Project Number: 693349

Interface for content creators

Giuseppe Airò Farulla, Pietro Braione, Marco Indaco, Mauro Pezzè, Valentina Porzio

CINI, Aster SpA

Version 1.4 – 15/11/2017

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: 3.4

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	18.05.2017	Initial version.	Paolo Prinetto, Josep Quer, Jordina Sánchez Amat
1.1	21.06.2017	History of changes added. Section 2, General architecture and Work Plan, added. Section 5, Architectural Design of the Internal Interfaces, added.	Paolo Prinetto, Josep Quer, Jordina Sánchez Amat
1.2	15.10.2017	Document completely revised w.r.t. its original structure. Added Appendix A: Sign Language Test Suite.	Paolo Prinetto, Josep Quer, Jordina Sánchez Amat
1.3	18.10.2017	Methodology and Software Suites sections revised	Paolo Prinetto, Josep Quer, Jordina Sánchez Amat
1.4	15.11.2017	Document completely revised w.r.t. its original structure. In particular, hereby follows the most important changes made: • Overall description of the motivations behind the work done for this deliverable that result to be different, in contents, from what originally expected. • Overall description of the adopted methodology reporting how the agile methodology is used in the project. • Clarification of the interactions between WP3 and WP2. • Reported the interaction between WP3 team especially between who define technical specification and who implement the platform. • Removed the confusion on architectures, technologies, components not strictly related to the objectives of the project. • Added Section 2 describing the overall methodology used for designing and developing the whole platform including general project decision. • Old Section 2 has been completely revised moving the essential description of the whole system in new Section 3. • Old Section 4 and 5 have been completely revised and merged in new Section 4 where a more consistent architectural description is provided w.r.t. project objectives and technological choices are here motivated. • Old Section 5 has been completely revised w.r.t. the new sign hub architectural design and the content has been moved to the Appendix A. • Renamed Appendix A Sign Language Test Suite in Distribution Suite for Sign Languages.	Pietro Braione, Mauro Pezzè, Jordina Sánchez Amat

INDEX

Scope of the document

Originally, according to the project proposal, this deliverable should have contained the description of the implemented software graphical user interfaces enabling content creators (i.e., scientists, researchers) to populate the platform with data resulting from the work of WP2 and WP4 teams. Nevertheless, this goal could not be addressed from WP3 team at month 10 mostly for two reasons:

- When WP3 team delivered D3.1 we started thinking about the overall design of the platform, figuring out that the software modules required to implement the interfaces of content creators and interfaces for content providers showed a wide overlap from functional perspective. For this reason, we took the decision that it was not convenient starting developing the interfaces for content creators without having firmly defined a high level architectural design of the SIGN-HUB web platform in its entirely, involving, thus, also the software modules required to implement the content providers interfaces (Task 3.6, Task 3.7, Task 3.8, Task 3.9, Task 3.10).
- The software development team was formally involved within the project at month 13 causing a delay in WP3 activities as already explained and motivated in the annual technical report. Therefore, it was unfeasible to develop, test, and deliver the interfaces for content creators in only 1 month.

For these reasons, WP3 team decided, in accord with SIGN-HUB partners, to modify the rationale of this deliverable so to describe a first-level design of the web platform (while the design of the graphical user interface is out of the scope of this deliverable) that must be intended as a track for developers whenever they will start implementation activities. The technological choices are also part of this deliverable.

The implementation activities will follow an agile methodology and so the design could potentially be changed, resulting in further releases of this deliverable.

Summarizing, this deliverable presents:

- A high-level description of the software architecture (i.e., modules and interfaces) of the back-end modules of the SIGN-HUB web platform;
- The definition of the software suites that will compose the SIGN-HUB web platform;
- The description of the interconnections among software modules needed to accomplish software suites functionalities.

1. Introduction

SIGN-HUB has the ambitious goal to preserve the historical and cultural heritage of Deaf communities in Europe through the digitalization of linguistic and cultural resources (e.g., creation of grammars, creation of assessment tools for sign languages, digitalization of old multimedia content). For this purpose, an open and technological web platform will be developed to provide a collection of valuable tools to keep and convey digital assets to scientists, deaf people, signers' communities, and more generally communities with no technological expertise.

SIGN-HUB web platform will be composed of 4 tools (i.e., Software Suites), seamless 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 SignGram Blueprint**: an online grammar writing tool to produce digital grammars of 6 sign languages;
- **The Atlas of Sign Language**: an interactive digital atlas of linguistic structures of the world's sign languages;
- **The Test Administration Suite for Sign Languages:** an online sign language assessment instruments for education and clinical intervention;
- **The Digital Archive of Old Signers' Linguistic and Cultural Heritage**: a Digital archive of life narratives by elderly signers, subtitled and partially annotated for linguistic properties.

From a technological perspective, each aforementioned tool is composed of software modules that, interacting among each other, accomplish the user requirements. Generally speaking, when dealing with web applications, software modules are classified in mainly two sets: 1) front-end modules and 2) back-end modules that are seamless integrated to perform user requests:

- Front-end module is about user interfaces and the related controllers to translate user inputs in web requests for a back-end module.
- Back-end modules are about the application logic and the capability to persistently store or retrieve data.

This deliverable deals with an overall description of the back-end design, while the front-end, and so the user interface, will be separately described in dedicated deliverables that will be released contextually with the releases of the aforementioned software tools.

The structure of the deliverable is the following:

- Section 2 *Methodology*: describes the methodology behind the work done for this deliverable by WP3 team describing the fundamental decisions that were taken to design the back-bone of the interface to meet the users' requirements at desiderata at the best;
- Section 3 *SIGN-HUB Platform Overview*: describes the operational context of the platform, focusing on the external systems the platform will interact with, and the actors that will use the platform.
- Section 4 *Software Architectural Design*: describes the fundamental modules composing the platform, focusing on their components and interfaces. The purpose of each module and component is here reported especially targeting their role with respect to the functionalities required by the project, the final users and stakeholders. Finally, a view about the suites of the project is presented, highlighting the functional dependency among modules involved in each suite.

- Section 5 *Conclusion*: drives briefly the conclusion of the deliverable.
- Annexes: contain the back-end software interface specification with input/output parameters description.

2. Methodology

The work has been carried out by WP3 team involving software developers (i.e., Eclettica as third party) and software architects from CINI. WP3 had regular meetings (almost weekly) to refine the design of the software platform architecture.

WP3 identified 4 main activities regarding the SIGN-HUB Web platform:

- Back-end design: preliminary software architecture design to define main modules and their functional relations.
- Front-end design: design of the graphical user interface and definition of the user experience.
- Back-end implementation: Implementation of the software modules and components.
- Front-end Implementation: Implementation of the graphical user interface.

In this Deliverable, WP3 team reports on the results of the activity related to the back-end design. For this task, the agile methodology was not adopted. The reason being that we have firstly chosen to provide a macro level design of the platform intended as a guideline for the development team. The preliminary interactions with WP2 that led to the deliverable D3.1 has been considered sufficient to design the back-end of the platform.

The undertaken methodology for back-end design is here summarized:

1. Architects deeply analyzed D3.1 to elicit all the functional blocks and services to be exposed to the front-end modules. For each user requirement, WP3 defined a list of software functions (e.g., functions to create a test, functions to import media file within a test), no worrying about repetition or logical overlaps among functions.
2. Architects focused on the overlapping software functions. Logical overlaps are fixed by deleting repeated functions.
3. Architects grouped software functions that accomplish similar tasks in software modules. Similar software modules are combined.
4. For each software module, the interfaces and their responsibilities are defined. Interfaces make available module functionalities. The specific input and output parameters are not defined at this stage.
5. Developers were involved in this phase to assess the software architecture with respect to the user requirements. Feedbacks were collected and applied.
6. Finally, developers and architects selected the best set of technologies to implement the web platform.

The design of the front-end and the overall platform development will be pursed in accord with SCRUM agile methodology as performing fast platform prototyping and collecting user feedbacks about user experience and GUI design at each agile cycle (i.e., 3 weeks) is considered a strategic asset. The expected outcomes of these 3 activities will be provided in D3.6, D3.7, D3.8, D3.9, D3.10.

The undertaken scrum methodology will be composed of the following steps:

1. From the prioritized requirement list presented in D3.1, developers and architects select a set of user requirements to fill the product backlog.
2. Selected user requirements are decomposed in technical tasks, and the sprint activities are planned.
3. WP3 team starts working for 3 weeks on the technical tasks that can involve: the graphical design of a functionality, the class diagram design of a back-end compo-

nent, the development of a piece of software, the test of the developed functionalities.

4. The software prototype is delivered. WP2 team and a dedicated focus group are involved in this phase to interact with the delivered prototype and to provide useful feedbacks and suggestions to WP3 team.
5. WP3 team analyzes the received feedbacks and reviews the work done during the sprint and updates, if required, the list of user requirement (i.e., D3.1) and the product backlog.

3. SIGN-HUB Platform Overview

The description of the SIGN-HUB Web Platform will be accomplished by firstly describing the operational context of the platform (Section 3.1) defining the external systems the platform will interact with and the actors who are expected to use the platform.

3.1. Operational Context

The operational context of the SIGN-HUB web platform is illustrated in Figure 1. In the upper part of the picture are grouped the *actors*, i.e., the end users that will use the platform. On the right side are represented the external systems the platform will interact with exchanging data.

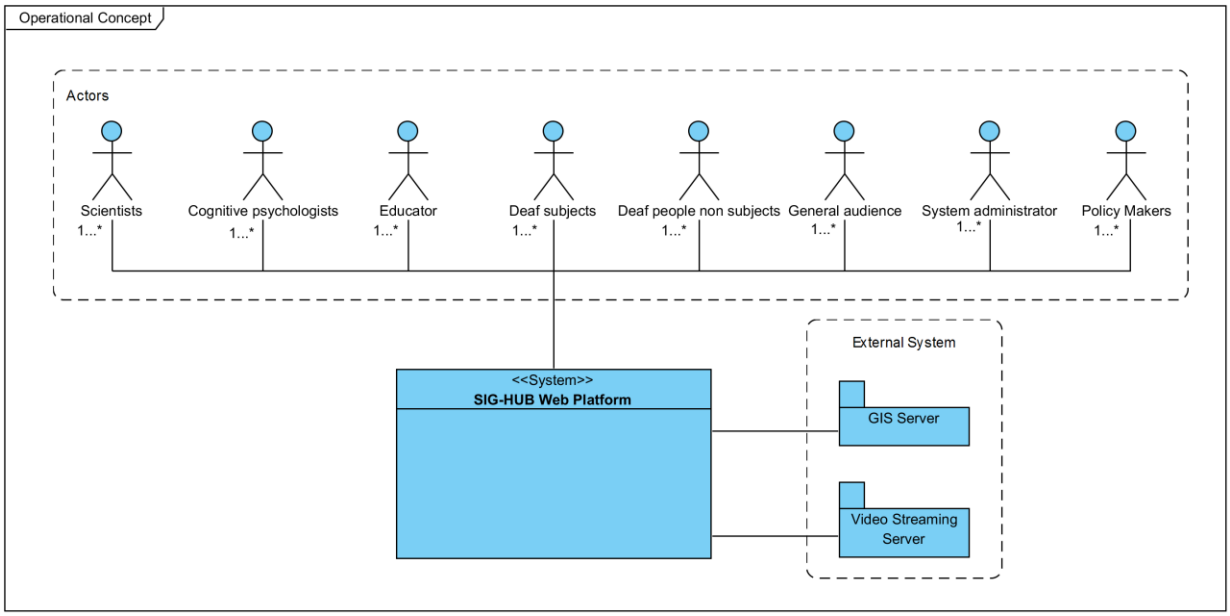

Figure 1: SIGN-HUB Web Platform Operational Context

3.1.1. Actors

An actor is considered a generic person who interacts with the SIGN-HUB platform. Typically, the Actors are identified as end users of the platform but in this case, we have also considered as actor the administrator of the platforms. The actors of the platform are the following users (as in deliverable D3.1):

- Scientists
- Cognitive psychologists
- Educator
- Deaf subjects
- Deaf people non-subjects
- General audience
- System administrator
- Policy makers.

3.1.2. External Systems

An external system is a system that provides specific functionalities and interacts with the platform by sending or retrieving data. A system is considered as "external", when there is no need to implement it as a part of a software module of the platform as it is already available as commercial or open source solution. Afterward the user requirement analysis, WP3 team identified the need to have a map provider system for the ATLAS tool and a video streaming server where hosting video contents that must be streamed to the SIGN-HUB web platform. Caused by their functional complexity, there is no need to implement them from scratch especially because relevant open source solutions already exist.

The list of the external systems is reported as follows (they are better introduced and reviewed in deliverables D3.2 and D3.11):

- **GIS Server**: The GIS Server is a module that exposes a digital base map by means of a protocol named Web Map Server (WMS). The WMS is a web service that allows end-users to gather a map tile. Usually a digital map is divided in small tiles in order to minimize both data transfer and rendering time. WMS accepts the coordinates of a single point as input (e.g., Latitude and Longitude) and It provides as output the tile the point is localized in. Open street map[3] is an open source map provider and it is the best candidate to be integrated within the SIGN-HUB web platform. Digital maps will be mainly integrated in the Atlas tool.
- **Video Streaming Server**: it is an external service able to store video contents in several formats and to distribute them on-demand over internet. Actual streaming servers are able to automatically adapt data rate according to the specific device (i.e., mobile or desktop) which is requiring the access to a video content. Moreover, it is highly flexible to scale w.r.t the variable number of connected users. Both Grammar and Digital Archive tools will integrate external video sources.

[3] *https://www.openstreetmap.org*

4. Software Architectural Design

The design of SIGN-HUB web platform will be described following the UML 2.0[4] approach.

UML is a modeling language used to specify, display, construct and document artifacts of a software system. It is used to understand, design, configure, maintain, and control information associated with such systems. This language aims to unify past experiences of modeling techniques and incorporate new software trends according to a standard approach.

UML modeling tool uses stereotypes (mechanisms of controlled extensions) that allow it to adapt to the domain of interest. The UML is able to represent the static structure and dynamic behavior of a software system. A software system is modeled as a collection of discrete objects that interact to perform features that are ultimately exploited by an external user.

Modeling a system through different, but connected, points of view allows to understand the system for different purposes.

The software tool, that will be used in this document to describe the UML model of the SIGN-HUB web platform, is Visual Paradigm[5]. The set of UML stereotypes used within the diagrams presented in this deliverable is:

- ***Component***: It is an abstract design element that hides its implementation behind a set of interfaces.
- ***Interface:*** it describes the features and operations a component makes available to other components.
- ***Module:*** it is modeled as a collection of components, and they are usually associated with a real and concrete piece of software (e.g., executable, library, web service).
- ***Platform*****:** it is generally defined as a collection of Modules hat describe a complete system view.
- ***Suite***: is a product or tool with well-defined functional boundaries from end-user perspective. SIGN-HUB web platform is composed of 4 Suites (i.e., tools) that are Grammar Writing Tool, ATLAS Tool, Test Tool, Digital Archive Tool. Each Suite is composed of modules.

[4] *http://www.omg.org/spec/UML/2.0/*

[5] *https://www.visual-paradigm.com/*

4.1. SIGN-HUB Software Modules

The activities described in Section 2 resulted in defining a set of modules that will compose the software architecture of SIGN-HUB web platform (see Figure 2: Software modules of the SIGN-HUB web platform). For sake of clarity, they are classified with respect to their functionalities, assigning to modules sharing the same functional domain a specific color.

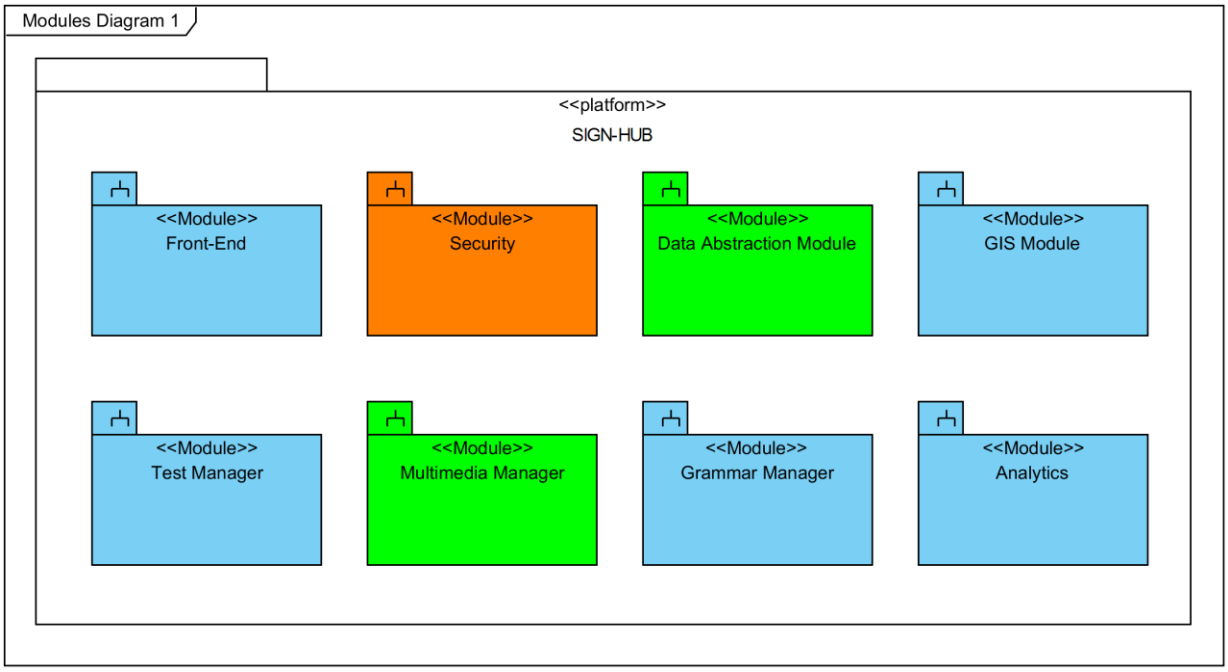

Figure 2: Software modules of the SIGN-HUB web platform

Table 1: Software Module Classification

Type	Description	Color Code
Application Modules	Core modules that provide the key functionalities of the platform, specified by the functional requirements.	
Data Providers Modules	Modules that manage, ingest and store all data inserted by the actors.	
Utility Module	Module features functionalities which are exploited to satisfy non-functional requirements.	

Each of the modules contained in Figure 2 is described in detail in the next paragraphs.

4.1.1. Front-end Module

The "Front-end" Module can be considered, from the user point of view, the entry point for the SIGN-HUB web application and its services and functionalities. It implements all the views (i.e., graphical user interfaces) of the platform and so all the user interfaces required by the 4 Suites. Additionally, it implements all the controllers which have the role to transform the user input in a command (or in a sequence of commands) to the back-end of the platform. The controllers have also to check the consistency of the data format insert by the user, and if it is not compliant with the expected one, a pop-up with an error message is showed to the user.

As it is stated in Section Methodology2, the design of the graphical user interfaces and, consequently, of the user experience is out of the scope of this deliverable. Such activities will be performed, according to the agile methodology, whenever the tasks for implementing the suites will start.

The "Front-End" Module (see Figure 3) is composed of 5 components: the "HMI" and 4 "Controllers".

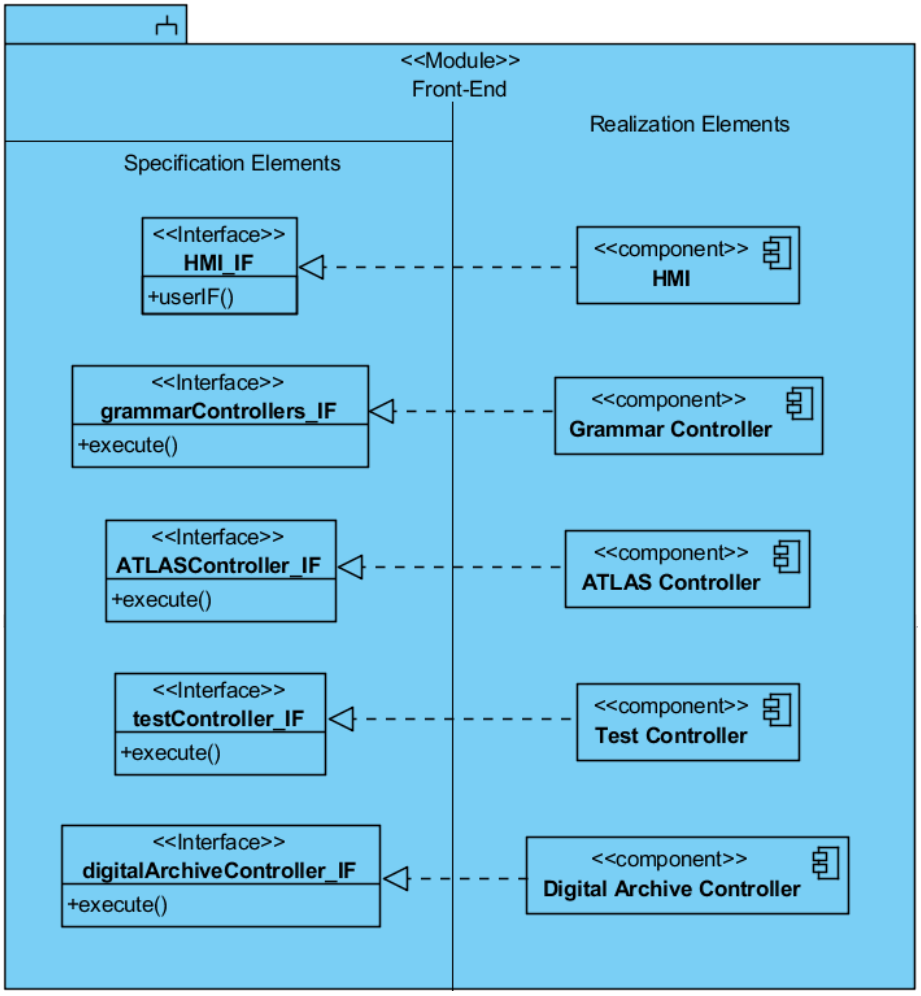

Figure 3: The Front-End Module

The "HMI" component is the collection of all the GUIs required by the four Suites. It provides a single and generic interface "HMI_IF" that can be considered as the full set of graphical element enabling a generic actor to have access to all the services exposed by the platform.

The "Controllers" component have to check and process user commands. It translates user interactions with the graphical objects and widgets in specific commands to the platform services offered by the other modules. The "Grammar Controller" manages the data flow resulting from an interaction between the end users and Grammar Suite. In this specific case, platform can generate notifications which are managed by this controller when, for instance, a user adds a new section in an existing grammar. The "ATLAS controller" manages all the interactions between end users and Atlas Suite (e.g., map interaction), the "Test Controller" manages the interactions between the user and the Suite for Testing for instance for the creation of a new test or for visualize the test results. Finally, the "Digital Archive Controller" manages the interaction between the user and the Digital Archive Suite for instance when the user selects specific subtitles for a video or when metadata are required.

Each controller has its specific interface. The name of the interfaces is made up by adding the suffix IF to the component's name (e.g., testController_IF). These interfaces are invoked when an actor interacts with a graphical widget.

4.1.2. Security Module

The "Security Module" (see Figure 4) aims at providing the functionalities for the overall security of the platform. Specifically, it guarantees data security and access control. The UML Diagram is composed of a single component "AAA". Triple A stands for authentication, authorization, accounting.

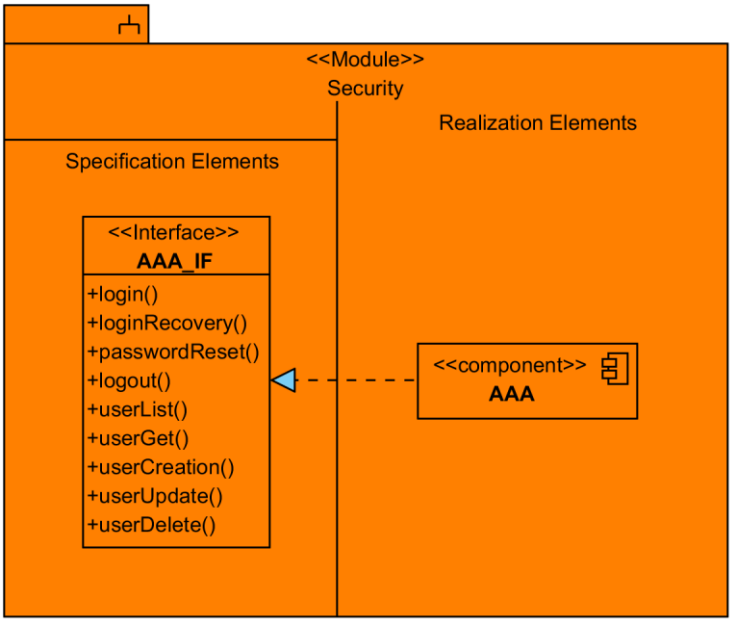

Figure 4: The Security Module

The "AAA" realizes a single interface "AAA_IF" which is commonly called by the "Front-End Module". The provided functions are:

- *login():* This function is used to authenticate the user.
- *loginRecovery():* This function sends an email to the input email value (if exists) with a validation code to reset the user password.
- *passwordReset():* If email and validationCode are validated, this function saves the new password for the specific user.
- *logout():* This function allows the user to logout from the platform.
- *userList():*This function returns the list of users; could be used some input filters.

- *userGet():*This function returns a specific user given an id as input.
- *userCreation():* This function creates a new user.
- *userUpdate():* This function updates the user configuration establishing user permission rights. For instance, a user could access to The Sign Language Assessment Suite only for answering to a specific test without the grants to create a new one.
- *userDelete():* This function allows administrator to delete a specific user.

4.1.3. Test Manager Module

This module (see Figure 5) exposes both the functionalities required by a researcher to build an experiment composed of different types of questions and the functionalities required by a subject under test to perform the test itself. It manages all the primitives required to create a test. Each test is composed of a set of stimulus and a set of answers. Stimuli and answers can be constructed by using different type of data such as images, video or text and different kind of controls such as check boxes and multiple choices. Moreover, this module allows the researcher to revise the collected answers for a given test before saving them permanently. This is especially important for the ATLAS Sign Language suite where the collected answers are stored in the database and used for generating data to be shown on the digital map. This module is used in two suites: The Test Administration Suite for Sign Languages and the ATLAS Sign Language suite.

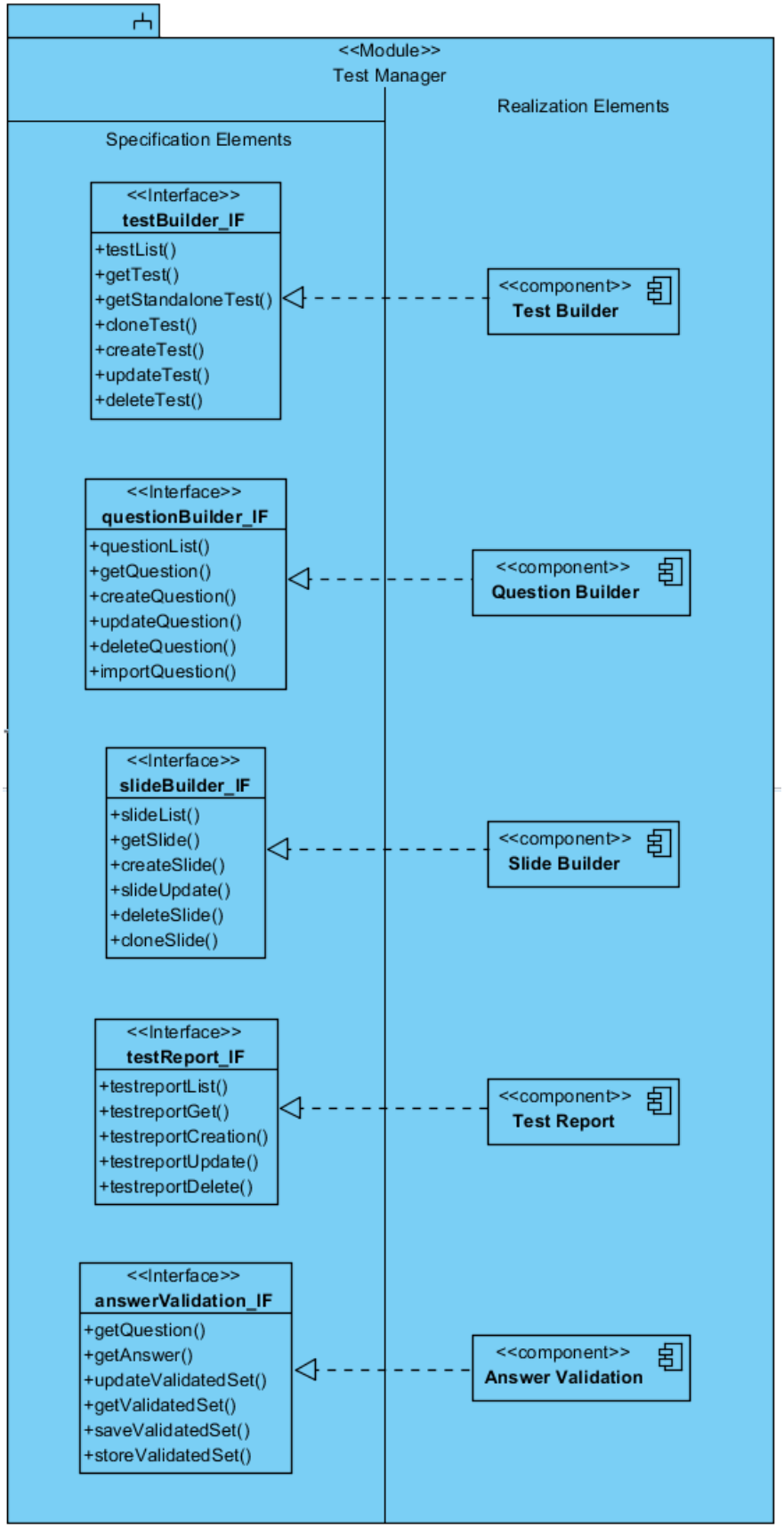

Figure 5: The Test Manager Module

The UML components defined for this module are:

- "Test Builder Component": it is responsible for managing all the operations related to create, clone or delete a test. A test is composed of questions and questions are composed of slides.
- "Question Builder Component": it manages all the operations required to create a question. A question can be composed of one or more slides. For example, a first slide can be a stimulus that must be shown on the screen for 10 seconds and a second slide can be an answer (e.g., check box) that is shown on the screen until the user interacts with it.
- "Slide Builder Component": it is the component that manages the atomic unit of a test. It allows user to create, delete or to clone a slide in another test.
- "Test Report Component": It serves as report generator. When a subject under test finishes to answer to all the provided questions, a report is automatically created and saved on the database. The researcher can thus access to it to analyze the test.
- "Answer Validation Component": this component is used in the "ATLAS of Sign Language Suite" to revise the answers before these are permanently stored. Validation procedure has the purpose to check and validate the ATLAS database population.

The interfaces provided by these components allow the specific controller defined in the "Front End Module" to exploit the functionalities provided by this module. The defined interfaces are:

- "testBuilder_IF": it is the interface exposed by the "Test Builder Component". It is invoked by the "Front-End Module" for the creation of a new test or for managing an existing test. The operations of this interface are:
    - *testList()*: This function returns the list of the available tests. If a test is tagged as private, only the user who created it can open it.
    - *getList()*: This function returns a specific test composed of its questions and slides.
    - *getStandaloneTest()*: this function returns a zip file containing the test structure and all medias used in the test. This can be used when the location where the test must be done has not internet available. In this case the test can be downloaded to be answered in offline mode.
    - *cloneTest()*: this function clones a test.
    - *createTest()*: this function allows authorized user to create a new Test.
    - *updateTest()*: this function updates a specific test.
    - *deleteTest()*: this function deletes a specific test, if the test exists and the user has the proper authorization.
- "questionBuilder_IF": it is the interface exposed by the "Question Builder Component". It is invoked by the "Front-end Module", when the user is creating a test and he wants to create a new question. The defined operations are:
    - *questionList()*: this function returns the list of questions related to the selected Test.
    - *getQuestion()*: this function returns the selected question object.
    - *createQuestion()*: this function allows authorized user to create a new question.
    - *updateQuestion()*: this function updates the question and the related configuration (e.g., the number of slides question is composed of).

  - *deleteQuestion()*: this function deletes the selected question, if the question exists and the user has the authorization.
  - *importQuestion()*: this function imports the question passed in input to the current Test.

- "slideBuilder_IF": it is the interface exposed by the "Slide Builder Component". It is invoked by the "Front-End Module" when the user is interacting with the Test Administration Suite for Sign Languages to create/modify/delete a slide. The defined operations are:
  - *slideList():* this function returns the list of slides related to the question in input.
  - *getSlide():* this function returns the selected slide.
  - *createSlide():* this function allows authorized user to create a new slide.
  - *slideUpdate():* this function updates the slide when the user adds a new stimulus or a new answer.
  - *deleteSlide():* this function deletes the question with id in the path, if the question exists and the user has the authorization.
  - cl*oneSlide():* this function clones the slide selected by the user in the current question.
- "testReport_IF": it is the interface exposed by the "Test Report Component" to generate the report for a selected test. The report is made available to the user as CSV file. If needed, it can be downloaded. The defined operations are:
  - *testReportList():* this function returns the list of report object. The user can apply also a temporal filtering.
  - *testReportGet():* this function returns the report for a selected test.
  - *testReportCreation():* this function creates a new report.
  - *testReportUpdate():* this function updates the selected report.
  - *testReportDelete():* this function allows authorized user to delete the selected report.
- *"answerValidation_IF":* it is the interface exposed by the "Answer Validation Component". This interface is invoked by the "Front-End Module" when the user is expected to validate the test answers before they get permanently store in the database to populate the data analytics for the ATLAS of Sign Language suite. The defined operations are:
  - *getQuestion():* this function returns the questions defined for a specific test.
  - *getAnswer():* this function returns the answers given for a specific test.
  - *updateValidatedSet():* this function updates the validation state for one or more answers.
  - *getValidatedSet():* this function returns the validation state for the set of answers contained in a specific test.
  - *saveValidatedSet():* this function saves in draft the answer validation state performed by the user for a specific test.
  - *storeValidatedSet():* this function stores permanently the answer validation state performed by the user for a specific test.

4.1.4. Grammar Module

This module (see Figure 6) exposes the functionalities to create specific and formatted text required by researchers to upload a grammar on the web platform. It implements functions of a rich-text editor augmented with custom functionalities required to define specific styles and to link, when needed, specific words to thesaurus. Moreover, it allows end-user to navigate the grammar providing all the custom search functions to easily jump to the section (or test) of interest.

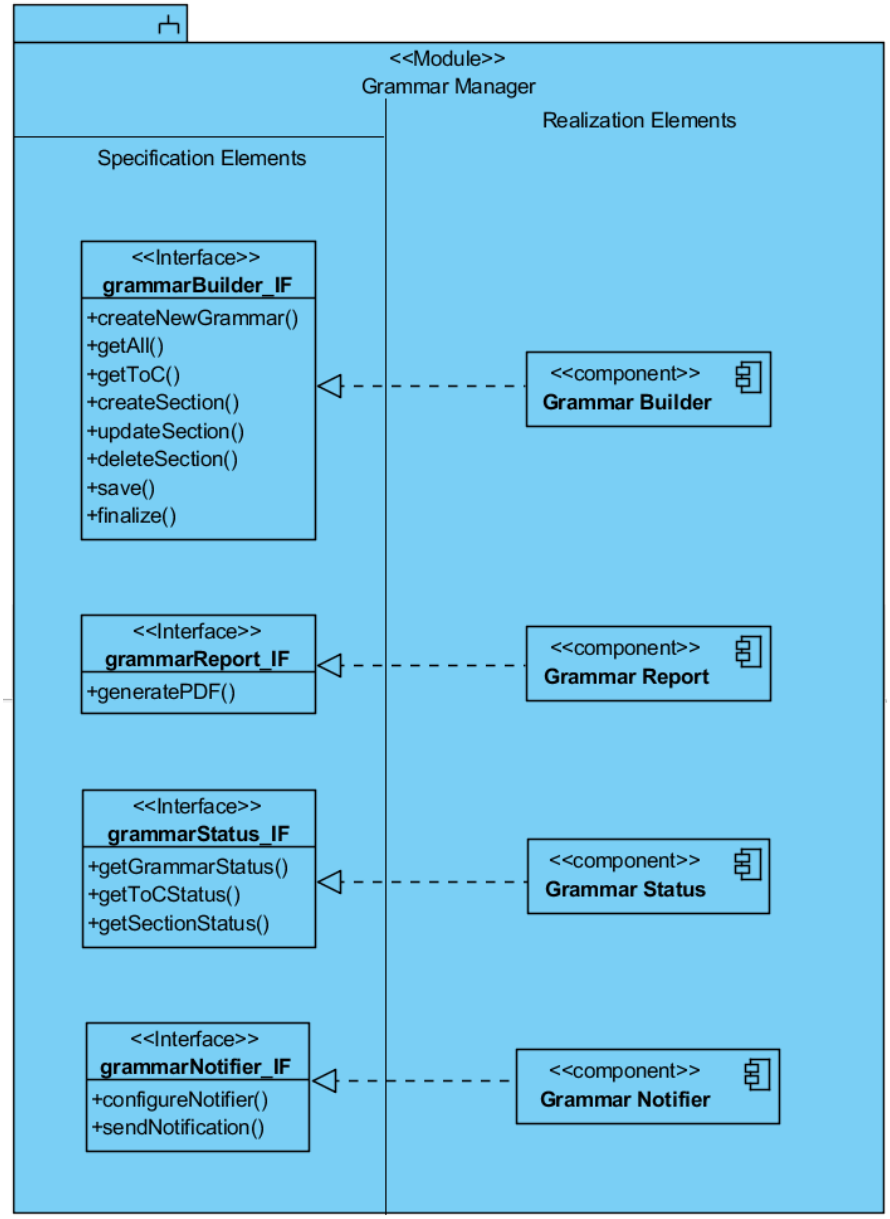

Figure 6: The Grammar Manager Module

The UML components defined for this module are:

- "Grammar Builder Component": It allows the creation of a new grammar and to extend or modify an existing one exporting services to support the most recent WYSIWYG editors. This component manages the grammar object supporting all editor features: formatting text, adding rich text elements such as captioned images, code snippets, tables, mathematical formulas, working with text styling, adding lists and bullet points. Supporting multi-collaborative grammar writing, this component

sends a notification to the researcher that created the grammar, if another user has modified or added a section to the same grammar.

- "Grammar Report Component": it generates a PDF of a specific grammar so that users can download it and read it offline.
- "Grammar Status Component": it is responsible for the current status of a grammar. It has all the information to assess the number of sections still open before a grammar can be considered completely compiled or the state of a single section saved in draft or finished.
- "Grammar Notifier Component": it is responsible to send a notification to the owner of a specific grammar, if the grammar itself has been modified. This component can be opportunely configured with a specific set of rules to trigger the required actions when a grammar update occurs. Within the rule set the following parameters can be configured: email address of the grammar owner, content of the email message, if the owner is expected to accept or reject the modification.

The interfaces provided by these components allow the specific controller defined in the "Front End Module" to exploit the functionalities provided by this module. The defined interfaces are:

- "grammarBuilder_IF": it is the interface composed of all the operations to, create modify or delete a grammar (or a section). The defined operations are:
  - *createNewGramma():* this function allows user to create a new grammar for a specific sign language. The user become the owner of the grammar and so it will be notified when another granted user modifies the content of the grammar.
  - *getAll():* this function returns all the content of a specific grammars, required by the user.
  - *getToC():* this function returns the defined Table of Content given a specific grammar.
  - *createSection():* this function allows the user to create a new section of the grammar.
  - *updateSection():* this function allows the user to modify the content of the grammar, supporting all rich-text editor features.
  - *deleteSection():* this function allows the user to delete a section.
  - *save():* this function allows the user to save in draft a section of the grammars.
  - *finalize():* this function allows the user to set as complete the content of a section.
- "grammarReport_IF": it is the interface that must be invoked when user requires to download the grammar in PDF format. The define operation is:
  - *generatePDF():* this function generates automatically the PDF of the grammar.
- "*grammarStatus_IF*": this interface exposes all the operations required to assess the completion status of a selected grammar. Completion states for: the number of sections\sub sections\paragraphs composing the ToC that have been finalized or just saved as draft. The defined operations are:
  - *getGrammarStatus():* this function returns the overall completion status of a specific grammar.
  - *getToCStatus():* this function returns the completion status of a portion of the ToC for a specific grammar.

- *getSectionStatus():* this function returns the completion status of a specific grammar section/sub section/paragraph.

- "grammarNotifier_IF": this interface exposes all the operations to configure and send notifications when grammar state is updated. The defined operations are:
  - *configureNotifier():* this function allows the creator of a grammar to configure the email address where he wants receive the notification, the content of the email, the sections/sub-sections/paragraphs for which the user wants receive a notification.
  - *sendNotification():* this function is automatically invoked by the "Grammar Builder Component" when a sections/sub-sections/paragraphs of the grammar is saved or finalized.

4.1.5. GIS Module

This module (see Figure 7) provides the functionalities to properly process, store and geo-localize over a web-GIS grammar rules related to the different sign languages. This module provides a dedicated web-GIS component to allow end-user to navigate on the world map to retrieve the detailed information of interest about a specific sign language. Moreover, this module allows also to automatically compute the query performed by the user, classifying and clustering the obtained results from the database. For instance, the user requires to visualize geo-data about a specific feature (e.g., Consonant). The module is able to elaborate the request, to enumerate the number of feature per indexed sign language and to properly generate the resulting layer associated with a color-based legend. All the functionalities required to only visualize maps and layers are implemented in the Front-end Module.

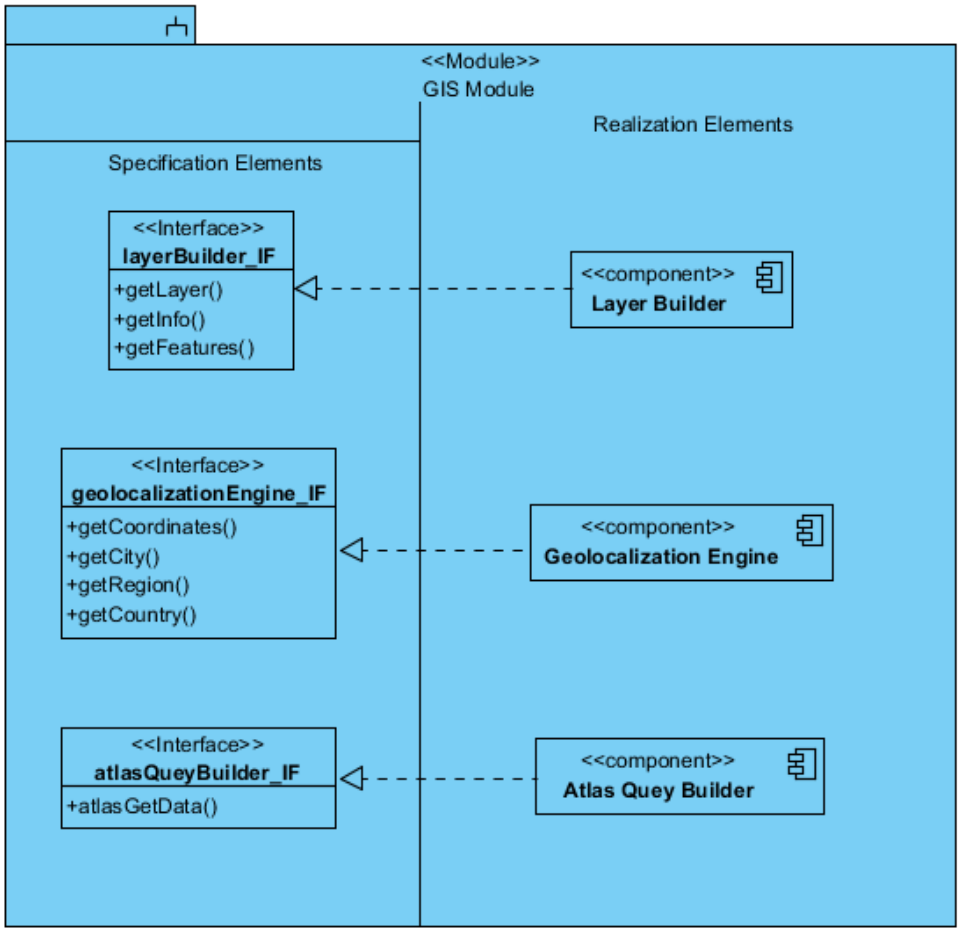

Figure 7: The GIS Module

The UML components defined for this module are:

- "Layer Builder Component": after that user performed a query on a specific set of linguistic features, this component is able to build the layer (i.e., shape file) to be visualized. Moreover, when user clicks on a geo-localized sign-language to obtain additional info on the specific feature, this component returns data of interest (e.g., value associated to the selected sign language).
- "Geolocalization Engine": this engine computes which is the latitude and longitude for the specific graphical element composing the layer. Moreover, it allows to calculate the country or the region the geographical element is part of.
- "Atlas Query Builder": this is a sophisticated engine that allows user to build complex query composed of multiple features. Features can be linked between each other with an AND/OR logic. Furthermore, this component allows user to perform a further spatial and temporal filter on the resulting data.

The interfaces provided by these components allow the specific controller defined in the "Front End Module" to exploit the functionalities provided by this module. The defined interfaces are:

- *"layerBuilder_IF":* this interface exposes all the operations required to build a vectorial layer that must be loaded on a GIS viewer. The defined operations are:
  - *getLayer():* this operation returns the layer composed of geo-localized entities.
  - *getInfo():* this operation returns info associated to the specific layer: generation time, the name of the layer and the combination of features composing the query done by the user.
  - *getFeatures():* each layer is composed of several entities (i.e., sign language) that can be clicked by the user to have more information about them. This function returns all the information related to the specific entity. For example, such information can be: a link to the section of the grammar where the searched feature is explained, the number of feature.
- "*geolocalizationEngine_IF":* this interface exposes all the operations to calculate the geographical coordinates for each entity composing the layer. The defined operations are:
  - *getCoordinates():* this operation returns latitude and longitude for a specific entity, if an entity has been stored in the database with the coordinates.
  - *getCity():* this operation returns the coordinates of the closest city to a specified entity.
  - *getRegion():* this operations returns the coordinates of the center of the region the entity is part of.
  - *getCountry(): this operations returns the coordinates of the center of the country the entity is part of.*
- "*atlasQueryBuilder_IF":* this interface exposes the operation to allow user to perform a query about the number of features present in the stored sign languages grammars.
  - *atlasGetData():* this operation is invoked by the "Layer Builder Component" when the user requires some features to be visualized on the webGIS. It returns the list of entities composing of the layer and for each of them the resulting value.

4.1.6. Multimedia Manager Module

The Multimedia Manager Module (see Figure 8) is responsible to manage all the multimedia data of the SIGN-HUB Web Platform. This module is strictly connected to the Data Abstraction Module to create, read, update or delete (i.e., CRUD) a multimedia content. Multimedia data are of utmost importance for SIGN-HUB Web platform especially for the Digital Archive of Old Signers' Linguistic and Cultural Heritage Suite. For this reason, this module provides all the functions needed to smartly manage and upload media contents such as video, text, documents and metadata files.

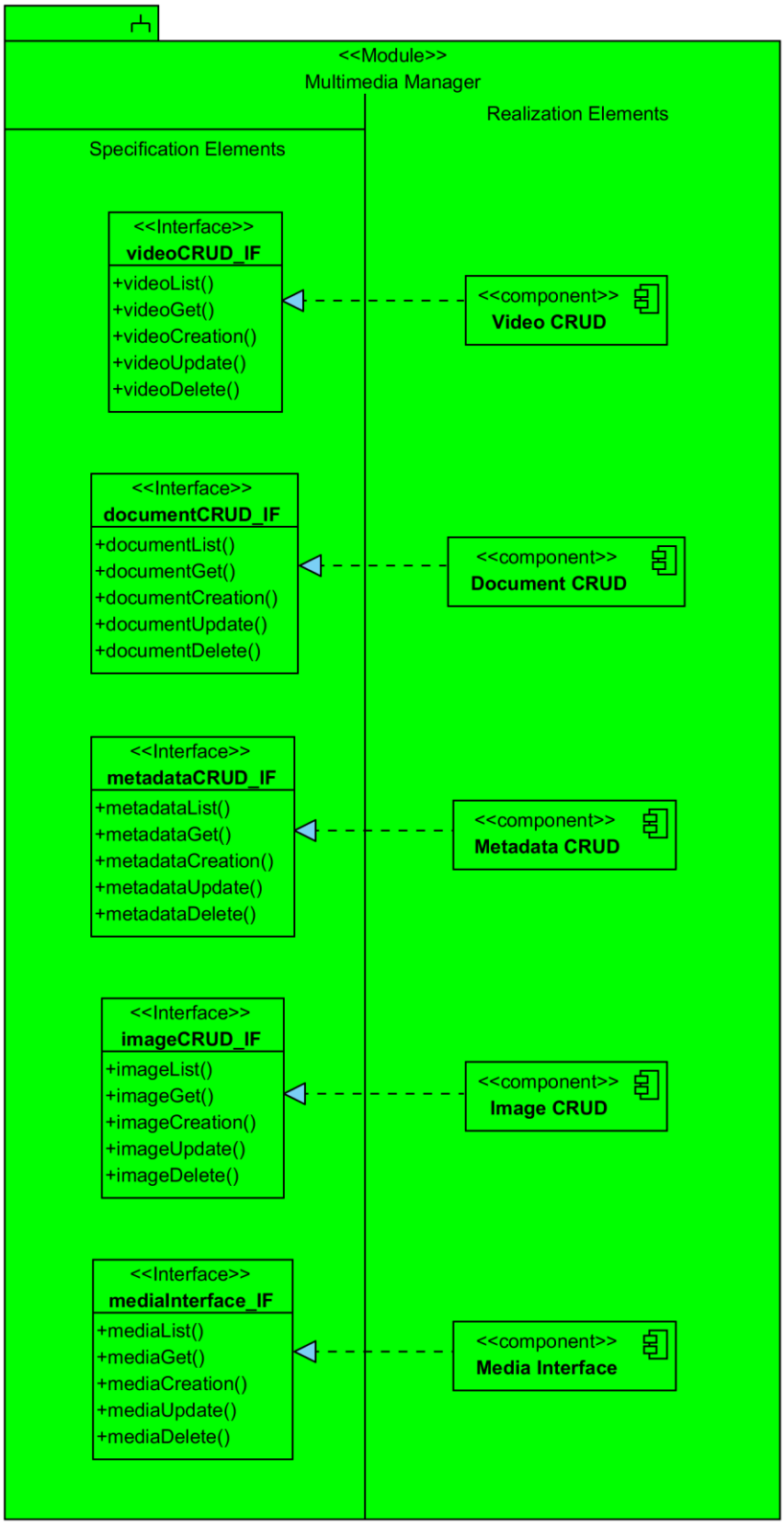

Figure 8: The Multimedia Manager Module

The UML components defined for this module are:

- "Media Interface Component": this component is the entry point of the module. When the user or a module of the platform require to interact with a multimedia data, they perform a request to this component. According to the type of file required, this component forward such a request to one of the other components of this module.
- "Video CRUD Component": this component manages video file stored in the database of the platform, only. As already described in Deliverable 3.2, the majority of the video materials will be uploaded on an external video streaming server. On the contrary, small video content related to specific tools (e.g., the video answers of the subjects when a test requires it) will be stored on the database of the platform.
- "Document CRUD Component": this component manages structured documents (PDF, DOCX) stored in the database of the platform. This is required when for Digital Archive of Old Signers' Linguistic and Cultural Heritage Suite or SignGram Blueprint Suite, structured documents could be attached to a video or to a grammar rule.
- "Metadata CRUD Component": this component manages metadata stored in the database of the platform. Metadata are especially linked to video contents (independently from the storage source) where subtitles are mostly required. Moreover, metadata are also adopted to tag video contents allowing users to rapidly search on a specific section of interest within the video itself.
- "Image CRUD Component": this component manages images stored within the database.

The interfaces provided by these components allow the other modules to access a multimedia resource managed by SIGN-HUB web platform. The defined interfaces are:

- "mediaInterface_IF": this interface is the entry point of the module. It parses the request to figure out the required data type and forward the request to the specific component. It can manage also multiple requests. The defined operations are:
    - mediaList(): this operation returns a list of multimedia files for a specific type (i.e., metadata, video, document, image). Filter options are also supported.
    - mediaGet(): this operation returns a single multimedia file.
    - mediaCreation: this operation stores on the database a multimedia file.
    - mediaUpdate: this operation updates an existing multimedia file.
    - mediaDelete: this operation deletes an existing multimedia file.
- "videoCRUD_IF": this interface can be invoked by the "Media Interface Component" to require a video file stored on the database. This interface returns also the metadata identifiers linked to the required video so that the "Media Interface Component" can perform a specific query to the "Metadata CRUD Component" to get the related metadata.
    - videoList(): this operation returns a list of video files. Filter options are also supported.
    - videoGet(): this operation returns a single video file and the associated metadata identifiers.
    - videoCreation: this operation stores on the database a single video file and the associated metadata identifiers.
    - videoUpdate: this operation updates an existing video file and the associated metadata identifiers.
    - videoDelete: this operation deletes an existing video file and the associated metadata identifiers.

- "documentCRUD_IF": this interface can be invoked by the "Media Interface Component" and returns a specific document.
    - documentList(): this operation returns a list of document files. Filter options are also supported.
    - documentGet(): this operation returns a single document file.
    - documentCreation: this operation stores on the database a single document.
    - documentUpdate: this operation updates an existing document file.
    - documentDelete: this operation deletes an existing document file.
- "metadataCRUD_IF": this interface can be invoked by the "Media Interface Component" and returns a specific metadata.
    - metadataList(): this operation returns a list of metadata files. Filter options are also supported.
    - metadataGet(): this operation returns a single metadata file.
    - metadataCreation: this operation stores on the database a single metadata file.
    - metadataUpdate: this operation updates an existing metadata file.
    - metadataDelete: this operation deletes an existing metadata file.
- "imageCRUD_IF": this interface can be invoked by the "Media Interface Component" and returns a specific image.
    - imageList(): this operation returns a list of image files. Filter options are also supported.
    - imageGet(): this operation returns a single image file.
    - imageCreation: this operation stores on the database a single image file.
    - imageUpdate: this operation updates an existing image file.
    - imageDelete: this operation deletes an existing image file.

4.1.7. Data Abstraction Module

The Data Abstraction Module (see Figure 9) offers services of object-relational mapping (ORM) to the other modules of the platform. Therefore, it manages data persistence on the database by representing and maintaining data of relational database as objects in the programming logic. This strategy facilitates modeling entities based on application concepts (e.g., each grammar is considered as an object) rather than based on the inherent database structure having the whole data model implemented according to the object-oriented programming paradigm. Since many different open source software solutions exist with such functional requirements, we have chosen to model this module with only a generic component as it will not be implemented by our team rather it will be selected among the open source products.

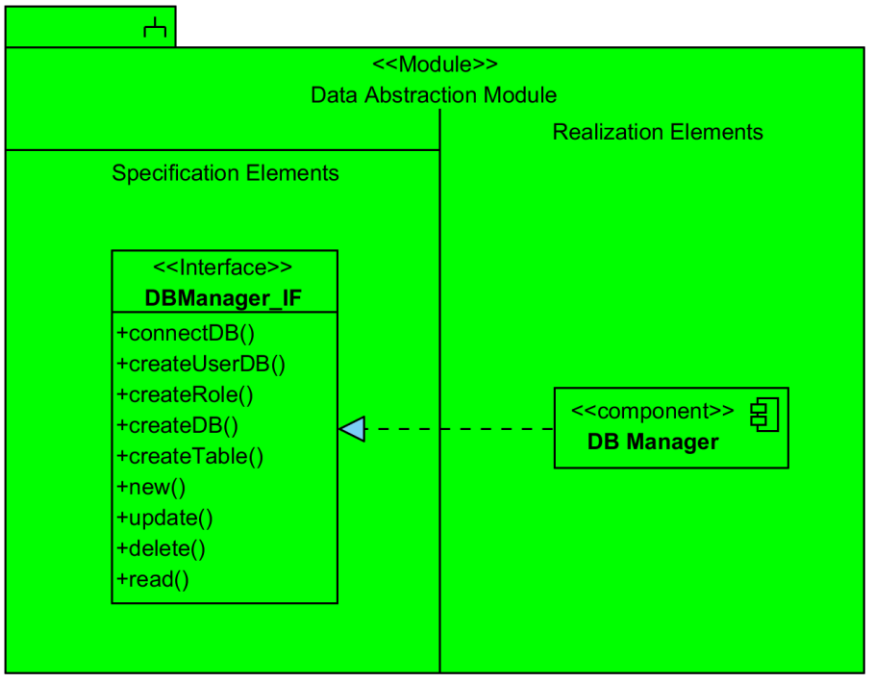

Figure 9: The Data Abstraction Module

The UML component defined for this module is the "DB Manager Component". It manages data persistence and the incoming transaction requests to the database. This component is invoked whenever each module of the SIGN-HUB platform requires to access to data stored in the database. A single interface is defined:

"DBManager_IF" this interface can be invoked to interact with the object representations of data. The defined operations are:

- *connectDB():* this interface is to connect this module with a specific DB (e.g., Oracle, MySQL).
- *createUserDB():* this interface is to create a new user and to choose the role it belong to.
- *createRole():* this interface is to create a specific role determining access grants.
- *createDB():* this interface is to create a new DB.
- *createTable():* this interface is to create a new table.
- *new():* this interface is to create a new object.
- *update():* this interface is to update the state of an object.
- *delete():* this interface is to delete one or more objects or an entire table.
- *read():* this interface is to read the state of an object.

4.1.8. Analytics Module

This module has two main purposes.

Firstly, it provides all of the other modules of the platform with functionalities for the generation of reports. Each Suite has functions that allow user to generate and download a report. For instance, the SignGram Blueprint Suite allows the generation of a PDF for a specific Sign Language and the Test Administration Suite for Sign Languages allows the generation of a report containing all of the answers for a specific compiled test.

Secondly, it manages complex query to retrieve geo data related to the features that can be required by the GIS Module.

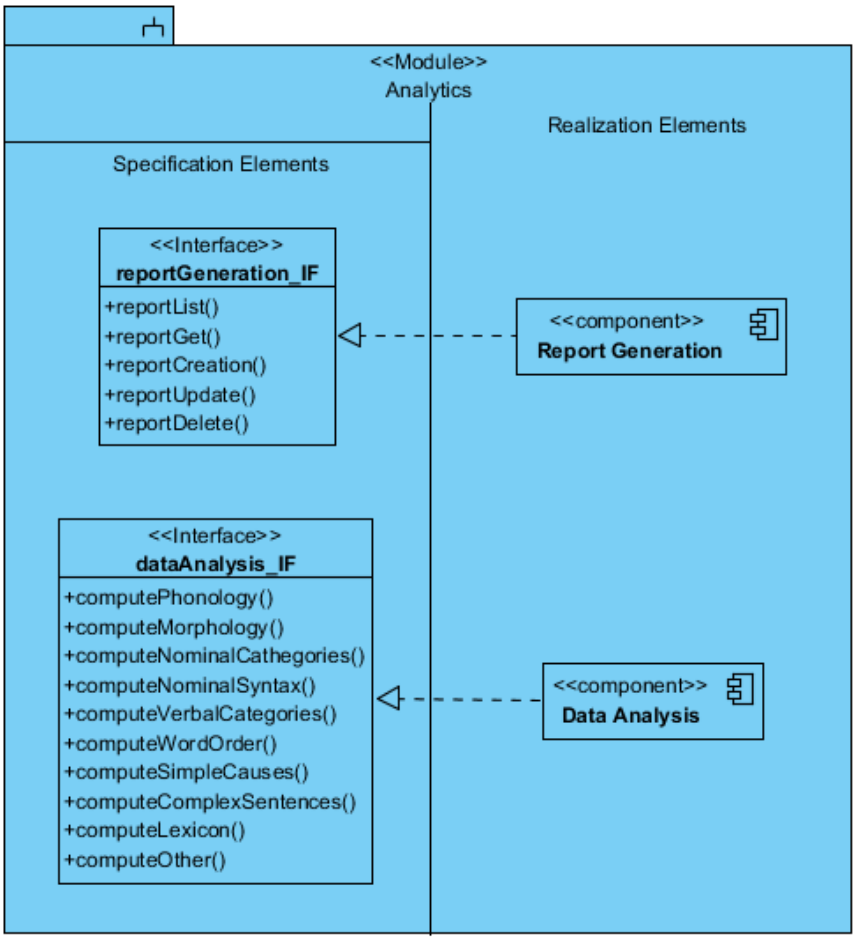

Figure 10: The Analytics Module

The UML components defined for this module are:

- "Report Generation Component": this is the component that manages the generation of reports. This component strictly interacts with the Data Abstraction Module to retrieve data of interest to create the report.
- "Data Analysis": this is a sophisticated analytical module. It is invoked by the GIS Module when a query about linguistic features that have to be geo-referenced is required by the user. When invoked this component has to compute the number of the required feature within the sing language grammars stored in the database. Data generated by this component are used by the GIS Module to generate layers.

The defined interfaces for these components are:

- "reportGeneration_IF": this interface provides the other modules with a rich set of API to create, delete or update a report. The defined operations are:
    - *reportList()*: this operation returns a list of reports. For instance, it is invoked when the list of test reports is required.
    - *reportGet()*: this operation returns a single report.
    - *reportCreation()*: this operation creates a new report based on the user request.
    - *reportUpdate()*: this operation updates an existing report.

- *reportDelete()*: this operation deletes an existing report.

- "dataAnalysis_IF": this interface allows GIS Module to retrieve the linguistic information required to generate a layer. The defined operations are:
  - *computePhonology():* this operation manages the queries that involve features that belong to the phonology of the language (e.g., consonants, vowels). It returns the numeric value of the feature contained in the specific language.
  - *computeMorphology():* this operation manages the queries that involve features that belong to the morphology of the language (e.g., Inflectional Synthesis of the Verb, Locus of Marking in the Clause). It returns the numeric value of the feature contained in the specific language.
  - *computeNominalCathegories():* this operation manages the queries that involve features that belong to the nominal cathegories of the language (e.g., gender, Plurality in personal pronouns). It returns the numeric value of the feature contained in the specific language.
  - *computeNominalSyntax():* this operation manages the queries that involve features that belong to the nominal sintax of the language (e.g., possessive classification, genitives). It returns the numeric value of the feature contained in the specific language.
  - *computeVerbalCategories():* this operation manages the queries that involve features that belong to the verbal categories of the language (e.g., future tense, perfect). It returns the numeric value of the feature contained in the specific language.
  - *computeWordOrder():* this operation manages the queries that involve features that belong to the word order of the language (e.g., Order of Subject, Object and Verb). It returns the numeric value of the feature contained in the specific language.
  - *computeSimpleCauses():* this operation manages the queries that involve features that belong to the simple causes of the language (e.g., passive construction, applicative construction). It returns the numeric value of the feature contained in the specific language.
  - *computeComplexSentences():* this operation manages the queries that involve features that belong to the complex sentences of the language (e.g., when clause, reason clause). It returns the numeric value of the feature contained in the specific language.
  - *computeLexicon():* this operation manages the queries that involve features that belong to the lexicon of the language (e.g., finger, hand and arm). It returns the numeric value of the feature contained in the specific language.
  - *computeOther():* this operation manages the queries that involve features that do not belong to no one of the aforementioned categories. It returns the numeric value of the feature contained in the specific language.

4.2. Software Suites

A software suite is a product or tool with well-defined functional boundaries. According with D3.1, we have grouped the user requirements of the SIGN-HUB web platform in 4 different suites. Even though the platform will be composed of 4 suites users can separately access in, they completely share the logic resources and the database provided by the modules de-

scribed in Section 4.1. In this Section, the modules composing of each Suite and how they interact between each other to accomplish user requests are here described. The functional diagrams show such interconnections even though it is not specified which operation of the interface is invoked. Being a high-level diagram, it is more focused on component interaction activities than how the interaction happens. In the following functional diagrams, if a component within a module is not functionally involved (e.g., functionalities offered by that specific component are not required by the other components), it is represented without no open arrow linked to its interface.

4.2.1. The Test Administration Suite for Sign Languages

The Test Administration Suite for Sign Languages imports (i.e., uses) the modules of the platform shown in Figure 11. The interactions between components are shown in Figure 12. Each action starts always from the user that interacts with the user interface of the Test Administration Suite for Sign Languages that requires to successfully login into the system by invoking operations of the Security Module. According to the specific action performed by the user and his access grant, the Grammar Controller can invoke either an operation of the Test Manager Module or an operation of the Media Interface component. All the components invoked for test creation then invoke DB interface to store/get test objects. While if the testReport_IF is invoked, the Test Report component invokes the reportGeneration_IF for the generation of query to be performed to the Database for the specific report. On the contrary when a media element is required by the user, the Grammar Controller Component invokes the mediaInterfcae_IF Interface. Then, the Media Interface Component invokes the other interfaces of its module to access to the desired media contents.

Even if, from the design perspective, the Answer Validation component is part of the Test Manager Module, its functionalities are not required for the Test Administration Suite for Sign Languages. For this reason, this component will be implemented for the Atlas of Sign Language Suite as explained in Section 4.2.2.

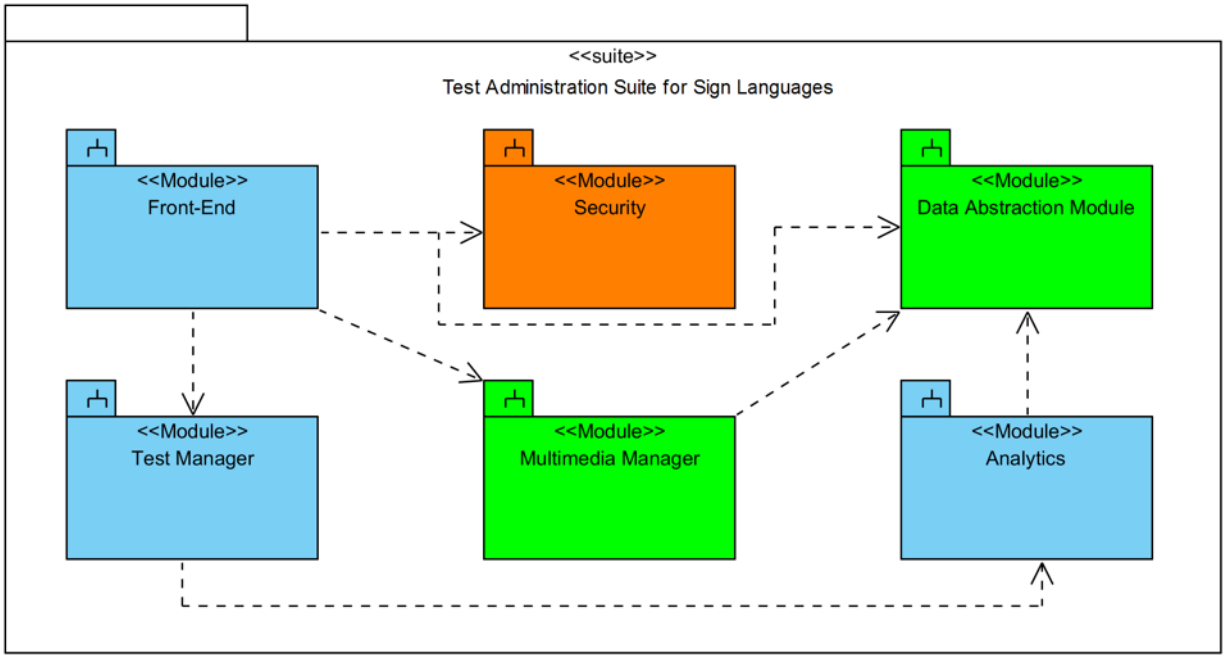

Figure 11: View of the Test Administration Suite for Sign Languages

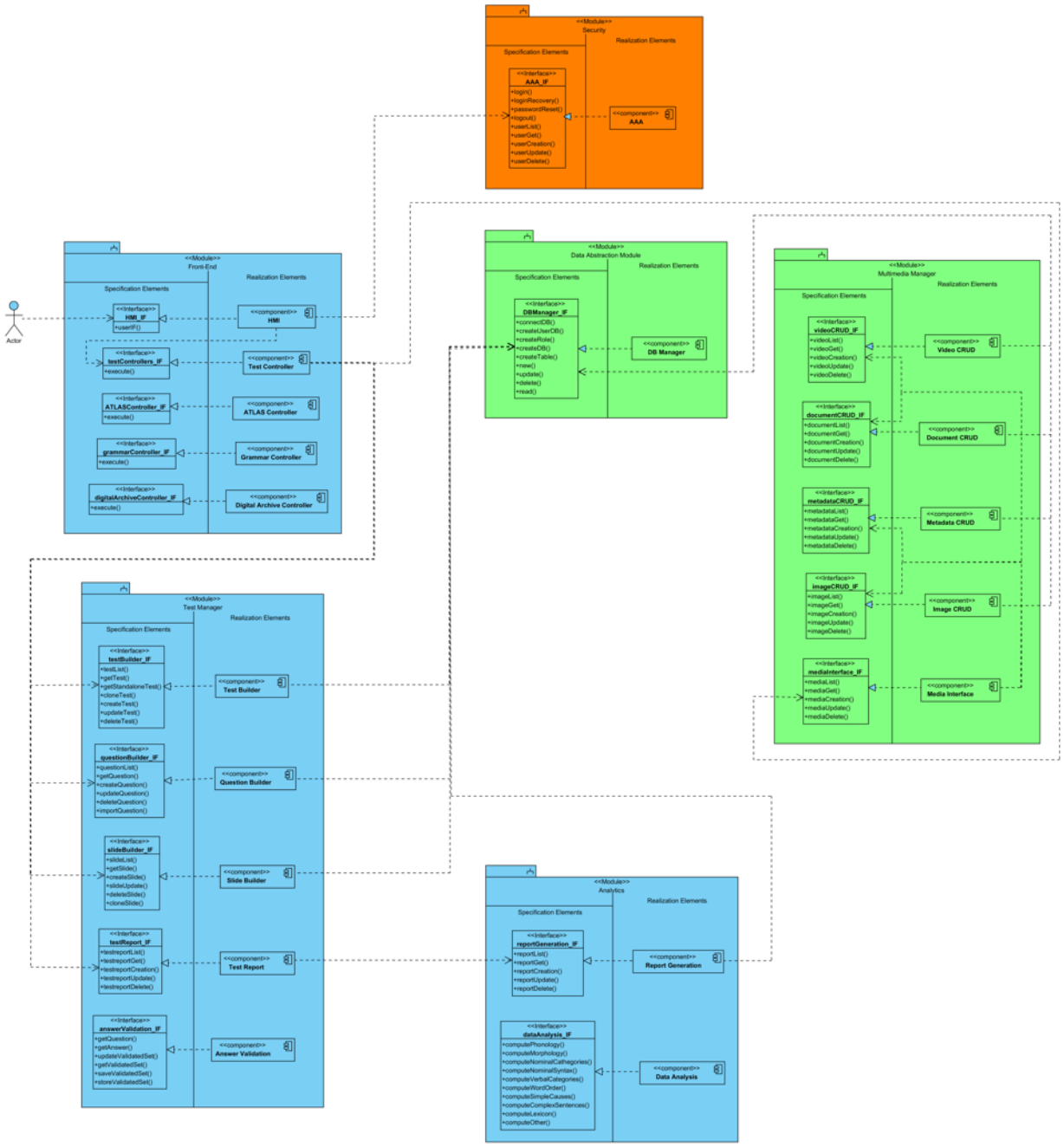

Figure 12: Functional diagram of the Test Administration Suite for Sign Languages

4.2.2. The Atlas of Sign Language Suite

The Atlas of Sign Language suite imports (i.e., uses) the modules of the platform shown in Figure 13. The interactions between components are shown in Figure 14. The user interacts with the specific GUI of this Suite fundamentally to: 1) create a test, 2) validate the obtained answers, 3) navigate the linguistic atlas. The functional interaction among components for test creation task is described in Section 4.2.1. The only difference is that the Atlas Controller component in this case invokes the interfaces provided by the Test Manager Module. The reason is that, even if not designed yet (it will be provided in the Deliverable 3.7), the GUI for test creation for users of sign language atlas could be slightly different and so we prefer to have a dedicated controller. The answer validation task requires that the ATLAS Controller invokes the An-

swerValidation_IF interface. Then the Answer Validation Component invokes directly the DBManager_IF to provide the user with proper data to be validated.

Instead, when the end user wants to navigate the atlas of sign languages, the controller invokes the layerBuilder_IF interface. According to the performed query (e.g., about morphology), the Layer Builder Component invokes the atlasQueryBuilder_IF interface for retrieving data to populate the GIS layer. The Atlas Query Builder Component invokes one of the operations provided by the dataAnalysis_IF interface. When data have been loaded from the Data Abstraction Module (e.g., number of consonants per sign language enriched with specific data), the Layer Builder Component and the Geolocalization Engine build the specific layer geolocalizing sign languages on a digital word map. Finally, the user can navigate the digital map and can require the suite to generate a report. In this case, the controller invokes directly the reportGeneration_IF interface.

The Front-End Module integrates dedicated functions to invoke external Digital Map providers (e.g., Open Street Map) to load on the web GUI a digital map which is considered as an external system and for this reason it will not be described in this deliverable.

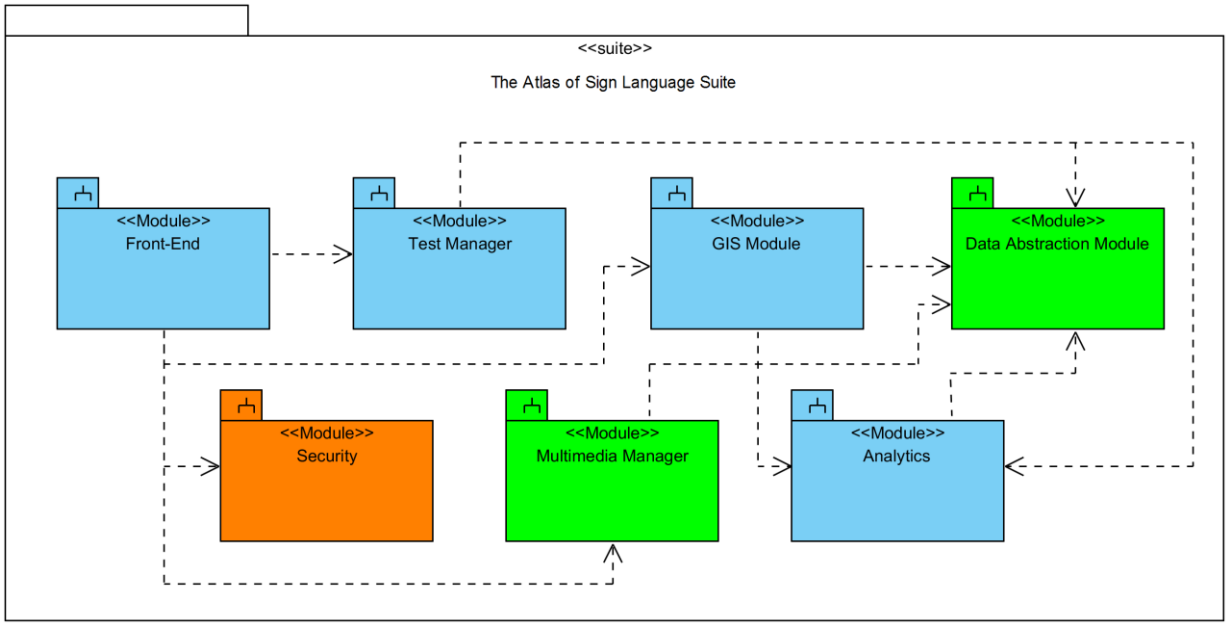

Figure 13: View of the Atlas of Sign Language Suite

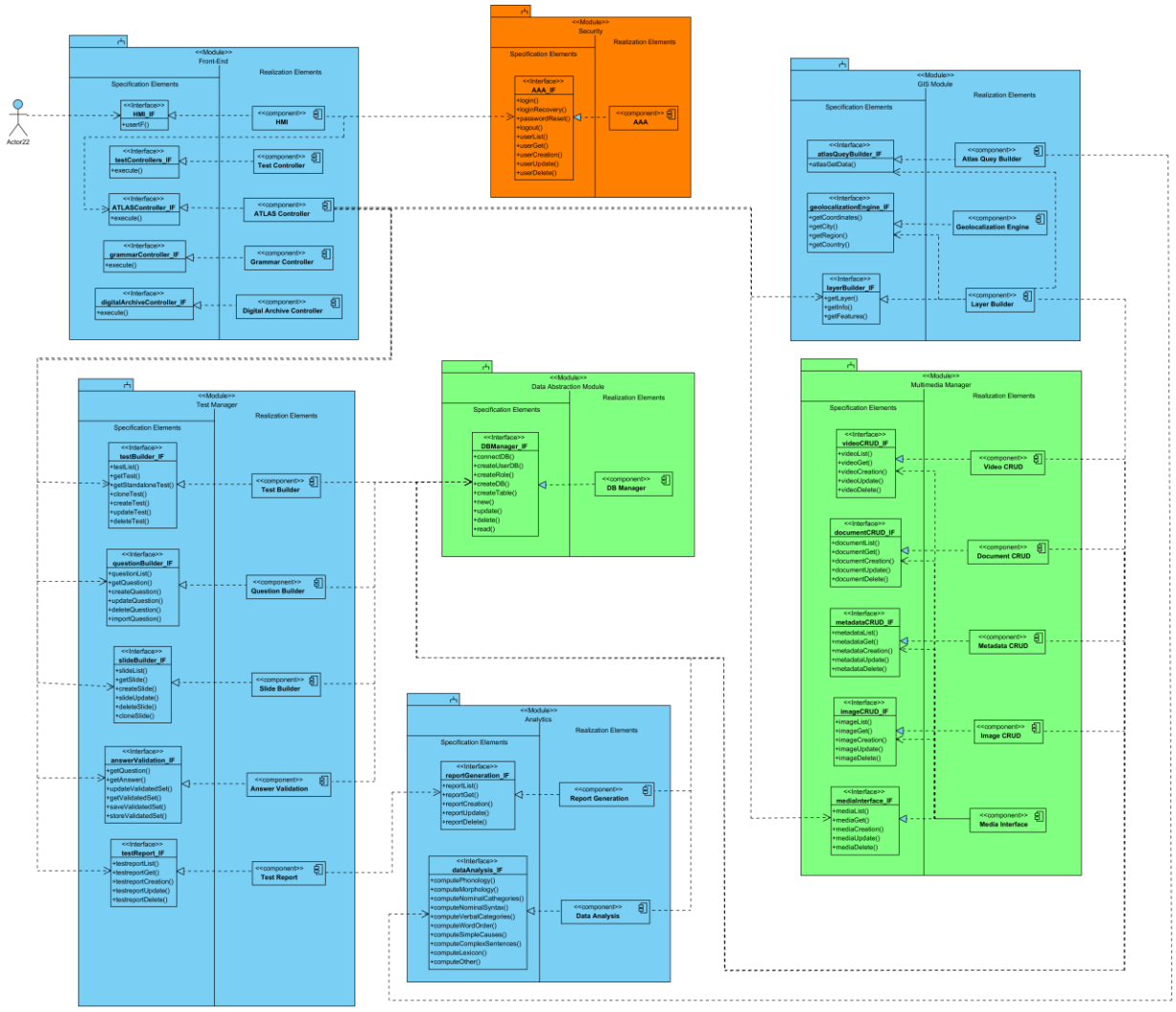

Figure 14: Functional diagram of the Atlas of Sign Language Suite

4.2.3. The SignGram Blueprint Suite

The SignGram Blueprint suite imports (i.e., uses) the modules of the platform shown in Figure 15. The interactions between components are shown in Figure 16. When any user logs into this suite, according to his grant access, he/she can require seeing the status of a sign language grammar by interacting with the proper Suite GUI. The Grammar Controller sends proper messages to the grammarBuilder_IF and grammarStatus_IF interfaces to load the requested grammar and its current state of completion. After that, each sign language grammar update is directly managed by the Grammar Builder component that directly invokes the mediaInterface_IF interface when images, documents or video are requested to enrich grammar description. According to the required type of multimedia, the dedicated component of the Multimedia Manager module (e.g., Video CRUD) invokes the DBManager_IF to access the Database.

If a sign language grammar is updated, the Grammar Builder component invokes the grammarNotifier_IF interface to send a notification (i.e., email) to the person who is in charge, for that specific sign language grammar, to manage the process of grammar creation and update validation. Then the contacted person can access to this suite and can definitively accept or decline the modifications. This process is fully managed by the Grammar Manager Module.

When a report generation activity is required, the Grammar Controller Component invokes the grammarReport_IF that process the request and thus invokes the reportGeneration_IF. Finally, the Grammar Report Component invokes the DBManager_IF interface to get the required data.

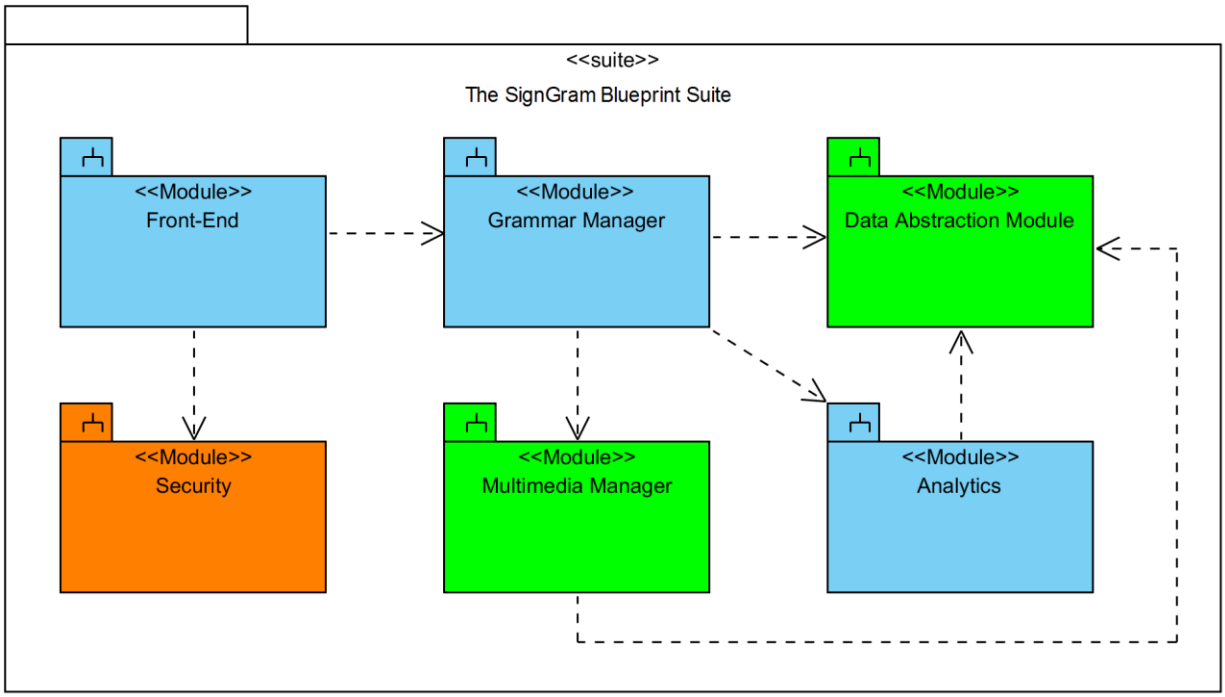

Figure 15: View of SignGram Blueprint Suite

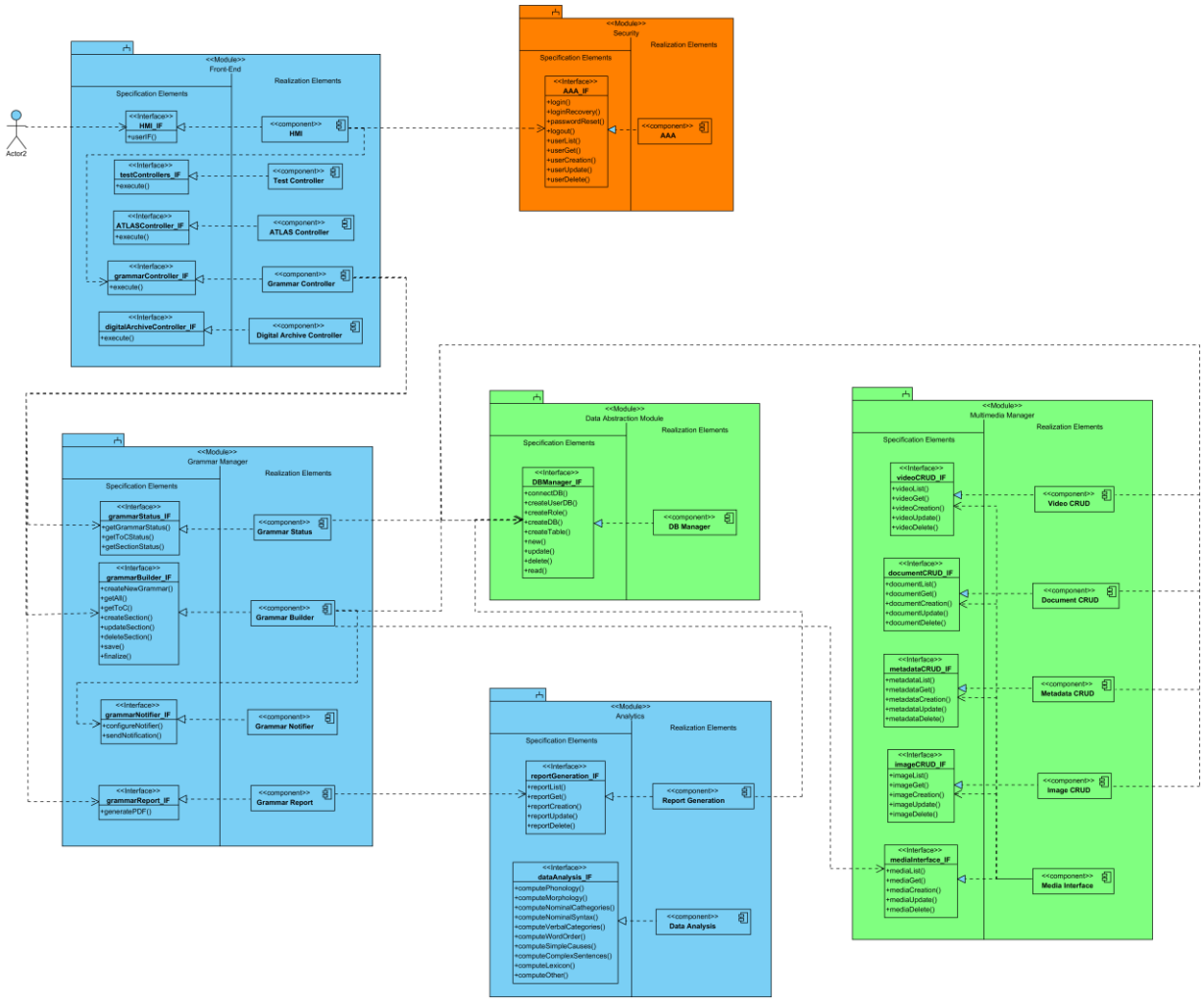

Figure 16: Functional diagram of the SignGram Blueprint Suite

4.2.4. The Digital Archive of Old Signers' Linguistic and Cultural Heritage Suite

The SignGram Blueprint suite imports (i.e., uses) the modules of the platform shown in Figure 17. The interactions between components are shown in Figure 18.

The specific Digital Archive Controller component manages all the interactions between the user and the Digital Archive of Old Signers' Linguistic and Cultural Heritage user interface. This suite can be considered as a container of multimedia resources, some stored within the database and some provided by external services that are invoked directly by the related component of the Front-End module. For instance, when a video content is requested, the Digital Archive Controller component send a request to the mediaInterface_IF. Then, if the video content is available in the platform database, the Media Interface component invokes the internal videoCRUD_IF interface otherwise it invokes only the metadataCRUD_IF to gather all the metadata associated to the video (e.g., subtitles and video tagging). Metadata CRUD component, thus, invokes the DBManager_IF to retrieve the metadata. Finally, all data are sent back to the Front-End Module that creates dynamically the web page structure with the implemented presentation logic.

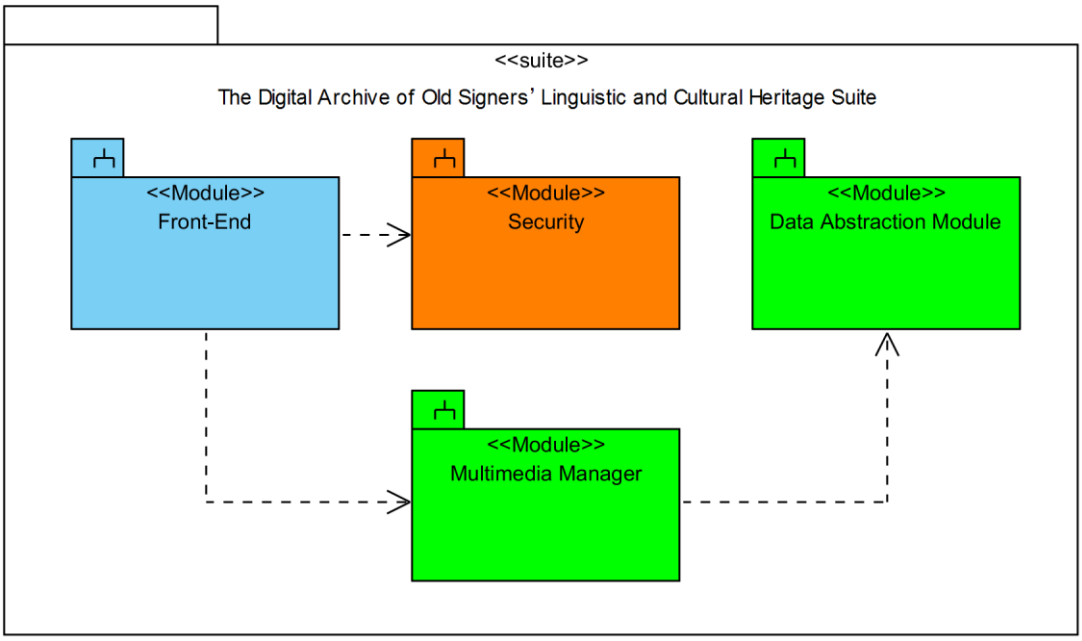

Figure 17: View of The Digital Archive of Old Signers' Linguistic and Cultural Heritage Suite

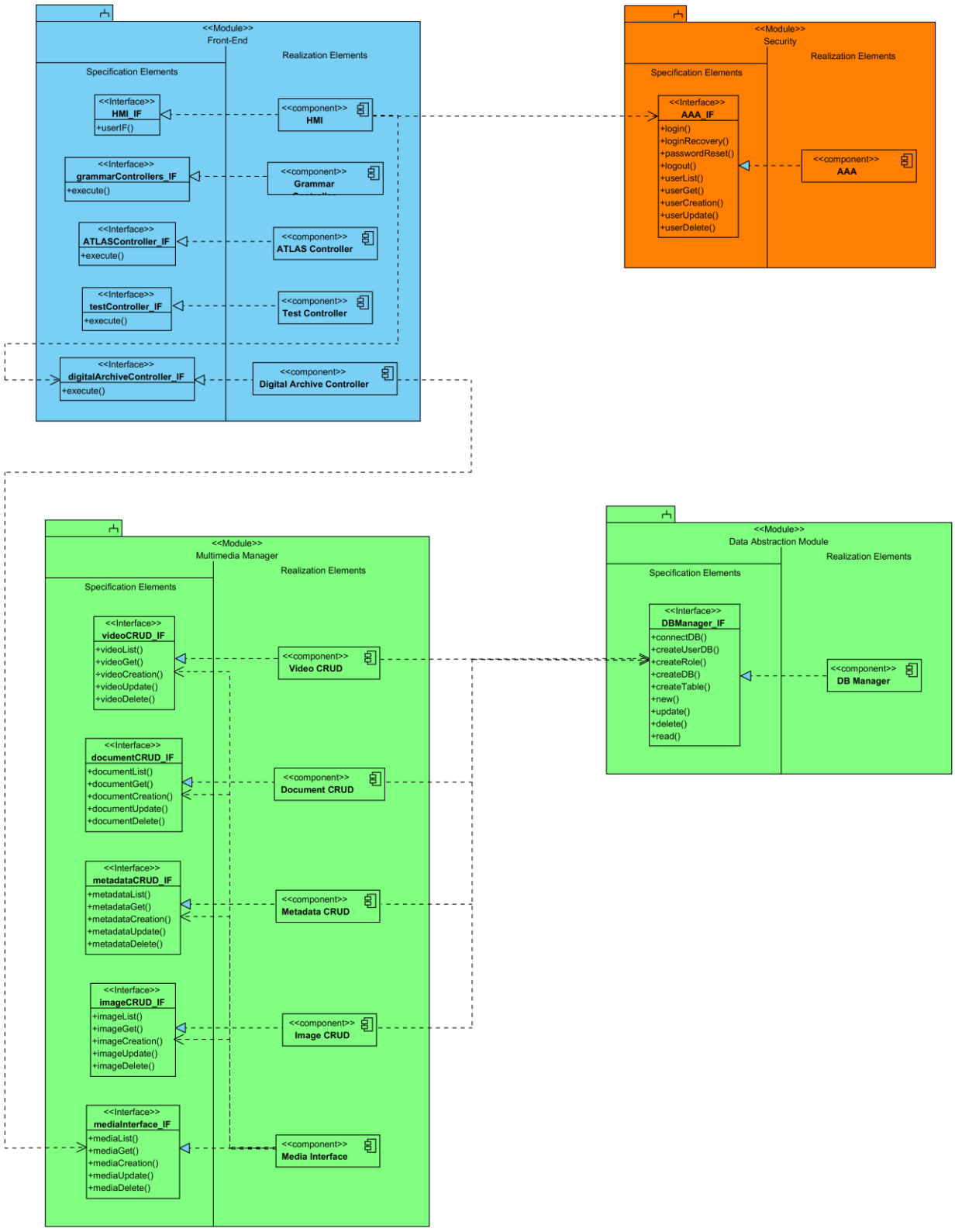

Figure 18: Functional diagram of the Digital Archive of Old Signers' Linguistic and Cultural Heritage Suite

4.3. Technological Considerations

After defining a preliminary architectural design of the SIGN-HUB web platform, the design team and the development team collaborated to choose the technologies for implementing each module described in Section 4.

The **Front-End module** is fully implemented by means of Angular JS[6]. Angular is a well-known framework which allows the development of rich web applications characterized by a higher grade of client-side functionality and complexity. This allows for user interfaces that are more user friendly and more performant and this is exactly the purpose of the Front-End module since, as explained in Section 4.2, Controller components are enough smart to choose, according to the context, the proper back-end operation to invoke. Angular JS is based on HTML 5 and Javascript technologies.

The Security, GIS Module, Test Manager, Multimedia Manager, Grammar Manager, Analytics aredeveloped with Java technology[7]. Java is one of the most adopted server-side programming language. It is highly flexible and cross-platform language.

The **Data Abstraction Module** is implemented through Hibernate[8]. It is a Java-based object-relational mapping framework designed to facilitate the development of database-oriented modules. The database is implemented with Maria DB[9] which is one of the most popular open source database.

Table 2: SIGN-HUB Web Platform Technologies

Module	Technology	Version
Front-End	Angular JS	2.0
Security	Java	8.144
Data Abstraction	Hibernate	5.2
GIS	Java	8.144
Test Manager	Java	8.144
Multimedia Manager	Java	8.144
Grammar Manager	Java	8.144
Analytics	Java	8.144
Database	Maria DB	10.2.9

[6] *https://angularjs.org/*

[7] *https://www.java.com*

[8] *http://hibernate.org/*

[9] *https://mariadb.org/*

As previously discussed, the Front-End module manages both the graphical presentation of the contents and the interactions between the user and the GUI. In order to properly manage the multiple connections between front-end and back-end and to assure high scalability in regard to a high number of parallel users, the platform has been designed to be web service-oriented. This means that each module is developed as a web service (i.e., software module that can receive messages via web) and that at least one component of each module exposes a web service interface (except the Data Abstraction Module that has a local interface), while the other components could have a local interface not reachable from internet. The benefits of a web service architecture is that, in principle, each web service runs in its own execution environment and so if one service fails, the platform is still on-line. Moreover, a service can be easily replaced without deploy again the whole platform but just the service itself. From scalability perspective, if the number of user requests outperforms the foreseen request limits, system administrator can double the number of running web services to load balance the incoming web traffic. When Suites will be implemented and delivered, a new issue of this deliverable will be released updating contents and adding an appendix reporting a detailed description of the implemented web services.

In Figure 19 the functional diagram of the SIGN-HUB Web Platform and the type of interface for each component are specified, providing, thus, an overall picture of the interactions among all of the modules rather than focusing on the interactions among a sub-set of modules as previously depicted in Section 4.2. The interfaces, defined as stereotype within the UML Module, are classified in: external web service and internal web service. External web services are interfaces that can be invoked from the Front-End while internal web services are interfaces that are invoked directly from component that are within the back-end layer.

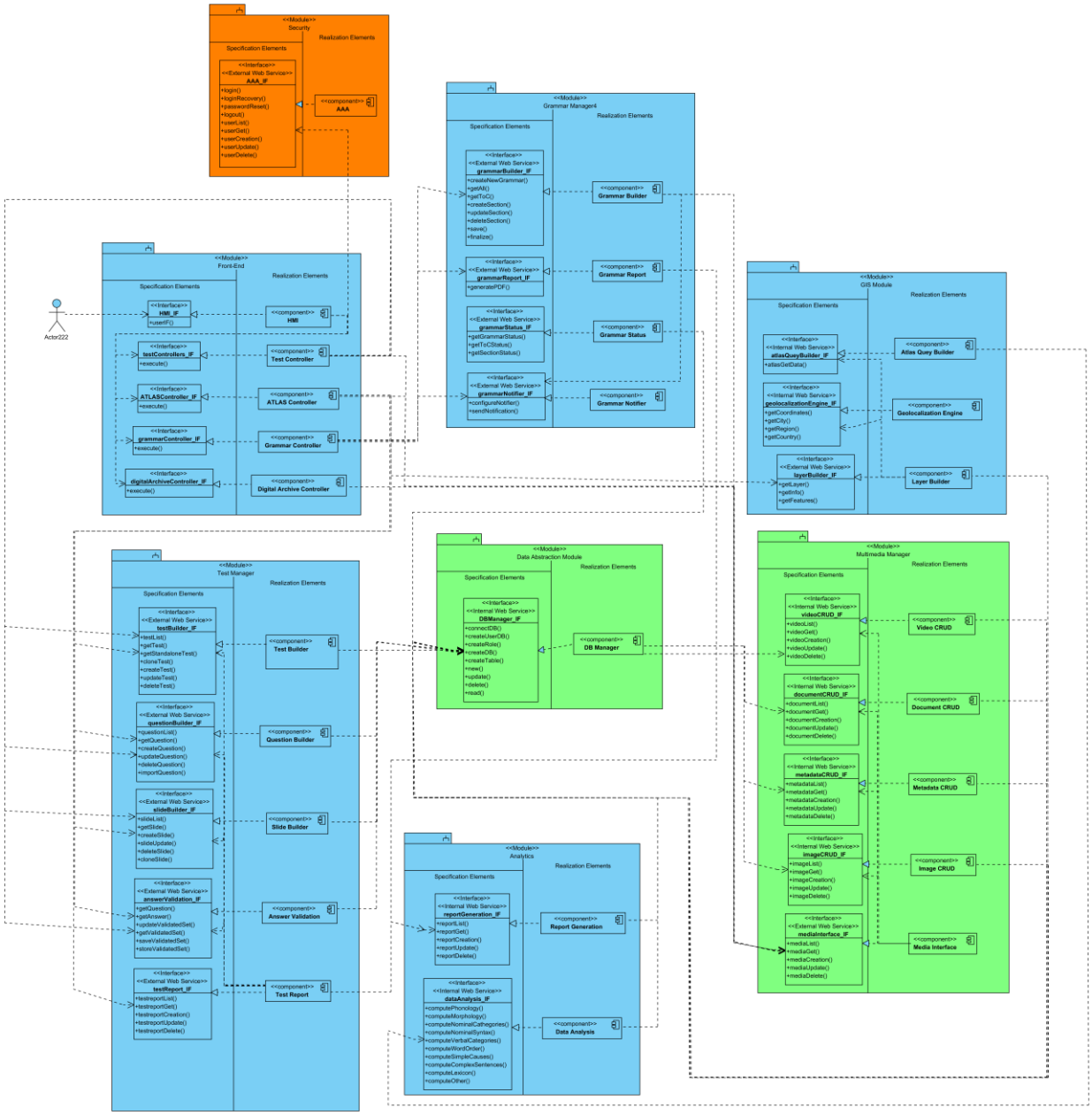

Figure 19: Functional diagram of SIGN-HUB Web Platform

5. Conclusion

The Deliverable 3.5 presents the preliminary design of the back-end of the platform that will be developed to foster the creation of a performant, scalable, and reliable web-based platform. This Deliverable presents the overall software system architecture and a close-up description of each module, as well as the relationships developed among them and the functionalities they expose. Specifically, 8 modules have been identified: Front-end Module, Security Module, Multimedia Manager, Data Abstraction Module, Analytics Module, Test Manager, GIS Module, and Grammar Manager.

Moreover, a fully description of the Suites composing the platform is also provided focusing on their functional description with respect to the specific purpose of each suite described in the requirement specification document (D3.1). The Suites that have been defined are: the SignGram Blueprint, the Atlas of Sign Language, the Test Administration Suite for Sign Languages, and the Digital Archive of Old Signers' Linguistic and Cultural Heritage.

Finally, the strategic decisions about technologies are also provided, presenting the technology that has been chosen to implement each module of the platform, with a focus on the type of interface: web service or local. The design of the front-end module is out-of-scope of this deliverable and will be presented contextually with the delivery of each suite. When Suites are implemented, the appendix will be enriched with a detailed description of the implemented operations.

Appendix A: Test Administration Suite for Sign Languages

In this appendix, the detail of the web services implemented for the Test Administration Suite for Sign Languages (see Deliverable 3.8) is provided.

First of all, the list of the structures which have been created is provided. The structure will be used as input/output of the web service interfaces.

Response Object

All response has the following structure:

```
{
  "status": OK || NOK,
  "errors": [ERROR_OBJ],
  "response": { OBJ }
}
```

Error Object

```
{
"errorCode" : 5, //error code number
"errorMessage" : "Generic error" //error message
}
```

Test Object Small

```
{
"TestId" : 232, //Test id
"TestName" : "Test Baldassarre for hospice" //Test name
"authorId" : "24680it", //author userID
"toEdit" : true||false //this flag indicates if the current user can delete the report
}
```

Test Object

```
{
"TestId" : 232, //Test id
"TestName" : "Test Baldassarre for hospice", //Test name
"authorId" : "24680it", //author userID
"deleted" : false, //flag for logical deletion
"state" : "published", //state of the test draft||published
```

"revId" : 3,

"toEdit" : true||false //this flag indicates if the current user can delete the Test

"questions" : [{questionId1:questionName1}, {questionId2:questionName2}, ..., {questionIdN:questionNameN}], //array of question Ids and name

"options" : ["key" : "value", "key1": "value1", ... , "keyN" : "valueN"] //array of options

}

Test Complete Object

{

"TestId" : 232, //Test id

"TestName" : "Test Baldassarre for hospice", //Test name

"authorId" : "24680it", //author userID

"deleted" : false, //flag for logical deletion

"state" : "published", //state of the test draft||published

"revId" : 3,

"toEdit" : true||false //this flag indicates if the current user can delete the Test

"questions" : [QUESTION_COMPLETE_OBJ], //array of question comple objects

"options" : ["key" : "value", "key1": "value1", ... , "keyN" : "valueN"] //array of options

}

Question Object

{

"questionId" : "34fr63", //id of the question

"slides" : [slideId1, slideId2, ..., slideIdN], //array of slides ids

"transitionType" : ["time||click||enter||action"], //type of transition between questions

"options" : ["key" : "value", "key1": "value1", ... , "keyN" : "valueN"], //array of options

"toEdit" : true||false //this flag indicates if the current user can delete the question

}

Question Complete Object

{

"questionId" : "34fr63", //id of the question

"slides" : [SLIDE_OBJ], //array of slides

"transitionType" : ["time||click||enter||action"], //type of transition between questions

"options" : ["key" : "value", "key1": "value1", ... , "keyN" : "valueN"], //array of options

"toEdit" : true||false //this flag indicates if the current user can delete the question

}

Slide Object

```
{
"slideId" : "sl23itv2395", //slide id
"type" : "blank||info||stimulus||distraction||question||answer", //the type of the slide
"transitionType" : ["time||click||enter||action||answer"], //type of transition between slides
"options" : [ "key" : "value", "key1": "value1", ... , "keyN" : "valueN"], //array of options
"slideContent" : {SLIDE_CONTENT_OBJECT}, //the slide content depends on slide type
"toEdit" : true||false //this flag indicates if the current user can delete the slide
}
```

Slide content object

```
{
"componentArray" : [{
    "componentType" : "button",
    "mediaId" : NULL,
     "pos" : "250,120", //the left,top position in %
    "dim" : "100,200", //the width,heigth dimension in %
    "options" : [
            {"name" : "elephant" },
            {"value" : 20 },
            {"label" : "" },
            {"text" : "Elephant" },
            {"transition" : false }
            ]
    },
    {
    "componentType" : "clickArea",
    "mediaId" : NULL,
     "pos" : "0,0", //the left,top position in %
    "dim" : "100,100", //the width,heigth dimension in %
    "options" : [
            {"name" : "cat" },
            {"value" : 20 },
            {"isCorrect" : true},
            {"transition" : false }
            ]
    }
```

{

"componentType" : NULL,

"mediaId" : "541it", //the media id

"pos" : "0,0", //the left,top position in %

"dim" : "100,100", //the width,heigth dimension in %

"value" : "25",

"options" : [ ]

}

],

"toEdit" : true||false //this flag indicates if the current user can delete the content map

}

Media Object

{

"mediaId" : "541it", //the media id

"mediaType" : "VIDEO||PHOTO||AUDIO||TEXT", // media type

"mediaName" : "elephant.jpg", // the name of the media

"mediaPath" : "media/bald_media/elephant.jpg", // the path to download the media

"mediaAuthorId" : "24680it", //author userID

"mediaAuthorName" : "Pasqualino Andrea Baldassarre", // the name of the author

"mediaDate" : "17/05/2017", // the date of media creation

"toEdit" : true||false //this flag indicates if the current user can delete the media

}

Report Object Small

{

"reportId" : "r354plv97sd", // the id of the report

"reportDate" : "13/06/2017", // the date of Test submission

"reportCsvPath" : "/csv/TestBaldassarreForHospice/r354plv97sd.csv", //the path to download csv report

"reportTestName" : "Test Baldassarre for hospice", //the name of the related Test

"reportTestId" : 232, //the id of the related Test

"isSaved" : true, //this flag indicates if the report is saved (locally)

"isUploaded" : false, //this flag indicates if the report is uploaded to the server

"toEdit" : true||false //this flag indicates if the current user can delete the report

}

Report Object

{

"reportId" : "r354plv97sd", // the id of the report

"reportDate" : "13/06/2017", // the date of Test submission

"reportCsvPath" : "/csv/TestBaldassarreForHospice/r354plv97sd.csv", //the path to download csv report

"reportTestName" : "Test Baldassarre for hospice", //the name of the related Test

"reportTestId" : 232, //the id of the related Test

"toEdit" : true | |false, //this flag indicates if the current user can delete the report

"metadata" : ["meta1":"value1","meta2":"value2",...,"metaN":"valueN"], //the array of all metadata

"questions" : [REPORT_QUESTION_OBJ], //array of all question report for the test

"authorId" : "24680it", //the id of the author

"startTime" : 1435674, //timestamp of start time

"endTime" : 1435904, //timestamp of ent time

"elapsedTime" : 230, //effective elapsed seconds (needed for test pausing)

}

Report Question Object

{

"questionId" : "34fr63", //id of the question

"order" : 4, //the presentation order of the question

"startTime" : 1435674, //timestamp of start time

"endTime" : 1435904, //timestamp of ent time

"elapsedTime" : 230, //effective elapsed seconds (needed for test pausing)

"slides" : [REPORT_SLIDE_OBJ], //the array of report for significant slides

}

Report Slide Object

{

"slideId" : "34fr63", //id of the slide

"presentationOrder" : 2, //the order of presentation of the current slide (a report could have two REPORT_SLIDE_OBJ with the same slideId and two different presentationOrder because the user returns on the same slide and give another answer).

"startTime" : 1435674, //timestamp of start time

"endTime" : 1435904, //timestamp of ent time

"elapsedTime" : 230, //effective elapsed seconds (needed for test pausing)

"slideType" : "answer", //the type of the answer

"transitionPerformed" : "answer", //the action that performed the transition

"answers" : ["elephant", "dog"], //the array of all user answers

"expectedAnswers" : ["elephant", "cat"], // the array of all expected answers

"mouseTracking" : [{100,250},{120, 252}, ... , {450, 620}], //the array of all positions of the mouse

}

User Object

{

"userId" : "24680it", // the id of the user

"registrationDate" : "13/06/2017", // the date of User registration

"name" : "Pasquale", //the name of the user

"surname" : "Baldassarre", //the surname of the user

"role" : "ADMIN||CON_PRO||USER", //the role of the user: ADMIN for administrator, CON_PRO for content provider, USER for viewer user

"email" : Giuseppe.airofarulla@polito.it, //email address of the user

"password" : "sdf7a9ya9997da", //Encrypted password

"deleted" : "true||false"", //indicate if the user is deleted

"verificationCode" : "dsf89sd9sadf89" //if not null this field contain the verification code that allow the user to password recovery

}

In the following the descritption of the web services is provided.

Security Module: AAA_IF Interface

login()

This function is used to authenticate the user.

@POST()
@Path("/login")
@Consumes("application/json")
@Produces("application/json")
Input parameters:

- login : email of the user;
- password : user password;

Output example

```
{
  "status": OK,
  "errors": [ ],
  "response": {
    "userId":"24680it",
    "authToken":"dsf77sa_f8+9af79s-df98"
     }
}
```

loginrecover()

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

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

Input parameters:
- login : email of the user;

Output

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

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 parameters:
- login : email of the user;
- validationCode : the code received through mail;
- newPassword : the new password;
- rePassword : the new password;

Output

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

Logout()

This function allow the user to logout from the platform

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

Output

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

userList()

This function returns the list of users; could be used some input filters.

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

Input parameters:

- role (optional): String used to filter user by role.
- email (optional): String to filter users by mail
- name (optional) String to filter user by name or surname (sql like function)

Output

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

userGet()

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

```
@GET
@Path(“user/{userId}”)
    @Produces("application/json")
```

Output:

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

userCreation()

This function creates a new user object.

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

Input parameters:

- user: USER_OBJ to create;

Output

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

userUpdate()

This function updates the user. The id of the user is passed into the path. The new user configuration is passed as json object in input.

```
@POST()
@Path("/user/{userId}")
    @Consumes("application/json")
    @Produces("application/json")
```

Input parameters:

- user: USER_OBJ to update;

Output

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

userDelete()

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

```
@POST
@Path("user/{userId}")
    @Produces("application/json")
```

Output:

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

Test Manager Module: testBuilder_IF Interface

TestList()

This function return the list of Tests.

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

Output

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

getTest()

This function returns the Test json object that have the id in input.

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

Input parameters:

- complete: true/false (if true, the response obj is of type TEST_COMPLETE_OBJ; TEST_OBJ otherwise)

Output

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

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 mapping from media id and the local file.

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

Input parameters:

- complete : true/false (if true, the response obj is of type TEST_COMPLETE_OBJ; TEST_OBJ otherwise)

Output zip file containing:

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

cloneTest()

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

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

Input parameters:

- testId: Test id to clone;

Output

```
{
  "status": “OK”,
```

```
  "errors": [ ],
  "response": { TEST_OBJ_SMALL }
}
```

createTest()

This function allows 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 }
}
```

updateTest()

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

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

Input parameters:

- test: TEST_OBJ to update;

Output

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

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": { }
```

```
}
```

Test Manager Module: questionBuilder_IF Interface

questionsList()

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

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

input parameter:

TestId : the id of the Test;

Output

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

getQuestion()

This function returns the question json object that have the id in input.

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

Input parameters:

- complete : true/false (if true the type of the response is QUESTION_COMPLETE_OBJ; QUESTION_OBJ otherwise)

Output

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

createQuestion()

This function allows 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 parameters:

TestId : the id of the Test to add the question

Output

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

updateQuestion()

This function updates the 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 parameters:

- question: QUESTION_OBJ to update;

Output

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

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": { }
}
```

importQuestion()

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

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

Input parameters:

- questionId: the id of the question to clone;
- testId : the id of the current Test where import the question;

Output

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

Test Manager Module: slideBuilder_IF Interface

slideList()

This function returns the list of slides related to the question in input.

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

input parameter :

questionId : the id of the question;

Output

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

getSlide()

This function returns the slide json object that have the id in input.

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

Output

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

createSlide()

This function allows 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 parameters:
questionId : the id of the question to add the slide

Output

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

slideUpdate()

This function updates the 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 parameters:

- slide: SLIDE_OBJ to update;

Output

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

deleteSlide()

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

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

Output

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

cloneSlide()

This function clones the slide passed in input into the current question.

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

Input parameters:

- slideId: the id of the slide to clone;
- questionId: the id of the current question where import the slide;

Output

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

Test Manager Module: testReport_IF Interface

testReportList()

This function returns the list of report object; some input filters could be used.

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

Input parameters:

- reportName (optional) : String used to filter report by Test name ("like" sql function)
- reportDate (optional) : String (dd/MM/yyy format) to filter report by date

Output

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

testReportGet()

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

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

Output:

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

testReportCreation()

This function creates a new report object.

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

Input parameters:

- report: REPORT_OBJ to create;

Output

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

testReportUpdate()

This function updates the report. The id of the report is passed into the path. The new report configuration is passed as json object in input.

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

Input parameters:

- report: REPORT_OBJ to update;

Output

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

testReportDelete()

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

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

Output:

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

Analytics Module: reportGeneration_IF Interface

reportList()

This function returns the list of report object; some input filters could be used.

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

Input parameters:

- reportName (optional) : String used to filter report by report name ("like" sql function)
- reportDate (optional) : String (dd/MM/yyy format) to filter report by date

Output

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

testReportGet()

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

```
@GET
@Path("/report/{Id}")
    @Produces("application/json")
```

Output:

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

testReportCreation()

This function creates a new report object.

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

Input parameters:

- report: REPORT_OBJ to create;

Output

```
        {
         "status": "OK",
         "errors": [ ],
```

```
  "response": { REPORT_OBJ }
}
```

testReportUpdate()

This function updates the report. The id of the report is passed into the path. The new report configuration is passed as json object in input.

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

Input parameters:

- report: REPORT_OBJ to update;

Output

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

testReportDelete()

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

```
@POST
@Path("/report/{Id}")
    @Produces("application/json")
```

Output:

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

Multimedia Manager Module: mediaInterface_IF Interface

mediaList()

This function returns a list of media objects. In input could be set some filters.

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

Input parameters:

- mediaType (optional) : "VIDEO||IMAGE||METADATA||DOCUMENT" Filter of media by type
- mediaName (optiona) : String used to filter media by name ("like" sql function)
- mediaAuthor (optiona) : String used to filter media by author name ("like" sql function)

- mediaDate (optiona) : String (dd/MM/yyy format) to filter media by date

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

mediaCreation()

This function saves the file in input and create 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 parameters:

- file : multipart binary file to upload to the server;

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

mediaDelete()

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

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

Output:

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

mediaGet()

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

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

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

mediaUpdate()

This function updates the media. The id of the media is passed into the path. The new media is passed as json object in input overwriting the existing one.

```
@POST()
@Path("/media/{mediaId}")
      @Consumes("application/json")
      @Produces("application/json")
```

Input parameters:
- media: MEDIA_OBJ to update;

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

Multimedia Manager Module: imageCRUD_IF Interface

imageList()
This function returns a list of image objects. In input could be set some filters.

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

Input parameters:

- imageType : “IMAGE” Filter of media by type
- imageName : String used to filter media by name (“like” sql function)
- imageAuthor (optiona) : String used to filter media by author name (“like” sql function)
- imageDate (optiona) : String (dd/MM/yyy format) to filter media by date

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

imageCreation()

This function saves the file in input and create a new image object representing the uploaded file. The response is the json representation of the image object.

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

Input parameters:

- file : multipart binary file to upload to the server;

Output:

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

imageDelete()

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

```
@DELETE
@Path("media/image/{imageId}")
      @Produces("application/json")
```

Output:

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

imageGet()

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

```
@GET
@Path("/media/image/{imageId}")
      @Produces("application/json")
```

Output:

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

imageUpdate()

This function updates the image. The id of the image is passed into the path. The new image is passed as json object in input overwriting the existing one.

```
@POST()
@Path("/media/image/{imageId}")
    @Consumes("application/json")
    @Produces("application/json")
```

Input parameters:

- media: IMAGE_OBJ to update;

Output

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

Multimedia Manager Module: videoCRUD_IF Interface

videoList()

This function returns a list of video objects. In input could be set some filters.

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

Input parameters:

- videoType : "VIDEO" Filter of media by type
- videoName : String used to filter media by name ("like" sql function)
- videoAuthor (optiona) : String used to filter media by author name ("like" sql function)
- videoDate (optiona) : String (dd/MM/yyy format) to filter media by date

Output

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

videoCreation()

This function saves the file in input and create a new video object representing the uploaded file. The response is the json representation of the video object.

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

Input parameters:

- file : multipart binary file to upload to the server;

Output:

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

videoDelete()

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

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

Output:

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

videoGet()

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

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

Output:

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

videoUpdate()

This function updates the video. The id of the video is passed into the path. The new video is passed as json object in input overwriting the existing one.

```
@POST()
@Path("/media/ video /{ videoId}")
      @Consumes("application/json")
      @Produces("application/json")
```

Input parameters:

- media: VIDEO_OBJ to update;

Output

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

Multimedia Manager Module: documentCRUD_IF Interface

documentList()

This function returns a list of document objects. In input could be set some filters.

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

Input parameters:

- documentType : “DOCUMENT” Filter of media by type
- documentName : String used to filter media by name (“like” sql function)
- documentAuthor (optiona) : String used to filter media by author name (“like” sql function)
- documentDate (optiona) : String (dd/MM/yyy format) to filter media by date

Output

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

documentCreation()

This function saves the file in input and create a new document object representing the uploaded file. The response is the json representation of the document object.

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

Input parameters:

- file : multipart binary file to upload to the server;

Output:

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

documentDelete()

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

```
@DELETE
@Path(“media/ video /{ videoID }'')
      @Produces("application/json")
```

Output:

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

documentGet()

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

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

Output:

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

documentUpdate()

This function updates the document. The id of the document is passed into the path. The new video is passed as json object in input overwriting the existing one.

```
@POST()
@Path("/media/ video /{ videoId}")
      @Consumes("application/json")
      @Produces("application/json")
```

Input parameters:

- media: VIDEO_OBJ to update;

Output

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

Data Abstraction Module: DBManager_IF Interface

connectDB()

This function connects this module to a specific DB.

@POST
@Path("database/connect")
@Consumes("application/json")
@Produces("application/json")

Input parameters:

- connectionString : string used to connect to the database

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

createUserDB()

This function creates a new user in the DB.

@POST
@Path("database/newUser")
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_ID : identifier of the database
- user: username;
- pwd: password

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

createRole()

This function allows to assign role to authorized users.

This function creates a new user in the DB.

@POST
@Path("database/role")
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_ID : identifier of the database
- user_ID: user identifier;
- role: the role of the user

Output:

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

createDB()

This function creates a new database.

```
@POST
@Path("database/createDB")
  @Consumes("multipart/form-data")
      @Produces("application/json")
```

Input parameters:

- database_name: name of the database

Output:

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

createTable()

This function creates a new table.

```
@POST
@Path("database/createTable")
  @Consumes("multipart/form-data")
      @Produces("application/json")
```

Input parameters:

- table_name: name of the table
- column_name: array of names of columns

- column type: array of types {STRING | INT | DATE |FLOAT|CLASS}

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

new()

This function adds a value in a table.

@POST
@Path("database/newData")
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_name: name of the database
- table_name: name of the table
- object: array of value objects to fill table's raw

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

update()

This function updates a value in a table.

@POST
@Path("database/updateData")
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_name: name of the database
- table_name: name of the table
- objects: array of value objects to fill table's raw

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

delete()

This function deletes a value in a table.

@DELETE
@Path(“database/delete”)
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_name: name of the database
- table_name: name of the table
- object: value object to be deleted

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

read()

This function deletes a value in a table.

@GET
@Path(“database/get”)
@Consumes("multipart/form-data")
@Produces("application/json")

Input parameters:

- database_name: name of the database
- table_name: name of the table
- object: value object to be retrieved

Output:
{
"status": “OK”,

```
    "errors": [ ],
    "response": { STATUS_OBJ }
}
```