Managing heterogeneous data in the HEALS project

L.J.M. Cluitmans¹

¹VTT Technical Research Institute of Finland, Tampere, Finland

Abstract— A database system to support researchers in the EU HEALS project was implemented, storing heterogeneous data such as medical data, questionnaire results, omics data, satellite images, and more. The system provides hybrid storage supporting SQL, NoSQL and file style data. A flexible access control subsystem regulates how each user can access each data resource.

Keywords— healthcare information systems, health informatics, bioinformatics, database.

I. INTRODUCTION

This paper describes the design of the *Geodatabase* implemented as part of the EU HEALS project (*Health and Environment-wide Associations based on Large population Surveys*) [1]. This database system acts as a central database in the project, storing a wide variety of heterogeneous data types, (such as study subject biomedical data, satellite data, questionnaire results, omics data, intermediate analysis tables, etc.) with a wide variety of access restrictions.

The Geodatabase is not the only database system in the HEALS project. The project also includes other, more specialized, databases, such as an “*Environmental Data Management System*” (EDMS); these databases are interlinked.

Heterogeneous database systems like the one described in this paper are appearing more frequently these days in a multitude of research domains. They serve projects and environments that need to store and analyze data of different types.

This paper focuses on the challenges and design of this Geodatabase from a software architectural point of view, and does not go into matters relating to the actual data inside.

II. REQUIREMENTS

The requirements for this database system have several aspects, each with their own challenges and impact: *technology*, *content*, *access control*, *exposure*.

For the *technology* aspect, choices have been made right at the start of the project, and these choices restrict all other design decisions. For this database system, the key requirement is that it must be usable over the internet by the HEALS project partners: researchers spread all over Europe.

For the *content* aspect, the main challenge is that very heterogeneous data needs to be stored. To name a few examples:

satellite images, geo-indexed pollutant data, subject movement and exercise information, subject health related measurements, indoor pollutant and climate data, questionnaire response data, pre-existing cohort study data, omics data, intermediate analysis results, and many more. How to best store each kind of data depends on the nature of that data. Some data is best stored generically, as “files” in a system that appears similar to a conventional file system to the users. Other data better fits storing as tables in a relational database, enabling flexible querying. Yet other data better fits a NoSQL database approach, supporting data that is a collection of hierarchically structured objects rather than as rows in a table, or whose structure is only partially predefined. Such data objects are typically represented as JSON or XML.

For the *access control* aspect, different data have different requirements. Some data is public. Other data must only be accessible by approved users. There are various reasons for having access restrictions. Some of the data is privacy sensitive. Other data is obtained under legal conditions or licenses that restrict who is allowed to access the data. These restrictions are not necessarily hierarchical: user *A* may have access to data *X* but not data *Y*, while the reverse is true for user *B*. A fine-grained access system is needed to express such restrictions.

For the *exposure* aspect, there are several relevant questions to answer. “*In what ways should users be able to get data out of the system?*” “*What kinds of visualization should be provided?*” “*How powerful do querying capabilities need to be and what can be left to the user’s own off-line tools?*” “*What server side analysis and preprocessing options are needed?*” “*What file formats should be supported for data export?*” Not all of these can easily be answered *a priori*, but only become clear in time.

III. DESIGN AND IMPLEMENTATION

A. Technology

From the outside, the database system appears as a web site (located at <https://heals.vtt.fi/heals/>), supporting both normal interactive web access to users, as well as providing web services for server-to-server interaction.

The underlying databases are not directly accessible via the internet for security reasons. Gating all access through a web site provides multiple advantages, including:

- Access control, site misuse prevention, and other security-related features are easier to implement and manage, since there is only one access channel.
- Underlying storage system(s) are decoupled from what the users perceive, adding flexibility in implementation and allowing features like on-the-fly data transforms.
- Computer-to-computer communication (as opposed to user-to-computer communication) can be implemented as web services.

The web site has been implemented as an ASP.NET MVC application running on IIS as a web server. Actual data storage functionality is provided by multiple database server applications and the server's file system, as explained later. A system component overview is depicted in figure 1.

Fig. 1: Overview of the “HEALS Geodatabase” system

B. Storage subsystems

The data to be stored is heterogeneous, and a “one size fits all” solution to storing all the different types of data would not lead to a useful system. Our implementation uses four separate storage sub-systems, providing different storage options to data of different types.

1. Document storage (backed by the server’s file system), for storing data files and documents as BLOBs.
2. Tabular dataset storage (backed by a MySQL database) for storing tabular/relational data.
3. NoSQL style dataset storage (backed by a MongoDB database), for storing JSON style data objects, or row-like data whose structure is only partially predefined.
4. System data (backed mostly by an MS SQL Server database and partly directly by the file system).

The first three storage systems are for storing what the site users perceive as the “actual data”, while the “system data”

subsystem stores information used in the web site’s implementation and is not directly accessible to end users as such. This category for instance includes user and “authentication and authorization” information, as well as the metadata that makes the other three subsystems “work”.

The fact that there are two relational databases in the system (items 2 and 4 above) may look strange at first. One reason is isolation: by separating these, user data and system data are guaranteed not to accidentally interfere. Another reason is related to the technologies used in implementing these systems, and the way the web application implementing the site “sees” these systems. The system data is tightly coupled to the site’s code via an ORM (Entity Framework 6). On the other side, the user-visible data is not coupled to the site’s code, but is processed at a level that is closer to raw SQL. A third reason is that this split solved a problem that appeared during development: originally the system database was intended to also store user-visible data, but it became clear that storing the unpredictably sized user provided data there might cause hitting license restrictions.

C. Document storage

The document storage subsystem allows the users to upload and download files (called “documents” in the system). While the system appears as a familiar file-system-like folder structure, the implementation and backing storage is different from this façade shown to site users. The implementation allows for some bonus features that may or may not be directly visible to users, including document versioning, file reuse, and features related to access control (described later).

The system internally distinguishes the concepts of “file”, “document” and “document version”. The distinction is not related in any way to the content of these, but related to the semantics. A “document” is located inside a “folder”, and tracks a list of “versions” of the document, each of which is a pointer to a “file”. A “file” is immutable and only accessed by a unique identifier.

Users interact with “documents” directly, while “files” are only created as a side effect of these document manipulations. When a user uploads a brand new document to the system, they create a new “file” and a new “document”. When uploading a newer version of the same document, a new “file” is created, and a new *version* is added to the existing document (no new document is created).

When a user uploads a file that was previously uploaded to a different folder (or using a different name), a new document is created, but the previously uploaded file is reused and the new identical uploaded file is silently discarded. Note that for this to work the “file” objects indeed must be immutable, since they may be silently linked to different (versions of) documents. This mechanism allows the “same” document to

appear in multiple places in the folder tree without actually taking any extra space, creating a functionality similar to “hard links” in classic file systems; Due to files being immutable, this functionality can be provided automatically and silently without any need for user intervention

The files uploaded to the “document storage” subsystem are stored as files in the server’s filesystem. The files are all stored in the same server folder, using a name based on the file’s unique identifier (based on hashing). To preserve properties of the originally uploaded file, namely that file’s original name and its content-type, a “metadata” record for the file is stored separately.

As mentioned, the “documents” are logically stored inside “folders”, and those folders are nested hierarchically to form a folder tree. Each folder can contain *child folders*, *documents*, and *links*. Logically there is one single “root” folder. The root folder and its immediate children play a special role in the access control system (explained later). As a side effect, the root folder can only contain child folders, not document or links.

The above-mentioned “links” are similar to what in a normal filesystem would be “soft links” or “shortcuts”. Links may point to anything that can be identified by an “object identifier” in the site, including folders and documents, but also to items such as datasets that do not appear in folders themselves.

D. Dataset storage

The *dataset* storage subsystems provides storage of tabular and hierarchical “object” datasets, supports query capabilities for those, and maintains access control metadata for those. Some datasets are stored in the SQL database, others in the NoSQL database.

Tabular datasets are stored as tables in a MySQL database. These datasets can be queried in a traditional SQL query style, as explained later in the “*exposing the data*” section. Site administrators can create views to expose predefined queries on those tables. Using views in this way allows predefining some useful joins of related tables, predefining aggregate queries and defining subsets of the original tables, in particular subsets that exclude “sensitive” columns.

Datasets of hierarchical data and “flexibly structured data” are stored as collections in a MongoDB database. For a user such datasets are different from the tabular datasets, since the way they are queried is based on “query by example” rather than via SQL-like functionality.

E. Access Control

As mentioned before, the access control subsystem needs to support defining fine grained and non-hierarchical access

restrictions. To help enable this, each user (HEALS researcher) using the system has their own account; there are no shared / general accounts in our system.

The aim of the access control subsystem can be described as being able to answer the question *what level of access does this user have to this resource*. This question has three key elements: “level of access”, “user” and “resource”. A “resource” is, as far as the access control system is concerned, anything in the system that can be identified with an object identifier: a folder, a document, a dataset or some other object in the system. The identifiers used in the system combine a short type tag (such as “folder” or “dataset”) and a 128-bit GUID, the latter externally expressed as a 26-character string of seemingly random characters.

Our system defines four *levels of access* in a hierarchy, each granting more powerful access than the previous level: “none”, “meta”, “read” and “write”. The “none” access level does not grant any access to an item at all (not even the right to confirm that a resource exists). Such resources are hidden from the user, and attempts to access them behave in the same way as if the resource would not exist at all. The “meta” access level does not allow reading or writing a resource, but does allow the user to learn about its existence, and metadata such as size, name and content type or table structure. The “read” access level extends the “meta” right with the right to read or query the resource. The “write” access level further extends the “read” right with the right to modify or write the resource. The exact meanings of the concepts of “reading” and “writing” depend on the type of resource.

Access to resources is configured indirectly. Each user is assigned a set of *roles*. A system table stores *grants* that map a resource + role pair to an access level. To determine the access level a user has to a resource, the highest access level in all grant records for that resource for each of the user’s roles is determined. If there are no such record, the access defaults to “none”. Administrators have an implied “write” grant to all resources. To make a resource more widely accessible, an administrator creates grants for that resource.

To avoid making management of the grant database overly unwieldy, two additional mechanisms are in play, reducing the number of grant configurations required. One mechanism affects roles, while the other affects resources.

On the role side, there is a set of predefined special roles, and each user is explicitly assigned precisely one of these roles. These special roles form a classical linear access hierarchy, with each of them implying the lower ranked special roles, in this order: “anonymous” < “user” < “contributor” < “project partner” < “admin”. For example, if a resource is granted *read* access for the *anonymous* role, that resource will also have at least *read* access for users with the *user*, *contributor*, or *partner* special roles, without those grants needing to be made explicit.

On the resource side, the access level for certain types of resource is *delegated* to a different resource. In the *document storage* subsystem, any folder, document or link object delegates its access control decisions to the *top-level folder* it is nested in (the folder directly below the root folder). For their special role in the access control logic, these top-level folders are treated specially in the user interface, and are referred to as *document areas*. This special role of has a few implications. Since these now are the boundaries of access control in the document storage subsystem, users are not allowed to move documents from one document area to another. Allowing that would run the risk of exposing a document to other users that are not supposed to have access to it, or conversely, prevent other users from rightfully accessing it. Another consequence is the special status of the root folder itself, since it is effectively outside the access control settings; for that reason access rules to the root folder are handled explicitly, and access rules (including visibility) of its subfolders are determined by those subfolders.

For datasets, a similar delegation system exists: datasets are grouped into *data groups* that determine access rights.

F. Exposing the data

The data on the database site can be accessed by users in different ways, depending on the nature of the data and its storage medium.

For data that is stored as *documents*, users can view metadata about the document (such as its size and original name) and either download that document's file to their computer or, if applicable, open it in their browser.

For data that is stored as *tables* (or *views*) in the SQL based storage, a system exists that allows users to build a *query* to select the data rows and columns they are interested in, or to aggregate data. The results of these queries can be explored directly in the browser, or can be exported as CSV or XLSX files. The query system allows the user to construct a query in a web form, which on the server side is translated to an SQL query. Allowing the user to enter SQL directly was considered as an option, but it was deemed too hard to implement the access control checks reliably for this scenario.

For data stored as NoSQL collections, the user can build a "*query document*" and optional "*projection document*" that play a similar role in MongoDB as an SQL query does for a table or view. There is an important difference though: the "NoSQL query" can be exposed in a more raw way without running the risk of the equivalent of an "SQL injection attack". However, since constructing these "query documents" is error prone and probably unfamiliar to the users, a user interface to help construct these will be provided as well.

Depending on the nature of the data, the site also provides a variety of visualization modules. For example, data that

maps geographic regions or geo-coordinates to values can be visualized on maps. For other kinds of data, it can be useful to display histograms or time-based charts.

G. Interacting with other sites

As mentioned at the start of this article, this database site is not the only one in the HEALS project, and data that users would like to include in their analysis may exist in other HEALS databases, or non-HEALS external sites. Conversely, other HEALS sites may want to access the data in this Geodatabase programmatically. The following styles of site-to-site communication will be implemented:

- Obtain data from external sites to be joined in query results.
- Have links to external sites in query results
- Web services that allow programmatic access to our Geodatabase by external sites.
- These web services can also provide programmatic access for client applications, for instance an access library for users that use R as their data analysis system.

H. Cybersecurity

Security concerns and implications were already mentioned a few times in the preceding text. All database server applications are running on the same host as the web server and are only accessible locally on that host, dramatically simplifying the security design of the whole system. Only the HTTP and HTTPS ports are accessible through the firewall. For publicly accessible data, the server can be accessed anonymously via plain HTTP, but any authorized access requires HTTPS, ensuring that non-public data is encrypted in-flight.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No. 603946-2 (HEALS project).

CONFLICT OF INTEREST

The authors declare that they have no conflict of interest.

REFERENCES

1. HEALS portal at <http://www.heals-eu.eu/>