SIGN HUB

European Commission | Horizon 2020 European Union funding for Research & Innovation

Project Number: 693349

Report on platform development prototype (first release)

Pasquale Baldassarre, Pietro Braione, Francesco Sessa

CINI

Version 1.9 – 01/10/2018

Lead contractor: Universitat Pompeu Fabra
Contact person: Josep Quer Departament de Traducció i Ciències del Llenguatge Roc Boronat, 138 08018 Barcelona Spain Tel. +34-93-542-11-36 Fax. +34-93-542-16-17 E-mail: josep.quer@upf.edu
Work package: 3
Affected tasks: 3.3

Nature of deliverable[1]	R	P	D	O
Dissemination level[2]	PU	PP	RE	CO

[1] *R: Report, P: Prototype, D: Demonstrator, O: Other*

[2] ***PU**: public, **PP:** Restricted to other programme participants (including the commission services), **RE** Restricted to a group specified by the consortium (including the Commission services), **CO** Confidential, only for members of the consortium (Including the Commission services)*

COPYRIGHT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 693349.

History of changes

VERSION	DATE	CHANGE	REVIEWER(S)
1.0	15/03/2018	Initial version.	Mauro Pezzè
1.1	17/03/2018	Revised.	Mauro Pezzè
1.2	29/03/2018	Revised.	Jordina Sánchez Amat
1.3	29/03/2018	Revised.	Josep Quer
1.4	30/03/2018	Finalized changes.	Pietro Braione
1.5	21/09/2018	• Implemented actions proposed in the Reply to the 1st reporting period project review v 1.0, section 5.6; • Improved description of the architecture; • Improved description and figures on the internals of the software implementation; • Updated list of used technologies.	Pietro Braione
1.6	28/09/2018	Revised.	Jordina Sánchez Amat
1.7	29/09/2018	Revised.	Pietro Braione
1.8	30/09/2018	Revised.	Josep Quer
1.9	01/10/2018	Finalized.	Pietro Braione

INDEX

Executive summary

This deliverable presents the status of the development of the software platform as implemented in the first prototype, released at month 24 of the project (MS26). It contains:

- A high-level description of the software architecture of the SIGN-HUB platform and of how the software is deployed in a front-end, running in the users' local browsers, and a back-end, running on servers in the cloud (Section 1);
- The status of the four tools that compose the SIGN-HUB software platform, expressed as the number of software requirements implemented with respect to the total number of requirements (Section 2);
- The detailed description of the internal structure of the software, and the design of the data (Section 3);
- The specification of the communication and data exchange protocol between the front-end and the back-end of the software (Section 4);
- Finally, the description of how the grammar tool implements the interface to the SignGram Blueprint (Section 5). This section implements the action proposed in Section 5.6 of the *Reply to the 1st reporting period* document ("D2.2 will be incorporated to the deliverable … by month 24 (D3.3)").

1. Overview of the SIGN-HUB Platform software architecture

This section briefly presents the high-level software architecture of the SIGN-HUB platform. The architecture presented in this section is a slight evolution of the architecture proposed in D3.5.

1.1. Software suites and modules

Vertically, the SIGN-HUB web platform is composed of 4 software *suites*, also referred to as *tools*, that are seamlessly integrated to provide a comprehensive and holistic access to the linguistic digital assets provided by project partners. The tools, whose functional requirements have been defined in D3.1, are here summarized:

- The *Test Administration Suite for Sign Languages (Testing Tool)*: A set of online sign language assessment instruments for education and clinical intervention;
- The *Atlas of Sign Language Suite (Atlas Tool)*: An interactive digital atlas of linguistic structures of the world sign languages;
- The *SignGram Blueprint Suite (Grammar Tool)*: An online grammar writing tool to produce digital grammars of sign languages, which will be used to create the grammars of 6 sign languages during the project;
- The *Digital Archive of Elderly Signers' Linguistic and Cultural Heritage Suite (Streaming Tool)*: A Digital archive of life narratives by elderly signers, subtitled and partially annotated for linguistic properties.

At a finer grain, the software is built of 8 modules that implement specific application functionalities. The modules collaborate realizing the application logic of the 4 software suites. Most modules are shared across at least two suites, but only 4 modules offer functionalities that are used by all the suites: User interface; Authentication, authorization, and accounting; Management of multimedia content, and Management of persistent data.

1.2. Tiers and deployment structure

Horizontally, the SIGN-HUB platform is a three-tier web application. A three-tier architecture fosters modular software with well-defined interfaces that allows any of the tiers to be upgraded independently in response to changes in requirements or technology. A three-tier application is organized into three major parts (tiers), each of which can be deployed on different physical places in a network. The three parts are:

- Presentation tier,
- Business logic tier, and
- Data tier.

The presentation tier is visible to the users and includes the user's computer system and network (Internet) that is used for accessing the cloud system (the *front-end*). The business logic and data tiers include the software, servers, data storage systems and local network that together constitute the services accessed by the presentation tier (the *back-end*). The front-end module manages the interactions with the users through the graphical user interfaces and translates the user inputs to web requests for a back-end module. These web requests follow a RESTful protocol based on the exchange of JSON objects. The protocol is fully detailed in Section 4 of the current document. The back-end modules implement the application logic and manage the persistent storage of the data.

In the SIGN-HUB platform the presentation tier is deployed and runs on the web browsers at the users' workstations; the business logic tier is located on the network server that responds to browser requests. In turn, it determines what data is needed and where it is located and acts as a client to the third tier located on the DBMS, deployed on the same or on a different server than the business logic tier.

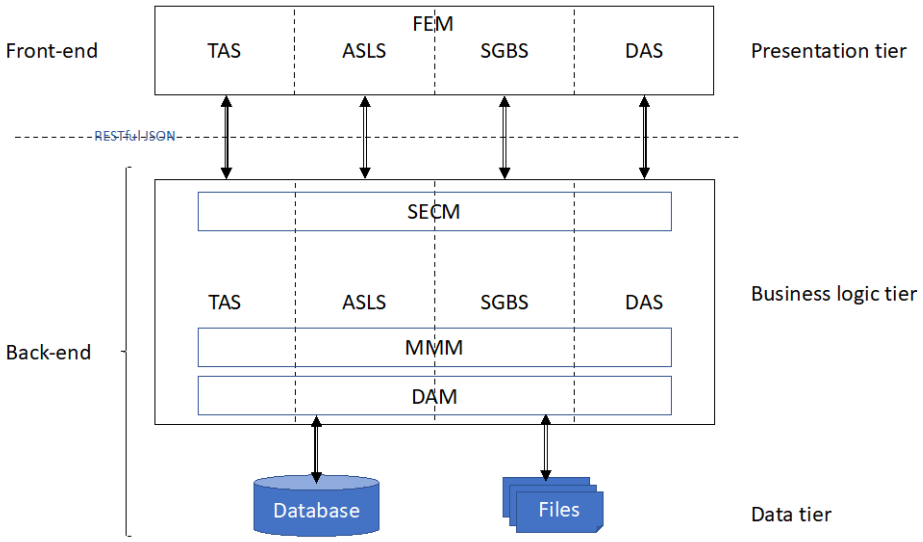

Figure 1. Software architecture of the SIGN-HUB platform

Figure 1 briefly illustrates the above concepts. It shows the suites as vertical partitions:

- TAS is the Test Administration Suite for Sign Languages;
- ASLS is the Atlas of Sign Language Suite;
- SGBS is the SignGram Blueprint Suite;
- DAS is the Digital Archive of Elderly Signers' Linguistic and Cultural Heritage Suite.

Figure 1 also shows the 4 of the 8 software modules that are shared by all the suites, as boxes overlapping the vertical partitions:

- The Front-End Module (FEM);
- The Security Module (SECM);
- The Multimedia Management Module (MMM);

- The Data Abstraction Module (DAM).

1.3. Technological Considerations

The previous considerations brought the development team to rethink the technologies that are used for developing the SIGN-HUB platform. The team decided to develop each tier of the SIGN-HUB platform based on different software technologies.

The **presentation tier** is based on Angular[3]. Angular is a client-side JavaScript framework that can run in any browser that supports HTML 5 and JavaScript. Angular follows a Model-View-Controller (MVC) pattern, which encourages loose coupling between presentation, data, and application logic. The data for a web application based on Angular are obtained as objects in the JSON format by invoking RESTful API calls to the business logic tier. With respect to the technological choice reported in D3.5, we decided to abandon the obsolescent AngularJS framework and adopt its most recent evolution Angular.

The **business logic tier** is based on the Play Framework[4], a fully RESTful Java framework that ensures a high degree of application scalability, supporting thousands of concurrent client connections. We chose to base the development of the business logic tier on the Play Framework because of the potential beneficial effects it yields on the development (less development effort and better scalability of the resulting application), when compared with developing in pure Java with Hibernate, as proposed in D3.5. The Play Framework integrates Hibernate, thus it allows the development of the Data Abstraction Module as planned in D3.5.

The **data tier** is implemented using MariaDB, a secure, highly available, scalable and open source relational database server with a modern, extensible architecture widely adopted in large and mission-critical environments. This confirms our initial technological choice in D3.5.

Table 1 reports the chosen technologies and supersedes Table 2 of D3.5.

Table 1 - SIGN-HUB Web Platform Technologies

Tier/Module	Technology	Version
Presentation Tier	Angular	4.0
Business Logic Tier	Java	8.181
	Play	1.4.4
Data Abstraction Module	Hibernate	4.2.19
Data Tier	Maria DB	5.5

[3] *https://angular.io*

[4] *https://www.playframework.com/*

2. Status of the development

To improve communication with users, we chose whole suites as the units of software releases. This is because users perceive, access, and use tools, thus it makes sense for users knowing that a specific tool was released in a new version. This section will therefore report the status of the development of the four suites (Testing tool, Atlas tool, Grammar tool and Streaming tool) at the end of project year 2 (month 24).

We should however consider that the Security, Multimedia Management, and Data Management Module provide essential services to the business logic of the four suites, and all the suites depend on them. For this reason, we chose to report the progress in the development of these modules also.

We will quantify the progress by reporting the percentage of the requirements in D3.1 that have been completed.

2.1. The SignGram Blueprint Suite

This suite exposes the functionalities to edit a sign language grammar. It provides a rich-text editor augmented with custom functionalities required to define specific styles. Moreover, it allows end-user to navigate the grammar providing custom search functions to easily find the sections of interest. More information is given in Section 5.

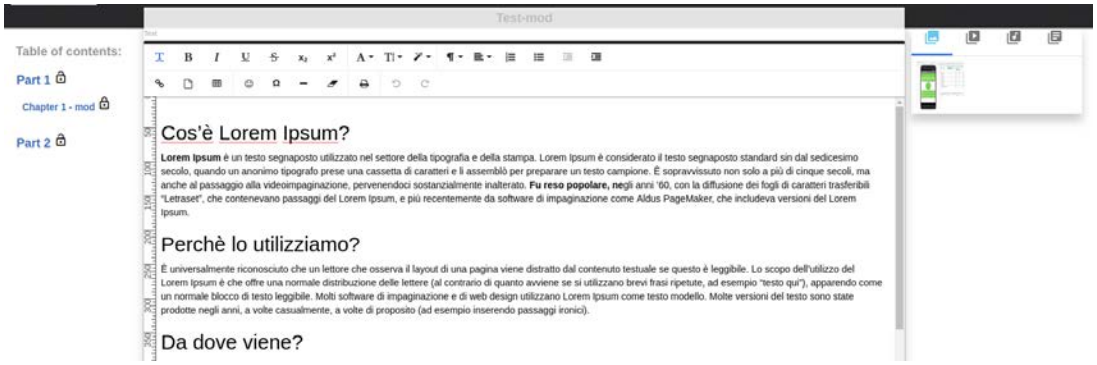

Figure 2. User interface of the grammar tool

2.1.1. Status

The suite is in its 1st release. Currently we have developed:

- 75% of the requirements for content providers (D3.1 section 4.b.i), and
- 0% of the requirements for end users (D3.1 section 4.c.i).

2.2. The Atlas of Sign Languages Suite

The Atlas of Sign Languages Suite lets user interact with the specific GUI to create a test and validate the obtained answers. In the next steps it will use answer in the Geographic graphic view.

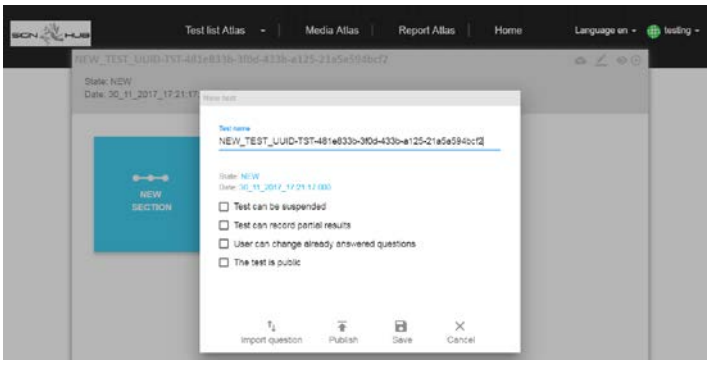

Figure 3. User interface of the atlas tool

2.2.1. Status

The suite is in its 2nd release. Currently we have developed:

- 30% of the requirements for content providers (D3.1 section 4.b.ii), and
- 0% of the requirements for end users (D3.1 section 4.c.ii).

2.3. The Test Administration Suite for Sign Languages

This suite exposes both the functionalities required to build a test composed of different types of questions, and the functionalities required by a subject under test to perform the test. Each test is composed of a set of stimuli and a set of answers. Stimuli and answers can be constructed by using different types of data such as image, video or text and different kinds of controls such as check boxes and radio buttons.

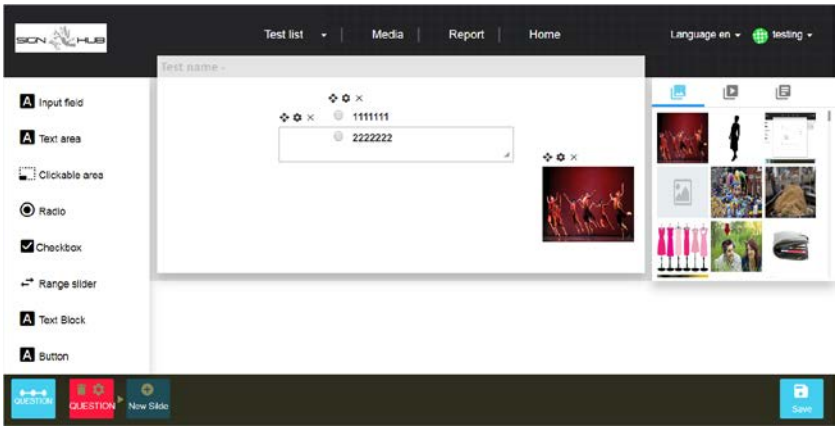

Figure 4. User interface of the testing tool

2.3.1. Status

The suite is in its 2nd release. Currently we have developed:

- 80% of the requirements for content providers (D3.1 section 4.b.iii), and
- 0% of the requirements for end users (D3.1 section 4.c.iii).

2.4. The Digital Archive of Elderly Signers' Linguistic and Cultural Heritage Suite

The Digital Archive of Elderly Signers' Linguistic and Cultural Heritage Suite mediates all interactions between the user and the Digital Archive of Elderly Signers' Linguistic and Cultural Heritage. This archive will be a container of multimedia resources, mainly videos with subtitle annotations. The suite allows to search through the archive and streams the video resources for visualization.

2.4.1. Status

The suite is in its 1st release. Currently we have developed:

- 20% of the requirements for content providers (D3.1 section 4.b.i), and
- 50% of the requirements for end users (D3.1 section 4.c.i).

2.5. The Security Module

The Security Module performs a set of functionalities that are indicated with the acronym "AAA", standing for Authentication, Authorization, and Accounting. The module is used for controlling user access to resources, enforcing policies and auditing usage. Authentication provides a way of identifying the user, by requiring the user to enter a valid username (email) and password before access to the SIGN-HUB platform is granted. Authorization is the process of enforcing policies: determining what types or qualities of activities, resources, or services a user is permitted to access. Accounting measures what a user does during access. Accounting is carried out by logging session statistics and usage information.

This module will implement the functionalities necessary to the administration of the platform.

2.5.1. Status

The module is in its 3rd release. Currently we have developed:

- 50% of the requirements for administrators (D3.1 section 4.a).

2.6. The Multimedia Module

The Multimedia Module is responsible for managing all the multimedia data of the SIGN-HUB Web Platform. This module provides all the functions needed to smartly manage and upload media contents such as video, text, documents and metadata files. This module interacts with the Data Abstraction Module to permanently store multimedia data, and all the tool modules use the Multimedia Module to manage multimedia content.

2.6.1. Status

The module is in its 4th release. It does not implement any requirement (it implements exclusively internal functionalities).

2.7. The Data Abstraction Module

The Data Abstraction Module offers services of object-relational mapping (ORM) to the other modules of the platform. It manages data persistence on the database by representing and maintaining data as objects in the programming logic. This strategy facilitates modeling entities based on application concepts, rather than based on the inherent database structure having the whole data model implemented according to the object-oriented programming paradigm. The module is an adaptation of the open source Hibernate framework to the data stored by the SIGN-HUB application.

3. Detailed Design of the Software

This section reports the detailed design of the business logic and data tiers as they have been implemented for release 1 of the platform.

3.1. Business Logic Tier

This subsection presents the software architecture of the business logic tier as a set of class diagrams in UML notation. Each UML class describes a corresponding Java class in the software implementation. For the sake of simplicity in presentation, the classes will be divided according to the modules they belong to.

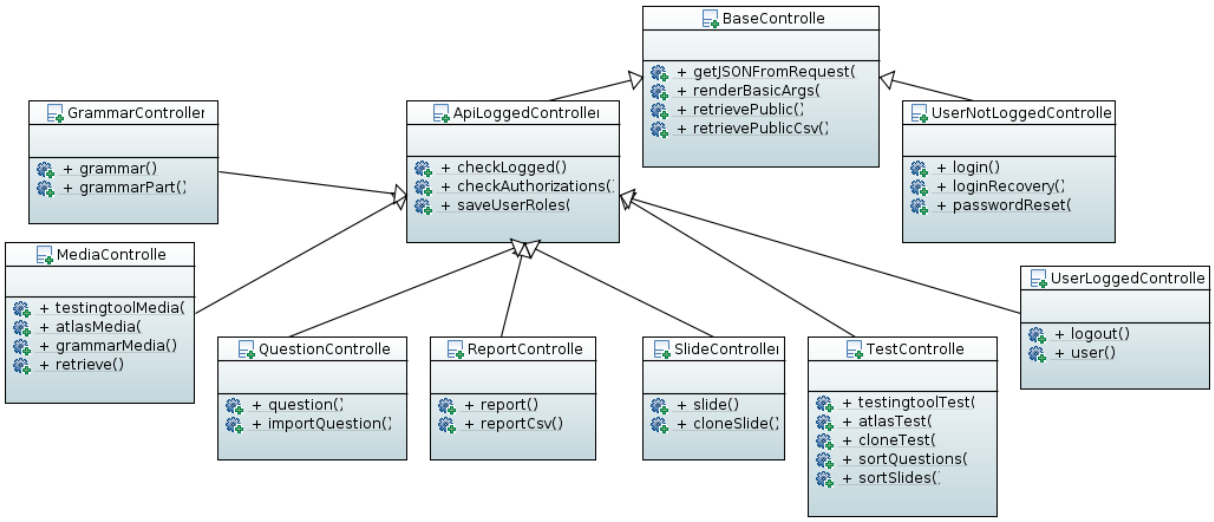

Figure 5. Class diagrams: JSON REST controllers

Figure 5 reports the JSON REST controller classes together for readability (they will also be replicated in the diagrams of the modules where they belong). These classes are the "entry point" of the business logic tier, and their operations (method) are invoked as the presentation tier sends a RESTful request for an operation. There are distinct controllers for unlogged user, logged users, and for the different kind of functionalities offered by the four suites. Inheritance is used whenever different controllers must share a common functionality.

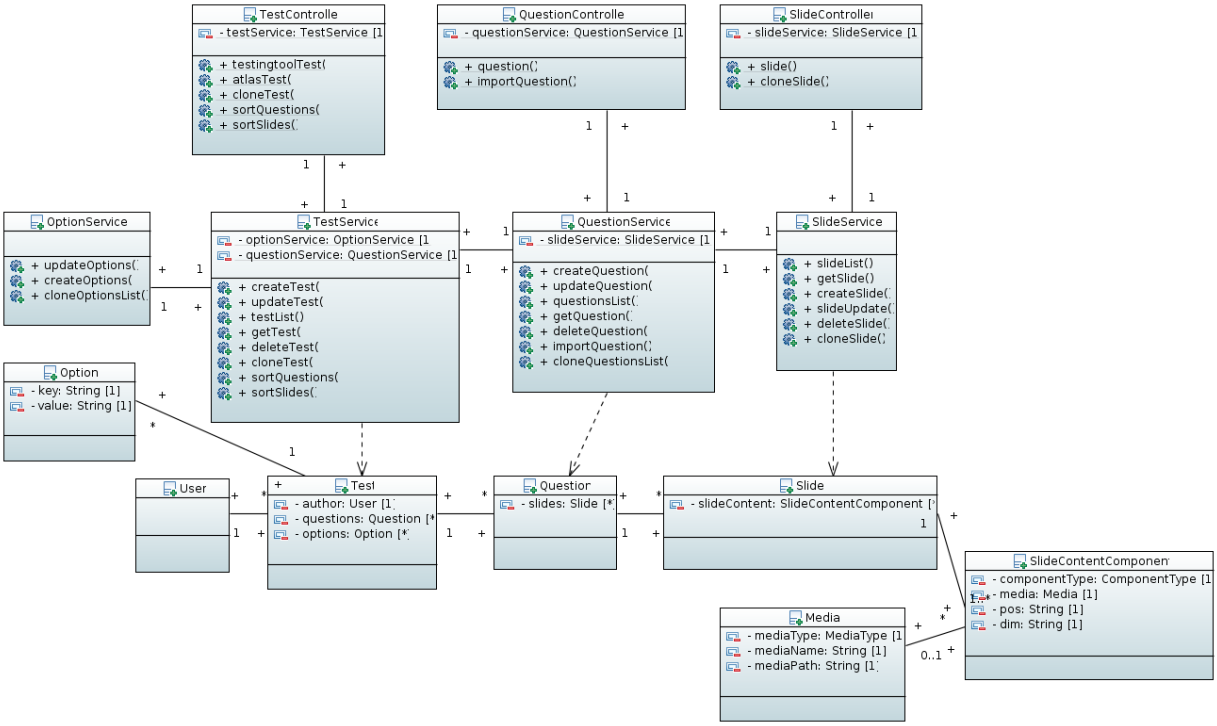

Figure 6. Class diagram: Test Manager module

Figure 6 represents the Java classes of the Test Manager module. While the controller classes (replicated on top of the diagram) have the responsibility of managing the interaction with the presentation tier, the service classes (at the center of the diagram) actually implement the requested operations. They depend on the data classes (at the bottom of the diagram), that mirror the data contained in the data tier. The data classes, their members and associations are almost identical to the corresponding entities, attributes and relationships in the E-R diagram for the tests. The lifecycle of the data classes is transparently managed by the Data Abstraction Module.

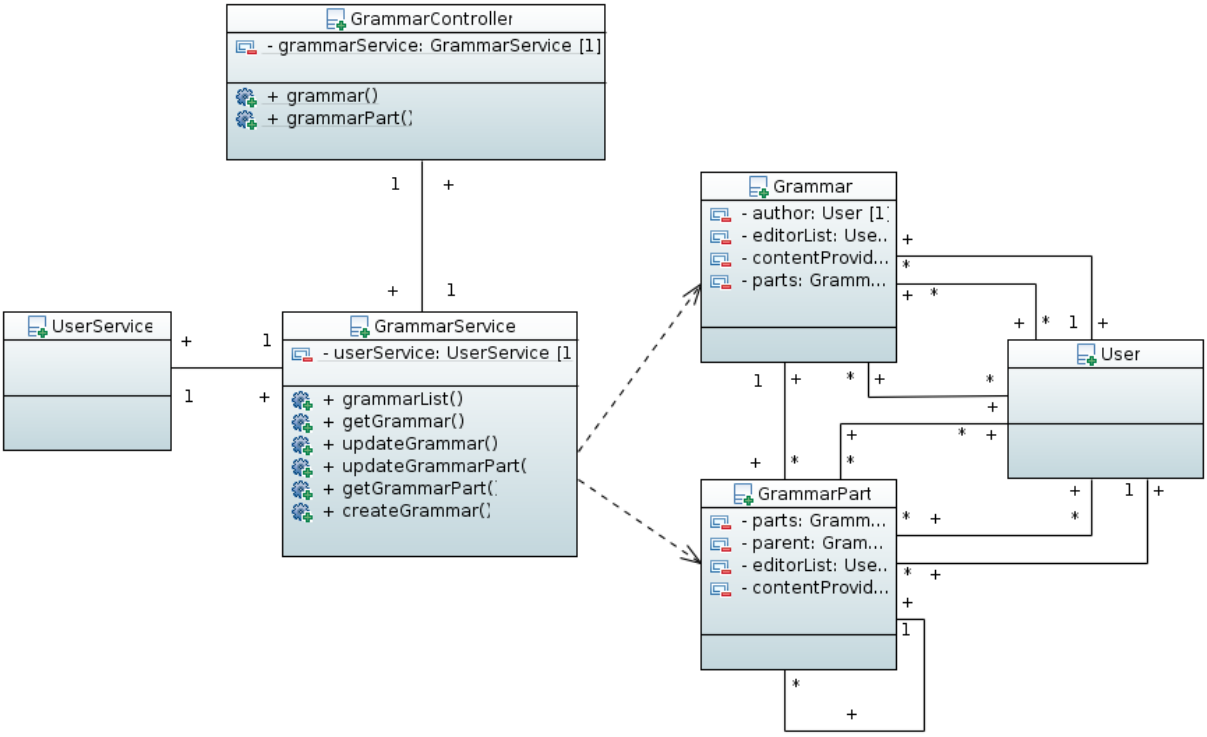

Figure 7. Class diagram: Grammar Manager

Figure 7 represents the Java classes of the Grammar Manager module. As in Figure 6 the controller class (on top) interacts with a service class that actually performs the operations on the grammars. The data classes for the grammars and grammars' parts are on the right.

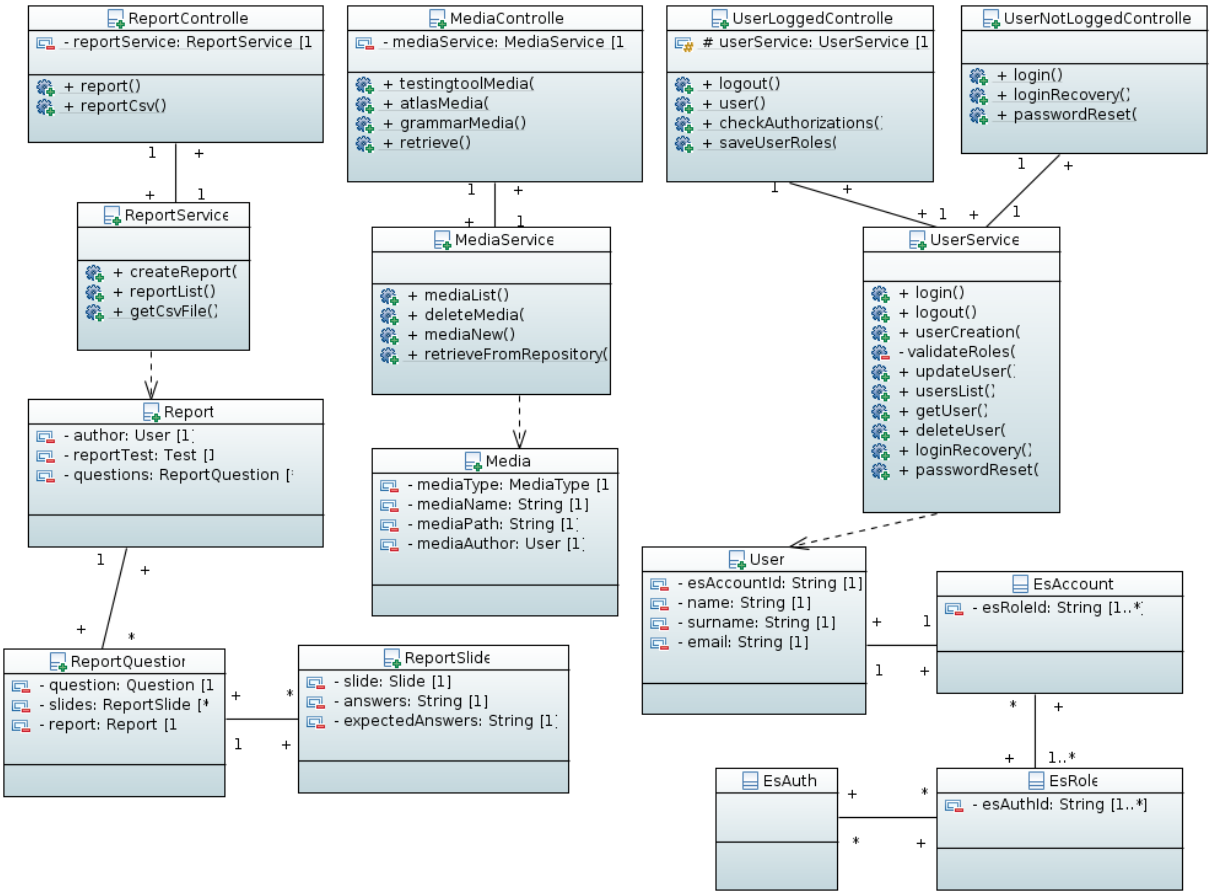

Figure 8. Class diagram: Analytics, Multimedia Manager and Security modules

Similarly, Figure 8 represents together the classes in the Analytics, Multimedia Manager and Security modules. The structure is similar: The controller classes redirect operation requests to suitable service classes, which in turn manipulate data object, whose classes directly mirror the structure of the data stored in the data tier.

3.2. Data Tier

This subsection presents the design of the data tier as a set of E-R diagrams in "crow's foot" notation, describing at a high level of abstraction the application data that will be stored and managed by the tier (actually, by the MariaDB relational database).

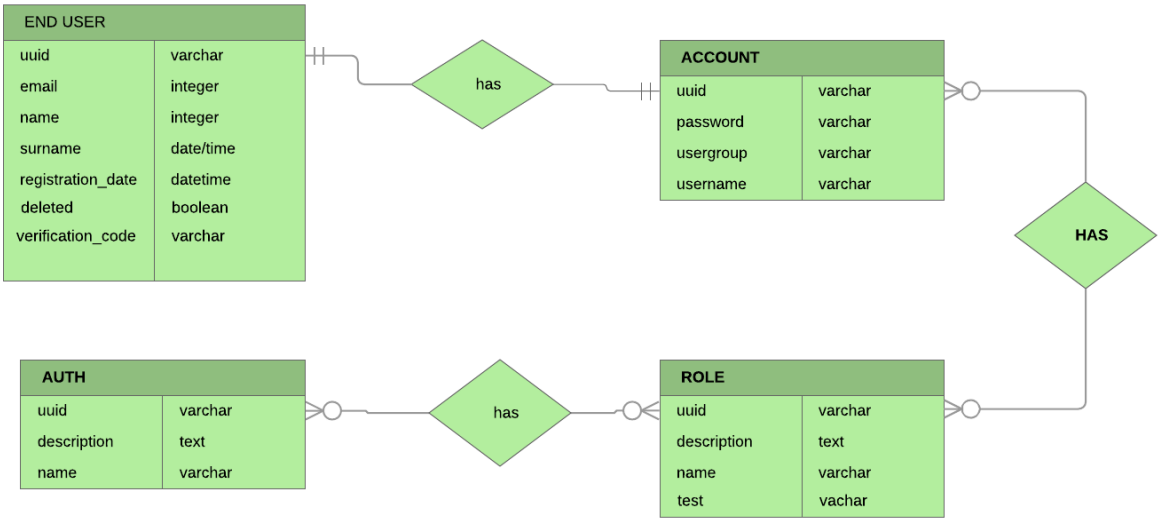

Figure 9. E-R diagram: AAA data.

Figure 9 reports the data design for the AAA data. The END USER entity describes a generic user of the platform as his/her anagraphical data. The ACCOUNT entity describes an account registered at the platform. The association between END USERs and ACCOUNTs is one-to-one. Every account can have one or more ROLEs, where each ROLE is associated to a set of AUTHorization rights, i.e., of operations the ACCOUNTs with that ROLE are allowed to perform.

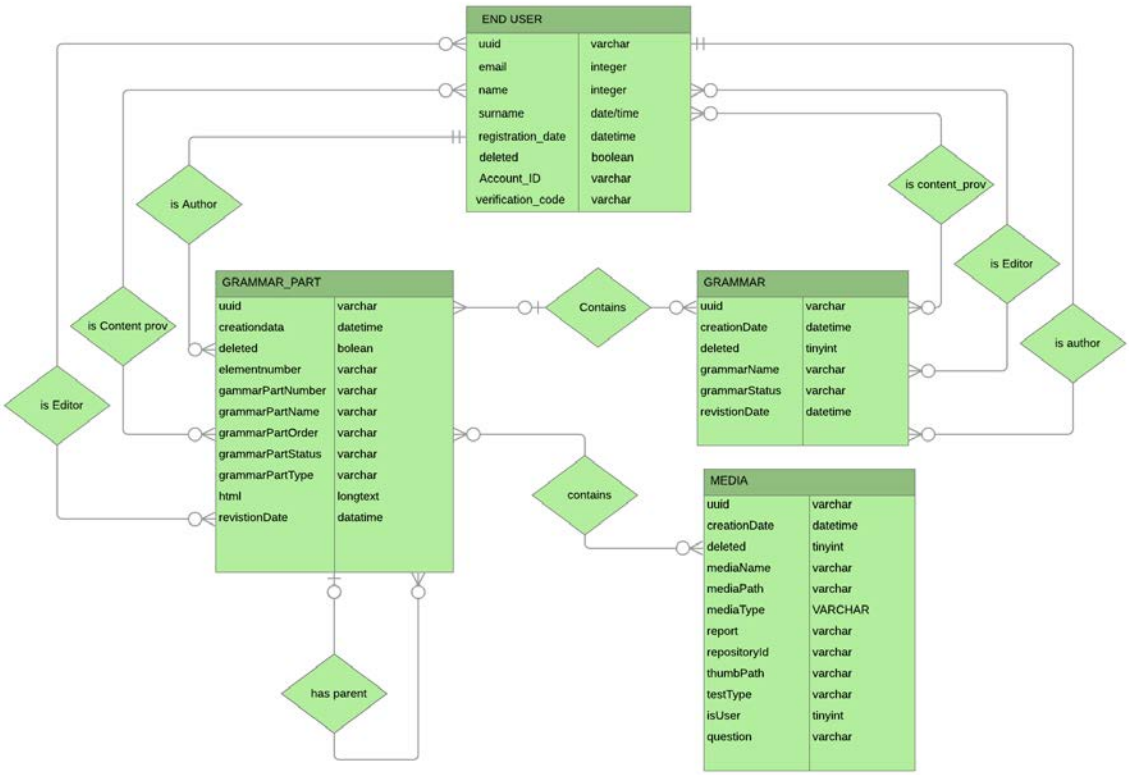

Figure 10. E-R diagram: Grammars

Figure 10 reports the data design for grammars. A GRAMMAR contains a number of GRAMMAR_PARTs, and each GRAMMAR_PART can contain other GRAMMAR_PARTs via the "has parent" relationship, yielding a hierarchical structure of sections and sub-sections. A GRAMMAR_PART can contain a number of MEDIA objects. There are many relationships between GRAMMARs and END USERs: each GRAMMAR has exactly one author, one or more editors, and one or more content providers. Similar relationships exist between GRAMMAR_PARTs and END USERs.

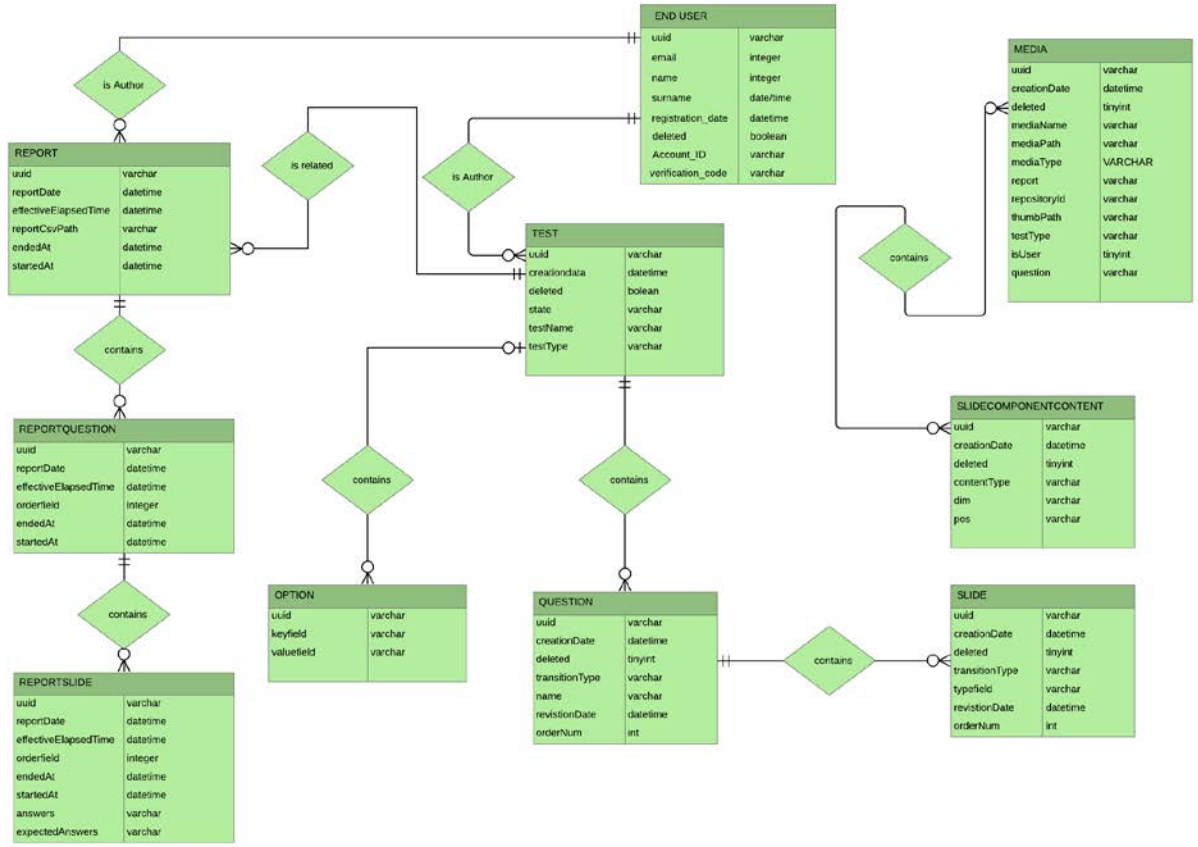

Figure 11. E-R diagram: Tests

Finally, Figure 11 reports the data design for tests, that encompasses the assessment tests and the atlas questionnaires. A TEST is authored by one END USER and contains a set of QUESTIONs. These in turn contain a set of SLIDEs, with their components (SLIDECOMPONENTCONTENT). Slide components can contain MEDIA. When an END USER performs a TEST, his/her answers are registered in a REPORT, which is structured in REPORTQUESTIONs and REPORTSLIDEs similarly to the corresponding TEST.

4. RESTful back-end interface

This section reports the detailed RESTful interfaces offered by the back-end of the SIGN-HUB platform to the front-end. In Section 4.1 we will report on the structure of the data structures that are exchanged between the front-end and the back-end via the functions that will be detailed in Sections 4.2, 4.3, 4.4 and 4.5.

We remark that the Security module offers a common RESTful interface distinct from all the interfaces offered by the four suites. This interface will be reported on in Section 4.2. Moreover, the design of the interface between the Digital Archive of Elderly Signers' Linguistic and Cultural Heritage Suite and the Front-end module will be addressed by the second release of the platform.

4.1. JSON Data Structures

This section reports the JSON objects that are exchanged between the front-end and the back-end. Note that:

- We describe a JSON structure type as a sequence of field descriptions enclosed in curly brackets. Every field description is in the form "fieldName" : fieldType. An optional field is indicated as (optional) "fieldName" : fieldType.
- A field type can be the primitive JSON type string, the primitive JSON type number, a date, a pair of percent values (percentPair), an associative map, an enumeration of strings separated by the | | symbol, another JSON structure type, a homogeneous array whose elements have all same type, or a union type.
- The type date is a string in the form dd/mm/yyyy, where dd is a two-digit day of the month number (from 01 to 31), mm is a two-digit month of the year number (from 01 to 12) and yyyy is a four-digit year number (e.g., 2018).
- The type percentPair is a string containing a pair of numbers between 0 and 100 separated by a comma, e.g., "47,12".
- The type map is an associative array in the form ["key1" : "value1", "key2" : "value2" ... "keyN" : "valueN"], where keys and values have all type string.
- The | | symbol indicates an enumeration of values with type string; for instance, "status": OK | | NOK indicates that the status field of the object can assume either the value "OK" or the value "NOK".
- Capitalized names ending with OBJ indicate a JSON structure type; for instance, "response": { OBJ } indicates that the response field of the object is another object of an unspecified type, enclosed in curly brackets.
- Square brackets indicate a homogeneous array of data all of the same type; for instance, "errors" : [ERROR_OBJ] indicates that the errors field of the object is an array of objects with type "Error Object".
- The union of two types T1 and T2 is indicated by T1 or T2.

4.1.1. Response Object

Response objects are returned by the bac-kend as answers to all the functions exposed to the front-end. Here we report on the general structure of the response object. Every function refines this structure by adding function-specific information in the

“response” field. Sections 4.2, 4.3, 4.4 and 4.5 report on the actual structure of the response objects returned by each function.

```
{
    "status": OK | | NOK,
    "errors”: [ERROR_OBJ],
    "response”: OBJ //an arbitrary JSON object that will be detailed in Sections 4.2,
                    //4.3, 4.4 and 4.5
}
```

4.1.2. Error Object

One or more error objects are returned embedded in a response object whenever the back-end fails in satisfying a function request from the front-end. Error objects indicate the nature of the error that caused the disservice.

```
{
    "errorCode" : number, //error code number
    "errorMessage" : string //error message
}
```

4.1.3. Small Test Object

A Small Test object is returned by all the functions of the Atlas Tool and Testing Tool back-end modules that produce as answer a list of previously created assessment tests. Such objects contain only a minimal set of information about the test, e.g., do not return the list of the questions contained in the test.

```
{
    "testId" : number, //Test id
    "testName" : string //Test name
    "authorId" : string, //author userID
    "toEdit" : true | | false //this flag indicates if the current user can delete the report
}
```

4.1.4. Test Object

A Test object is returned by all the functions of the Atlas Tool and Testing Tool back-end modules that are used to obtain one or more assessment tests registered in the platform, in full detail.

```
{
    "testId" : number, //Test id
    "testName" : string, //Test name
    "authorId" : string, //author userID
```

```
    "deleted" : true | | false, //flag for logical deletion
    "state" : draft | | published //state of the test
    "revId" : number,
    "toEdit" : true | | false //this flag indicates if the current user can delete the Test
    "questions" : [QUESTION_OBJ], //array of questions
    "options" : map //array of options
}
```

4.1.5. Question Object

Question objects are embedded in Test Objects. A question object fully describes a single question contained in an assessment test.

```
{
    "questionId" : string, //id of the question
    "slides" : [SLIDE_OBJ], //array of slides
    "transitionType" : time | | click | | enter | | action,  //type of transition between
                                                              //questions
    "options" : map, //array of options
    "toEdit" : true | | false     //this flag indicates if the current user can delete the
                                  //question
}
```

4.1.6. Slide Object

Slide objects are embedded in Question objects. A slide object fully describes a single slide contained in a test question.

```
{
    "slideId" : string, //slide id
    "slideType" : blank | | info | | stimulus | | distraction | | question | | answer,
                  //the type of the slide
    "transitionType" : time | | click | | enter | | action | | answer,  //type of transition
                                                                        //between slides
    "options" : map, //array of options
    "slideContent" : {SLIDE_CONTENT_OBJ},        //the slide content depends on
                                                 //slide type
    "toEdit" : true | | false //this flag indicates if the current user can delete the slide
}
```

4.1.7. Slide Content Object

Slide Content objects are embedded in Slide objects. A Slide Content object describes which media objects are contained in a slide, at which position, and with which size. The media object itself is not described, and is instead identified by a unique id string.

```
{
    “mediaArray” : [{
        “mediaId” : string, //the media id
        “pos” : percentPair, //the left,top position in %
        “dim” : percentPair, //the width,heigth dimension in %
        “value” : number,
        “options” : [
            {“autostart” : true | | false }
        ]
    }],
    “toEdit” : true | | false        //this flag indicates if the current user can delete
                                     //the content map
}
```

4.1.8. Media Object

A Media object describes in more detail a media object contained in a test slide.

```
{
    “mediaId” : string, //the media id
    “mediaType” : VIDEO | | PHOTO | | AUDIO | | TEXT, // media type
    “mediaName” : string, // the name of the media file, e.g., “elephant.jpg”
    “mediaPath” : string,     //the path to download the media, e.g.,
                              //“media/images/elephant.jpg”
    “mediaAuthorId” : string, //author userID
    “mediaAuthorName” : string, // the name of the author
    “mediaDate” : date, // the date of media creation
    “toEdit” : true | | false   //this flag indicates if the current user can delete
                                //the media
}
```

4.1.9. Small Report Object

A Small Report object is returned by all the functions of the Atlas Tool and Testing Tool back-end modules that produce as answer a list of previously created reports. A Small Report object does not contain all the information about the reports.

```
{
    “reportId” : string, // the id of the report
    “reportDate” : date, // the date of Test submission
    “reportCsvPath” : string,  //the path to download csv report,
                                //e.g., “/csv/ r354plv97sd.csv”
    “reportTestName” : string //the name of the related Test
    “reportTestId” : number, //the id of the related Test
    “toEdit” : true||false  //this flag indicates if the current user can
                            //delete the report
}
```

4.1.10. Report Object

A Report object is returned by all the functions of the Atlas Tool and Testing Tool back-end modules that produce as answer the detailed information about a previously created report.

```
{
    “reportId” : string, // the id of the report
    “reportDate” : date, // the date of Test submission
    “reportCsvPath” : string,    //the path to download csv report, e.g.,
                                 //“/csv/ r354plv97sd.csv”
    “reportTestName” : string, //the name of the related Test
    “reportTestId” : number, //the id of the related Test
    “toEdit” : true||false,    //this flag indicates if the current user can delete
                               //the report
}
```

4.1.11. Small Grammar Object

A Small Grammar object is returned by all the functions of the Grammar Tool back-end modules that produce as answer a list of previously created grammars. A Small Grammar object does not contain all the information about the grammars, e.g., does not contain the grammar parts.

```
{
    "uuid" :  string, // the id of the grammar
```

```
"grammarName": string, //the name of the grammar
"creationDate": date, //the date of creation
"revisionDate": date, //the date of the last revision
"grammarStatus": DRAFT | | OK, //the status of the grammar
"isDeleted": true | | false //is the grammar deleted?
}
```

4.1.12. Grammar Object

A Grammar object is returned by all the functions of the Grammar Tool back-end modules that produce as answer the detailed information about a previously created grammar.

```
{
"uuid" : string, // the id of the grammar
"grammarName": string, //the name of the grammar
"creationDate": date, //the date of creation
"revisionDate": date, //the date of the last revision
"grammarStatus": DRAFT | | OK, //the status of the grammar
"isDeleted": true | | false, //is the grammar deleted?
"parts": [SMALL_GRAMMAR_PART_ OBJ], //array of parts
"author": string, //author user id
"editors": [string], //array of user ids
"contentProviders": [string] //array of user ids
}
```

4.1.13. Small Grammar Part Object

A Small Grammar Part object is embedded in a Grammar object and describes (not in full details) a part or chapter of a grammar.

```
{
"uuid" : string, // the id of the grammar part
"isDeleted": true | | false,
"elementNumber": number, //the number of the part
"name": string, //the name of the part
"status": DRAFT | | OK, //the status of the part
"type": PART | | CHAPTER,
"creationDate": date
"revisionDate": date,
"parts": [SMALL_GRAMMAR_PART_OBJ], //array of subparts
```

```
    "author": string, //author user id
    "editors": [string], //array of user ids
    "contentProviders": [string] //array of user ids
}
```

4.1.14. Grammar Part Object

A Grammar part object describes in full details a part or chapter of a grammar.

```
{
    "uuid" :  string, // the id of the grammar part
    "isDeleted": true | | false,
    "elementNumber": number, //the number of the part
    "name": string, //the name of the part
    "status": DRAFT | | OK, //the status of the part
    "type": PART | | CHAPTER,
    "html": string, //the HTML text of the part
    "parts": [string], //array of part ids
    "author": string, //author user id
    "editors": [string], //array of user ids
    "contentProviders": [string] //array of user ids
}
```

4.2. AAA Functions

This section reports all the RESTful functions exposed by the Security module to the Front-end module. Note that:

- The first row indicates the HTTP method (GET, POST, PUT, DELETE).
- The second row indicates the relative path of the URL that allows to access the function.
- The third and fourth row indicates the MIME type of the data that the function consumes from the front-end and produces to the front-end.
- The "Input" section details the format of the consumed data. If the consumed data have "application/json" MIME type, it reports the JSON structure of the consumed object.
- The "Output" section details the format of the produced data. If the produced data have "application/json" MIME type, it reports the JSON structure of the produced Response Object in the case the response has OK status. More precisely, it details the structure of the "response" field of the returned Response Object.

The JSON structures are detailed in Section 4.1.

4.2.1. login

This function is used to authenticate the user.

```
@POST()
@Path("/login")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "login" : string, //email of the user
   "password" : string //user password
}

Output:
{
    "status": OK,
    "errors": [ ],
    "response":
    {
        "userId" : string
    }
}
```

4.2.2. loginrecover

This function sends a mail to the input email value (if exists) with a validation code to reset the user password.

```
@POST()
@Path("/passwordRecover")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "login" : string //email of the user
}

Output:
{
    "status": OK,
    "errors": [ ],
    "response": { }
}
```

4.2.3. passwordreset

If email and validationCode are validated, this function saves the new password for the related user.

```
@POST()
@Path("/passwordReset")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "login" : string, //email of the user
   "validationCode" : string, //the code received through mail
   "newPassword" : string, //the new password
   "rePassword" : string //the new password, again
}

Output:
{
    "status": OK,
    "errors": [ ],
    "response": { }
}
```

4.2.4. logout

This function allows the user to logout from the platform.

```
@POST()
@Path("/logout")
    @Produces("application/json")

Output:
{
    "status": OK,
    "errors": [ ],
    "response": { }
}
```

4.3. Testing Tool Functions

This section reports all the RESTful functions exposed by the Test Administration Suite for Sign Languages to the front-end module. The functions are described in the same format as indicated at the beginning of Section 4.2.

4.3.1. testList

This function returns the list of all the Tests.

```
@GET()
@Path("/Test")
    @Produces("application/json")

Output:
{
    "status": OK,
    "errors": [ ],
    "response": [SMALL_TEST_OBJ]
}
```

4.3.2. getTest

This function returns the Test JSON object that has the id in input.

```
@GET()
@Path("/Test/{TestId}")
    @Produces("application/json")

Output:
{
    "status": "OK",
    "errors": [ ],
    "response": TEST_OBJ
}
```

4.3.3. cloneTest

This function clones, if possible, the Test with the id in input. The response contains a small representation of the cloned Test.

```
@POST()
@Path("/cloneTest")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "testId" : number //id of the Test to clone
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: SMALL_TEST_OBJ
}
```

4.3.4. createTest

This function allows an authorized user to create a new Test. The Test object is returned in the response.

```
@POST()
@Path("/Test")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: TEST_OBJ
}
```

4.3.5. updateTest

This function updates a Test. The id of the Test is passed into the path. The new Test configuration is passed as a JSON object in input.

```
@POST()
@Path("/Test/{testId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
```

```
{
    "test" : TEST_OBJ //the Test to update
}

Output:
{
     "status": “OK”,
     "errors”: [ ],
     "response”: SMALL_TEST_OBJ
}
```

4.3.6. deleteTest

This function deletes the Test with id in input, if the Test exists and the user has the authorization.

```
@DELETE
@Path("/Test/{testId}")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.3.7. questionsList

This function returns the list of Questions related to the Test in input.

```
@GET()
@Path("/question")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "testId":  number //the id of the Test
}

Output:
{
    "status": OK,
    "errors”: [ ],
    "response”: [QUESTION_OBJ]
}
```

4.3.8. getQuestion

This function returns the Question JSON object that has the id in input.

```
@GET()
@Path("/question/{questionId}")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.3.9. createQuestion

This function allows an authorized user to create a new Question. The Question object is returned in the response.

```
@POST()
@Path("/question")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "testId": number //the id of the Test
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.3.10. updateQuestion

This function updates a Question. The id of the Question is passed into the path. The new Question configuration is passed as JSON object in input.

```
@POST()
@Path("/question/{questionId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "question" : QUESTION_OBJ
}
```

```
Output
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.3.11. deleteQuestion

This function deletes the Question with id in the path, if the Question exists and the user has the authorization.

```
@DELETE
@Path("/question/{questionId}")
    @Produces("application/json")

Output
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.3.12. importQuestion

This function imports the Question passed in input to the current Test.

```
@POST()
@Path("/question”)
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "question" : QUESTION_OBJ,
    "testId" : number //the id of the current Test where to import the question
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.3.13. slidesList

This function returns the list of Slides related to the Question in input.

```
@GET()
@Path("/slide")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "questionId" :  string //the id of the question
}

Output:
{
    "status": OK,
    "errors": [ ],
    "response": [SLIDE_OBJ]
}
```

4.3.14. getSlide

This function returns the Slide JSON object that has the id in input.

```
@GET()
@Path("/slide/{slideId}")
    @Produces("application/json")

Output:
{
    "status": "OK",
    "errors": [ ],
     "response": SLIDE_OBJ
}
```

4.3.15. createSlide

This function allows an authorized user to create a new Slide. The Slide object is returned in the response.

```
@POST()
@Path("/slide")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "questionId" : string //the id of the Question to which to add the Slide
}

Output:
{
```

```
    "status": “OK”,
    "errors”: [ ],
    "response”: SLIDE_OBJ
}
```

4.3.16. updateSlide

This function updates a Slide. The id of the Slide is passed into the path. The new Slide configuration is passed as JSON object in input.

```
@POST()
@Path("/slide/{slideId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "slide" : SLIDE_OBJ //the slide to update
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
     "response”: SLIDE_OBJ
}
```

4.3.17. deleteSlide

This function deletes the Slide with id in the path, if the Slide exists and the user has the authorization.

```
@DELETE
@Path("/slide/{slideId}")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.3.18. mediaList

This function returns a list of Media objects, optionally filtered according to suitable input parameters.

```
@GET
@Path(“media”)
```

```
@Consumes("application/json")
@Produces("application/json")
```

Input parameters:

```
{
    (optional) "mediaType" : VIDEO | PHOTO | | AUDIO | | TEXT,    //Filter of media
                                                                  //by type
    (optional) "mediaName" : string,     //used to filter media by name (“like”
                                         //SQL function)
    (optional) "mediaAuthor" : string,  //used to filter media by author name (“like”
                                        //SQL function)
    (optional) "mediaDate" : date, //to filter media by date
    (optional) "mediaTestId" : number //to filter media by Test id
}
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: [MEDIA_OBJ]
}
```

4.3.19. mediaNew

This function saves in the backend file system the file in input and creates a new Media object representing the uploaded file. The response is the JSON representation of the Media object.

```
@POST
@Path(“media”)
  @Consumes("multipart/form-data")
  @Produces("application/json")
```

Input: multipart binary file to upload to the server

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: MEDIA_OBJ
 }
```

4.3.20. mediaDelete

This function allows authorized users to delete the Media object with the id in input.

```
@DELETE
@Path(“media/{mediaId}”)
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
     "response”: { }
}
```

4.3.21. reportList

This function returns the list of Report objects, optionally filtered according to suitable input parameters.

```
@GET
@Path(“report”)
    @Consumes("application/json")
    @Produces("application/json")

Input:

{

    (optional) "reportName" : string,  //used to filter Reports by Test name
                                       //(“like” sql function)
    (optional) "reportDate" : date //to filter Reports by date


Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: [SMALL_REPORT_OBJ]
}
```

4.3.22. reportGet

This function returns the Report object with the id in input.

```
@GET
@Path(“media/{reportId}”)
    @Produces("application/json")


Output:
{
```

```
    "status": “OK”,
    "errors”: [ ],
    "response”: REPORT_OBJ
}
```

4.3.23. reportDelete

This function allows an authorized user to delete the Report object with the id in input.

```
@POST
@Path(“media/{reportId}”)
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.4. Atlas Tool Functions

This section reports all the RESTful functions exposed by the Atlas of Sign Language Suite to the front-end module. The functions are described in the same format as indicated at the beginning of Section 4.2.

4.4.1. testList

This function returns the list of Tests.

```
@GET()
@Path("/atlas/test")
    @Produces("application/json")

Output:
{
    "status": OK,
    "errors”: [ ],
    "response”: [SMALL_TEST_OBJ]
}
```

4.4.2. getTest

This function returns the Test JSON object that has the id in input.

```
@GET()
@Path("/atlas/test/{testId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "complete" : true || false      //if true the response object must be of type
                                    //TEST_OBJ; SMALL_TEST_OBJ otherwise

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: TEST_OBJ or SMALL_TEST_OBJ
}
```

4.4.3. getStandaloneTest

This function returns a zip file containing a JSON file with the test structure, all medias used in the test and a JSON file containing the mappings from media ids to media files.

```
@GET()
@Path("/atlas/test/{testId}")
    @Consumes("application/json")
    @Produces("application/zip")

Input parameters:
{
   "complete" : true || false     //if true the JSON object in the test_{{testId}}.json file
                                  //is of type TEST_OBJ; SMALL_TEST_OBJ
                                  //otherwise
}
```

Output: zip file containing:

- test_{{testId}}.json
- {{mediaName1}}
- {{mediaName2}}
- ...
- {{mediaNameN}}
- media_{{testId}}.json

4.4.4. cloneTest

This function clones, if possible, the Test whit the id in input. The response contains a small representation of the cloned Test.

```
@POST()
@Path("/atlas/cloneTest")
```

```
@Consumes("application/json")
@Produces("application/json")

Input:
{
   "testId" : number //id of the Test to clone
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: SMALL_TEST_OBJ
}
```

4.4.5. createTest

This function allows an authorized user to create a new Test. The Test object is returned in the response.

```
@POST()
@Path("/atlas/test")
    @Produces("application/json")

Output
{
    "status": “OK”,
    "errors”: [ ],
    "response”: TEST_OBJ
}
```

4.4.6. updateTest

This function updates a Test. The id of the Test is passed into the path. The new Test configuration is passed as a JSON object in input.

```
@POST()
@Path("/atlas/test/{testId}")
    @Consumes("application/json")
    @Produces("application/json")

Input parameters:
{
   "test" : TEST_OBJ //the Test to update
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
```

```
    "response": SMALL_TEST_OBJ
}
```

4.4.7. deleteTest

This function deletes the Test object with id in input, if the Test exists and the user has the authorization.

```
@DELETE
@Path("/atlas/test/{testId}")
    @Produces("application/json")

Output:
{
    "status": "OK",
    "errors": [ ],
    "response": { }
}
```

4.4.8. questionsList

This function returns the list of Questions related to the Test in input.

```
@GET()
@Path("/question")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "testId" : number //id of the Test to clone
}

Output:
{
    "status": OK,
    "errors": [ ],
    "response": [QUESTION_OBJ]
}
```

4.4.9. getQuestion

This function returns the Question JSON object that has the id in input.

```
@GET()
@Path("/question/{questionId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
```

```
{
    "complete" : true || false     //if true the type of the response is
                                   //QUESTION_OBJ; SMALL_QUESTION_OBJ
                                   //otherwise
}

Output:
{
     "status": “OK”,
     "errors”: [ ],
     "response”: QUESTION_OBJ or SMALL_QUESTION_OBJ
}
```

4.4.10. createQuestion

This function allows an authorized user to create a new Question. The Question object is returned in the response.

```
@POST()
@Path("/question")
     @Consumes("application/json")
     @Produces("application/json")

Input:
{
    "testId" : number //id of the Test to add the Question to
}
Output:
{
     "status": “OK”,
     "errors”: [ ],
     "response”: QUESTION_OBJ
}
```

4.4.11. updateQuestion

This function updates the Question. The id of the Question is passed into the path. The new Question configuration is passed as a JSON object in input.

```
@POST()
@Path("/question/{questionId}")
     @Consumes("application/json")
     @Produces("application/json")

Input:
{
    "question" : QUESTION_OBJ //the Question to update
}

Output:
```

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.4.12. deleteQuestion

This function deletes the Question with the id in the path, if the Question exists and the user has the authorization.

```
@DELETE
@Path("/question/{questionId}")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.4.13. importQuestion

This function imports a Question in a Test.

```
@POST()
@Path("/question”)
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "questionId" : string, //the id of the Question to clone
   "testId" : number //the id of the Test where to import the Question
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: QUESTION_OBJ
}
```

4.4.14. slideList

This function returns the list of Slides related to the Question in input.

```
@GET()
@Path("/slide")
    @Consumes("application/json")
    @Produces("application/json")
```

Input:
{
"questionId" : string //the id of the Question
}

Output:
{
"status": OK,
"errors": [ ],
"response": [SLIDE_OBJ]
}

4.4.15. getSlide

This function returns the Slide JSON object that has the id in input.

@GET()
@Path("/slide/{slideId}")
@Produces("application/json")

Output:
{
"status": "OK",
"errors": [ ],
"response": SLIDE_OBJ
}

4.4.16. createSlide

This function allows an authorized user to create a new Slide. The Slide object is returned in the response.

@POST()
@Path("/slide")
@Consumes("application/json")
@Produces("application/json")

Input:
{
"questionId" : string //the id of the Question to add the Slide to
}

Output:
{
"status": "OK",
"errors": [ ],
"response": SLIDE_OBJ
}

4.4.17. slideUpdate

This function updates a Slide. The id of the Slide is passed into the path. The new Slide configuration is passed as a JSON object in input.

```
@POST()
@Path("/slide/{slideId}")
    @Consumes("application/json")
    @Produces("application/json")

Input parameters:
{
   "slide" : SLIDE_OBJ //the Slide to update
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: {SLIDE_OBJ}
}
```

4.4.18. deleteSlide

This function deletes the Slide with id in the path, if the Slide exists and the user has the authorization.

```
@DELETE
@Path("/slide/{slideId}")
    @Produces("application/json")

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.4.19. mediaList

This function returns a list of media objects, optionally filtered according to suitable input parameters.

```
@GET
@Path(“/atlas/media”)
    @Consumes("application/json")
    @Produces("application/json")
```

Input:

```
{
    (optional) "mediaType" : VIDEO | | PHOTO | | AUDIO | | TEXT,  //to filter media
                                                                  //by type
    (optional) "mediaName": string,    //used to filter media by name (“like”
                                       //sql function)
    (optional) "mediaAuthor" : string, //used to filter media by author name (“like”
                                       //sql function)
    (optional) "mediaDate" : date, //to filter media by date
    (optional) "mediaTestId" : number //to filter media by Test
}
```

Output:

```
{
     "status": “OK”,
     "errors”: [ ],
     "response”: { [MEDIA_OBJ] }
}
```

4.4.20. mediaNew

This function saves in the back-end file system the file in input and creates a new Media object representing the uploaded file. The response is the JSON representation of the Media object.

```
@POST
@Path(“/atlas/media”)
  @Consumes("multipart/form-data")
  @Produces("application/json")
```

Input: multipart binary file to upload to the server.

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: MEDIA_OBJ
 }
```

4.4.21. mediaDelete

This function allows an authorized user to delete the Media object with the id in input.

```
@DELETE
@Path(“/atlas/media/{mediaId}”)
    @Produces("application/json")
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.4.22. reportList

This function returns the list of Report objects, optionally filtered according to suitable input parameters.

```
@GET
@Path(“/atlas/report”)
    @Consumes("application/json")
    @Produces("application/json")
```

Input:

```
{
    (optional) "reportName" : string,    //used to filter Reports by Test name (“like”
                                         //sql function)
    (optional) "reportDate" : date //to filter Reports by date
}
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: [SMALL_REPORT_OBJ]
}
```

4.4.23. reportGet

This function returns the Report object with the id in input.

```
@GET
@Path(“/atlas/report/{reportId}”)
    @Produces("application/json")
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: REPORT_OBJ
}
```

4.4.24. reportCreation

This function creates a new Report object.

```
@POST()
@Path("/atlas/report”)
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "report": REPORT_OBJ //the Report to create
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: REPORT_OBJ
}
```

4.4.25. reportUpdate

This function updates a Report. The id of the Report is passed into the path. The new Report configuration is passed as a JSON object in input.

```
@POST()
@Path("/atlas/report/{reportId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "report": REPORT_OBJ //the Report to update
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: REPORT_OBJ
}
```

4.4.26. reportDelete

This function allows an authorized user to delete the Report with the id in input.

```
@POST
@Path(“/atlas/report/{reportId}”)
    @Produces("application/json")
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { }
}
```

4.5. Grammar Tool Functions

This section reports all the RESTful functions exposed by the SignGram Blueprint Suite to the front-end module. The functions are described in the same format as indicated at the beginning of Section 4.2.

4.5.1. grammarList

This function returns the list of Grammar objects.

```
@GET
@Path(“/grammar”)
    @Produces("application/json")
```

Output:

```
{
    "status": “OK”,
    "errors”: [ ],
    "response”: [SMALL_GRAMMAR_OBJ]
}
```

4.5.2. grammarGet

This function returns the Grammar object with the id in input.

```
@GET
@Path(“/grammar/{grammarId}”)
    @Produces("application/json")
```

Output:

```
{
```

```
    "status": “OK”,
    "errors”: [ ],
    "response”: GRAMMAR_OBJ
}
```

4.5.3. grammarUpdate

This function updates a Grammar. The id of the Grammar is passed into the path. The new Grammar configuration is passed as a JSON object in input.

```
@POST()
@Path("/grammar/{grammarId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
    "grammar" : GRAMMAR_OBJ //the Grammar to update
}

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: GRAMMAR_OBJ
}
```

4.5.4. grammarPartGet

This function returns the Grammar Part object with the id in input.

```
@GET
@Path(“/grammarpart/{grammarPartId}”)
    @Produces("application/json")


Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: { GRAMMAR_PART_OBJ }
}
```

4.5.5. grammarPartUpdate

This function updates a Grammar Part. The id of the Grammar Part is passed into the path. The new Grammar Part configuration is passed as a JSON object in input.

```
@POST()
```

```
@Path("/grammarpart/{grammarPartId}")
    @Consumes("application/json")
    @Produces("application/json")

Input:
{
   "grammar" : GRAMMAR_PART_OBJ //the Grammar Part to update

Output:
{
    "status": “OK”,
    "errors”: [ ],
    "response”: GRAMMAR_PART_OBJ
}
```

5. Implementation of the SignGram Blueprint

This Section describes in detail the activities following the first release of the SignGram Blueprint as reported in D2.2, and the current implementation in the SIGN-HUB software platform.

5.1. Formstack prototype: Lessons learned

The first implementation of the SignGram Blueprint interface was developed in lack of human resources for development, as discussed during the First periodic review, and it was prepared in the attempt at achieving the T2.1 objective, as stated in the DoA, of having for month 12 "an offline database and a beta version [of the interface] ... in order to be able to incorporate the existing descriptions and data automatically into the online one". This implementation was based on the Formstack platform and online service. Formstack helps in preparing and administering questionnaires through user-friendly forms. Forms are structured as a collection of fields, where each field allows the user to input a piece of information. Among the different types of data fields that Formstack offers there are fields for inputting a person's name and surname, fields for inputting email addresses and dates, fields for uploading video or audio files, and fields for inputting free-form text.

An advantage of Formstack is the fact that the structure of a set of forms can easily mirror the structure of a grammar. A sign language grammar is hierarchically divided in parts, chapters and sections, and correspondingly Formstack allows to organize questionnaires in forms, and groups of fields on a same form in sections. The most prominent disadvantage of Formstack is its limited support for the type of textual information that is found in a sign language grammar. This limitation stems from the fact that Formstack is oriented to producing and administering questionnaires, and that textual answers to questionnaires' questions are mostly short, loosely-correlated chunks of pure text. By contrast, grammar chapters are long, rich sections of text interleaved by pictures, movies and tables. Formstack offers two different types of text fields for the input of free-form text, but none of them support features as, e.g., insertion of cross-references, inclusion of pictures, videos and tables, use of multiple fonts (required for phonology), etc.

Notwithstanding these issues, we developed a very limited Formstack-based prototype so that the content providers of sign language grammars could start experimenting editing structured grammars based on the SignGram Blueprint, and could start gathering grammar data before the development of the WP3 software. To allow content providers to insert pictures, videos, formats and other items that are not supported by Formstack, we envisioned a solution based on the use of special markup text strings that the content providers would have to insert in the body of the grammar, e.g., `<picture filename="...">` to include a picture at some point in the text. This would have required us to equip the planned software platform with a text processor able to interpret the markup text and produce the suitable formatting upon display. It turned out, however, that content providers did not trust the markup-based solution: a correct formatting and alignment of text is essential to render many constructs in a grammar.

position while its restriction stays in the position which corresponds to its grammatical function (the subject position in the following sentence).

wh
BOY BOOK THREE STEAL WHICH
'Which boy stole three books?' (adapted from Cecchetto et al. 2009: 285)

1.2.3.7. Doubling of the *wh*-sign

In LIS, it is possible to find cases where a content interrogative contains two copies of the same wh-sign, as in the following example. The non-manual component can either occur with the wh-signs only, or optionally spread over the whole clause.

wh wh
WHAT YESTERDAY IX$_2$ STEAL WHAT
'What did you steal yesterday?'

When doubling takes place, one wh-sign sits in sentence-initial position while the other

Figure 12 - An example from the LIS grammar

Figure 12 shows an example of these constructs taken from the Italian Sign Language (LIS) grammar included in D2.4. The reproduced text reports the sentences 'Which boy stole three books?' and 'What did you steal yesterday?'. Above these sentences (the *translation tiers*), there are glosses indicating the manual gestures performed by the signer to produce the sentence (the *gloss tier*) and the other non-gestural articulations (the *non-manual tier*). Formatting features as, e.g., alignment of glosses, use of subscripts, underlines and dashed underlines, have precise grammatical meanings that must be preserved when rendering the grammar on-screen. We found that users prefer to have a complete WYSIWYG control on the formatting of grammar text, and do not trust delegating the formatting phase to an external text processor. Moreover, Formstack does not allow users to load as input into a form an arbitrary number of pictures and videos, requiring instead the designer of the form to specify the maximum number of files that can be uploaded. This would have required the content provider to store the pictures and videos on a platform different from Formstack, making the user experience even more unfriendly.

For all these reasons, the content providers rejected the proposed Formstack-based implementation. After a discussion within T2.1 the following solution was adopted:

- The Microsoft Office word processor would be used to produce the first description of the grammars to be delivered at month 24 (D2.4). Microsoft Office is well known to all the content providers, allowing them to be immediately productive.
- The content providers would manually enter the text of the produced grammars through the platform grammar tool starting from its second release. The production of an offline database and a beta version of the SignGram Blueprint interface was consequently deemed unnecessary.
- Thanks to the reschedule of T2.1 activity, the development effort would be partially refocused to give priority to the development of the Atlas and the Testing tool.

5.2. The SignGram Blueprint at release 1

Notwithstanding the rescheduling of T2.1 activities, we were committed to implementing the SignGram Blueprint into the first release of the online Grammar tool as originarily planned. This objective was achieved. As reported in Section 2.1.1, at release 1 the Grammar tool implements the 75% of the requirements for the content providers. Table 2 reports the list of the implemented requirements that are related to the implementation of the SignGram Blueprint in the Grammar tool.

Table 2 - Grammar tool requirements related to the SignGram Blueprint

SGI-RQ-CPR-3	The SignGram Blue Print Module should allow new parts/chapters/section to be added in the grammar of a specific sign language. NOTE: REQUIREMENT CHANGED. The Grammar structure is defined and imported in the platform.
SGI-RQ-CPR-5	The SignGram Blue Print Module should show in the public Table of Content of the specific grammar, messages saying if some part of the ToC has been changed/reorganized
SGI-RQ-CPR-6	The SignGram Blue Print Module should provide an easy to use rich-text content editor
SGI-RQ-CPR-7	The SignGram Blue Print Module should implement a template inside the rich-text content editor
SGI-RQ-CPR-8	The Rich-text content editor should allow the content provider to transfer text and tables on the platform in the following ways: - copy, choose what format (text, title, example, bullet list, etc.) and paste; - copy, paste, choose what format (text, title, example, bullet list, etc.). NOTE: REQUIREMENT CHANGED. The Grammar template is defined by platform to give one user experience. The paste function is implemented and converts the format to the destination template.
SGI-RQ-CPR-10	The Rich-text content editor should cancel gross formatting and superimpose the template formatting, when content providers transfer text and tables on the platform.

The evaluation of the (limited) use of the Formstack prototype affected some design decisions we took about how to implement the SignGram Blueprint in the Grammar tool. The unwillingness of users to adopt an interface based on markup and automatic formatting led to the choice of integrating a WYSIWYG rich text editor in the user interface of the grammar tool and offer to the content providers a user experi-

ence as similar as possible to that of a word processor. The effort necessary to implement in-house a WYSIWYG rich text editor supporting all the formatting features required by the SignGram Blueprint would have easily exceeded the project budget. For this reason, we decided to integrate in the Grammar tool a WYSIWYG rich text editor developed by a third-party, and we started an evaluation of both commercial and open source solutions.

The main disadvantage of a WYSIWYG solution is that it offers a limited array of functionalities for structuring text. The use of a markup syntax would have easily allowed us to define tags with semantic meaning, allowing, e.g., to identify glossed text in the grammar. By contrast, the text structures a WYSIWYG editor are limited to sections, tables, and figures. The solution agreed upon with T1.2 is that the Grammar tool would represent the section structure of the SignGram Blueprint, and that all the other constructs in the Blueprint would be manually rendered by the content providers through the usual formatting tools offered by the editor (boldface, underline, indent, subscript...). This choice reflects the importance of the section structure in the SignGram Blueprint. The Blueprint prescribes a very precise section structure for a sign language grammar, that must be identical to the ToC of the Blueprint book[5].

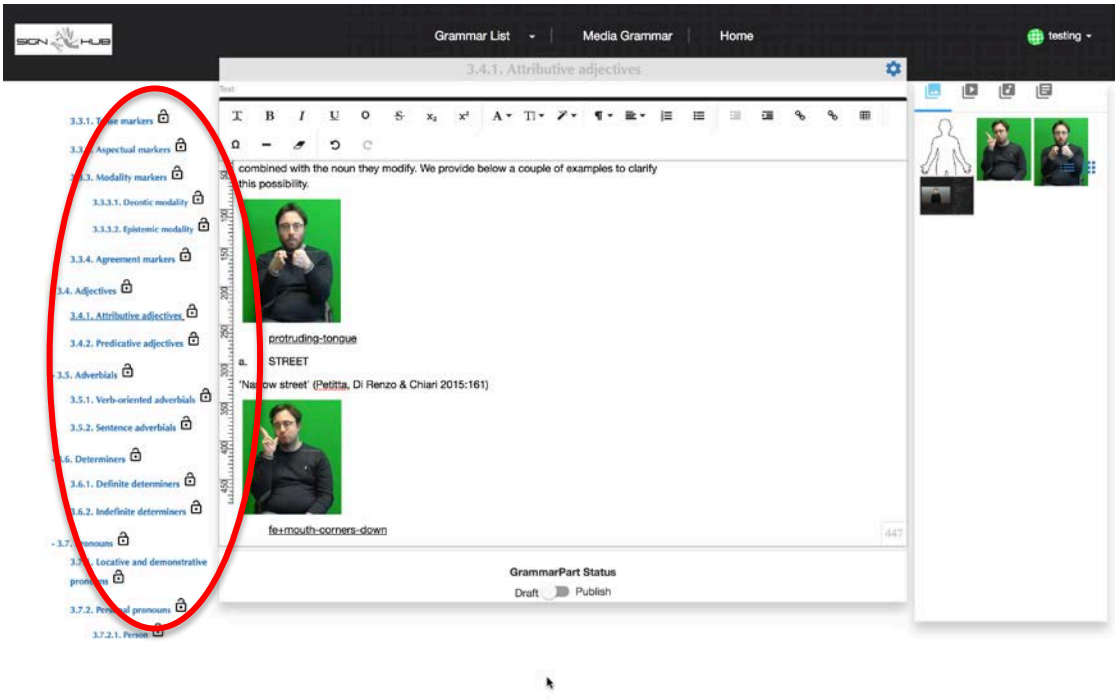

Figure 13 - Grammar tool with interface to SignGram Blueprint

Figure 13 illustrates the interface of the Grammar tool showing a short excerpt from the grammar of Italian sign language (LIS). On the left, the section structure of the SignGram Blueprint is reported as the ToC of the online grammar (enclosed in a red oval). Each section name is clickable, and by clicking it the corresponding section text is loaded in the editor at the center. In the figure, Section 3.4.1 of the grammar (Lexicon – Attributive Adjectives) was selected. The title of the section is also displayed on

[5] *See Josep Quer, Carlo Cecchetto, Caterina Donati, Carlo Geraci, Meltem Kelepir, Roland Pfau, and Markus Steinbach, eds. "SignGram Blueprint: A Guide to Sign Language Grammar Writing", De Gruyter Mouton.*

top of the editor. The editor also shows two examples of glossed text. The appearance was obtained with the editing instruments offered by the embedded WYSIWYG editor. By comparison, Figure 14 shows the same excerpt in the original Word file.

Figure 14 – Section 3.4.1 of the LIS grammar (excerpt)

In conclusion, we feel that the release 1 of the Grammar tool has achieved the objective of adequately representing the structure of the SignGram Blueprint in a way that non-intrusively guides the content providers in inserting grammar text. We maintain that the compromise chosen of constraining the organization of the text in sections, and allowing the users to enter free-form text structures, will meet the T2.1 researchers requirement of working with a familiar and well-known user interface.