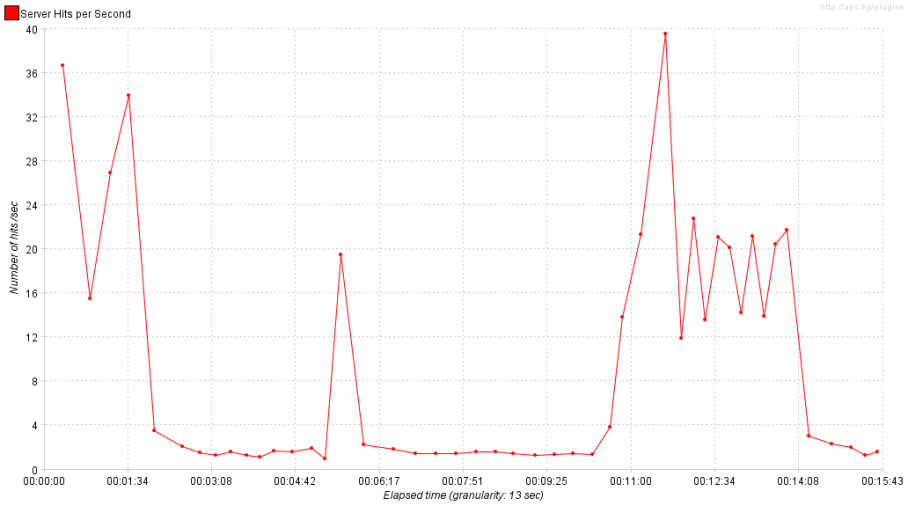

Figure C.4: Hits per second with Admission import

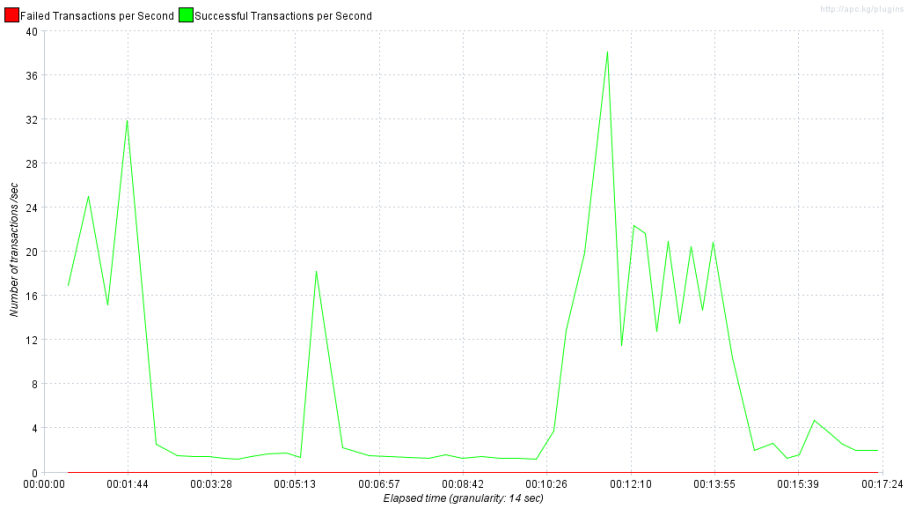

Figure C.5: TPS with Admission import

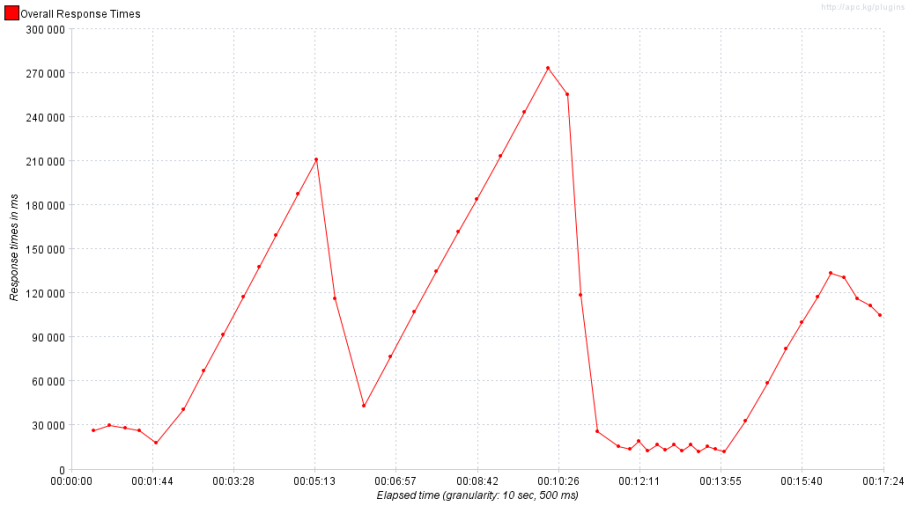

Figure C.6: Response times with Admission import

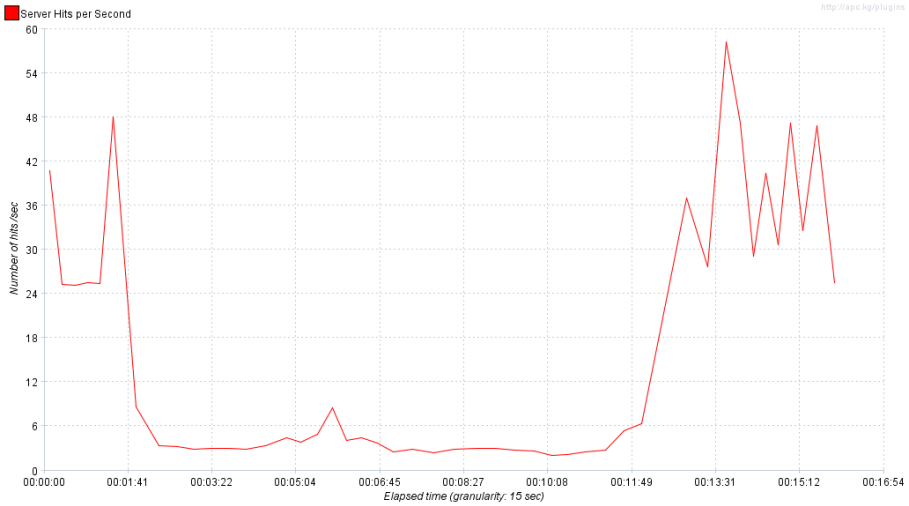

Figure C.7: Hits per second without Admission import

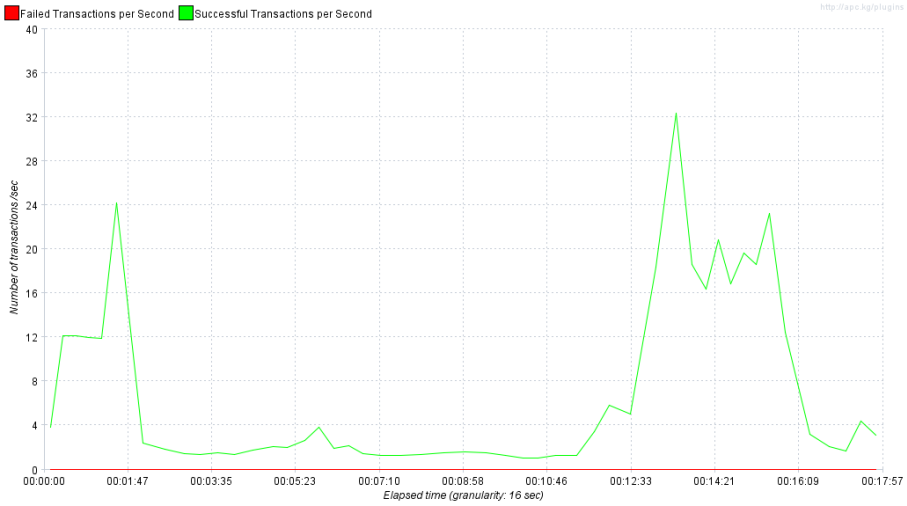

Figure C.8: TPS without Admission import

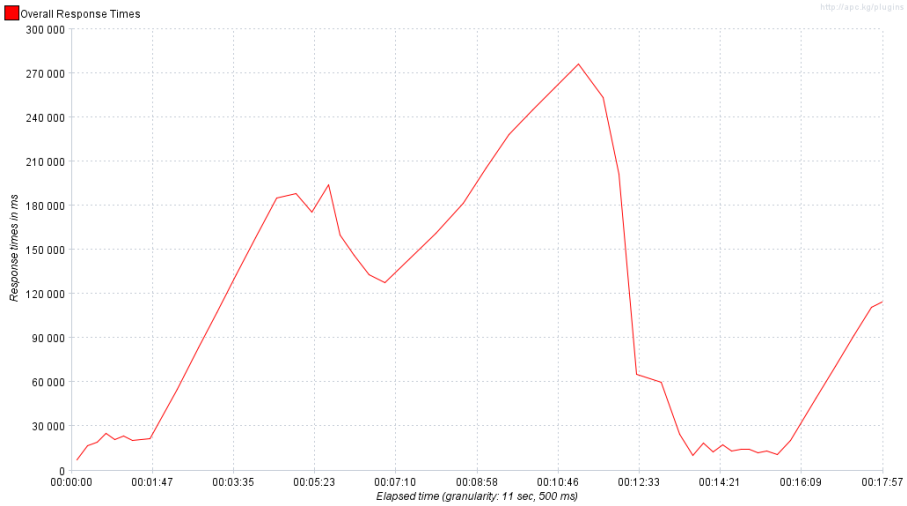

Figure C.9: Response times without Admission import

APPENDIX D

Admission processes in BPMN 2.0

D.1 Bachelor's main process

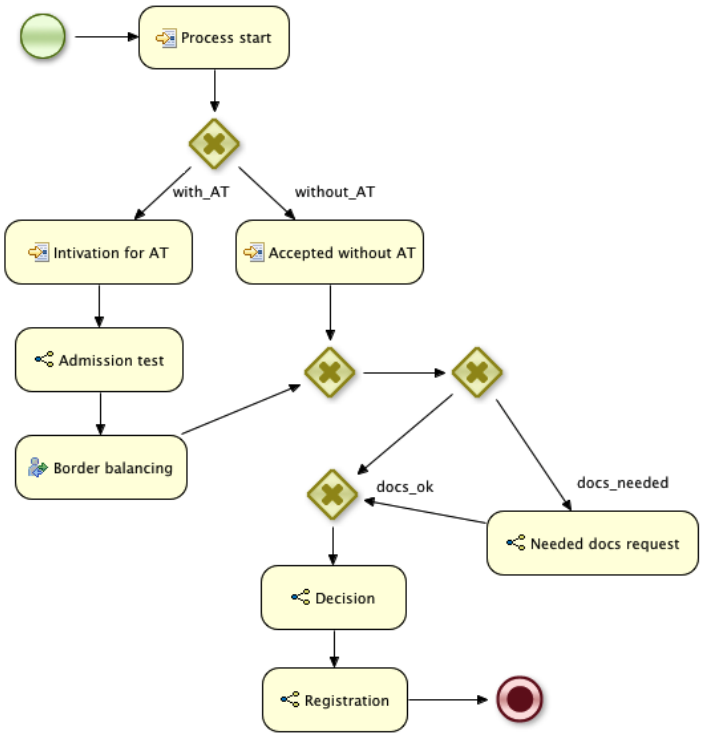

Figure D.1: Bachelor's admission process for 2012

D.2 Masters's main process

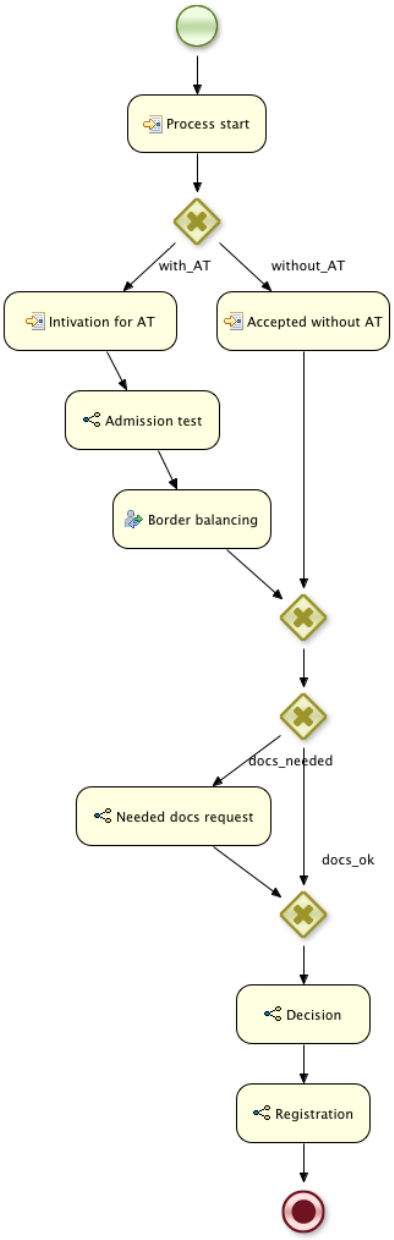

Figure D.2: Master's admission process for 2012

D.3 Shared processes

With closer look at both main processes, it is clear, that they basically share everything. They just differ in evaluation, which is parametrized. The rest of the processes is shared across both main ones.

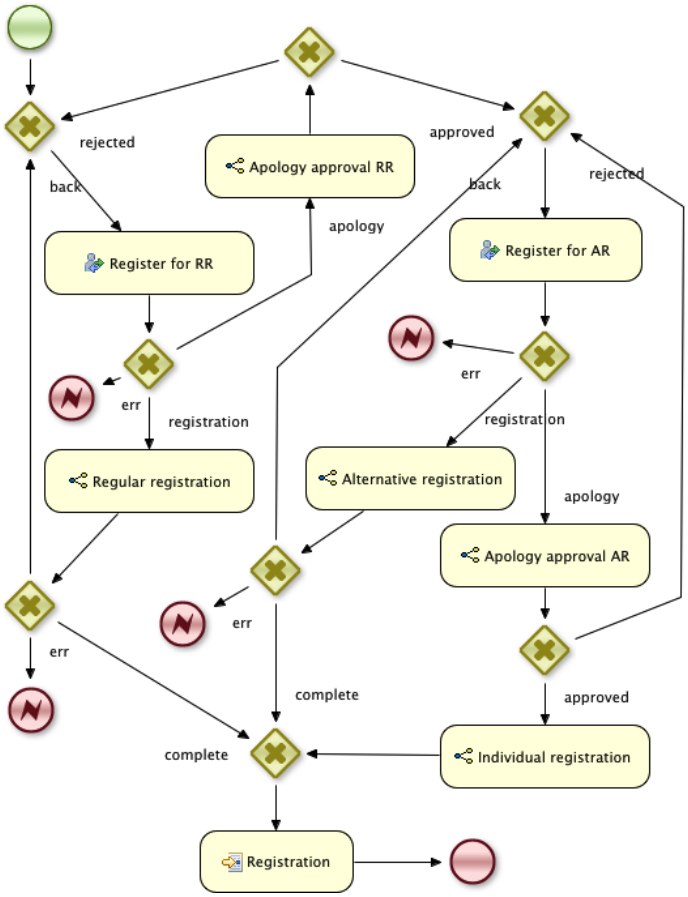

Figure D.3: Registration for 2012

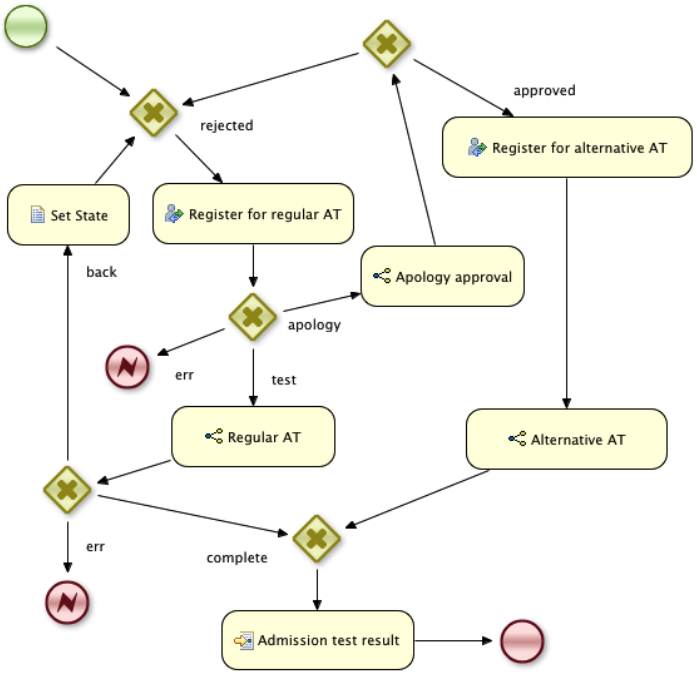

Figure D.4: Admission test for 2012

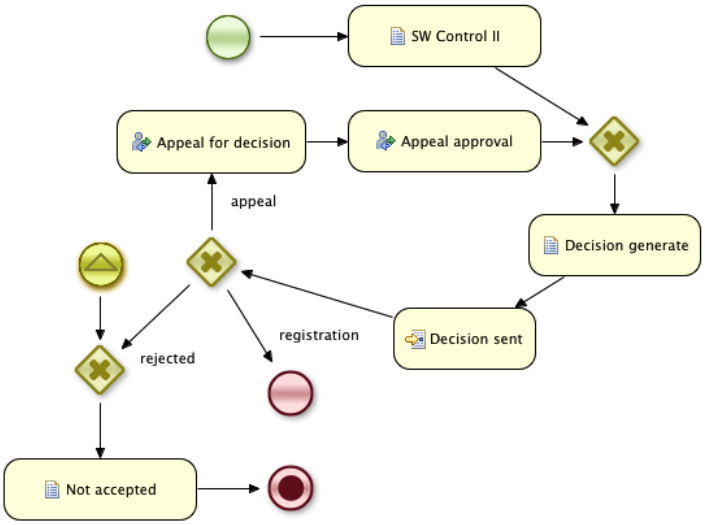

Figure D.5: Decision for 2012

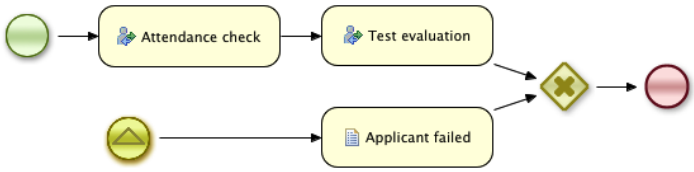

Figure D.6: Test for 2012

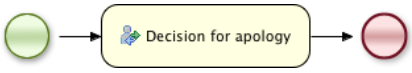

Figure D.7: Apology approval

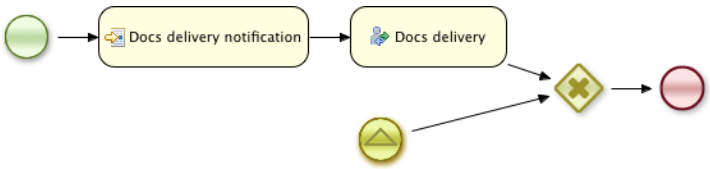

Figure D.8: Document request

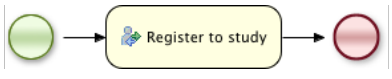

Figure D.9: Register to study

APPENDIX E

Content of CD

```
.
‘-- MI-MPR-DIP
|   ‘-- appendices
|   |   ‘-- appendix_01.tex
|   |   ‘-- appendix_02.tex
|   |   ‘-- appendix_03.tex
|   |   ‘-- appendix_04.tex
|   ‘-- chapters
|   |   ‘-- chapter_00.tex
|   |   ‘-- chapter_01.tex
|   |   ‘-- chapter_02.tex
|   |   ‘-- chapter_03.tex
|   |   ‘-- chapter_04.tex
|   |   ‘-- chapter_05.tex
|   |   ‘-- chapter_06.tex
|   |   ‘-- chapter_07.tex
|   ‘-- DP_Ondrusek_Jan_2012.pdf
|   ‘-- DP_Ondrusek_Jan_2012.tex
|   ‘-- figures
|   |   ‘-- bpm
|   |   ‘-- cvut-logo-bw.eps
|   |   ‘-- cvut-logo-bw.pdf
|   |   ‘-- eclipse_drools.png
|   |   ‘-- fis_architecture.png
|   |   ‘-- jbpm_process.png
|   |   ‘-- jmeter
|   |   ‘-- pririz_project_component.png
|   |   ‘-- restful_api_architecture.png
```

```
|   `-- FITthesis.cls
|   `-- mybibliographyfile.bib
|   `-- README.md
|   `-- rebuild.sh
`-- MI-MPR-DIP-Admission
|   `-- examples
|   `-- jmeter
|   `-- pom.xml
|   `-- README.md
|   `-- schema.xsd
|   `-- src
|       `-- main
|       `-- test
`-- parent
|   `-- pom.xml
|   `-- README.md
`-- test-templates
    `-- pom.xml
    `-- README.md
    `-- src
        `-- main
        `-- test
```