University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

Karol Kański
Student no. 248362

Integration of services in the Mobility Project

Master's thesis
in COMPUTER STUDIES

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

June 2011

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Student and staff mobility is one of the most important activities in today's higher education. Number of students traveling to other universities for short-term studies is continuously increasing, which makes administration of this process more and more difficult. This is compounded by the fact that the whole process of exchanging students is handled in a traditional way, which involves a lot of paperwork and communication by phone, fax, and e-mail. Such bureaucracy can seriously slow down the growth of a number of exchanged students. IT infrastructure of higher education institutions allows to change this state, as it provides almost all needed resources. The only missing thing is a sensible, widely recognized standard and its software implementation.

This thesis documents an effort on filling this gap. This includes creating a data model related to student mobility and a prototype of a software solution enabling electronic exchange of the mobility data.

Keywords

student mobility, Erasmus programme, data exchange, SOA, REST, P2P, web services, WSDL, WADL, UDDI

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics

Subject classification

J. Computer Applications
J.1 Administrative Data Processing - *Education*
H. Information Systems
H.3 Information Storage and Retrieval
H.3.4 Distributed systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Student and staff mobility is one of the most important activities in today's higher education. Creating opportunities for personal growth, developing international cooperation between individuals and institutions, enhancing the quality of higher education and research, and giving substance to the European dimension are, in the opinion of Ministers responsible for Higher Education in the countries participating in the Bologna Process, the most important benefits that can be taken from this phenomenon. These features have also resulted in student and staff mobility becoming one of the most important priorities of the European Higher Education Area promoted by the Bologna Process.

The significance of this process is also confirmed by an amount of people and institutions involved in it. The most important European programme promoting mobility (called 'Erasmus') involves over 4000 higher education institutions (HEI) from Europe. The number of participating students exceeds 2 millions (about 213 thousands in academic year 2009/10). Moreover, long-term trend for this programme shows almost linear growth of a number of exchanged students. This number stimulated by the new Lifelong Learning Programme introduced by the European Commission will probably reach 3 millions by 2012. Consequently, handling student mobility demands from HEIs a lot of administration work.

Today, in the age of the Internet, web services and digital signatures, HEIs keep all the necessary data needed to run the studies, i.e. personal data, learning achievements data, accounting, etc. in an electronic form. However, they still exchange it on paper. In the case of student mobility a lot of information must be exchanged before mobility can take place: organization data, agreement and cooperation conditions, list of nominated students. Then, for each traveling student learning agreement, transcript of records, etc. have to be transferred. In the view of numbers given in the preceding paragraph handling mobility means sending the huge amount of letters, emails and faxes, which is time consuming and expensive. Moreover, the same data are entered into system multiple times (in the home organization and in the partner organization), which can generate errors.

The process of data exchange is a serious obstacle to exploiting the full potential of this phenomenon – students and academic teachers are deterred by all this bureaucracy. This is even more sad, because almost all HEIs have an IT infrastructure appropriate to solve the problem. Of course, the IT infrastructure is insufficient, because some work has to be done in order to make mobility without paper happen.

In order to create some standard regarding electronic data exchange in the context of mobility scenarios two European Higher Education consortia – Polish MUCI [MUCI] and Italian CINECA [CINECA] – initiated the Mobility Project: an agreement upon cooperation in developing a prototype which could eventually evolve into a mature standard equipped with a reference implementation [AMDR09]. A brief history of the Mobility Project can be found in Section 1.1 and the work presented in this document is a next step on a way to achieve this goal.

1.1. Previous work

As mentioned in the previous section, work presented in this document is done as part of the Mobility Project lead by the Rome Student Systems and Standards Group (RS3G, see [RS3G]). The main implementation work is done by the University of Warsaw. To acquaint the reader with what these two organizations have done in the project so far this section contains a brief list of the most important events in the project's history. The more comprehensive list can be found in [JMD10]. The history of the project is tightly connected with the workshops of EUNIS (European University Information Systems, [EUNIS]) and RS3G. Other important platforms for exchange of ideas are: teleconferences, wiki [WikiTeria] and WSDL Team mailing list [WSDLList].

This thesis is a continuation of R. Nagrodzki's master's thesis [Nag09] and therefore [Nag09] will be frequently mentioned. Mr. Nagrodzki identified business processes occurring at a Higher Education Institution, prepared a data exchange format and created a software package enabling mobility data exchange. In this thesis the software package prepared by R. Nagrodzki will be called the Mobility, while the one created during work on this thesis will be called the REST Mobility. The whole project will be called the Mobility Project.

The main project's milestones are:

1. June 2008, EUNIS in Aarchus – MUCI (Polish consortium) and CINECA (Italian consortium) decide to start the Mobility Project.
2. June 2009, EUNIS in Santiago de Compostela – First version of the mobility network (with two nodes) is presented (software version 1.0).
3. November 2009, Warsaw – R. Nagrodzki's master's thesis is finalized [Nag09] (software version 2.0).
4. December 2010, RS3G Workshop in Malaga – Technical security details are discussed and approved – SSL/TLS is chosen for security assurance.
5. March 2011, RS3G Coding Camp in Barcelona – Joint meeting of software developers and people working in International Relations Offices, extending network of mobility nodes.

The Mobility Project gathers many universities, companies and consortia. The most active ones are:

- CINECA, Italy,
- FS, Norway,
- HIS, Germany,
- KION, Italy,
- Ladok, Sweden,
- MUCI, Poland,

- OODI, Finland,
- SIGMA, Spain,
- SURF, Netherlands,
- University of Malaga, Spain,
- University of Porto, Portugal,
- VHS, Sweden.

1.2. Project goals

Generally speaking, the idea behind this project was to investigate the REST style of creating web applications (see Section 3.2.3) and to examine what advantages can the Mobility Project take from it. In order to achieve this intangible goal some more specific objectives were established.

The main one was to create an alternative RESTful implementation of the Mobility. By giving European Higher Education Institutions two solutions, which are almost identical with respect to functionality, but differ with regard to underlying technology and some other aspects, we can decide if REST is an appropriate style for creating mobility web services. Institutions may test both implementations and after that decide which one better suits their needs.

Having REST in mind some other minor goals were also set. These are:

1. Creating a data model for the Mobility Project– that model has been lacking and since REST is resource-oriented style it is quite important to have it defined.
2. Updating the data format and consequently WSDL.
3. Creating a description of the Mobility RESTful web services in a WADL format.
4. Exchanging UDDI registry with a RESTful registry suitable for both the Mobility and the REST Mobility.

There is also a set of goals, which are not strictly connected with REST but are important in the context of electronic data exchange:

1. Securing data transmission.
2. User authentication.
3. User authorization.

1.3. Overview

The rest of this thesis is structured as follows. Next chapter contains detailed requirements description and analysis with the main emphasis put on the data model related to student mobility. Chapter 3 concentrates on the architecture and design of the developed software system. Chapter 4 presents interesting aspects of the implementation of the system. The last chapter summarizes the thesis and gives some ideas on what can be done in the future in the Mobility Project.

The thesis contains two appendices: Appendix A provides a comprehensive documentation of the software, whereas Appendix B lists the contents of a CD media attached to the thesis.

Chapter 2

Requirements analysis

Much work in the area of requirement analysis was done by R. Nagrodzki in his master's thesis [Nag09]. This thesis makes use of the results presented there. However, that previous work lacks a clearly defined data model – there are only data types defined in a WSDL document. Moreover, the WSDL document has slightly changed since then (the most actual can be found at [MobWSDL]). There are also some extra non-functional requirements resulting from the fact, that this thesis is a continuation of [Nag09]. This chapter is intended to present the actual state of the requirement analysis for the Mobility Project. The main emphasis is put on the data model related to student mobility.

2.1. Basic terms

This section contains the most important terms related to international cooperation and student mobility. Definitions, terms and entities in this section mostly come from [Kra06] and [Lom08], but some of them have been changed to meet requirements defined in [Nag09]. They are listed here to make this work self-contained and provide a reader, who is only interested in the mobility data model, with all the necessary information.

2.1.1. Abbreviations

HEI – higher education institution.

IRO – International Relations Office.

SIMS – student information management system.

2.1.2. Definitions

Exchange Programme – set of purposes and tasks established in order to achieve certain results. From the Mobility Project point of view 'programme' stands for international academic exchange programme between HEIs. Examples of exchange programmes are: 'CEEPUS' or 'ERASMUS'. Programme can be part of another programme called 'master programme'.

Exchange Project – set of persons, institutions and undertakings characterized by the following features:

1. they are bind together in a complex way,

2. ways of achieving goal change, often by generating a unique service, product or result,

3. they have pre-planned budget and schedule.

Projects are created in exchange programmes and are funded by these programmes. Projects form a network of inter-university cooperation, whose structure is determined by the exchange programme. Home organization can play in exchange project one of three roles (called home organization status): Coordinator, Partner or Contractor.

Agreement – an arrangement between two or more parties (organization, country, person, organizational unit), that settles their rights and responsibilities.

Academic Period – part of academic year (e.g. winter semester, first trimester, academic year).

Conditions (of cooperation) – part of agreement defining rights and responsibilities in the area affected by it.

Contractor – person (organization's employee) responsible for project's funds. At the University of Warsaw it is usually vice-rector responsible for international cooperation.

Coordinator – person (organization's employee) responsible for realization of a given undertaking (programme, project, agreement) at this undertaking level (e.g. agreement coordinator). At the University of Warsaw this can be employee of faculty involved in the undertaking.

Institutional Coordinator – person (organization's employee) responsible for realization of a given undertaking (programme, project, agreement) at the level of organization. At the University of Warsaw it is usually head of IRO.

Organization – organization that is taking part in the mobility process. In most cases it will be HEI but it can be any other organization/institution (e.g. bank where student's internship takes place).

Supervisor – research fellow (academic fellow) assigned by the dean/head for scientific contacts (assistance in selecting courses, creating research programme etc.).

Tutor – person assigned by the dean/head for assistance in organizing foreigner's stay (dealing with administrative issues, medical attention etc.).

2.1.3. Home and partner organizations

R. Nagrodzki in his work (see [Nag09], Section 1.2) uses adjectives 'home' and 'partner' in the following meaning:

> **home** – denotes institution which sends its students; a sending institution,
>
> **partner** – denotes institution at which mobility students arrive and stay for a study during mobility; a host institution.

In this work meaning of these adjectives is different:

home – denotes the organization deploying Mobility, from which point of view activities related to international cooperation (including student and employee mobility) are performed (e.g. if we are acting on behalf of the University of Warsaw then it is called 'home organization' and even if we are dealing with a German student who studies in Warsaw then 'home supervisor' is a staff member from the University of Warsaw);

external or partner – denotes the other organization taking part in the mobility process.

For example, when a list of students from the University of Warsaw nominated for studies at the University of Parma is being sent from Warsaw to Parma, then the University of Warsaw is 'home organization' and the University of Parma is 'partner organization'. However when Warsaw sends to Parma transcripts of records of Italian students who have spent a semester in Warsaw then the University of Parma is 'home organization' and the University of Warsaw is 'partner organization' (context of information is important, transcripts of records being sent contain grades of Italian students for whom the University of Parma is the home institution).

2.2. Business processes

Business processes occurring at a HEI have not changed since 2009 and therefore readers interested in them should read [Nag09].

2.3. Data model

Most of the informational requirements were gathered by M. Krawczyński in his master's thesis (see [Kra06]). The following sections present these requirements from the Mobility Project point of view.

2.3.1. Informational requirements

Student mobility data model can be divided into three main parts: agreements, arrivals, departures. Informational requirements for each of them are presented in the next three sections.

Agreements

The exchange of persons is possible only on the basis of signed agreements. Each agreement is characterized by the following attributes (see also Figures 2.1 and 2.2):

1. **number** – text uniquely identifying the agreement at home institution,
2. **start date** as originally planned,
3. **end date** as originally planned,
4. **date signed** – can be different from the start date,
5. **status** – describing current state of the agreement (e.g. signed, not signed by a partner),

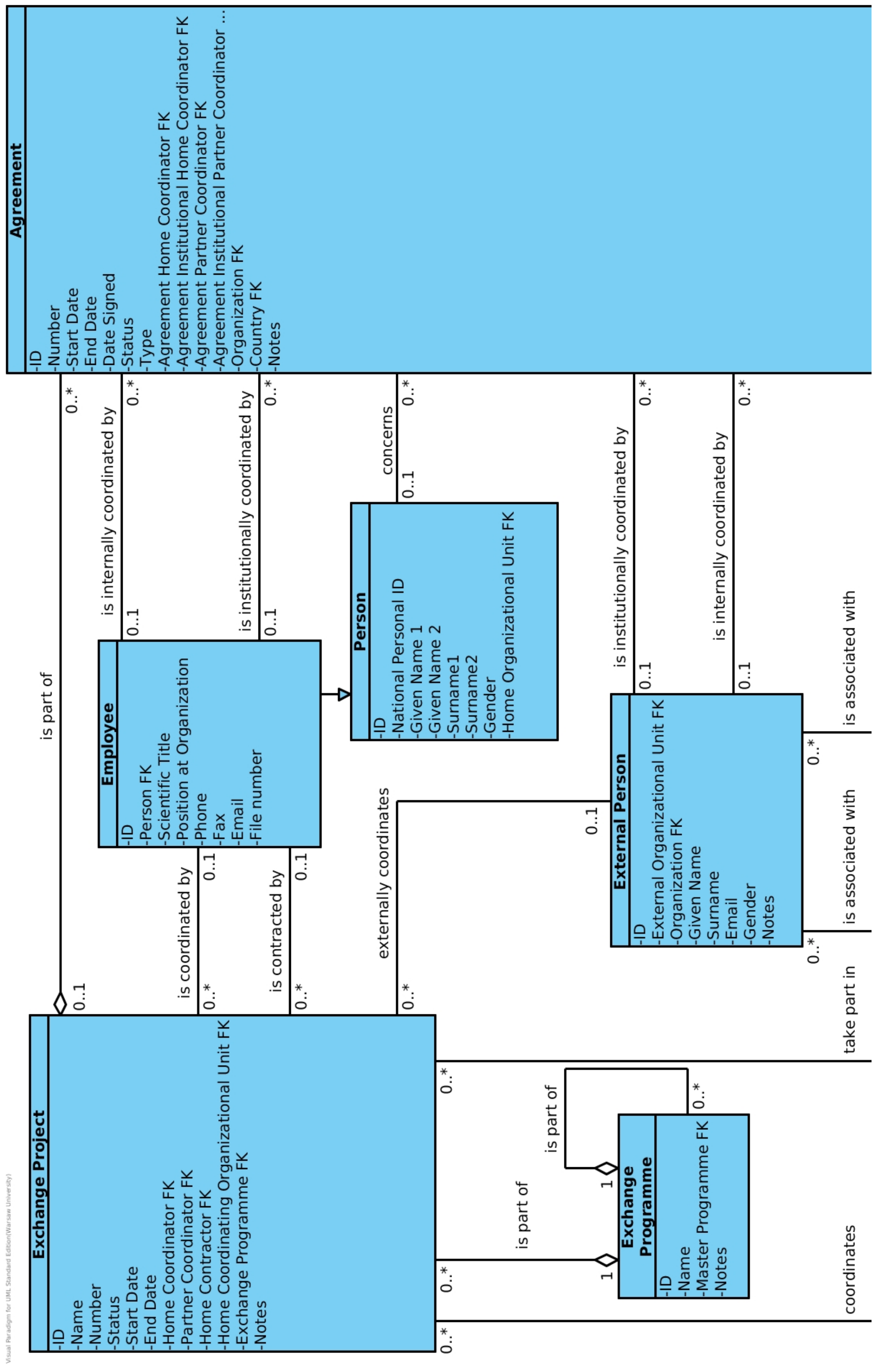


Figure 2.1: Agreements – part I

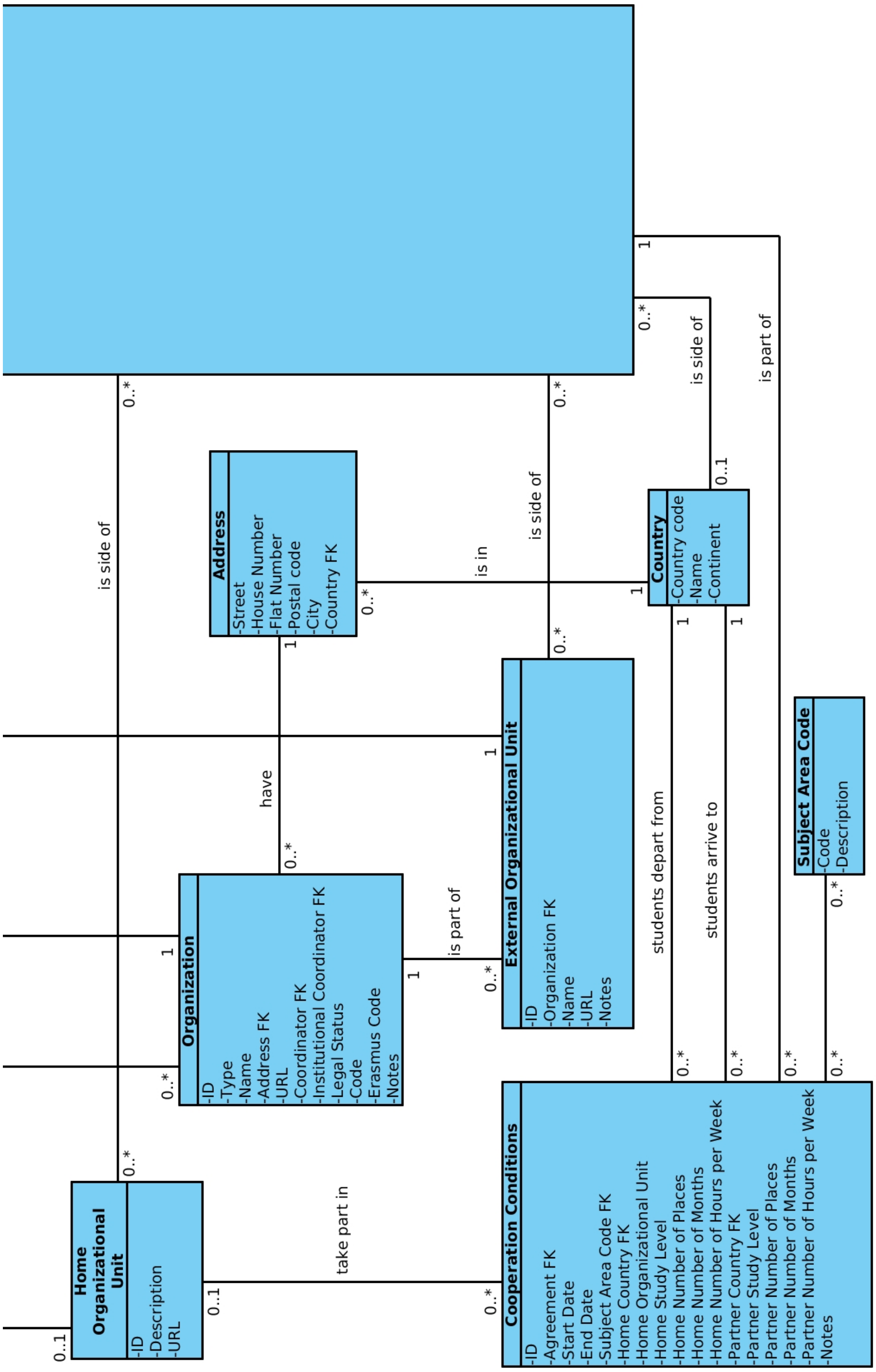


Figure 2.2: Agreements – part II

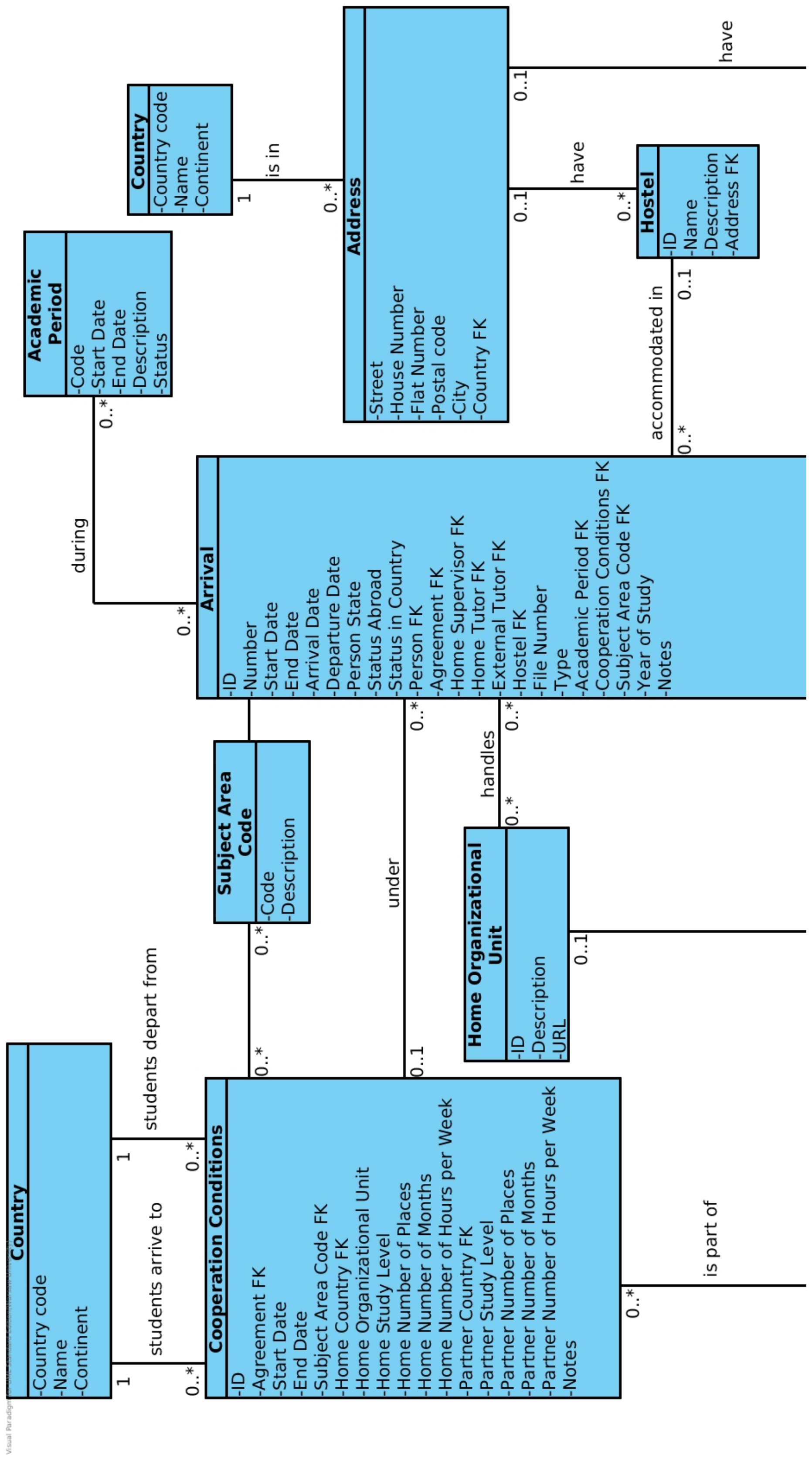


Figure 2.3: Arrivals – part I

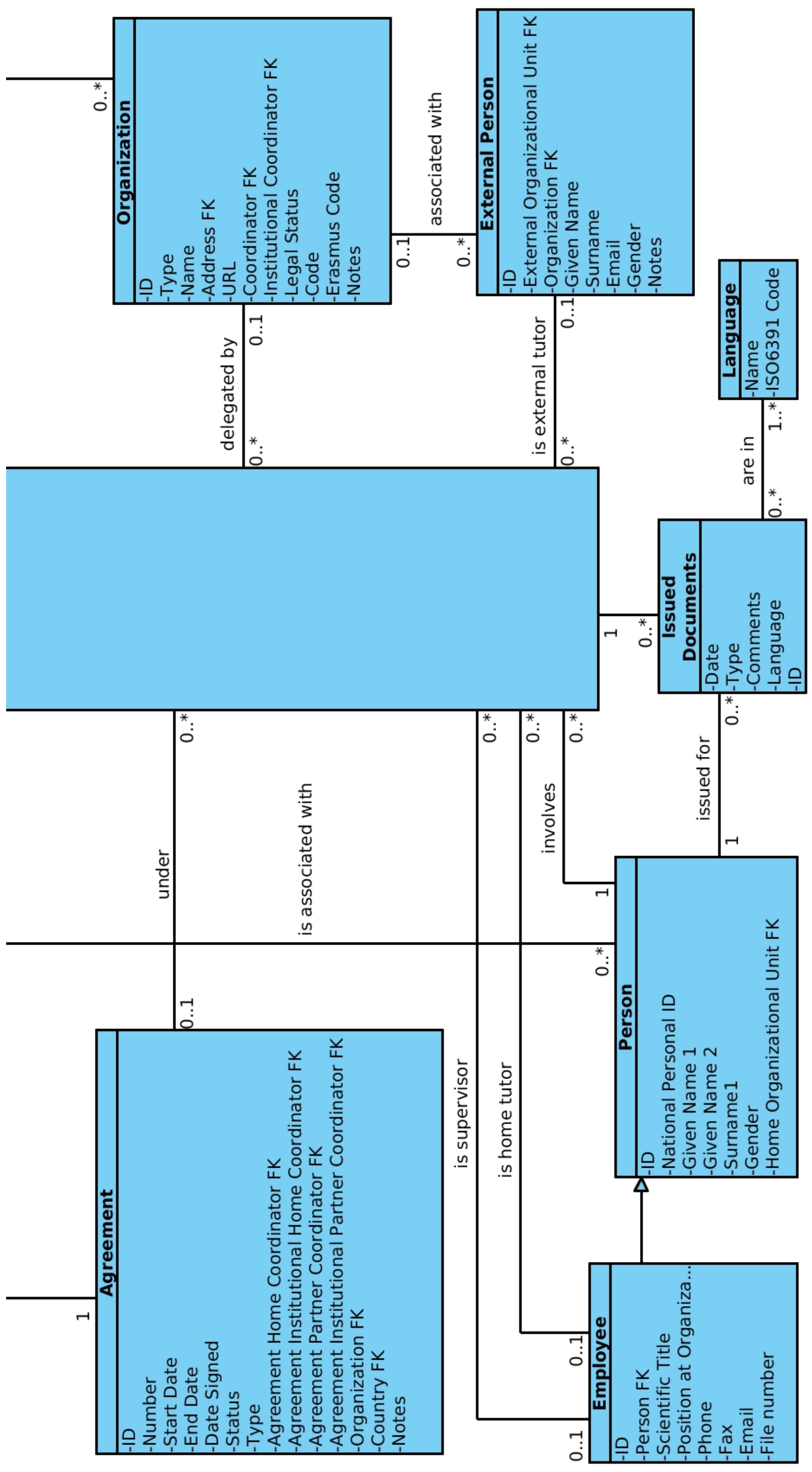


Figure 2.4: Arrivals – part II

6. **agreement home coordinator**,
7. **agreement home institutional coordinator**,
8. **agreement partner coordinator**,
9. **agreement partner institutional coordinator**,
10. **home organizational units** – sides of agreement,
11. **partner organizational units** – sides of agreement,
12. **agreement type** – IRO at the University of Warsaw defines five types of agreements and further informational requirements for each type:
    (a) **bilateral** agreement: **partner organization** – side of the agreement, organization's **country** of origin,
    (b) **governmental** agreement: **country** – side of the agreement,
    (c) **individual** agreement: **external person** – side of the agreement, **organization** associated with this person,
    (d) agreement under **exchange project**: **exchange project** under which the agreement is signed,
    (e) **faculty** agreement: **partner organization** – side of agreement, organization's **country** of origin.
13. **cooperation conditions** – defined for all types of agreements except individual agreements. Should contain the following information:
    (a) cooperation's **start** and **end dates**,
    (b) **organizations** involved in the agreement,
    (c) **organizational units** of the organizations, which are involved in the agreement,
    (d) **countries** involved in the agreement,
    (e) **subject area codes**,
    (f) **programme participants** (e.g. students, Ph.D. students etc.),
    (g) **home and partner level of studies** (if cooperation involves students),
    (h) **number of persons** that can be involved in exchange under the agreement,
    (i) **number of months** programme participants can spent at partner institution,
    (j) **number of weeks** programme participants can spent at partner institution (usually applies to academic teachers and researchers),
    (k) **number of hours** programme participants can spent at partner institution (usually applies to academic teachers and researchers).

Arrivals

An arrival always involves only one person and only one agreement. Each arrival is characterized by the following attributes (see also Figures 2.3 and 2.4):

1. **number** – text uniquely identifying the arrival at home institution (hosting the arrival),

2. **arriving person**,
3. **start date**,
4. **end date**,
5. **arrival date** – can be different from start date,
6. **departure date** – can be different from end date,
7. **state of the arriving person** – describing state of the arriving person, e.g. qualification for the arrival, arrival, resignation from the arrival,
8. **status of the arriving person at home organization** – e.g. student or Ph.D. student,
9. **status of the arriving person at partner organization** – e.g. student or Ph.D. student,
10. **accommodation** – information about accommodation of the arriving person (organization's hostel, on his/her own),
11. **agreement** under which the person arrives,
12. **cooperation conditions** associated with the arrival (only if they are defined in the agreement),
13. **purpose of arrival** e.g. studies, language course, summer school, internship, research, lectures,
14. **external organization** – associated with the arriving person,
15. **external tutor** – tutor from the external organization,
16. **home supervisor**,
17. **home tutor**,
18. **home organizational unit** hosting the arrival,
19. **documents** issued to the arriving person,
20. **file number** (of the paper documentation),
21. **year of study** of the arriving student,
22. **academic period** during which arrival takes place (e.g. winter semester 2010/2011).

Departures

A departure always involves only one person and only one agreement. Each arrival is characterized by the following attributes (see also Figures 2.5 and 2.6):

1. **number** – text uniquely identifying the departure at home institution (sending student),
2. **leaving person**,

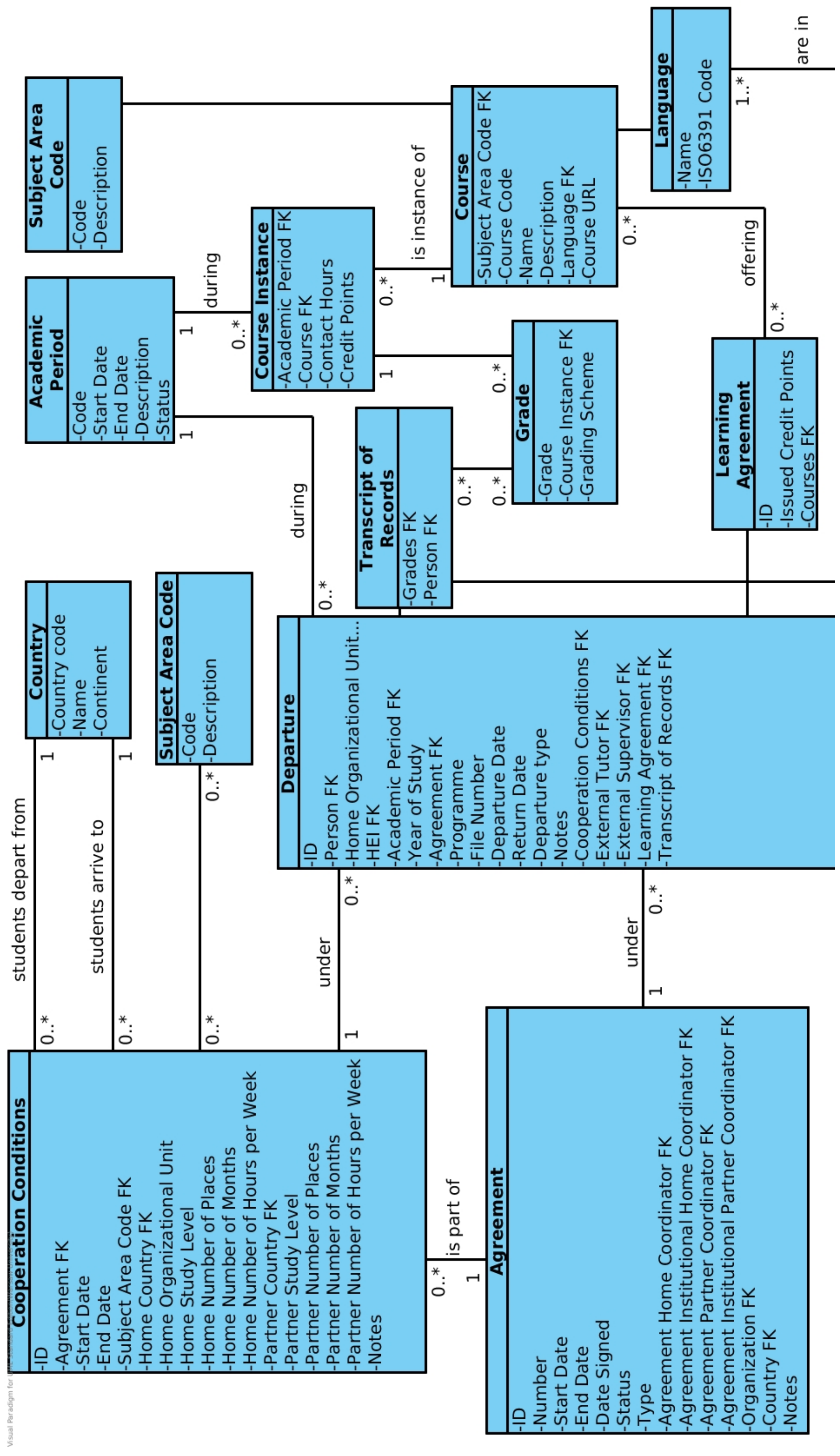


Figure 2.5: Departures – part I

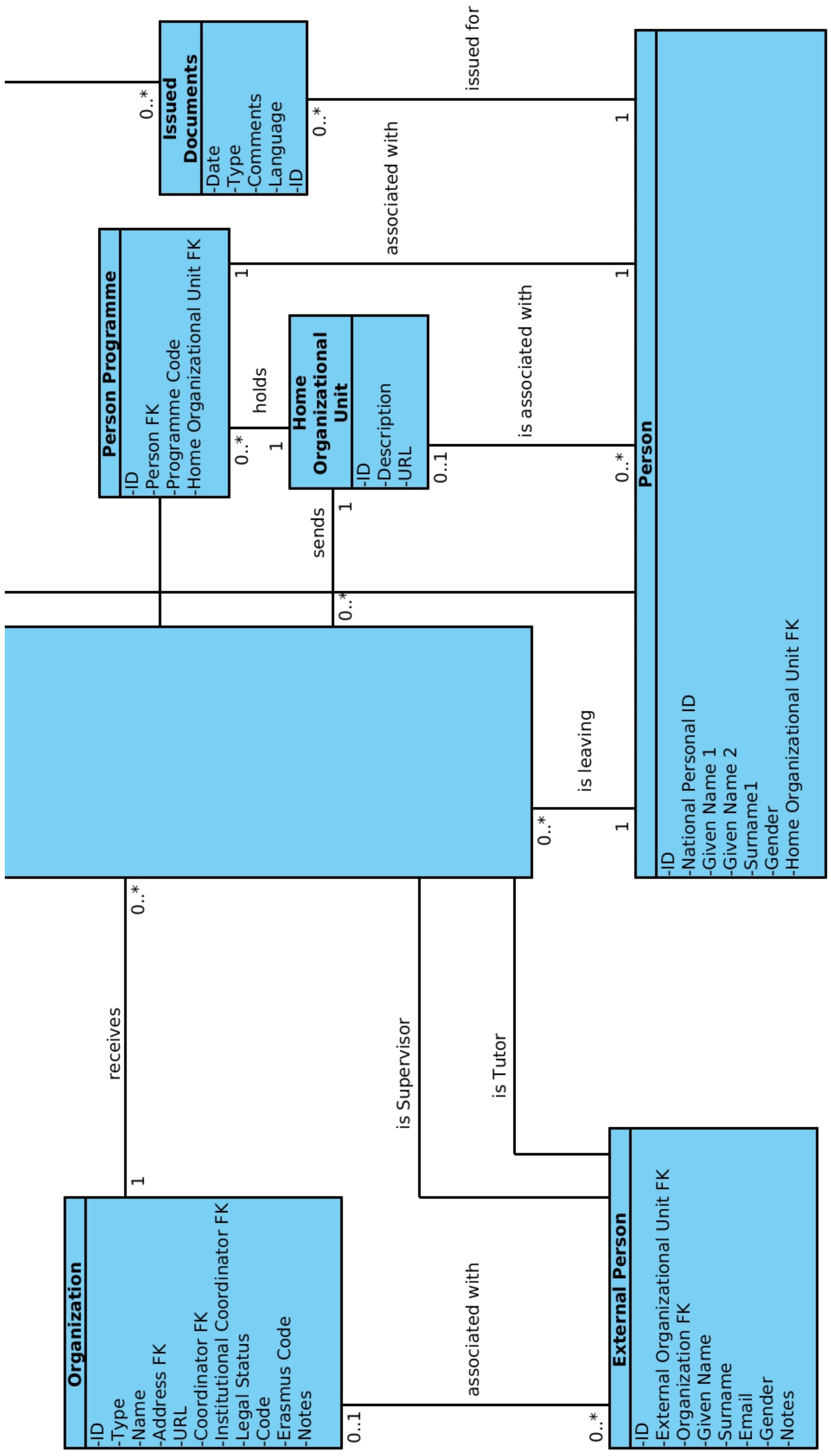


Figure 2.6: Departures – part II

3. **home organizational unit** sending the leaving person,
4. **academic period** during which the departure takes place (e.g. winter semester 2010/2011).
5. **programme** of studies of the leaving person,
6. **HEI** to which student is going,
7. **agreement** under which the person is leaving,
8. **cooperation conditions** under which the person is leaving,
9. **type of departure** e.g. studies, language course, summer school, internship, research, lectures,
10. **file number** (of the paper documentation),
11. **year of study** of the leaving student,
12. **departure date**,
13. **return date**,
14. **external supervisor**,
15. **external tutor**,
16. **learning agreement**,
17. **transcript of records**.

2.3.2. Mapping on Mobility WSDL

This section presents correspondence between the data model introduced in the previous section and the data types used in the Mobility Project. The later ones are well described in [Nag09] and their technical definitions in the XML format, which has slightly evolved since 2009 (see also discussion at [RS3GDiscussion]), are available at [MobWSDL]. Data type definitions in the Mobility WSDL document are divided into some groups and mapping of each of these groups is shown in Tables 2.1 – 2.6. The group 'general auxiliary elements and element groups' is omitted because it contains only data types introduced to make transfer of data easier. Groups with names starting from 'get/send' are merged into one group shown in Table 2.6. In the following tables the first column contains entities from the presented data model, the second one elements from Mobility WSDL, while the third one provides some additional comments.

2.3.3. Other Models

The mobility data model presented in this document does not intend to be standard. Its purpose is to show how such model can look like. In fact this model is in use at the University of Warsaw where it works well. Some parts of it may be influenced by UW standards and EU law, so may have to be changed while adopting it at another organization, especially in countries not belonging to the EU. Another examples of such data model can be found in [Van10] and [Hea10] (U.S. standards).

Data Model	WSDL	Comment
Type: *Country*, field: *Country Code*	countryCodeT	
Type: *Language*, field: *ISO6391 Code*	languageCodeT	
NA	nameT	This type is introduced in WSDL to constrain available content of some elements.
NA	countryCodeorIntT	This type is introduced in WSDL to constrain available content of some elements.
NA	internationalizedStringT	This type is introduced in WSDL to constrain available content of some elements.
NA	domainT	This type is introduced in WSDL to constrain available content of some elements.
Nothing	academicPeriodT	It is not equivalent to *Academic Period* type in the data model, see academicPeriodSinceT.
NA	nonNegativeFloatT	This type is introduced in WSDL to constrain available content of some elements.
NA	positiveIntT	This type is introduced in WSDL to constrain available content of some elements.
Type: *Academic Period*	academicPeriodSinceT	See academicPeriodT.
Type: *Address*	addressT	
NA	emailT	

Table 2.1: General types

Data Model	WSDL	Comment
Type: *Organization*, field: *Type*	organizationTypeT	
Type: *Organization*, field: *URL*	organizationIdT	In the Mobility Project URL is used to identify organizations.
Type: *Organization*, field *URL* + type: *Person / External Person*, field: *ID*	organizationalPersonalIdT	WSDL documentation: unique identifier of a person within an organization.
Type: *Organization*	organizationDataT	

Table 2.2: Organization related types

Data Model	WSDL	Comment
Type: *Person*, field: *National Personal ID*	nationalPersonalIdT	
Type: *Person*, field: *Gender*	genderT	It is a data type in WSDL and an attribute in the data model, but they both refer to the same portion of information.
Type: *Person / External Person*	personalCharacteristicsT	In the data model there is no distinction between a person and a student, while in the WSDL there is *Person* type and its two subtypes: *Employee* and *Student*. In the data model there is a distinction between a person and an external person, which is not present in the WSDL. That is why in most cases one WSDL type can be equivalent to *Person* type and to *External Person* type.
Type: *Employee*, field *Position at Organization*	personalPositionT	
Type: *Employee / External Person*	employeePersonal-CharacteristicsT	See comment to personalCharacteristicsT.
Type: *Person / External Person*	studentPersonal-CharacteristicsT	See comment to personalCharacteristicsT.

Table 2.3: Personal data related types

Data Model	WSDL	Comment
Type: *Subject Area Code*	subjectAreaCodeT	
Type: *Course*, field: *Course Code*	courseCodeT	
Type: *Learning Agreement*, field: *Issued Credit Points*	studyCreditsT	
Type: *Course Instance*, field: *Contact Hours*	contactHoursT	
Type: *Course*	courseDataT	
Type: *Course Instance*	courseInstanceT	
Type: *Grade*	gradeT	
Type: *Cooperation Conditions*, field: *Home/External Study Level*	studyLevelT	
NA	academicYearT	This information is incorporated into *Academic Period* type.

Table 2.4: Course related types

Data Model	WSDL	Comment
Type: *Agreement*, field: *ID*	localAgreementIdT	
Type: *Agreement*, field: *Number*	agreementIdT	
Type: *Cooperation Conditions*, field: *ID*	cooperationConditionsIdT	
Type: *Cooperation Conditions*	cooperationConditionsT	

Table 2.5: Agreement related types

Data Model	WSDL	Comment
Type: *Agreement*	agreementData	Part of agreement information that does not include personal data.
Type: *Agreement*	agreementDataWith-PersonalData	
Type: *Arrival*, field: *Arrival date* / type: *Departure*, field: *Departure date*	arrivalDate	It depends on the context - for home organization it is departure and for partner organization it is arrival.
Type: *Arrival*, field: *Arrival date* / type: *Departure*, field: *Departure date*	studentArrivalDate	See comment to arrivalDate.
Type: *Arrival*, field: *Departure date* / type: *Departure*, field: *Return date*	departureDate	It depends on the context - for home organization it is return from departure and for partner organization it is departure.
Type: *Arrival*, field: *Departure date* / Type: *Departure*, field: *Return date*	studentDepartureDate	See comment to departureDate.
Type: *Transcript of Records*	courseInstancesWith-PersonalData	
Type: *Transcript of Records*	gradesFromCoursesWith-PersonalData	

Table 2.6: Elements used in get/send operations

2.4. Requirements specification

Software developed during this master project provides functionality identical to this presented in [Nag09] and works in similar environment. That is why lists of functional requirements and assumptions are the same as those in [Nag09]. The list of non-functional requirements has undergone bigger changes. Some requirements are added, some are deleted, some are changed and therefore the whole list is presented.

2.4.1. Functional requirements

See "Functional requirements" section in [Nag09].

2.4.2. Non-functional requirements

1. Solution should conform to REST style architecture guidelines (see also Section 3.2.3).
2. Data exchange format used by the Mobility (the one presented in [Nag09] with later changes – see discussion at [RS3GDiscussion]; current version can be found at [MobWSDL]) should be used without any major changes.
3. Software should act as a complete, generic node being able to carry out transmission on its own.
4. Solution should be scalable.
5. Data transmission must be secure, i.e. it cannot be intercepted and/or modified, data integrity must be preserved. Way of securing data transmission should be the same in the Mobility and the REST Mobility.

6. Way of user authentication should be the same in the Mobility and the REST Mobility.
7. Personal data exchange legal issues should be considered.
8. Software should be relatively easy to install and maintain.
9. Tools and technologies used to implement the software should be:
   (a) freely available; preferably licensed in a way which allows redistribution – ideally the licenses should be open source compatible,
   (b) maintained and widely used; with strong, active community,
   (c) independent of any specific OS or hardware architecture.
10. Solution should use RESTful registry prepared for GEMBus (see [GEMBus]) instead of UDDI.

2.4.3. Assumptions

Assumptions are identified in [Nag09].

Chapter 3

System architecture and design

This chapter starts with description of Service Oriented Architecture (SOA). Then various aspects of SOA realization (WS-* web services, RESTful web services, services registry, service description languages, etc.) are presented. The rest of this chapter is devoted to the utilization of SOA in the REST Mobility. This includes: URL scheme created for the Mobility RESTful web services and the REST Mobility software architecture.

3.1. Service Oriented Architecture

This section presents architectural style called Service Oriented Architecture (SOA). Service Oriented Architecture has been adopted both in the Mobility and in the REST Mobility, but the ways of realizing SOA were different in these two projects. The idea of exchanging mobility data through web services comes from [AMDR09].

3.1.1. Definition. Main principles.

The OASIS Organization (see [OASIS]) defines Service Oriented Architecture (SOA) as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. Understanding this definition is a little hard. The simpler one can be: SOA is a software architecture paradigm for building enterprise applications, which is based on the concept of service. SOA has emerged from older concepts of object-oriented programming and distributed computing.

Service Oriented Architecture Principles

Like each architectural style, SOA also has its guiding principles which define the ground rules for development, maintenance, and usage. The [SOAWiki] describes SOA principles in the following words:

> "One of the first publicly published research of SOA was made by Thomas Erl of SOA Systems Inc. Erl defined eight service-orientation principles and published them in "Service-Oriented Architecture: Concepts, Technology, and Design", on the www.soaprinciples.com research site, and in the September 2005 edition of the Web Services Journal. These are:
>
> - **Standardized Service Contract** – Services adhere to a communications agreement, as defined collectively by one or more service-description documents.

- **Service Loose Coupling** – Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- **Service Abstraction** – Beyond descriptions in the service contract, services hide logic from the outside world.
- **Service Reusability** – Logic is divided into services with the intention of promoting reuse.
- **Service Autonomy** – Services have control over the logic they encapsulate.
- **Service Statelessness** – Services minimize resource consumption by deferring the management of state information when necessary.
- **Service Discoverability** – Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- **Service Composability** – Services are effective composition participants, regardless of the size and complexity of the composition."

For many people the most important principle of SOA is 'Service Interoperability', so its absence on the list above may seem strange. However, Erl has a good explanation for that fact. In his opinion, interoperability is fundamental to every one of the principles listed above. Therefore, in relation to service-oriented computing, stating that services must be interoperable is just about as evident as stating that services must exist.

3.1.2. Components

The idea of service has been present in people's life for centuries. Any person performing some task on the other person request is providing a service. Some processes can be divided into a single tasks, carried by different agents. Each agent has a clearly defined responsibility. Agents are interested in what can be done for them by others and are able to learn it, but are not interested in how it will be done. These concepts have been transferred to computer science. According to [KBS04], four main elements can be identified in the SOA (see Figure 3.1):

Application frontend

Application frontends are the active parts of SOA. They are invoking services and gathering results. The most obvious examples of applications frontends are GUIs.

Services

A service is a software component providing some business functionality. It consists of a few parts:

- contract – an informal specification of the purpose, functionality, constraints, and usage of the service. See also Section 3.3.
- interface – by interface the functionality of the service is exposed to the clients that are connected to the service by network.
- implementation – technical realization fulfilling the service contract. There are two main aspects of service implementation:

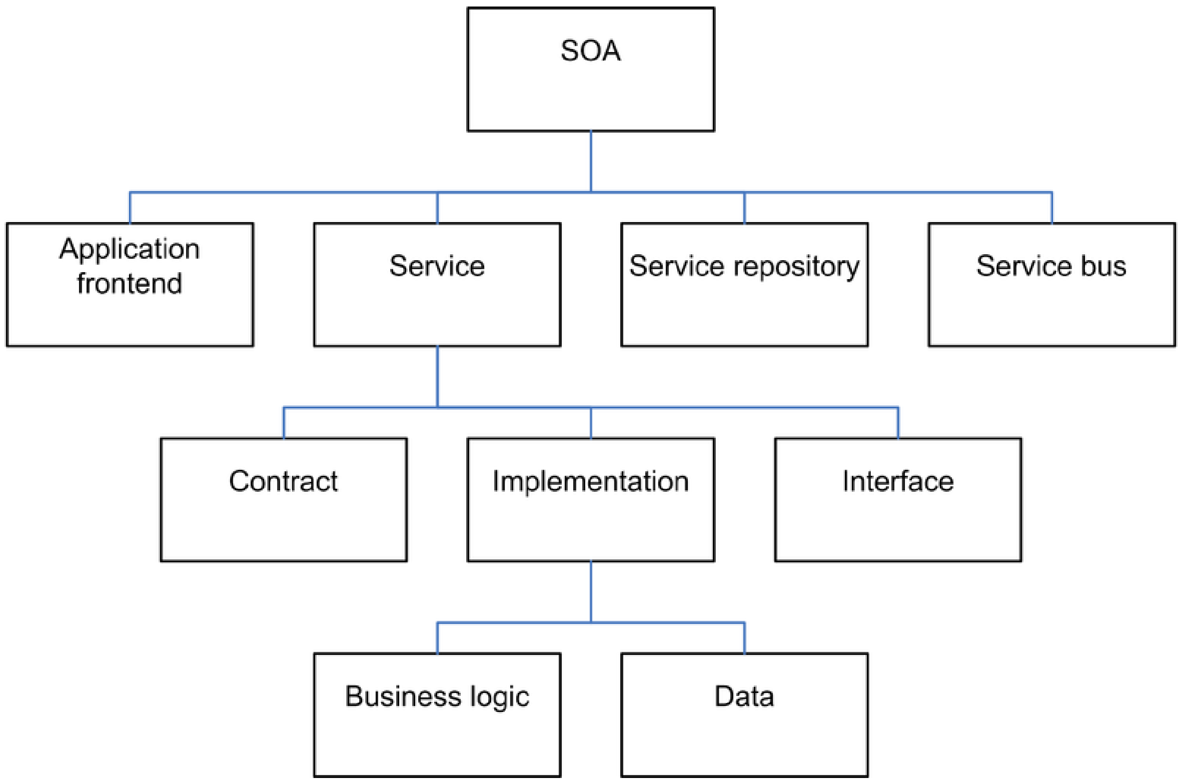


Figure 3.1: SOA elements. Source: [KBS04]

- data,
- logic.

Service repository

A service repository provides facilities to discover services and acquire all information to use the services. Much of this information is already part of the service contract, so in practice repository is closely connected to the service contracts. More about these two elements can be found in Section 3.3.

Service Bus

The most important task performed by a service bus is connecting all participants of the SOA, services and application frontends with each other. However, its current implementations called Enterprise Service Bus are performing many additional tasks. The most important can be found in Table 3.1.

3.1.3. Building SOA

Adopting the SOA in an application built from scratch should be started in the design phase. Apart the fact that it is complex and application-specific process, it is not the case of the Mobility Project. In the Mobility Project there are many local systems who need to communicate and exchange data with each other. Introducing the SOA in such case consists of the following steps (see also Figure 3.2):

Category	Functions
Invocation	support for synchronous and asynchronous transport protocols, service mapping (locating and binding)
Routing	addressability, static/deterministic routing, content-based routing, rules-based routing, policy-based routing
Mediation	adapters, protocol transformation, service mapping
Messaging	message-processing, message transformation and message enhancement
Process choreography	implementation of complex business processes
Service orchestration	coordination of multiple implementation services exposed as a single, aggregate service
Complex event processing	event-interpretation, correlation, pattern-matching
Other quality of service	security (encryption and signing), reliable delivery, transaction management
Management	monitoring, audit, logging, metering, admin console, BAM (business activity monitoring)

Table 3.1: ESB tasks. Source: [ESBWiki]

1. The service provider should:
   (a) create service;
   (b) describe it in some service description language;
   (c) publish this description in some services registry.
2. The service requester should:
   (a) search for the service description in some services registry;
   (b) download the service description;
   (c) use the service.

3.1.4. Benefits

Thomas Erl identified several benefits resulting from adopting SOA (see [WISOA]):

- Increased Intrinsic Interoperability – interoperability is natively established within services, so the need for integration is reduced.
- Increased Federation – creating federated IT environment (the one where resources and applications are united while maintaining their individual autonomy and self-governance) is natural in SOA.
- Increased Vendor Diversification Options – loose-coupling of services allows changing, extending and replacing solution implementations and technology resources without disrupting the overall, federated service architecture.
- Increased Business and Technology Alignment – pre-existing representations of business logic (business entities, business processes) can exist in implemented form as physical services.

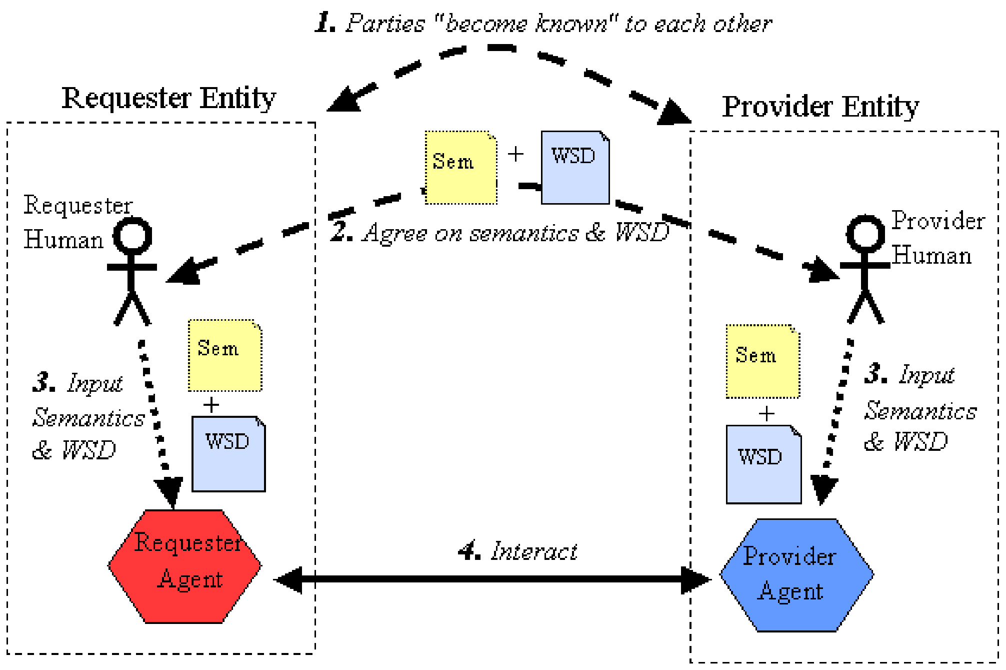


Figure 3.2: Building SOA. Source: [W3C]

- Increased ROI – creation of reusable software results in increased return on investment.
- Increased Organizational Agility – creation of services that are highly standardized, reusable, agnostic to parent business processes and specific application environments results in ability to adapt quickly to industry changes.
- Reduced IT Burden – applying SOA results in IT enterprise with reduced waste and redundancy, reduced size and operational cost, and reduced overhead associated with its governance and evolution.

3.2. SOA implementation

Service Oriented Architecture paradigm presented in Section 3.1 is just a set of principles and guidelines. It does not impose any specific standard nor technology. In fact, one of its principles states, that different services can be created by means of different languages and tools. Of course, full freedom would lead to a failure rather than a success. Considerable popularity of web services suggests that they are closest to the golden mean between standardization and freedom. Mature and widely adopted web standards combined with freedom to choose the most appropriate backend technology makes web services the most popular way of creating SOA systems. There are currently two main styles of creating web services: WS-* style (more standardized) and REST style (less standardized). The following section contains short description of HTTP Protocol being common denominator of this two styles and Sections 3.2.2 and 3.2.3 provide more details on each of them.

3.2.1. HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is a networking protocol for distributed, collaborative, hypermedia information systems. The version of HTTP in common use is HTTP/1.1, which is defined in the RFC 2616 ([RFC2616]). HTTP is a request-response protocol. A client is sending a request to a server. The second one is performing some operation and sending its result back to the client. In the most common case web browser acts as a client, locating resource is an operation and resource representation is sent back in the response. HTTP is stateless protocol.

Request

HTTP request consists of the following components:

- Request Line (Method Name, Request URI, Protocol Version),
- Headers,
- Empty Line,
- Optional Message Body.

List of possible headers can be found at [RFC2616], while the most common method names are:

- HEAD,
- PUT,
- GET,
- POST,
- DELETE,
- OPTIONS.

The first line indicates an operation (method name) and a resource, on which the operation should be performed (request URI). Header fields allow to pass additional data about the request or the client itself. The message body is used to transfer the proper content of the request.

Response

Structure of HTTP response is similar to structure of a request:

- Status Line (Protocol Version, Status Code, Reason Phrase),
- Headers,
- Empty Line,
- Optional Message Body.

List of possible headers can be found at [RFC2616], while the most common method names are listed in Table 3.2. The first line indicates the result of an attempt to understand and satisfy the request, and reason phrase gives a short textual description of the status code. Header fields allow to pass additional data about the response. The message body is used to transfer the proper content of the response.

Code	Reason Phrase
100	Continue
200	OK
201	Created
202	Accepted
301	Moved Permanently
307	Temporary Redirect
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
500	Internal Server Error

Table 3.2: HTTP response codes

3.2.2. Web Services in WS-* style

WS-* style of creating web services (also called: Big Web Services, SOAP Web Services) is associated with the set of standards and recommendations, whose names usually, but not always start with a 'WS-' prefix. These specifications are published by several organizations, the most important of which are: World Wide Web Consortium (W3C, see [W3C]) and Organization for the Advancement of Structured Information Standards (OASIS, see [OASIS]). The most important standards connected with WS-* web services are certainly: Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI) and Web Services Description Language (WSDL). Apart from these, there are over fifty specifications, which may complement, overlap, and compete with each other. Meeting all of these standards is impossible, but in practice there is no need to do so.

The first works on the SOAP are dated back to 1998 and in May 2000 SOAP 1.1 becomes acknowledged submission. The WSDL 1.0 has been developed by IBM, Microsoft and Ariba to describe web services for their SOAP toolkit which resulted in WSDL becoming acknowledged submission in March 2001. First versions of the UDDI come from 2001. Finally in January 2002 Web Services Activity (see [WSA]) was formed to lead web services to their full potential.

Transport protocol

The transport protocol used for invoking web services and collecting results is HTTP (see Section 3.2.1). A request content is transferred in the body of HTTP POST request and the result of the operation in the response for this request.

Extensible Markup Language and Simple Object Access Protocol

Extensible Markup Language (XML) is a way of encoding data together with its structure. Apart from XML definition in W3C Recommendation (see [XML]), there are many related standards called XML applications. One of XML applications is called Simple Object Access Protocol (SOAP), definition of which can be found at [SOAP].

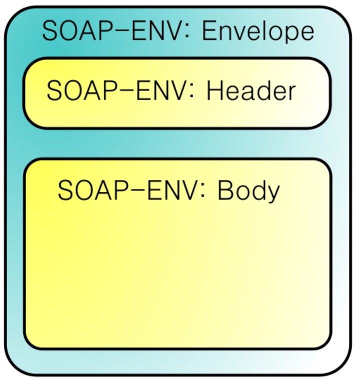


Figure 3.3: SOAP construction. Source: [SOAPWiki]

SOAP message (envelope) consists of two parts (see Figure 3.3): headers (optional) and body (mandatory). The SOAP Header element contains application-specific information (like authentication, payment, etc.) about the SOAP message, while the SOAP Body element contains the actual SOAP message. Examples of the SOAP request and response are shown in Listings 3.1 and 3.2.

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.mimuw.edu.pl/students">
    <m:GetStudent>
      <m:StudentID>10</m:StudentID>
    </m:GetStudent>
  </soap:Body>
</soap:Envelope>
```

Listing 3.1: SOAP Request

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.mimuw.edu.pl/students">
    <m:GetStudentResponse>
      <m:Student>
      <m:FirstName>Karol</m:FirstName>
      <m:LastName>Kański</m:LastName>
    </m:Student>
  </m:GetStudentResponse>
 </soap:Body>
</soap:Envelope>
```

Listing 3.2: SOAP Response

Universal Description, Discovery and Integration and Web Services Description Language

Universal Description, Discovery and Integration (UDDI) is an XML-based registry tightly connected to WS-* standards. It stores documents in a Web Services Description Language

(WSDL) format and can be interrogated by SOAP messages. More about UDDI and WSDL can be found in Section 3.3.

3.2.3. Web Services in REST style

Representational State Transfer (REST), in contrast to WS-* web services, is not a standard, but a software architectural style. The term Representational State Transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation (see [REST]) with regard to how the World Wide Web is constructed. In the REST style the central concept is 'resource'. Such approach is called resource-oriented. Placing 'resource' in the center instead of 'service' can be confusing, when we recall, that we are building service-oriented solution. However, the great part of service's tasks is adding, replacing or removing some elements from some collections (e.g. flight ticket booking can be considered as adding our reservation to the set of reservations). In the following sections we will describe the most important principles of the REST style and a way of implementing web services in pursuance of it (RESTful web services).

REST Principles

RESTful applications should conform to some number of architectural constraints (as described in the [REST]):

1. **Client-Server.** In the client-server style a client initiates a request to a server, which processes it and returns an appropriate response. A client waiting for user actions is called to be 'at rest'. When the client is ready to make a transition to a new state it sends a request. The client is in the transition between application states until it receives a response to the sent request. Received response can contain links, which may be used in the subsequent state transitions.

2. **Stateless.** Communication between a client and a server must be stateless, i.e. it cannot take advantage of any stored context on the server and consequently each request from the client to the server must contain all information necessary to understand the request. It also means that a session state is entirely held in the client. Respecting this constraint ensures visibility, reliability, and scalability.

3. **Cache.** A data within a response to a request should be implicitly or explicitly marked as cacheable or non-cacheable. In the first case the client can reuse that response data in the further equivalent requests. Caching can remove some part of client-server interactions, improving efficiency and scalability.

4. **Uniform Interface.** Generality of component interfaces improves visibility of interactions. Obtaining an uniform interface can be achieved by following the four REST interface constraints:

   - identification of resources;
   - manipulation of resources through representations;
   - self-descriptive messages;
   - hypermedia as the engine of application state.

Resource	Collection URI, such as http://example.com/resources/	Element URI, such as http://example.com/resources/ef7d-xj36p
GET	List the URIs and perhaps other details of the collection's members.	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.
PUT	Replace the entire collection with another collection.	Replace the addressed member of the collection, or if it does not exist, create it.
POST	Create a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Treat the addressed member as a collection in its own right and create a new entry in it.
DELETE	Delete the entire collection.	Delete the addressed member of the collection.

Table 3.3: RESTful web service HTTP methods. Source: [RS]

5. **Layered System.** The system architecture should be divided into hierarchical layers, such that each component sees only the immediate layer with which it is interacting. This constraint promotes components independence, reduces complexity and improves scalability.

6. **Code-On-Demand.** Client functionality can be extended by downloading and executing code in the form of applets or scripts. It improves extensibility, but reduces visibility and thus is optional constraint in REST.

Transport Protocol

Because REST is not a standard, but just a set of guidelines there is not one particular protocol, which must be used in the RESTful applications. However, REST was originally described in the context of the HTTP and is most associated with this protocol. Using HTTP forces adherence to the principles listed in Section 3.2.3. The example of how HTTP methods can be used for implementing web services is shown in Table 3.3.

Resources and representations

The central concept in REST is 'resource'. Resource is any information that can be named. Resource identifiers are used in REST to identify the particular resource involved in an interaction between components (in the web-based systems resources can be identified using URIs). The resource is conceptually separated from its representation. Resource is an abstract entity while its representation is a sequence of bytes, plus metadata to describe these bytes. Representations are used to manipulate resources. In REST there is not one right format of representation (like XML in WS-*, see Section 3.2.2). The most common ones are: HTML, XML, JSON (see [JSON]) or just plain text.

RESTful registries and Web Application Description Language

In a pure REST system a client should start from one fixed entry point and then it can only follow links from representations it received. However, in RESTful web services more natural

	WS-*	REST
Advantages	– easy to consume (XML) – development tools – built-in error handling (faults) – built-in type checking – extensible – standardized	– lightweight – human readable results – easy to build – closer in philosophy and design to the Web
Disadvantages	– more difficult – heavyweight – verbose – harder to develop	– tied to HTTP – lack of standards support for security, policy, reliable messaging, etc. – assumes a point-to-point communication model

Table 3.4: Comparison of WS-* and REST

is finding URI of the resource in the service's description downloaded from the service's registry, rather than crawling to it from starting point. Role of such registries and descriptions is lesser than in WS-* web services, but still significant. The most common is using RESTful registries and Web Application Description Language (WADL). For more information see Section 3.3.

3.2.4. Comparison

As always we cannot tell, what approach is better in an isolation from the problem we want to solve. However, we can list some general pros and cons of both solutions. It can be found in Table 3.4, which shows that number of advantages and drawbacks is similar in both cases. In fact, among large companies popularity of both solutions is similar. The following companies provide some services by means of WS-* web services:

- Amazon,
- eBay,
- Flickr,
- Microsoft (Bing),
- PayPal.

The following companies provide some services by means of REST web services:

- Amazon,
- eBay,
- Facebook (REST API is now abandoned in favor of the so-called Graph API),
- Flickr,
- Google,
- Twitter.

In the Mobility Project we can identify one more advantage of the REST approach. In the WS-* Mobility, there are several pairs of the get/send operations. Depending on the content of the message both get and send operations can set a client organization as the home organization or as the partner organization. In the REST Mobility knowing URI of the operation means knowing student home and partner organization. This makes the system more intuitive.

3.3. Registry and service descriptors

One of the non-functional requirements identified in Section 2.4.2 was to use RESTful registry instead of the UDDI. In order to give the reader some idea on the consequences of such decision, this section describes both solutions. The most popular service description languages for each approach are also described. However, it must be said, that WSDL is not inextricably tied to UDDI and it can be used also with RESTful registries.

3.3.1. Universal Description Discovery and Integration

Universal Description Discovery and Integration (UDDI) was the first initiative for creation of an electronic database of web services. But, since its beginnings in late nineties of the twentieth century, it has been intended to be something more than just a registry of services. It was designed to store not only technical information about web services but also some business information associated with them. Therefore, the UDDI consist of the three main components:

1. **White Pages.** White pages contains information about the business providing the service. These could be service name and description (in multiple languages) but also some information about the party publishing service: address, phone number, etc.
2. **Yellow Pages.** Yellow pages provide a classification of the service or business, based on standard taxonomies. Examples of such standard taxonomies are: Standard Industrial Classification (SIC, see [SIC]), the North American Industry Classification System (NAICS, see [NAICS]), or the United Nations Standard Products and Services Code (UNSPSC, see [UNSPSC]).
3. **Green Pages.** Green pages describe how web service can be used. Among others they contain the following information: address of the service, parameters or references to specifications of interfaces.

The last specification of the UDDI comes from 2005 (version 3.0.2) and since then there have been little work on this document. Today, ten years after first works on the UDDI, it is clearly visible, that it has not become worldwide standard. However, its JAVA implementations (see [jUDDI]) are still released and using such registries within single company is still popular.

3.3.2. Web Services Description Language

Web Services Description Language (WSDL) is an XML-based language for describing web services and how to access them. WSDL 2.0 became a W3C Recommendation ([WSDLRec]) on 26. June 2007. Version 1.1 has not been endorsed by W3C. Figure 3.4 shows the main differences between these two versions. We will describe version 1.1 as it is still more popular than its successor.

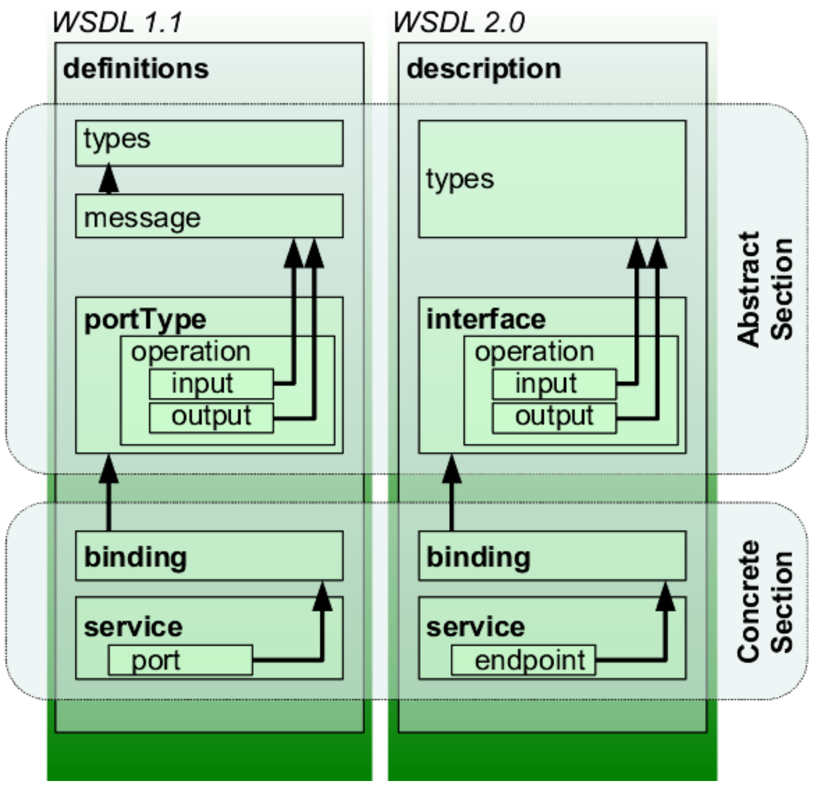


Figure 3.4: WSDL 1.1 vs 2.0. Source: [WSDLWiki]

WSDL document consists of four main parts:

1. Definition of the data types used by web service. These definitions are placed within <types> element. XML Schema syntax (see [XMLSchema]) is used for this purpose, to provide maximum platform neutrality.

2. Definition of the messages used by web service. These definitions are placed within <message> element. Each message consists of several part which defines arguments of the operation.

3. Definition of the operations performed by web service. These definitions are placed within <portType> element. This element describes operations (and messages involved in it) exposed by web service.

4. Definition of the communication protocols used by web service. These definitions are placed within <binding> element. This element defines protocol details and message format for web service.

3.3.3. RESTful registry

Web services description documents are some resources, so it is obvious that we can create RESTful directory service with such descriptions. One of the biggest advantage of registry created in the REST way is possibility of storing any type of description documents. The UDDI is not so flexible, as it was designed to store WSDL documents.

The first considered registry was EdUnify (see [EdUni]), which works well on the United States market but in the end Registry distributed with GEMBus was chosen. GEMBus

(GEANT Multi domain Bus) is the federated multi-domain service-oriented infrastructure being developed in the GN3 project. In the future GEMBus can be used for building a higher education data exchange infrastructure and the Mobility Project is to be one of the 'use cases' for it. GEMBus registry can be searched by means of SPARQL (see [SPARQL]), OpenSearch (see [OS]) and RESTful queries. The service's metadata will be stored using the Web Ontology Language (OWL, see[OWL])/Resource Description Language (RDL, see [RDF]) combo. Stored information depends on the ontology used, but eventual change of the ontology does not affect registry architecture.

3.3.4. Web Applications Description Language

Web Applications Description Language (WADL), which is a W3C Member Submission (see [WADL]), is an XML-based language for describing applications created in the REST style. That task can be also performed using WSDL 2.0, but WSDL 2.0 has not gained much popularity and is not used even for describing WS-* applications. There is a lot of controversy about the need to describe REST applications in a machine-readable way. Such description can be used for automated client code generation but such client will surely be worse than modern web browsers, which can be used as REST clients. Moreover, XML description ignores non-XML content of the Web (HTML, CSS, JavaScript, podcasts, videos, JSON, etc.). There is also a danger of transforming WADL, which was intended to be simple, into something like WSDL, which is complicated. The whole discussion about the merits of using WADL can be found in [DWNW]. However, because the REST Mobility provides a WADL document we will briefly describe its construction.

WADL document consists of the following parts:

1. **Documentation.** This part contains documentation of the application.
2. **Grammars.** This part is a container for definitions of the format of exchanged data (it is similar to <types> element described in Section 3.3.2).
3. **Resources.** This part presents an application URL scheme and resources associated with given URLs.
4. **Methods.** This part presents binding between HTTP methods, data types from 'Grammars' part and resources from 'Resources' part.

3.4. URL design

This section presents URL scheme used for RESTful implementation of the Mobility Project. The idea of implementing the Mobility web services RESTfully comes from Roland Hedberg [Hed10]. In the mentioned paper Hedberg also outlined how such implementation can be done in the REST style. This section introduces some changes to the schema presented in [Hed10] and provides some examples for that new schema focusing on the fact that there are two sides of the process, each of which can be home organization and partner organization.

3.4.1. Operations

Assumptions

For the purpose of this presentation we assume that data are exchanged between the University of Warsaw, Poland (UW) and the Umea University, Sweden (UmU). Other assumptions stated in [Hed10] are also valid in this work. However, in the discussion to follow we always act on behalf of the University of Warsaw. In the examples we show URLs for the situation when UW is home organization (sending its students) and also for the situation when UW is partner organization (hosting mobility students arriving for short-term studies). As introduced in [Hed10] the general URL scheme for all operations is:

- *organization/operation*,

where:

- *organization* part depends on the organization (it is an address of the root of the Mobility RESTful web services);
- *operation* part depends on the invoked operation.

We assume that we have the following organization parts:

- http://rs3g.uw.edu.pl/ – for the University of Warsaw;
- http://rs3g.umu.se/ – for the Umea University.

We also assume that we have the following organization ids:

- uw.edu.pl – for the University of Warsaw;
- umu.se – for the Umea University.

Further sections provide description of operations defined in [Nag09] in the language of RESTful web services. In all cases, when organization operates on the data located in its resources tree it can choose to do it through RESTful interface or by other means. This work utilizes data formats defined in [MobWSDL].

Overview

Next sections show example URLs for the Mobility methods. The following general rules apply:

- data that are not related to students (general HEI data like courses or Erasmus coordinators) are maintained on that data owner's side (home institution);
- list of nominated students is maintained on the partner organization side (hosting institution);
- all data related to nominated students are maintained together with the students' data (hosting institution).

Organization data

When exchanging information about organizations it is not important, which institution is the home one and which is the partner one. The SOAP method *getOrganizationData* is used for getting information about another institution from that institution and the SOAP method *sendOrganizationData* is used for sending our data to another institution and both are invoked at the partner organization side (server). In the proposed REST approach situation is different – information for UW from Umea is stored at the Umea's side so when UW wants to obtain Umea's data it should look for it at Umea's server. On the other hand when UW wants to send its data to Umea it should put it on its own server. In practice, it is not important that Umea will be receiver of UW's organization data – the same date can be sent to e.g. the University of Parma. Therefore, an URL for uploading and downloading data is different from this presented in [Hed10] (receiver id is omitted). We can specify two scenarios:

- UW sends its data to Umea – UW should PUT the data to:
  http://rs3g.uw.edu.pl/organization/
- UW gets Umea's data – UW should GET the data (assuming that Umea has previously PUT data there) from:
  http://rs3g.um.se/organization/

In this and the following examples PUT, GET, DELETE, POST should be regarded in the meaning of the HTTP protocol.

Agreement data

In [Hed10] there is an unspoken assumption that there can be only one agreement between two sides and that purpose of exchange of messages like *getAgreementData/sendAgreementData* is to work out an agreement that both sides will abide. In fact this calls are responsible for exchanging contents of a bilateral agreement signed by two HEIs ([Nag09]) and this means that URL's method part for uploading and retrieving data should contain agreement id: /agreement/<organization id>/<agreement id>. There are the following scenarios:

- UW sends its agreement data to Umea – UW should PUT the data to:
  http://rs3g.uw.edu.pl/agreement/um.se/982-E-II08
- UW gets Umea's agreement data from Umea – UW should GET the data from:
  http://rs3g.um.se/agreement/uw.edu.pl/982-E-II08

where 982-E-II08 is the agreement id.

Course data

The organization maintains information about its courses at its site, so this implies the following scenarios:

- UW gets course related data from Umea – UW should GET the data from:
  http://rs3g.umu.se/course/5EL143
- UW provides its course related data – UW should PUT the data to:
  http://rs3g.uw.edu.pl/course/5EL143

where 5EL143 is the course identifier.

Validating National Personal Id

This method has quite utilitarian, non-RESTful character. However, it also can be implemented in the RESTful style. The scenario is as follows:

- UW wants to validate National Personal Id of Umea's student – UW makes GET from:
  http://rs3g.um.se/validateNIN/201005271234,

where 201005271234 is the id to be validated. Response with HTTP code 200 OK means that this id is valid and response with HTTP code 400 means that this id is not valid.

Nominated students

Student departures are held as part of an agreement. Moreover, under one agreement student can depart several times – in different years or even in one year but in a different academic period or under different cooperation conditions. These facts are not reflected in the URL scheme presented in [Hed10] (see also Section 3.4.1). URLs for uploading and retrieving data pertaining to nominated students (Sections 3.4.1 - 3.4.1) should contain agreement id, year of departure, cooperation conditions id and academic period id:
/nominatedStudents/<organization id>/agreement/<agreement id>/year/<year>/
coopCond/<cooperation conditions id>/period/<academic period id>/student/...
In order to make URLs presented in this document shorter in this and the following sections we will use the following abbreviations:

- uwPref=http://rs3g.uw.edu.pl/nominatedStudents/umu.se/agreement/
1207-E-XI08:982-E-II08;
- umPref=http://rs3g.umu.se/nominatedStudents/uw.edu.pl/agreement/
982-E-II08:1207-E-XI08;

where 1207-E-XI08:982-E-II08 and 982-E-II08:1207-E-XI08 are agreement ids. For exchanging lists of nominated students it is important to state two things:

- which side is home and which is partner;
- what is the scope of the list (all students for a given year, all students for a given year and cooperation conditions or all nominated students for a given year, cooperation conditions and academic period).

At first we assume that UW is home organization (sending its students). This means that it knows a list of nominated students and wants to send it to Umea. Afterwards UW may want to check if Umea has introduced any changes to that list, so it retrieves the possibly updated list of its students from the Umea server. There are two main scenarios:

- UW sends a list of its students nominated for short term studies at Umea – depending on what is the scope of the list UW should PUT the data to:
  - *umPref*/year/2008 – all nominated students for the given year;
  - *umPref*/year/2008/coopCond/1:11.3-eu:1 – all nominated students for the given year and cooperation conditions;
  - *umPref*/year/2008/coopCond/1:11.3-eu:1/period/S1:1 – all nominated students for the given year, cooperation conditions and academic period;
- UW retrieves actual state of the list of its nominated students – depending on what is the scope of the list UW should GET the data from:
  - *umPref*/year/2008 – all nominated students for the given year;
  - *umPref*/year/2008/coopCond/1:11.3-eu:1 – all nominated students for the given year and cooperation conditions;
  - *umPref*/year/2008/coopCond/1:11.3-eu:1/period/S1:1 – all nominated students for the given year, cooperation conditions and academic period.

List of nominated students is of course a collection of students, so being RESTful means that we can also perform operations such as:

- UW adds a student to its list of nominated students – UW should PUT the data to:
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234
- UW gets a student from its list of nominated students – UW should GET the data from:
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234
- UW deletes a student from its list of nominated students – UW should DELETE the data from:
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234

On the other hand UW can also be the hosting organization. In that case there are the following scenarios:

- UW gets a list of students nominated by Umea for short term studies in Warsaw – UW should GET the data from:
  uwPref/<list scope part>
- UW gets a student from Umea's list of nominated students – UW should GET the data from:
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234
- UW deletes a student from Umea's list of nominated students – UW should DELETE the data from:
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234

In all cases 2008 is the year of departure, 1:11.3-eu:1 is the cooperation conditions id, S1:1 is the academic period id and 201005271234 is the student id.

Arrival date

The hosting organization is aware of a student arrival date and it places it on its site together with other student data. Assuming that UW is the home organization we have:

- UW gets arrival date of its student to Umea – UW should GET the data (assuming that Umea has previously PUT data there) from :
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/arrival

When UW is the hosting (partner) organization the scenario is:

- UW sends to Umea date of arrival to UW of Umea's student – UW should PUT the data to:
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/arrival

In both cases 2008 is the year of departure, 1:11.3-eu:1 is the cooperation conditions id, S1:1 is the academic period id and 201005271234 is the student id.

Departure date

The hosting organization is aware of a student departure date and it places it on its site together with other student data. Assuming that UW is the home organization we have:

- UW gets date of departure of its student from Umea – UW should GET the data from (assuming that Umea has previously PUT data there):
  umPref/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/depart

When UW is the hosting (partner) organization the scenario is:

- UW sends date of departure of Umea's student from UW – UW should PUT the data to:
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/depart

In both cases 2008 is the year of departure, 1:11.3-eu:1 is the cooperation conditions id, S1:1 is the academic period id and 201005271234 is the student id.

Learning agreement

The home organization is aware of courses chosen by its students from partner organization's course catalog (the choice has to be approved by the home Erasmus coordinator), so it places student's learning agreement on the partner organization site together with other student data. Assuming that UW is the home organization we have:

- UW sends Learning Agreement of its student to Umea – UW should PUT the data to:
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/
  learningAgreement

When UW is the hosting (partner) organization the scenario is:

- UW gets Umea's student Learning Agreement – UW should GET the data (assuming that Umea has previously PUT data there) from :
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/
  learningAgreement

In both cases 2008 is the year of departure, 1:11.3-eu:1 is the cooperation conditions id, S1:1 is the academic period id and 201005271234 is the student id.

Transcript of records

The hosting organization is aware of grades of students who come to this organization for short term studies (we may say that the grades are generated there) and it places them on its site together with other student data. Assuming that UW is the home organization we have:

- UW gets its student's transcript of records – UW should GET the data (assuming that Umea has previously PUT data there) from:
  umPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/
  transcriptOfRecords

When UW is the hosting (partner) organization the scenario is:

- UW sends Umea student's transcript of records – UW should PUT the data to:
  uwPref/year/2008/coopCond/1:11.3-eu:1/period/S1:1/student/201005271234/
  transcriptOfRecords

In both cases 2008 is the year of departure, 1:11.3-eu:1 is the cooperation conditions id, S1:1 is the academic period id and 201005271234 is the student id.

3.4.2. Identifiers

URLs introduced in Section 3.4.1 and in [Hed10] make use of some IDs like student id, organization id etc. These IDs are introduced in Section 2.3, [Nag09] and [MobWSDL]. Unfortunately mapping between IDs from these documents and those from Section 3.4.1 is not clear. The situation is even more complicated by the fact that we should take into consideration context of home and partner organization. This section intends to clarify how IDs used in Section 3.4.1 should be constructed.

Organization ID

Organization ID has the same meaning as in the [MobWSDL] (see *organizationIdT* type in that document).

Course ID

Course ID has the same meaning as in the [MobWSDL] (see *courseCodeT* type in that document).

Agreement ID

Agreement ID as introduced in [MobWSDL] consists of four components:

- home organization ID,
- agreement ID at home organization,
- partner organization ID,
- agreement ID at partner organization.

In URLs in Section 3.4.1 home organization ID and partner organization ID are already present (explicitly or as an organization part of the URL) so an agreement ID is a combination of an agreement ID at home organization (ahID) and an agreement ID at partner organization (apID). The home organization is always the organization at which side operation is performed and all data with adjective 'home' are referring to it. The agreementID has the following form:

- <agreementID> ::= <ahID>:<apID>

There is one more important note pertaining to agreement ID: because it is a part of an URL it must not contain '/ ' sign. If this sign exist in ahID or apID it should be replaced by '-'.

Example

Consider the following part of an XML document with request for agreement data:

```
<agreementId>
  <homeId>
    <organizationId>uw.edu.pl</organizationId>
    <value>982/E/II08</value>
  </homeId>
  <partnerId>
    <organizationId>umu.se</organizationId>
    <value>1207/E/XI08</value>
  </partnerId>
</agreementId>
```

This should be mapped to the following URL:

- http://rs3g.uw.edu.pl/agreement/umu.se/982-E-II08:1207-E-XI08

Cooperation conditions ID

Cooperation conditions ID has the meaning of *cooperationConditionsRedundantContextG* from [MobWSDL] (see also[Nag09]) which consists of three components: cooperation conditions id (cci), subject area code group (sacg) and study level (sl). Cooperation conditions ID present in URLs in this thesis has the following structure:

- <cooperation conditions id> ::= <cci>:<sacg>:<sl>

Subject area code group consists of subject area codes (sac) separated by commas:

- <sacg> ::= <sac>(,<sac>)*

Subject area code consists of two components: classification (c) and value (v) so sac has the following form:

- <sac> ::= <v>-<c>

Example
Consider the following part of an XML document with *cooperationConditionsRedundantContextG*:

```
<cooperationConditionsId>1</cooperationConditionsId>
<subjectAreaCode>
  <value classification="eu">11.3</value>
  <value classification="ISCED97">32</value>
</subjectAreaCode>
<studyLevel>1</studyLevel>
```

This should be mapped to:

- 1:11.3-eu,32-ISCED97:1

Academic period ID

Academic period ID has the meaning of *academicPeriodSinceT* [MobWSDL] (see also[Nag09]) which consist of three components: academic period (ap), duration (d) and year. Year is already present in the URL so academic period ID is the combination of academic period and duration and has the following form:

- <academic period id> ::= <ap>:<d>

Example
Consider the following part of an XML document with *academicPeriodSinceT*:

```
<stayPeriod>
  <academicYear>2008</academicYear>
  <academicPeriod>S1</academicPeriod>
  <duration>1</duration>
</stayPeriod>
```

This should be mapped to:

- S1:1

Student ID

Course ID has the same meaning as *nationalPersonalIdT* type in the [MobWSDL].

3.4.3. Final remarks

URL scheme described in this thesis is of course only an example. It is assumed that users will be provided with description of URL scheme in the WADL format and therefore organizations may define their own scheme. However organization part of the URL cannot be changed or must be changed in both organizations (otherwise some operations will fail as no part will be responsible for handling them). The users should also be instructed on which servers they should perform the particular operation. It can be achieved by fixing that given operation will be always held on servers of the given side.

3.5. SSL/TLS

Sending sensitive personal data through the Web should not be done without any protection. The two most obvious ways of stealing such data are: intercepting transmission and pretending that we are someone else, who is authorized to receive the data. Therefore security of the transmission and user authentication are listed as non-functional requirements in Section 2.4.2. The first idea of how to assure security was to use GEMBus (see [GEMBus]) and its Authentication and Authorization Service, which provides authentication and authorization mechanisms for other services attached to the bus. The more detailed description of this mechanism can be found at [PERFSONAR]. Other solutions were proposed by R. Hedberg in [Hed10] and by V. Giralt and M. Sova in [GS10]. The second mechanism, which is based on SSL/TLS, was finally chosen.

Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS), are cryptographic protocols that provide integrity and confidentiality on the transmission channel. Moreover, SSL/TLS can provide server and sometimes client authentication. SSL/TLS uses several different cryptographic processes: public key cryptography to provide authentication, and digital signatures to provide privacy and data integrity. In order to fully understand how SSL/TLS can be used in the Mobility Project reader should be familiar with the most important terms related to cryptography, cryptographic processes and SSL/TLS mechanism. Below is the short summary of these issues.

Most of the algorithms used to encrypt and decrypt data depend on the use of an agreed-upon cryptographic key or pair of keys and can be divided into two main categories:

1. **Secret key cryptography.** Both communicating parties must agree on the cryptographic algorithm, which will be used for encryption and decryption, and on the cryptographic key (the same for encryption and decryption). The secret key cryptographic

algorithms encrypt data relatively quickly and provide excellent security. However, there is one major logistic problem: how to exchange keys.

2. **Public key cryptography.** In the public key cryptography there is a pair of keys, each one being cryptographic inverse of the other – while one encrypts the data the other decrypts it. The public key can be safely sent through the network. The data encrypted with the public key can only be decrypted with the associated private key (transmission security). From the other side: if we can decrypt the message with the public key, we can be sure that it was encrypted with the associated private key (we know that the other part is really what we think it is). In the second scenario everybody can read the message, so it does not provide secure communication, but it provides basis for the digital signature (part of a public key certificate, see below) used for authenticating a client or a server. The public key cryptography is very slow.

Public Key Certificates (PKCs) are a way of proving our identity in the Web. PKC is issued by a trusted organization (certificate authority – CA) and provides identification for the bearer. PKC contains public key of the other side, so after checking certificate authenticity we can be sure that the other side is what it claims to be. In the case of the Mobility Project certificates will be issued by TERENA (see [TERENA]).

Cryptographic Hash Functions and Message Authentication Code are message's checksums designed for checking if message was tampered during transmission. Digital signature is cryptographic hash of the message encrypted with the senders private key.

SSL/TLS process consists of three main phases:

1. **Negotiating the cipher suite.** A client tells a server the cipher suite (set of cryptographic algorithms and key sizes, e.g. the public key exchange algorithms, key agreement algorithms and cryptographic hash functions) it supports. The server chooses the best mutually accepted one.

2. **Performing authentication.** It is an optional phase. The server presents its certificate to the client and eventually asks the client to do the same (the client encrypts previous messages to prove that it has access to the private key associated with sent certificate). If it is not achieved by a certificate exchange then the server sends to the client its public key. The client generates a secret key, encrypts it with the server public key and sends to the server. Since then both the client and the server has the same secret key and can move to the next phase.

3. **Sending the encrypted data.**

It is obvious how to use SSL/TLS for providing transmission security or a server and a client authentication in the Mobility Project. Having the client certificate we know who it really is and can decide if it is allowed to do a requested operation.

3.6. Architecture

Overview of the system architecture is presented in Figure 3.5. Four of the components presented in Figure 3.5 are provided with the REST Mobility: Client GUI, Client transport, Server and Persistent Storage (simple implementation). Client and server Keystores/Truststores are simple databases of keys and certificates (see Sections 3.5 and A.4). Registry means RESTful Registry provided with GEMBus. There are also several communication channels:

0. These channels are used for interaction between user and application (GUI, terminal etc.).

1. These channels are used for communication with some software providing HTTP connection (a web browser, client transport library). The other side can be: user (1c) or other applications (1a, 1b).

2. This channel is used for retrieving WADL documents from the registry.

3. These channels are used for making HTTP requests.

4. This channel is used by the server for retrieving its certificate from a keystore and for checking the client certificate in a truststore.

5. This channel is used by the client for retrieving its certificate from a keystore and for checking the server certificate in a truststore.

6. This channel is used by the server for checking if the client is allowed to do a requested operation – the client has a certificate for domain=clientorganization.org and wants to perform some operation on behalf of a organization with id=organizationId (this id is in some way present in a request). The server checks in the registry if organization with id=organizationId is responsible for domain=clientorganization.org. If yes the request is allowed and otherwise it is rejected.

7. This channel is used by the server for storing and retrieving data from the persistent storage.

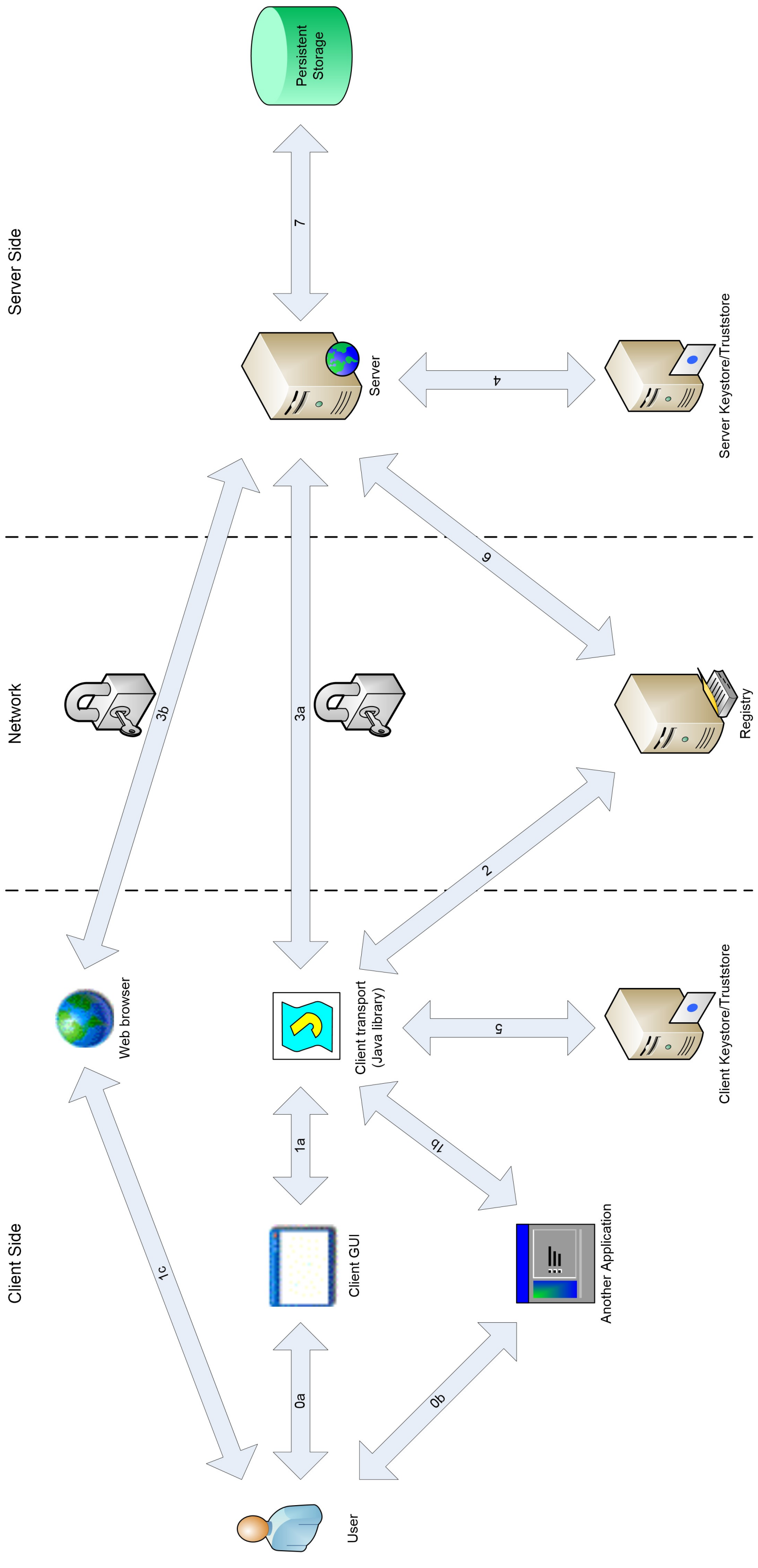


Figure 3.5: Architecture of the system

Chapter 4

Implementation

4.1. Technologies

This chapter describes chosen technologies, important implementation decisions and other relevant implementation details.

The software platform chosen for the project is Java. It is a modern, object-oriented language based on open, public standards. Moreover, it has a rich collection of core functions and many libraries. Last but not least Java applications can execute with little or no change in different environments. The other considered choice was Python, but the fact that the Mobility has been written in Java prevailed.

4.1.1. Servlet container

The server application (see Section 4.3) can be executed in a servlet container. The most popular free servlet containers are: Apache Tomcat (see [Tomcat]), GlassFish (see [GlassF]) and JBoss (see [jBoss]). The REST Mobility was tested on the first two servers. There were some minor problems connected with logging on the GlassFish. Generally speaking, the REST Mobility does not demand a lot from server container (SSL/TLS support would be appreciated, but it is rather standard), so any reasonable choice should be good.

4.1.2. Spring Framework

The Spring Framework (see [Spring]) is a general purpose open source application framework for the Java platform. The most important feature provided by Spring Framework is Inversion of Control container. The container's main task is managing object lifecycles: creating objects, calling initialization methods, and configuring objects by wiring them together. Because object creation is managed by the container and object dependencies are declared in an XML file we can exchange implementation of some interfaces without need to change code of other libraries (we only change the XML file).

4.1.3. Restlet

Restlet (see [Restlet]) is a lightweight, comprehensive, open source REST framework for the Java platform. One can ask if there is a need for a framework, when creating RESTful web services which have to be simple and lightweight. Such question would be senseless and not justified: one thing is being easy and lightweight in terms of network transfer, type checking or number of needed libraries and the other is care of good code structure and obeying REST rules. The last two features are imposed by Restlet, because it directly models concepts (Resource, Representation, Connector, Component, etc.) from Fielding's doctoral dissertation. In addition, it supports major Internet transport, data format, and service description standards like HTTP and HTTPS, SMTP, XML, JSON, Atom, and WADL. A large number of available connectors makes possible deployment of software in various environments.

4.2. Tools

4.2.1. Eclipse

I used Eclipse IDE (see [ECL]) as the main Java and XML editor. Like all good IDEs, Eclipse offers: syntax highlighting, syntax completion, a-jump-to-declaration feature and more. Moreover, its capabilities can be extended by means of plug-ins – e.g. you can add Maven (see Section 4.2.2) support.

4.2.2. Apache Maven

Maven (see [MVN]) is a software tool for project management and build automation. It is based on the concept of a project object model (POM), which is an XML document describing the software project being built, its dependencies on other external modules and components, and the build order. Maven comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. These tasks are performed by plug-ins, which are automatically downloaded from repositories. Moreover, required Java libraries are also downloaded automatically.

4.2.3. Subversion (SVN)

I used Subversion as a software versioning and a revision control system.

4.2.4. UML

The mobility data model in Section 2.3 is described using Unified Modeling Language (UML). UML is a standardized general-purpose modeling language, which includes a set of graphic notation techniques to describe various aspects of a software system.

4.2.5. Visual Paradigm

UML diagrams were prepared using Visual Paradigm.

4.3. Components

The project has the following hierarchy:

```
rest-mobility-project
|--rest-mobility-basics
|--rest-mobility-client
|  |--rest-mobility-client-gui
|  |--rest-mobility-client-transport
|--rest-mobility-persistence
```

```
|--rest-mobility-registry
|--rest-mobility-server
   |--rest-mobility-server-standalone
   |--rest-mobility-server-transport
   |--rest-mobility-server-war
```

Rest-mobility-basics provides some general functionality used by other components. The main responsibility of this component is XML-processing.

Rest-mobility-server is responsible for retrieving WADL documents from a given registry and resource's URIs from WADL.

Rest-mobility-persistence is responsible for storing resource's representations. It defines an interface, that one should implement when linking the REST Mobility with his or her system. Simple file-based implementation is also provided.

Rest-mobility-client-transport is responsible for making HTTP requests and extracting representations from responses.

Rest-mobility-client-gui provides a graphical user interface for the rest-mobility-client-transport library.

Rest-mobility-server-transport is responsible for receiving HTTP requests, routing them to appropriate objects from rest-mobility-persistence library and sending responses.

Rest-mobility-server-war contains files needed for packaging the rest-mobility-server-transport library as a war archive.

Rest-mobility-server-standalone contains files needed for packaging the rest-mobility-server-transport library as a standalone application.

These libraries form two applications:

- client application – GUI application for making REST calls;
- server application – implementation of the Mobility RESTful web services, available in two forms:
  - war archive, that can be deployed into servlet container;
  - standalone application, that can be run from command line.

A documentation for both applications can be found in Appendix A.

Chapter 5

Summary

5.1. Goals achieved

A discussion of achieved goals will start from the functional requirements, which were identified in [Nag09] in Section 3.3.1.

The first one of this requirements was to provide necessary infrastructure for electronic data exchange over network between HEIs participating in student mobility programmes. The solution should be easily integrated with SIMS and should allow for exchanging the data multiple times. Some of the work in this area was done by R. Nagrodzki, who identified mobility use cases and established vocabulary for the modeled domain. My work organizes and extends the vocabulary by providing a high level mobility data model. Attached network transport software implements mobility use cases and allows for multiple data exchange. Integration with SIMS has not been tested but it seems not to be a problem (see also Section 5.2.1).

The system is testable. Operations can be invoked through GUI client. With help of a web-browser the current state of the resources stored on the server can be viewed. Sample data files prepared for the Mobility can be used by the client and can be transformed into hierarchy of files used by the server.

Now we will investigate realization of non-functional requirements, which are defined in Section 2.4.2.

The main non-functional requirement was to obey REST style architecture guidelines. Conditions listed in Section 3.2.3 are satisfied (see also Section 3.4), which leads to conclusion that non-functional requirement no. 1 is met.

The software uses data types defined in [MobWSDL]. In fact, the only elements directly depending on the data format are GUI client and persistent storage implementation. Serious changes in the data format would cause a need to re-design URL hierarchy (see Section 3.4) on the server.

Functional requirement of the user authentication and non-functional requirements no. 5 and no. 6 are ensured by SSL/TLS (see Section 3.5).

Systems built in peer-to-peer (P2P) architecture, in which there is no one central machine coordinating the whole network traffic, usually scale well. Similarly should be in case of the REST Mobility, so non-functional requirement no. 4 is met.

Referring to non-functional requirement no. 9 it must be said, that there are not many tools and technologies used in this project. Technologies like Java, Spring or used servlet containers (Apache Tomcat, GlassFish) are widely used and recognized. Used tools: Eclipse and Maven are known to every Java developer. The only one library, that might be less known is Restlet. However, it is quite mature technology (the REST Mobility uses version 2.0, but 2.1 is already available), which is gaining more and more users.

Non-functional requirement no. 10 is not entirely met. GEMBus registry has been launched but it is not stable yet. Integration will be continued. Switching to the new registry will not cause any major changes in the REST Mobility architecture and code.

In my opinion, non-functional requirement no. 8 is also met. Installation requires some experience in Java technology – a person without such experience probably will not be able to install the software. However, tasks during the installation process are not more difficult than compilation using Maven, packaging into jars, creating directories or filling configuration files.

5.2. Future work

5.2.1. Persistence Layer and SIMS Integration

A main emphasis in this project was put on defining and implementing HTTP interface of RESTful web services. A decision of how data are stored on the server was left to the users – intention was that everyone should implement interface defined in the rest-mobility-persistence library on his own. However, to make the whole solution testable and to show that the solution is really working, an implementation of the persistence layer is provided in the REST Mobility. This implementation is not elaborate. It checks if representations are valid and stores them directly in the filesystem. It does not check if received data are sensible nor does it address issues of concurrent access. Future work in this area may include adding these features or creating new implementation, that will store data in some kind of database. Choosing one of this ways will depend on how the system will be integrated with SIMS.

One possible solution is that the REST Mobility server will be separated from SIMS and SIMS will communicate with it through defined HTTP interface. This is quite elegant and RESTful as everyone reaches data through the same interface. Moreover, it is the interface, through which SIMS will reach data stored on another organization's server, so mechanism of communicating through this interface will anyway have to be supported by SIMS. There are also drawbacks: the same data are stored in two places and are not immediately available. In case of choosing this way it will be enough to improve existing persistence solution.

The other way of integration is implementing persistence layer, so that it stores data in the SIMS database. In such case there is no need for synchronization but the cost of integration will probably be larger.

5.2.2. Extending HTTP interface

HTTP interface presented in Section 3.4 has been designed to offer a functionality similar to that in the Mobility. However, due to the REST style constraints, some actions available in the Mobility cannot be performed in the REST Mobility (i.e. you cannot download a list of nominated students for a given academic year and a selected set of cooperation conditions – you have to choose all or only one cooperation conditions). On the other hand, some operations have been added in the REST Mobility – i.e. you can delete a single student from a list of nominated students.

Designing and implementing new operations may be one of the tasks for future developers of the REST Mobility. This work should be done carefully and with prudence to not create unnecessary operations, which will only mess the code. For sure there is no need for operation, that will change only organization's address, but change in coordinators telephone number can happen from time to time.

5.2.3. Registry and GEMBus Integration

One of the goals, which has not been fully achieved is integration with RESTful registry from GEMBus project. Therefore, realization of this aim should be part of the future work on the REST Mobility. As stated in Section 5.1, retrieving WADL documents from this registry should not be a problem. The registry is also to be used during authorization process (from TLS certificate we get client domain, then we check in the registry which organization is associated with that domain and what it is allowed to do). For sure many other ways of utilization of the registry can be found. Possibly general information about the organization (like address, phone number etc.) could be stored there.

The registry is part of the GEMBus infrastructure, which has the ambition to become an infrastructure for exchanging data between HEIs. Moreover, service buses are a natural part of a service-oriented infrastructure (see Section 3.1.2). Consequently, there is a space for using it in the Mobility Project. One of the ideas can be allowing for co-existence of the Mobility and the REST Mobility and using bus for mediation between these two solutions.

Bibliography

[AMDR09] Fabio Arcella, Janina Mincer-Daszkiewicz, Simone Ravaioli, *Web-services for Exchange of Data on Cooperation and Mobility between Higher Education Institutions.* EUNIS 2009, The 15th International Conference of European University Information Systems, Santiago de Compostela, Spain, 23-26 June 2009.

[CINECA] CINECA – Interuniversity Consortium, `http://www.cineca.it/`

[DWNW] Joe Gregorio, *Do we need WADL?* `http://bitworking.org/news/193/Do-we-need-WADL`

[ECL] The Eclipse Foundation open source community website. Homepage of the Eclipse IDE. `http://www.eclipse.org`

[EdUni] A community registry for Web Services and Service-Oriented Architecture governance. `https://demo.edunify.pesc.org/`

[ESBWiki] Article about enterprise service bus on Wikipedia. `http://en.wikipedia.org/wiki/Enterprise_service_bus`

[EUNIS] European University Information Systems. `http://www.eunis.org/`.

[GEMBus] *Composable Network Services Framework and General Architecture: GEMBus.* Private Correspondence.

[GlassF] GlassFish community. Homepage of the GlassFish Server. `http://glassfish.java.net/`

[GS10] Victoriano Giralt, Milan Sova, *Trust model for EMAM. How can it help our project.* RS3G Mobility Coding Camp, Barcelona, March 24th 2011.

[Hed10] Roland Hedberg, *Supporting Student Mobility using RESTful Web Services*, Umea University, 2010, `http://wiki.teria.no/download/attachments/19005468/mobility_rest-0.2.pdf`.

[Hea10] Paul Heald, *Student Identity Defined: a Comparison of the Data Elements of Four Higher-Education Standards.* Technical Brief, Sigma Systems, 2010.

[jBoss] JBoss. Homepage of the JBoss Application Server. `http://www.jboss.com/products/platforms/application/`

[jUDDI] Homepage of the jUDDI project. `http://juddi.apache.org/`

[JMD10] Janina Mincer-Daszkiewicz, *The Mobility Project. Review of the current status.* RS3G workshop, Warsaw, June 23rd 2010.

[JSON] JavaScript Object Notation homepage. `http://www.json.org/`

[KBS04] D. Krafzig, K. Banke, D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices.* Prentice Hall PTR, November 2009.

[Kra06] Marcin Krawczyński, *Uniwersytecki System Obsługi Studiów. Biuro Współpracy z Zagranicą: umowy i przyjazdy.* Master's thesis, Institute of Informatics, University of Warsaw, 2006.

[Lom08] R. Z. Łomowski, *Uniwersytecki System Obsługi Studiów. Moduł wyjazdy.* Master's thesis, Institute of Informatics, University of Warsaw, 2008.

[MobWSDL] *Mobility WSDL*, `http://usos.edu.pl/Mobility/`

[MUCI] Międzyuniwersyteckie Centrum Informatyzacji, `http://www.muci.edu.pl/`

[MVN] Homepage of the Apache Maven Project. `http://maven.apache.org/`

[Nag09] Rafał Nagrodzki, *The Mobility Project.* Master's thesis, Institute of Informatics, University of Warsaw, 2009.

[NAICS] North American Industry Classification System, `http://www.census.gov/eos/www/naics/`

[OASIS] Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org/`

[OS] Homepage of the OpenSearch project. `http://www.opensearch.org/Home`

[OWL] W3C Recommendation, *OWL Web Ontology Language*, 2004, `http://www.w3.org/TR/owl-features/`

[PERFSONAR] The Authentication and Authorization Service in perfSONAR. `https://wiki.man.poznan.pl/perfsonar-mdm/index.php/Authentication_Service_resources`

[RDF] W3C Recommendation, *RDF Vocabulary Description Language*, 2004, `http://www.w3.org/TR/rdf-schema/`

[REST] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine, 2000.

[Restlet] Homepage Restlet Framework. `http://www.restlet.org/`

[RFC2616] RFC 2616, *Hypertext Transfer Protocol*, 1999, `http://tools.ietf.org/html/rfc2616`

[RS] Leonard Richardson, Sam Ruby, *RESTful Web Services*, O'Reilly, May 2007.

[RS3G] Rome Student Systems and Standards Group. `http://www.rs3g.org/`.

[RS3GDiscussion] Discussion on personal data in Mobility WSDL. `http://wiki.teria.no/display/RS3G/Person`

[SIC] Standard Industrial Classification, `http://www.sec.gov/info/edgar/siccodes.htm`

[SOAWiki] Article about service-oriented architecture on Wikipedia. http://en.wikipedia.org/wiki/Service-oriented_architecture

[SOAP] W3C Recommendation, *SOAP Version 1.2*, 2007, http://www.w3.org/TR/soap/

[SOAPWiki] Article about SOAP on Wikipedia. http://en.wikipedia.org/wiki/SOAP

[SOAPrinciples] Thomas Erls, *SOA Principles. An Introduction to the Service-Orientation Paradigm.* http://soaprinciples.com/

[SPARQL] W3C Recommendation, *SPARQL Query Language for RDF*, 2008, http://www.w3.org/TR/rdf-sparql-query/

[Spring] Homepage of the Spring Framework. http://www.springsource.org/

[SSL] *SSL 3.0 Specification*, 1996, http://www.freesoft.org/CIE/Topics/ssl-draft/3-SPEC.HTM

[TERENA] The Trans-European Research and Education Networking Association. http://www.terena.org/

[TLS] RFC5246, *The Transport Layer Security (TLS) Protocol Version 1.2*, 2008,http://tools.ietf.org/html/rfc5246

[Tomcat] Homepage of the Apache Tomcat project. http://tomcat.apache.org/

[UDDIIntr] *Introduction to UDDI: Important Features and Functional Concepts*, http://uddi.org/pubs/uddi-tech-wp.pdf, Organization for the Advancement of Structured Information Standards, October 2004.

[UNSPSC] United Nations Standard Products and Services Code, http://www.unspsc.org/

[Van10] Geir Vangen, *Mobility Data Model*, University of Oslo, 2010, http://wiki.teria.no/display/~geir.vangen@usit.uio.no

[W3C] World Wide Web Consortium. http://www.w3.org/

[WADL] W3C Member Submission, *Web Application Description Language*, 2009, http://www.w3.org/Submission/wadl/

[WISOA] Thomas Erl, *What Is SOA? An Introduction to Service-Oriented Computing.* http://www.whatissoa.com/

[WikiTeria] Portal for projects related to RS3G. http://wiki.teria.no/

[WSA] W3C Web Services Activity. http://www.w3.org/2002/ws/

[WSDLRec] W3C Recommendation, *Web Services Description Language (WSDL) Version 2.0*, 2007, http://www.w3.org/TR/#tr_WSDL

[WSDLWiki] Article about WSDL on Wikipedia. http://en.wikipedia.org/wiki/WSDL

[WSDLList] *WSDL Team Mailing List* mobility-wsdl@mimuw.edu.pl

[XML] W3C Recommendation, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008, http://www.w3.org/TR/REC-xml/

[XMLSchema] W3C Recommendation, *XML Schema*, 2004, http://www.w3.org/TR/#tr_XML_Schema

Appendix A

Documentation

A.1. Overview

This chapter provides documentation of the REST Mobility system. It is divided into three main sections:

- **Server installation and configuration.** This section describes the process of installation and configuration of the server-side software (server and persistence libraries).
- **Client installation and configuration.** This section describes the process of installation and configuration of the client-side software (client and registry libraries).
- **User's guide.** This section describes, how the whole system can be used.

A.1.1. General notes

There are two ways of installing the provided software:

- downloading and compiling sources,
- downloading war or jar archives.

In the first case we will assume that after downloading sources you entered the root directory of the REST Mobility sources (probably called **source**, containing **pom.xml** file and several directories with names starting with **rest-mobility-** prefix). In this case any configuration change can be made by altering an already built war or jar archive. In the second case any configuration change can be made by editing configuration files before compilation and packaging or also by altering an already built war or jar archive after it. In addition, when starting the client GUI or the server standalone applications you can specify path to their configuration files, which will have precedence over these from jars.

A.2. Server installation and configuration

The server can be installed and run in a servlet container (war archive) or as a standalone application. This section covers both cases.

A.2.1. Requirements

The following requirements need to be fulfilled in order to successfully install and run the REST Mobility server:

- Java SE Development Kit (JDK) 1.6 (the latest build recommended),
- some servlet container installation (if you are going to run the server in a servlet container; Apache Tomcat can be recommended),
- SVN installation (if you are going to compile the sources yourself),
- Apache Maven 2 installation (if you are going to compile the sources yourself),
- Internet access if there exists a project dependency which is not present in the local Maven repository (if you are going to compile the sources yourself),
- a web browser (optional – this can be used as a simple client).

A.2.2. Installation

Installing and configuring the server from sources consists of the following steps:

1. Obtain the actual version of the sources from SVN repository.
2. Edit configuration files (see Section A.2.3).
3. Enter the root directory of the REST Mobility sources (see Section A.1.1). To compile and package the sources issue:

   ```
   mvn package
   ```

   You will find applications in the:

   - war – **rest-mobility-server-transport.war** file in the **./rest-mobility-server/rest-mobility-server-war/target/**
   - standalone – **rest-mobility-server-standalone-release** directory in the **./rest-mobility-server/rest-mobility-server-standalone/target/**
4. If you are going to run the server in a servlet container deploy the REST Mobility war archive into the servlet container.

If you do not want to compile the sources yourself, you should do the following:

1. Obtain an application (war archive or directory with jar application, libraries and sample files).
2. Unzip the archives:
   - In case of war archive you should unzip it and then in the **WEB-INF/lib** directory you will find jar archives. You should then extract jars that you are willing to edit.
   - In case of standalone application in the directory with application you should find three components: **lib** directory, **xml** directory and **rest-mobility-server-standalone.jar** file. In the **lib** directory you will find jar archives. You should then extract jars that you are willing to edit.

3. Edit configuration files (see Section A.2.3).
4. Zip the archives, that you extracted and edited in the previous steps. Remember that archives should be given the same name as before extracting. Structure and location of the archives should not be changed.
5. If you are going to run the server in a servlet container deploy the REST Mobility war archive into the servlet container.

A.2.3. Configuration

There are four main components that should be configured in the REST Mobility server:

1. ports,
2. logging,
3. persistence,
4. SSL/TLS.

Point 3 is described in Section A.2.4 and Point 4 is described in Section A.2.5. To configure Point 1 you should specify at least one of two numbers:

- a port on which HTTP connections will be accepted – property **httpPort** in the **server.standalone.properties** file; setting it to *0* means that you do not want to accept HTTP connections,
- a port on which HTTPS connections will be accepted – property **httpsPort** in the **server.standalone.properties** file; setting it to *0* means that you do not want to accept HTTPS connections.

Of course, the above configuration is only needed when you run the server as a standalone application.

As we use SLF4J/log4j combo for logging, you should also tailor log4j to your needs (Point 2). Log4j configuration will not be described here. However, in its configuration files you can find one non-standard property called **redirectJUL**. You should set it to *true* if you want SLF4J to catch log messages coming from JUL (java.util.logging) and pass them to log4j. Redirecting JUL allows you to have all log messages in one format but can affect system performance. Moreover, if you are going to run the server in GlassFish, you have to turn off redirecting JUL, as it causes GlassFish to crash.

A.2.4. Persistent Storage

Configuration

To use persistence layer provided with the REST Mobility you should perform several actions. Firstly, create a directory, in which representation will be stored. Let **$StorePath** denote path to this newly created directory. Set value of the **filesystem.prefix** property in **persistence.properties** file (**rest-mobility-persistence.jar**) to the **$StorePath**. Finally you should place in the **$StorePath** the following files and directories:

1. empty directory named **agreement**,

2. empty directory named **course**,
3. empty directory named **nominatedStudents**,
4. empty directory named **organization**,
5. XML Schema file named **MobilitySchema.xsd**, which contains data type definitions from the Mobility WSDL.

Without these files and directories the persistence layer and consequently the whole server will not work.

The use of other implementations

To use another implementation you should do the following steps:

1. Write it. It should be placed under **rest-mobility-persistence** project and implement interface from **pl.edu.usos.mobility.restimpl.persistence** package.
2. Substitute the current entries in the file **rest-mobility-persistence-context.xml** with ones that will suite your implementation. Bean's ids should not be changed.

A.2.5. SSL/TLS

In the case of a standalone server, to enable SSL/TLS certificates you should set the following properties in the **server.standalone.properties** file:

1. keystorePath,
2. keystorePassword,
3. keyPassword,
4. keystoreType,
5. truststorePath,
6. truststorePassword,
7. truststoreType.

If you are going to run the server in a servlet container you should consult the container's documentation to discover how SSL/TLS certificates can be enabled.

For more informations on using SSL/TLS certificates see Section A.4.

A.2.6. Running

If you want to run the server in a servlet container you should run the container and deploy the server war archive into it. A way of achieving this should be described in the container's documentation.

If you want to run the server as a standalone application you should enter the directory with it and type:

```
java -jar rest-mobility-server-standalone.jar $path
```

where **$path** denotes a path to the configuration file (if it is not present, then default configuration file will be taken).

A.3. Client installation and configuration

A.3.1. Requirements

The following requirements need to be fulfilled in order to succesfully install and run the REST Mobility client:

- Java SE Development Kit (JDK) 1.6 (the latest build recommended),
- SVN installation (if you are going to compile the sources yourself),
- Apache Maven 2 installation (if you are going to compile the sources yourself),
- Internet access if there exists a project dependency which is not present in the local Maven repository (if you are going to compile the sources yourself).

A.3.2. Steps

Installing and configuring the client from sources consists of the following steps:

1. Obtain the actual version of the sources from SVN repository.
2. Edit configuration files (see Section A.2.3).
3. Compile and package the sources. In the directory named **source** type the following:

   ```
   mvn package
   ```

   You will find applications in the:

   - client-transport– **rest-mobility-client-transport.jar** file in the **./rest-mobility-client/rest-mobility-client-transport/target/**
   - client-gui– **rest-mobility-client-gui-release** directory in the **./rest-mobility-client/rest-mobility-client-gui/target/**

If you do not want to compile the sources yourself, you should do the following:

1. Obtain an application (war archive or directory with jar application, libraries and sample files).
2. Unzip the archives: in the directory with application you should find four components: **lib** directory, **xml** directory, **wadl** directory and **rest-mobility-client-gui.jar** file. In the **lib** directory you will find jar archives. You should then extract jars that you are willing to edit.
3. Edit configuration files (see Section A.2.3).
4. Zip the archives, that you extracted and edited in the previous steps. Remember that archives should be given the same name as before extracting. Structure and location of the archives should not be changed.

A.3.3. Configuration

In order to run the client GUI application you should configure the following components:

1. logging,
2. SSL/TLS,
3. registry,
4. resources path.

As we use SLF4J/log4j combo for logging, you should tailor log4j to your needs (Point 1). Log4j configuration will not be described here. However, in its configuration files you can find one non-standard property called **redirectJUL**. You should set it to *true* if you want SLF4J to catch log messages coming from JUL (java.util.logging) and pass them to log4j. Redirecting JUL allows you to have all log messages in one format but can affect a system performance.

Client GUI is provided with some sample files (these sample files are taken from the Mobility). If you want the client GUI application to help you select the appropriate file you should set the property **resourcesDirectory** in the **client.gui.properties**. If you do so and choose **LOAD FROM FILE** button the application will take you to the appropriate file for given client organization, server organization and operation name (if such file exists).

Configuration of Points 2 and 3 are described in Section A.3.4 and A.3.5.

A.3.4. TLS/SSL

To enable SSL/TLS certificates on the client side you should set the following properties in the **client.transport.properties** file in the **rest-mobility-client-transport** project:

- keystorePath,
- keystorePassword,
- keyPassword,
- keystoreType,
- truststorePath,
- truststorePassword,
- truststoreType.

For more informations on using SSL/TLS certificates see Section A.4.

A.3.5. Registry

In order to prepare registry library for operation you should specify the following properties in the **registry.properties** file in the **rest-mobility-registry** project:

- method – you should set it to *file* if WADL document will be retrieved from a file system or *http* if WADL will be retrieved through HTTP/HTTPS from GEMBus registry,

- pathTemplate – in case of **method**=*file* you should set it to a path to a WADL – the string **{organizationId}** in this property will be replaced with actual value of the server organization (the sample value of this property is **/home/restMobility/{organizationId}/wadl.xml**); in case of **method**=*http* you should set it to the registry URL (e.g. **http://registry.restmobility.usos.edu.pl/**).

A.3.6. Running

If you want to run the client GUI application you should enter the directory with it and type:

```
java -jar rest-mobility-server-standalone.jar $path
```

where **$path** denotes a path to the configuration file (if it is not present, then default configuration file will be taken).

A.4. Creating SSL/TLS certificates

This section contains superficial instruction of creating SSL/TLS certificate. This process is quite complex and detailed instruction is out of the scope of this documentation. However, it is well documented and the relevant information can be found on the Internet. The process of creating signed SSL/TLS certificate and using it in Java programs consists of the following steps:

1. Creating a new keystore.
2. Generating the Certificate Signing Request (CSR).
3. Certificate signing by a CA.
4. Importing signed and CA certificates into the keystore.

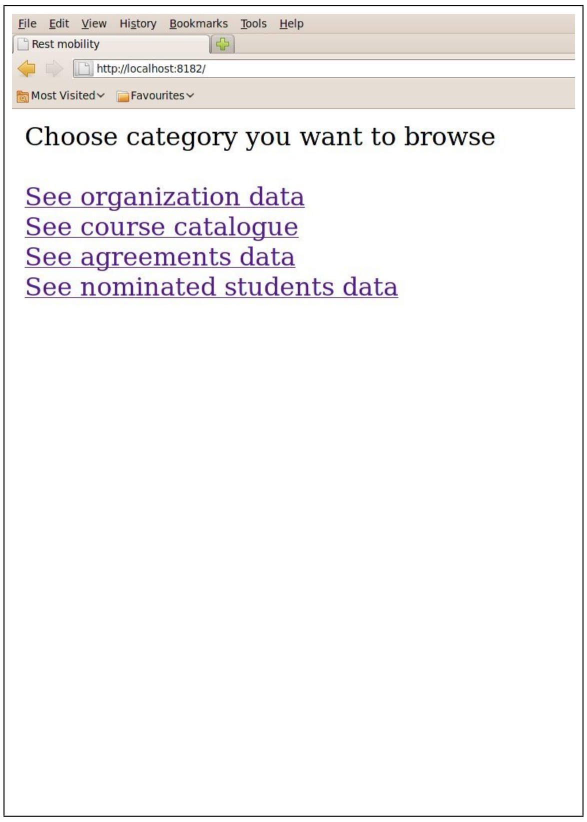
If you are going to use a web browser as a client you should also convert your certificate into PKCS12 format.

A.5. User's guide

A.5.1. Client-transport libraries

RestMobilityClientImpl object from **rest-mobility-client-transport.jar** archive can be used in other applications to perform requests to the REST Mobility server. It depends on **rest-mobility-registry.jar**. When creating RestMobilityClientImpl object you should provide it with RestMobilityRegistryImpl object. Both RestMobilityClientImpl and RestMobilityRegistryImpl should be properly configured (properties listed in Sections A.3.3, A.3.4 and A.3.5 should be set in these two objects).

By invoking appropriate methods from the RestMobilityClientImpl object you can make requests to the REST Mobility server.



Choose category you want to browse

See organization data
See course catalogue
See agreements data
See nominated students data

Figure A.1: Homepage of the REST Mobility server

A.5.2. Web-browser as client

You can use a web-browser as a client. In order to do so, just type the address of the server in the browser address bar. The page that you will see is shown in Figure A.1. If you are going to use encrypted connection you will probably need to import and/or accept some certificates (see Section A.4). In Figure A.2 you can see a prompt to accept a server certificate. You have to do it manually because the web browser does not know the location of your keystore/truststore and tries to use commonly known ones from the Web. When using provided client-transport library it is enough to import CA certificate into the keystore/truststore. In Figure A.3 you can see how to choose a certificate that will be presented to the server.

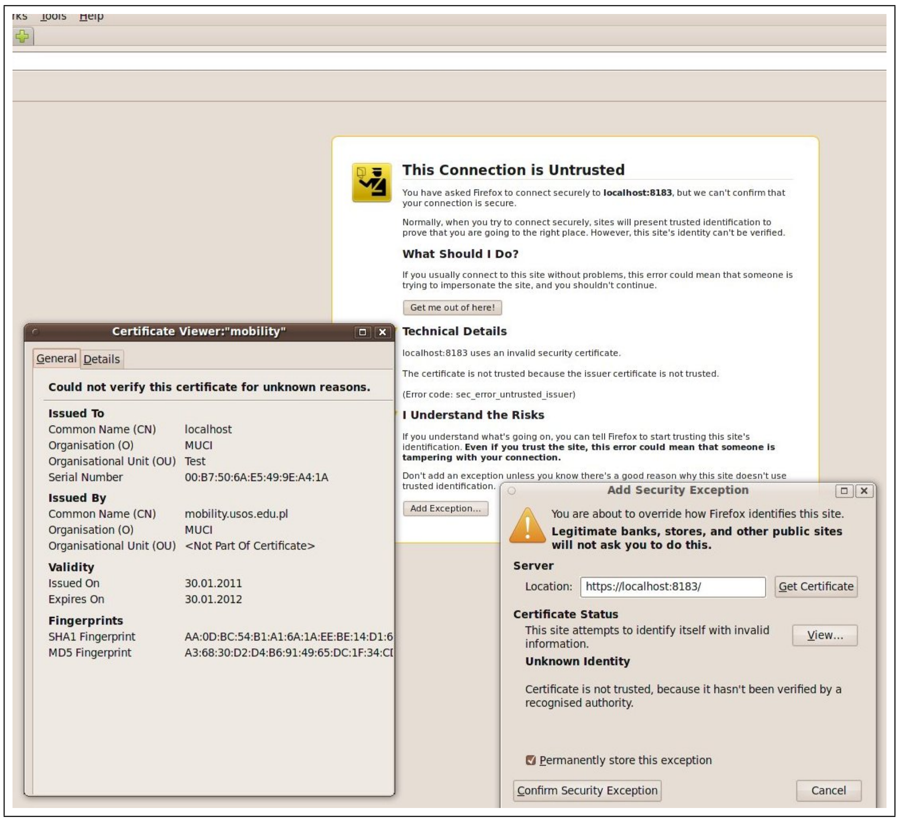
A.5.3. Client GUI

The Client GUI program consists of three windows:

- Main window – see Figures A.4 and A.7.
- Preview window – see Figures A.5 and A.8.
- Response window – see Figures A.6 and A.9.

A typical process of making a request in the Client GUI program consists of three steps:

- Provide the program with all necessary information (main window).
- Check and eventually correct the data that will be used for making the request (preview window).



This Connection is Untrusted

You have asked Firefox to connect securely to **localhost:8183**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Technical Details

localhost:8183 uses an invalid security certificate.

The certificate is not trusted because the issuer certificate is not trusted.

(Error code: sec_error_untrusted_issuer)

I Understand the Risks

If you understand what's going on, you can tell Firefox to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

Certificate Viewer:"mobility"

Could not verify this certificate for unknown reasons.

Issued To	
Common Name (CN)	localhost
Organisation (O)	MUCI
Organisational Unit (OU)	Test
Serial Number	00:B7:50:6A:E5:49:9E:A4:1A
Issued By	
Common Name (CN)	mobility.usos.edu.pl
Organisation (O)	MUCI
Organisational Unit (OU)	<Not Part Of Certificate>
Validity	
Issued On	30.01.2011
Expires On	30.01.2012
Fingerprints	
SHA1 Fingerprint	AA:0D:BC:54:B1:A1:6A:1A:EE:BE:14:D1:6
MD5 Fingerprint	A3:68:30:D2:D4:B6:91:49:65:DC:1F:34:CI

Add Security Exception

You are about to override how Firefox identifies this site.
Legitimate banks, stores, and other public sites will not ask you to do this.

Server

Location: https://localhost:8183/

Certificate Status

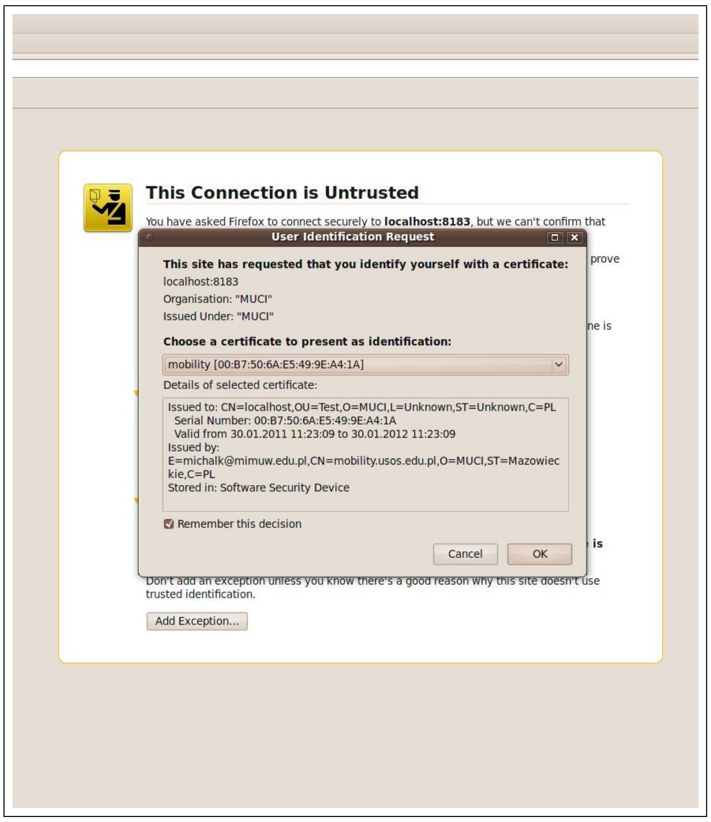
This site attempts to identify itself with invalid information.

Unknown Identity

Certificate is not trusted, because it hasn't been verified by a recognised authority.

☑ Permanently store this exception

Figure A.2: Information about the REST Mobility server certificate



This Connection is Untrusted

You have asked Firefox to connect securely to **localhost:8183**, but we can't confirm that

User Identification Request

This site has requested that you identify yourself with a certificate:
localhost:8183
Organisation: "MUCI"
Issued Under: "MUCI"

Choose a certificate to present as identification:

mobility [00:B7:50:6A:E5:49:9E:A4:1A]

Details of selected certificate:

```
Issued to: CN=localhost,OU=Test,O=MUCI,L=Unknown,ST=Unknown,C=PL
  Serial Number: 00:B7:50:6A:E5:49:9E:A4:1A
  Valid from 30.01.2011 11:23:09 to 30.01.2012 11:23:09
Issued by:
E=michalk@mimuw.edu.pl,CN=mobility.usos.edu.pl,O=MUCI,ST=Mazowiec
kie,C=PL
Stored in: Software Security Device
```

☑ Remember this decision

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

Figure A.3: Providing the REST Mobility server with a client certificate

- Watch a result of your request (response window).

Such processes for sending agreement data can be seen in Figures A.4 – A.6 and for getting agreement data in Figures A.7 – A.9.

Main window

In the main window you should insert information about operation you want to perform:

- Client Organization Id – first text box from the top on the left side of the window. This is id of the organization, on behalf of which you are performing operation.
- Server Organization Id – second text box from the top on the left side of the window. This is id of the organization, whose servers will perform the operation.
- Resource – first combo box from the top on the left side of the window. This is a resource that will be the subject of the operation.
- Method – second combo box from the top on the left side of the window. This is an action that you are going to perform on the resource.
- Message content – text area on the right side of the window. In this window you should insert appropriate XML document prepared for the Mobility. The program will try to extract from it information need to perform an operation (like agreementId, courseId, studentId, etc.). You can load a content of this area from file – to do this press **LOAD FROM FILE** button and choose the file.

The program continuously tries to transform data from the main window into that in the preview window. However, it will start doing it only after you leave the area, that you have just edited. If data in the preview window does not change you can try pressing **UPDATE PREVIEW** button. If still nothing changes you should check the data that you have just edited – it can be impossible to extract needed information from it. To clean all the data that you have entered press **CLEAN** button.

Preview window

In the preview window you can see the actual data, that will be used in a request:

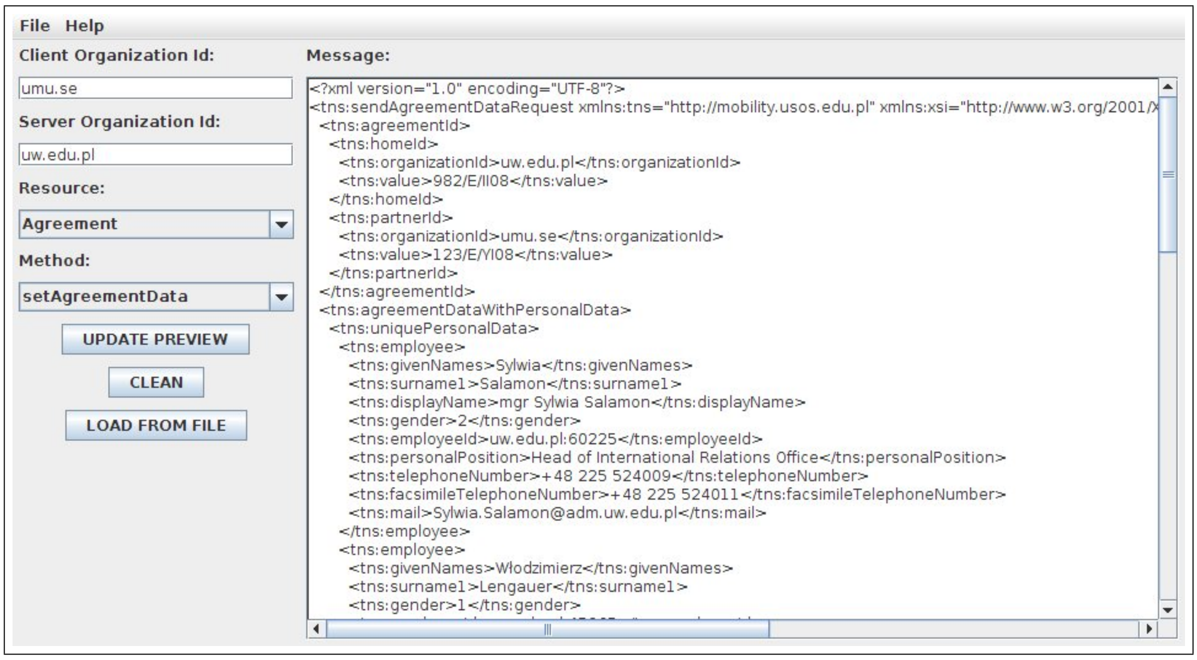
- URL template used for constructing actual URL – first label from the top.
- The actual URL – first text box from the top. You can edit this property before making the request.
- The HTTP method that will be used in the request – first combo box from the top. You can edit this property before making the request.
- The message body – first text area from the bottom.

Making the request is achieved by pressing **PROCEED** button.

Warning: If you edited some properties in this window you have to be prepared that these changes will be lost when you switch to the main window. This is due to keeping data in the main window and in the preview window consistent.

A.5.4. Response window

In this window you can see a result of your request.



Client Organization Id: umu.se

Server Organization Id: uw.edu.pl

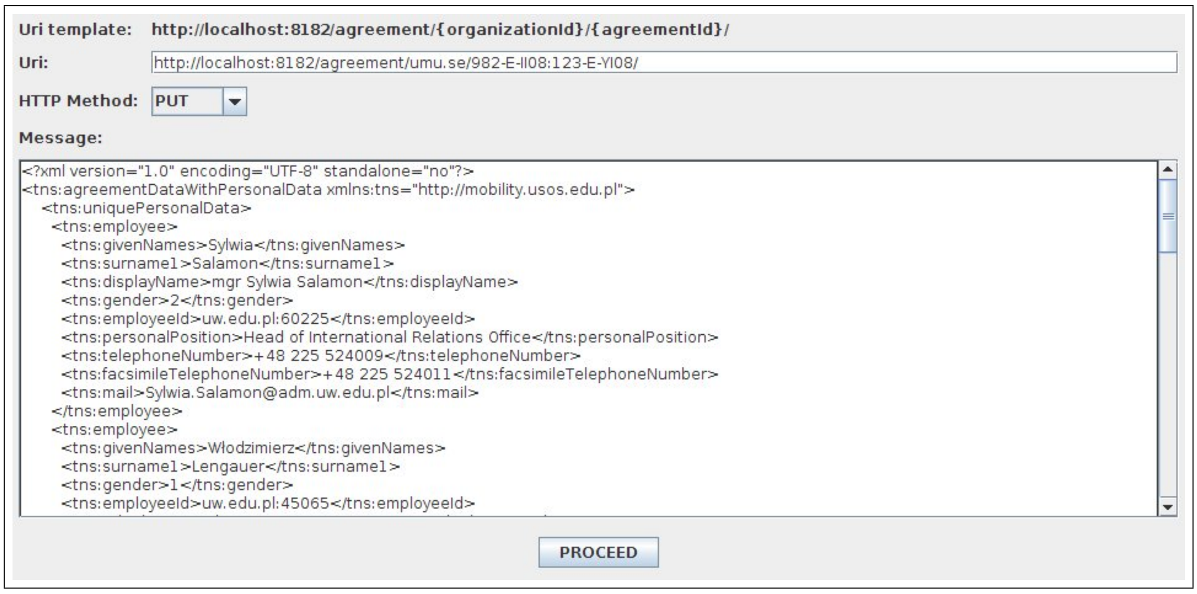
Resource: Agreement

Method: setAgreementData

Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:sendAgreementDataRequest xmlns:tns="http://mobility.usos.edu.pl" xmlns:xsi="http://www.w3.org/2001/
  <tns:agreementId>
    <tns:homeId>
      <tns:organizationId>uw.edu.pl</tns:organizationId>
      <tns:value>982/E/II08</tns:value>
    </tns:homeId>
    <tns:partnerId>
      <tns:organizationId>umu.se</tns:organizationId>
      <tns:value>123/E/YI08</tns:value>
    </tns:partnerId>
  </tns:agreementId>
  <tns:agreementDataWithPersonalData>
    <tns:uniquePersonalData>
      <tns:employee>
        <tns:givenNames>Sylwia</tns:givenNames>
        <tns:surname1>Salamon</tns:surname1>
        <tns:displayName>mgr Sylwia Salamon</tns:displayName>
        <tns:gender>2</tns:gender>
        <tns:employeeId>uw.edu.pl:60225</tns:employeeId>
        <tns:personalPosition>Head of International Relations Office</tns:personalPosition>
        <tns:telephoneNumber>+48 225 524009</tns:telephoneNumber>
        <tns:facsimileTelephoneNumber>+48 225 524011</tns:facsimileTelephoneNumber>
        <tns:mail>Sylwia.Salamon@adm.uw.edu.pl</tns:mail>
      </tns:employee>
      <tns:employee>
        <tns:givenNames>Włodzimierz</tns:givenNames>
        <tns:surname1>Lengauer</tns:surname1>
        <tns:gender>1</tns:gender>
```

Figure A.4: Main window of the Client GUI – sending agreement data



Uri template: http://localhost:8182/agreement/{organizationId}/{agreementId}/

Uri: http://localhost:8182/agreement/umu.se/982-E-II08:123-E-YI08/

HTTP Method: PUT

Message:

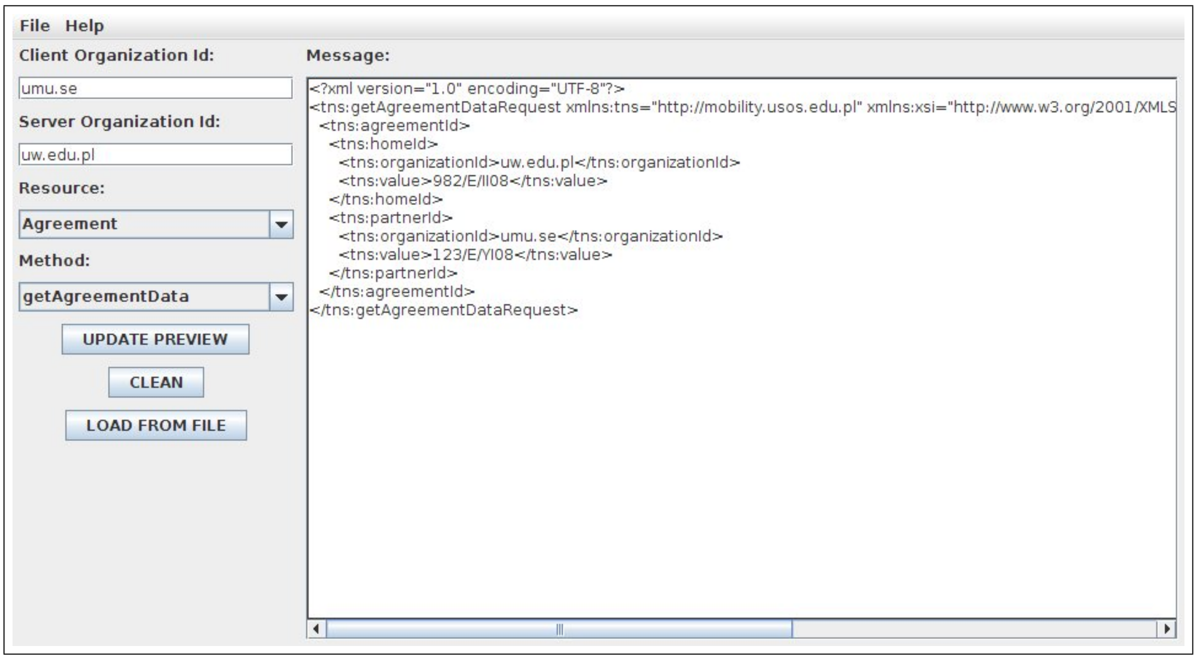
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tns:agreementDataWithPersonalData xmlns:tns="http://mobility.usos.edu.pl">
    <tns:uniquePersonalData>
      <tns:employee>
        <tns:givenNames>Sylwia</tns:givenNames>
        <tns:surname1>Salamon</tns:surname1>
        <tns:displayName>mgr Sylwia Salamon</tns:displayName>
        <tns:gender>2</tns:gender>
        <tns:employeeId>uw.edu.pl:60225</tns:employeeId>
        <tns:personalPosition>Head of International Relations Office</tns:personalPosition>
        <tns:telephoneNumber>+48 225 524009</tns:telephoneNumber>
        <tns:facsimileTelephoneNumber>+48 225 524011</tns:facsimileTelephoneNumber>
        <tns:mail>Sylwia.Salamon@adm.uw.edu.pl</tns:mail>
      </tns:employee>
      <tns:employee>
        <tns:givenNames>Włodzimierz</tns:givenNames>
        <tns:surname1>Lengauer</tns:surname1>
        <tns:gender>1</tns:gender>
        <tns:employeeId>uw.edu.pl:45065</tns:employeeId>
```

Figure A.5: Preview window of the Client GUI – sending agreement data

Response:

```
OK
```

Figure A.6: Response window of the Client GUI – sending agreement data



Client Organization Id: umu.se

Server Organization Id: uw.edu.pl

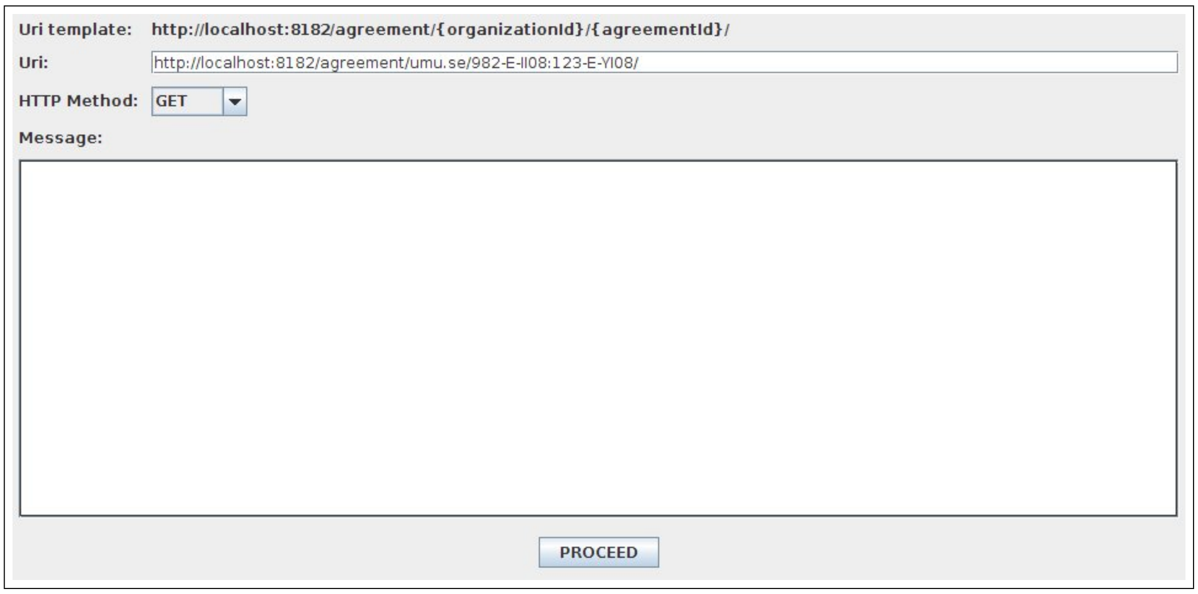
Resource: Agreement

Method: getAgreementData

Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:getAgreementDataRequest xmlns:tns="http://mobility.usos.edu.pl" xmlns:xsi="http://www.w3.org/2001/XMLS
  <tns:agreementId>
    <tns:homeId>
      <tns:organizationId>uw.edu.pl</tns:organizationId>
      <tns:value>982/E/II08</tns:value>
    </tns:homeId>
    <tns:partnerId>
      <tns:organizationId>umu.se</tns:organizationId>
      <tns:value>123/E/YI08</tns:value>
    </tns:partnerId>
  </tns:agreementId>
</tns:getAgreementDataRequest>
```

Figure A.7: Main window of the Client GUI – getting agreement data

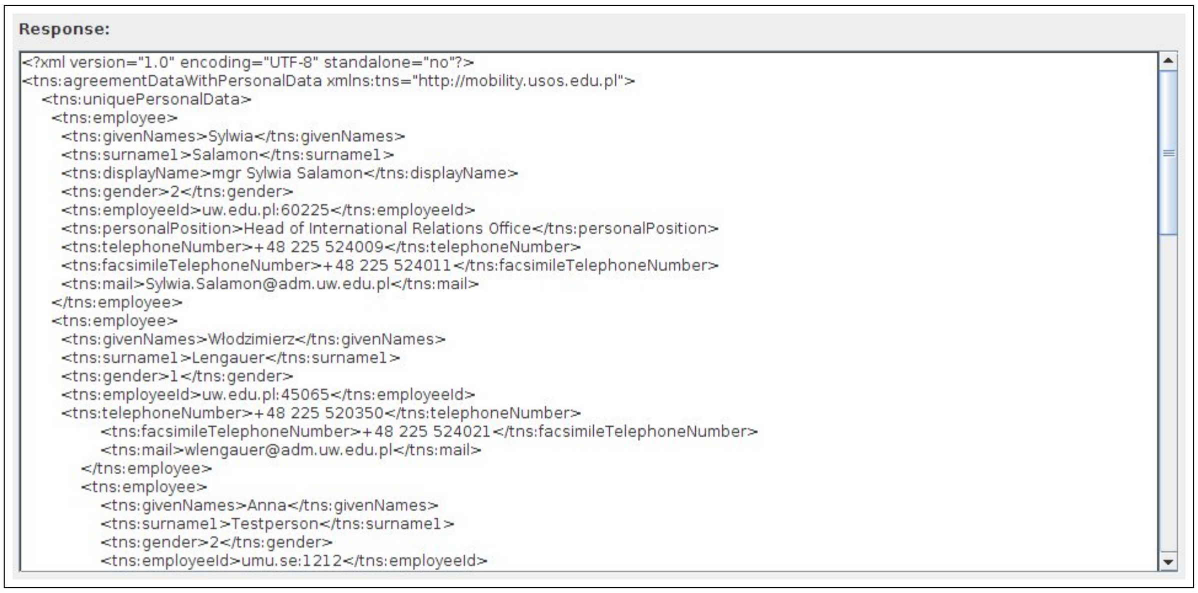

Uri template: **http://localhost:8182/agreement/{organizationId}/{agreementId}/**

Uri: http://localhost:8182/agreement/umu.se/982-E-II08:123-E-YI08/

HTTP Method: GET

Message:

Figure A.8: Preview window of the Client GUI – getting agreement data


Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tns:agreementDataWithPersonalData xmlns:tns="http://mobility.usos.edu.pl">
  <tns:uniquePersonalData>
    <tns:employee>
      <tns:givenNames>Sylwia</tns:givenNames>
      <tns:surname1>Salamon</tns:surname1>
      <tns:displayName>mgr Sylwia Salamon</tns:displayName>
      <tns:gender>2</tns:gender>
      <tns:employeeId>uw.edu.pl:60225</tns:employeeId>
      <tns:personalPosition>Head of International Relations Office</tns:personalPosition>
      <tns:telephoneNumber>+48 225 524009</tns:telephoneNumber>
      <tns:facsimileTelephoneNumber>+48 225 524011</tns:facsimileTelephoneNumber>
      <tns:mail>Sylwia.Salamon@adm.uw.edu.pl</tns:mail>
    </tns:employee>
    <tns:employee>
      <tns:givenNames>Włodzimierz</tns:givenNames>
      <tns:surname1>Lengauer</tns:surname1>
      <tns:gender>1</tns:gender>
      <tns:employeeId>uw.edu.pl:45065</tns:employeeId>
      <tns:telephoneNumber>+48 225 520350</tns:telephoneNumber>
          <tns:facsimileTelephoneNumber>+48 225 524021</tns:facsimileTelephoneNumber>
          <tns:mail>wlengauer@adm.uw.edu.pl</tns:mail>
        </tns:employee>
        <tns:employee>
          <tns:givenNames>Anna</tns:givenNames>
          <tns:surname1>Testperson</tns:surname1>
          <tns:gender>2</tns:gender>
          <tns:employeeId>umu.se:1212</tns:employeeId>
```

Figure A.9: Response window of the Client GUI – getting agreement data

Appendix B

CD-ROM Contents

The attached CD-ROM contains:

- this document in two forms: PDF format and LaTeX source,
- WADL document describing the Mobility RESTful web services,
- software package in a form of war and jar archives and its sources (project tree).