APPLICATION FORM COMPOSER AND INTERVIEW TOOL

TECHNICAL DOCUMENTATION

Last Modified: 13.01.2010
Author: Guillermo C. Martínez
Email: gcomesana@cnio.es
Team: -
Client: Genetic & Molecular Epidemiology Group

Table of Contents

APPLICATION FORM SYSTEM. TECHNICAL DOCUMENTATION

1. Introduction

This documentation is intended to give a technical description of the questionnaire tool delivered to the Epidemiological Group at CNIO from the technological, architectural and organizational point of view.

The result is a **Java-based web application** which complies with the **Servlet 2.4 specification**, is deployed on a **Tomcat6** application server (ideally it can be deployed on any other servlet 2.4-certified application server, like *JBoss*) and uses as backend data repository a **PostgreSQL 8.3** server. In addition, taking into account the necessary, sometimes complex but continuous database access, **Hibernate3** ORM was used as middleware between the server side code and the database objects. Table 1 shows the layers the application is split in.

From this point on, the document is organized as follows. First, an overview of the MVC *design pattern* used to build the application is commented to give an overview of the application architecture. Then, the overall application description is split in *server side* and *client side*. Server side includes database and application server logic description. Then, files used only on client side (images, css, …) and their organization are described

User Interface Layer	**Web Browser (AJAX)**	
Network Layer	**Tomcat 6.0.18 application server**	
Logic Layer	**Servlet 2.4 spec (servlet, filters, jsp's)**	
Middleware Layer	**Hibernate3 & Annotations**	
Data Layer	**File system**	**PosgreSQL 8.3 RDBMS**

Table 1. Application layers structure

2. Application Structure - MVC (Model-View-Controller pattern)

As many other web applications, this one was written keeping in mind the use of the **Model-View-Controller** design pattern. The principle of this pattern is the *separation of different parts of the application in order to allow to perform changes without affecting only to a small part of the application*.

So, the **model** layer refers to the datamodel of the application; the **view** layer means the user interface (in this case web pages) and the **controller** is the layer that gets the user input to do something in the data model. Or, with different words, MVC breaks the problem into three distinct pieces: model (stores the application's state), view (interprets data in the model and presents it to the user) and controller (processes user input, and either updates the model or displays a new view).

The approach used here is not considered a pure MVC pattern as the boundaries between view and controller are blurred in the J2EE technology used in the application and, as is known, JSPs can

hold both controller and view logic. This decision was adopted in the early development stages but it was evolving to a different implementation approach, by removing code from the new coded JSPs and using servlets as controller and dynamic HTML (javascript and AJAX technology and frameworks) as the view layer and some kind of control logic. With this approach, and by using some standard format to set the communication between server and client, it is possible a completely separation between server and client logic, while the model remains separated.

Lately, the *ExtJs* framework was used to implement dialogs and to help interview configuration proccess in order to avoid semantic errors as much as possible. In a hypothetical future evolution of the tool, this framework would be a good choice to implement logic and control. More information about the framework can be obtained from its website at www.extjs.com.

3. Server side architecture

The server side is the side where the backbone of any web application is found. For this application, the following components are supporting the application:

Database server	**PostgreSQL 8.3 on Ubuntu Linux 2.6, supports the *model***
Application server	**Apache Tomcat 6.0.18 on Ubuntu Linux 2.6, supports most of the *controller***

Following, the server side infrastructure supporting the application is described. This infrastructure is formed by the **datamodel** implementation and the **application server logic**. First, the database *conceptual* and *physical* models (the datamodel) are described. Then, the *application server logic*, which is the controller part of the application, is explained.

3.1 Conceptual data model

The **conceptual data model** describes the data requeriments of a (new) information system using different notations and formal models. So, it can be said the conceptual data model is one of the results of the requirement analysis. The model used is the well known **Extended Entity-Relationship (EER)** model. No matter the database server or implementation technology used in this stage, only the data conceptual requeriments have to be taken into account.

The picture of the EER model is showed in a whole[x]. Besides, as this diagram can be heavy at first sight, three submodels are provided.

3.1.1 Users submodel

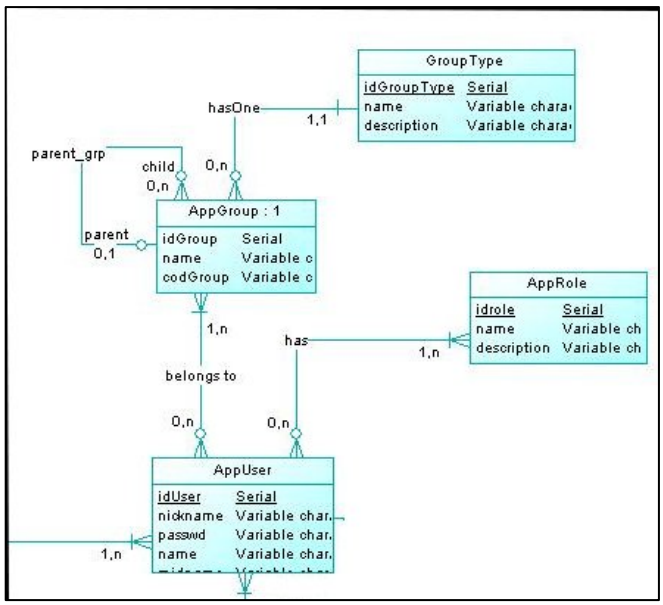

Figure 1. Users conceptual submodel

The model captures the user requeriments for this system. Users have to have a set of users privileges (given by **AppRole** entity), which control the user's available actions in the application; and data access (given by **AppGroup** and **GroupType** entities), which control the set of data an user can access. Besides, the groups can get grouped into hierarchies.

3.1.2 Questionnaires submodel

The following conceptual submodel (blue shaded in the global model) supports the structure of the questionnaires, which means all sections and questionnaire items belong to interview templates or questionnaires and, these ones, belong to projects.

As **Project**, (entity/table name given for questionnaire or interview template), **Section** and **Item** have just a few attributes (*name* and *description* besides foreign keys), it is worth a commentary about the rest of tables in this submodel.

The most important attributes in the **Interview** table are:

- *source*, to know whether or not an interview is cloned from another source interview. This is done to be able to have similar interviews with different languages. There is not features to make changes in a cloned interview if changes are made in the source interview.
- *can_create_subject*, it allows the creation of a subject when performing an interview.
- *can_shorten*, to be able to perform a subset of a whole interview. Implies a justification
- *is_sample_intrv*, as samples are intended to have a different type of identifier, this one (and other features) has to be indicated by this switch

An **Item** is defined like *something* inside a section in a questionnaire. It has the following particular attributes:

- *item_order* is the order of the item in the section, necessary to be rendered properly
- *ite_iditem* is a foreign key representing a container-containee association between items

- *highlight*, a rough definition of decorations to display the item content
- *repeatable* gives the possibility to replicate the item on performance. This one is useful when one question can have several answers.

As *something* is an abstract concept, it has to get concrete in **Texts** and **Questions**. A *Text* is just that, a text in the interview, without possible answer; a *Question* is an Item which has to be answered with some **AnswerItem**. Again, this is an abstract concept (answer item can be an integer, a decimal, a word or a choice from a list). So, in order to get this concept concrete, **AnswerType** and **EnumType** are related. An EnumType object is just a list of choices as answer; an AnswerType is just a simple type (like a lable or number) as answer for some question.

Next the submodel for questionnaires is showed (Figure 2).

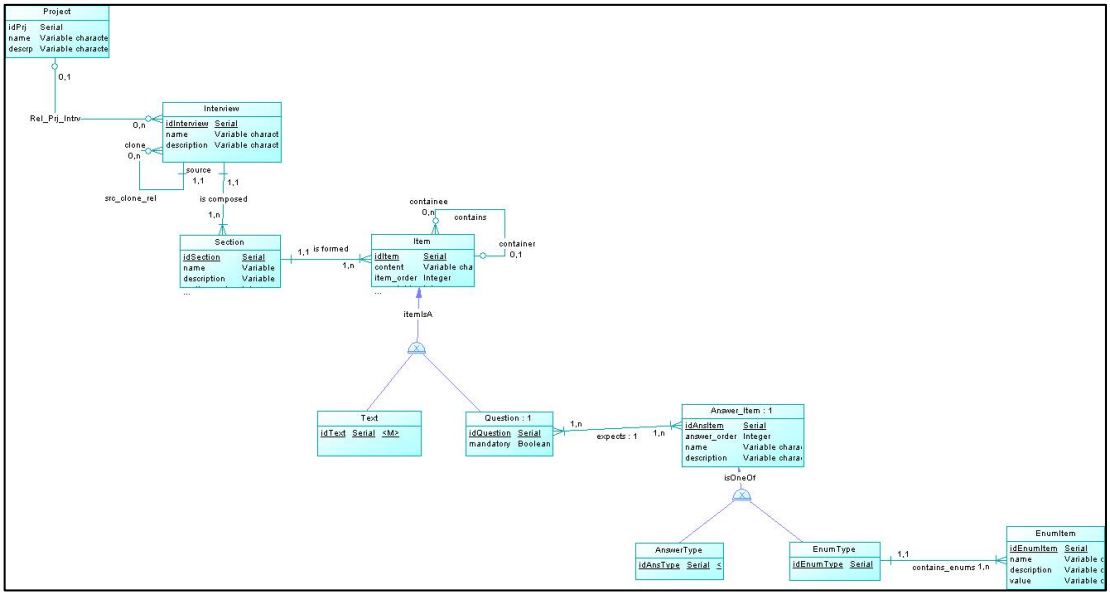

Figure 2. Questionnaire or interview template submodel

An EnumType, as it was said above, is composed by a list of choices. In the database, the choices are represented by the **EnumItem** entity, where one EnumType element can have several EnumItem elements, but one EnumItem element is related to only one EnumType element. The EnumType table has three particular attributes (in addition to primary key id):

- *name*, this is the name will be displayed
- *thevalue* is the value which will be stored on the database if this entry is chosen as answer
- *listorder*, the order of this choice in the list of choices

Remark that an Interview is self-contained, which makes possible to be cloned. This means a questionnaire contains (via relationships) all questions, texts and answer types necessary to be exported or, as it is done, replicated

3.1.3 Performance submodel

This submodel (green shaded in the global model) supports the interviews to the subjects involved in the study (Figure 3). There are two entities that take part in many relationships.

Entity **Performance** represents a questionnaire instance. Actually, it is a quaternary relationship raised to a weak entity. Its meaning can be made out as "*AppUser (an user) performs an Interview*

to a Patient and belongs to AppGroup". The AppUser entity in the relationship is mostly intended for audit purposes. The performances are accessible for the group of users who belongs to AppUser's group. **Patient** can be either a case or a control, and it is the naming convention used by monitors.

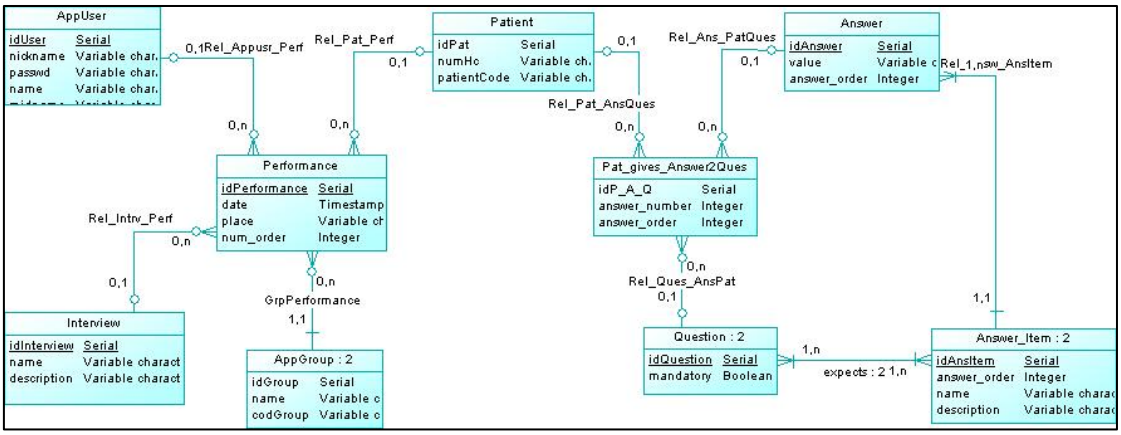

Figure 3. Interview Performance and patient submodel

The other ternary relationship raised to weak entity is **Pat_gives_Answer2Ques**. This one asserts that it "*holds the Answer(s) given by a Patient for each Question from an Interview*". In this way, thanks to the this ternary relationship, every question for every interviewed subjet can be retrieved.

Rest of entities, those ones with a *:2* on the caption, are duplicated entities which are there to clarify the diagram model.

3.2 Physical data model

The **physical data model** is the conversion of the conceptual data model into another model ready to be implemented in a database server. The **relational model** is used here as the previous step to set up the physical model in the database server, in this case a relational database server. By using relational mapping rules, entities and relationships in the EER model are converted into tables in the relational model, which is implemented in the server.

3.2.1 Tables

So, following is the list of relational tables that are part of the database and a comment for each of them describing its purpose.

- **ANSWER**. This table represents the value(s) answered for a question (notice that one question can have several answers for a single subject).
- **ANSWER_ITEM**. This table represents each item of response for a question, considering that any question can be more than one item as response (for instance, month and years, or amount and a dimensional unit as hours, minutes or seconds). It can be seen as a generalized concept of data types answers in the context of the application as generalizes answer types (ANSWERTYPE table) and enum types (ENUMTYPE table, types which are actually enumerated types), both of them are, actually, types of answers.
- **ANSWERTYPE**. Represents simple answer types. Up to date, there are three simple answer types: label (a normal string), number (an integer number) and decimal (a float number).

- **APPGROUP**. A set of users with similar data access boundaries. All users in a group will have access to the same data, but not all of them will have the same rights (which will be given by the role concept/APPROLE table)
- **APPLOG**. This table hold log messages from all over the application.
- **APPUSER**. This table represents the individuals which are going to use the application.
- **ENUMITEM**. The items belonging to a single enumeration type (ENUMTYPE). These items are basically trios (name, value, order).
- **ENUMTYPE**. The type enumeration is intended as a container of a set of discrete values (ENUMITEM).
- **GROUPTYPE**. This is a table to define the group type, which can be, in this case, a country as a main group and a hospital or lab as the secondary group. Larger hierarchies can be set (but they need programming logic to be effective).
- **INTERVIEW**. This table holds the interview templates, which means the questionnaires, opposite to PERFORMANCE interviews, which are intended as the interviews to the study subjects. Three particular attributes of this entity are *can_shorten*, *can_create_subject* and *is_sample_intrv* which are switches indicating that the interview can be performed just by answering a short number of questions (can_shorten); whether or not a new subject can be performed by the interview (can_create_subject) and if the questionnaire is to manage samples (is_sample_intrv).
- **INTRV_GROUP**. Relation between an interview and a group. This is to define which main group (country in this case, could be a company or something more generic) the interview belongs to.
- **ITEM**. Superclass entity which serves as an entity to represent both texts, questions and text/question grouping (item grouping).
- **PAT_GIVES_ANSWER2QUES**. This is a table result of a ternary relationship with the meaning: 'Several patients gives several answers for the questions'. The ternary relationship connects or relates the patients and answers and questions. There are two important attributes here:

  - *answer_number*, for questions with several answers (repeatitivity)

  - *answer_order*, for questions with multiple values for the answer, for ex, a *frequency* value.
- **PATIENT**. A subject involved in the study and who was interviewed.
- **PERFORMANCE**. This can be defined as the *instance* of a questionnaire. The database stores application form models or questionnaires (called *interview* in the database context), but these questionnaires have to be realized (performed) by performing an interview to some subject (patient). So, the realization (performance) of the interview is done by an interviewer (in this context, an user with the appropiate role) to some patient chosen from some target patient/people database for some application form model (chosen from the interview repository or database).
- **PERF_HISTORY**. This is a sort of history log to track the performances for a subject. Every time a performance is started or resumed for a subject (defined by the subject code), information about the performance (such that user, performance, timestamp and justification in the case of short interview) is stored in here

- **PROJECT**. A project is composed by interviews and has several users associated with it
- **QUESTION**. This represents the question in a section in a interview
- **QUESTION_ANSITEM**. One question can expect more than one item as response, even although only one would be the normal situation. Opposite, an answer-item can be present as an answer-item for several questions. So, this table is the result of this M-N relationship
- **REL_GRP_APPUSR**. Typical membership relation between users and groups
- **REL_PRJ_APPUSERS**. Similar to groups and users, this is a relation between the projects and the users which can work on this project
- **ROLE**. The roles will be assigned to the application users. As usual, the roles will allow different capabilities (edit interviews, just viewing interviews,...)
- **SECTION**. A piece a interview is splitted in to contain several related questions and text items
- **TEXT**. This is just an item text. The content of an instance of this entity is the field '*content*' of the item table
- **USER_ROLE**. Relationship to set the roles for a application user

Every table has a *sequence* object (to increase automatically the primary key for each new inserted row) and one or more *index* on columns (index by the primary key, to autoincrement the primary key value, or unique index on columns where the values have to be unique). Along with this set of tables, the database holds the following objects:

3.2.2 Triggers

- **onInsertLog**. This trigger is used in order to get the server correct time to insert into the APPLOG table
- **onLogout**. This trigger logs to the applog table the users' logout in the application. This logout can be automatic as the server session can expire
- **logAnswerUpdate**. This one audits changes in answers
- **onCreateUser**. This one is triggered when a new user is created to set the default very first password

3.2.3 Functions

- **log_appsession_end**. This function gets the previous row in the APPLOG table for the same session id and inserts a new row auditing the user logged out. This function is used by the *onLogout* trigger.
- **set_log_time**. This is a trigger function to insert the correct time value in the '*thetime*' field in the APPLOG table to avoid:

  - nulls from hibernate (when the user session automatically expires)

  - time mismatching between the application server machine and the db machine.
  This function is used in the onInsertLog trigger to avoid the time mismatching.
- **answers_curation**. When a interview is performed in autosaving mode, questions can be left out. But, by requeriments, all questions have to be an answer in the database. This function loops over interviews to fill all missing answers for questions.

- **fill_answers**. This is the function which, actually, insert the value for missing answers from a interview id and a patient id. It does not insert values for repeatable grouped questions.
- **log_upd_answer**. Function used by *logAnswerUpdate* trigger to audit the change in an answer.
- **resetPasswd**. Function used by *onCreateUser* trigger to encrypt and set the very first password.

The rest of the functions on the database come from the PostgeSQL *pg_crypto* library.

3.2.4 Views

- **viewlog**. This object provides a view of the applog table ordered by time descendent.

This model will continue evolving as new requeriments can affect the datamodel.

3.3 Application Server logic

The business logic located in the server is supported by Apache Tomcat 6.0.20, which is a JSP 2.1 and Servlet 2.5 container. So, it is a J2EE based application server (not supporting J2EE full specification). Most of the application code in the server side is part of the controller but there is an object-relational mapping (ORM) which can be considered part of the model.

The server logic application Java source code is organized in packages as follows:

- *org.cnio.appform.entity*, package where (annotated) entity classes are placed
- *org.cnio.appform.util*, package where utility classes can be found
- *org.cnio.appform.util.captcha*, package where *jcaptcha* classes are held
- *org.cnio.appform.util.dump*, database data dump classes. The only interface is command line
- *org.cnio.appform.audit*, package where logging related classes are
- *org.cnio.appform.servlet*, package for servlet implementation classes
- *org.cnio.appform.servlet.filter*, package for servlet filters implementation classes
- *org.cnio.appform.jaas*, package for classes implementing a JAAS authentication service
- *org.cnio.appform.test,* package holding (not formal) test classes to test specific functionality difficult to test on application server. Not included on deployment

On the other hand, many of these classes are used by **JSP pages**, which are organized in several directories (all of them under a common */web* directory) as is showed next:

- *jsp*, main jsp's are here. The only used files here are *index.jsp* and noprofile.jsp and setprimarygrp.jsp.
- *jsp/ajax*. These ones are jsp pages which, actually, act like scripts. All of them are written to communicate back to client *json* strings. This is done to avoid continuous page transitions when many requests have to be sent to server. Here, these jsps are acting in the controller layer.
- *jsp/inc*. These are includes or jsp fragments. The template composing part of the application is made with this jsp fragments.

- *jsp/intrv, jsp/intrv/ajaxjsp*. These two directories are used in the Interview tool, the part of the web application employed for interview performances. The functionality of this part of the application is based on the *ajaxjsp* files, which also act in the contoller layer.
- *getpass*. It is not inside the *jsp* directory and holds classes files involved in the password recovery/change procedure. There are several js files in *getpass/js* subdirectory, but the only important are those named **passyui.js*, *ajaxresponses.js* and *changepasswd.js*.

The base /web directory is in the development structure is not deployed when the war file is yielded, and it is a convenient way to develop the project inside the applications directory of the server only reloading when classes are changed.

Besides all of this, a *build.xml* ant file is provided to produce the war file necessary for deployment. This war is an absolutely standard web application archive file deployable on any Servlet 2.4, JSP 2.0 complaint server. To customize the tasks in the build ant file the variables on the top of the file have to be rewritten.

3.3.1 Object-Relational Mapping

In a nutshell, **object-relational mapping** (**ORM**) is the automated (and transparent) persistence of objects in a application to the tables in a relational database, and/or viceversa, using metadata that describes the mapping between the objects and the database. In this application, this mapping is done by using **Hibernate3**, which is a framework for Java and .Net technologies providing ORM mapping.

In addition to provide an ORM framework, Hibernate provides some more features, as connection pooling or efficient object retrieving among others. Details about Hibernate are out of the scope of this document and there are plenty of resources about it on Intervet.

The *entity classes* (classes which map to relational tables) are located in the *org.cnio.appform.entity* package. Barely business logic is found in this classes (except for replication logic in some classes), but only mappings.

3.3.2 Utility clases

The *org.cnio.appform.util* package holds utility classes, which make heavy use of the entity classes. Classes in this package encapsulates many of the business logic methods used to implement the application functionality. Many of these methods are static as it is not necessary to instanciate classes to perform most of the business logic. The classes in this package are:

- **AppUserCtrl**. This class contains methods to control the application users properties and relationships. Basically, read operations are made on users and user-related objects (groups, roles and projects) in the database (via instantiation as entity class objects). The only write-operations remove an user from a group or a project.
- **HibController**. This class contains business methods and inner classes to control objects of different persistent classes. It has three inner classes:
  - *HibController*.**EnumTypeCtrl**, contains business methods for the enumeration types (EnumItem and EnumType clases)
  - *HibController*.**ItemManager**, contains convenient methods for questionnaire items (texts and questions) management.

- *HibController*.**SectionCtrl**, inner static class encapsulating methods with operations on Section objects.

- **HibernateUtil**. Set of convenient and utility *static* methods, many of them answer and answer types oriented and others doing miscelaneous logic. The most important feature of this class is the initialization of the Hibernate configuration with the annotated entity classes and the creation at startup time of the Hibernate *SessionFactory* object, which is responsible for the Hibernate session creation.
- **IntrvController**. This class contains methods to deal with questionnaire (interview template) tasks such as creation, removing, cloning or assignment to project and users.
- **IntrvFormCtrl**. Business logic used in interview-performance time, mostly related to CRUD operations on performance and answer objects.
- **LogFile**. This is a simple wrapper for the log4j API (logging capabilities). The web application creates 100Kb long log files in `${catalina_home}/logs/appform`. This features can be changed by modifying the *log4j.xml* file in the *WEB-INF* directory.
- **RenderEng**. This inner class contains methods to render the questions and answers in a correct way. This class is employed in the file *items4sec.jsp* to render the forms per section on-demand.
- **TransportSender**. This class encapsulates all neccessary logic to send and email. It is used only when an email has to be sent upon password change or recovery. For inmediate information, the mail sent to users has the following features:
  - mime type is "text-plain/charset=UTF-8"
  - sender is simply noreply@cnio.es
  - the reply to field is null

  The body is a text in english with a embedded URL where the user can change the password.

3.3.2.1 Dump classes

This classes are located in org.cnio.appform.util.dump package. There are just three classes:

- **DBDump** is just a interface for the real dump classes
- **NewRetriever** is the class where the dumping logic is. In order to dump database data correctly, the next steps have to be followed:
  - the questions will be the columns of the table and the rows will display the answer values for a patient for each question. As there will be patients with more answers than others, because of questions with variable number of answers, so there will be rows with larger number of values.
  - so, to accomplish with the previous requierement, first of all, the largest number of answers for all patients has to be worked out, as well as the patient who yielded it
  - from that patient, the header for the dump can be built up
  - then, we can retrieve the answers for each patient and place them in the right column.

  So, the class implents this functionality through three private methods, plus a couple of interface methods (named getdump (…))
- **DataWriter** class writes out a file the rows retrieved by a NewRetriever object.

- **KeyComparator** and **InvKeyComparator** are two key comparator classes to set the order of the header items. There is three elements to set the order: i) the order of the question in the section (as the questions are rendered on screeen); ii) the number of answers for a question (repeatable questions); and iii) the order of the answer items (when a question has to be answered with two or more values). So, the KeyComparator class set the order taking first i) and the ii); and the InvKeyComparator set the order taking ii) then i). Setting exactly the same order than questionnaire rendering is not affordable as, when making a dump so many subjects data are being dumpled, whereas rendering is just for one subject.

3.3.3 Servlets and filters

Servlets are Java scripts that acts as snippets for server programming. They only receive requests, make some processing and return a response to the client. *Filters* are like interceptors: intercept requests matching some rules and perform some actions before the request is relayed to the next filter or to the target servlet resource.

There are four servlets registered in the web descriptor file (web.xml, see [x]). They are located in the *org.cnio.appform.servlet package* and have quite specific functions:

- **AjaxUtilServlet**. This is a servlet which started being used as controller for some *admintool* application. As requirements were growing up, this class got fatter to accommodate more functions, specially related to POST actions such as *removing items* and *saving single answers* when working in *autosaving* mode.
- **JaasServlet**. This servlet is started at application startup time to set up the authentication service (Authentication system, see below).
- **MngGroupsServlet**. Servlet for requests to active a group if the user belongs to several groups, primary or secondary ones. The *doPost (req, resp)* method will be operating here as the database has to be changed.
- **MngPasswordServlet** handles the password change/recovery procedure by checking the captcha value, composing the email and sending back to user and checking new passwords to meet the requirements. All this actions are performed upon receiving ajax requests and send back a json object as response.
- **QryServlet**. This servlet is mostly focused to response GET requests. It will have a bunch of getter methods for any kind of requesto to retrieve information. So, it is mostly a servlet to response queries.
- **ImgCapchaServlet**. This servlet generates a captcha for the client using the *jcaptcha* api.
- **IntrvServlet**. This servlet manages requests and updates the database with the parameters provided by the user at interview configuration time, just before staring the interview.

On the other hand, only a filter is implemented and registered for the application. This filter. *org.cnio.appform.servlet.filter.***AuthenticationFilter** filters all http requests for this application. Its function is to set up session parameters for the user just after signing in and creating the session. In this way, this task is carried out just once in a defined location in the source code.

3.3.4 Authentication system

There are three elements involved in the authentication system, which are:

- Database server, as the user authentication data are taken from the database, this means, users' information is stored there.
- Tomcat 6 application server, provides embedded services to use different types of authentication services.
- The **Java Authentication and Authorization Service**, **JAAS**, by which the authentication process is done.

JAAS allows doing authentication and authorization by using an abstract layer between the application and the underlying authentication mechanisms. For instance, the underlying authentication mechanism here can be seen as the database, but minor change should be made to the entire application in the case of changing that mechanism in order to use, for example, an LDAP server. Figure 4 depict the JAAS high level overview diagram, where the different underlying systems are showed (in this case, RDBMS is used).

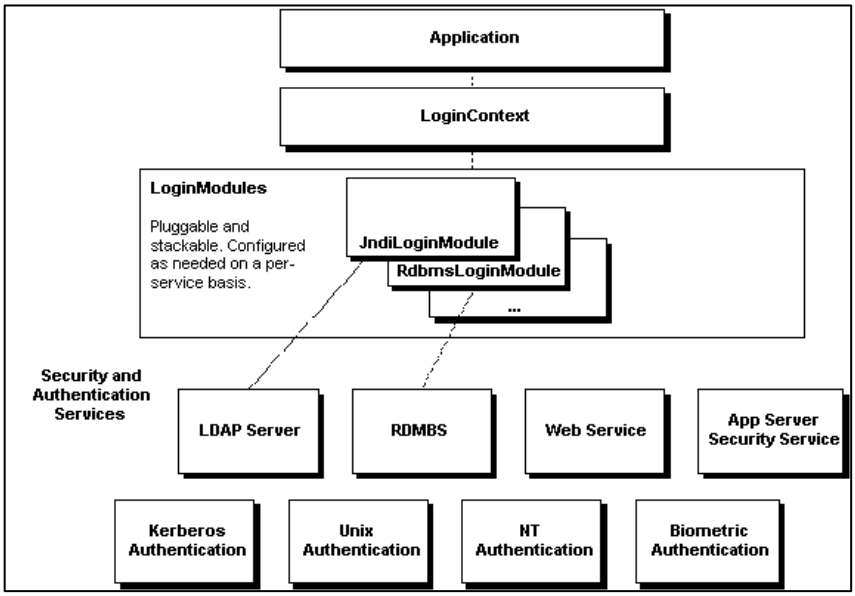

Figure 4. JAAS overview diagram (extracted from http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-jaas.html)

The JAAS related classes are located in the *org.cnio.appform.jaas* package. Besides, the *JaasServlet* servlet class is used as a startup servlet (as defined in the *web.xml* application descriptor file, [x]) to load the JAAS configuration file (as also defined in *web.xml* as a servlet initialization parameter, `<init-param …>`) into the JAAS configuration class. In this configuration file, the implementation of the *LoginModule* to be used and some flags are declared.

A tutorial to know the JAAS how-to can be found in the Javasoft website or from the URL indicated at Figure 4. Regarding to this application, the process is outlined as follows:

- At startup time the initialization method of the JaasServlet is executed, the JAAS configuration file is loaded and the implementation of the JAAS service in the application is initialized. At this time, the authentication system is ready to be used.
- The JAAS realm of Tomcat 6 is used as *authentication realm* (this is defined in *META-INF/context.xml*, only for this application). So, when an username and password is submitted from the application form login, the implementation of the *LoginModule* interface (as is in the JAAS configuration file, *org.cnio.appform.jaas.AppLoginModule* in this case) gets into the scene.

- The *LoginModule* implementation has to implement a *login ()* method. This is the method called back (behind the scenes) to perform the authentication mechanism, which was custom coded to use the application own authentication method. In this case, username-password matching is checked in the database.
- A few more actions are carried out in the *login()* method taking advantage of the flexibility provided for the authentication service, specially some user data update in the database and set credential to use in the just started session.

3.3.4.1 Password recovery

In order to fully comply with the data protection laws it was necessary to implement a recovery password system in the case of the password expires or is lost. The procedure follows the typical steps to recover passwords: the user makes a request to reset the password introducing some "private" parameters (as email); an email is sent back to the user with an url to reset the password; the user introduce the new password. This proccess is depicted in Figure 5.

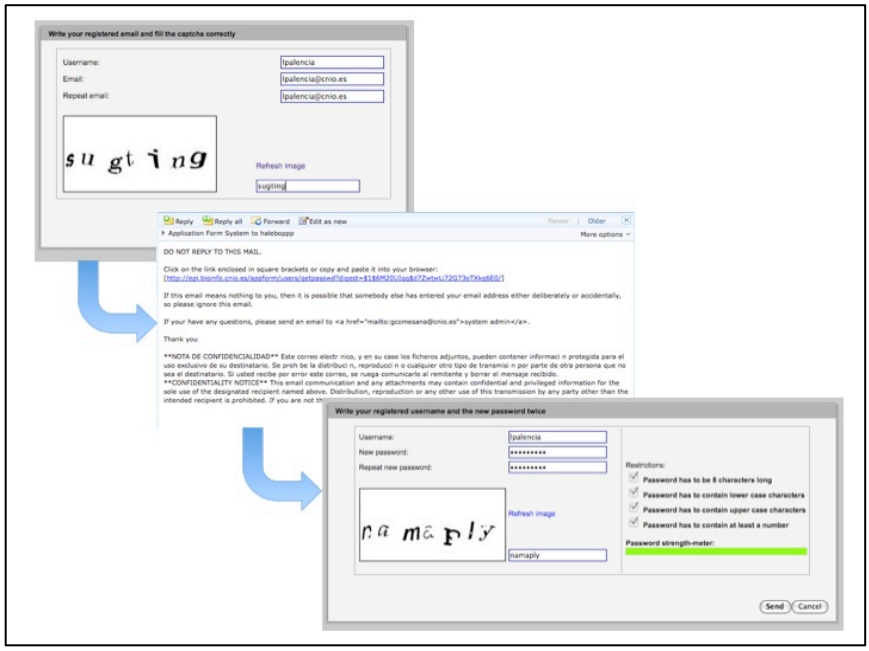

Figure 5. Recovery password process. Top dialog is to request password; then, an email with the URL to reset the password and the dialog to reset it. Here, the password strength bar shows a good password. In both dialogs, captchas can be seen

From a development point of view, all files involved in this process are located under the */appform/web/getpass* directory in the repository. There are just few custom files involved and the process is AJAX based involving both YUI and JQuery libraries. The files involved (all of them under *getpass* directory) are as follows:

- *index.jsp*, is a very simple jsp page which shows the entry dialog to submit in order to get an email with the location of the reset password form. This file is supported by the next file described
- *js/reqpassyui.js*, is the js file which uses YAHOO.widget.Dialog component to render a modal dialog from some component in *index.jsp*. Validation for the form is provided in this file as well as AJAX submission of parameters. In addition, as a captcha image is in the form dialog as an additional control, a method to refresh captcha on user demand is provided (*PasswdReqCtrl.refreshCaptcha()*). The submission URL goes to a servlet to process the request an send an email

- *org.cnio.appform.servlet.MngPasswordServlet* as described in section *3.3.3 Servlets and filters*.
- *resetpass.jsp*, as the index.jsp file, this one, supported by *resetpassyui.js*, contains a form to set the new password. The bit java code in the jsp is to retrieve the user from the encrypted password received as parameter (the digest parameter). There is a dynamic behaviour to check if the new password comply with the constraint rules controlled by the resetpassyui.js file.
- *js/resetpassyui.js,* has the same structure that reqpassyui.js with an additional method *onKeyUp (...)* to process *onkeyup* javascript events when introducing the new password in order to check password restrictions on the fly and display the strength of the new password. The form submission is sent to the same MngPaswordServlet as before.

3.3.5 Audit

The last set of classes are those which do logging or audit actions. These (two) classes are found in the *org.cnio.appform.audit* package. The most important logging information (related to authentication and data manipulation by users) is stored in the database APPLOG table as it can have impact in the data, as well as signing in and out information.

The *org.cnio.appform.audit.ActionsLogger* class is just a convenient class to insert log entries in the database. This class is often used all along the application.

The *org.cnio.appform.audit.AppSessionListener* performs actions to update the user state on session destruction time. Actually, this is a session listener captures creation and destruction events. When the session is destroyed, the database is updated for the user data and, inside the database, a procedure is triggered to insert in the log table a row to audit the user sign out.

3.3.6 Java Server Pages Resources

The Java Server Pages (JSPs) resources are located under the */web/jsp* directory, which has the following structure:

```
web
+-jsp
| +---ajax
| +---inc
| +---intrv
| | +---ajaxjsp
```

Most of them are used in the *Composer tool*, as the *jsp/index.jsp* resource is composed by several jsp fragments (located in the *jsp/inc* directory), which are included in depending on query string parameters. Specifically, every page that can be seen across the Composer tool is formed by different jsp fragments from the *inc* directory. So, the page composition is built up on two files:

- *jsp/index.jsp*. First of all, a main variable is defined with the current type of object (if this request is about project, interview, item…), then, the fragments which define the page layout are included (header, center, left and footer, all jsp pages in *inc* dir). The parameters on query string, which are interpreted on the different fragments, are the following (for composer tool):
  - *t*, the type of item the page is displaying information of
  - *op*, the operation which is to be performed on the page
  - *frmid*, is the database identifier for the item (whether it is project, question, …)

- *spid*, is the container for the current element. This parameter only is present depending on the operations to perform on the item.

The page can be the result of some operation or, just a request to display elements. This means the page can be reached after update or create an element or from the container part -for instance, the list of elements for a section can be reached from the list of sections or, after performing an action, from the element edition-.

- *jsp/inc/header.jsp*. The main logic in this jsp fragment is aimed at the creation of "breadcrumbs" to display the "place" from the root project where the user is working and can go back quicker. The long script for the breadcrumb creation takes most of the logic in the file and it is based on the combination of parameters above described.
- *jsp/inc/center.jsp*. This is a kind of proxy and content page. The proxy capabilities come from include diferent operational jsps depending on the operation and type of element and the roles of the user. Here is a short description of them:
  - *detailprj.jsp* gets the list of interviews for the project identifier parameter.
  - *detailsec.jsp* show details for a section, including its contained items
  - *editelem.jsp* is the main page to edit/compose an section item, which can be a text or a question
  - *editintr.jsp, editprj.jsp, editsec.jsp* are scripts to edit interview, project and sections respectively regarding to names and descriptions.
  - *mngsec.jsp*, lists sections based on an questionnaire identifier. Similar to detailprj.jsp.
  - *nogranted.jsp*, is a error page to report the user is not granted to perform the requested operation.
  - *prjlist.jsp*, lists the projects. It is the first page after loging in and choosing group (if necessary)

On the other hand, there are a bunch of jsp resources under *jsp/ajax* directory which are used as server scripts without presentation purposes. They just perform a particular action (as saving sections, items, reordering items) and return a simple JSON response to client.

3.3.6.1 Interview tool JSP resources

Finally, the *Interview tool* has its own JSP resources. They are located under the *jsp/intrv* directory. Taking into account this is a full AJAX development The *intrv.jsp* just has the basic layout of the page. The *mycodes.jsp* page is to retrieve the codes introduced for an hospital. The resources under the *jsp/intrv/ajaxjsp* directory do the controller tasks and they work like server scripts. Here is a list and short description of the jsp directly involved in the Interview tool, all of them under *jsp/intrv* directory:

- *index.jsp* is the entry point to the interview performance. This page has not HTML code as it just raises a dialog to configure the interview is about to be performed. This dialog is ExtJS based and it is found in */js/intrvctrl/sampledlgcfg.js* or *introdlg.js*.
- *intrv.jsp* is the file which supports the entire layout for the questionnaire after rendering.
- *mycodes.jsp* is a script to show the subject codes used for the active hospital for the current interview, which means, the subjects registered for the current active hospital.

- *setsecondarygrp.js* is currently not used. Actually it raises a dialog to choose the hospital to use with the interview

On the other hand, the jsp files under *jsp/intrv/ajaxjsp* are focused in posting and retrieving data to/from the database. The data can be the answers for the questions, in the case of resuming an interview and/or data to refresh the sections on the left side. These files along with the javascript files in */js/intrvctrl* work closely to provide the dynamic functionality to the Interview tool. This process is described in more detail in **Interview Tool Design & Implementation** section further below. The list of files and short description is as follows:

- *intropage.jsp* renders the first page in the interview. The first page is a bit different as, in previous requeriments, it was which the subject code was input in (subject code is the key data for the interview) and, with current requirements, a special textarea has to be rendered depending on the type of interview (short or normal).
- *items4sec.jsp* renders a section items on demand. If the very first page of a questionnaire is what has to be rendered, the previous script is used; otherwise, the component described in 3.3.6.1.1 is used to render a form with all items (questions and text items) for the current section. The buttons of the bottom of the page are included in this file.
- *saveintro.jsp*; this script is called when saving the first page of every interview. This was decided in this way as special proper components can be placed on this page.
- *saveform.jsp* is the jsp which handles and posts answers for questions into the database. This file does not have logic to save individual items on the fly, which is done in *AjaxUtilServlet* described in the *Servlets and filters* section.
- *secs4intrv.jsp* retrieves the sections for the interview and returns to the client a JSON object with the list of the sections and, in addition, a switch to indicate if the section was visited previously.

3.3.7 Form renderer

As the very first version of the renderer made strong use of hibernate entity classes to access data, the form rendering got too much time to complete. This issue wouldn't be acceptable for most of the users and bottlenecked the application performance. In addition, the most used and targeted part of the application usability was being clearly down-performed.

Therefore, a strong update of the form rendering was decided to carry out. It was found out the key point to increase the performance was to take the database accesses to minimum. In addition, to use a sort of library of package to facilitate the HTML form rendering process, both from the code and performance points of view, was necessary.

In order to minimize database accesses, the use of hibernate was largely discarded in benefit of native queries to retrieve the specific data for the form rendering. This decision have a significant impact on portability, but there are no plans to use another RDMS but Postgesql in the near future.

This decision was taken in order to be able to retrieve all data necessary to render the questionnaire with only one query, avoiding to use hibernate DAOs for this process. But, on the other hand, to render the form properly, much more bussines logic in server side is needed, as the form compounded items are different among them (there are text items, repeatable items, item groups…). In order to facilitate the development and enhance the performance, Groovy, a JVM-based dynamic language, was chosen to write the new piece of code.

Groovy supports a handy tool to render the HTML in an easy way, known as builders and a HTML builder is built-in in Groovy by default (more info about Groovy builders can be found at http://groovy.codehaus.org/Builders). As a result, joining both native queries and Groovy builders, the rendering speed and development have been increased and at the same time the usability was improved.

As the bytecode compatibility between Groovy and Java bytecode is guaranteed, but not many IDEs support both Groovy and Java development, the new form renderer was developed as a separated project and it is added to the main **appform** project as a library (specifically a jar file, **formrenderer.jar**) which can be found at the typical appform/WEB-INF/lib directory. [El proyecto se puede encontrar en el mismo repositorio SVN que el resto del proyecto con nombre FormRenderer.]

3.3.7.1 Bussiness logic

The main class of this new project is *org.cnio.appform.groovy.util.FormRenderer*. This class, along with some other utility classes, basically caches answer types, retrieves the items for the current form and runs the bussines logic to get the form properly rendered.

The answer types are cached by the *org.cnio.appform.groovy.util.TypesCache*, which runs a native query to retrieve the answer types for the current interview the form belongs to and cache them in a convenient way.

The class org.cnio.appform.groovy.util.ItemsFetcher is reponsible for retrieving data from database through native queries. In addtion to answer types (also retrieved by this class) the items for the current form are also fetched by it.

These two resultsets (form items and answer types), along with another resultset with the item groups -no matter the interview subject, different than items retrieved- for the current section the form belongs to, feed the render engine (implemented in *FormRenderer* class). This render engine loops through the entire items resultset to render the form properly based on the items features and the implemented bussines logic. Inside this loop, the Groovy HTML builder plays a key role, as it is, in the end, which yields the HTML markup will be sent to client

The business logic is mostly placed in the FormRender.groovy and FormRenderUtil.groovy files, which are full of comments and log traces in order to make the logic understable. If watched, one can realize the high number of different cases to render based on the resultset (just a set of rows) retrieved from the database. The file org.cnio.appform.groovy.test.integration.FormRenderMock shows an standalone example (very verbose regarding to its output) to properly use the render engine.

4 Client Side Architecture

This side of the application is mainly focused on view layer in the MVC pattern regarding to this application. However, some logic of the controller layer is performed at client side to minimize the server requests.

4.1 Client Side resources

These client resoures are organized, just under the application root directory, as follows:

- *img*, pictures and images files.

- *css*, cascading style sheets files.
- *html*, holding just newtypedlg.jsp which is the application answer items management form.
- *js*, where javascript files are located.
- *js/lib,* holds the ExtJs files.

The CSS and javascript code are who provide the dynamic behaviour for the user interface.

4.2 Dynamic behaviour

The application user interface is so simple as it was developed thinking in minimize the page size regarding to network traffic in any conditions. But, on the other hand, the application (and, so, web pages) needs to be dynamic and friendly enough to allow using it to interviewers in almost any condition.

In order to achieve it, many simple checks and view features make heavy use of AJAX and DHTML. This dynamic functionality is achieved by using javascript and cascading style sheets technologies.

4.2.1 Cascading Style Sheets

Cascading style sheets (CSS) files, which provides visual features across the application, are found under the *css* directory. The main file is *portal_style.css*, which is used all across the application and set the visual style. The rest of css files found under this directory are:

- *display.css*, used by the Questionnaire Comparison tool to display the results of the questionnaires comparison.
- *newtype.css*, used in the answer types form window.
- *overlay.css*, necessary to implement web dialogs using a jquery library.

There is a directory under css, called *theme*, which is again necessary for the web dialog implementation. This is the dialog that can be seen when the user has to select a group.

4.2.2 Javascript

But most of the dynamic behaviour is provided by javascript code and javascript frameworks. Specifically **JQuery 1.3** and **YUI 2.7** and **ExtJs 3.0** are used. All the scripts are found under the *js* directory, and all directories under that one (except *intrvctrl*) are resources for the javascript frameworks. Following is a description of every custom javascript file in the *js* directory:

- *main.js*, this is the main javascript file for the Composer tool. Instantiate objects belonging to custom javascript classes depending on whether or not they are needed. Classes are in the javascript fileas and depending on the side of the application (if editing an interview or a question, for instance) different of these javascript files are included.
- *formitemctrl.js*, contains classes to provide dynamic and checking features when editing a text or question element. This file contains two classes: *FormItemCtrl* and *ElemAjaxResponse*, the latter to host the AJAX callbacks. This is included only when editing questions or texts.
- *listitemctrl.js*, contains classes to provide dynamic behaviour to list of elements (in order to delete, update and/or rearrage) and to raise preview and interview screens. Two classes in it: *ListItemsCtrl*, providing checks and dynamic behaviour and *ListAjaxResponses*, which support the AJAX callbacks.

- *ctrlprj.js*, provides dynamic behaviour and checkings in the project list page and for the interview list (project details) page. As usual, two classes (*PrjFormCtrl* and *PrjAjaxResponse*) are defined in this file.
- *ctrlintr.js*, contains two classes (IntrFormCtrl and IntrAjaxResponse for AJAX callbacks) to provide control and dynamic capabilities for the form on the questionnaire list of sections (questionnaire details).
- *ctrlsec.js*, similar to the two previous ones, provide dynamic functionality for the sections list page and the sections details. Similar structure than the previous ones (SecFormCtrl and AjaxResponse are the classes in this file)
- *core.js*, contains three accesory classes (*Dom*, *Event* and *BrowserDetect*) and initializes switches to control the other custom classes objects, as different javascript files are loaded depending on each part of the application.
- *ajaxreq.js*, this is a wrapper for encapsulating an AJAX request based on the YUI *Connection Manager* component. In this way, handy methods to make AJAX requests are published.
- *intrvctrl/ctrlforms-prot.js*, contains a prototype class to provide dynamic behaviour to the entire Interview tool, along with the support provided by another files. This implementation is based on the *prototype* pattern more than the *module pattern*, which is the pattern used to implement all other files. This is necessary here as this script perform more complex actions and has to be self-reference. This file contains the class *ControlForms* with a bunch of methods to control and provide dynamic behaviour for section transition, answer checking, *real time* answers sending, components colouring, dynamic section form loading and group change (when available).
- *intrvctrl/ajaxresp.js*, contains AJAX callback methods to perform actions like colouring left sections list as they are completed, get the forms for every section and set into the page and or display information messages, all actions on response to the server requests.
- *intrvctrl/repelems.js*, contains only a class (*FormManager*) and two methods (*addElem* and *rmvElem*) and is focused on providing functionality to repeat interview elements for questions with undefined number of answers (as it would be a question for the name of subject's relatives –the number of them is unknown-).
- *intrvctrl/mycodes.js*, contains classes and methods to send an AJAX request to retrieve the codes for the current active hospital group
- *intrvctrl/introdlgactions.js*, contains ajax callback functions, data and validation definitions to use by the *extjs* components on the interview configuration dialog.
- *intrvctrl/introdlg*.js,* contain *extjs* scripts to build the interview configuration dialog when the interview is going to be performed to a subject (not a sample)
- *intrvctrl/sampledlg*.js* are the files containing the logic necessary to initiate and configure a sample-oriented interview. This files contain classes based on *ExtJs 3.0* to display and control a dialog to set the interview for a sample handling.
- the rest of files in those directories are files belonging to the javascript frameworks used to support and simplify the development.

4.2.3 Interview Tool Design & Implementation

This section can be intended as a special section as it is addressed to explain the implementation and design of the *Interview Tool*, as there are many elements which have to be coordinated to provide the dynamic functionality.

This is a completely AJAX based application. Just only two web pages make up the site, and the rest of processing is made by using AJAX calls against JSP's (which could be just servlets) whose response is embedde on a html layer over and over. Lets see in more detail.

4.2.3.1 Interview configuration

As the subject interview became more complex with more constraints than ever before, a dialog is raised to ease the configuration parameters, as group, subject code (tightly related to group) and check for questionnaire constraints (enable to create new subjects in the system or perform short interviews).

This dialog was implemented by using the *extjs* capabilities to build windows, forms and, especially, (cross) component validation. The main file implementing the dialog is *introdlg.js*. The file *introdlgactions.js* contains data definitions, ajax callbacks for the dialog components and validation code necessary for the dialog, entirely defined in *introdlg.js*. The dialog implementation make requests (both GET and POST) to server in order to retrieve and create (when needed) data in database. The middleware in server side to process the request is the *IntrvServlet* servlet.

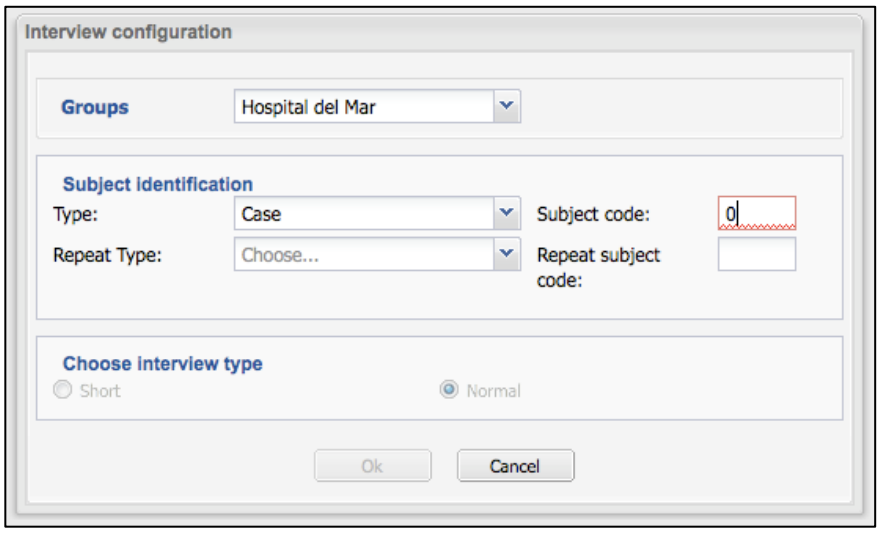

Figure 6. Interview configuration dialog

In the figure above a snapshot of the interview configuration dialog is showed. The red-bordered textfield shows an error input as three digits are needed to satisfy the subject code restriction. Besides, the panel to choose the type of interview is disabled until the subject code is filled and correct (this panel is only showed if the questionnaire allows short interviews). The submit button is also enable when all fields are filled and correct.

Mostly, the heaviest AJAX processing is done when the *Ok button* is clicked. At that time, several AJAX requests are submitted to server to check the restrictions against the database and, if everything is ok, make the changes on the database (insert a new subject and performance if required and, always, saving a new entry in the performance history table).

For more information about stores, validation and component extension check the freely available ExtJs documentation at http://www.extjs.com.

4.2.3.2 Interview performance

When the interview configuration is done, the interview performance starts. It looks mostly as follows:

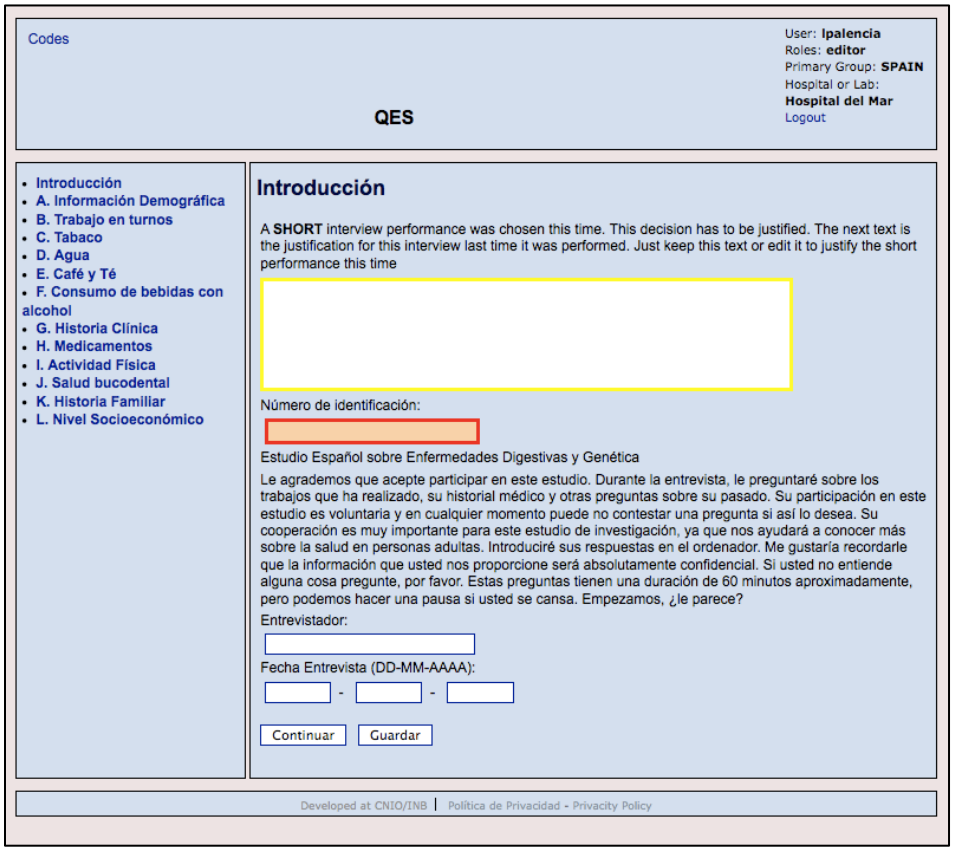

Figure 7. Introduction section for a interview. The yellow-bordered textarea has the focus. The red-bordered textfield shows an error as it has to be filled before committing the section

In the header side, information about the current session in the application is displayed, the name of the questionnaire which is going to be used and, when interviewing, the code for the current subject. A link to show the subject codes currently in database belonging to this hospital is displayed on the left side of header. The footer just holds information about the project and group. The other two areas (left and center, sections and forms) actually hold the interesting things. All four areas are laid out in *jsp/intrv/intrv.jsp* file and all dynamic behaviour is performed by using AJAX and DHTML.

The javscript files involved are *js/intrvctrl/ajaxresp.js* and *js/intrvctrl/ctrolforms-prot.js*. One third file, *js/intrvctrl/repelems.js* is also used, but it is focused on replicate form elements dynamically on questions with undefined number of answers.

When the page *intrv.jsp* is loaded, after configuring the interview and creating objects to control de dynamic behaviour, sections and initial form are loaded (rendered) on the page. The process of section transition when a form is fulfilled is an event-driven proccess.

YUI (and also ExtJs) features custom events definition and creation. So, YUI was used to create two custom events on *IntrvAjaxResp* object creation, which is done on *ControlForms* object initialization (no creation, which is earlier). The events are named *formComplete* and *secsComplete*. On *ControlForms* initializiation, this object is suscribed to these events. When any of Continue or Save buttons on the bottom of any form is clicked, the method *send (`form`)* in ControlForms class is

run, and the *formComplete* event is fired when callback function IntrvAjaxResp.onSaveForm (o) is executed.

When an event is fired, a method of the suscribed objects is then executed. So it is for the ControlForms object. And what the callback method does is load the new section (which is another AJAX call). Approximately the same applies to the form load when a section (in blue font color) is clicked.

All the code is enclosed in two files: *intrv/ctrlforms-prot.js* and *intrv/ajaxresp.js*. Both are close related.

Appendix A. Administration Tool application

The Adminisration Tool was developed as necessary companion to the main application to manage and administer mostly users, but also projects and groups and questionnaire copies. An screenshot can be seen on the next figure.

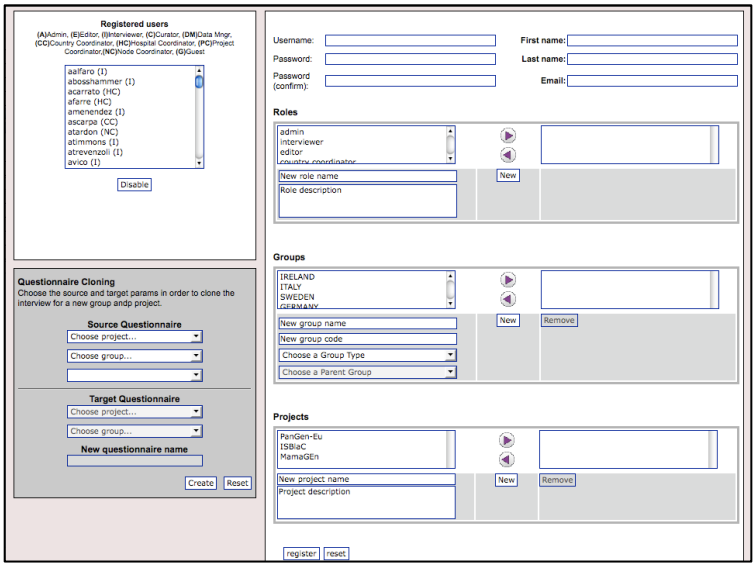

Figure 8. Administrator tool. On the left side, user list (top) and questionnaire copy component (bottom). Right side is the content area, displaying user information

Although it was set as a separate application, the core of this one, which means the entity classes, database schema, utility classes and application server, as well as the authentication system, are the same as the main questionnaire builder application. The main difference is the the client interface and the way it uses to communicate with server: this is done by communicating client and server through JSON objects by using Javascript classes in client side and Java servlets and few Java server pages in server side. So, the client it can be seen is formed by a single JSP page, the rest of JSP pages perform processes for saving and getting data.

The same SVN repository where the files for the questionnaire composer and performer application are placed stores the files for the administration tool. The URL to the root of this project in the SVN repository is http://trac.bioinfo.cnio.es/svn/episrc/admtool.

In order to describe the development of the administration tool, the common parts with the main application will be skipped, focusing in this application specific parts.

A.1 Server logic

Just a few JSPs and servlets are specifically involved. They are found under the *jsp* directory and in the *org.cnio.appform.servlet* package in the source (*src*) directory. The rest of source code is similar to that used in the main application as it was said above.

So, the files can be found under *jsp* directory are:

- *index.jsp*; this is the entry point to the admin tool. Actually, because of authentication restrictions, the entry point is *login.jsp*, which is just in the root (upper) directory. The index.jsp contains the layout can be seen on Figure 7 and few Java snippets embedded into the HTML to

preload the default elements: list of users, roles, groups and projects. No AJAX communication at this point, just page rendering with data retrieved from database by using utility methods. As this is the only webpage renderable, all javascript files (described below) will be loaded in this file.

- *getdata.jsp*; this scripts returns a JSON object with all attributes for an user identified by the param *frmid*, which has to have a valid database identifier for an user. The normal call from the client is */jsp/getdata.jsp?what=usr&frmid=XXXXXX*, XXXXXX a database identifier.
- *admin.jsp*; this script accepts requests for a new user signing up. It performs all necessary checks in order to assure the received parameters match each other in order to keep consistency among the parameters. Returns a JSON object with all attributes stored for the user or a smaller JSON object if there was an error.
- *addelem.jsp;* this scripts add a single element, group, project or role in the database. It returns an usual JSON object to the client
- *swithcdata.jsp*; switches the state of an user from enable to disable or viceversa. A disabled user won't be able to log in the application

A.2 Client logic

As it was mentioned above, the administration tool application is an one-single-webpage application. It performs requests to application server without page transitions via AJAX calls. These calls along with client Javascript logic allows dynamic page components updates instead doing page requests, transitions and updates.

All javascript code is located inside de */js* directory, as showed on the directory tree:

```
+-js
| +---event
| +---jquery
|     +---ui
| +---lib
| |   +---ext
| +---yahoo
```

Four Javascript files hold all client logic for this tool, which are located just in */js* directory. Before describing these four scripts, the directory tree under *js* directory is as follows:

- *event*; contains YUI 2.0 Event utilities to create custom events.
- *jquery*; contains the jquery library, versions 1.2.6 and 1.3 (the version used when this document was writing is 1.2.6). Plus, JQuery plugins Form and Json.
- *ui*; user interface componens for JQuery (unused)
- *lib/ext*; ExtJs library version. Currently not used; future (complex) visual features can be made up using this library
- *yahoo*; YUI 2.0 library directory. There is a custom file, *ajaxreq.js*, which stores the *AjaxReq* custom class that encapsulates all AJAX request programming logic.

Just under js directory, the custom application scripts are located, which are:

- *adminctrl.js;* this file contains one class (*AdmFormCtrl*) and it is the entry point of the javascript execution. On page ready, an AdmFromCtrl object (and an Overlay object) is created. This class has methods to send AJAX requests to create or update users, roles, groups or projects; AJAX methods to retrieve all information from an user and to disable/enable an user, both of this features upon selecting an user from the users list. In order to dynamically refresh on-page visual components (lists, textfields,…) *JQuery 1.2.6* was used and an proprietary AJAX class based on *YUI 2.0 ConnectionManager* was used, as throughout the application, to make the Ajax requests.
- *admin-ajaxresp.js;* this is the counterpoint to the previous file, as it stores all AJAX callbacks for all AJAX methods in *adminctrl.js*. The name of the methods in this file is like onNNNN, where NNNN is an arbitrary name. The structure of all this methods is similar among them. As previously, JQuery is used to update visual elements in the web page.
- *cloningctrl.js;* this file is composed by one class (*CloneFormCtrl*) and controls the small form on the lower left side focused on cloning questionnaires. This class controls the small cloning form works correctly and no request for cloning can be sent with the wrong values and/or number of parameters. There is just two public methods: *init ()* and *fillIntrvSrc ()*. The method *createClone ()* is which send the ultimate request to clone the questionnaire.
- *clon-ajaxresp.js;* As the above mentioned file *admin-ajaxresp.js*, this one is the counterpoint to *cloningctrl.js* file. Which means, it has the callback methods to AJAX requests for cloning questionnaires. Only two public methods, *onGetIntrvs(JSON)* is triggered when a request to get questionnaires based on project and/or group is accomplished; *onCloned(JSON)* is triggered when a request to clone a questionnaire is accomplished as well. The method onFail is raised when the client gets a wrong response from server.

Appendix B. Building, Deploying and Installation from the scratch

A bundle with the source code, documentation and sql scripts for the application form tool can be found in a svn repository through the URL http://trac.bioinfo.cnio.es/svn/episrc/appform (/admtool for the administration tool as described in Appendix A). In this location, the preferences to set up the project in an **Eclipse IDE** environment are included, but it is not guaranteed this settings work fine on every Eclipse environment. For the entire development, the Lomboz distribution (http://lomboz.ow2.org/) was used and it is mostly recommended for a straight set up. For other Eclipse versions, the documentation should be checked.

As it was told on the *Application Server logic* section, a *build.xml* file is supplied to be able to build the *war* file for the application without any IDE. The file also has several targets also to build up jar files for different purposes, but there are two main targets as the deployment is for a pre-production or production environment. The main difference between the deployment tasks in the Ant build file is the *hibernate.cfg.xml* shipped in the war file, which is different for the tasks. In this file, the database server address is defined, while the rest of the application keeps the same. In addition, it can be necessary to update some configuration properties from those found at the top of the file.

B.1 Setting up the source code

While developing the project, the entire application source was placed on the *webapps* directory of Apache Tomcat. When an application war file or 'unfolded' application is dropped under the webapps directory, the application is automatically deployed and started up by Tomcat. In such a way, instant access to new application features as they are added is provided if no code recompilation is necessary (recompilation affects to java code, not JSPs). To do this, just checkout the repository source code into the $TOMCAT_DIR/webapps directory, load or import the project in Eclipse and review the build settings to be sure everything remains correct.

It is possible to place the application code in any other location as well. If this one is the decision and the application wants to be seen through Tomcat, a file called *context.xml* (samples are located in *META-INF* directory) has to be placed under $TOMCAT_DIR/conf/Catalina/*hostname,* where hostname use to be *localhost*. Substitute the docbase attribute for the path where the application is located to get the web application working. The *Eclipse Lomboz* configuration process regarding to some new project location is well-described in the wide Eclipse documentation.

B.2 Build

Although, as mentioned, it is strongly recommended to build the whole project from an IDE (*Eclipse Lomboz* bundle specially recommended as it was said above), it is possible to use *Ant* (http://ant.apache.org) along with a *build.xml* file to build the project. In fact, the final build, the application war file, is yielded by using Ant. In order to build the application with Ant and the build.xml file provided, first thing is to review and configure the properties at the top of the file to match with the environment if necessary.

Then run `ant <target>` as specified in the Ant manual. The supplied build.xml file provides targets to build production and pre-production war files (only difference is the *target server* and *hibernate.cfg.xml* file, so these targets MUST be customized) and to compile the source code. There is NO target to move/copy/transfer the yielded war files to the deployment server machines, but they can be added.

B.3 Deployment

After building it, the yielded war file has to be deployed on a servlet and JSP container in order to work. The only container installed in the INB-Central Node environment is *Tomcat 5.5.x* and *6.0.x*, as mentioned in the introduction, although any other servlet and JSP container (lets say JBoss) could be used, although further custom server configuration could be necessary.

To deploy the application on the server in a easy way, two approaches can be followed:

- just drop the war file on the *$(TOMCAT_HOME)/webapps* directory and it will be automatically deployed. A file named appform.xml will be created in *$(TOMCAT_HOME)/conf/Catalina/localhost/* with the content of the *META-INF/context.xml* file. This is the configuration file for the application and contains the realm used for authentication.
- a server manager can be used for deployment. For Tomcat, Lambda Probe (http://www.lambdaprobe.org/d/index.htm) is recommended, as provides a clean and detailed view of all applications deployed on the server while it is free of charge. To deploy a new war file on the server, just click on the *Deployment* tab and follow in-page instructions.

After this deployment, assuming the *hibernate.cfg.xml* file is properly configured to connect the right database server and it is up and running, the web application should startup and run.

B.4 Deployment from the scratch

The question appears when no database is set to work with the application. So, this section will describe the whole process in the case of this application is to be installed in a completely clean (no previous installations of this software) and new environment.

B.4.1 Setting up the database

As it was said at the first sections of this document, the database server used to work along with the application is *PostgreSQL 8.3 ('postgres' from this point on)*. In order to run successfully the application, a database schema with few initial data has to be set up in advance.

Two SQL scripts are provided to set up the database, which are located under the *sql* directory from repository. A file sql script named appform-schema.sql builds up the database, but it requires the database is created in advance. This script build all database objects into de database: tables, views, triggers, functions, indexes and sequences.

So, starting off a previously untouched postgres server, the sequence of commands to init the database would be:

```
hostname$ psql -h <dbserver> -U <dbuser> -d postgres
Welcome to psql 8.2.5 (server 8.3.8), the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

WARNING:  You are connected to a server with major version 8.3,
but your psql client is major version 8.2.  Some backslash commands,
such as \d, might not work properly.

SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
```

```
postgres=# create database appform with encoding 'UTF-8';
postgres=# \c appform;
postgres=# \i startup-appform-schema.sql
...
```

A few notes on the lines above:

- The user <dbuser> has to be **granted to create databases** and have **superuser** privileges in order to be able to access to *pgsql* language and compile the functions and triggers.
- If the user who is going to access and use the database objects (tables, views, triggers, functions, sequences) is other than the <dbuser> and has no privileges to perform CRUD operations on those database objects (namely tables), privileges on objects has to be assigned to new user by using the grant-privileges.sql script, which contains the pg_grant () function which will assign privileges for the user on all database objects.
- As the application needs the postgres library *pgcrypto* (library used for encrypting sensitive data), errors can be raised if *pgcrypto* functions are already available when running the script or the library itself is not available in the database server. You will have to check your postgres version documentation and installation upon raising this kind of errors.
- The name of the database is arbitrary and the user who owns the database as well, but all these parameters have to match with the database parameters for Hibernate in the configuration file *WEB-INF/classes/hibernate.cfg.xml*.

At this point, the database schema with all necessary objects (structure) is ready, but it is empty. Some init data needs to be preloaded into the database to start working with the form application tool or, at least, with the administration tool. A minimal initialization data script is found in a script file called *startup-appform-data.sql* -which is located just beside the database initialization file- and needs to be run just after the schema initialization, such as:

```
postgres=# \i startup-appform-data.sql
```

The last two commands in the postgres client (*psql*) assume the script files are located in the directory where the psql program was run from.

B.4.2 Configuration custom deployment files

Some files can or must be edited to configure a custom deployment in terms of database names or web application name. These files are as follows:

- **web.xml**. Database parameters can be configured here thorugh context parameters named `dbServerName`, `dbUserName`, `dbPassword` and `dbName` place on the top of the file.
  In addition two servlets can be provided with initialization params:
  - *JaasServlet*, which is the authentication servlet, can be configured through two init params, `max_long_attempts` and `jaasconfig.file` to set properties for the authentication system
  - *MngPasswordServlet*, has two initialization parameters (URLs) `urlPassGet` (url to retrieve a password) and `resetPasswdLoc` to reset the password. As these two parameters are URLs, the *domain* and *application name* (first bit of the path after the domain) have to be properly set.
- **java.util.HibernateUtil.java,** by Oct the 2nd, 2011, the contstant `HibernateUtil.DB_USERNAME` could've to be changed in order to accomodate the database username.

- **js/core.js**. In this file, the name which the application is to be deployed with has to be set through the javascript global variable `APPNAME`, on the top of the file. If wrong, ajax requests won't be calling the proper URLs.

B.4.3 Application(s) deployment

At this point, the database is initialized with a single user ('*adminusr*'), seven roles and the two group types ('*COUNTRY'* and '*HOSPITAL'*) and the next step is to deploy the administration tool (Appendix A). In order to do that, the following steps should be done:

- if no war file was yielded, get the administration tool code from the repository as it was said in Appendix A. The administration tool has a *build.xml* file to be built with Ant. To yield the war file, the same steps used to build the main application form builder are valid
- once the war file is yielded with the correct configuration in META-INF/context.xml and WEB-INF/classes/hibernate.cfg.xml (note the former file has to have configured the docbase and realm and the latter the database and connection names) it can be already deployed on the application server by dropping it on the web applications directory or by using some deployment manager
- suppossing the server has auto-deploy (auto detection of new applications), the administration tool should be ready at this point. The login screen should look like that showed in the Figure 8.
- log in using the only user available and, once inside, create groups and projects and users for those groups and projects with the right roles. The application form builder needs at least **an user with a country and a hospital assigned to him as,** in order to work normally, **no user can work with the form application builder if no group is assigned to it**. This one affects to the default user as well –note the default admin user has **NO** group assigned to, just role–.

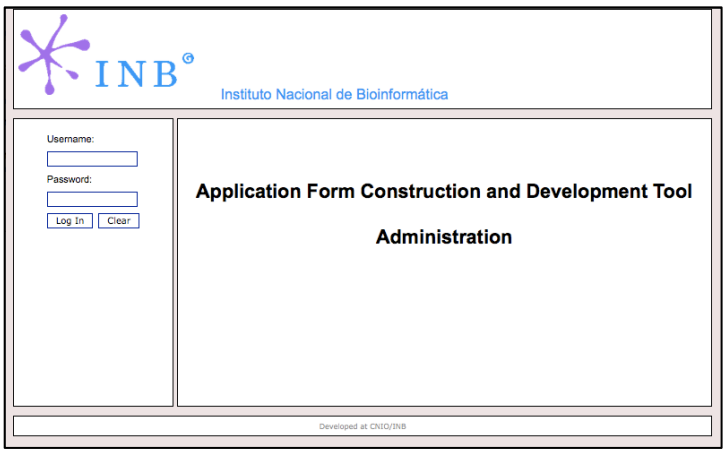

Figure 9. Administration tool entry point

- now the application form builder tool can be deployed (follow the steps to build the application if not built yet). Log in using the administration account or one of the new created users.

Appendix C. Database description report

C.1 Tables description

Date / Time:	**11 December 2009**	User:	**gcomesana**
Database:	**appform**	Table:	**public.answer**

Table: public.answer

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idanswer	integer	Not Null	Yes	Yes	nextval('answer_idanswer_seq'::regclass)
thevalue	varchar(8192)				
Description:	The value which will held the answer				
answer_order	integer				
Description:	This is the order of the answer when the question has several answer items (f.ex. when the question ask a frequency, amount/time units)				
codansitem	integer	Not Null			

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_answer_rel_answ__answer_i	codansitem	public.answer_item	idansitem	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_answer	idanswer	Yes	btree	

Triggers

Trigger Name	A/B	Events	Function	Arguments	Disabled
logAnswerUpdate	after	update	public.log_upd_answer		

Description

This table represent the value(s) answered for a question (pay attention that one question can have several answers for a single patient)

Table: public.answer_item

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idansitem	integer	Not Null	Yes	Yes	nextval('answer_item_idansitem_seq'::regclass)
answer_order	integer				
Description:	Not used				
name	varchar(128)				
Description:	This is the name of the answer item (ex, decimal)				
description	varchar(128)				
codintrv	integer				
Description:	The interview which the answer item belongs to. This is to be able to encapsulate a whole interview template and to be able to use the same answer item with different interviews by cloning them				
for_clone	integer				
Description:	This field says whether or not this answer item was created for a clone interview in a xclusive way or it was created previously. This is done to be able to assign new answer items to a new questions in clones (supposed forbidden...)				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_ansitem_intrv	codintrv	public.interview	idinterview	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_answer_item	idansitem	Yes	btree	

Description

This one represents each response for a question, considering that any question can be more than one item as response (for ex, month and years, or amount and a dimensional unit as time, weight...). It includes answer types (answertype table) and enum types (enumtype table, types which are actually enumerated types)

Table: public.answertype

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idanstype	integer	Not Null	Yes	Yes	
Description:	This is a type which represents a 'simple' type like numbers or labels				
pattern	varchar(256)				
Description:	Not used. Intended to define a fixed pattern for labels				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_answerty_isoneof_answer_i	idanstype	public.answer_item	idansitem	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_answertype	idanstype	Yes	btree	

Description

Represents normal answer types, as label, number...

Table: public.appgroup

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idgroup	integer	Not Null	Yes	Yes	nextval('appgroup_idgroup_seq'::regclass)
name	varchar(128)				
codgroup	varchar(128)				
tmpl_holder	integer				0
parent	integer				
codgroup_type	integer				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_grouptype	codgroup_type	public.grouptype	idgrouptype	Cascade	Set Default		
fk_parent_group	parent	public.appgroup	idgroup	Cascade	Set Default		

Indices

Index Name	On Field	Unique	Method	Function
pk_appgroup	idgroup	Yes	btree	

Description

A group of users. All users in a group will have access to the same data, but not all of them will have the same rights (which will be given by the role concept/table)

Table: public.applog

Fields

Name	Type	Not Null	Unique	P/K	Def Val
logid	integer	Not Null	Yes	Yes	nextval('applog_logid_seq'::regclass)
userid	integer	Not Null			250
Description:	This is the id of the user in the appuser table. Useful to query by user. The default value is the id of the admin user by 08.2009.				
sessionid	varchar(256)				
Description:	This is the application session id. It can be null as sometimes the session id is not available (for scheduled db process, for example)				
thetime	timestamp				
patientid	integer				
intrvid	integer				
logmsg	varchar(1024)				
lastip	varchar(128)				
Description:	This is the last ip got from the ip packet as returned by ServletRequest.getRemoteAddr() method				

Indices

Index Name	On Field	Unique	Method	Function
applog_pkey	logid	Yes	btree	

Triggers

Trigger Name	A/B	Events	Function	Arguments	Disabled
onInsertLog	before	insert	public.set_log_time		
Description:	This trigger simply set the database server time in the NEW row to insert in the applog table				

Description

This table hold log messages from all over the application

Table: public.appuser

Fields

Name	Type	Not Null	Unique	P/K	Def Val
iduser	integer	Not Null	Yes	Yes	nextval('appuser_iduser_seq'::regclass)
username	varchar(128)	Not Null	Yes		
passwd	varchar(128)	Not Null			
c_date	timestamp				now()
u_date	timestamp				
codhosp	integer				
Description:	Not used. The table rel_grp_appusr supplies this information and replaces this one.				
country	varchar(16)				
Description:	Not used. Intended to set the country for the user. Replaced by the relationship rel_grp_appusr				
firstname	varchar(255)				
lastname	varchar(255)				
removed	integer				0
Description:	This field says if the user was removed (actually disabled) from the database. This means it can not access to the app				
loggedin	integer				0
Description:	Indicates in the database when one user is using the application or not. This field is deprecated. Concurrent login is controlled by the singleton pattern (a 'formal' generalization of a static class). Further developments should use the object construction of scala language				
loggedfrom	varchar(128)				
Description:	This field represents the IP address where the user is online from. If the user is not online, will be null				
login_attempts	integer				
Description:	This is a counter to track how many login attemps the user tries and to disable the user if so				
email	varchar(1024)				
Description:	The email of the user. Necessary to contact with him/her, mostly in the case of password loose				
last_passwd_change	date				
Description:	The date of the last password change. It will be the reference to check for the next passwd change				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_hospital_user	codhosp	public.hospital	idhosp	No Action	No Action		

Indices

Index Name	On Field	Unique	Method	Function
appuser_username_key	username	Yes	btree	
pk_appuser	iduser	Yes	btree	

Triggers

Trigger Name	A/B	Events	Function	Arguments	Disabled
onCreateUser	before	insert or update	public.resetPasswd		
onLogout	after	update	public.log_appsession_end		

Description

This entity represents the individuals which are going to use the application.

Table: public.enumitem

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idenumitem	integer	Not Null	Yes	Yes	nextval('enumitem_idenumitem_seq'::regclass)
codenumtype	integer	Not Null			
name	varchar(1024)				
Description:	The name of this item in the enum type				
description	varchar(128)				
thevalue	varchar(128)				
Description:	The value which will be held by this enum item				
listorder	integer				
Description:	The order of this enumeration item in the enumeration type				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_enumitem_contains_enumtype	codenumtype	public.enumtype	idenumtype	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_enumitem	idenumitem	Yes	btree	

Description

The items belonging to a single enumeration type

Table: public.enumtype

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idenumtype	integer	Not Null	Yes	Yes	
numitems	integer				
Description:	Intended to provide the number of enum items for this enumeration type				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_enumtype_isoneof2_answer_i	idenumtype	public.answer_item	idansitem	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_enumtype	idenumtype	Yes	btree	

Description

The type enumeration is intended as a container of a set of discrete values (enumitem)

Table: public.grouptype

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idgrouptype	integer	Not Null	Yes	Yes	nextval('grouptype_idgrouptype_seq'::regclass)
name	varchar(128)	Not Null			
description	varchar(1024)				

Indices

Index Name	On Field	Unique	Method	Function
pk_grouptype	idgrouptype	Yes	btree	

Description

This is a relation to define the group type, which can be, in this case, a country as a main group and a hospital or lab as the secondary group. Higher hierarchies can be set

Table: public.hospital

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idhosp	integer	Not Null	Yes	Yes	nextval('hospital_idhosp_seq'::regclass)
name	varchar(128)	Not Null			
hospcod	integer	Not Null			
c_date	timestamp				now()
u_date	timestamp				

Indices

Index Name	On Field	Unique	Method	Function
pk_hospital	idhosp	Yes	btree	

Description

Not used. This was replace by a combination of appgroup and grouptype elements

Table: public.interview

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idinterview	integer	Not Null	Yes	Yes	nextval('interview_idinterview_seq'::regclass)
name	varchar(128)				
description	varchar(128)				
codprj	integer				
Description:	FK. The id of the project which this interview belongs to				
codusr	integer				
Description:	FK. This is the user which created this interview.				
c_date	timestamp				now()
Description:	Creation date. Not used so far.				
u_date	timestamp				
Description:	Update date. Not used so far.				
source	integer				
Description:	FK. Parent-child relationship to set if an interview is cloned from another. This has to be done as the clone interviews don't have the same properties as the source (parent) interview				
can_create_subject	integer				0
Description:	This is a parameter to define whether the performances for this questionnaire can create new subjects.				
can_shorten	integer				0
Description:	This is a parameter to define whether or not a performance for this interview can be made shorter than normal				
is_sample_intrv	integer				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_cloned_from	source	public.interview	idinterview	Cascade	Set Default		
fk_interview_belongs_user	codusr	public.appuser	iduser	Set Default	Set Default		
fk_interview_is_formed_project	codprj	public.project	idprj	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_interview	idinterview	Yes	btree	

Description

This table holds the interview templates, which means the questionnaires, opposite to performance interviews, which are intended as the interviews to the study subjects.

Table: public.intr_instance

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idinstance	integer	Not Null	Yes	Yes	nextval('intr_instance_idinstance_seq'::regclass)
place	varchar(256)				
date_ini	timestamp				
date_end	timestamp				
codinterview	integer	Not Null			

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_instance_interview	codinterview	public.interview	idinterview	Restrict	Restrict		

Indices

Index Name	On Field	Unique	Method	Function
pk_intr_instance	idinstance	Yes	btree	

Description

(none)

Table: public.intrv_group

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idintrv_group	integer	Not Null	Yes	Yes	nextval('intrv_group_idintrv_group_seq'::regclass)
codintrv	integer	Not Null			
Description:	The id of the interview				
codgroup	integer	Not Null			
Description:	The id of the group which the interview will belong to				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_intrvgroup_group	codgroup	public.appgroup	idgroup	Cascade	Cascade		
fk_intrvgroup_intrv	codintrv	public.interview	idinterview	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_intrv_group	idintrv_group	Yes	btree	

Description

Relation between interviews and a group. This is to define which main group (country in this case, it should be a company or something more generic) the interview belongs to

Table: public.item

Fields

Name	Type	Not Null	Unique	P/K	Def Val
iditem	integer	Not Null	Yes	Yes	nextval('item_iditem_seq'::regclass)
idsection	integer				
Description:	FK. The section id which this item will be inside of				
ite_iditem	integer				
Description:	The parent item for this item in the case of this item belongs to a group of items				
content	varchar(10240)				
Description:	The text for this item, either text item or question				
item_order	integer				
Description:	The order of the item into the section				
c_date	timestamp				now()
Description:	Creation date. Not used so far.				
u_date	timestamp				
Description:	Update date. Not used so far.				
repeatable	integer				
Description:	Indicates whether or not this item (and its subitems) are repeatable in the interview to the subject (ex. for names of the siblings is necessary this functionality as you dont know how many siblings a subject has)				
highlight	integer				0
Description:	A number for indicating if this item has to be highlighted (this means to use bold, italic or underline)				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_item_contains_item	ite_iditem	public.item	iditem	Cascade	Cascade		
fk_item_is_formed_section	idsection	public.section	idsection	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_item	iditem	Yes	btree	

Description

Superclass entity which serves as an entity to represent both texts, questions and text/question grouping (item grouping)

Table: public.pat_gives_answer2ques

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idp_a_q	integer	Not Null	Yes	Yes	nextval('pat_gives_answer2ques_idp_a_q_seq'::regclass)
codpat	integer				
Description:	FK. The patient id.				
codanswer	integer				
Description:	The answer id for this question and this patient				
codquestion	integer				
Description:	The question id.				
answer_number	integer				
Description:	The number of answer in the case of the question is a repeatable question with a undefined number of answers.				
answer_order	integer				
Description:	The answer order in the case this question has several answer items (ex, in the case you have to provide a frequency, which is a number / time unit)				
answer_grp	integer				
Description:	Not used. This is in the case there is several group of question nested each other.				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_pat_give_rel_ans_p_answer	codanswer	public.answer	idanswer	Cascade	Cascade		
fk_pat_give_rel_pat_a_patient	codpat	public.patient	idpat	Cascade	Cascade		
fk_pat_give_rel_ques__question	codquestion	public.question	idquestion	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_pat_gives_answer2ques	idp_a_q	Yes	btree	

Description

Several patients gives several answers for the questions. This is the ternary relationship to connect the patients and answers and questions.
The two attributes are:
- answer_number, for questions with several answers (repeatitivity)
- answer_order, for questions with multiple values for the answer, for ex, frequencies

Table: public.patient

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idpat	integer	Not Null	Yes	Yes	nextval('patient_idpat_seq'::regclass)
name	varchar(256)				
codpatient	varchar(15)	Not Null	Yes		
Description:	This will be the subject identifier to use all along the study and for the tracking interviews. This is the full code for the patient, built with the study code, hospital/group code, case/control digit and the number of patient				
address	varchar(512)				
phone	varchar(20)				
numhc	varchar(32)				
c_date	timestamp				now()
u_date	timestamp				
codprj	varchar(8)				
Description:	The code of the project or study where this patient is involved				
codhosp	varchar(8)				
Description:	The code of the hospital (secondary group) where this subject belongs to				
cod_type_subject	varchar(8)				
Description:	This field indicates if the subject is case (1) or control (2)				
codpat	varchar(8)				
Description:	The patient code as assigned by monitor at interview start				

Indices

Index Name	On Field	Unique	Method	Function
patient_codpatient_key	codpatient	Yes	btree	
pk_patient	idpat	Yes	btree	

Description

The subject who is gonna be interviewed.
The fields here are not commented as it is possible this information is moved to another database

Table: public.perf_history

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idhistory	integer	Not Null	Yes	Yes	nextval('perf_history_idhistory_seq'::regclass)
thetimestamp	timestamp				now()
coduser	integer	Not Null			
Description:	FK for the current user				
codperf	integer	Not Null			
Description:	FK for current performance				
iduser_role	integer				
justification	varchar(8192)				
Description:	In this field a comment for a short interview is set. It is found here as there can be several different justifications on different performances				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
perf_history_fk	codperf	public.performance	idperformance	Cascade	Cascade		
user_history_fk	coduser	public.appuser	iduser	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
perf_history_pkey	idhistory	Yes	btree	

Description

(none)

Table: public.performance

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idperformance	integer	Not Null	Yes	Yes	nextval('performance_idperformance_seq'::regclass)
coduser	integer				
Description:	The user which has done the interview.				
codinterview	integer				
Description:	The interview id which was performed by the user to the subject (patient)				
codpat	integer				
Description:	The subject id (patient id) for the interview which has done by the interviewer (user)				
date_ini	timestamp				
Description:	The starting date of the performance. Mostly not used so far.				
date_end	timestamp				
Description:	The finish date of the performance. Mostly not used so far.				
place	varchar(256)				
num_order	integer				
Description:	Not used.				
last_sec	integer				1
Description:	This is the last section which was last performed. As the interview performance can be resumed, this field is useful to resume from the last point.				
codgroup	integer				
Description:	The group id this performance will belong to				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_performa_rel_appus_appuser	coduser	public.appuser	iduser	Cascade	Set Null		
fk_performa_rel_group	codgroup	public.appgroup	idgroup	Cascade	Cascade		
fk_performa_rel_intrv_intervie	codinterview	public.interview	idinterview	Cascade	Cascade		
fk_performa_rel_pat_p_patient	codpat	public.patient	idpat	Cascade	Cascade		

Index Name	On Field	Unique	Method	Function
perf_unique_intrv-pat-grp	codinterview, codgroup, codpat	Yes	btree	

pk_performance	idperformance	Yes	btree	

Description

This is the instance of a interview.
The database stores application form (interview) models, but these models have to be realized (performed) by making the interview to someone (a patient). So, the realization (performance) of the interview is done by an interviewer for some application form model (chosen from the interview repository or database) to some patient chosen from some target patient/people database.

Table: public.project

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idprj	integer	Not Null	Yes	Yes	nextval('project_idprj_seq'::regclass)
name	varchar(128)				
description	varchar(128)				
c_date	timestamp				now()
u_date	timestamp				
project_code	varchar(255)				

Indices

Index Name	On Field	Unique	Method	Function
pk_project	idprj	Yes	btree	

Description

A project is compound by interviews and has several users associated with it

Table: public.question

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idquestion	integer	Not Null	Yes	Yes	nextval('question_idquestion_seq'::regclass)
repeatable	integer				
codquestion	varchar(16)				
Description:	This is a particular question code for every question in order to discriminate questions when the gross data has to be processed				
mandatory	integer				0
Description:	Not used so far. All questions are mandatory. Further functionality can make use of this field.				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_question_itemisa2_item	idquestion	public.item	iditem	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_question	idquestion	Yes	btree	

Description

This represents the question in a section in a interview

Table: public.question_ansitem

Fields

Name	Type	Not Null	Unique	P/K	Def Val
id	integer	Not Null	Yes	Yes	nextval('question_ansitem_id_seq'::regclass)
codansitem	integer	Not Null			
Description:	The id of the answer item for this question				
codquestion	integer	Not Null			
Description:	The question id which the answer item is related to				
answer_order	integer				
Description:	The answer order for this answer item for the question. Answer items has to be ordered to know which answer item is being used in every case				
pattern	varchar(255)				
Description:	Not used.				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_answer_item	codansitem	public.answer_item	idansitem	Cascade	Cascade		
fk_question	codquestion	public.question	idquestion	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_question_ansitem	id	Yes	btree	

Description

One question can expect more than one item as response, even although only one would be the normal situation. Opposite, an answer-item can be present as an answer-item for several questions

Table: public.rel_grp_appusr

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idgrp_usr	integer	Not Null	Yes	Yes	nextval('rel_grp_appusr_idgrp_usr_seq'::regclass)
codgroup	integer	Not Null			
Description:	The group id for this relationship				
coduser	integer	Not Null			
Description:	The user id for this relationship				
active	integer				0
Description:	This field indicates when a group is CURRENTLY active for the current session for the owner user. 1 means active				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_rel_grp__rel_grp_a_appgroup	codgroup	public.appgroup	idgroup	Cascade	Cascade		
fk_rel_grp__rel_grp_a_appuser	coduser	public.appuser	iduser	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_rel_grp_appusr	idgrp_usr	Yes	btree	

Description

Typical membership relation between users and groups

Table: public.rel_prj_appusers

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idprj_usrs	integer	Not Null	Yes	Yes	nextval('rel_prj_appusers_idprj_usrs_seq'::regclass)
codprj	integer	Not Null			
coduser	integer	Not Null			

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_rel_prj__rel_prj_a_appuser	coduser	public.appuser	iduser	Cascade	Cascade		
fk_rel_prj__rel_prj_a_project	codprj	public.project	idprj	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_rel_prj_appusers	idprj_usrs	Yes	btree	

Description

Similar to groups and users, this is a relation between the projects and the users which can work on this project.

Table: public.role

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idrole	integer	Not Null	Yes	Yes	nextval('role_idrole_seq'::regclass)
name	varchar(128)				
description	varchar(128)				
c_date	timestamp				now()
u_date	timestamp				

Indices

Index Name	On Field	Unique	Method	Function
pk_role	idrole	Yes	btree	

Description

The roles will be assigned to the application users. As usual, the roles will allow different capabilities (edit interviews, just viewing interviews,...)
This table can be expanded in the case of implementing a theoretical RBAC (Role-Based Access Control) model.

Table: public.section

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idsection	integer	Not Null	Yes	Yes	nextval('section_idsection_seq'::regclass)
name	varchar(128)				
description	varchar(128)				
section_order	integer				
Description:	The order of the section in the interview				
codinterview	integer				
Description:	The interview where this section is contained in. This is, the parent interview				
c_date	timestamp				now()
Description:	Creation date. Not used so far.				
u_date	timestamp				
Description:	Update date. Not used so far.				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_section_is_formed_interview	codinterview	public.interview	idinterview	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_section	idsection	Yes	btree	

Description

A piece which a interview is splitted in to contain several related questions. This is the normal way to compose an interview template.

Table: public.text

Fields

Name	Type	Not Null	Unique	P/K	Def Val
idtext	integer	Not Null	Yes	Yes	nextval('text_idtext_seq'::regclass)
highlighted	integer				
Description:	Not used. Instead, the highlight field in the item table.				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_text_itemisa_item	idtext	public.item	iditem	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_text	idtext	Yes	btree	

Description

This is just an item text. The content of this element is the field 'content' of the item table

Table: public.user_role

Fields

Name	Type	Not Null	Unique	P/K	Def Val
iduser_role	integer	Not Null	Yes	Yes	nextval('user_role_iduser_role_seq'::regclass)
coduser	integer	Not Null			
Description:	The id of the application user				
codrole	integer	Not Null			
Description:	The id of the application role				
username	varchar(128)				
Description:	This is necessary to implement JDBCRealm in Tomcat. It is redundant, as the username field is in the appuser table				
rolename	varchar(128)				
Description:	This is necessary to implement JDBCRealm in Tomcat. It is redundant, as the rolename field is in the role table				

Foreign Keys

Foreign Key Name	On Field	FK Table	FK Field	On Update	On Delete	Defer	Moment
fk_userrole_role	codrole	public.role	idrole	Cascade	Cascade		
fk_userrole_user	coduser	public.appuser	iduser	Cascade	Cascade		

Indices

Index Name	On Field	Unique	Method	Function
pk_user_role	iduser_role	Yes	btree	

Description

Relationship to set the roles for a application user.

C.2 View description

View: public.viewlog

DDL

```
[redacted] "public"."viewlog" (
[redacted],
[redacted],
[redacted],
[redacted],
[redacted])
[redacted]
[redacted]
    applog [redacted]
[redacted]
```

```
[redacted] "deleteRow" [redacted] "public"."viewlog"
[redacted] ( [redacted] applog [redacted] applog. [redacted] = [redacted] . [redacted] applog. [redacted]
= [redacted] . [redacted] applog. [redacted] ::text = [redacted] . [redacted] ::text) [redacted]
```

```
[redacted] "public"."viewlog"
[redacted] 'This view shows the last entries in the applog table, this means, the last
logged entries' [redacted]
```

Description

This view shows the last entries in the applog table, this means, the last logged entries

C.3 Functions and triggers description

Function: public.answers_curation()

DDL

```
"public"."answers_curation" () integer
integer
integer
integer
varchar
:= 0
:= 0
:= 'Starting answer values curation process for all questionnaires'
applog ( , , )
( (), , )
, performance
1, 2
:= fill answers ( . , . )
( > 0)
:= +
:= + 1
:= 'End of answer values curation procces. Interviews updated: '
:= '. Rows updated: '
applog ( , , )
( (), , )
```

```
        'plpgsql'

                    "public"."answers_curation"()
   'This functions gets all interviews from performance table and, for each of the
rows, performs the fill_answers function. This is the function who "curate" the
database taking care of filling missing answers. Missing answers will occur mostly
during real time performances'
```

Description

This functions gets all interviews from performance table and, for each of the rows, performs the fill_answers function. This is the function who "curate" the database taking care of filling missing answers. Missing answers will occur mostly during real time performances

Function: public.fill_answers(intrid integer, patid integer)

DDL

```
"public"."fill_answers" (integer, integer)
integer
integer
integer
integer
integer
varchar

(integer, integer)
.  ,  .  ,  .  ,
.  ,  .
interview , section , item , question , question_ansitem
.  =  .
.  (  )
.  =  .
.  =  .
.  =  .
.  ,  .  ,  .  ,
.  ,
.
pat_gives_answer2ques , question , answer , question_ansitem , item
.
(  .
interview , section , item , question
.  =  .
.  =
.  =  .
```

[redacted]

[redacted]

[redacted]

[redacted]

```
[redacted]
[redacted]
[redacted]
[redacted] := '' [redacted] -1) [redacted] ' missing values were curated for the
questionnaire with id ' [redacted]
[redacted] := [redacted] ' and subject with id ' [redacted]
[redacted] applog ( [redacted] , [redacted] , [redacted] , [redacted] , [redacted] )
[redacted] ( [redacted] (), [redacted] , [redacted] , [redacted] , [redacted] )

[redacted]
[redacted]

--    cursor_count := cursor_count + 1;
[redacted]
[redacted] := [redacted] + [redacted]
-- inserting new values into the answer and pag_gives_answer2ques table
[redacted] answer ( [redacted] , [redacted] , [redacted] , [redacted] )
[redacted] ( [redacted] , '9999', [redacted] . [redacted] , [redacted] . [redacted] )
[redacted]
[redacted] pat_gives_answer2ques ( [redacted] , [redacted] , [redacted] ,
[redacted] , [redacted] )
[redacted] ( [redacted] , [redacted] , [redacted] . [redacted] , [redacted] . [redacted] , 1)
-- end of inserting new rows

[redacted] := 'last testId: ' [redacted] :: [redacted] ': '
[redacted] . [redacted] ',' [redacted] . [redacted] ',' [redacted] . [redacted] ','
[redacted] := [redacted] ',' [redacted] . [redacted]
-- insert into test (content) values (testvar);

[redacted] := [redacted] + 1
[redacted]
[redacted]
[redacted] ('answer_idanswer_seq', [redacted] ( [redacted] )) [redacted] answer
[redacted] ('pat_gives_answer2ques_idp_a_q_seq', [redacted] ( [redacted] ))
[redacted] pat_gives_answer2ques
[redacted]

[redacted] -- end loop
[redacted]
```

```
                    -1
    -- function

        'plpgsql'

                    "public"."fill answers"(       integer,       integer)
   'This function takes as parameters identifiers for subject and interview and
scans the interviews for unanswered questions, this is, questions without any row in
answer table. For that set of questions, inserts a new row in answer table with the
value 9999 (default value for missing answers).
Returns the number of rows inserted and audit the process beginning and end'
```

Description

This function takes as parameters identifiers for subject and interview and scans the interviews for unanswered questions, this is, questions without any row in answer table. For that set of questions, inserts a new row in answer table with the value 9999 (default value for missing answers).
Returns the number of rows inserted and audit the process beginning and end

Function: public.log_appsession_end()

DDL

```
                                "public"."log_appsession_end" ()
/*
  OLD and NEW are the row updated on appuser table
*/
            applog
          varchar
          timestamp
           varchar
          (       = 'UPDATE')
                                        *       applog
                  =    .                        <> ''

                  (    .          = 0          .          = 1)
                  =       ()
                  =          (        , 'DD Mon YYYY HH24:MI:SS')
                    = 'User \''||NEW.username||'\' logged out sucessfully at
'

                          applog (        ,            ,        ,          ,        )
                  (         .        ,          .          ,        ,      (),
        .       )

          'plpgsql'
```

```
                    "public"."log_appsession_end"()
    'This function gets the previous row in the applog table for the same session id
and inserts a row logging the user logged out. This function is used by the onLogout
trigger'
```

Description

This function gets the previous row in the applog table for the same session id and inserts a row logging the user logged out. This function is used by the onLogout trigger

Function: public.log_upd_answer()

DDL

```
"public"."log_upd_answer" ()
applog
varchar
varchar
integer

integer

( = 'UPDATE')
. , . , .
. , . , .
. , .
question , pat_gives_answer2ques , answer , item ,
section , interview , patient , performance , appuser
,
perf_history
. = .
. = .
. = .
. = .
. = .
. = .
. = .
. = .
. = .

=
```

```
        (            = 1)
                                    .*
            applog
              .        =        .
                  .
```

```
              = 'Value for question '          .          ' from interview '
              =                    .              ' updated: OLD='        .
              =            ' -> NEW='        .              ' for patient with code: '
              =                    .            ' by user '''            .            ''''
```

```
        (            .                        )
                    =            ' and sessionid is: '            .
                    applog (        ,            ,        ,            ,        ,        ,
        )
                    (        .        ,            .        ,        (),        .        ,
                                                .        ,        ,            .        )
```

```
                    applog (        ,            ,        ,            ,        ,
        ,        )
                    (        .        ,'-1',        (),        .        ,
                                                .        ,        ,            .        )
```

```
    else
      insert into test (content) values ('Nothing was updated');
*/
        end if;

  end if; -- if TG_UPDATE
  return null;

END;
$body$
LANGUAGE 'plpgsql' VOLATILE CALLED ON NULL INPUT SECURITY INVOKER;
```

Description

(none)

Function: public.resetPasswd()

DDL

```
[redacted] "public"."resetPasswd" ()
[redacted]
[redacted]
[redacted] varchar
[redacted] varchar

[redacted] appuser [redacted]
[redacted]
[redacted]
[redacted] ( . = . )
[redacted] = .
[redacted] = crypt ( . , gen_salt('md5'))
[redacted] . =
[redacted]
[redacted]
[redacted] ( . <> . )
[redacted] = crypt ( . , gen_salt('md5'))
[redacted] . =
[redacted]
[redacted]
[redacted]
[redacted]
[redacted]
[redacted] 'plpgsql' [redacted]
```

Description

This function reset the password in the database for a user by setting it to the username

Function: public.set_log_time()

DDL

```
[redacted] "public"."set_log_time" ()
[redacted]
[redacted]
[redacted] timestamp
[redacted]
[redacted] = ()
[redacted] . [redacted] =

[redacted]
[redacted]
[redacted]
[redacted] 'plpgsql'

[redacted] "public"."set_log_time"()
[redacted] 'This is a trigger function to insert the correct time value in the ''thetime'' field in the applog table to avoid:
[redacted] - nulls from hibernate
[redacted] - time mismatching between the application server machine and the db machine

[redacted] The function returns null to skip the current operation on the row using a row-level trigger'
```

Description

This is a trigger function to insert the correct time value in the 'thetime' field in the applog table to avoid:
- nulls from hibernate
- time mismatching between the application server machine and the db machine

The function returns null to skip the current operation on the row using a row-level trigger

Trigger: logAnswerUpdate on public.answer

DDL

```
"logAnswerUpdate"
"public"."answer"
"public"."log_upd_answer"()
```

Description

(none)

Trigger: onCreateUser on public.appuser

DDL

```
"onCreateUser"
"public"."appuser"
"public"."resetPasswd"()
```

Description

(none)

Trigger: onInsertLog on public.applog

DDL

```
"onInsertLog"
"public"."applog"
"public"."set_log_time"()

"onInsertLog" "public"."applog"
'This trigger simply set the database server time in the NEW row to insert in the applog table'
```

Description

This trigger simply set the database server time in the NEW row to insert in the applog table

C.4 EER Model diagram

Take into account this diagram (and the other database figures all along this document) were generated using Sybase Powerdesigner 12. Files **.cdm* are ready to be loaded on this program to make reports and evolve the schema. In addition, this is an high level view of the database. For a detailed view, the previous sections in this appendix should be refered.

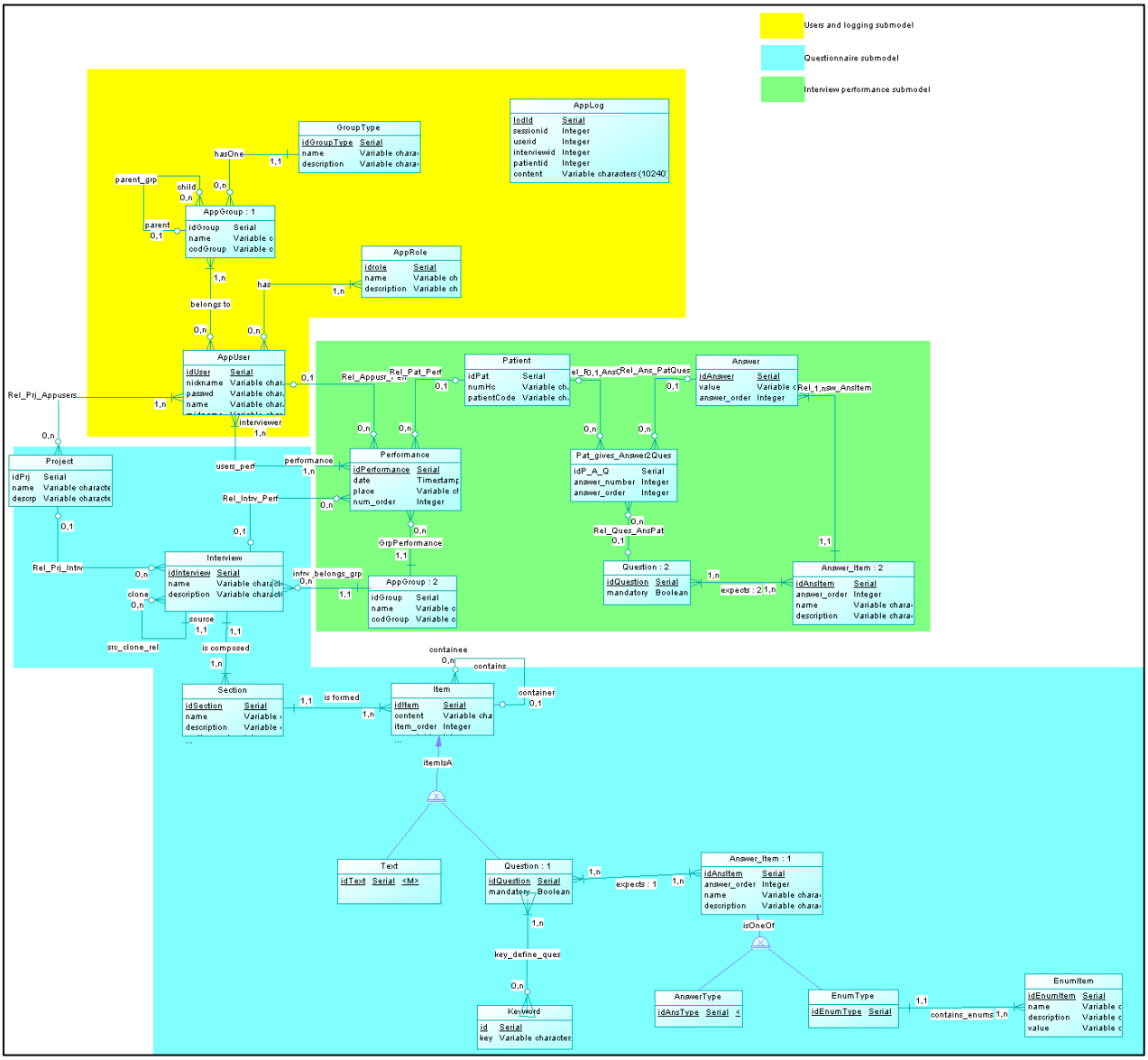

Figure 10. Full EER model