

# SYSTEM DESIGN

---

D18

IDEALVis Consortium

<http://idealvis.inspirecenter.org/>



**European Union**  
European Regional  
Development Fund



**Republic of Cyprus**



**Structural Funds**  
of the European Union in Cyprus



**ΙΔΡΥΜΑ  
ΕΡΕΥΝΑΣ ΚΑΙ  
ΚΑΙΝΟΤΟΜΙΑΣ**

# Executive Summary

This task will design the architecture of the IDEALVis platform by dictating the technical infrastructure, standards, specifications and strategies to be followed throughout the development, testing, and implementation of the platform. Hence, this task will start early in the project accompanying the work of the technical WPs (WP4, WP5, WP6). A number of desirable software qualities for all components will be proposed, including performance, correctness, robustness and reliability. Particularly, the desired software qualities that we will pursue could be summarized as modularity, expandability, interoperability, openness and security. The architecture design will adopt the SOA paradigm in order to ensure a high degree of flexibility and transparency in developing the various diverse technological components required by the IDEALVis platform. Additionally, this approach will promote re-usability of the different low-level services (e.g., acquisition of user model characteristics), which will be utilized for the efficient composition of high-level services. A conceptual view of the architecture as it is envisioned by the consortium can be seen in the diagram on the right.

# Table of Contents

<b>EXECUTIVE SUMMARY .....</b>	<b>2</b>
<b>LIST OF FIGURES .....</b>	<b>5</b>
<b>SYSTEM OVERVIEW .....</b>	<b>6</b>
Use Case Analysis .....	6
System Administrator .....	6
Admin.....	6
Data Protection Officer (DPO).....	7
Researcher .....	8
Participant.....	9
<b>SYSTEM DESIGN .....</b>	<b>11</b>
Introduction.....	11
Overall Architecture .....	11
Database.....	12
Database API.....	13
User-Defined Data Types .....	13
Views.....	14
Functions.....	14
Stored Procedures .....	15
Business Layer API .....	16
Class Library .....	16
Service Methods .....	17
Web Service API.....	18
Web Application .....	18
Overall Web Application Structure .....	18
Web API Communication .....	21
Authentication .....	22
Localization .....	23
Research Studies .....	24
Joining Research Studies.....	25
Dynamic Creation of Tests / Questionnaires .....	26
Security and Data Protection.....	30

**APPENDIX 1: DATABASE DICTIONARY..... 31**  
**..... 31**

# List of Figures

Figure 1: Use Cases for System Administrator Actor .....	6
Figure 2: Use Cases for Admin Actor .....	7
Figure 3: Use Cases for DPO Actor .....	8
Figure 4: Use Cases for Researcher Actor .....	9
Figure 5: Use Cases for Participant Actor .....	10
Figure 6: System Architecture .....	11
Figure 7: The Database Design of the System .....	12
Figure 8: Main MasterPage .....	19
Figure 9: Resource Manager Table Values .....	23
Figure 10: RESEARCH_STUDY Table .....	25
Figure 11: RESEARCH_TEST Table .....	25
Figure 12: RESEARCH_STUDY Relationships.....	25
Figure 13: Join Research Study Page (Logged Out) .....	25
Figure 14: Join Research Study Page (Logged In) .....	26
Figure 15: List of Available Research Study Tests.....	27
Figure 16: Research Test Steps.....	27
Figure 17: Questionnaire / Test Resource Tables.....	29
Figure 18: Research Test Generation Architecture .....	29
Figure 19: Security Architecture.....	30

# System Overview

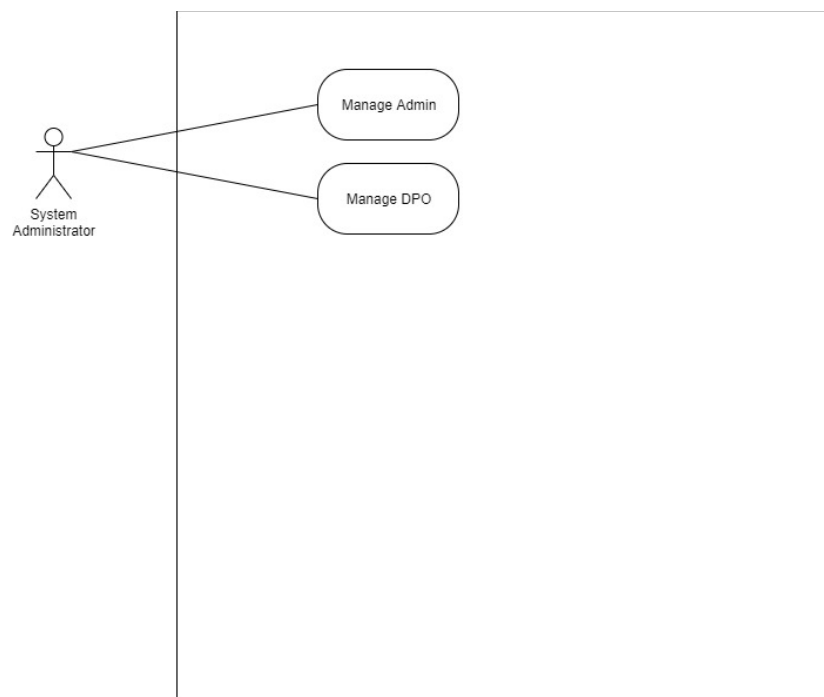
This section acts as an introductory section before the System Design section. Here we describe some use case scenarios that illustrate some of the important user role interactions within the system. By viewing this information prior to the System Design, will help the reader better understand the upcoming subjects like the interconnections and the overall architecture of the system.

## Use Case Analysis

This section presents various use case analyses aiming to identify the requirements of the system and define the main actors and processes within the system. The use case analyses will form the foundation upon which the final system will be implemented.

### *System Administrator*

The *System Administrator's* primary function relates to managing Admin and DPO profiles which includes processes for: *a)* creating new profiles; *b)* viewing data of existing profiles; *c)* editing data of existing profiles; and *d)* deleting profiles.



*Figure 1: Use Cases for System Administrator Actor*

### *Admin*

The *Admin*'s primary functions include: *a)* viewing system statistics in a dashboard page for visually tracking key performance indicators (KPI), metrics and key data points; *b)* managing notification types *i.e.*, create and view notifications; *c)* managing researchers, *i.e.*, select active and inactive researchers; *d)* viewing researchers; and *e)* managing settings, *i.e.*, view, edit, reset various settings related to general, privacy and security.

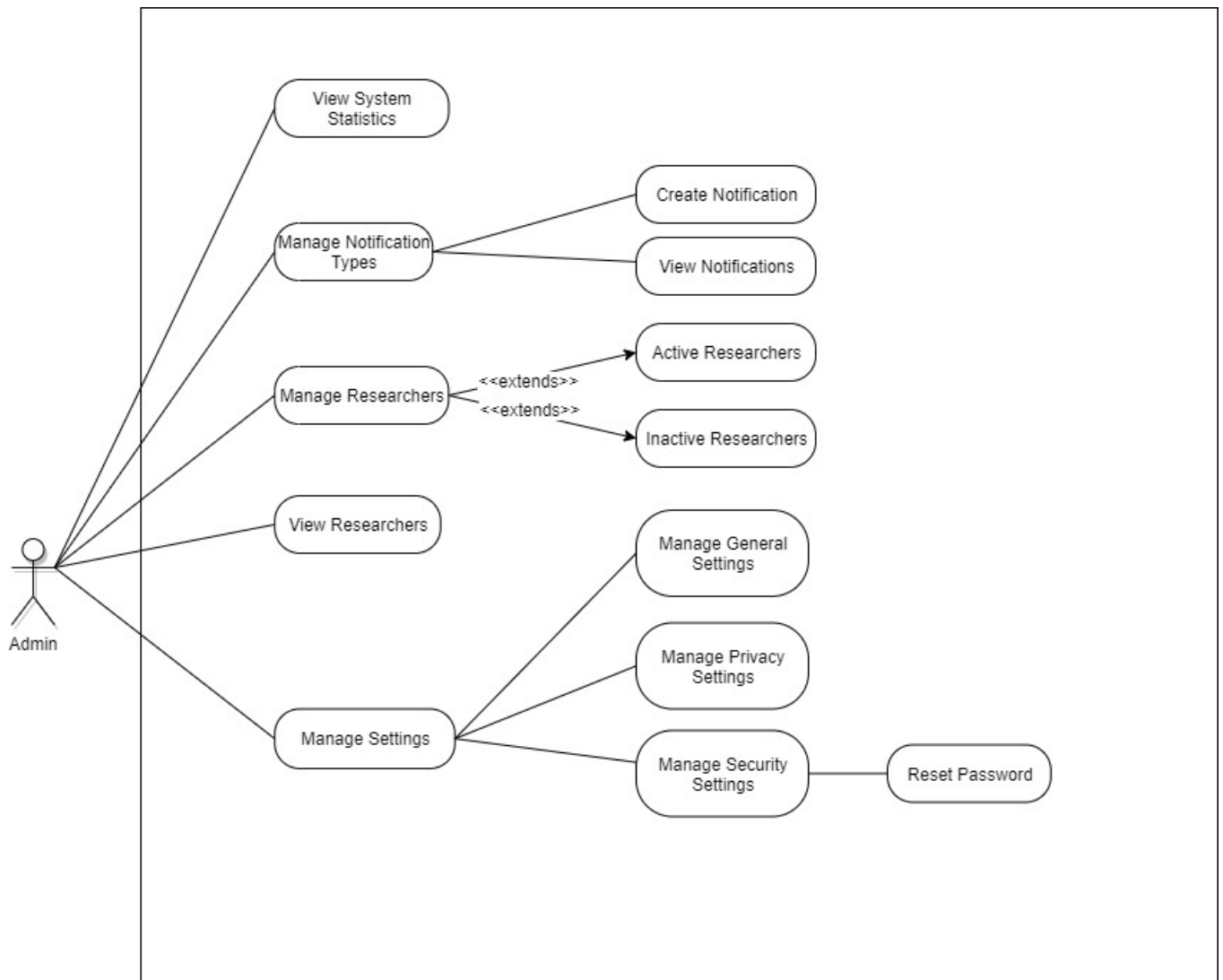


Figure 2: Use Cases for Admin Actor

The DPO's primary function relates to managing privacy claims, *i.e.*, clear erasure request and *stop processing*.

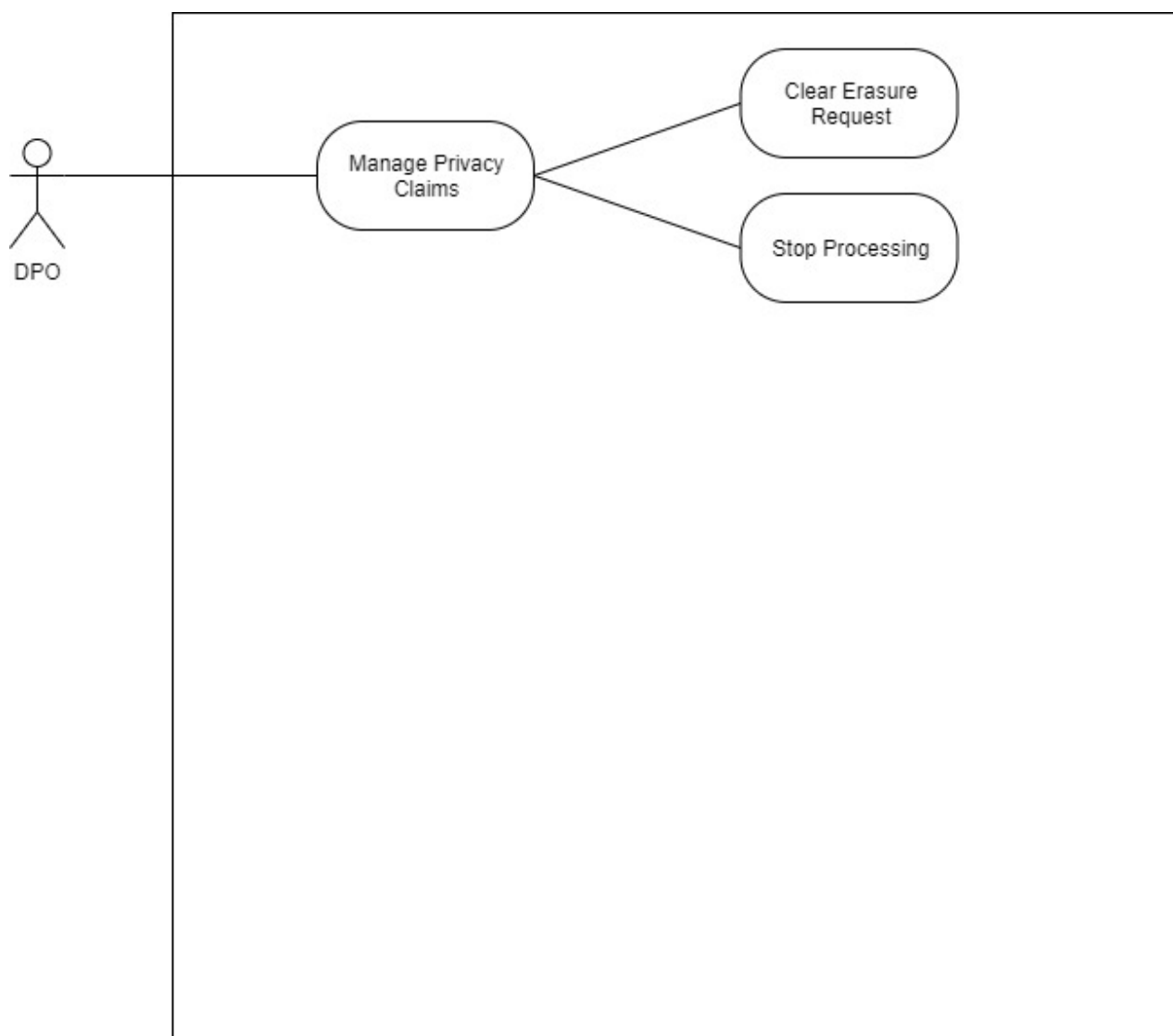


Figure 3: Use Cases for DPO Actor



The *Researcher's* primary functions include: *a)* registering to the system, activating his/her account and completing his/her profile; *b)* managing his/her profile information (*e.g.*, name, email, etc.); *c)* creating research studies *i.e.*, select open or closed study and managing existing research studies *i.e.*, publish, edit and view studies and invite participants; and *d)* managing settings, *i.e.*, view, edit and reset various settings related to general, privacy, security and notifications.



Figure 4: Use Cases for Researcher Actor

The *Participant's* primary functions include: *a)* registering to the system, activating his/her account and completing his/her profile; *b)* managing his/her profile information (e.g., name, email, etc.); *c)* enrolling in study, i.e., receive invitation by link or email, viewing studies, completing questionnaires and tests and viewing results; and *d)* managing settings, i.e., view, edit and reset various settings related to general, privacy, security and notifications.

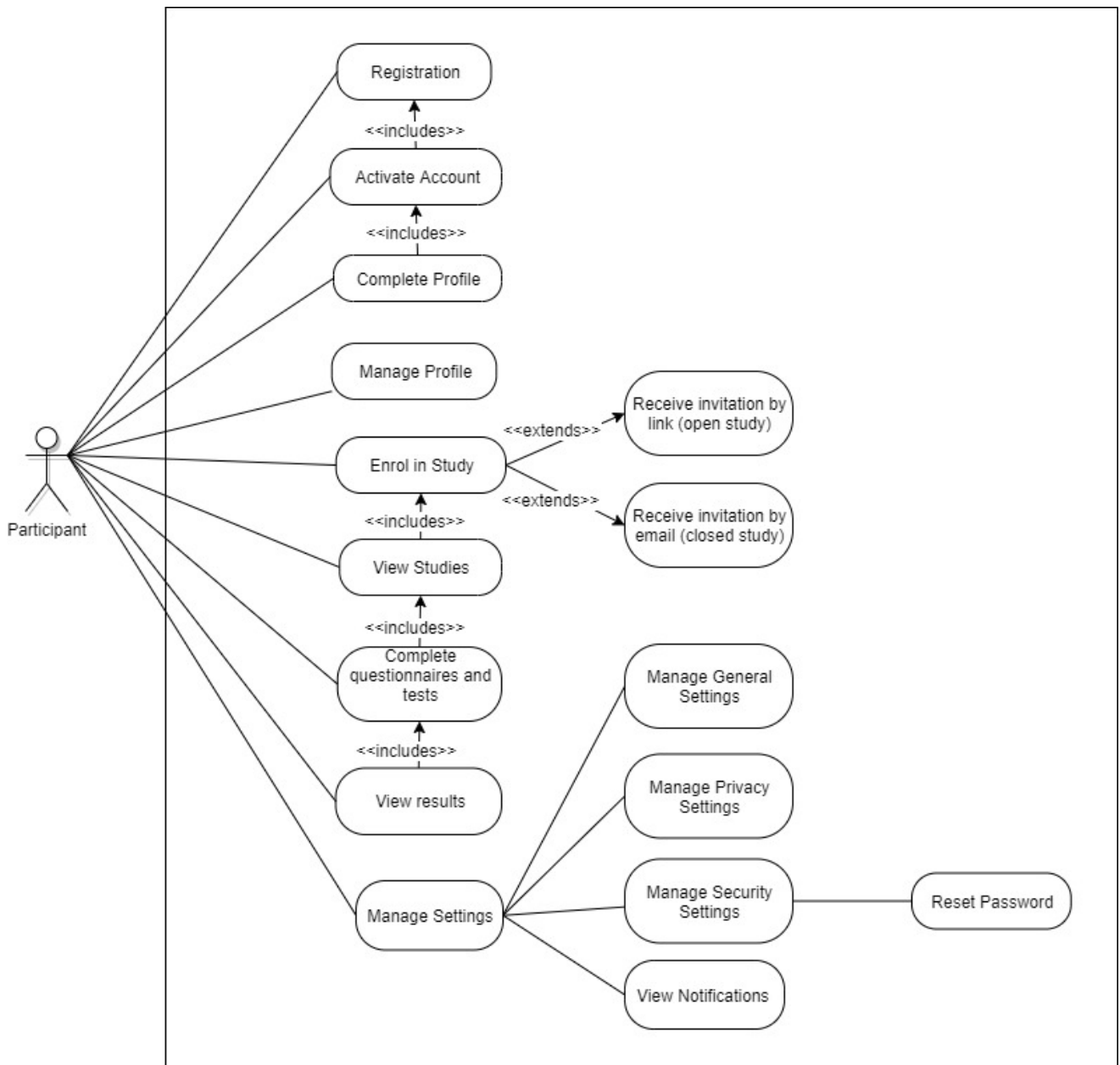


Figure 5: Use Cases for Participant Actor

# System Design

## Introduction

This deliverable provides the overall design of the system. It includes the design of the overall architecture, technical infrastructure, standards, specifications and strategies to be followed throughout the project, testing and implementation of the system. The design of the system also includes an overview of the security and privacy requirements that are thoroughly described in D18.1.

**It is important to note, that throughout the next phases of the project, it is expected that the design will be refined according to accumulated requirements and end user feedback.**

## Overall Architecture

The overall architecture of the system is depicted in Figure 6. The system features a multi-layered design that includes a *Database*, built using SQL Server, where all data are stored using appropriate data structures and relationships. The data are accessed using the *Database API*, which includes system defined data type, functions and stored procedures that conceal the low-level database constructs and provides a uniform way to access data using standardized query mechanisms. Next, the *Business Layer API*, built as a WCF service library, exposes the functionality of the Database API by providing appropriate object-to-relational mappers and other services that allow for manipulation and query of the data. The *Web Service API*, built using RESTful webservice, allows for access of the Business Layer API through the web and includes appropriate security mechanisms to secure the communication of the system with the web application. Finally, the Web Application facilitates user interaction through well designed and responsive web interface.

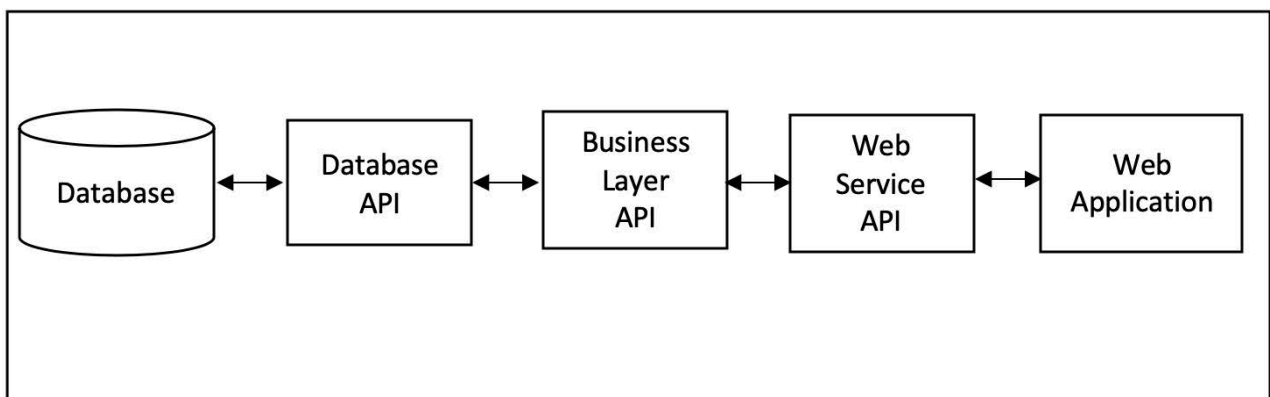


Figure 6: System Architecture

All the layers/components of the system are developed on a single machine. The technologies that were selected for the development of the project are among the key technologies used for web development within the industry.

The following sections provide detailed descriptions on each layer and highlight the key security and data protection measures that are incorporated within each component.

# Database

The Database is built using the relational paradigm since the low volume of data that are expected to be generated within this project cannot be classified as big data. The database is built using Microsoft SQL Server, and includes data structures for storing Admin, Researchers, DPO, Participants and their interactions. The Database Design for the System is presented in Figure 7.

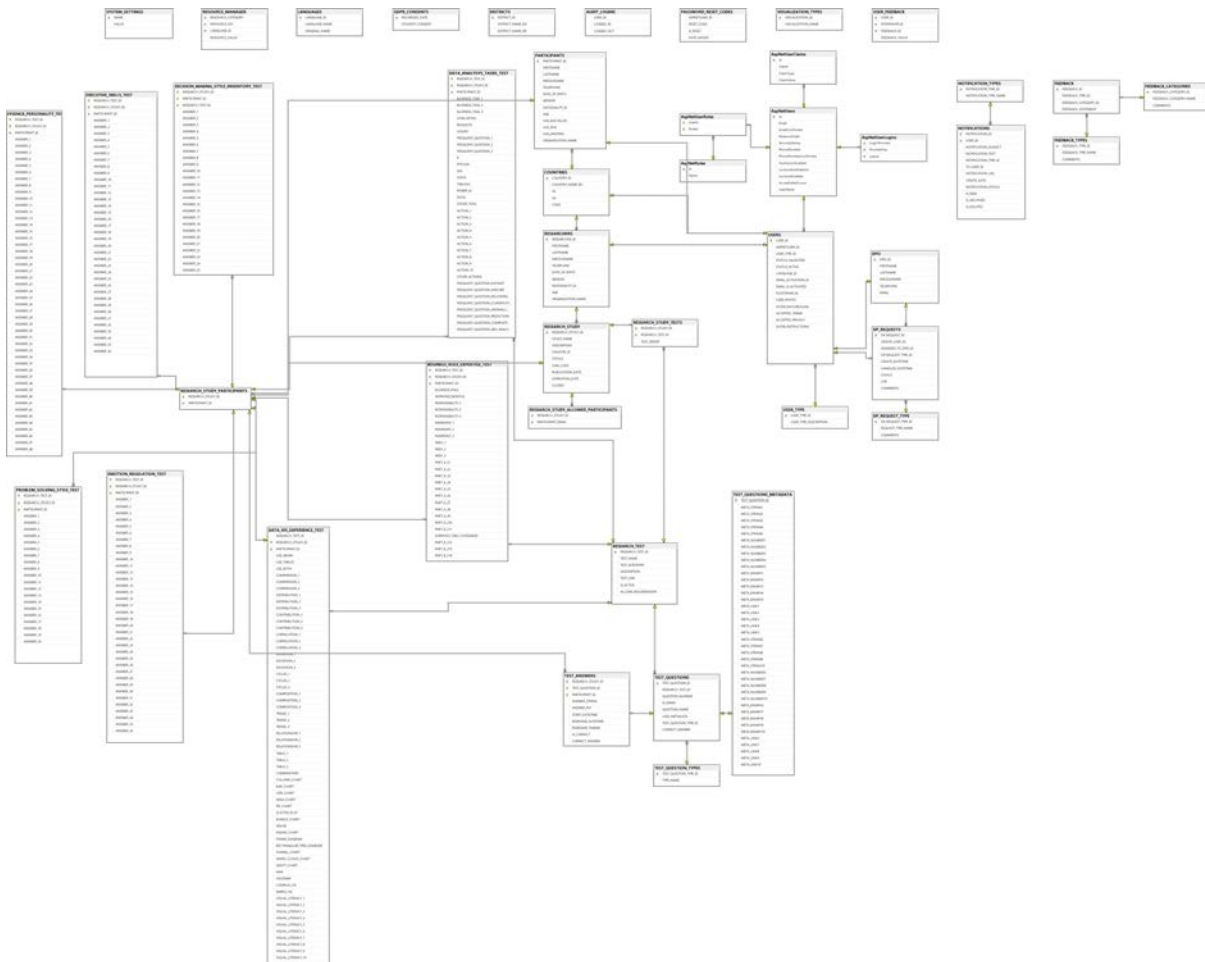


Figure 7: The Database Design of the System

As you can see from the design, the key roles that interact with the system have been modelled accordingly. In particular, the roles that are currently supported are Admin, Researchers, DPO, Participants. The common elements of these entities have been generalized in entity USER that also serves as the bridge between the Authentication/Authorization related entities.

Additionally, the Research Studies and their appropriate Questionnaires and Tests have been modelled appropriately so that are referenced by the entities and indicate where each user is affiliated. Moreover, number of entities have been created to model the Questionnaires and Tests. These include the management of the studies by Researchers and the completion of the studies by the Participants.

Finally, some entities have been created for the management of the system, security and for performance monitoring. Some of these entities, such as Countries, Gender, Industry Classification, use appropriate standards such as ISO3166, ISO5136 and NASE codes accordingly. The Database Dictionary in Appendix 1 provides all the low-level information about the database design.

## Database API

The primary objective of the Database API is to provide a uniform, standardized way of accessing the data structures, hiding the underlying complexity and providing access only where it is required. Additionally, it incorporates security mechanisms that prevent unauthorized access and minimize the risk for common attacks such as SQL injections. In particular, the Database API, operationalizes the access to the database using the following database programming techniques:

- **User-defined data types**, to standardize types used in multiple entities, such as names
- **Views**, to provide a uniform way to access data regardless of data structure changes
- **Functions**, to provide parameterized access to views
- **Stored Procedures**, to provide access to all functionality in the database and to implement complex mechanisms at the database level

The following sections describe each of the aforementioned mechanisms.

### *User-Defined Data Types*

In order to provide standardized access to all commonly used fields in each entity, a number of user defined data types have been created that represent Names, Comments and large text objects. In particular, the following types were created:

- **UDT\_NAME**
  - Usage: Firstname, Middlename, Lastname
  - Type: Unicode variable string
  - Maximum length: 100 characters
  - Not nullable
- **UDT\_COMMENTS**
  - Usage: Comments, Remarks
  - Type: Unicode variable string
  - Maximum length: 500 characters
  - Not nullable
  - Default Value: empty string

- UDT\_TEXT
  - Usage: large descriptions
  - Type: Unicode text
  - Maximum length: unlimited
  - Not nullable
  - Default Value: empty string

## Views

In order to eliminate problems that may occur from updates in database structure, all entities and relationships in the database are only accessible through views. We provide below, a simplified example to facilitate our description. Consider the following table:

```
RESEARCH_STUDY_TESTS ([RESEARCH_STUDY_ID],[ RESEARCH_TEST _ID], [TEST_ORDER])
```

When the study is retrieved, we will require not only to display the research study number (RESEARCH\_TEST\_ID) but also the name of each test, that resides in table RESEARCH\_TEST. To disallow direct query access to both tables, VIEW\_ RESEARCH\_STUDY contains all the fields from the RESEARCH\_STUDY table but also retrieved the TEST\_NAME by joining with the appropriate table. All tables in the systems are exposed similarly.

Furthermore, to facilitate the operations components such as the analytics the system provides specialized views.

All VIEWS are prefixed with <VIEW\_> to indicate that they are views.

## Functions

The primary objective of functions is to provide parameterized access to views. For example, assuming there exists a VIEW\_RESEARCH\_STUDY that provides access to all studies data, a function FN\_GET\_RESEARCH\_STUDY\_BY\_CREATOR\_ID(creator) allows for accessing the view to retrieve only the studies of a specific Creator according to the provided parameter.

Furthermore, functions are used to validate information in the database, to calculate computed fields (e.g., age from the date of birth) and to assist in virtually every component that requires some form of rudimentary processing, such as statistical functions for the analytics component and ranking for the study component.

All Functions are prefixed with <FN\_> to indicate that they are functions.

## Stored Procedures

Stored procedures form the bridge between the Business Layer API and the database. In particular, all access to the database is limited to the implemented stored procedures thus hiding all structures, relationships and constraints in the Database.

To this end, Stored Procedures provide a uniform naming scheme for Create, Read, Update and Delete (CRUD) operations, access to reporting functions and facilitate every low-level operation required by all components.

All Stored Procedures are prefixed with <SP\_> to indicate that they are stored procedures.

The naming scheme for CRUD operations is described below:

- All stored procedures that are used for deleting one or more records in an entity are named SP\_DELETE\_[<<entity>>] and they return the number of affected rows
- All stored procedures that are used for inserting a new record in an entity are named SP\_INSERT\_[<<entity>>] and they return the number of affected rows
- All stored procedures that are used for updating a new record in an entity are named SP\_UPDATE\_[<<entity>>] and they return the number of affected rows
- All stored procedures that are used for retrieving a single record from an entity are named SP\_GET\_[<<entity>>] and they return all the information for the entity from the appropriate view.
- All stored procedures that are used for retrieving all records from an entity are named SP\_LIST\_[<<entity>>] and they return all the information for the entity from the appropriate view.

For example, the RESEARCH\_STUDY table is accompanied by the following stored procedures:

- SP\_INSERT\_RESEARCH\_STUDY for inserting a new research study record
  - Parameter(s): all fields of the entity except PK/ID
- SP\_DELETE\_RESEARCH\_STUDY for deleting an existing research study record
  - Parameter: PK/ID of record
- SP\_UPDATE\_RESEARCH\_STUDY for updating an existing research study record
  - Parameter(s): all fields of the entity and PK/ID to identify record
- SP\_GET\_RESEARCH\_STUDY for getting information about a single research study record
  - Parameter: PK/ID of record
  - Result: SELECT \* FROM VIEW\_ RESEARCH\_STUDY WHERE ...

- SP\_LIST\_RESEARCH\_STUDY for listing records of the research study entity
  - Optional Parameter: keyword or specific parameters for filtering
  - May support pattern matching operations on one or many fields
  - May support ordering by one or more fields

Besides CRUD operations, a number of stored procedures are also available to enable access to statistical information for studies and tests, exporting or importing of data or anything complex required.

## Business Layer API

The Business Layer API is a service library that enables access to the Database API. It contains Classes, which store data elements, Interfaces, which store service function signatures, and Methods, which implement the functionality according to the interfaces. For better maintainability, the Business Layer API is developed using a partial classes paradigm (e.g., one file for CRUD operations, one file for reports, etc.) and all classes combined formulate the complete library.

### *Class Library*

The Class library implemented contains classes for each entity exposed from the Database API, e.g., class Research\_Study will represent an object extracted from VIEW\_RESEARCH\_STUDY. All fields are initialized through a constructor and are accessible through appropriate get/set Properties. Below, we provide a simplified example of the Research\_Study class.

```
[DataContract]
public class Research_Study {
    private Research_Study () {}
    public Research_Study (Int32 ID, String Name, String Description, ...) {
        ...
    }
    [DataMember]
    public Int32 ID { get; }
    [DataMember]
    public String Name { get; }
    [DataMember]
    public String Description { get; }
    ...
}
```

Additionally, there are many classes for facilitating specialized functionality such as reporting.



### Service Methods

The Service Methods map the actions of the database API. For example, method DeleteRESEARCH\_STUDY (Int32 id) calls SP\_DELETE\_RESEARCH\_STUDY for the purpose of deleting a Reserch Study record. In the case of Get/List methods, they return one or more records accordingly and in the case of Insert/Update/Delete they return the status of the operation and the number of records that were affected.

The service methods are also organized appropriately for easier maintainability. Below, we provide an example for the Research Study concept.

```
#region Research_Study
```

```
public OpCode Insert_Research_Study ( ... ) { ... }
```

```
public OpCode Update_Research_Study ( ... ) { ... }
```

```
public OpCode Delete_Research_Study ( ... ) { ... }
```

```
public Research_Study Get_Research_Study ( ... ) { ... }
```

```
public List< Research_Study > List_Research_Study ( ... ) { ... }
```

Furthermore, to ensure that future changes in the database do not affect the maintainability of the code, mapping methods are created that act as intermediary reading methods. Below, we provide the read method for Research Study records.

```
public static Research_Study ReadResearch_Study ( SqlDataReader resultSet ) {  
    return new Research_Study(  
        ResearchStudy_Id: Convert.ToInt32(resultSet[0]),  
        ResearchStudyName: resultSet[1].ToString(),  
        Description: resultSet.IsDBNull(2) ? Null : resultSet[2].ToString(),  
        ...  
    );  
}
```

# Web Service API

Web Services expose the Business Layer API for the web application by interacting with XML or JSON data. These web services are built as restful services that follow the same standardized schemes of the other layers. Every method in the Business Layer API is exposed in a manner similar to the below.

```
[WebMethod]
[ScriptMethod] (
    ResponseFormat = ResponseFormat.Json,
    XmlSerializeString = false
)
public Research_Study Get_Research_Study (Int32 ResearchStudyID) {
    Research_Study s = services. Get_Research_Study(ResearchStudyID);
    Context.Response.Clear();
    Context.Response.ContentType = "application.json";
    Context.Response.Write(new JavaScriptSerializer().Serialize(s));
}
```

# Web Application

The UI/UX design of the web application along with its main functionality are explained in D10. In this section we discuss the web application's components structure, how it communicates with the Web Services API for data exchange, authentication and localization. Moreover, we discuss how a Research Study is created by a researcher and how a participant can join this research study. Also, we go through the overall architecture of how cognitive tests and questionnaires are dynamically created.

## *Overall Web Application Structure*

The structure of the web application is built around three main components **MasterPages**, **UserControls** and **Pages**.

### **MasterPages**

Generally, web applications consist of multiple pages with each page containing its own content and having its own uses. Another common aspect though in web applications is that some parts of the application are consistent through all the pages, these parts can be the header, navigation bar, footer etc. The previous statements are true for the IDEALVis web application as well. One way to approach such design is to replicate the code for the parts that are the same across pages, but this approach poses a problem of reusability and maintainability. For keeping a consistent structure, we made use of the MasterPages that are essentially site-wide page templates. In other words, a master page is a special type of ASP.NET page that defines the markup that is common among all content pages as well as regions that are customizable by other content pages that use the MasterPage as their template.

The main MasterPage of the web application is illustrated as a picture below in Figure 5. As illustrated the main elements that the MasterPage keeps consistent across all pages are the top menu bar with the logo, settings and user/notification area. Moreover, the MasterPage has two main areas that are dynamically loaded. The first area is called Navigation Area where navigation links appear depending on the role of the logged in user. Next, the Content Area is where the content of the current viewed page is shown.



Figure 8: Main MasterPage

Using this technique allows us to make changes on the MasterPage, changes that are directly reflected on every page, without having to maintain the same code in a number of pages and thus allowing for potential inconsistencies.

**Notes:** The master page is also responsible for loading stylesheets and JavaScript files that are globally required. A different (more simple) **MasterPage** than the one in Figure 8 is used by pages that contain tests and questionnaires for removing unnecessary stimuli.

### UserControls

This type of component enhances the web application structure as it allows for reusability. UserControls allow for grouping markup and code together into a reusable container, so that the same interface with the same functionality can be used around the web application without having to recode it again. Other than reusability UserControls allow us to break the web app code down into logical components. The web application makes extensive use of UserControls especially for keeping some UI parts isolated from others. To demonstrate the usage of UserControls refer to Figure 8 where the MasterPage dynamically loads the navigation bar. Each user type in the web application has its own navigation bar and thus we implemented each navigation bar into a separate UserControl that the MasterPage then selects and loads into the specified area. An example of this

implementation is shown below where the MasterPage uses a switch statement to load the appropriate controls depending on the user type.

```
switch (UserProfile.UserType)
{
    case (int)UserType.DPO:
        ctrlTopMenu=Page.LoadControl("UserControls/ucDPOTopMenu.ascx");
        break;
    case (int)UserType.Admin:
        ctrlTopMenu=Page.LoadControl("UserControls/ucAdminTopMenu.ascx");
        break;
}
```

**Note:** A similar approach is used for loading the appropriate dashboard for each user type.

## Pages

This web application consists of multiple pages and for keeping navigation and further development flexibility a naming convention was applied on pages. Moreover, here we discuss the several categories of pages and their naming conventions.

- **User Pages** allow each user type to register their profile (before first login), view their profile, edit their profile:
  - Edit{UserType}.aspx
  - Register{UserType}.aspx
  - View{UserType}.aspx
- **Entity CRUD pages** allow for CRUD operations related to a database entity (i.e. NotificationTypes):
  - {Entity}.aspx – form for updating or creating a record of an entity
  - {Entity}List.aspx – table for viewing and deleting records of the entity
- **Error Pages** provide feedback to the user when a specific error is thrown:
  - 401.aspx
  - 403.aspx
  - 404.aspx
  - 500.aspx
  - 504.aspx
  - 505.aspx

- **Test / Questionnaire Pages.** Each test or questionnaire has a folder, named after its name and the folder contains three pages.
  - {TestName}/Intro.aspx
  - {TestName}/Test.aspx
  - {TestName}/Finish.aspx

### *Web API Communication*

As we have seen in the abovementioned system layers a chain is formed where a request flows in a bidirectional manner from the Web Service API to the Business Layer API and finally to the Database API Layer. The Web Service API exposes the Business Layer API service methods that map the actions of the database API. For the web application we kept a similar communication architecture where each entity i.e. NotificationTypes has its own JavaScript file that contains all the CRUD operations in the form of AJAX (Asynchronous JavaScript and XML) calls that map to the appropriate Web Service API controller functions. All communication between the web application and the Web Service API is done using JSON (JavaScript Object Notation). For having a consistent and maintainable architecture that can appropriately expand for future requirements a library of JavaScript files was created. Each of those files represents an entity and its underlying CRUD operations. Other than basic CRUD operations though, those JavaScript files contain code that is used for dynamically building the UI i.e. Data Tables for an entity in a {Entity}List.aspx page. Moreover, the JavaScript library extends its functionality and goes beyond entity operations as it contains code that is related to specific system operations such as localization and authentication that require the Web Services API to communicate with the underlying service functions of the Business Layer API.

For a consistent and clean architecture, we further developed a class called ResponseMessage that encapsulates the data from the Web Service API responses into a common format. All the Business Layer API service methods return a ResponseMessage object to the Web Service API that in turn serializes the object to JSON so it can be then delivered through HTTP to the client JavaScript library for further processing. The ResponseMessage class encapsulates 4 items, the Result which represents the requested data, a Success flag that denotes whether the request was processed successfully, an Exception string that contains any exception thrown and lastly a Message string that is used for informing the user regarding his/her request. The ResponseMessage class is shown below.

```

public class ResponseMessage
{
    public Object Result;
    public bool Success;

    public String Exception;
    public String Message;

    public ResponseMessage()
    {
        Result = null;
        Exception = null;
        Message = null;
        Success = true;
    }
}

```

### *Authentication*

All the functionality of the system is accessible only after a user has signed up and successfully passed the authentication by logging in, using a valid email address and a corresponding password. The system used for managing user registration and authentication is ASP.NET Identity, a membership system offered by Microsoft that allows for customized login/logout functionality, user role management, customized user profiles, multiple storage providers (in our case Microsoft SQL Server) and automatic generation of the required table schema in the database for keeping user data. ASP.NET Identity offers great security as it follows industry standards for authentication and as it uses hashing for the user's passwords, a mandatory requirement for any system that keeps user information.

The web application has a login page and a registration page as shown in D10. Those pages have the required fields for the user to enter his/her details. Once the user provides the data for login or registration the JavaScript library (UserAuthentication.js) as mentioned above submits the data through AJAX calls to the corresponding controller (AuthenticationController.cs). The controller then calls the specific function to either register or authenticate the user as per the request. Both registration and authentication functions contain the required business logic that acts on the ASP.NET Identity library. The registration function uses the UserStore class from the Identity library for creating the new user if one with the same email does not exist, code below:

```

var userStore = new UserStore<IdentityUser>();
var manager = new UserManager<IdentityUser>(userStore);
var user = new IdentityUser(){UserName = jsonObject.Email, Email= jsonObject.Email};
IdentityResult IDresult = manager.Create(user, jsonObject.Password);

```

Similarly, the authentication function of the controller uses the UserStore class in a similar fashion for making sure that the user about to login exists in the system.

```

var userStore = new UserStore<IdentityUser>();
var userManager = new UserManager<IdentityUser>(userStore);
var user = userManager.Find(jsonObject.Email, jsonObject.Password);

```

One important note here is that authentication is achieved through the use of cookies. Once correct credentials are given and the user is found through the UserStore, the authentication controller function issues a cookie that contains a FormsAuthenticationTicket. The cookie is then added to the response that is received by the user's browser and therefore the system now allows the authenticated user to access his/her dashboard that is the starting point of the web application.

Any subsequent calls to the Web Service API only pass through if the user's browser contains a valid non-expired cookie.

### *Localization*

Localizing a web application is not always a straightforward task and for that reason we developed a localization system that meets the platform's specific requirements. Most of the localization systems are built for Multi-Page Applications where all requests to the server issue a full-page refresh. In this approach the localized strings like all other page components are constructed on the server as per the request, and once the final HTML content is composed the server then sends it as a response to the client browser so the page can be rendered. This platform uses a Multi-Page Application design but many of its components like Questionnaires and Tests are constructed using JavaScript and without the page fully refreshing. For these reasons a localization system that acts in an asynchronous manner was devised.

Each user has a LanguageID associated with him/her stored inside the Users table in the database. This ID specifies which language was selected by the user as his/her default language. All users have English as the default language once they register, but they can navigate to General Settings and change the default language anytime. This LanguageID is then used by our localization component that returns strings in the required language.

The main component of our localization engine is the ResourceManager. This component is broken down in multiple parts and each part exists in a different system layer. First of all, the most prominent part of the ResourceManager exists in the database as a table that stores all the resource values. Each record in the ResourceManager table consists of an ID, a ResourceCategory, a LanguageID, a ResourceKey and a ResourceValue. The ResourceCategory is used to describe a category of resources i.e. all the string resources of all the form buttons. LanguageID denotes the current language of the string resource. Finally, ResourceKey is what we use later on to fetch a specific value from a category of resources and ResourceValue is the actual string resource. A representation of some values in the ResourceManager table is shown in Figure 9.

	RESOURCE_CATEGORY	RESOURCE_K...	LANGUAGE_ID	RESOURCE_VALUE
875	Notifications	No_Notifications	1	Nothing to see here!
876	Notifications	No_Notifications	2	Δεν έχετε καμία ειδοποίηση!

*Figure 9: Resource Manager Table Values*

Moving on, the ResourceManager has appropriate stored procedures in the Database API that are accessed by the Business Layer API that maps those stored procedures on service methods that the ResourceManagerController can access and publish through the Web Services API. Requests to the ResourceManager endpoint require only the ResourceCategory as a parameter as the LanguageID of the user requesting is automatically passed to the stored procedures through the Business Layer API service methods.

The last component of the ResourceManager that actually applies the localization is a JavaScript Library created for just this purpose. The library does 2 simple operations. First, it calls the ResourceManager endpoint in an asynchronous manner (after the page is fully loaded) with the resource category as a parameter. Second, once the server responds with the resource category and all its values the ResourceManger client library constructs a ResourceValueHelper object that It returns to the page that requested the localized category of string resources. Moreover, the page that requested the resource values can now use that helper object and get any resource value using the ResourceKey of that specific resource. The code below illustrates an example where the ResourceManager is used for fetching the localized value for a specific message that is under the Toaster ResourceCategory.

```
GetResourceValueHelper('Toaster').then(function (resourceValueHelper) {
    var Message = resourceValueHelper.getValue("Toastr_Questionnaire_Draft_Submitted");
})
```

### *Research Studies*

One vital part of the system is the creation of Research Studies. Users with the Researcher role are able to create and publish research studies. Essentially a research study contains the following aspects **Name, Description, Publication Date, Expiration Date, Closed Status, Join Code and Availability Status**. The RESEARCH\_STUDY database table of research studies can be seen in Figure 10. The name and description are simple text fields that allow the researcher to name and describe a research study's purpose. Moreover, the publication and expiration dates denote the availability of a research study so participants can only access it in the predefined period. The closed status is a Boolean flag that denotes whether a study is accessible by any participant or by a selected group of participants. If the Researcher decides that a research study is a closed study a prompt appears for the Researcher to write the emails of the allowed Participants, then the system only allows those participants to participate in the research study (system sends an invitation link through an email to all invited participants). Those selected participants are saved on a different table called RESEARCH\_STUDY\_ALLOWED\_PARTICIPANTS with a one to many relationship Figure 12. The join code is a randomly generated code that is a unique for every research study and it is used in the invitation link for distinguishing a specific research study. If a research study is selected to be an open study, then the Researcher can get a the autogenerated invitation link that can then be send to potential participants. The availability status is a flag that denotes whether a research study is Published, Drafted, Withdrawn or Finished. The last and most important aspect of a research study are the tests / questionnaires that a researcher makes available through it for the participants to complete. The system contains a variety of tests and questionnaires that makes available to



research studies, the tests are stored under the RESEARCH\_TEST database table Figure 11. Since the system can hold multiple research studies and multiple research tests within each study a many to many relationship between the two tables had to be created Figure 12. Finally, after a research study is published Participant users are able to join it, thus the RESEARCH\_STUDY table also has a many to many relationship with the PARTICIPANTS table Figure 12.

	RESEARCH_STUDY_ID	STUDY_NAME	DESCRIPTION	CREATOR_ID	STATUS	JOIN_CODE	PUBLICATION_DATE	EXPIRATION_DATE	CLOSED
1	1	IDEALVis Study	IDEALVis Study	1	3	DDA7FA	2019-12-14 09:00:00.000	2020-01-02 17:00:00.000	NULL

Figure 10: RESEARCH\_STUDY Table

	RESEARCH_TEST_ID	TEST_NAME	TEST_ACRONYM	DESCRIPTION	TEST_LINK	IS_ACTIVE	ALLOWS_RESUBMISSION
1	1	Business Role Questionnaire	BusRole	Business Role Questionnaire	Tests/BusinessRoleExpertise/intro.aspx	1	1
2	2	Data Analysis Tasks Questionnaire	DataTasks	Data Analysis Tasks Questionnaire	Tests/DataAnalysisTask/intro.aspx	1	1

Figure 11: RESEARCH\_TEST Table

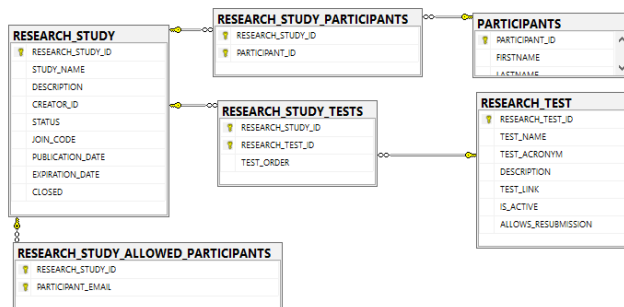


Figure 12: RESEARCH\_STUDY Relationships

### Joining Research Studies

Any user with the Participant role can join a particular research study if he/she has the appropriate invitation link and the privilege to do so (case of closed research study). To allow more flexibility for the user we created a page that handles the research study joining process and we made this page available to both logged in and logged out users. This way the system allows for people that are not yet registered on the web application to join a research study by registering first. The page is shown in Figure 13.

Join Research Study

Research Study Name: IDEALVis Study  
IDEALVis Study

Duration of study 01/01/2019 09:00 - 02/02/2020 17:00

Login or Register to join the study!

Figure 13: Join Research Study Page (Logged Out)

Moreover, when the user selects Login or Register the system redirects the user on a different page and for this reason the system makes sure to keep the research study code in the user's session as seen in the code below. The research study join code is kept in the session temporarily for redirecting the user back to the join page once he/she registers or logs in the web application Figure 14.

```
Session["JOIN_CODE"] = joinCode;
```

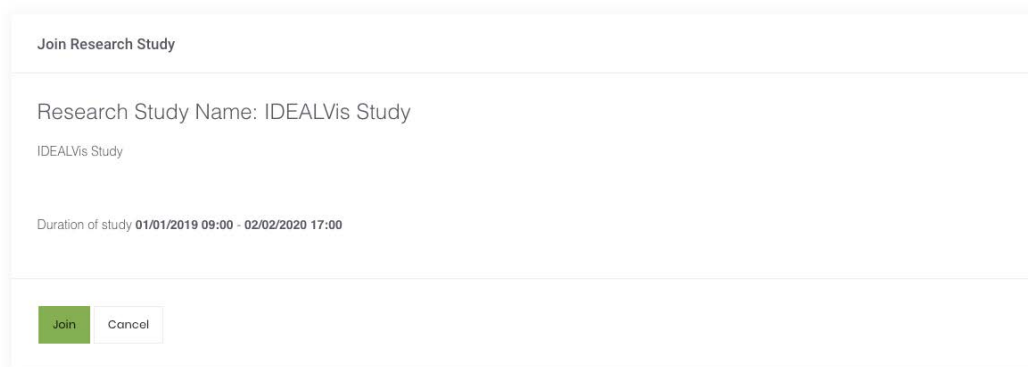


Figure 14: Join Research Study Page (Logged In)

Finally, once a Participant clicks the Join button, he/she successfully joins a research study, the code below executes and the RESEARCH\_STUDY\_PARTICIPANTS table receives a new record that links the specific user to the joined research study.

```
RESEARCH_STUDY.AddRESEARCH_STUDY_PARTICIPANT(UserProfile.UserID, this.researchStudy.RESEARCH_STUDY_ID);
```

**Note:** The join research study page receives the join code from the URL parameters. There is also a system function that takes in the join code and returns the specific research study object. Code below.

```
this.researchStudy = RESEARCH_STUDY.GetRESEARCH_STUDY_BY_JOIN_CODE(joinCode);
```

### Dynamic Creation of Tests / Questionnaires

As previously explained the system keeps information about tests and questionnaires in the RESEARCH\_TEST database table. Some important values held in this table are the name of the research test / questionnaire and the link to the actual test / questionnaire. In the participant's dashboard a dropdown option allows the participant to select a research study to complete. By selecting a research study, the participant is then shown the available test / questionnaires of this research study Figure 15. The name and the link of the test is used in that list so the participant can access it. This information is asynchronously loaded using the appropriate AJAX calls to the corresponding Web Services API endpoint.

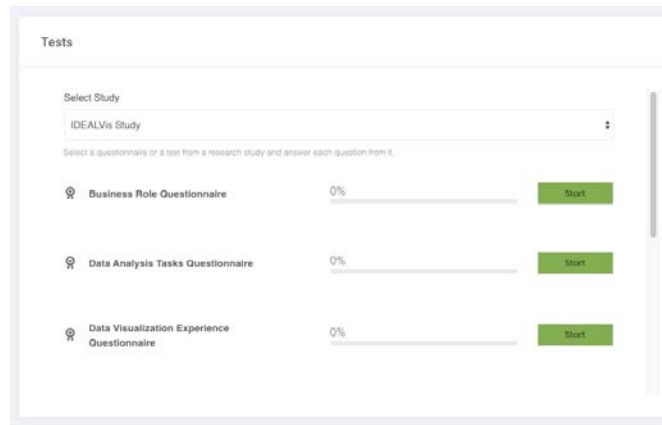


Figure 15: List of Available Research Study Tests

From now on I will be referring to both cognitive tests and questionnaires as research tests for simplicity. All research tests are composed of three steps that the participant has to go through. The steps are illustrated in Figure 16.

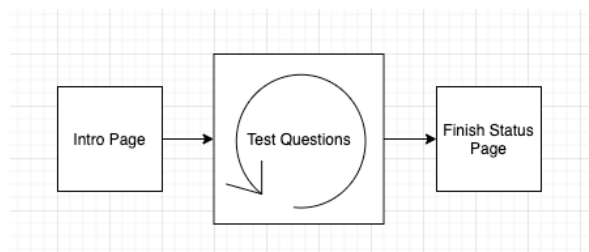


Figure 16: Research Test Steps

The illustrated steps correspond to the three pages that make up a research test. As previously explained in the **Web Application Structure** section, each research test consists of three pages, Intro.aspx, Test.aspx and Finish.aspx. Once a participant selects a research test the system redirects him/her to the intro page where information on how to complete the test correctly are showed. The last step of this process is the finish page where the user can see his/her results like average completion time in case the research test was a cognitive test, if the research test completed was a questionnaire no results are showed, instead only a thank you message is displayed (More details in D10). Until now we talked about the start and the finish stages of the research test. Moving on, we will discuss about the actual test component (central step) that is responsible for two scenarios. First scenario takes the user through multiple iterations (tasks / questions) until the research test is fully completed and the second scenario automatically generates a questionnaire by iterating on a set of database values related to that research test. The architecture here is similar across scenarios but for clarity we break down the logic in two architectures one for each scenario. As explained all research tests have the three abovementioned files. The most important file is the Test.aspx file that in conjunction with a corresponding Test.js creates the actual research test (the two files together implement the architecture). Before we go further and explain each architecture it is worth noting that at the database level both architectures use the same tables for retrieving and persisting data Figure 17.

Below we will explore the two scenarios and their architectures as illustrated in Figure 18, lastly, we will go through the database structure that accommodates these architectures.

### **Generating Cognitive Tests (1<sup>st</sup> Architecture)**

In first scenario the Test.aspx file contains the several areas (placeholders) where the question / task related data will go. Moreover, the Test.js file is responsible for calling the appropriate Web Service API endpoint for performing each task as illustrated in the left bottom box of Figure 18 (Get Question / Post Answer). One fundamental part of this architecture is the procedure that brings the next available question. This procedure using the current research study ID, the current research test ID, the participant ID and the already completed answers related to the current research test can infer the next question to be answered. The user continually submits answers while the next answer is coming until all of them are answered. Once the component has no other questions to display the user is redirected to the Finish.aspx. Response time in milliseconds is also recorded for each answer.

### **Generating Questionnaires (2<sup>st</sup> Architecture)**

In the second scenario we see a similar component that creates multiple questions into one page. The Test.aspx page contains a table placeholder for the questions to be generated in, and appropriate buttons for submitting all the answers together. Moreover, the Test.js file is responsible for calling the appropriate Web Service API endpoint for performing each task as illustrated in the right bottom box of Figure 18 (Get All Questions / Build Layout Depending on Questionnaire Type (Close Ended, 5 Likert, 7 Likert etc), submit answers, retrieve already completed answer values. The most important component here is the JS Library that contains a set of builder functions that can generate the layout of the questionnaire depending on the type of the questionnaire. For each questionnaire type we generated a set of rules that creates the appropriate mark-up that accommodates the questionnaire by inserting the appropriate HTML elements, like for example the number of radio buttons for a Likert Scale question of 5 Scales. Given the questionnaire type the library returns a builder function that is then used for inserting every question in the Test.aspx page.

### **Research Test Database Tables**

The abovementioned architecture makes extensive use of the database. This usage as explained above is established through the abovementioned Web Service API calls. Here we go through some of the most important tables to illustrate the inner workings of the architecture, relationships of tables are also explained and further illustrated in Figure 17.

**RESEARCH\_TEST** table as explained in the beginning of this section holds the name of the test and the link to that research test along with other required data.

**TEST\_QUESTIONS** table contains all the questions related to each research test. Here is where we also assign the type of each question by adding a FK to the test question types table, the text of each question and the correct answer if there is one.

**TEST\_QUESTION\_TYPE** table holds the various types of questions so that the UI can generate the appropriate layout.

**TEST\_QUESTIONS\_METADATA** table extends the test questions table by providing extra meta data that can be used for more complicated cognitive tests or questions that require more than text i.e a coloured string.

**TEST\_ANSWERS** table holds all the answers of all questions from all tests related to all research studies. Since a user can be in multiple research studies that share the same research tests here, we are using a composite key to distinguish and separate each answer that comes from the same research test found in two different research studies. The composite key is made up of the **RESEARCH\_STUDY\_ID**, **TEST\_QUESTION\_ID** and **PARTICIPANT\_ID**. A particular question is linked with a single test therefore the research test can be inferred from the **QUESTION\_ID**.

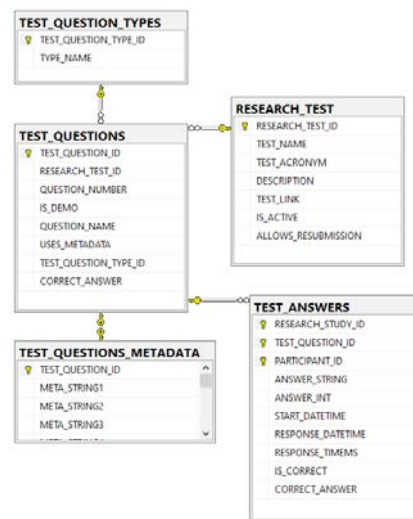


Figure 17: Questionnaire / Test Resource Tables

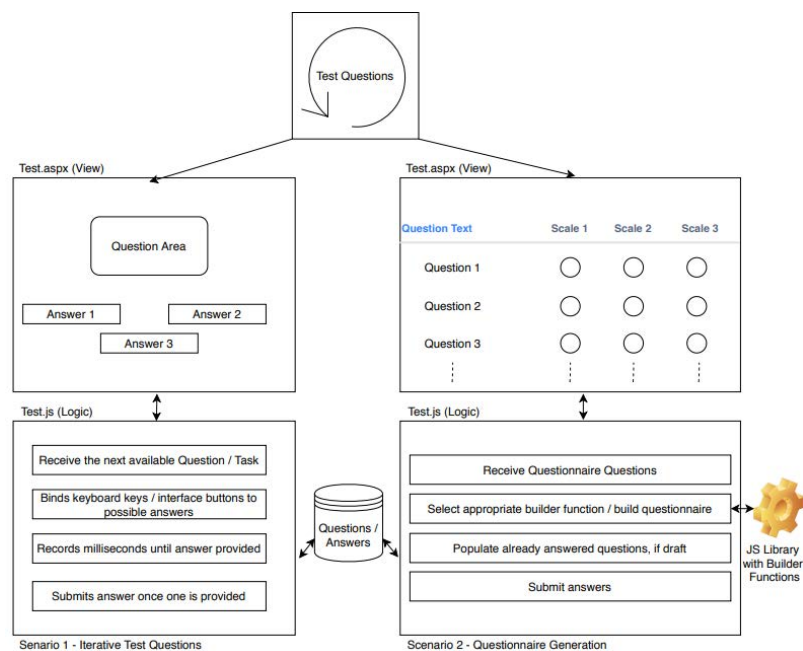


Figure 18: Research Test Generation Architecture

# Security and Data Protection

Regarding security and data protection measures, there are three tiers that protection measures need to be addressed as presented in Figure 19: at the network, host and application level. At each tier, appropriate security controls are specified to protect specific elements and comply with the General Data Protection Regulation (GDPR). At the network tier, the objective is to secure the traffic exchanged between the clients and the server. At the host tier, the objective is to secure the hardware and operating system elements and at the application tier the objective is to protect the operation of the web server, the database server and the relevant applications. Overall security objective across all tiers is to achieve data privacy. Security controls specified fall under the broad categories of access control, cryptography, security policies, hardening, auditing and logging, and backups. For further information on the security aspects, reference deliverable D18.1. Initial cybersecurity and data protection framework.

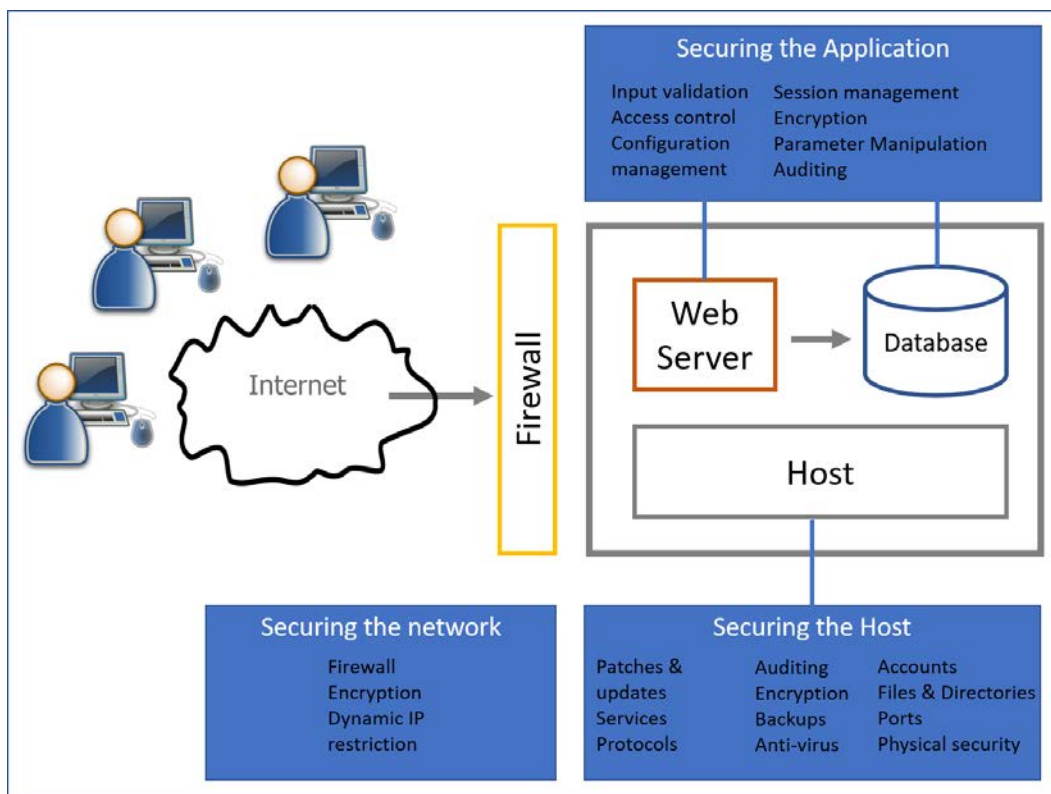


Figure 19: Security Architecture

## Appendix 1: Database Dictionary

[illegible]

