U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

MSc

Academic Management Information Interoperability Platform for Higher Education Institutions

Lucas Carvalho de Paula

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Ciência de Computadores

2021

MSc

2.º CICLO

FCUP
2021

U.PORTO

Academic Management Information Interoperability Platform for Higher Education Institutions

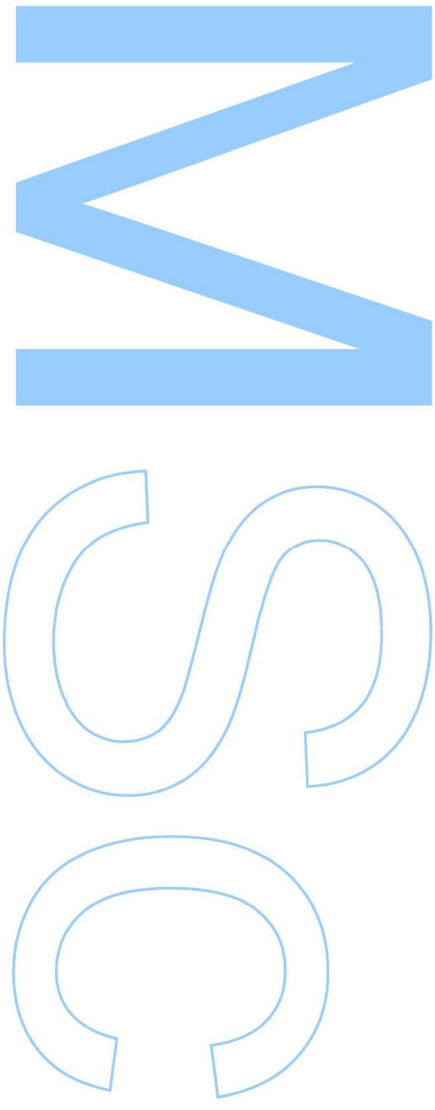
Lucas Carvalho de Paula

FC

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

MSc

Academic Management Information Interoperability Platform for Higher Education Institutions



Lucas Carvalho de Paula
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2021

Orientador
Manuel Eduardo Correia, Professor Associado,
Faculdade de Ciências da Universidade do Porto

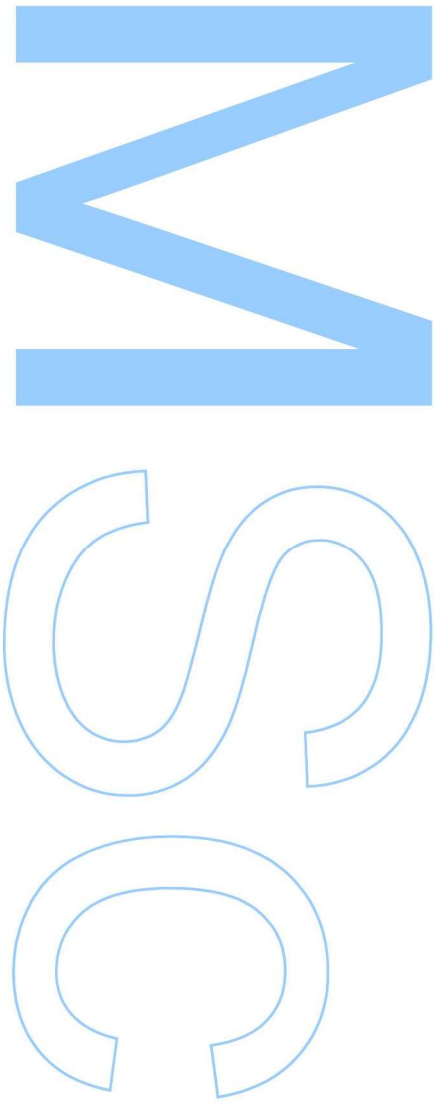
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

I want to thank my supervisor, Professor Manuel Eduardo Correia, for all his support and guidance. I also extend my appreciation to my UP Digital co-workers from the UNISF project. Without them, this work would not be the same.

Lastly, I want to thank my family and friends for their continuous support.

Abstract

The generality of Higher Education Institutions (HEIs) developed their academic information systems through the years, and these systems, regarding dimension and functionality, are complex. However, the HEIs are also feeling the necessity of developing shared courses (shared study cycles) and optimize their formative offer and extract the maximum benefit from the academic resources provided by the union. The realization of such a task brings difficulties to the different information systems, given the constant need to exchange administrative management and financial information. This project aims to build an integration middleware that will allow each HEI to share the desired data using a canonical format in a trusted channel to provide support to the information management and the required processes related to shared courses. Changes to existing HEI information systems must be minimal, and the user of these systems shall not notice the data exchange process. In addition, the aim is to simplify the whole process of having a shared course by dramatically reducing human inputs.

This dissertation describes the development of a framework based on Web Service (WS) that supports the business processes of sharing a course. This framework is meant to integrate with the HEI academic information system and provide mechanisms to share the data safely and canonically. We aim to cease the manual document exchange that comes from the nature of having a shared course. A canonical data format is a data schema that everyone in the network can understand and transform internal data to canonical and vice versa. A shared course is a course provided by more than one HEI and can benefit from the resources of all involved HEIs.

In order to do our work, we studied the Erasmus Programme's [6] data interoperability platforms.

Contents

List of Tables

List of Figures

Listings

Acronyms

UP University of Porto

HEI Higher Education Institution

API Application Programming Interface

WS Web Service

EWP Erasmus Without Paper

URL URL

UNISF University without Borders

FCUP Faculty of Sciences of the University of Porto

SIGARRA Sistema de Informação para Gestão Agregada dos Recursos e dos Registos Académicos

CNR Change Notification Receiver

IIA Interinstitutional Agreements

SC Study Cycle

UI User Interface

HR Human ResourcesHuman Resources

ECTS European Credit Transfer System

LA Learning Agreement

ToR Transcription of RecordsTranscripts of Records

P2P Peer-to-peer

SMP Student Mobility Plug

NCP National Contact Point

DNS Domain Name System

RAIDES Registo de Alunos Inscritos e Diplomados do Ensino Superior

LOS Learning Opportunity Specification

LOI Learning Opportunity Instance

IDN Internationalized Domain Names

ESI European Student ID

PIC Participant Identification Code

Chapter 1

Introduction

This chapter will present the project, give some context for the problem at hand, motivation to solve the problem, the goals, and show the outline of the following chapters.

1.1 Project Presentation

This project is the result of a partnership of universities of Porto and the University without Borders (UNISF) [24] consortium. It aims to create an academic management information interoperability platform for Higher Education Institutions (HEIs), in a way to implement shared courses (subsection 1.2.4) with minor changes to the existing academic information systems. For the users of these systems, there must be no difference in the system after connecting to the Broker.

The project proposal was defined as a development of a data exchange broker for the involved HEIs, using the best practices from systems like Erasmus Without Paper (EWP) [16] and EMREX [15] that are already validated in Europe. The exchanged data need to be digitally signed (this kind of system guarantees validation). When flowing from an HEI to another, the data must have a canonical representation, and every HEI broker must be able to translate internal data format to the canonical one and vice versa. Each HEI will have a broker to transform the data, sign, and send.

The canonical data must allow the different HEI systems to see the shared student as a regular student and take advantage of universal authentication systems like eduGAIN [13] so that the student can have access to all the services available at the university. The canonical

representation must provide a unique id for students. The eduGAIN is a global infrastructure that connects national identity federations that are operated in the Higher Education sector around the world.

The Broker will have a set of Application Programming Interfaces (APIs) that implements these data exchange processes. Some API sets are optional so that it can ensure the phased entry in the process. Every HEI broker must have a standard manifest file with a list showing what kind of functionalities and business processes are implemented by a particular HEI. From the distributed nature of the network, it is necessary to exist a particular broker that will gather the manifest of every connected HEI broker and help them to find each other to make Peer-to-peer (P2P) connections, like a Domain Name System (DNS) server. The Figure 1.1 shows the initial draft of the system.

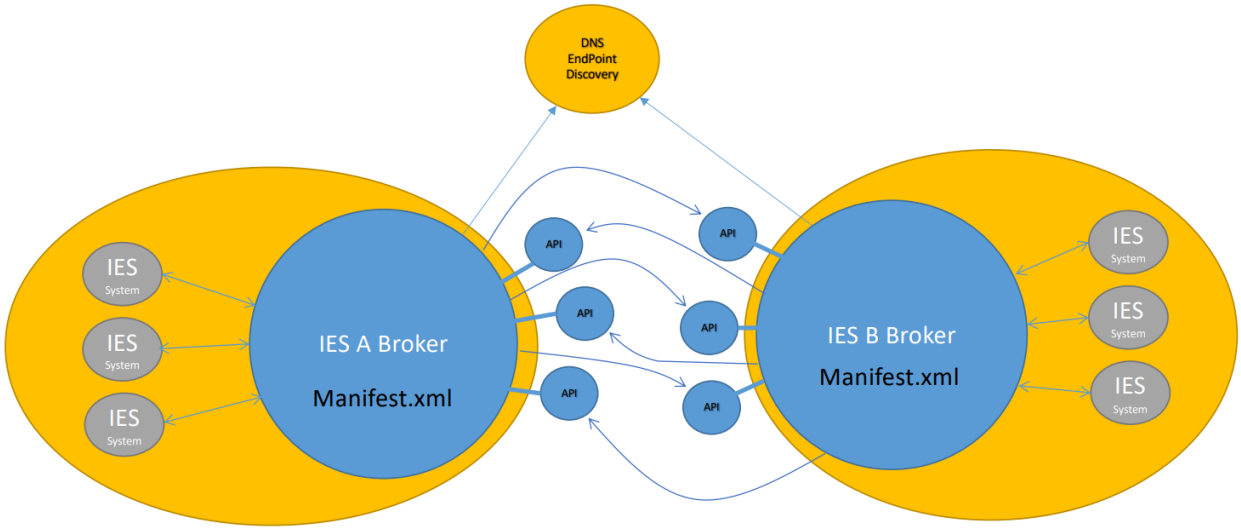


Figure 1.1: Initial project architecture.

In summary, if an HEI A wants to exchange data with HEI B, then they need to start by fetching the manifest of one another to verify what kind of APIs are available and know what business processes are exposed.

1.2 Context

1.2.1 Canonical Form

The canonical form is the platform standard way of representing data. It is the minimum representation of data that anyone joining the platform must support. As an example, when sending data of a student in a canonical format, it will assume a unique representation, and any HEI can transform that standardized data to its internal data.

Without the canonical form, every HEI needs to know how to transform each of others HEIs internal data format to its own.

1.2.2 Information Systems

An information system is an essential part of every organization. We can describe it as an automated system to handle high loads of information. Through the years, the evolution of computers, regarding hardware and software, allowed the enhancement of these systems. As a consequence, people stopped manually archiving physical documents and saved time to engage in other matters.

The central purpose of such systems is the possibility to achieve an efficient organization of resources like Human Resources (HR), materials, equipment, technology, money, and the information itself. Since the stored data is relevant for human analysis afterward, the system must have the latest version of the stored information or quickly get it.

Our focus will be on the student information management systems, which are information systems that handle academic student data in universities. This kind of system is there to help the academic staff with administrative tasks centralized in one place. It also supports faculty to streamline work processes.

We must keep in mind that each university keeps the data in its own defined format and concepts. Two different universities working with educational data and doing the same processes from an outside view does not imply that the steps to achieve similar results are the same, and integrating inter-university activities can get messy quickly. That is where our problem steps in. We are aiming to implement a tool to help universities to have shared courses.

1.2.3 Course and Course Units

A course is an organized set of course units that do not offer an academic degree. A Study Cycle (SC) is a course that provides such a degree, like graduation, master course, and doctorate.

A course unit is a unit of learning that, after some assessment, is translated to a final grade.

1.2.3.1 Naming convention for the dissertation

From now on, every mention of ”courses” is implied that we are mentioning ”course/SC”.

1.2.4 Shared Course

A shared course [5] is when a course/SC is provided by more than one HEI and is a result of a partnership, either national or international. It has an administrative HEI responsible for registering the administrative acts, student data editing, the definition of study plans, grading, and certificate emission. The non-administrative ones act like the information reader, meaning they can neither register nor edit the information. It is also possible to have shared courses where the non-administrative HEIs ones can handle administrative processes like student applications and subscriptions. However, they have to communicate the list of students to the administrative HEI.

A shared student needs to be distinguishable from ordinary students. They, by default, are seen as normal students by the HEIs. In some specific courses, they can be divided into two cases: if the student enrolls into the administrative HEI, it is seen as normal; if the student enrolls into a non-administrative HEI, it is seen as external. A normal student can become external and vice versa if the administrative HEI changes. Regardless, the students should be able to identify themselves in any university.

The processes required to establish shared courses are:

1. Student applications: the administrative HEI commands the application process and communicate to others, or every HEI handles applications, and each one sends its applications list;
2. Student enrollments: for normal students is handled smoothly as for ordinary students, for

external ones administrative intervention from HEI's academic services is needed;

3. Course Frequency: the students that do not complete the course must enroll again on the HEI of the new course edition;
4. Non-degree Diplomas: as soon as the students complete the teaching part of the shared course, this information and the grade must be communicated;
5. Dissertations: the dissertation defense must be required on the institution that the student enrolled on the dissertation course unit, the jury must include at least one person from each involved HEI;
6. Completed students: each HEI must communicate the list of students that completed the course (completed the teach and dissertation parts);
7. Communication between HEIs: through an electronic document, usually Excel, issued within certain defined time windows;
8. Diplomas: may be issued by the administrative HEI or each can provide one;
9. Fees and Tuition: the student must pay the fees on every HEI involved, but the tuition is paid on the host HEI;
10. Government inquiry report: each HEI must be able to gather information for the report (RAIDES report, for Portuguese HEIs).

Due to the nature of different information systems, internal HEI regulations, and the law in different countries, it is impossible to have shared courses working out of the box. The information needs to be transformed, either leaving or arriving at the institution. To reach this goal, we need middleware. As the name suggests, middleware is a piece of software that sits in the middle of a process. It can interact with the information before return to the natural flow.

1.2.4.1 Problems to fix with an integration middleware

The main problem we want to fix when establishing a shared course is the communication of the data itself. It is currently done by producing and exchanging an electronic document manually, usually Excel, with the data they want to communicate. Also, the data is always sent using the sending HEI data standards. We will propose a canonical standard.

No implemented system/platform does the job of automatizing this sharing process. However, we will introduce two systems that are notable close to what we want. In the next chapter, we will analyze them in more depth. The shared courses working with an automatized system will help to make the process more streamlined, saving time and money. Otherwise, we would need mass amounts of emails with the requested information about the courses every day. Keep in mind that a shared course is not student mobility. The student does not leave his HEI to take a feel course units abroad. It is simultaneously active in all HEIs.

In chapter 3, we will present the approach we took to streamline the process of sharing a course.

1.2.5 Erasmus+ Programme

We studied two systems from literature, EWP and EMREX. Both belong to the Erasmus+ Programme project and tackle student mobility, which means they created a piece of software to make the bridge we need.

Erasmus Programme [6] is a project that has been in operation since the 1980s. It allows students to study at universities in the EU member states for set periods, from three months to one year. The biggest motivation of the program is to stimulate opportunities for learning across Europe. Some benefits of an abroad experience are broadening horizons, increase motivation to learn, develop cultural awareness; improve and gain language skills. The members of Erasmus use the European Credit Transfer System (ECTS) to implement the academic credits. This way, the national and the abroad credits speak the same language. Consequently, a student can use the credits it earns in its course abroad to its qualification.

The ECTS [7] is a tool that makes studies more transparent. The unified system help in recognizing academic qualifications and study periods abroad. It is present on the Erasmus Programme learning agreement and the transcript of records. The Learning Agreement is the agreement signed by the student, the receiving, and sending HEIs. It contains all the course units and their respective ECTS the student will take abroad. The Transcript of Records is the document that the receiving HEI sends after the student ends the time abroad. It contains the course units, the ECTSs and the grades.

The steps needed to have an Erasmus process working are:

1. Two or more HEIs sign an Interinstitutional Agreements (IIA), it sets the framework of the mobility among them.
2. After IIA is signed, the sending institution begins the recruitment of students to the program, and it sends the nominations to the receiving HEI.
3. Some receiving HEIs may require Transcription of Records (ToR) from the sending one, this step is not covered by the EWP, students can use EMREX for that [10].
4. The receiving HEI accepts the nominations.
5. After the receiving HEI accepts the nominations, the student and both HEIs prepare the Learning Agreement (LA). The LA is what the student will study abroad, the grading system, and the ECTS credits. It needs to be accepted by the student and all the involved HEIs.
6. After the LA is approved, the student can travel to the receiving HEI to start the studies.
7. When the student finishes the studies abroad, the receiving HEI sends the ToR, this document allows the sending HEI to recognize the abroad credits. The ToR is a validated document with the results the student got abroad of what was previously planned.

The EMREX [8] was born with the idea of facilitating student access and recognition of his credits abroad. The main idea of this system is to connect different universities so the student can easily use the platform to get his abroad credits. These credits are validated and can be used for recognition or other ends. In this system, the student is the one starting the action and using the data.

The EWP [14] was born to automatize the steps of the Erasmus+ Programme itself, meaning that it will be able to: sign documents digitally, exchange fact sheets about ECTS, requirements, and grading systems for people coming from abroad, send nomination lists, negotiate study programs with the student and the other HEI, after the study period, it gets easy to send and get the transcript of records of the student. Differently from EMREX, on this one, the HEI starts the process.

In the next chapter, we will see in more depth how EWP and EMREX work and reuse everything we can from both systems. EWP implements student mobility Web Services (WSs). Still, we will quickly see that it can support any WSs because of its generic nature.

1.3 Motivation

This system will aim to automatize the whole process of sharing information of a pointed course with the selected partners and hopefully minimizing the manual intervention and making the process more streamlined.

This project is being developed as one of the projects of UP Digital. The connection of the University of Porto with the mentioned EWP system is under development in parallel to this project. They already had the piece of code that implements the EWP platform's security, and we had access to it so that we could reuse as much code as necessary.

The most obvious motivation is that achieving such a system will hopefully reduce most of the paperwork and the HR related to activities of this nature that involves other institutions. Of course, humans will still be needed, but in small numbers.

1.4 Goals and Contribution

The main goal of this master thesis is to define and implement the necessary components to create the platform that will be used to share the academic data. The following objectives were defined:

1. To search for systems that are up and running with the purpose of share or automatize any academic data/process;
2. To study the motivation and architecture of these existing platforms;
3. To study the security layer of these platforms;
4. To propose a common set of academic concepts related to the courses so WSs can be defined;
5. To propose a definition and normalization of data in order to have interoperability of the information;
6. To propose the architecture and the components of our platform;
7. To expose some data endpoints that are related to the academic concepts normalization [18];

8. Show in detail how the components work;
9. Expose a use case;
10. Show the prototype.

The main contribution will be the design of such a system and a working prototype showing the flow to share the data among two HEIs.

The shared data will not force any pattern (like a standard data format), meaning that any institution can share course information as it pleases. However, the one thing that must be followed is the academic concepts to have well-defined endpoints for each concept.

1.5 Thesis outline

This section presents the order of the subsequent chapters, followed by their contents.

In chapter 2 we present the analysis of the studied systems, showing the one that served as the main inspiration to the contribution of this thesis.

In chapter 3 we present the solutions we thought to solve the problem and show the one we considered to work on.

In chapter 4 we show how it is organized the architecture of the system and also details the communication processes of the components.

In chapter 5 we show the data integration mechanisms, where the information about data mapping and notifications is detailed.

In chapter 6 we show the prototype and detail every interface.

In chapter 7 we conclude this thesis, presents the future work and some final thoughts.

1.6 Chapter Summary

We presented an overall view of the project, showing some early concepts. Then, we gave some context to the concepts that will be handled in the dissertation, specially stating the problems

we want to fix on shared courses. We presented some motivation, goals, and, finally, the thesis outline.

Chapter 2

State of the art

The analysis presented in this chapter is based on two systems that share academic information, not exactly the way we want. However, student information systems, Erasmus Without Paper (EWP) [16] and EMREX [15], with emphasis on the first and lately explaining why the latter was not so relevant.

Both analyzed projects are pioneering in integrating Student Information Systems and are funded by the Erasmus+ Programme. The main goal of both systems may look similar, internationalization and student mobility, but we will see the differences in more detail in the following sections.

2.1 EWP

EWP [16] is a system focused on supporting the Higher Education Institution (HEI) administration for student exchanges through the Erasmus+. The system was thought to automatize the steps of the Eramus+ mobility process from start to end.

We will start presenting the mobility process and the motivation, then show from a network architecture level and end showing the system Application Programming Interfaces (APIs) and how they are used.

2.1.1 Eramus+ mobility process

Eramus+ mobility process [14] is a program that improves the quality and the dimension of the high education and scaffolds the cooperation among HEIs from Europe. It stimulates mobility in Europe and improves transparency and the recognition of academic credits.

To tackle this problem, EWP divided the Erasmus+ process into the following steps:

1. Signing the interinstitutional agreements between HEIs. It specifies the number of mobilities, the type, and more;
2. Institutions exchange fact sheets with basic information of mobilities for people coming from abroad. Among the basic information is e-mail addresses; language requirements; ECTS requirements; grading system, and more;
3. The institution sending the students starts to prepare a list of nominations that need to be approved/rejected by the receiving institution;
4. The nominated students negotiate the study program with receiving and sending mobility coordinators, then sign a learning agreement, so access to the course catalog is needed. This step is quite common to change;
5. After the study period, the receiving institution should send the transcript records with the student's achievements to the sending institution (This step is technically carried out in the same way as EMREX).
6. Calculate student stipends based on the length of the study period, which is calculated from the arrival and departure dates from receiving institution.

The description of how this steps are automatized with the EWP Web Services (WSs) are described in subsection 2.1.7.

2.1.2 Motivation

Reduce the human intervention in the Erasmus Mobility process by breaking the manual paperwork and turning them into WSs to have a more optimized flow.

2.1.3 Architecture

EWP uses a Peer-to-peer (P2P) architecture [25] that uses the service discovery pattern [23] with a central node that holds a list with all partner nodes connected and its available WS. The central node is unique and provides a catalog to the other nodes, and it is called Broker. The other nodes are the Host nodes, the ones that are acting on the system. Finally, an HEI is an institution that uses a Host as an interface to communicate with other Hosts Figure 2.1.

2.1.3.1 Service discovery pattern

Service discovery pattern provides a distributed network architecture that allows better interoperability and reduced maintenance. It is great for scalability. The network available WS endpoints do not need to be static. They need to be communicated to the broker node so that other clients can always find the available WS in each node. The broker node keeps a catalog of WS. Whenever a new node joins the network, the other nodes do not need to change the implementation.

2.1.3.2 HEI

HEI is the institution that has an interest in receiving and collect data. To be able to join the system, an HEI needs to be winged by a Host.

2.1.3.3 Host

A Host is composed of a set of HEIs and a set of APIs to be implemented. When an HEI asks for information from other HEI, what happens is a communication between the Hosts. If not specified, the HEI which the sender wants the data, only the Host, then the request is sent to every HEI winged by that Host. The same happens to the response, and the response goes to every HEI winged by the Host that started the request. An HEI being under a Host means that it is possible to answer every API available on that Host. It is identified by its SCHAC id [30]. A Host acts as a server and a client.

2.1.3.4 Registry

The Registry acts as a central node, the Broker, and it keeps a catalog with information about the Hosts and how to find them. Academic data do not pass through the Registry. It is only used to get information about APIs implemented by other Hosts. Once this information is obtained, the communication no longer needs the Registry. The Registry structure is similar to that of a Host but does not have any HEIs.

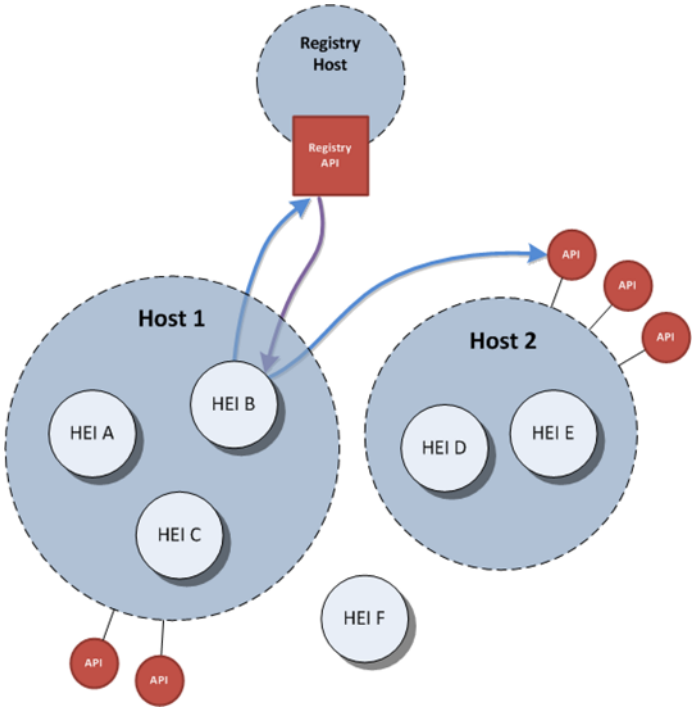


Figure 2.1: EWP Architecture **source**: https://github.com/erasmus-without-paper/ewp-specs-architecture/blob/v1.10.0/images/diagram-step3.png

2.1.4 APIs

EWP has a set of well defined APIs that can be divided in three categories [14]:

- **Primary Network APIs**: this category is meant for setup and communication of the network itself;
- **General Purpose APIs**: this category is meant for share academic related information;
- **Erasmus Mobility APIs**: this category is meant for implement the abstracted steps of the Erasmus+ process (subsection 2.1.1).

All the exchanged data uses the XML format. APIs that are not in the set of Primary Network can be seen as removable. It is possible to replace them with a completely different set of APIs and the system would work just fine. This fact will play a significant role later on in this thesis.

If we remove the General Purpose and the Erasmus Mobility APIs, we will have a clean system framework to implement our system. We need to develop our APIs to use over the clean EWP.

2.1.4.1 Primary Network APIs

Primary Network APIs are the network base APIs and are used to connect and find other Hosts inside the EWP network. It is a set composed of Discovery, Echo, and Registry APIs.

- **Discovery**: Send basic information of all the platform resources available on that Host. This WS is used by the Broker to gather information of all Hosts into a catalog;
- **Echo**: Receive all security headers of the system and confirms that the security protocols were correctly implemented. This WS is used by others to test its security middleware implementation;
- **Registry**: Implemented by the central network node to expose all the data gathered from the common nodes. This information works like a phone book to guide Hosts to other Hosts.

2.1.4.2 General Purpose APIs

General Purpose APIs are meant to provide support to feed general academic information needed by the Mobility APIs. These are simple data and are composed of the set Institutions, Organizational Units, and Courses. It is easy to know by its name.

2.1.4.3 Erasmus Mobility APIs

Erasmus Mobility APIs will cover the steps defined for the Erasmus+ process. It is composed of Interinstitutional Agreements; Interinstitutional Agreements Approval; Outgoing Mobilities; Outgoing Mobility Learning Agreements; Incoming Mobilities; Transcripts of Records; Change

Notification Receiver. For some context below, sending HEI is the one that sends the student, and receiving is the one receiving the student.

- **Interinstitutional Agreements**: Allow partners to compare their copies of the interinstitutional Erasmus+ mobility agreements to spot errors;
- **Interinstitutional Agreements Approval**: Allows an HEI to approve agreements sent by their partners in the interinstitutional agreements;
- **Outgoing Mobilities**: Implemented by the sending HEI, allows the receiving HEI to read, write and enumerate mobilities stored on the sending HEI;
- **Outgoing Mobility Learning Agreements**: Implemented by the sending HEI, allows the receiving HEI to read and accept learning agreements stored on the sending HEI;
- **Incoming Mobilities**: Implemented by the receiving institution. It allows the sending HEI to read the receiving HEI's part of the information related to outgoing mobilities;
- **Transcripts of Records**: Implemented by the receiving institution. It allows the sending institution to retrieve Transcripts of Records issued by the receiving institution.
- **Change Notification Receiver**: Allows partners to listen for changes in objects of interest kept in other partners. It uses the observer pattern. Partners subscribe to the owner of information so they can listen for changes.

2.1.5 Data representation standards

EWP uses an extension of the ELMO XML format to represent data (section 2.3), as addresses [32], phone numbers [33] and academic terms [34]. Major part of the data is standardized to the Erasmus+ agreement format which is known by all parts of EWP.

Hosts are identified by its SCHAC id [30].

2.1.6 Security

EWP has a set of security methods to be implemented (TLS Certificate, HTTP Signature), being these two only a part of the set. The developer chooses the security method. It is only necessary to explicit it on the manifest used by the Discovery. If compatible, it is possible to use more than

a single authentication method on the same request. It is also possible to implement security methods that are out of the EWP scope. If one of the sides of the communication does not offer support for it, the communication is aborted.

2.1.7 EWP mobility process

We will now describe a scenario of mobility, using the EWP WSs, between HEI A and HEI B.

The mobility process [10] starts with all HEIs taking part in the mobility process, signing Interinstitutional Agreementss (IIAs) with each other. These requirements are processed via the EWP network, fully electronically. In this process the IIA and IIA CNR APIs are used. IIA is used by an HEI B to access all the IIA related to HEI B stored on an HEI A. IIA Change Notification Receiver (CNR) allows an HEI to get notified whenever any IIA, related to them, is changed on other HEI servers.

In order to sign the IIA, the HEI B uses the IIA Approval API to get the IIA and the hash. After computing the hash locally and compare to the receiving hash, if the hashes match, HEI B confirms the approval. HEI B confirmation triggers an event that uses the IIA CNR and notifies HEI A. Hash comparison is used to ensure that HEI B is signing the version of the IIA it last saw. This signing process can be observed of the flow shown on appendix Figure B.1.

After approving the IIA, let us suppose that HEI B will send students to HEI A. In this nominations scenario, we will be using the Outgoing Mobility API, Incoming Mobility API, and the CNR for these two.

HEI B prepares a list of nominated students. Then uses HEI's A Outgoing Mobility CNR to tell HEI A. HEI A can now access the student mobility list using HEI's B Outgoing Mobility API.

After review and verify the students, HEI A uses HEI's B Incoming Mobility CNR to tell HEI B. HEI B can now access the list of approved students on HEI's A Incoming Mobilities API. This flow can be observed on appendix Figure B.2.

After the nominations are verified by the receiving partner, the student itself is notified, by his HEI internal system, that he should now fill his Learning Agreement (LA). The student uses the sending HEI web app to fill in the necessary information. The web app uses the receiving HEI Courses API to fill information. When the student is done, it approves and sends to the other actors to approve it (both sending and receiving HEIs).

The sending HEI verifies the LA approved by the student. After the verification, the sending HEI adds its approval and uses the Outgoing Mobility LA CNR of the receiving HEI. The receiving HEI can now check the Outgoing Mobility LA to approve or refuse the LA. Then the receiving HEI uses sending HEI Outgoing Mobility LA API to approve or refuse the LA. If refused, all the LA approvals are removed, and the LA is revised. All the actors get notified in the case of approval, and the student is ready to travel. The flow shown in appendix Figure B.3 shows in more detail.

After the mobility ends, the sending HEI can use the receiving HEI Incoming Mobilities API to get the dates of arrival and departure.

To get the student results, the receiving HEI make the student's Transcription of Records (ToR) available via the Incoming Mobility ToR API and uses the sending HEI Incoming Mobility ToR CNR to signal it. The student then, using the web app from the sending side, can verify the ToR and, if it is final, mark it as ready for recognition.

After mark that the ToR is ready for recognition, the sending HEI starts the recognition process. After this step, the mobility process is complete, check appendix Figure B.4.

2.2 Other related works

Here is a summary of another work that was not directly relevant to the platform's development. However, it added more overall knowledge about distributed systems and helped decide the type of architecture chosen for the project.

2.2.1 EMREX

EMREX [14] is a system that aims to help student credits recognition in Europe. Institutions do not take action to get data. Differently from EWP, the student is the one who gets data, and it helps students to transfer them in a secure and trustful way. Also, it does not handle all the steps of the mobility process, only the recognition.

2.2.1.1 Use case

A student finishes his short-term studies in a receiving institution and returns to a sending institution. It wants to transfer its achievements. It logs into the local system, uses the EMREX registry to locate the receiving institution, logs again on the receiving institution system. After approving for data export, the student is redirected back. Data is transferred in an XML format with PDF included.

2.2.1.2 Motivation

Make the student credits recognition smooth and allows the student to easily access his European Credit Transfer System (ECTS) credits in partner institutions, to use in any end.

2.2.1.3 Architecture

As shown on Figure 2.2, the EMREX, as EWP, uses the service discovery pattern [23] where the EMREG component is the catalog of registered network partners and available services. The institution that will receive the student data needs to implement the EMREX Client that will request the data. In order to respond to the EMREX Client, the institution that will provide the data need to implement the NCP, which is the component that will return the response after the second student login.

2.2.1.4 Network Components

- **EMREX Client**: Web application that the student uses to start the transfer of results.
- **NCP**: National Contact Point. Then point that the EMREX client contacts to fetch results.
- **SMP**: Student Mobility Plug. The plugin that the EMREX client uses to enable communication with the NCP.
- **EMREG**: Central registry with information of where to find the registered NCPs.

2.2.1.5 Security

Since EMREX is embedded in the institution's local system, each institution takes care of its security systems.

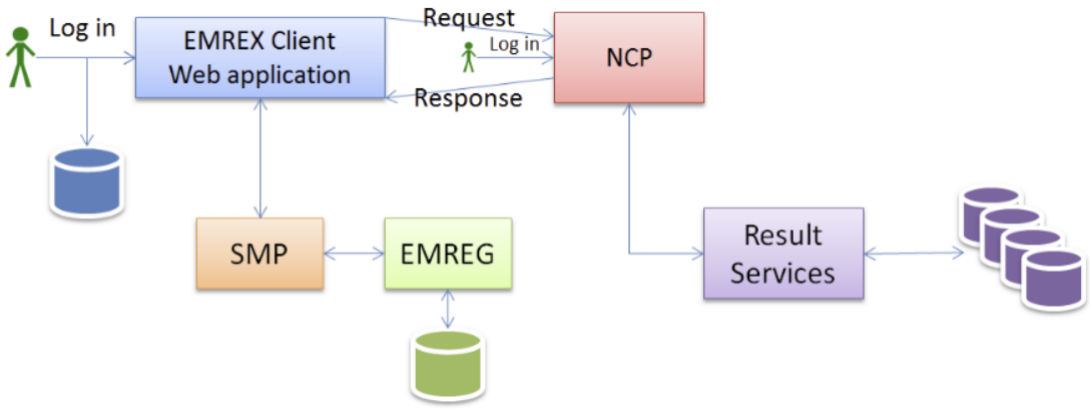


Figure 2.2: EMREX Architecture **source**: https://www.erasmuswithoutpaper.eu/sites/default/files/upload_files/EUNIS_2017_paper_3.pdf.

2.2.1.6 EMREX Recognition process

The business process of sign a learning agreement and send the students is made out of the scope of EMREX.

EMREX system takes place when the student needs to get a verified version of his ToR he got abroad to be recognized on his current institution, and this can lead to two mobility cases. On the first one, the student takes a few courses abroad, gets back, and wants to retrieve his results from the abroad institution to be recognized in his current one. On the second one, or the student will continue his studies in another country and want to use his already earned ECTS credits.

In both situations, the student needs to first log in on the institution's web app that will receive the credits.

The institution web app will use the Student Mobility Plug (SMP), and the SMP will retrieve the EMREX partners list from EMREG. The student then picks the institution that holds his credits and selects its National Contact Point (NCP). After selecting the NCP, the student is prompted to log in to the selected institution. When logged, the student can select the credits

and submit them to get the results signed on the receiving side. The abroad institution then sends the signed ToRs and the receiving institution can use its SMP to verify the signature before import the results and recognize the credits. Check the appendix Figure B.5 for a diagram of the recognition flow.

2.3 Data representation standards

Both EMREX and EWP use extensions of the ELMO [14] representation standard for mobility data. ELMO is a hierarchical structure focused on learning outcomes.

ELMO XML format is used to have a canonical form to describe assessments, diplomas, diploma supplements, and ToRs for HEIs.

Here is an example of the format EWP adopted from ELMO to represent learning opportunities.

The Learning Opportunity Specification (LOS) component describes a single learning outcome, and it can contain several LOS, through the ”hasPart” element. The model can be seen on appendix Figure B.6. It specifies the:

- **identifier**: there can be several identifiers, it is typically the code that identifies the LOS in the local system;
- **title**: the title of the LOS (the course name);
- **type**: Degree Programme, Module or Course;
- **subjectArea**: the Erasmus subject area code for the LOS;
- **uscedCode**: the Erasmus ISCED code for the LOS;
- **url**: an URL to a web site with more detailed information;
- **description**: a description of the LOS;
- **specifies**: a concrete instance of LOS, it has its own schema LOI;
- **hasPart**: a list of LOSs that is contained within this LOS.

2.4 Chapter Summary

After the research on EWP and EMREX, it got clear that EWP would be used as a base for the system. It supports APIs that are out of its scope. We could even use EWP itself to implement just a few WSs without the need to build an entire system; EMREX is one of the steps of EWP. EWP is the only one that covers the problem of mobility. EMREX covers easy recognition, which is one of the steps of the mobility process, EWP seems like an evolution of EMREX. Both platforms do not go further on the matter of academic data. They do not have any problems with data normalization since the essential part of the data there are grades and student credits. We will need to define a set of normalized concepts even to start to exchange academic data. Every institution needs to be aware of what is the meaning of the data that is being shared. This kind of system that tries to integrate academic data is hard to make, mainly because everyone who joins the system needs to agree on the data normalization, which is a complex task. That explains why, for now, we did not define a fixed structure for the data, only for the concepts.

In order to build our system, we first need to create the EWP nodes with Primary Network APIs only. Then, it will support whatever data exchange API we need.

The data representation used in the development of our project will be proposed by the University of Porto (UP) academic management team. We can not rely on ELMO alone because it limits itself for mobility data, and shared courses are not a mobility scenario.

Chapter 3

Problem Definition

This chapter will get back to the problem and show the decisions we made to start building the platform prototype using the information provided in the previous chapter.

3.1 The Problem

We want to build a system that will support shared courses between Higher Education Institutions (HEIs). We need to create a platform with well-defined Application Programming Interfaces (APIs) that will allow us to have one institution to share all the data related to a specific course and to support notifications over the data in case of any changes. An institution should use the platform to select which other institutions will receive the data and future updates for a given course.

3.2 Our Approach

In order to tackle this problem, we picked one of the three solutions we had.

3.2.1 First solution

Since the presented Erasmus Without Paper (EWP) system is well designed and flexible. Our first solution was to use the EWP platform and develop only the necessary APIs to exchange academic information. EWP system supports any API, it is possible to build our system over the

already online EWP system. The downside of this approach is the obligation of an University without Borders (UNISF) partner to join the EWP if they want to have a shared course. We would also need to request permission to use the platform for other ends.

3.2.2 Second solution

The second solution we thought of was to make our version of the EWP system, which ends up with the good side of the first solution, but the downside is to build our system from scratch. The downside of this solution is not a problem at all because EWP has all its technical documentation of how the system is structured, open. The fact that University of Porto (UP) has a team working on implementing its connection to the EWP also supports this solution. We can access already written code as a reference, so we do not write the same thing twice.

3.2.3 Third solution

This third solution was born as an upgrade for the second. Basically, from the second solution, use an integration BUS to the job assigned to the broker. This BUS is supposed to ease the development. Here follows a bit more into the integration BUS:

3.2.3.1 Integration BUS

An Integration BUS is a software solution for application integration that takes care of message routing and transformation for us [21]. It is an entirely programmable broker that connects different applications using a secure channel. It is possible to set up a node that will send the data and set up the nodes that will receive that output. The BUS is also able to make transformations to that input before output to its end nodes. As an example, the IBM Integration BUS provides a comprehensive set of integration capabilities on an agile, secure, high-performance platform. It enables universal connectivity across enterprise systems, applications, and data.

3.2.4 Taken solution

We will now explain why we did not pick the other solutions and how each one led to the taken solution.

As said on the first solution, we neither want to force joining the EWP for UNISF partners nor ask permissions to EWP administrators, so the solution to implement our APIs over the EWP platform was discarded.

The problem with the third solution is that the EWP broker specification is too simple to use a dedicated integration BUS (Peer-to-peer (P2P) connection). The broker we need does not carry any information. It is used to fetch information about where the other nodes are, like a phone book (from the Figure 2.1. It is possible to see that the actual request is made from Host 1 to Host 2. The registry broker was used to locate the Host 2 API), an integration BUS would be too overwhelming, all its features would be overlooked, so the third solution was discarded.

The second solution is much simpler, and we even got access to the UP's EWP development virtual machine to get hands-on the security middleware that helped the development a lot, part of the security code was directly translated to python. The development of our platform was made using the python web framework Django and all the messages will be using JSON when applicable.

3.3 Data Parameterization

For our project, the ELMO XML data standard is not enough. It is limited to mobility data. UP academic management team was requested to provide an initial file [18] with the data concepts and parameterization required for sharing a course.

The concepts defined in the document are: Course, Diploma, Branch, Admission Contest, Application, Person, Study Plan, Course Unit, Agenda, Term, Evaluation Period, Student, Student State, Student Statute, Frequency Regime, Tuition, Student Enrollment, Course Unit Enrollment, Mobility, Recognition, Student Diploma. Follows an example of the concepts using the basic course information.

3.3.1 Courses/SC

A Course is a set of course units, does not provide an academic degree. A Study Cycle (SC) is a course that provides an academic degree.

A Course/SC is characterized by basic information like course full name, acronym, official

code, type, level, duration, start and end years. Following is presented some of the parameterizations for types, levels, and language as an example.

3.3.1.1 Naming convention for the dissertation

As stated on subsubsection 1.2.3.1, every mention to ”courses” is implied that we are mentioning ”course/SC”.

3.3.1.2 Courses/SC types

The Table 3.1 shows the parameterization for courses types.

Acronym	Name	Estatute
D	Doctorate	CG - Provides Degree
L	Graduation	CG
M	Master course	CG
MI	Integrated Master course	CG
E	Specialization	EC - Continuing Education
EA	Advanced Studies	EC

Table 3.1: Course type.

3.3.1.3 Course/SC level

This information classify a course regarding the classification levels (final or intermediary) and respective diplomas. Consider the cycles Table 3.2 to build the level Table 3.3.

Acronym	Name
1	1st cycle
2	2nd cycle
3	3rd cycle

Table 3.2: Course cycles.

Initial cycle	Final cycle	Qualification level	Description
1	1	L	Graduation/1st cycle
1	2	MI	Integrated Master course
2	2	M	Master course/2nd cycle
2	2	-	Specialization
3	3	D	Doctorate/3rd cycle
3	3	-	Advanced Studies

Table 3.3: Course level.

3.3.1.4 Course/SC duration

Duration is the number of academic years, semesters, or quarters the student must carry out. Table 3.4 shows the parameterization.

Acronym	Name
A	Years
S	Semesters
T	Quarters

Table 3.4: Duration type.

3.3.1.5 Course/SC Learning Languages

The main learning language for a course is one of its characteristics. This domain will also be used for course units and diploma supplements. Table 3.5 shows the parameterization.

Acronym	Name
EP	Portuguese and Spanish
ES	Spanish
P	Portuguese
PI	Portuguese and English

Table 3.5: Language.

3.4 Available Services

UP academic management team created a set of WS that will be used by the project to get the needed data for the exchange. This WS are described in more detail on the document [19]. The document was created based on the parameterization one [18]. All the Web Service (WS) were thought to support the shared courses. Since the endpoints were created exclusively by UP, we have an artificial canonical representation of data. Before show the WSs, we will present some conventions taken to present the data.

3.4.1 Identifiers and Conventions

3.4.1.1 HEI and Organic Unit ids

Based on the work from EWP [11], we decided to identify institutions and organic units with the SCHAC code of the institutions. SCHAC code identifies the institution using its registered Internationalized Domain Names (IDN).

SCHAC code is simple, and the fact that every HEI has a URL makes this solution future-proof. For UP it would be up.pt and for Faculty of Sciences of the University of Porto (FCUP) would be fc.up.pt.

In addition to this identifier, it would also be possible to use the Participant Identification Code (PIC) or the Erasmus code.

3.4.1.2 Person id

We decided that a list of identifiers would be necessary to identify a person properly instead of a single one. This way, we can identify a person using the civil number, fiscal number, passport number, or any other legal identifiers. Like the EWP, it is also possible, in some situations, to use extra attributes like a birth date.

The identifier is composed by three attributes:

- **type**: the type of the identification (e.g. civil card, passport, etc);
- **value**: the value itself;

- **register_date**: A register date. Some ids can expire, but it is desirable that all ids are stored, even the expired ones.

The European Student ID (ESI) is an id from the context of the MyAcademicID [4], that aims to easy the student identification on european institutions and with eduGAIN it allows students to easily use federate authentication. It can assume two formats:

- student in a national or regional scope: urn:schac:personalUniqueCode:int:esi:<country-code>:<code>
- student in an HEI scope: urn:schac:personalUniqueCode:int:esi:<sHO>:<code>

Where: <country-code> is the ISO 3166 country identifier; <sHO> is the SCHAC code of the HEI; and <code> is the student code that identifies the student on the scope it was emitted.

Despite being globally unique, a single person can have more than one ESI. However, it is still possible to identify. Listing 3.1 shows some examples of ids.

```
{"type": "NIF", "value": "000000000", "register_date": "1-1-12"}
{"type": "NIF", "value": "111111111", "register_date": "1-1-20"}
{"type": "NIC", "value": "648716234", "register_date": "1-1-12"}
```

Listing 3.1: Person id example.

3.4.1.3 Representing an Academic Year

To represent an academic year, we followed the EWP steps, so we used the format RRRR/YYYY. Example: 2020/2021.

3.4.1.4 ID Mapping Table

We decided to use a table to map the identifications of all entities, like courses, branches, diplomas, study plans, agendas, terms, evaluation periods, admission contests, applications, persons, students, enrollments, course unit enrollments, mobilities, recognitions, student diplomas, course units.

The goal is to achieve a correlation between ids in different systems. This solution can be generalized to all entities we are going to exchange data. The idea is, when asking for data from another HEI, we can save the data and insert a new entry on the table with the type of the data (e.g., course), the original owner, the original owner id, and the local id we just saved.

Type	Internal ID	External ID	Owner
course	L1001	L090909	up.pt
study_plan	PE1111	P8989	university.int

Table 3.6: Map table example.

From the Table 3.6, we can see that we asked from up.pt the course with id L090909 and stored locally as the course L1001, and from university.int, we asked the study plan P8989 and stored locally as the study plan PE1111. Using the table is always possible to locate the original data.

3.4.2 Web Service (WS)

The document [19] describes in detail all the available services, but we will show as an example the WS that returns basic course information.

3.4.2.1 Get Information about a Course/Study Cycle (SC)

- **Goal**: Given a course id and an academic year, get detailed information about the course;
- **Input**: Course id and academic year;
- **Answer attributes**: Check appendix for Table C.1 and openapi format Figure C.1;
- **JSON example**: Listing 3.2 shows a response from SIGARRA WS for the course 19941.

3.5 Development from a high-level perspective

We will describe the overview of the platform prototype, stating some conventions taken, and in the next chapters show the details of implementation.

```
{   "cce_id": 19941,
    "name": [{"lang":"pt", "description": "Licenciatura em Engenharia
        Geospacial"}],
    "initials": "L:EG",
    "type": "L",
    "dges_code": "L096",
    "cycle_start": "1",
    "cycle_end": "1",
    "course_duration": {"type": "A", "value": "3"},
    "credit": {"scheme": "ECTS", "level": "L", "value": "180"},
    "parent_institution_id": [{"hei_id": "fc.up.pt", "admin": ""}]
    "academic_year_start": "2019/2020",
    "academic_year_end": null
    "language": "PI"
}
```

Listing 3.2: Course JSON example.

3.5.1 Reference Architecture

For the base of the project, the part that regards security and communication, we implemented the best practices learned from EWP. We implemented all the EWP Primary Network APIs, so we can have a solid framework to build the shared courses Web Services (WSs). All communications are P2P.

This base environment will provide the security and the communication layers. We need to take care of the shared course business process layer. We will also use the EWP naming conventions for network nodes, the node that integrates to the HEI will be called host node, and the node that acts as a DNS will be called broker node.

3.5.2 Integration Mechanisms

3.5.2.1 Conventions

For this project, we decided not to exchange data using a canonical format, so other HEIs from UNISF consortium can test the system fast (they will not need to discuss the canonical format before start testing). The only things we kept from data representation were the data concepts

to have well-defined endpoints. However, to support a canonical format, we just need to add a data transformation layer when pulling data from HEI databases.

We also decided to keep a single HEI for course administration. This HEI is responsible for manage the shared course data and notify other HEIs whenever the original data is updated. Non-administrative HEIs will act as information readers only.

For the prototype, all the operations to write data to the database were not implemented. We only read. When demonstrating the sharing process, all the copied data will be saved to memory.

3.5.2.2 Integration with HEI Information System (Data push and pull)

Before setting up the platform host node, the HEI needs to expose Web Services (WSs) of the concepts we discussed. The platform host node will use these WSs to pull data from the HEI. When HEI WSs are ready, the host node repositories must be configured to pull data automatically whenever needed. The host node never directly accesses the HEI database. When using a real canonical format, the data transformation happens inside the repository class.

The repository class is meant to push and pull data from its HEI. It is responsible for keeping the data in the platform canonical format when flowing to other HEIs, and return to the internal format when pushing to HEI internal system. Figure 3.1 shows a graphical representation of the repository pattern. The transformation blocks are easily attachable to the system. As said, for the prototype, we are neither pushing data back to the HEI information system nor transforming it to a canonical format.

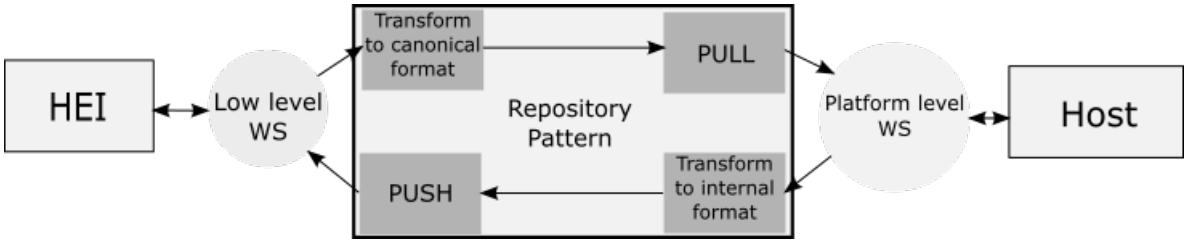


Figure 3.1: Repository Pattern data transformation.

3.5.2.3 Getting Visible on the network

In order to get visible to other nodes, it is needed to set up the host node Manifest WS. After the setup, we need to communicate the SCHAC code and the endpoint of our manifest to the network broker manager.

A broker is a special network node that holds information about all the host nodes connected, and it is used as means to achieve P2P connection between host nodes. It will harvest information from all the manifests periodically.

3.5.2.4 Course Sharing Process

After integrating the host node repository with HEI low-level HEI WSs, data will be available through the host node WS using a canonical format. Non-administrative HEIs need to be able to copy the data to their information system but keeping a mapping table, so they can know where is the source of the copy.

Before interacting with the non-administrative HEIs, the administrative HEI must insert the course information in its academic information system as it would do for a normal course (we do not want to modify processes from a user point of view), like create/add course units, create study plans, etc.

The course information WS is not public. In order to give access to the data, the administrative HEI must establish a handshake using the platform. We called this handshake ”agreement”. The agreement object carries information about the course to be shared and all the HEIs involved. To interact with the agreement, we created the tickets. The tickets are objects to locate the agreement on the administrative HEI. This agreement object needs to be accepted by all involved HEIs, in addition to grant access to course data, when the agreement is accepted by every HEI, it triggers the mapping event on the network.

The mapping event is when the non-administrative HEI automatically goes through a specific course and recursively copy, map and create an observer object on the administrative HEI. When the mapping event is over, the non-administrative HEI will have a copy of the agreement course and will be able to receive notifications from the administrative HEI in case of any changes to the linked data. The observer object allows the administrative HEI, having an id and type of data, to fetch all observers of that data so that it can use the notification WS.

After receiving the notifications of any changes, the non-administrative HEIs will have the opportunity to see what was changed before accepting the changes or not.

3.5.2.5 Triggering Notifications

The platform will have two endpoints to trigger notifications when any data is changed. The HEI information system must use both to send notifications automatically. The two notifications scenarios are:

- An existing data was modified: the HEI need to know the type and the id of the data to locate the observers (e.g., type is courseunits and id is 12837).;
- An non-existing data was just added to the system: for this case, the observers need to create a new map, so in addition to type and id, it needs to know the id for the parent of this new object that is already mapped, it is always possible to use the root course object as a parent (e.g., type is courseunits and id is 12837, parent_type is course and parent_id is 5874875).

An HEI must be able to embed the endpoint in its information system when updating or adding new information.

3.5.2.6 Human Interactions

After solving the problem of having a canonical format for the data and communicating the data automatically, we added a few human interactions with the system. As mentioned in subsubsection 3.5.2.4, after doing the process of creating a course, as a user would typically do on the HEI information system, the user needs to create the platform agreement and accept with the HEI partners in order to establish the shared course.

Further human intervention only takes place in case of data update on the administrative side. The non-administrative will need to check the changes before commit unless they modify the host node and their information system to commit data automatically.

3.5.2.7 Solving the Course sharing problem

As stated in subsection 1.2.4, the problem we wanted to solve was the communication of the data when dealing with shared courses and share the data in a canonical way. In addition to solving this problem, our solution allows us to perform every process (e.g., student applications and enrollments) required to manage a shared course on the administrative HEI as it was a normal course, and every non-administrative will have an updated copy. It happens thanks to the notification system.

3.6 Chapter Summary

We stated our problem and presented three approaches to solving it. We stated the pros and cons for all approaches and finished by picking one and presenting the reasons. After selecting the approach, we went through an example of the data parameterization that will be used in the platform and the required Web Services (WSs) to support shared courses' processes. Finally, we present a macro view of the platform architecture (the communication part) and the integration mechanisms (the academic process part).

Chapter 4

Reference Architecture

In this chapter, we go deep into the details of the reference architecture used to build the entire system, presenting the design patterns of choice and showing the function of each component of the network. The Web Services (WSs) that are going to be mentioned in this chapter are related to network security and communication only. Those used for mapping of academic information are going to be covered next, on the chapter 5. The code of the full implementation of the prototype is hosted on git in a private repository of University of Porto (UP).

All the WSs for the Host node is available for browsing on a swagger page when running the prototype.

4.1 Base

Following the format of the studied systems, the system we developed uses the service discovery pattern [23]. This pattern is characterized by a distributed software network with decoupled components that interact with WS calls, and there is a particular node named broker. This one is responsible for keeping a catalog of all services available on the network and where to find them. The only concern of the broker is to help hosts find and communicate with each other. All the academic data is communicated in a Peer-to-peer (P2P) way.

In order to implement the system, we used the python web framework Django [20], mainly because it is open source, provides fast development, and we are familiar with this tool.

Differently from Erasmus Without Paper (EWP) that uses XML format on all communications,

we decided to use JSON on this platform. The implemented WSs follows the best practices presented in the book [12].

As implemented in [25], for a host to be part of the system, it needs to implement the network discovery API and communicate to the broker administrator its intention to integrate. On the discovery, the host must expose all implemented Application Programming Interfaces (APIs) and institutions, using a JSON manifest.

In order to make the harvest of the manifest from all registered hosts, the broker part uses a harvest script that runs alongside the server, which is basically an HTTP client.

Some classes will use the singleton pattern[22]. The singleton metaclass implemented is thread-safe. The main idea of a singleton class is to have a single instance of that class, and every object will use this exact instance.

4.2 Broker

The broker is the only centralized node of the network. It is responsible for helping the host nodes to find each other with the aid of WSs. It has a single WS that returns a catalog of all hosts of the network.

In order to insert a host into the system, the broker administrator needs to know the Host SCHAC [30] and the discovery API URL.

The broker's catalog WS does not take parameters and can be accessed by anonymous users.

4.2.1 WS Catalog

This WS returns a compilation of all information harvested from registered hosts. The service structure is as follows:

- Method HTTP: **GET**
- URL path: **/api/catalog/**

After collecting the information provided by each host's manifest (subsection 4.3.2), the

broker returns the data as structured on block Listing 4.1. For each public key, a hash of the key is calculated using the SHA256 algorithm, which replaces the actual key on each host object. The actual value is indexed by hash in an entry called binaries. After receiving the header keyId, it is possible to recover the original key quickly. For the institutions, there is an entry to accumulate all institutions indexed by schacId.

```
{
  "hosts": [
    {
      "admin_email": ["string"],
      "admin_notes": "string",
      "apis_implemented": [...], //Listing 4.2
      "institutions_covered": {...}, //Listing 4.3
      "rsa_public_key": "<sha256-of-raw-rsa-pub-key>"
    },
  ],
  "institutions": {...}, //Listing 4.3
  "binaries": {
    "<sha256-of-raw-rsa-pub-key>": "<raw-rsa-pub-key>"
  }
}
```

Listing 4.1: Catalog response schema.

4.2.2 Harverster

Harvester is a script that runs in parallel with the broker server. The only role of this component is to: get the partners' information from the database shared with the broker; use the information to fetch the manifest file from each member; validate, sanitize and organize all manifests into a catalog. The changes mentioned on subsection 4.2.1 happen here.

The frequency of this process is configurable. Its default value is to execute the process every 30 minutes.

4.3 Host

The host is the leading actor of the network. It is the side that starts the communication process, which means the active part of the contact. In order to start the communication, the host needs to have its manifest exposed and registered on the broker.

The manifest holds some general-purpose information about the host, as administrator email; implemented APIs; covered institutions; and information about the public key (from the RSA key pair) for authentication. After being added, it can be found by others.

4.3.1 Security

Hosts use a security verification based on HTTP signatures inspired by [31] that follows the patterns presented on [17], [2] and [3]. This part was heavily inspired by the UP's EWP security middleware code that was provided to us. This helped us a lot. Instead of making the security from scratch, we had to understand what was already implemented.

A host must generate an RSA key pair and publish its public part using the WS Manifest (subsection 4.3.2).

In order to build the signature, the function core.util.payload.build_payload was designed. It takes the target URL, the HTTP method, and the body of the HTTP message. The central role of this function is to gather the necessary elements to generate the signature hash, as on the example 4.3.1, following the guidelines from [3] and [31], sign with the private key, using the algorithm rsa-sha256 [1], and return the headers in a way that the receiving host will be able to build the same hash and compare with the one that was sent, validating the signature.

Example 4.3.1 (request-target): post /foo\n
host: example.org\n
date: Tue, 07 Jun 2014 20:51:35 GMT\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
X-REQUEST-ID: 254eaafa-9343-4edd-87be-9d2b8da805a3

In order to know what items are part of the signature, it is necessary to send the header Authorization [3]. This header indicates the hash SHA256 of the public key (keyId); the algorithm

used to build the signature; the headers, which is the list of headers that are part of the signature; and the signature itself. An example can be seen at 4.3.2.

Example 4.3.2 Authorization: Signature keyId=”rsa-key-1”,
algorithm=”rsa-sha256”,
headers=”(request-target) host date digest”,
signature=”Base64(RSA-SHA256(signing string))”

4.3.2 WS Manifest

This WS returns the essential information about a host. The structure of this service is as follows (the URL is using a UP host as an example):

- Method HTTP: **GET**
- URL path: **/api/manifest/**

In order to make this WS, we created the class core.api.views.manifest.Manifest that follows the singleton pattern [22], this class delegates the functions of the management of the information returned by the WS. Each WS that is supposed to be exposed by the host needs to be signed to the manifest WS using the Manifest object.

The metadata of each WS is structured as a list. It takes as arguments only the data object, which follows the same structure shown on listing Listing 4.2.

```
{
    "name": "string",
    "version": "string",
    "url": "string"
}
```

Listing 4.2: WS metadata scheme.

Institution data is added to the manifest using the format described on Listing 4.3. The primary id to identify institutions is the SCHAC. The other ids are for future uses.

SCHAC code is the top website domain name of an institution. In the case of UP that would be up.pt.

```
"<schacId>": {
    "other_id": {
        "pic": "string",
        "erasmus": "string",
        "euc": "string",
    },
    "name": {
        "en": "string",
        "pt": "string"
    }
}
```

Listing 4.3: Institutions metadata schema.

In Listing 4.4 it is possible to observe the final form of sending, where the objects from Listing 4.2 are expanded on entry "apis_implemented" and the objects from Listing 4.3 are expanded on entry "institutions_covered".

```
{
    "admin_email": ["string"], """Admin email"""
    "admin_notes": "string", """Admin note"""
    "apis_implemented": [...], """Items from code listing 4.1"""
    "institutions_covered": [...], """Items from code listing 4.2"""
    "rsa_public_key": "string", """RSA public key"""
}
```

Listing 4.4: WS Manifest response.

4.3.3 WS Echo

This WS was created based on [26], its one and only function is to receive a HTTP GET request and return the result of the HTTP signature validation (subsection 4.3.1), to be used to test security implementation, the structure is as follows (UP example):

- Method HTTP: **GET**
- URL path: **/api/echo/**

The response format is the same as shown on Listing 4.5.

```
{
    "msg": "boolean",
    "extra": "string"
}
```

Listing 4.5: WS Echo response.

The entry "msg" is a boolean that indicates if the authentication was successful. The "extra" entry shows an error if needed.

4.3.4 Host-Broker interaction

In order to manage the communication of the requests to the Catalog, the class _Catalog was born. This class code can be seen. This class implements the singleton design pattern. In order to get an instance of this class, it is necessary to use the function get_catalog_instance Listing 4.6. This function is responsible for keeping a cached version of the Catalog that is at most 30 minutes old. The function catalog is thread-safe.

```
def get_catalog_instance():
    catalog = _Catalog()
    now = datetime.now()
    diff = (now - catalog.last_fetched).total_seconds()
    if diff > 1800:
        with lock:
            if diff > 1800:
                del Singleton._instances[_Catalog]
                return _Catalog()
    return catalog
```

Listing 4.6: Function to get catalog instance.

The class _Catalog is a wrapper for the retrieved JSON Catalog from subsection 4.2.1. Instead of accessing the dictionary indexes, we access them through an object. The methods are:

- List all schacIds of the network;

- List all schacIds of a host, parameter: schacId of the host;
- Get all WS implemented by a host, parameter: schacId of the host;
- Get the URL (URL) from a WS of a host, parameters: schacId of host and name of the WS;
- List all institutions of the network;
- List all institutions of a host, parameters: schacId of the host.

4.3.5 Host-Host interaction

To be able to communicate data with another host, the use of the security protocol (subsection 4.3.1) is necessary. The function send_unisf_request Listing 4.7 was made to wrap up what is needed for a valid authentication. The parameters for this wrapper are:

- api_uri: URL of the WS obtained through the catalog;
- method: HTTP method of the operation;
- body: body of the HTTP message.

```
def send_unisf_request(api_uri:str, method:str, body=None):
    method_upper = method.upper()
    json_body = b'' if not body else json.dumps(body).encode('utf8')
    headers_and_data = payload.build_payload(f'{api_uri}', method_upper,
        body=json_body)
    return requests.request(method=method_upper, url=api_uri,
        headers=headers_and_data['headers'], data=headers_and_data['data'])
```

Listing 4.7: UNISF request wrapper.

Internally, the method string is sanitized; the JSON body is encoded; the function that builds the authentication payload is called. All are used as input on the request function from python's requests library, basically an HTTP client. The return value is the resolved value of the HTTP call.

The Figure 4.1 shows the system architecture with a numbered sequence showing the communication process. The calls 1 and 2 represent the HOST A asking the BROKER for the HOST B set of WS. The calls 3 and 4 represent the HOST A using HOST B's WS.

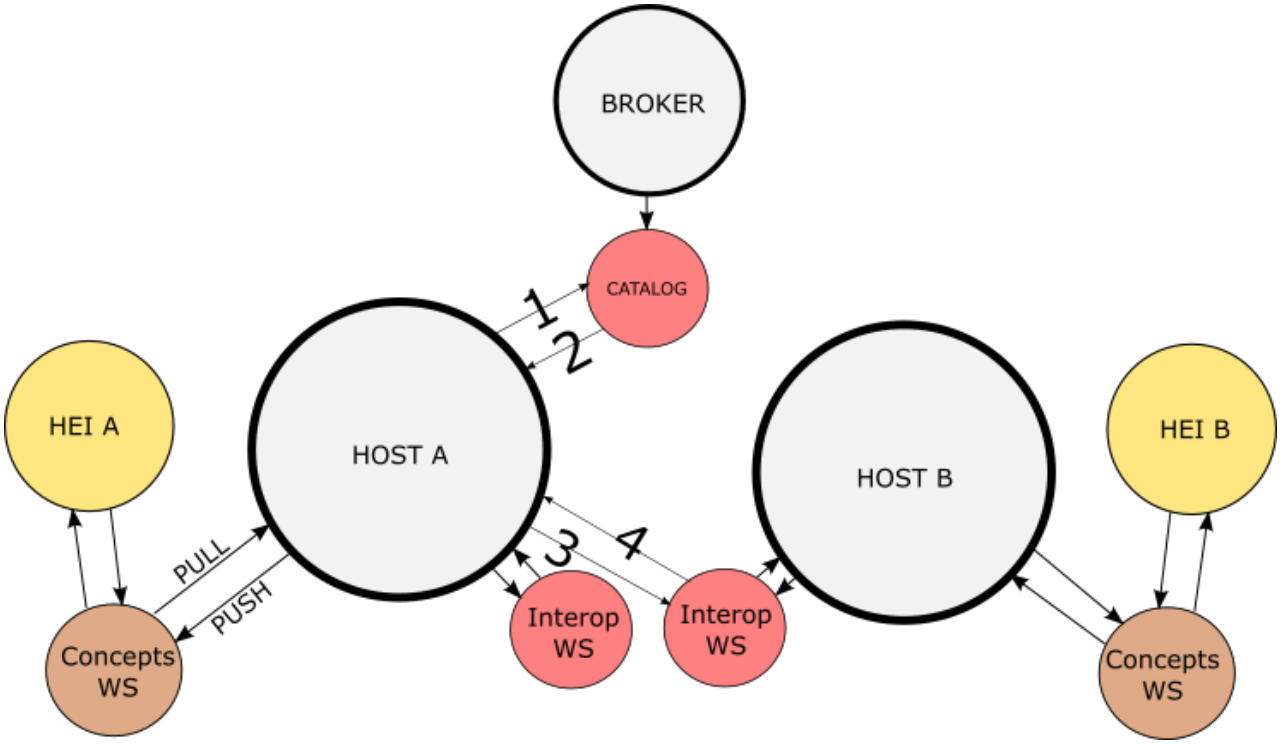


Figure 4.1: System Architecture.

4.4 Chapter Summary

We presented the base of the system (a P2P distributed network), Then we presented each node of the network, their Web Services (WSs) and how they use it to communicate with each other. The first presented node was the Broker node, which is responsible for helping Host nodes to find each other on the network. Then, we presented the Host node, which is the node that will integrate with the HEI's academic information system. Finally, we showed how nodes interact with each other.

Chapter 5

Integration Mechanisms

In this chapter, we present the Host node, showing in more detail its Web Services (WSs) and how it integrates with the Higher Education Institution (HEI) academic information system to support shared courses.

All the WSs for the Host node is available for browsing on a swagger page when running the prototype.

5.1 Overview of the integration mechanism

The integration mechanism described in this chapter will consist of some of the following steps:

1. The Host that wants to share the information knows the Hosts it wants to share to and the course id;
2. The Host with the data proceeds to create a platform agreement (subsection 5.4.1) for the course id and add all Hosts it wants to the partner list;
3. Finalizing the creation of the agreement will trigger an event to create agreement tickets (subsection 5.4.2) on all partners of the agreement;
4. Partners can use the agreement ticket to interact with the agreement. The interaction can be to accept, to refuse, or to ask for a review;
5. After every partner accepts the agreement. The mapping process is triggered. The data (not only the course data but also diplomas and branches related to the course) of the

owner is linked either with a local copy on the partner or existing data;

6. Finally, whenever a change occurs at the owner, it can use a WS to notify that a certain map was changed.

The section 5.5 presents two flow diagrams showing the enumerated processes and the WSs involved. The first one shows the process of agreement creation and approval. The second one is about making changes to mapped information and receiving a notification.

5.2 Collecting academic data (Data push and pull)

Since we can send any data format, the only thing we need to take care of is following the normalized concepts. This thesis will not cover in detail the format of data used by UP Host to share courses, branches, and diplomas. However, we took care to follow the mentioned standards to represent data (section 3.3). This section explains how to integrate the exposed information from a HEI to the platform.

Every HEI needs to have a set of WS of the academic data they want to communicate on the platform. Once they have it, they need to configure the proper repositories on the code. The data inside the platform uses the repository pattern [9]. For this thesis, we will implement data pulling only, but this is the module to attach data push to HEI information system. This section explains how HEIs can integrate their academic data to the platform.

5.2.1 Repository Pattern

The repository pattern [9] is used to decouple the business logic and the data access layers. The data access layer contains information about how to get data from the storage, in our case, from HEI WSs (for UP it will be SIGARRA data). We do not need to know how data persistence works. It is encapsulated.

5.2.2 Managers

The manager is an extra layer of interface between the business and data. It allows us to attach any repository we want and keep the same code structure. The Listing 5.1 is a cut of the courses repository abstract class. For instance, we could easily replace the HEI repository with a mock

```
class CourseRepository(ABC):
    @abstractmethod
    def course_list_all(self, schac, year_start):...
    @abstractmethod
    def course_detail(self, year_start, cce_id):...
    Diplomas and branches go here aswell, shortened on purpose.
```

Listing 5.1: Courses Repository Interface.

one, making tests easy to write. The existing managers should be used, no need to create extensions whatsoever.

A cut of the definition of CourseManager class can be seen on Listing 5.2, the same logic applies for all other entity managers.

```
class CourseManager():
    repository:CourseRepository = None

    def __init__(self, repository) -> None:
        self.repository = repository

    def courses(self, schac, year_start):
        return self.repository.course_list_all(schac, year_start)

    def courseById(self, year_start, cce_id):
        return self.repository.course_detail(year_start, cce_id)
    Diplomas and branches go here aswell, shortened on purpose.
```

Listing 5.2: Courses Manager.

After attaching the implemented repository to an instance of the manager, it is not necessary to make further changes to the code. All are encapsulated and will work as they should.

5.2.3 Academic Data WS

The data is handled on the implementation of repositories. An implemented repository can implements multiple interfaces, as in the example on Listing 5.3. For University of Porto (UP), the SigarraRepository is implementing both Orgunit and Course repositories, which means it

needs to implement every method from the interfaces. Here is where the actual internal WS calls to the HEI existing academic sector happen.

If needed, one can ignore unused parameters, as year_start on both course functions of Listing 5.3. The send_sigarra_request function is SIGARRA logic for the requests, not the concern of the platform.

```
class SigarraRepository(
    OrgunitRepository,
    CourseRepository,
):
    def course_list_all(self, schac, year_start):
        url = f'{SIGARRA_API_BASE_URL}courses/' \
        + '{schac}/{year_start}/{year_end_placeholder}'
        return send_sigarra_request(url)

    def course_detail(self, year_start, cce_id):
        url = f'{SIGARRA_API_BASE_URL}courses/' \
        + '{cce_id}/{year_start}/{year_end_placeholder}/details'
        return send_sigarra_request(url)
    Orgunits, diplomas and branches were cut for presentation purposes
```

Listing 5.3: SIGARRA Repository.

On the Listing 5.3, the UP Host will not send all its information through the platform, only those that are bounded to an agreement. SIGARRA Repository is not directly exposed. When a Host asks for course data from others, it first checks the existence of agreements. If none is found, it returns nothing, so the agreement acts as a filter.

The repository responsible for gathering the shared data information is the UNISF Repository class. It is active when Host partners invoke academic data WSs.

The Figure 5.1 shows a class diagram of the used repository pattern.

5.3 Mapping System

The Mapping System is the heart of the platform. Here is where the external data is linked with the desired internal data and where the notification system works. This section is the lowest level

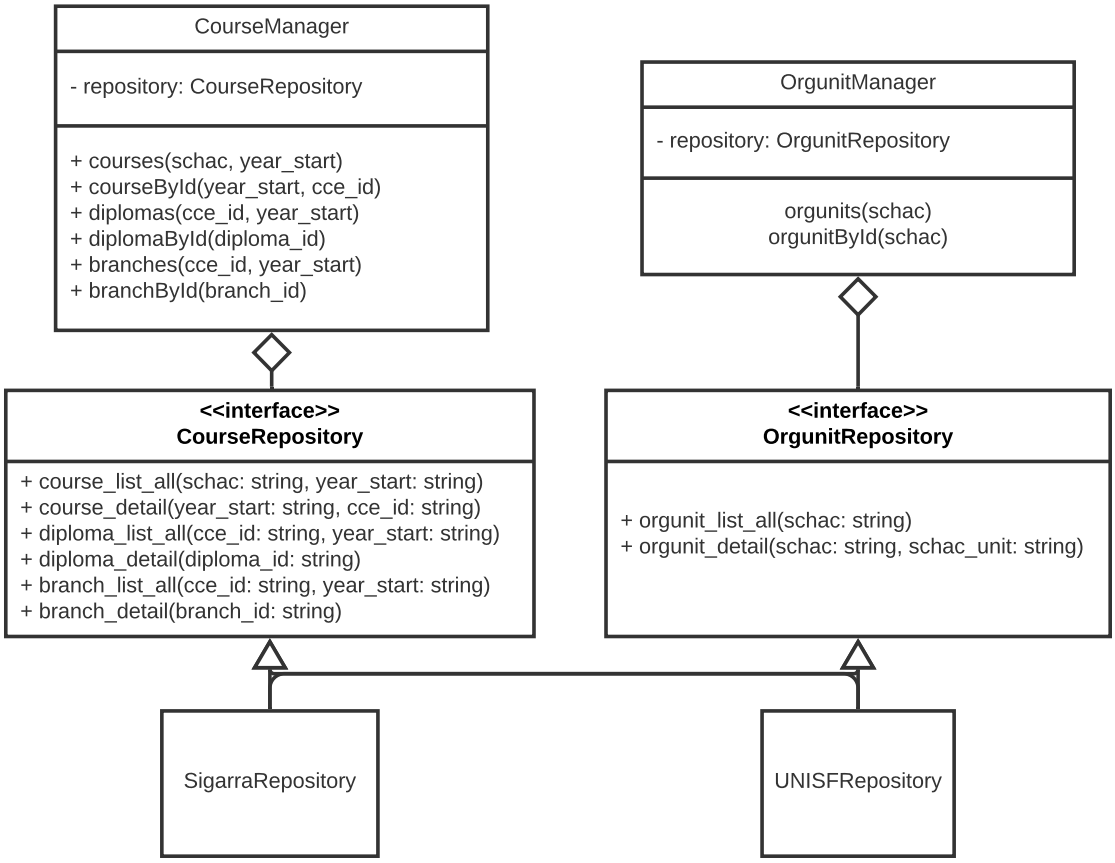


Figure 5.1: Class diagram of the repository pattern.

of data sharing. When dealing with the prototype, no endpoint here will be manually triggered unless needed. It follows the observer pattern [22] but adapted to a distributed system.

5.3.1 Observer Pattern

This pattern is a software design pattern in which an object keeps a list of observers and notifies them whenever a state change occurs.

5.3.1.1 Our adaptation of observer pattern

The observers will be created by the partners for every map object created. They need to carry sufficient information so the owner Host can find their maps later on. Whenever a change occurs, the Host uses the internal id of the changed object to get the list of observers from the database and trigger a notification on the target (partner).

5.3.2 Mapping data

The Mapping process uses four models: Mapping; MapGroup; Observer, and Notification. The system was build considering that a Map has only one administration Host. Suppose the Host that is not the original owner of the data (partner) makes some editing on its local copy of the shared information. In that case, the Host owner of the original information will not receive any updates.

The data mapping is straightforward. We take the type of data that we will link, the external id of the data we want, the owner, and the internal id of our copy of the data. It has a field map group to group related maps (like grouping the diplomas, branches, and the course map). The external id and the owner are sufficient information to find the external object. However, for the owner Host to know that we have access to that information, we need to pass the ids of all parents of that object so that the owner can reconstruct the path to the shared course id. We do that by using the extra_args entry. The format of extra_args is a dictionary, and the entries are of the format ”parent_depth”, where ”depth” is the depth of parenthood, Listing 5.4 shows a diploma with an extra_args pointing to the courses object using ”parent_0”.

The Mapping model is described on Table 5.1.

Mapping Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
map_group		Yes	foreign_key
unisf_type			type of the object linked
internal_id			the id of the internal object
external_id			the id of the external object
owner			the SCHAC of the data owner
extra_args			ids of object parents

Table 5.1: Mapping Model

The MapGroup model carries only the primary key field. It is used for group-related mappings. The Mapping object can be created either by WS or using the prototype platform. A Mapping object is automatically created after the agreement state changes to ready. Any Host can create it manually if it has access to the data on the partner side (in order to have this access, an agreement is still needed).

The WS for Mapping creation is called Data Link, the structure is as follows (UP example):

- Method HTTP: **POST**
- URL path: **/data_link**
- Body: check Listing 5.4

```
{
    "id": 75,
    "unisf_type": "diplomas",
    "internal_id": "4b92f631-5981-4d39-a965-7e2d4808d780",
    "external_id": "3622",
    "owner": "up.pt",
    "extra_args": {
      "parent_0": "2181/2020/2021"
    },
    "map_group": 1
  },
```

Listing 5.4: WS Data Link body example.

The success response format is a JSON of the Mapping object with HTTP status 201 (Object created).

5.3.3 Setup Notifications

The implemented notification system was deeply inspired by Erasmus Without Paper (EWP) Change Notification Receiver. API [29]

In order to set up the notifications for a map, the Host that created the map uses the owner field of the Mapping object to get from the catalog the owner WS to set up an observer on the owner's side. After getting the WS URL from the catalog, the Host with the Mapping packs an Observer object that carries sufficient information to find the Mapping on the partner's side that wants to be notified.

5.3.3.1 Observer

The Observer object is structured as on Table 5.2.

Observer Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
unisf_type			type of the object linked
internal_id			the id of the internal object
observer			the SCHAC of the data observer

Table 5.2: Observer Model

The WS for Observer creation is called Notification Setup, the structure is as follows (UP example):

- Method HTTP: **POST**
- URL path: **/api/notification_setup**
- Body: check Listing 5.5.

```
{
    "observer": "up.pt",
    "internal_id": "456",
    "unisf_type": "course"
}
```

Listing 5.5: WS Notification Setup body example.

In this case, the internal_id is the internal id for the owner of the data. The success response format is a JSON of the Observer object with status 201 (Object created).

5.3.4 Notify

When the observed data is changed, the Host that owns the data can use the WS Notify to create an Notification object on the observer Host side. the structure of this service is as follows (UP example):

- Method HTTP: **POST**
- URL path: **/api/notify/**

- Body: check Listing 5.6

```
{
   "internal_id": "456",
   "unisf_type": "diplomas"
}
```

Listing 5.6: WS Notify body example.

The internal id used here also takes the owner of the data as a reference. This information is only needed to locate the Mapping object on the observer Host and attach a Notification object. The success response is status 200 with no body.

The SCHAC of the information owner is also needed to identify the map object. It is passed as a header. So it does not need to go in the message body.

The Notification object has the structure shown on the Table 5.3.

Notification Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
mapping		Yes	mapping foreign key
solved			boolean flag, if the notification was dismissed

Table 5.3: Notification Model

When opening a Mapping object through the prototype interfaces, all notifications will be dismissed for presentations purposes. However, it does not mean it must be handled this way.

5.3.5 UpdateRequest

Since we are neither changing the administrative HEI nor allowing non-administrative to change global information directly, this endpoint will allow non-administrative partners to ask for an information change. It can be either existing or completely new information (inserting new students).

- Method HTTP: **POST**
- URL path: **/api/update_request/**

- Body: check Listing 5.7

```
{   "external_id": "123",
    "internal_id": null,
    "unisf_type": "student",
    "extra_args": {
        "parent_0": "2183/2020/2021"
    },
    "object_data": {
        "student_id": 910813429,
        "person_id": 146975426,
        "csc_id": 146972436,
        "name": "Student Name",
        "type": "N",
        "first_registration_year": "2014",
        "first_ies_registration_year": "1469726"
    }
}
```

Listing 5.7: WS UpdateRequest body example.

If the "external_id" is specified, then the administrative HEI knows it is a notify scenario. If the "external_id" is null, then it needs to use the **Update** WS (subsection 5.3.6) and pass the "internal_id" to trigger a map on every non-administrative partner and complete the map for the request issuer.

UpdateRequestModel Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
reviewed			alread reviewed
origin			request origin
internal_id			the id of the internal object
external_id			the id of the external object
unisf_type			type of the object linked
extra_args			ids of object parents
object_data			the data to be changed

Table 5.4: UpdateRequestModel Model

The entries ”external_id”, ”unisf_type” and ”object_data”, are meant for object identification. The object data is the new desired object to be broadcasted for the agreement.

This endpoint creates an object on the administrative side for later data analysis. The model for this object is described on Table 5.4.

5.3.6 Update

When instead of changed, new data is added to the observed course, this endpoint allows the owner to create a new map on partners for the new entry. the structure of this service is as follows (UP example):

- Method HTTP: **POST**
- URL path: **/api/update/**
- Body: check Listing 5.8

```
{   "proxy": none,
    "proxy_object_id": none,
    "id": "123",
    "unisf_type": "student",
    "extra_args": {
        "parent_0": "1295/2020/2021"
    }
}
```

Listing 5.8: WS Update body example.

In order to create the map, we need to know to which set of maps it will go. That is why the extra_args is necessary.

The SCHAC of the information owner is also needed to identify the map object. However, as stated before, we can get it from the headers.

If the update was issued due to an **UpdateRequest**, the ”proxy” and ”proxy_object_id” need to be used so that the partner that started the update request can link to its created map with null external id. As an example, a non-administrative agreement partner could insert a new

student to its database, create a map with null ”external_id”, use the UpdateRequest, passing its ”internal_id”, WS, and finally would be able to link the **Update** WS external id with its created map. The administrative side would pass the non-administrative SCHAC as ”proxy”, and the non-administrative object id as ”proxy_object_id”.

An example of the use of this endpoint would be adding a new course unit to a course. The information is not edited because it was not there before.

5.3.7 UML Model Diagram

The UML Model diagram for the mapping system showing all relationships can be seen at Figure 5.2. The relation between Mapping and Observer shown is symbolic since it happens in different nodes.

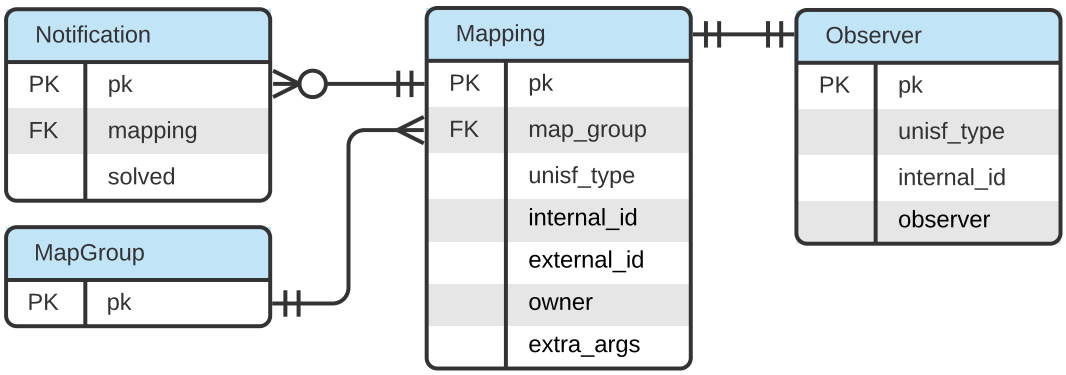


Figure 5.2: UML Model diagram for the mapping system.

5.4 Agreement System

The main difference between our agreement system and that of EWP [27] is that we have a HEI that is the owner of the information and acts as an admin. The Hosts on the agreement are not on the same level of importance as on [27].

The Agreement System will use every section we have seen so far. All the processes described will be automatized. We will only have to concern about the agreement. It is the highest level of data sharing. The implemented agreement system was inspired by EWP agreements API [27] and the agreements approval API [28].

5.4.1 Agreements

An Agreement object carries information about the top-level data that will be shared and the list of all partners related to it. This information tells the Host which of the partners are allowed to get information.

The one that starts the process of agreement creation is the Host that wants to share the data. It attaches a course id and a list of SCHAC from all Hosts it wants to share with. The creation process triggers an event that creates AgreementTicket objects on all partners listed. They will later use it to interact with the Agreement object.

The act of saving the Agreement object to the database also creates an AgreementTicket for the agreement owner.

The Agreement object can be seen on Table 5.5.

Agreement Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
course_reference			reference of the shared object

Table 5.5: Agreement Model

The agreement also has a pseudo attribute that is status. It is calculated every time it is asked. If one of the partners rejects the agreement, it changes to REJECTED. Suppose one of the partners asks for an agreement revision. In that case, it will create a Complain object (subsection 5.4.2). If the number of not solved complaints is greater than zero, the status is WAITING_FOR_REVIEW. If all partners accept the agreement, it changes to ACCEPTED.

For each agreement listed partner, it is also created a Partner object on the Agreement object owner. It keeps track of the partners' interactions with the Agreement object. The Partner object structure can be seen on Table 5.6.

The status of the partner depends on the interaction with the agreement. It can be PENDING when the agreement is waiting for interaction. APPROVED if the partner accepts the agreement. WAITING if the partner asked for some review related to the data attached to the agreement, and REJECTED if the partners reject.

Partner Model			
Attribute Name	Primary key	Foreign key	Description
pk	Yes		primary_key
agreement		Yes	reference of the agreement
schac			partner identifier
status			the agreement interaction

Table 5.6: Partner Model

5.4.1.1 Agreement WSs

Here are the WSs available for the agreement part and a brief description of what they do:

5.4.1.1.1 Agreement Create : is used by a Host admin to create an agreement. It only takes the course reference that will be shared and the list of partners to share with.

- Method HTTP: **POST**
- URL path: **/unisf/api/agreement**
- Body: check Listing 5.9 showing an example of agreement creation, sharing the course with reference "1203/2021", between UP and the dummy university.

```
{  "course_reference": "1203/2021",
   "partners": [
       "university.int"
   ]
}
```

Listing 5.9: WS Agreement creation body example.

The successful result would be the agreement object with the partners objects embed, like Listing 5.10. One can check that the UP was included in the partners set and will have its ticket as well.

5.4.1.1.2 Agreement List : is used by a Host admin to get the list of all agreements it created.

```
{   "pk": 1,
    "partner_set": [
      {
        "id": 1,
        "schac": "university.int",
        "status": "PENDING",
        "agreement": 1
      },
      {
        "id": 2,
        "schac": "up.pt",
        "status": "PENDING",
        "agreement": 1
      }
    ],
    "course_reference": "1203/2021",
    "status": "NOT READY"
  }
```


Listing 5.10: Agreement object in JSON.

5.4.1.1.3 Agreement Detail : is used by Hosts (owners and partners of the agreement) to get the details of an agreement.

5.4.1.1.4 Agreement Delete : used by a Host admin to delete an agreement, all shared bonds will cease after the agreement deletion.

5.4.1.1.5 Complain List : is used by a Host admin to get all complaints related to a particular agreement.

5.4.1.1.6 Complain Detail : is used by a Host admin to get the details of a complaint.

5.4.1.1.7 Agreement Accept : used by partners to accept an agreement. If all Hosts use this WS, it will trigger the mapping event described on section 5.3.

5.4.1.1.8 Agreement Reject : used by partners to reject an agreement. If at least one Host uses this WS, it will invalidate the agreement.

5.4.1.1.9 Agreement Review : used by partners to review an agreement, it takes a JSON as the body with the description of why it should be reviewed.

When using a choice WS like, **Agreement Accept** (paragraph 5.4.1.1.7), **Agreement Reject** (paragraph 5.4.1.1.8) or **Agreement Review** (paragraph 5.4.1.1.9), the system will always check if the agreement status changed and in case it did change, it will notify all interested partners using the wrapper notify_partners of listing Listing 5.11. The notify partner will use the Agreement Ticket Update WS to change the state of partners' tickets.This function will make automatically use of the update agreement ticket WS paragraph 5.4.2.1.5.

```
def notify_partners(agreement_pk, partners_list, new_status, *args, **kwargs):
    for partner in partners_list:
        notify_partner(agreement_pk, partner, new_status, *args, **kwargs)
```

Listing 5.11: Notify Partners wrapper.

5.4.2 Agreement Tickets

This Agreement Tickets were inspired by EWP agreement approval API [28].

The Agreement Ticket is a piece of information that is created on all partners related to an agreement. This information helps them to locate and fetch the Agreement object on the owner's side and interact with the agreement's WSs.

5.4.2.1 Agreement Tickets WSs

5.4.2.1.1 Agreement Ticket Creation : is automatically used by the system to create an agreement ticket object on all partners that are bounded to an agreement. This way, partners are aware of new contracts.

- Method HTTP: **POST**
- URL path: **/api/agreement/ticket**

- Body: check Listing 5.12 showing an example of how UP Host would automatically create the agreement ticket on the dummy university's side.

```
{  "origin": "up.pt",
   "reference": "1"
}
```

Listing 5.12: WS Agreement ticket creation body example.

The successful result would be the agreement object with the partners' objects embed, like Listing 5.13. One can check that the UP was included in the partners set and will have its ticket as well.

```
{ "pk": 1,
  "origin": "up.pt",
  "reference": "1",
  "choice": "PENDING",
  "status": "NOT READY"
}
```

Listing 5.13: Agreement ticket object in JSON.

5.4.2.1.2 Agreement Ticket List : is used by a Host admin to list all tickets it has.

5.4.2.1.3 Agreement Ticket Detail : is used by a Host admin to see the details of a given ticket.

5.4.2.1.4 Agreement Ticket Deletion : is used by a Host admin to delete the information of a ticket.

5.4.2.1.5 Agreement Ticket Update : is used by the system when the agreement status changed. The agreement owner Host will update the partner's agreement ticket with the new status.

- Method HTTP: **POST**

- URL path: **/api/agreement/ticket/update/**
- Body: check Listing 5.14 showing an example of how UP Host would update the agreement ticket on UP side.

```
{   "agreement": "2",
    "origin": "up.pt",
    "new_status": "READY"
}
```

Listing 5.14: WS Agreement ticket update body example.

5.4.2.2 Agreement Interactions

After the agreement owner updates the ticket of every partner with the status of READY, every partner will trigger the automatic mapping event, which will use the view function link_unisf_data from AgreementTicketUpdate to start the mapping process.

The mapping process of link_unisf_data follows the Command Design Pattern [22]. It starts by initializing the CourseMapManager that will recursively prepare a batch of Commands. For this thesis, MapCoursesCommand, MapBranchCommand, and MapDiplomaCommand. The CourseMapManager calls all its children managers, in this case, BranchMapManager and DiplomaMapManager. This Command batch is then fed to the CommandScheduler, where all Commands will be executed. The CommandScheduler keeps a list of all executed Commands and backtracks in case of failure.

The UML Class Diagram for the Command system can be seen on Figure 5.3. Each Command will automatically map information as described on section 5.3.

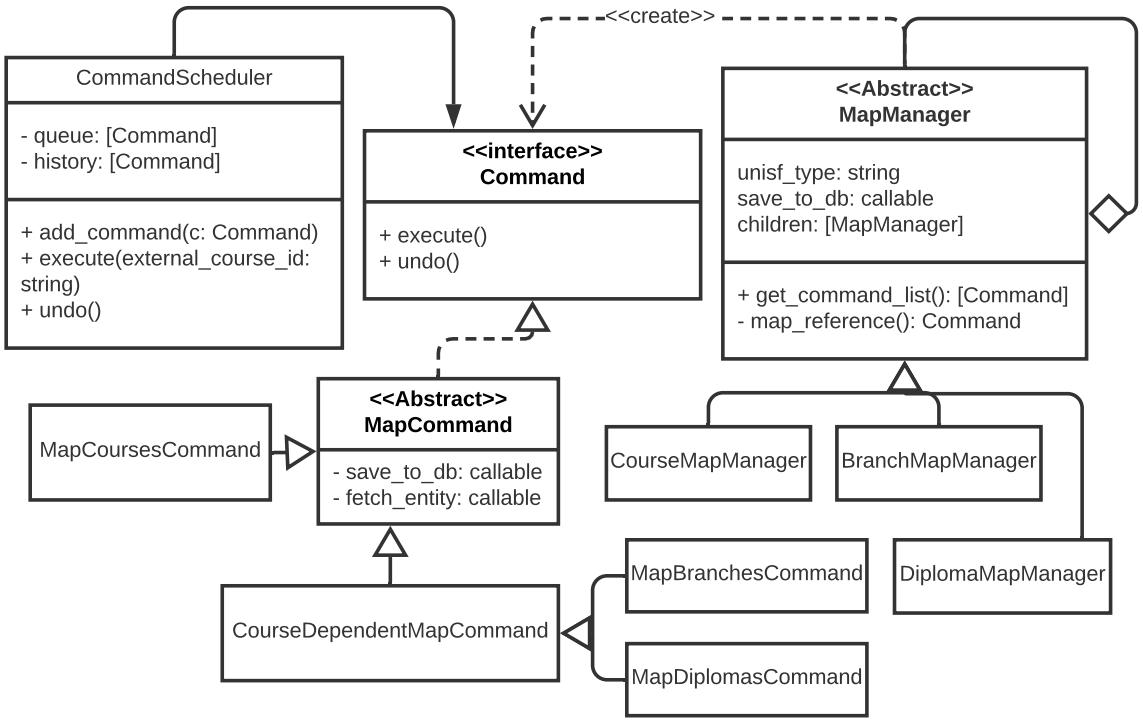


Figure 5.3: UML Class diagram for the Command system.

5.4.3 UML Model Diagram

The UML class diagram for the agreement system showing all relationships can be seen in figure Figure 5.4. The relation between agreement and agreement ticket shown is symbolic since it happens in different nodes.

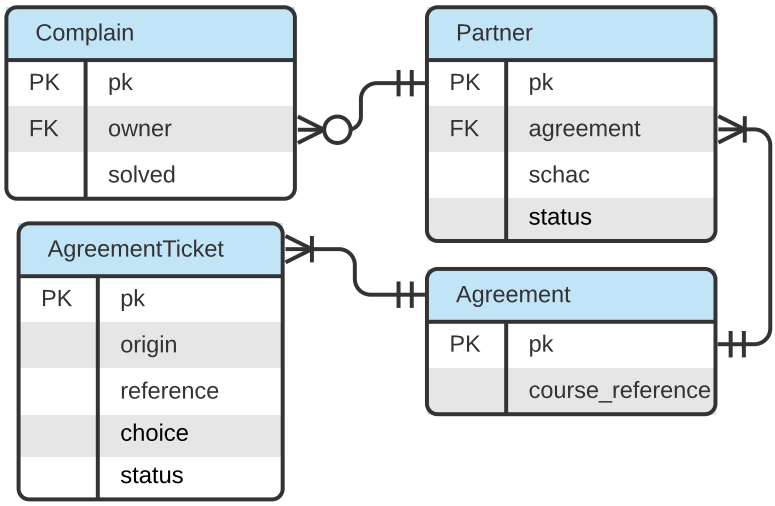


Figure 5.4: UML Model diagram for the agreement system.

5.5 Asynchronous notification flow

The Figure 5.5 shows the asynchronous notification process between HEI A and HEI B. From the figure, we can see that our notification system works like EWP's Change Notification Receiver (CNR) WSs. HEI A uses the Notify WS and HEI B will asynchronously act. The process of course creation can be checked in the appendix Figure D.1.

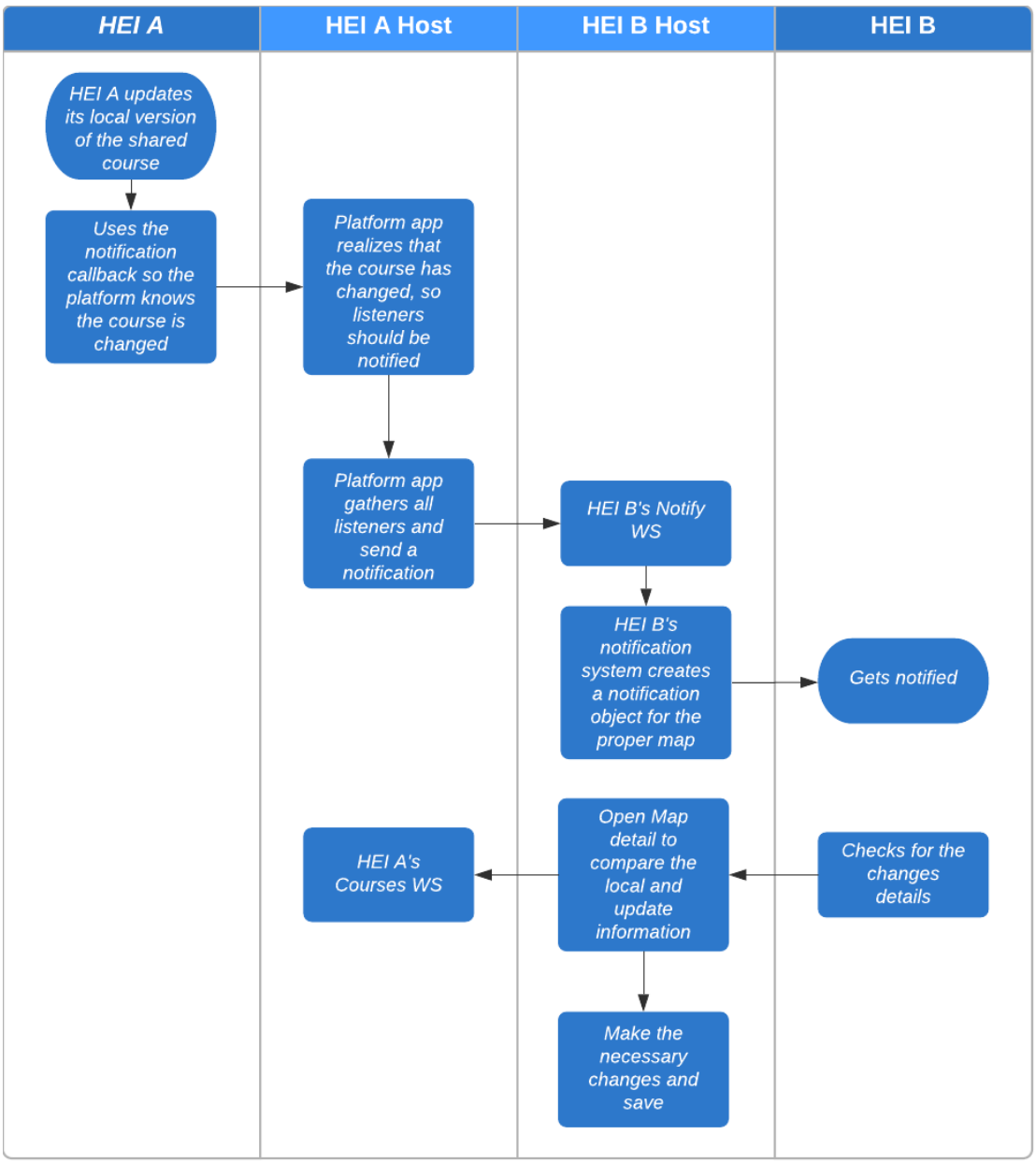


Figure 5.5: Editing and notification process flow chart.

The Figure 5.6 shows the same process in a node-oriented way.

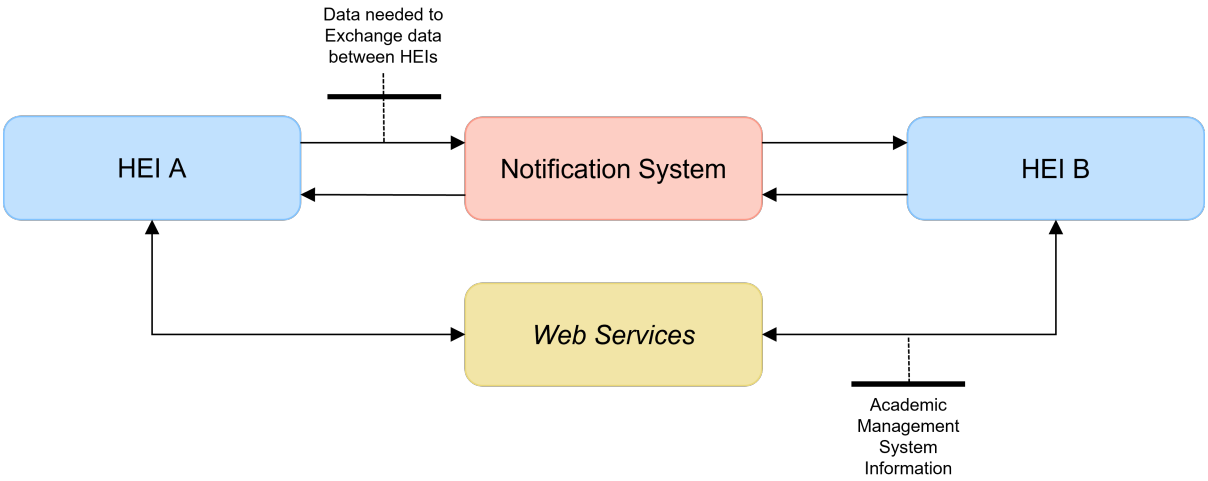


Figure 5.6: Editing and notification process.

5.6 Chapter Summary

We presented an overview of the steps for data integration. Then, we showed how to pull and push academic data from the HEIs' information systems. After that, we presented the information mapping system responsible for referencing external data and receiving change notifications (using the observer pattern adapted to distributed systems). Finally, we presented the Agreement System, which is responsible for selecting what HEIs will be able to fetch the data from a given shared course, closing with a graphical explanation of the notification system.

Chapter 6

Use cases and Prototype

This chapter will show a use case of how data would be shared between two actors and then show the flow they would follow through the prototype's screens.

6.1 Use cases

Regarding the academic business process, here are some use cases that serve as motivation to the thesis. For the cases, we have two actors, the course information owner (the administrative HEI) and the partner HEI that wants the information.

Actor 1: Administrative HEI

Use Cases | Course creation:

- **Collect data**: administrative HEI will collect legal and technical information related to course creation;
- **Create course**: administrative HEI will create the course using the gathered information;
- **Create course units**: administrative HEI create the new course units that are necessary to create the study plan;
- **Create study plan**: administrative HEI creates the study plan and associate all related course units;
- **Create HEI agreement**: administrative HEI uses the platform, in our case the prototype, to create the HEI agreement that will start the academic information share process.

Use Cases | Course information editing:

- **Search for the shared information locally**: administrative HEI uses its system to search a course;
- **Visualize the search course information**: administrative HEI visualize the course information;
- **Make some editing**: administrative HEI makes some changes to the course information;
- **Submit changed information**: administrative HEI submits the information, and this action triggers the notification on the partner's map.

Actor 2: Partner HEI

Use Cases | Course creation:

- **Visualize the HEI agreement**: partner HEI is notified that a new agreement is ready and waiting for approval;
- **Visualize agreement's attached information**: partner HEI may visualize the shared information to confirm the validity of the information.
- **Accept the agreement**: the partner HEI finally accepts the agreement.

Use Cases | Course information editing:

- **Verify the changed information**: partner HEI verify that a map has the notification flag on and verify the changes;
- **Compare the local mapped information with the remote**: the partner HEI compares the existing copy of the information with the newest;
- **Update the information**: partner HEI can update the information into its own system.

In the following section, we will show the business processes of how the University of Porto (UP) Host would share the course "2181" with a dummy university Host as we go through the screens of the prototype.

6.2 Prototype

We will now, in this section, present the prototype for the platform showing the interfaces for the Broker and a Host (for this matter is an example of implementation of Host for UP). Also, as we go, we show how the information sharing process would unravel through the screens. This demonstration will highlight the Hosts of UP and the dummy university. The data from UP is coming from a development SIGARRA database, and the dummy university data is stored in memory. In order to differentiate the Hosts, UP Host will have the navigation bar as black, and the UP logo, and the dummy university host will have the navigation bar as green.

6.2.1 Broker

In this subsection, we will be presenting all the screens for the Broker prototype. Remember, this is not a final version. Some interfaces were made this way to focus on functionality and ease the development process.

6.2.1.1 Landing

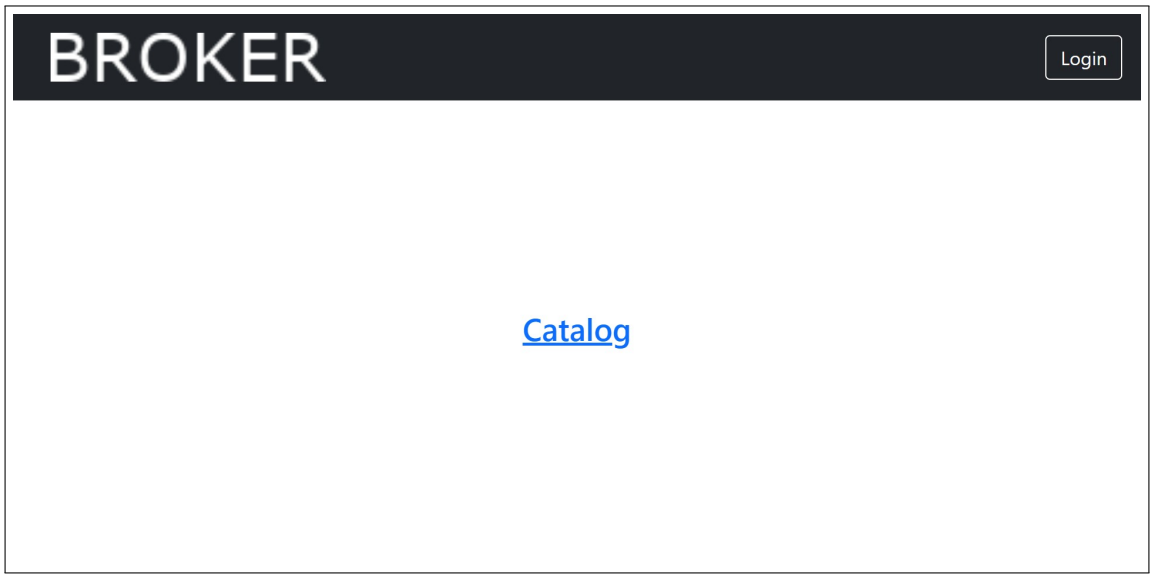


Figure 6.1: Broker landing page.

The broker landing page has a button to check the JSON for Host catalog (the same from Listing 4.1), a log-in button, and a button to add new Hosts. As shown on Figure 6.1.

6.2.1.2 Login

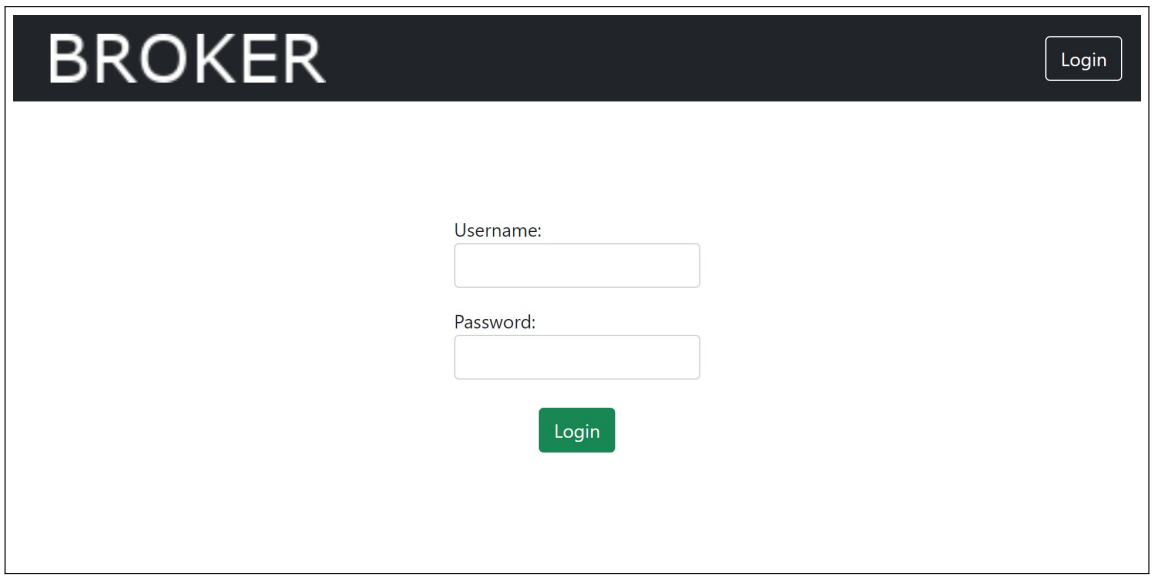


Figure 6.2: Broker login page.

The login page is simple, as shown on Figure 6.2.

6.2.1.3 Host Managing

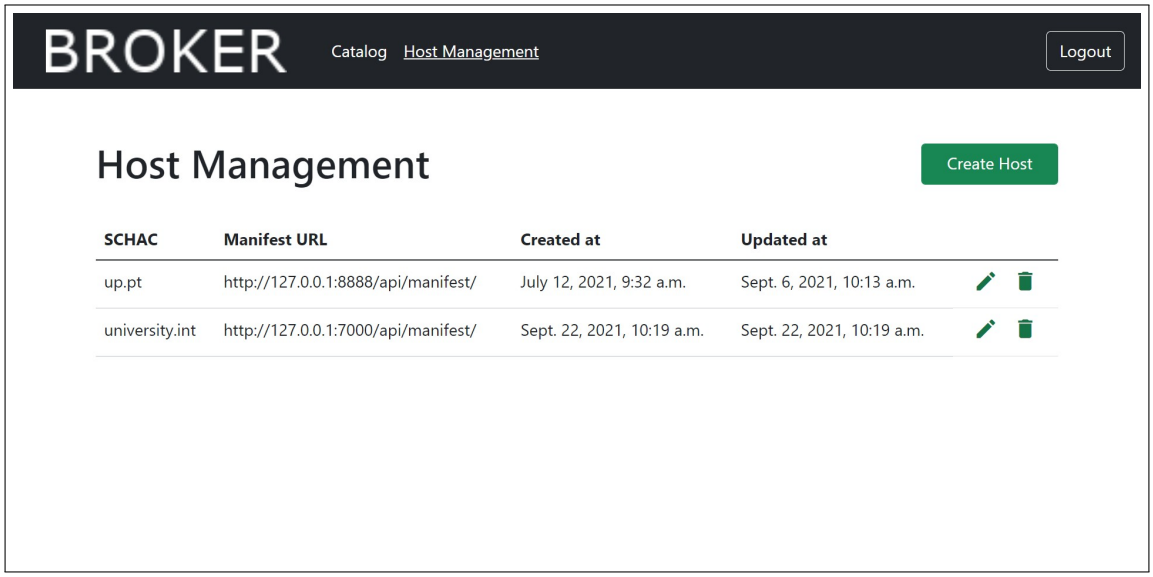


Figure 6.3: Broker Host Management page.

The Host Managing page allows the broker admin to add, edit and delete Hosts at will. The screen can be seen on Figure 6.3.

Figure 6.4 shows the form that servers as Host create and update, in this case, is being used to edit UP Host registry information.

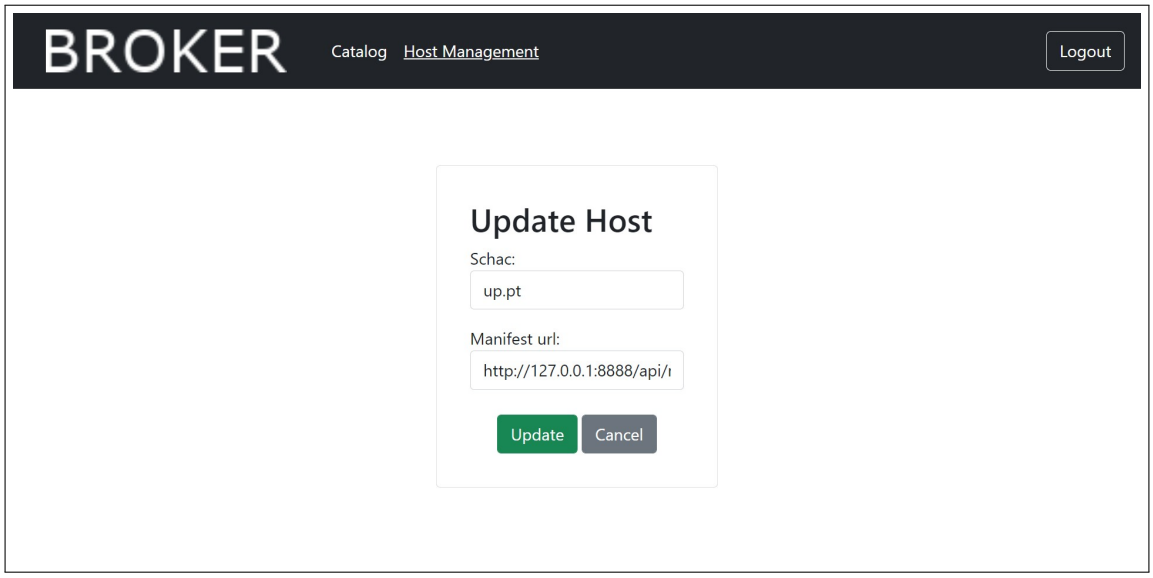


Figure 6.4: Broker Host create/update page.

6.2.2 Host

In this subsection, we will be presenting all the screens for the Host prototype, featuring how a UP Host would be. Remember, this is not a final version. Some interfaces were made this way to focus on functionality and ease the development process. Each Host can customize to their will or even not use it. The presented prototype is just an interface to use the Web Services (WSs) and can be used as a boilerplate code to make new Hosts for a Higher Education Institution (HEI). We will also show the screens in an order that follows the data sharing process.

6.2.2.1 Landing

The landing page for the Host shows the path to two different menus, the UNISF system and the internal information system. The UNISF area is where the platform WS will get in action. The internal system area was made to show how editing internal data would trigger notifications. On the top bar, there is a button to go to the swagger page, where the user can find information about the platform WS. The page can be seen on Figure 6.5.

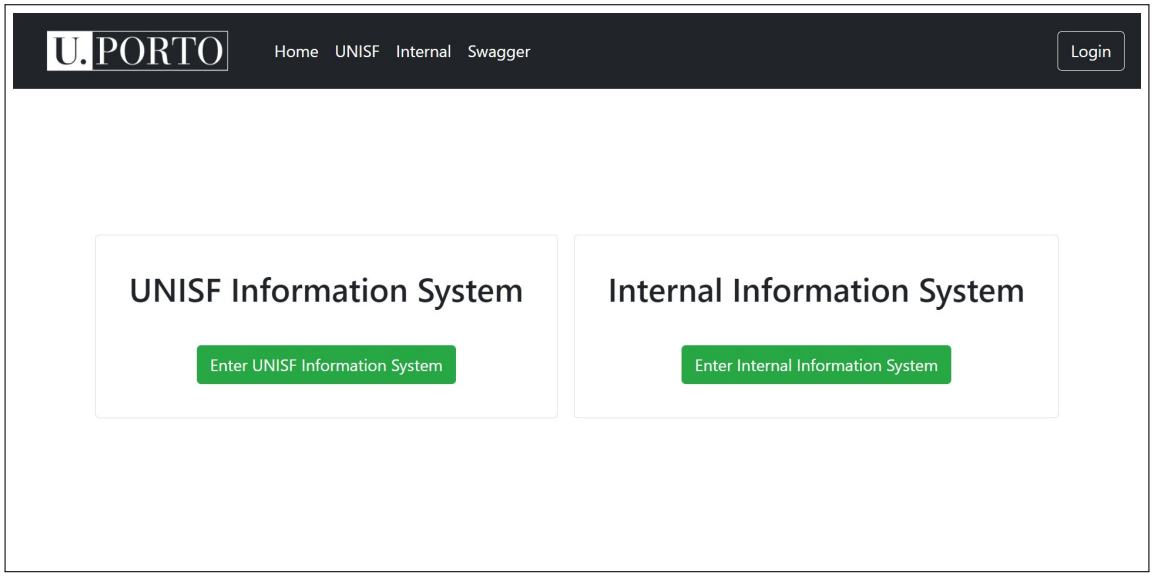


Figure 6.5: UP Host landing page.

6.2.2.2 UNISF menu

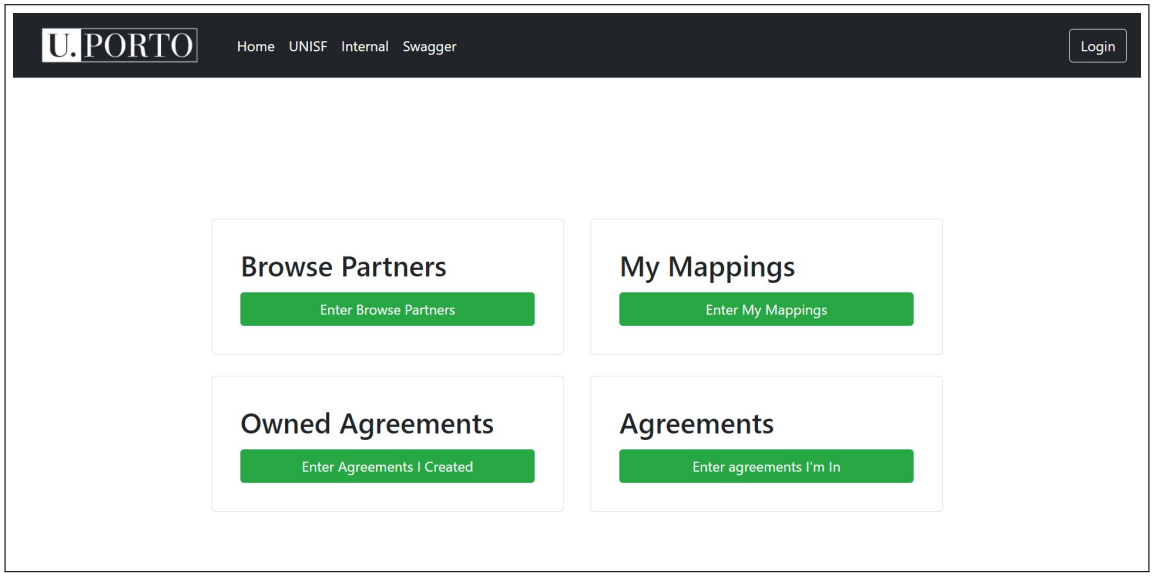


Figure 6.6: UP Host UNISF menu page.

From the UNISF menu page, it is possible to see: the list of partners connected to the Broker (Browse Partners subsubsection 6.2.2.3); the mapping menu, where one can manage the mappings (My Mappings subsubsection 6.2.2.6); the list and management of agreements created by the Host (Owned agreements subsubsection 6.2.2.4); and the list and interaction menu for agreement tickets of all agreements the Host is enrolled (Agreements subsubsection 6.2.2.5). These four paths can be seen on Figure 6.6.

6.2.2.3 Browse Partners

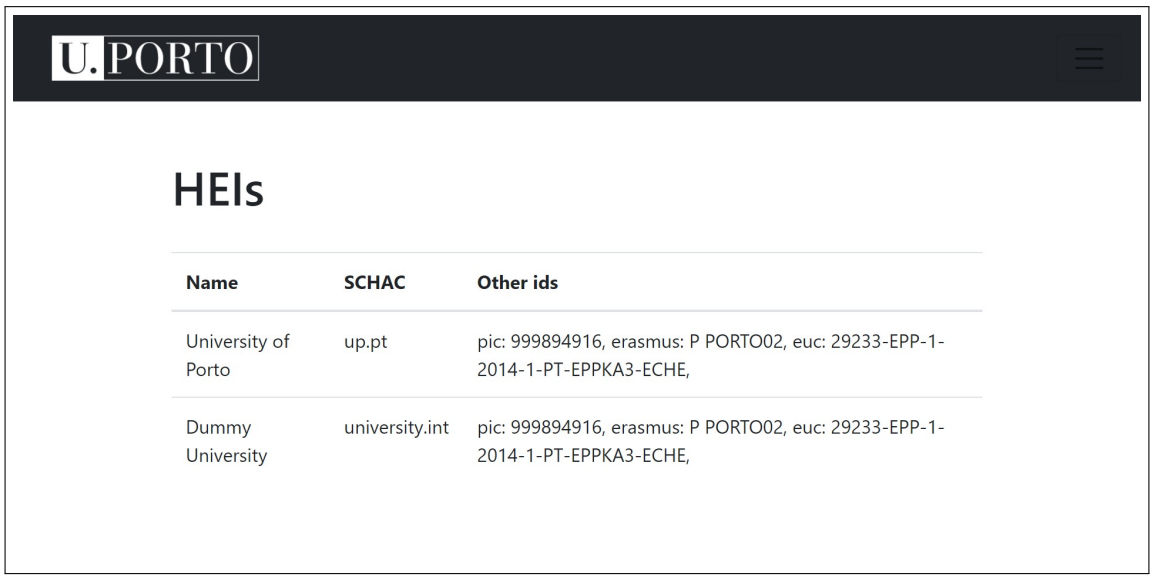


Figure 6.7: UP Host UNISF partners menu page.

This menu shows the list of all partners connected to the Broker. The Figure 6.7 shows UP and the dummy university as registered partners. The ”Other ids” column is dummy data.

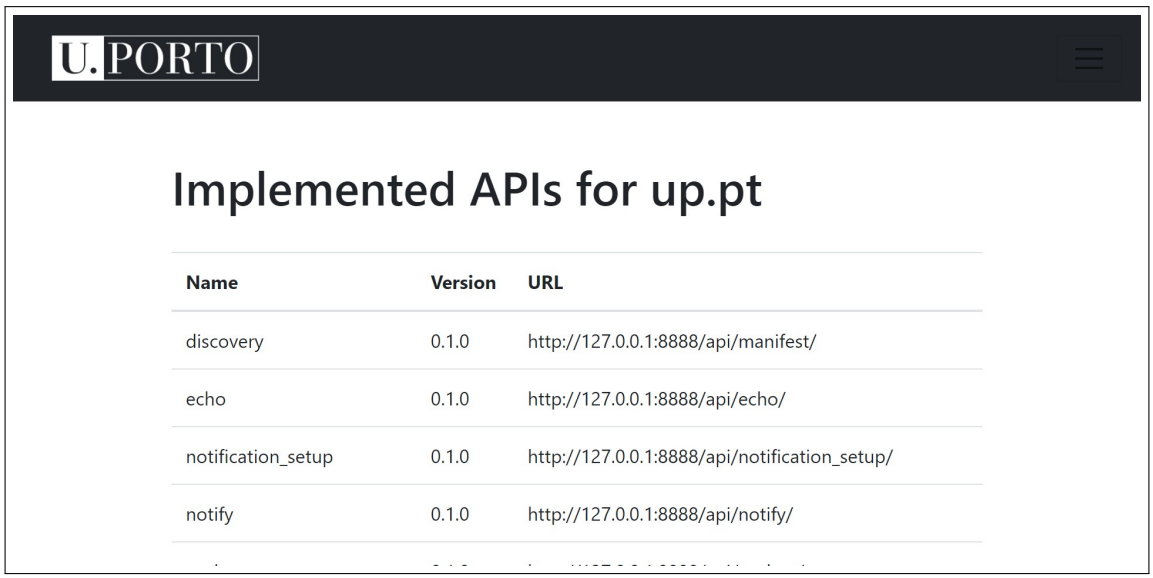


Figure 6.8: Page showing all implemented WSs for UP.

When picking any partner from the list, the result page will be a list of implemented WS, the result for UP can be seen on Figure 6.8. From here, only ”echo”, ”orgunits”, and ”courses” have User Interface (UI) showing data, ”echo” will display if the security implementation is sound. The courses WS will return information as long as it is used with a Host you have an agreement with.

6.2.2.4 Owned Agreements

Owned Agreements is the page that shows all agreements created by, in this case, UP Host, check Figure 6.9. It shows an agreement with the status ”NOT READY” that was created for the course reference ”2181/2020/2021” and has the dummy university as an agreement partner. The creation page can be seen on the Figure 6.10.

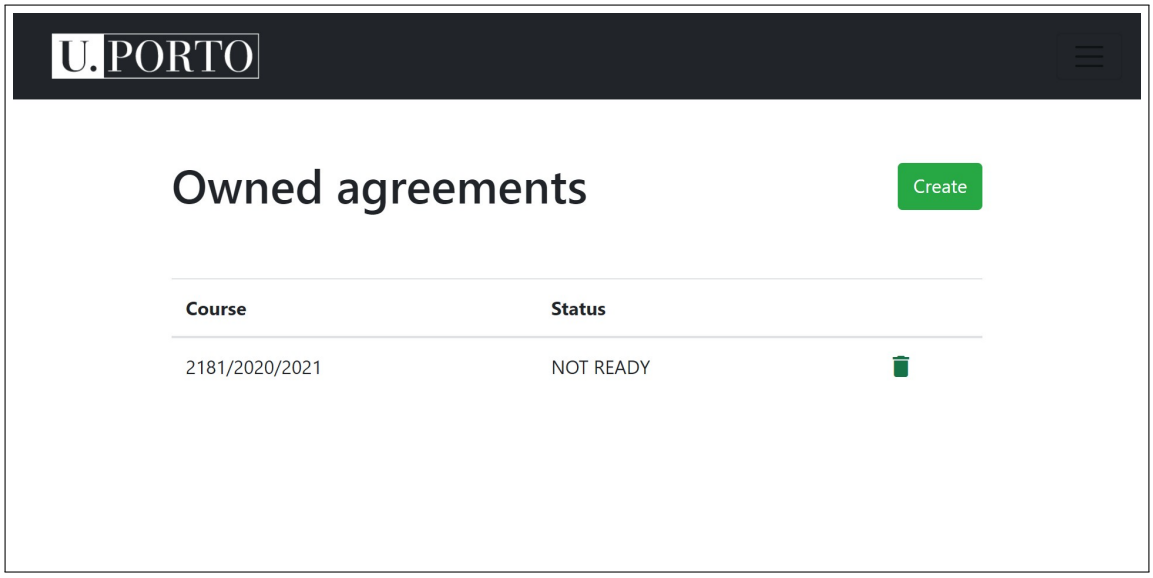


Figure 6.9: UP owned agreements list.

Figure 6.10: UP owned agreements creation page.

6.2.2.5 Agreements I am in

Since we have an agreement created, this agreement tickets page will show the ticket that is related to that agreement. The ticket is created on the partners' and the owner's sides. The UP ticket can be seen on Figure 6.11.

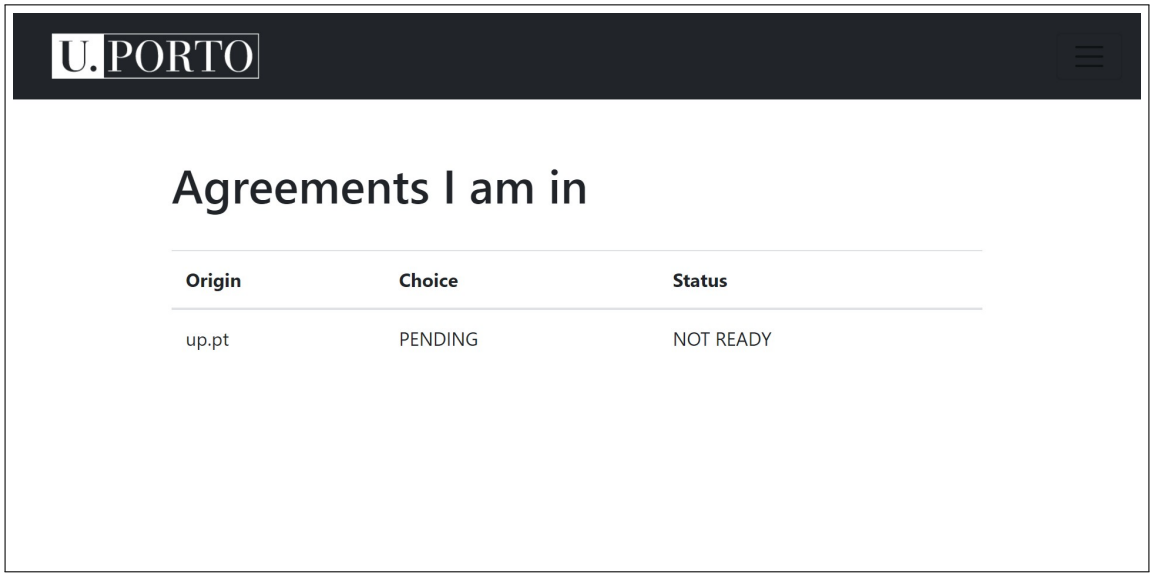


Figure 6.11: UP agreement tickets page.

After selecting a ticket, one can interact with it using the options available on the page shown on Figure 6.12. The dummy university Host has a similar ticket. When both accept, the information will be mapped on the dummy university's side.

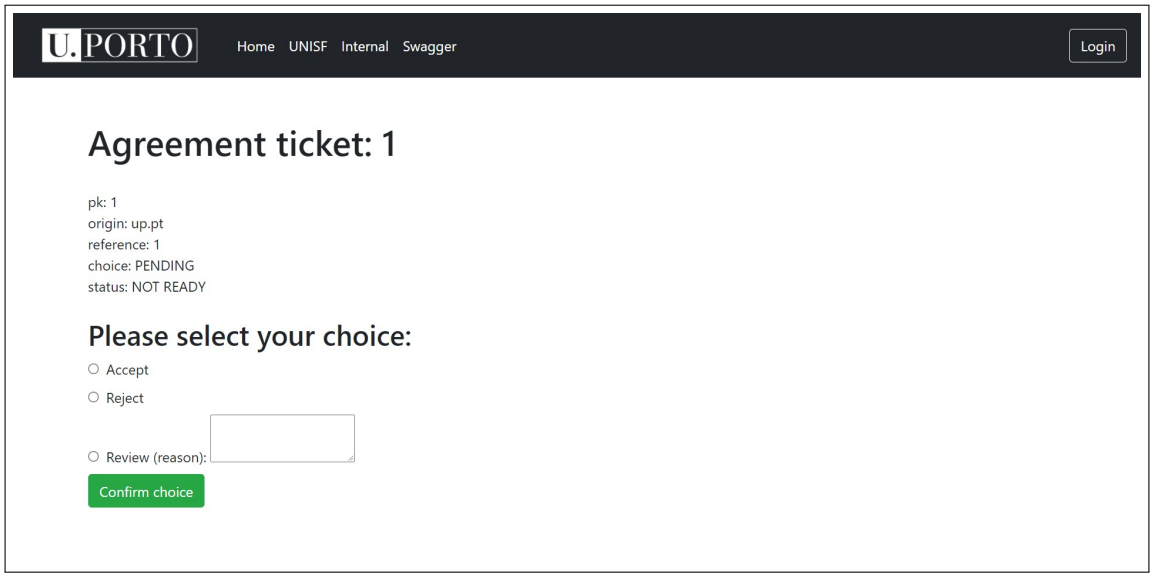


Figure 6.12: UP agreement ticket interaction page.

6.2.2.6 My Mappings

After every ticket related to the agreement is accepted, the mapping page on the dummy university's side, Figure 6.13, will show the mapped information. It automatically mapped the course's branches and diplomas and the course data.

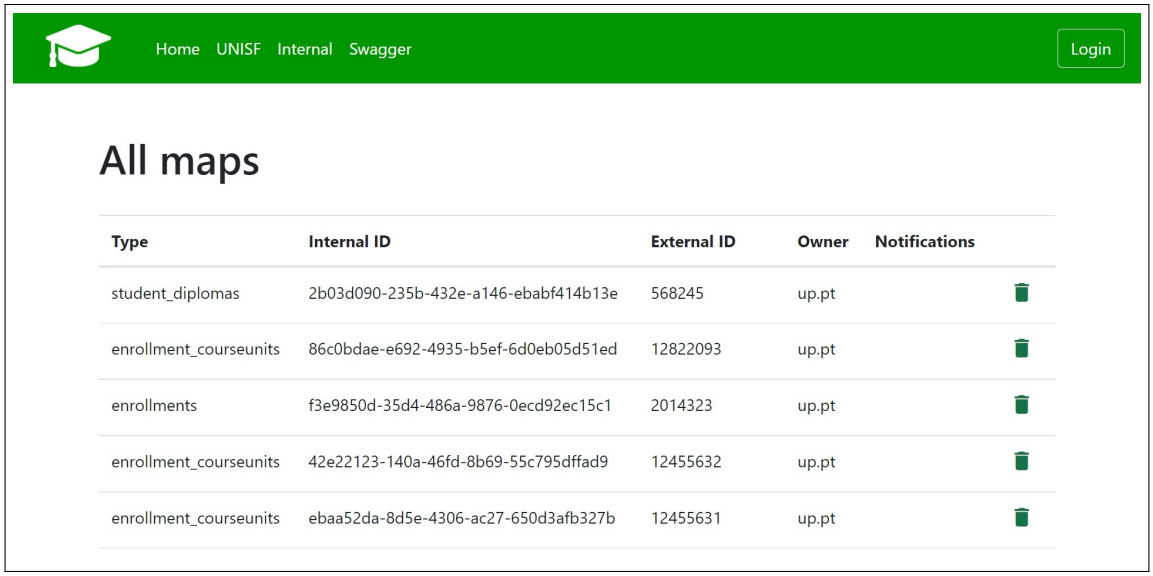


Figure 6.13: Dummy university map list page showing the course 2181 mapped.

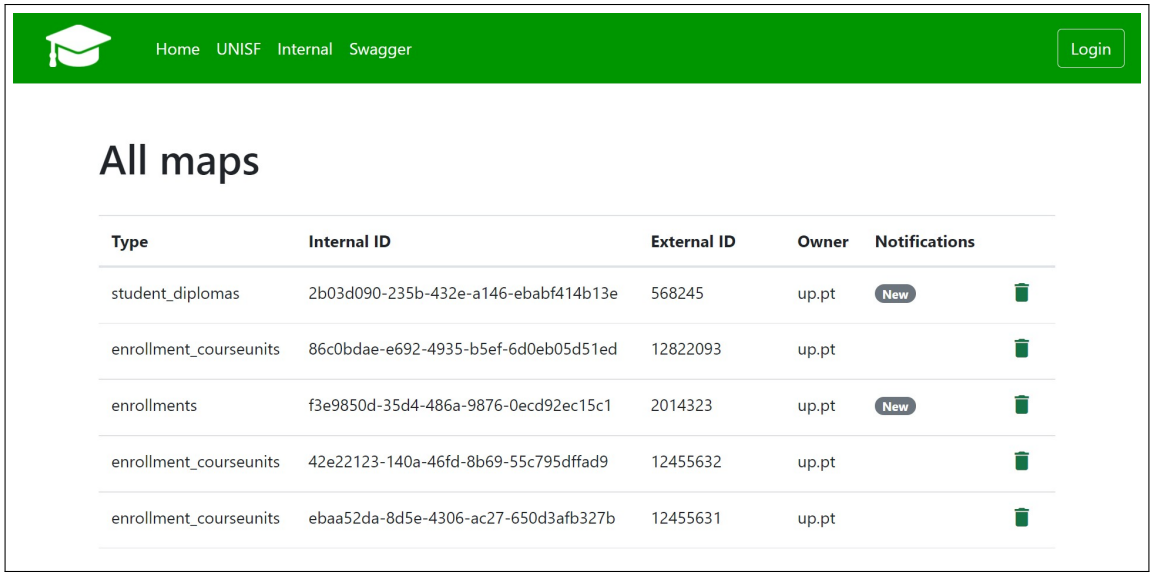


Figure 6.14: Dummy university map list page showing a notification badge on the mapped student_diploma and enrollment objects.

Suppose the owner of the data, UP, modify the information and use the notification callback, which is the case for our embedded internal system. In that case, a notification badge will

appear on the ”Notifications” column. The list with notifications can be seen on Figure 6.14

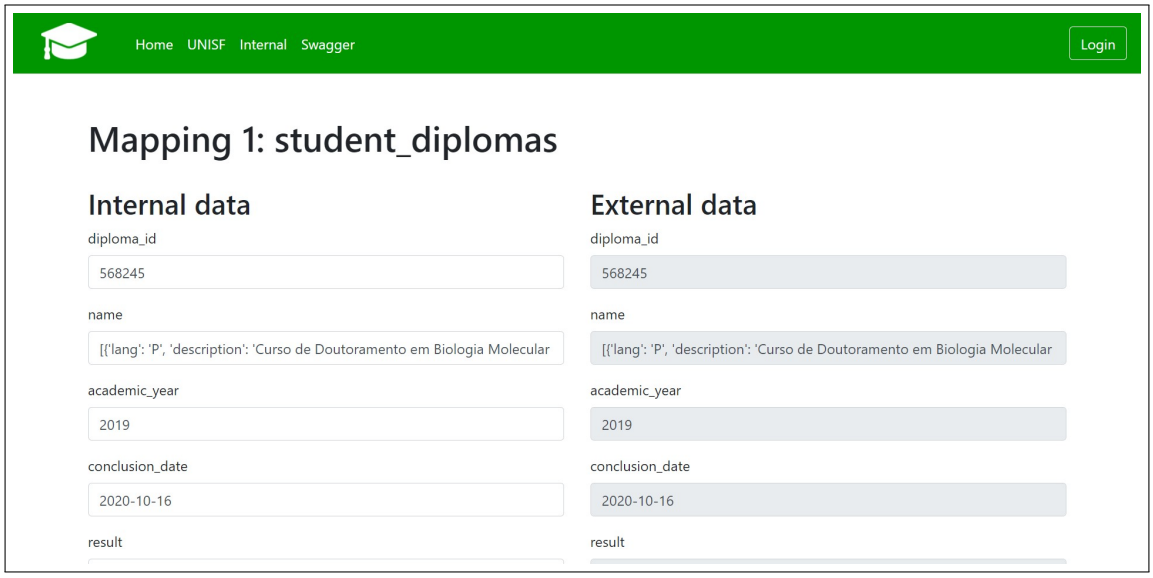


Figure 6.15: The dummy university map detail showing the local and origin (latest) version.

In a notification, one can see the details by clicking on a notified map and comparing the local version of the information with the latest version from the origin (External data). As shown on Figure 6.15, data can be updated from this page by clicking on the Update button at the end of the form. For this prototype, the action of opening the map details will dismiss all notifications for that map. Any Host can and is encouraged to deal with notifications the way they want to.

6.2.2.7 Internal System menu

This internal menu, as said, was built to trigger notifications events. The idea is to embed the notifications WS to the HEI existing internal system. Figure 6.16 shows the entry page for the courses browser.

Inside the shown courses menu, it is possible to search for the course. In this particular case of course ”2181/2020/2021”, this shared course is from FCUP. The course search page can be seen on Figure 6.17.

After selecting the course ”2181” from the list, we reach the edit mode, Figure 6.18. It is possible to trigger the notification event by clicking on the update button at the end of the form.

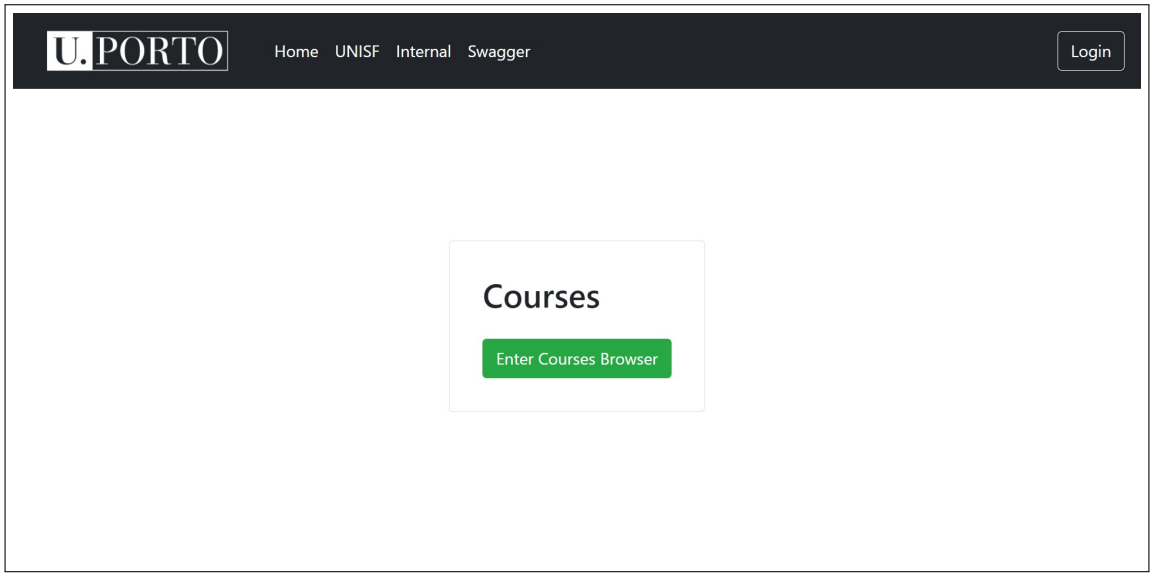


Figure 6.16: UP internal system menu.

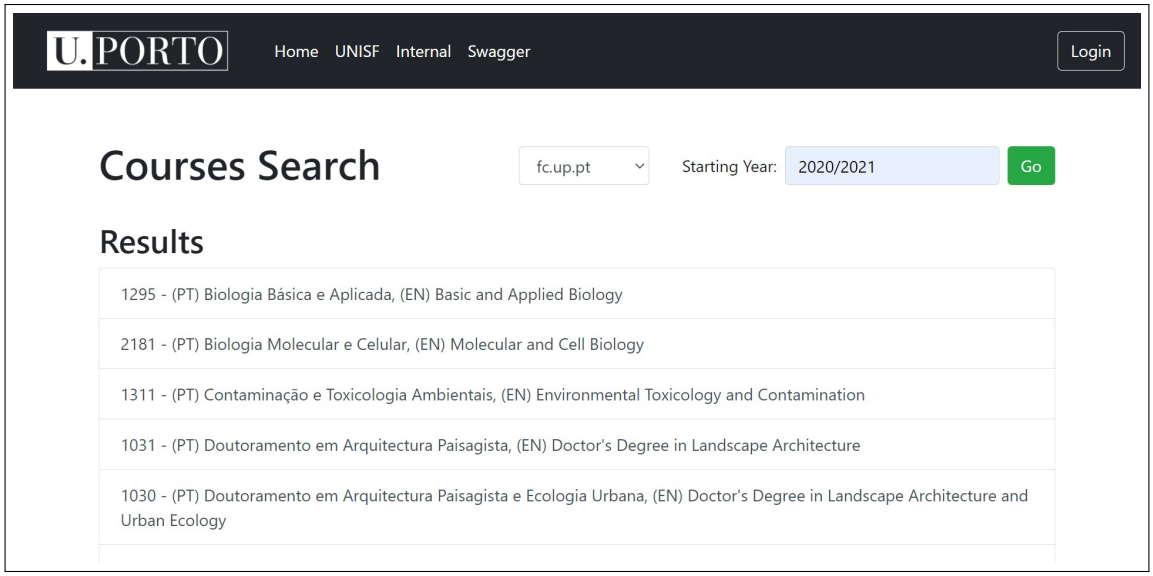


Figure 6.17: UP internal system search menu showing courses for FCUP in 2020/2021

6.2.3 Sharing process wrap up

Here is a wrap up of the business processes showed on the prototype screens:

Course creation and sharing

- UP has the course 2181 created on its internal system and wants to share with the dummy university;
- UP uses the owned agreement page Figure 6.9 in its UNISF menu Figure 6.6 to reach the

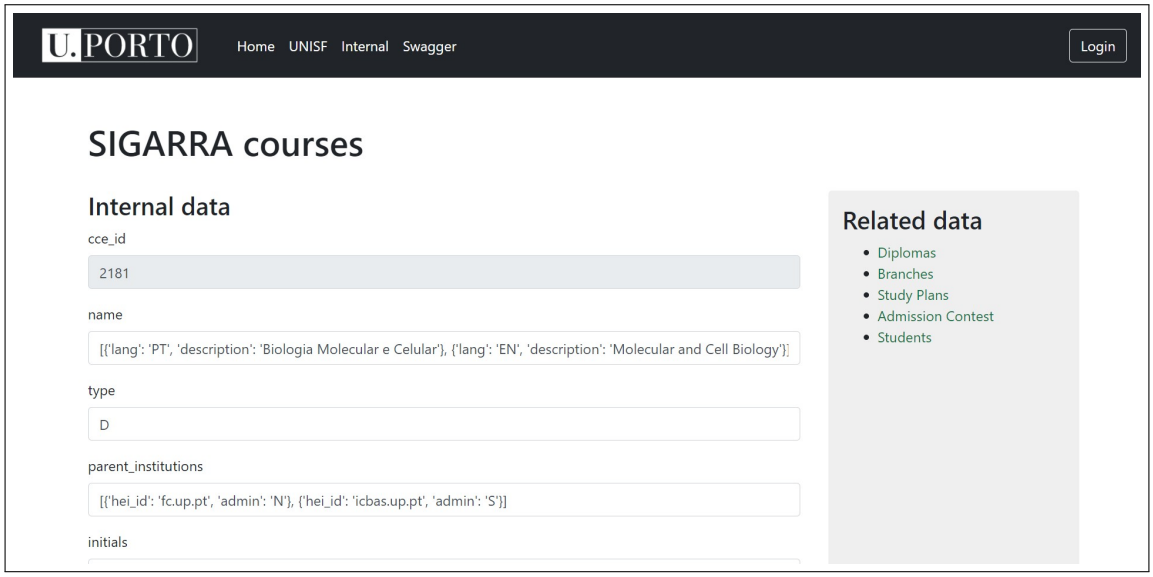


Figure 6.18: UP internal system editing course 2181.

agreement creation form Figure 6.10;

- Figure 6.9 shows the newly created agreement;
- both UP and the dummy university need to accept the agreement, to do that they reach the tickets list Figure 6.11 and get the ticket detail Figure 6.12;
- Figure 6.13 shows the newly mapped information that is ready to receive notifications.

Course editing and notifications

- UP uses the SIGARRA simulation on internal system area Figure 6.16, opens the course search page Figure 6.17 and search for the course 2181;
- UP opens the details of the course Figure 6.18;
- UP make some changes to the course and commit, the system will trigger a notification event and the dummy university will receive this notification;
- The dummy university opens the map menu and can check that some maps now have a notification badge Figure 6.14.

6.3 Configuring a test environment

Setting up a test environment is easy. Just follow these steps:

1. Clone, the Broker code, once and create a super admin account;
2. Run the docker-compose file for the Broker;
3. Run the broker server;
4. Clone the Host code as much as necessary to set up the institution nodes;
5. For each Host, generate an RSA key pair, provide the SCHAC, run the docker-compose file, and fill some generic Host settings.
6. Use the Broker to register all Hosts into the network;
7. Pick one of the hosts to be the data provider and connect the repositories objects with the desired institution WSs;
8. Now, the system is ready to share the course.

After these steps, the environment is ready. Use the platform on the configured data provider to start sharing the courses. All code is hosted on the UP private repository. Consult the Appendix A to access the material.

6.4 Chapter Summary

We presented the use case of how the course is shared, and the notifications are issued. Then, we presented the prototype, showing how the presented use case acts on each prototype screen. Finally, we showed how to set up a testing environment.

Chapter 7

Conclusions and future work

This chapter will describe the development and the achievements made through this dissertation by presenting an overview of the development and comment on the test results. Finally, we close with some future work.

7.1 Development

This dissertation described the development of a system to help the Higher Education Institution (HEI) information systems to support the processes. To do so, we studied the best practices of existing integration systems that are validated on Europe, Erasmus Without Paper (EWP) and EMREX, and support student mobility.

The problem we wanted to solve was the communication of data, safely and canonically, between different HEI information systems in order to have the processes of shared courses flowing. Shared courses are not student mobility, but we could still manage to reuse the base of the EWP system as a framework for our project, so we had a secure communication channel for free.

For the process of copy, link and keep the data updated between different HEIs, we decided to prototype a version of shared courses considering one administrative HEI that acts as a master, writing information, and the non-administrative acting like slaves, reading only.

7.2 Results

We tested the system using a host for University of Porto (UP) sharing a course with a dummy university Host. Only the UP was connected to a database to provide information. The dummy node was set to save the copied information to memory to be notified and compare exchanged information.

For this test scenario, we achieved positive results. The master host of UP was able to send all the notifications successfully, and the slave dummy university Host was able to notice what data was changed and confirm the new information before the push to its information system.

We stated that we wanted minimal changes for the users of the HEI information systems. All processes of the nature of having a course could be handled using UP information system, and the dummy host received all the updates.

7.3 Future Work

As mentioned on subsubsection 3.5.2.1, the notification system flows from the administrative host to the non-administrative ones, which means it is a one-way notification. So, as a suggestion for future work, we could implement a way to change the administrative host of an agreement so there can be an administrative rotation.

The mapping system is suited to support unique artificial keys. In a future version, it could support multiple column keys if necessary.

Another critical issue is that the platform was not tested using actual data on both sides of the communication. The documents produced by the UP academic management team were not validated by the other HEIs involved in the project. So we had to test the sharing process with dummy host nodes. Our update requests tests were made manually. That is why the canonical data obligation was removed when testing. A future version could include tests with real partners.

Bibliography

[1] Donald E. Eastlake 3rd. Additional XML Security Uniform Resource Identifiers (URIs). RFC 6931, April 2013. doi:10.17487/RFC6931.

[2] Anthony Bryan. Additional Hash Algorithms for HTTP Instance Digests. RFC 5843, April 2010. doi:10.17487/RFC5843.

[3] M. Cavage and M. Sporny. Signing http messages. Internet-Draft draft-cavage-http-signatures-07, IETF Secretariat, July 2017.

[4] MyAcademicID consortium. Myacademicid blueprint architecture [visited 2021].

[5] Universidade do Porto Digital. Ciclos de estudo em associação. Unpublished internal report, June 2013.

[6] Erasmus+. Part a: General information about the erasmus+ programme [visited 2021].

[7] European Commission. Directorate General for Education and Culture. *ECTS users' guide 2015.* Publications Office. doi:10.2766/87192.

[8] Emrex User Group. Emrex | supporting student mobility, 2015 [visited 2021].

[9] Bob Gregory Harry Percival. *Architecture Patterns with Python*. O'Reilly UK Ltd., April 2020. ISBN: 1492052205.

[10] Michał Kurzydłowski and Wojciech Rygielski. Ewp mobility process explained, [visited 2021].

[11] Michał Kurzydłowski and Wojciech Rygielski. Institutions api, [visited 2021].

[12] Arnaud Lauret. *The Design of Web APIs*. Manning Publications, November 2019. ISBN: 1617295108.

[13] Schmidt Michael and Ziegler Jule Anna. An identity provider as a service platform for the edugain research and education community. pages 739–740. IEEE, 2019. ISBN: 978-1-7281-0618-2.

[14] Janina Mincer-Daszkiewicz. Emrex and ewp offering complementary digital services in the higher education area. Technical report, University of Warsaw, 2017.

[15] Janina Mincer-Daszkiewicz and W. Rygielski. Emrex in poland supporting internal mobility. 2016.

[16] Janina Mincer-Daszkiewicz and W. Rygielski. Erasmus without paper - from the technical perspective. 2016.

[17] Jeffrey Mogul and Arthur van Hoff. Instance Digests in HTTP. RFC 3230. doi:10.17487/RFC3230.

[18] Carla Oliveira and Marco Nunes. Conceitos teóricos académicos da universidade do porto (updigital ed.). Unpublished internal report, April 2021.

[19] Carla Oliveira and Marco Nunes. SERVIÇOS DISPONÍVEIS POR ÁREA DE NEGÓCIO (updigital ed.). Unpublished internal report, April 2021.

[20] Ariel Ortiz. Web development with python and django (abstract only). In Laurie A. Smith King, David R. Musicant, Tracy Camp, and Paul T. Tymann, editors, *Proceedings of the 43rd ACM technical symposium on Computer science education, SIGCSE 2012, Raleigh, NC, USA, February 29 - March 3, 2012*, page 686. ACM. doi:10.1145/2157136.2157353.

[21] Robin G. Qiu. *Business-Oriented Enterprise Integration for Organizational Agility*. Business Science Reference, 2013. ISBN: 1466639105.

[22] Alexander Shvets. *Dive Into DESIGN PATTERNS*. Not published yet, 2021.

[23] Michael Stollberg. *Scalable Semantic Web Service Discovery*. Südwestdeutscher Verlag für Hochschulschriften AG Co. KG, 2015. ISBN: 383810451X.

[24] UNISF team. O PROJETO: UNIVERSIDADE SEM FRONTEIRAS (UNISF) [visited 2021].

[25] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Architecture and common datatypes, 2016 [visited 2021].

[26] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Echo api, 2016 [visited 2021].

[27] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Interinstitutional agreements api, 2017 [visited 2021].

[28] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Interinstitutional agreements approval api, 2017 [visited 2021].

[29] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Broadcasting changes (”cnr” apis), 2017 [visited 2021].

[30] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Schac identifiers, 2017 [visited 2021].

[31] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Authenticating clients with http signature, 2017 [visited 2021].

[32] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Ewp address data types, 2017 [visited 2021].

[33] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Ewp phone number types, 2017 [visited 2021].

[34] Michał Kurzydłowski Wojciech Rygielski and Marta Juzepczuk. Ewp academic term data types, 2017 [visited 2021].

Appendix A

Source Code

All the code used on the project can be found at the University of Porto (UP) private repository:

- Host nodes at: https://git.uporto.pt/di/si-cursos-interop.
- Broker node at: https://git.uporto.pt/di/si-cursos-interop-broker.

Appendix B

Erasmus Without Paper (EWP) and EM-REX processes

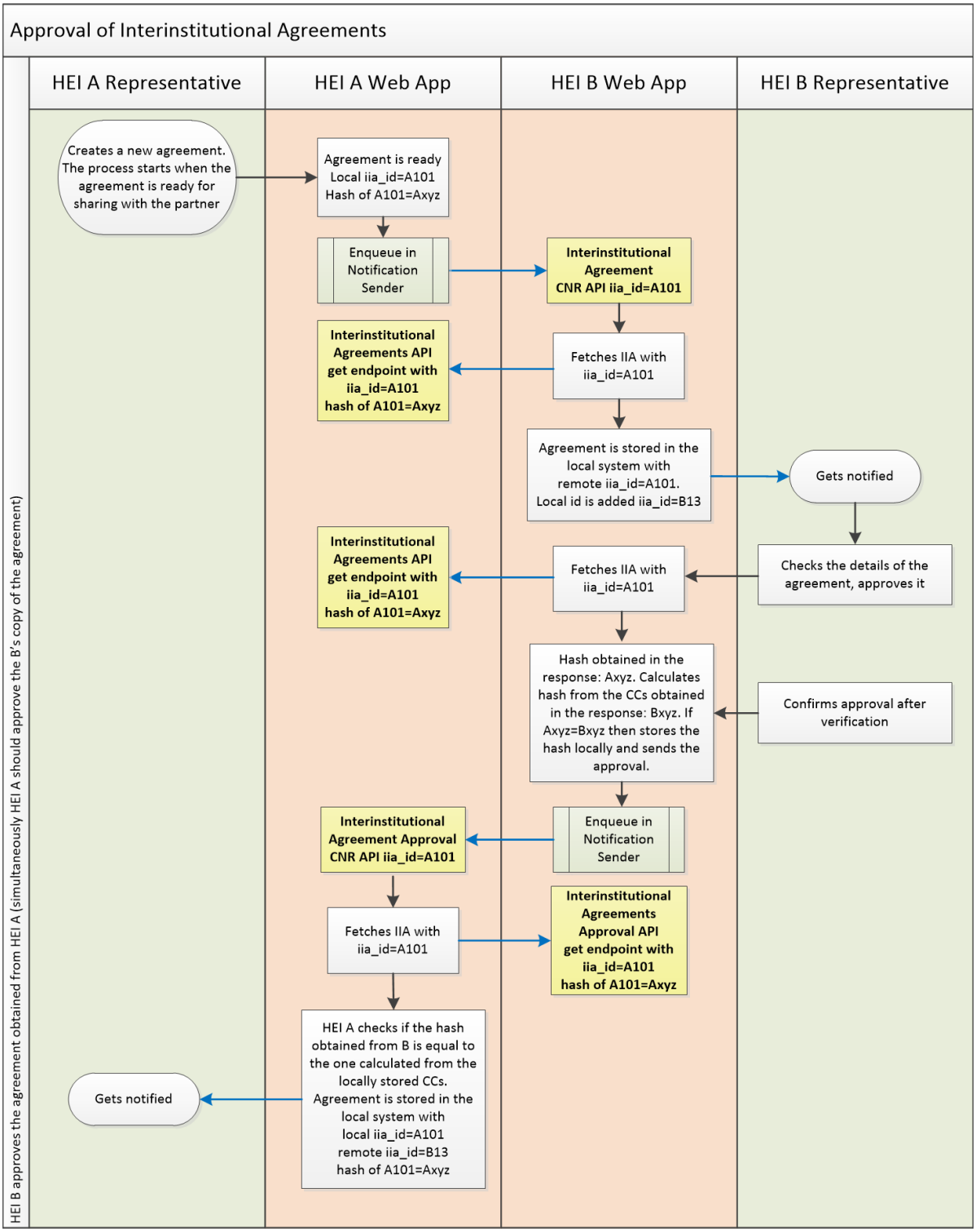


Figure B.1: EWP Mobility process: Approving the IIA **source**: https://github.com/erasmus-without-paper/ewp-specs-mobility-flowcharts/blob/v0.5.0/flowcharts/iia-approval.png

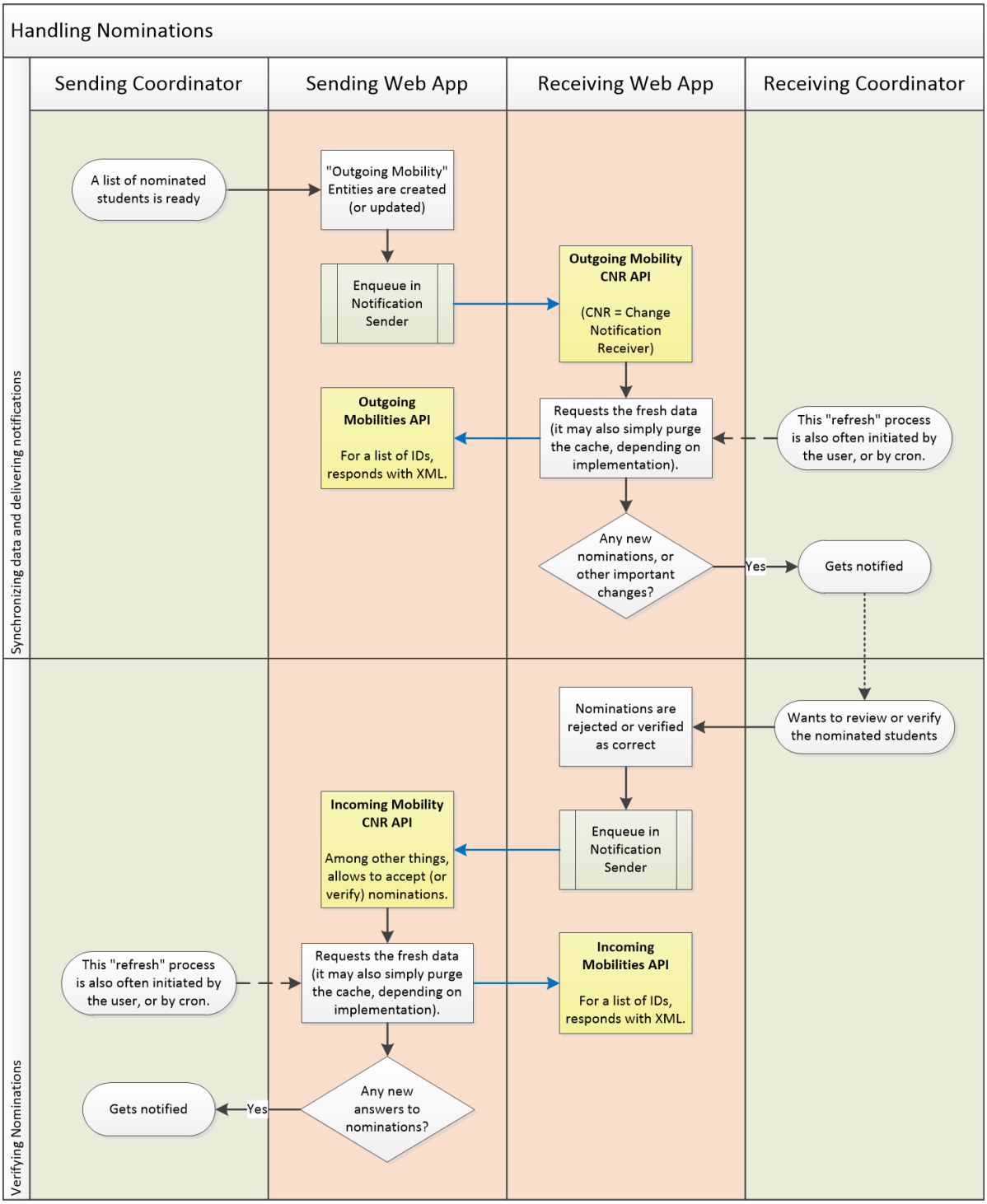


Figure B.2: EWP Mobility process: Mobility Nominations, **source**: https://github.com/erasmus-without-paper/ewp-specs-mobility-flowcharts/blob/v0.5.0/flowcharts/nominations.png

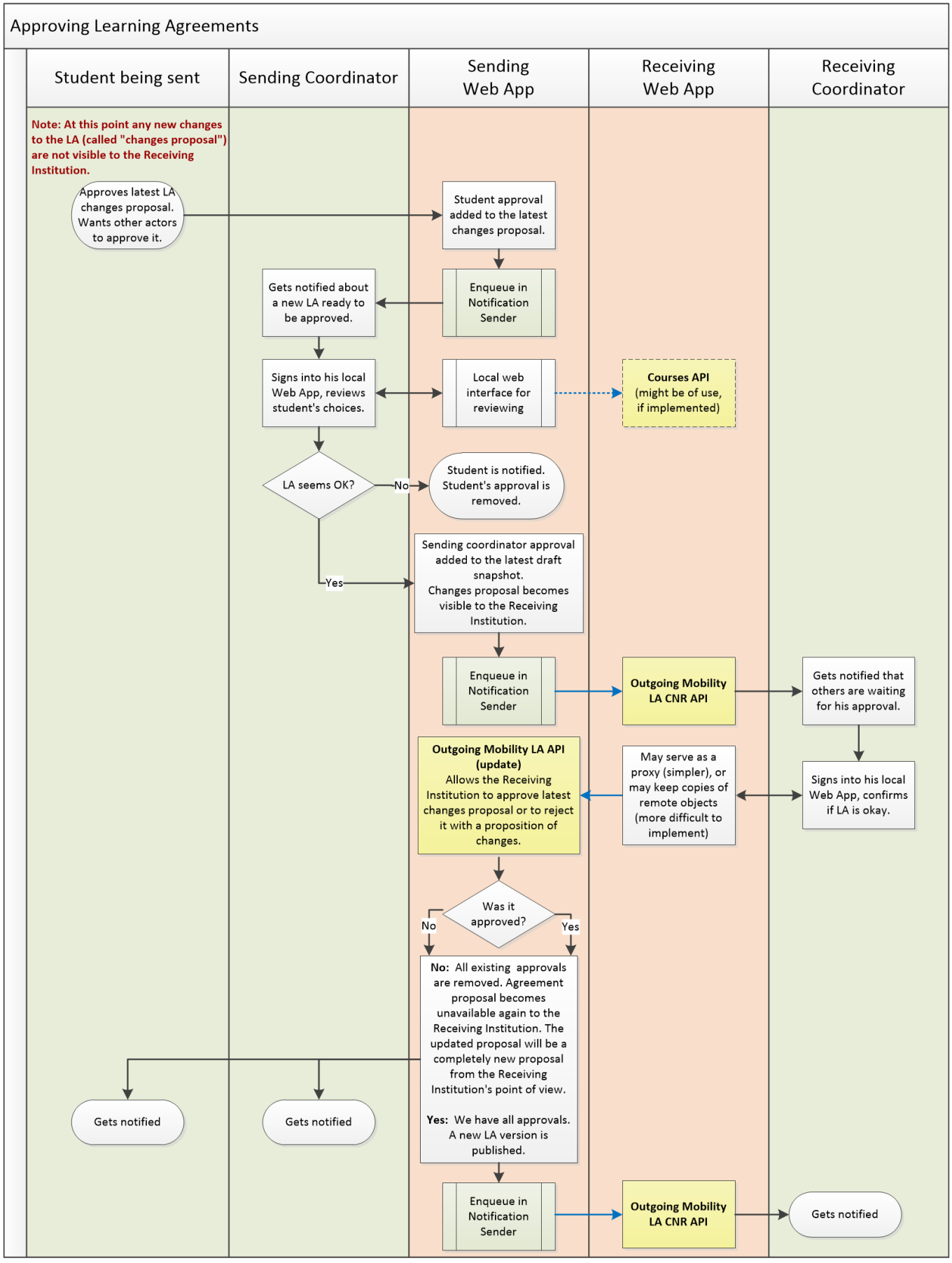


Figure B.3: EWP Mobility process: LA approval, **source**: https://github.com/erasmus-without-paper/ewp-specs-mobility-flowcharts/blob/v0.5.0/flowcharts/la-approving.png

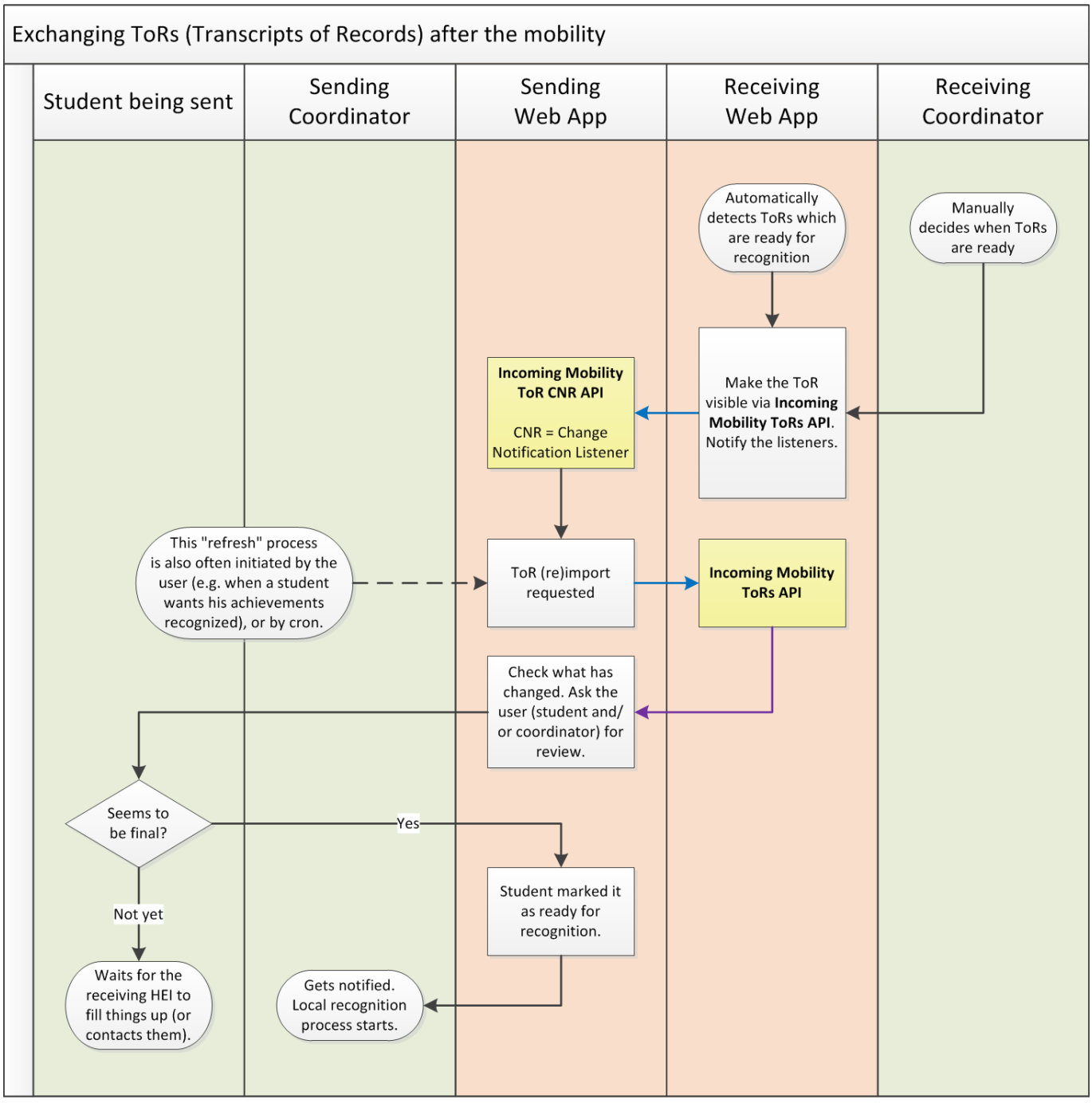


Figure B.4: EWP Mobility process: Exchanging ToRs, **source**: https://github.com/erasmus-without-paper/ewp-specs-mobility-flowcharts/blob/v0.5.0/flowcharts/exchanging-tors.png

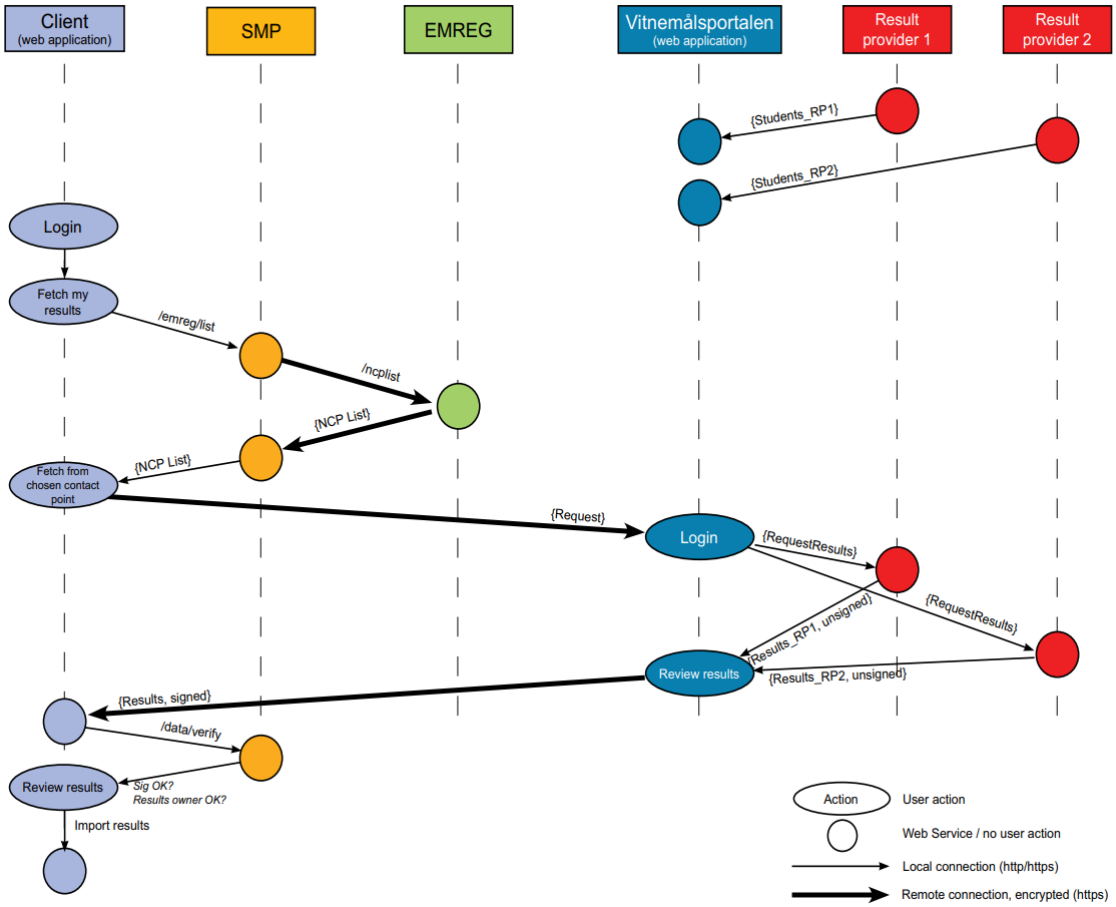


Figure B.5: EMREX Recognition, source: https://emrex.eu/wp-content/uploads/2020/01/Technical-Guide-to-EMREX.pdf.

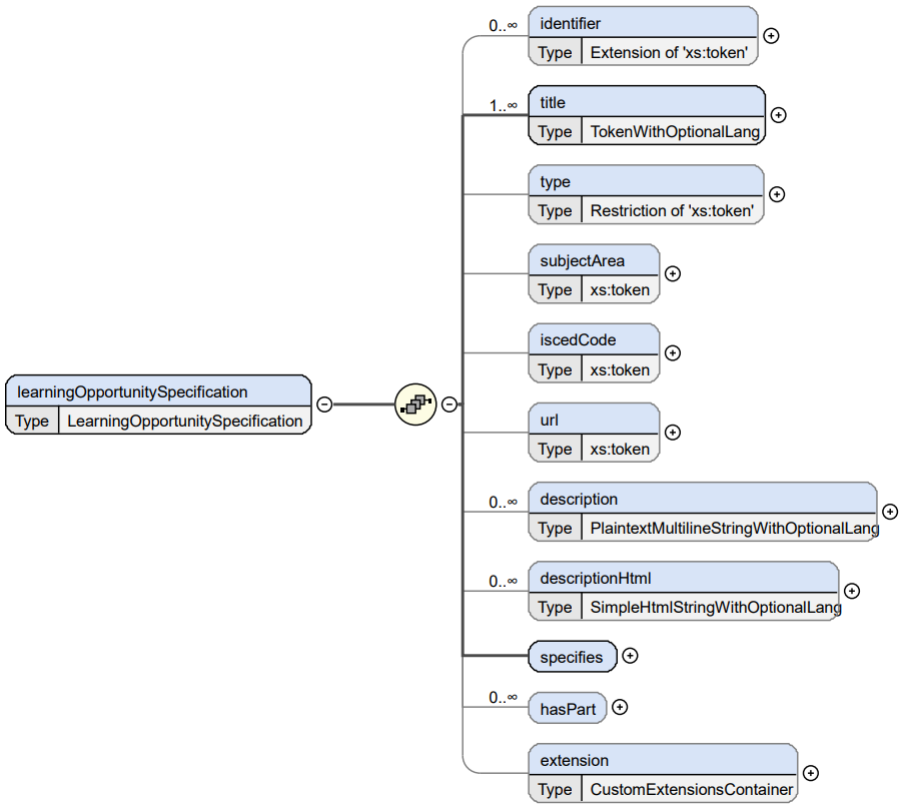


Figure B.6: ELMO LOS schema, source: https://emrex.eu/wp-content/uploads/2020/01/Technical-Guide-to-EMREX.pdf.

Appendix C

SIGARRA Web Service (WS)

Attribute	Description	Type	Obligatoriness
cce_id	Course ID	Int	Mandatory
name	Course Name	[lang:String(Alpha-2), description:String]	Mandatory
type	Course type	String	Mandatory
parent_institutions	Organic Units involved and if they are administrative	[hei_id:String, admin:String]	Mandatory
initials	Course initials/acronym	String	Mandatory
academic_year_start	beginning academic year	String	Mandatory
academic_year_end	ending academic year	String	Optional
dges_code	Course DGES code	String	Optional
cycle_start	Initial study cycle	Int	Optional
cycle_end	Ending study cycle	Int	Optional
course_duration	Course duration	type:String, time:Int	Optional
credit	Course credits	scheme:String, level:String, value:Float	Optional
language	Course language	String	Optional

Table C.1: Course information answer attributes.

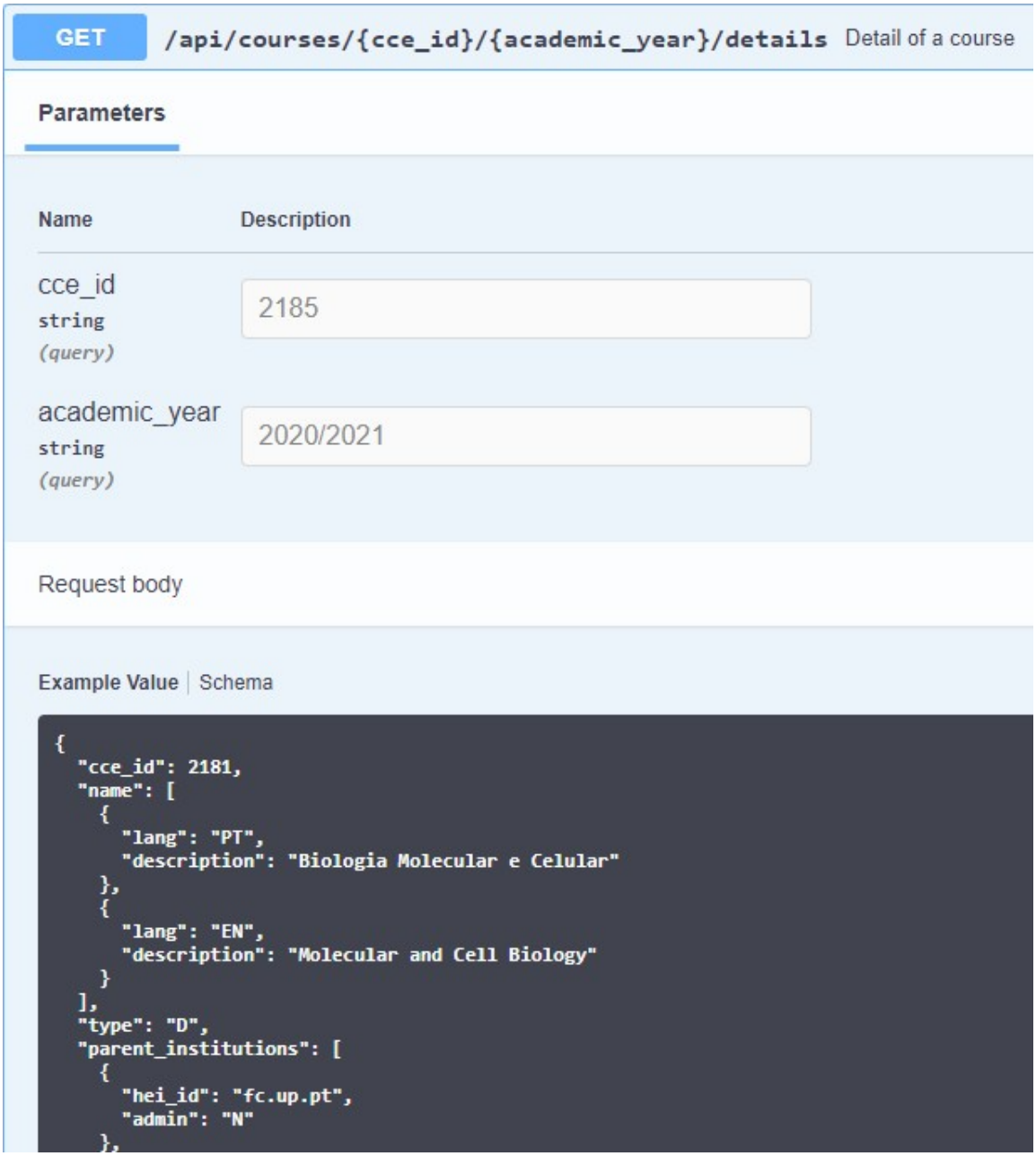


Figure C.1: Swagger print showing courses Web Service (WS).

Appendix D

Agreements and notifications flow chart

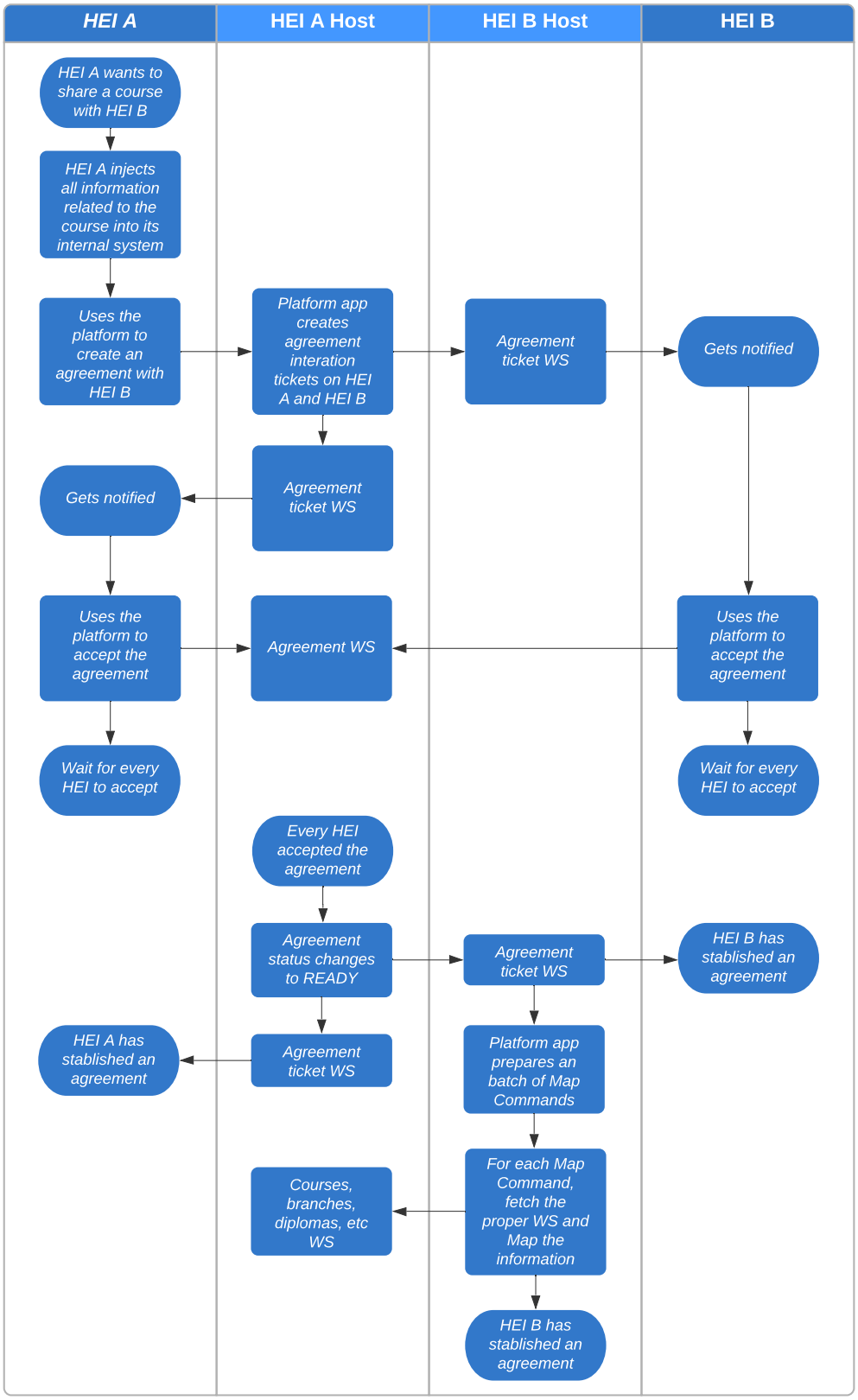


Figure D.1: Agreement flow chart.