Integrated intelligent LEARNing environment for Reading and Writting

D5.2 – Game Usage Logging Mechanism

Document identifier	**D5.2_Game_usage_logging_mechanism.docx**
Date	**2013-12-22**
WP	**WP5**
Partners	**DOLPHIN, NTUA, UoM, UOB, DYSACT, EPIRUS, LBUS**
WP Lead Partner	**UoM**
Document status	**Final**

Deliverable Number	D5.2
Deliverable Title	Game Usage Logging Mechanism
Deliverable version number	Final
Work package	WP5
Task	Task 5.2 Game Usage Logging Mechanism
Nature of the deliverable	Report (R)
Dissemination level	Public (**PU**)
Date of Version	2013-12-22

Author(s)	Cantemir Mihu; David Johansson; Chris Litsas
Contributor(s)	
Reviewer(s)	Antonios Symvonis, Ioan Mihu
Abstract	This document describes the mechanism by which the serious game, learning activities and other applications are allowed to store data related to the user's interaction with them in order to be analysed and used in improving the learning process.
Keywords	Logging; Usage; Data Logger

Document Status Sheet

Issue	Date	Comment	Author
v01	2013-11-12	Initial version of the deliverable	Cantemir Mihu; David Johansson
v02	2013-12-13	Section on "Implementation"	Cantemir Mihu; Chris Litsas
v03	2013-12-20	Review on v02	Antonios Symvonis; Ioan Mihu
v04	2013-12-22	Final version	Cantemir Mihu

Project information

Project acronym:	**ILearnRW**
Project full title:	**Integrated Intelligent Learning Environment for Reading and Writing**
Proposal/Contract no.:	**318803**

Project Officer: Krister Olson

Address:	L-2920 Luxembourg, Luxembourg
Phone:	+35 2430 134 332
E-mail:	krister.olson@ec.europa.eu

Project Co-ordinator: Noel Duffy

Address:	Dolphin Computer Access Ltd. Technology House, Blackpole Estate West, Worcester, UK. WR3 8TJ
Phone:	+01 905 754 577
Fax:	+01 905 754 559
E-mail:	noel.duffy@yourdolphin.com

Table of Contents

1 Introduction

Games that involve reading will be used to increase the degree of children involvement in learning activities. However, this isn't the only way ILearnRW aims to benefit from the employment of serious games.

A second, more ambitious goal is to use games as an evaluation tool of the learning process supported by ILearnRW and as a mechanism of updating a child's profile. Consider a game played by a child, the course of which depends on the child's performance at several reading/spelling tasks that have been incorporated in it. The numbers of (failed) attempts required by the child before she completes the task can be used to infer the degree of progress made towards a learning objective. In the case of reading tasks, the user's performance during the game can be used to evaluate whether comprehension actually took place. Moreover, the time required to make the correct decision/action can be used to evaluate the degree of progress towards the learning objective concerning reading and writing. To support the use of games in evaluating the child's progress, a game usage logging mechanism has to be developed which records the user's actions during the game as well as the evolution of the game over time so that several hypotheses with respect to the effectiveness of the learning process can be concluded by correlating the stored information.

A generic learning application can support its user by using the user's profile, which gives information about the current dyslexia status of the user's problems and the logic built into the application. Additionally, a learning application could use in its learning strategy also the history of the user, its evolution in time. By "history of a user" we understand the chronologic order of the actions she took during usage of an application and the outcomes of each action. By having access to the user's history, the application can adapt its learning strategy (e.g. not display the same words over and over again if the user has successfully dealt with in the past). The learning application must mimic the memory of a teacher, who knows exactly what lessons a child already has learned so far and also knows with what type of lessons the child has the most problems. Using this memory-kind of information, the learning process becomes adaptive, i.e. using the same learning application multiple times will imply different actions and different results.

ILearnRW consists of multiple applications supporting children in their learning process. All these applications must work together in order to better support the children. Each application could have its own "memory", and could try to improve the child's problems individually, using only this memory and the current state of the user's profile. But being able to share this memory between all applications would have the benefit of improving the overall experience for the child. The learning strategy of each application will be much improved by taking as input the information about actions from other applications and their outcomes. For instance, by using the past information, each game is able to intelligently select words to be displayed.

The child playing several learning applications one after the other is very similar to a child attending several courses with different teachers. If the teachers don't know what the other teachers teach the child, it might lead to the same information being presented twice or to activities inappropriate in terms of complexity or relevance. If the teachers inform each other about the lessons and the activities the child made so far, they can adapt their course and provide much more targeted actions or learning material to the child, thus improving the child's overall learning curve.

The ILearnRW system is built upon the concept of *collective, shared memory*. This memory holds every piece of information created by the usage of ILearnRW's applications. All applications are able to store information to this memory and are able to research in it.

Besides helping the applications adapt their learning strategies as described above, by having the complete history available we have the benefit of being able to generate statistics and reports. By analysing the data we can answer questions like:

- What is the most played game?
- How many rounds are played per game?
- How much time does a child spend playing the serious game?
- What is the problem area that children are having the most problems with?
- What is the success rate of each learning session?

An additional use of storing the trace of the application's trace is the possibility of *replaying* these actions. For this purpose, the logs stored could be seen as a complete recording of the child's interaction with the applications, allowing a teacher to see and review the child's course of actions. If the application implements a feature similar to "Replay", the children could use it themselves too, in order to see what they did well and what they did wrong. This feature isn't a requirement of the ILearnRW system, but the applications could make use of the Data Logger "history" to implement such a feature.

In terms of the ILearnRW architecture, the component responsible with storing the logging information is called the *Data Logger* component. Components which want to log data are called throughout this document *Applications* (games, activities or other applications). Applications have to instruct the Data Logger to store data, it has an imperative nature since it cannot observe and log actions by itself. It is the responsibility of each application to inform the Data Logger what and when to store.

2 Requirements

2.1 Functional requirements

The Data Logger must meet the following functional requirements:

- *Receive log entries from different applications*: several applications will use the Data Logger and will push data into it.
- *Store the received log entries*: All received data must be persisted in a safe and permanent manner. Data will be stored as-is, but also in a format which allows for fast retrieval.
- *Respond to different type of queries and return information needed* to cover at least the use cases described in 3.1.

2.2 Non-functional requirements

- *Performance*: applications will require information out of the Data Logger's database in real time. Therefore the Data Logger must be able to respond very quickly so that no lag can be noticed during game play. Both the retrieval and the saving of data must be equally fast.
- *Secure*: the data which will be stored into the Data Logger is tightly related to the child's dyslexia status and evolution. It is therefore sensitive data which must be securely stored.
- *Generic*: The Data Logger is designed without knowing all specific requirements of the applications that will use it. In this sense, the Data Logger will allow applications to store data that matches the designed contracts of the Data Logger, as well as application specific, own data.

3 Data Logger Component

The Data Logger is the software-based mechanism for enabling applications to store data that can be later recalled by applications. The Data Logger is generic in the sense that it will allow different components to store data that is appropriate to their nature.

The Data Logger is implemented as a web service, since it must be accessible for a wide range of applications and from several devices. It is also implemented as a web service in order to have all activity history of a child stored centralized, so that it can be easily analyzed by a teacher at any time.

3.1 Use Cases

When designing the functionality and the data model of the Data Logger some use cases and requirements were taken into consideration. These requirements can be grouped in two categories based on the goal for which the data is needed: statistical data and runtime data.

3.1.1 Statistical data

A. For a given app-round session, generate app-round-printout which includes
    a. Child name, application name
    b. Session identification details (learn-session, app-session, app-round session, etc).
    c. Date, time, duration.
    d. Specific problem addressed (problem area [row] and type [column]) of child's profile.
    e. Data used (which words were used in the round), sorted in order of appearance
    f. For each word used (some or all of the info below)
        i. The word
        ii. Relevant profile entry for the word (may be different from the "specific problem addressed")
        iii. Time of screen entrance and exit
        iv. Whether it was successful
        v. How many times it required to succeed.
        vi. Details on failed attempts

B. For each app-session, give details about each of the rounds it is made of (use data from **Error! Reference source not found.**).

C. For each app-session, give cumulative details such as:
    a. Child name, date, application name
    b. Number of rounds
    c. Total time spent
    d. All words used
    e. Specific problems addressed
    f. Success ratio (total number of successfully dealt with words divided by total number of words)

D. For each learning session, report
    a. Identification data
    b. The specific problems addressed
    c. The applications that the session used (totally or per problem)
    d. The words used
    e. Success rates

E. For a given child, print list of learn-sessions (using data from **Error! Reference source not found.**)

F. For a given child, print all words the child managed or didn't manage to read correctly. For each word print the games that used it, the problem area used for and the number of times tried.

G. For a given child, what is the hardest specific problem to overcome so far (based on failed rounds)?

H. What is the problem area that children are having the most problems with? Give statistics based on the age of children.

I. Rank the words used in games based on their difficulty (denoted by ratio of fails). Give statistics based on the age of children.

J. Slice the above statistics only for boys or girls.

K. How much time a child spends using the serious game?

L. How many app-rounds did a child play? How much time per round?

M. Is performance on the adventure better than that of the play mode?

N. What contributes more toward the "improvement of the profile"? Applications played during the adventure or during the play mode?

3.1.2 Runtime data

A. For a specific child, and a specific problem (i.e., profile entry) get the following lists (they will be used by the application in order to decide which words to present in the next round):
    a. Words-app pairs the child has used in IlearnRW during the last X learning sessions. Also give details whether successful or not with the specific word.
    b. Get the difficult words of a child (all words identified at some point as difficult). Which of them are still difficult and which aren't any more? How many efforts it took to learn them?
B. For a given app-round session, export the "play_app_round_recording" (like a recording, generated from data in the Data Logger)

3.2 Design considerations

During the analysis of requirements there were several models and implementation solutions taken into consideration.

3.2.1 Plain data storing

Because not all requirements of the applications were known at the beginning, the first solution which was proposed used a simple, straight-forward organization of the data. It was similar to a common log file or the Event Viewer of the Windows operating system: all events were marked with a timestamp, a text-based value, a tag (or a marker) and were attributed to a user and/or an application. In this sense, applications could "push" generic values to the Data Logger and leave a full trace of the activities inside the application. Using a generic text-based value, the applications could store there whatever they desired, without any constraints. The huge benefit of this solution is the ease of implementation, the fast storing (no constrains or computations needed) and the very fast retrieval. Additionally, the provided interface for the applications was extremely easy to understand and to implement. But this solution had also the following problems: extracting correlated data was difficult and time-consuming, since the stored rows didn't have any correlation between them and the applications were limited to storing one value per entry (multiple values could be stored in a specific format, like XML or JSON, but that would make targeted searching impossible).

3.2.2 Extended plain data storing

The above described model could have been extended by providing several value columns. In this way, the applications would have been free to decide what they store in the provided columns. This solution benefits from the same advantages as the previous one, but also with the same disadvantages (except for the extended values which applications could use).

3.2.3 Typed data storing

Providing only generic text-based fields makes things difficult when it comes to generating statistics and reports, since the Data Logger would not know what the applications actually stored as data. Since we know, based on the requirements, what the most important values to be stored by the applications are, we can provide typed value fields. In this way we can have value fields for storing words, since almost all applications deal with words. Or we can have a typed value field stored as a decimal number containing the duration of an action inside the application.

Although this solution has the most benefits over the previous ones, there still is the disadvantage of not being able to correlate rows easily.

3.2.4 Typed and aggregated data storing

Applications will need to ask the Data Logger to provide information from its data based on some logical grouping of the records previously stored. All actions triggered by an application must be logically grouped together based on some criteria. The most useful grouping is related to activities which happen in the same application or which happen inside a given time frame. Having information about the time of the action and the application that triggered it would allow this grouping, but no more. Additionally to these groupings, applications might want to group records based on sessions and/or rounds within a session. A session is thus defined by all activities which happen in a given time-frame. The time-frame will start when the application informs the Data Logger explicitly about the start and will end when the application explicitly informs the Data Logger about the end.

Handling sessions requires a deeper analysis. They have to be correlated between the client where they happen and the server where they are traced. Trying to synchronize these sessions will make the implementation of the clients more difficult, because they would need to keep synchronized information about these sessions. They could retrieve this information from the Data Logger and consequently reuse it when sending data which belongs to the same session. To make things easier for the clients, the handling of the sessions and the logic needed to correlate recorded data can be implemented in the Data Logger. In this way we will have

more flexibility because we can change and adapt the logic independently from the applications and we can keep the API for the applications very simple.

From the application's point of view, the information that they will send to the Data Logger is composed of data that needs to be logged and data that is used as special markers. The Data Logger's role in this case is to interpret the special markers and group the upcoming data using this interpreted information. In a generic sense, the applications trigger a marker, then they will send data belonging to this marker and when finished will send a marker's end message. In this sense the applications just need to provide simple messages to the Data Logger: one's that carry important values, like the used words during an application, and others who by their type are meaningful to the Data Logger.

This is only possible if the Data Logger exposes some predefined message types, which the applications must respect and use properly. This contract between Data Logger and the applications is defined throughout this document. Notice that the contract refers only to some message types, while the applications are still free to send their own custom message types with appropriate values.

3.3 Data requirements

The Data Logger needs to store a set of specific, typed fields as well as generic values. All stored records have a common field, the user-id, by which a user can be uniquely identified in the ILearnRW system, as well as a generic field, containing a value sent by the application.

Data is grouped in logical entities, called *LogEntries*. A LogEntry is usually triggered by an event inside a component of the ILearnRW system and consists of the following information:

```
{
   username: string,
   applicationId: string,
   timestamp: datetime
   tag: string,
   word: string,
   problem: string,
   duration: decimal,
   level: string,
   mode: string [ADVENTURE|PLAY|READ, …],
   value: string,
}
```

username: the unique identifier of the user triggering this action. Based on this identifier, the action can be bound to a user, so that the records make up the history of that user.

applicationId: since several applications will create LogEntries, they must be uniquely identified. All applications inside the ILearnRW system will have a unique identifier. Developers of applications running in the ILearnRW system will create a unique application ID which they will use and send along with each LogEntry. The list of existing applications ID is stored and maintained in the ILearnRW source code.

timestamp: the date and time information when the event was triggered. This information can be automatically set up by the server if it is missing, but it would be more accurate if the applications send this with the exact time when the event happened, otherwise there might appear time differences introduced by the time it took the message to reach the server and the time used by the server to process the request. If sent, the value must be in UTC time.

tag: the tag-field contains an identifier of the LogEntry itself. It indicates to the Data Logger what type of data was sent and allows for interpreting this data. Using tags we can allow a better classification of the LogEntries, making it easier to search, analyse and group events. Tags are described in detail in chapter 3.4.

word: all applications in the ILearnRW system use words. It is therefore very important to store all used words. Probably most of the data coming from the applications will be directly related to a word, and most analysis will be done using words. So having them stored in an own field will make analysis easier.

problem: this field will be filled in by the application if the event which has to be logged refers directly to a problem inside the user's profile. It directly identifies a profile entry by specifying the problem's category and its type.

duration: if the event which happened on the application that needs to be logged had a given duration known by the application, then this field can be used to store this value. For instance, if an application would like to allow a user to drag and drop something on the tablet's surface, then this field could be used to store the actual duration between the starting of the drag and the drop.

level: if the application consists of multiple levels in terms of difficulty or progress, then this field can be used to store events which happen in a given level.

mode: generally, the ILearnRW system will be used by users in different modes (for instance in Adventure Mode, Play Mode or Read Mode). Since it is important to know how the system is actually used and what impact each mode has on the learning process, we need to store this information as well.

value: the value field is generic and contains application specific information for the event. This data will not be interpreted by the Data Logger in any way, but the Data Logger allows to search for records containing a specific value, so that applications might use this field for own purposes, not known ahead. This field's type is set to "string" so that more complex structures or objects may be stored in it, given that they are serialized into a text-

based format like XML or JSON. Note that the Data Logger will not try to deserialize or "understand" these values, and when searching for matches in this field it will only use standard string matching patterns, like wildcards.

3.4 Tags

As described above, the Tags are used in the Data Logger in order to identify and give meaning to the log entries. Tags are strings, usually self-describing the nature of the log entry. For instance, an event which is triggered by an application when the child presses a button might be tagged using the string BUTTON_PRESSED or PUSHED_BUTTON.

The Data Logger needs to be able to "understand" some of the tags in order to build an extended data model used for statistics and analysis of the data. These known tags are called system tags. Currently, the following system tags are defined:

LEARN_SESSION_START	Sent whenever a learning session is started. Sessions are described in chapter 3.5.
LEARN_SESSION_END	Sent when a learning session ends
APP_SESSION_START	Sent when an application is started
APP_SESSION_END	Sent when an applications is closed
APP_ROUND_SESSION_START	Sent when a new round starts inside an application which supports rounds
APP_ROUND_SESSION_END	Sent when a round ends
WORD_SELECTED	Sent by an application when a word was chosen to be used inside an activity. This doesn't imply that the word has also been displayed to the child.
WORD_DISPLAYED	Sent whenever a word was displayed inside an application but was not processed in any way by the child
WORD_SUCCESS	Sent whenever a word was displayed inside an application and from the application's point of view the user handled the word successfully.
WORD_FAILED	Sent whenever a word was displayed inside an application and from the application's point of view the user failed to handle the word successfully.
PROFILE_UPDATE	Sent when the application triggered an update in the user's profile.

LOGIN	Sent when the application logged in the user successfully.
LOGOUT	Sent when the application logged out the user successfully.

Note: The guideline for defining tags is to use upper case and split words by an underscore.

3.5 Sessions

In an intervention session, the goal may be achieved by a combination of activities. All of these activities should be seen as a *learning session*. A learning session refers in terms of the ILearnRW system to all the activities a user made since he logged into the system and until he logged out. An activity played within a learning session (as part of a learning program) using a specific application is seen as an application session (*app-session).* App-sessions may concern mini-games, the serious game, the reader and other applications of ILearnRW. During an app-session a user may play several times the same activity (with different data each time). These are called "rounds". A *round session* consists of all actions made during the same round.

3.6 Data Model

Data is stored into the Data Logger into two different models. One is used for storing the data coming directly from the applications and the other one is used for building up an aggregated representation of the received data. The first model (*Table Model*) is the plain representation of the records and uses a table to store the received data. The second model (*Cube Model*) is designed with respect to fast queries, aggregated data and statistics.

3.6.1 Table Model

Since the Table Model uses just one database table, its representation is straight forward. The used table doesn’t have any relations to other tables. The primary key is only used for indexing purposes.

log_entries		
PK	**id**	**int**
	username	**varchar(100)**
	applicationId	**varchar(20)**
	timestamp	**timestamp**
	tag	**varchar(100)**
	word	varchar(100)
	problem_category	smallint
	problem_index	smallint
	duration	decimal(10;2)
	level	varchar(100)
	mode	varchar(100)
	value	varchar(2000)

Figure 1 Table structure used in the Table Model

3.6.2 Cube Model

The Cube Model is used to represent multidimensional data. In terms of Online Analytical Processing (OLAP)[1], the data consists of multiple dimensions and of numeric facts called measures. The dimensions are the characterization of the measures. Dimensions and measures build up an *OLAP cube*. The Data Logger's cube is stored in a relational database (ROLAP) using a *star schema*. This way of storing data makes use of the benefits of using a relational database in terms of performance, space usage and data integrity. The schema of the Cube Model is presented in the following figure:

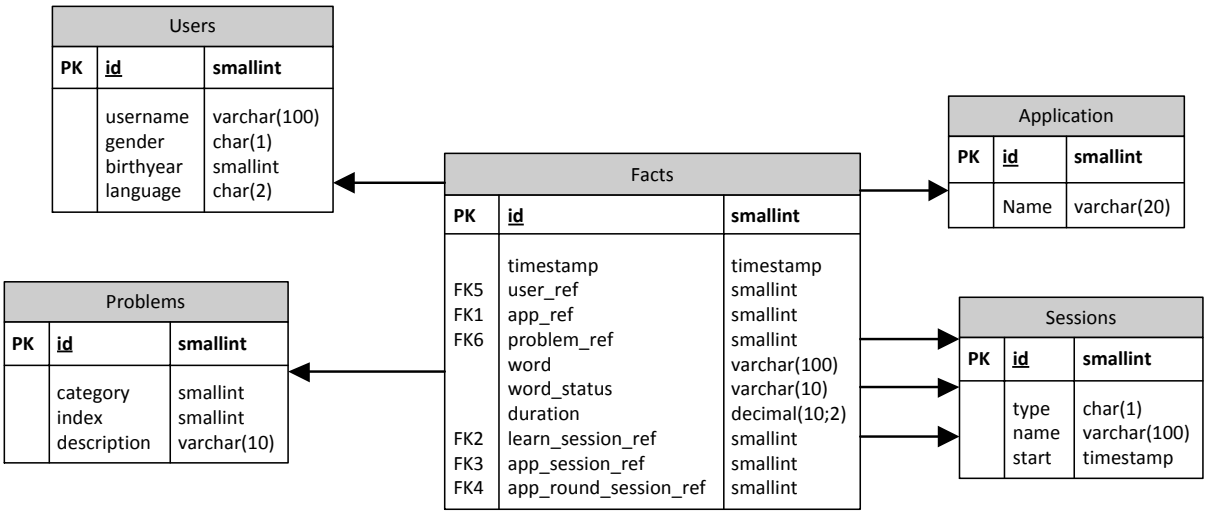

Figure 2 Schema of the Cube Model

[1] http://en.wikipedia.org/wiki/Online_analytical_processing

We defined the following dimensions which will be useful for analysis, statistics and research:

- The child, with the following sub-dimensions: gender, age and language
- The used applications
- The user's sessions
- Addressed problems

The following dimensions are stored in the facts table (they are called degenerated dimensions):

- The time
- Words used

As measures we currently have the count and the duration. These measures can be easily extended in the future, based on additional needs that will arise by extending the Facts table with additional columns and/or dimensions tables.

The logic by which the Cube Model is built by the Data Logger is shown in the following diagram:

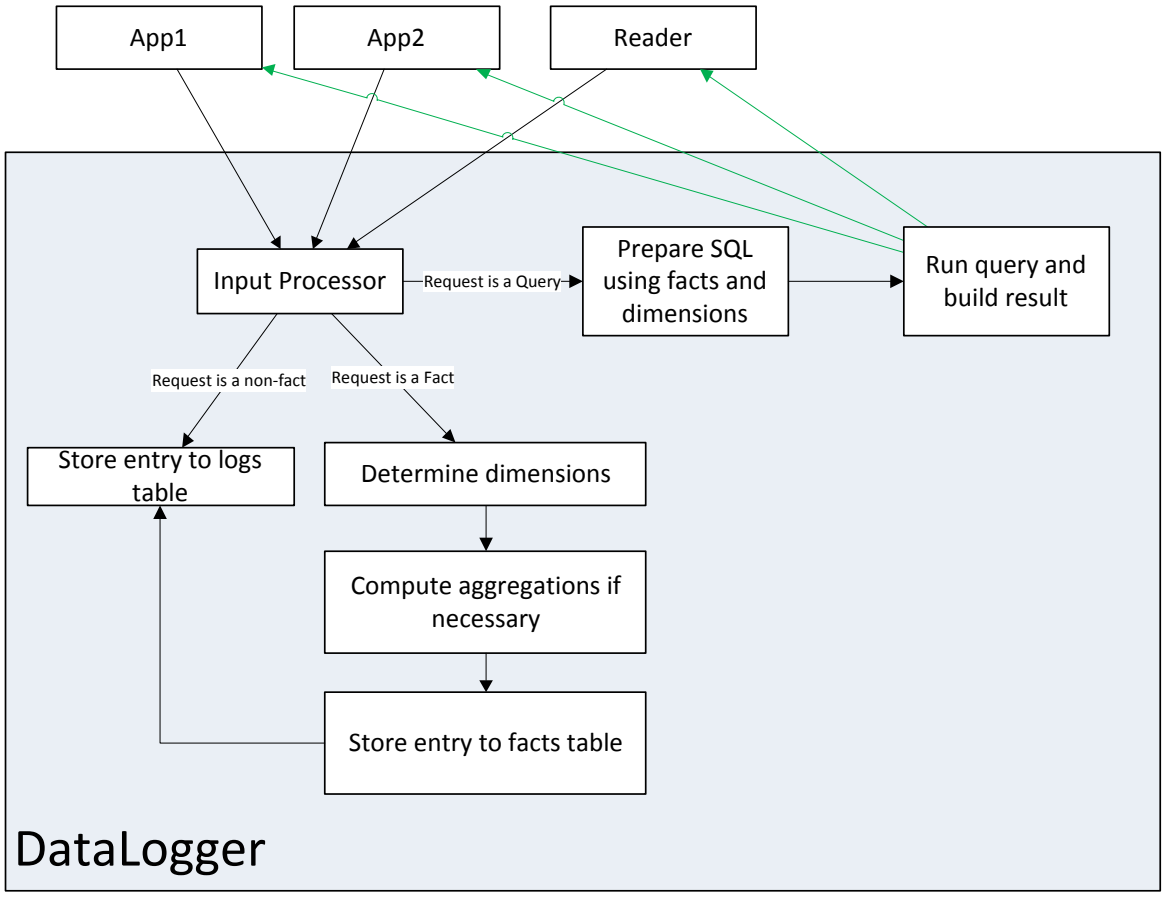

Figure 3 Data Logger logic for building up the Cube Model

Note that the LogEntry received from an application will always be stored in the log_entries table. The condition "LogEntry is a fact" is built into the Data Logger and is based currently on the system tags defined in 3.4.

3.7 Exposed API

The Data Logger exposes an interface which allows both pushing and receiving data. The API is built using the Representational State Transfer[2] (REST) concept in mind, but brings some simplifications especially in avoiding using the PUT and DELETE verbs. Pushing data means actually passing a JSON or XML encoded message via a HTTP POST to the Data Logger. The API for extracting data is built around the data which is needed for the applications and for the statistics.

The API is exposed via a secure channel using standard HTTPS. An authentication token must also be passed with each call to assure that creation and retrieving of logs is done by an authenticated user (see details about the User Authentication Component for details on authenticating users).

3.7.1 Storing and retrieving logs

For retrieving plain data (as they were sent to the Data Logger without any processing) the applications must create and send a *LogEntryFilter*. The LogEntryFilter is a filter that is used to choose which logs are to be retrieved from the database. This allows an application to request logs filtered by: user id, session id, one or more tags, start and end time.

Create a log:
To create a log entry you take all the variables in a LogEntry and put them in a JSON (JavaScript Object Notation) string.
e.g.

```
{
  username: "Joe",
  tag: "victory",
  value: "You won the game",
  applicationId: "Heads or Tails",
  timestamp: 2012-05-19 03:14:07
}
```

[2] http://en.wikipedia.org/wiki/Representational_state_transfer

Then you do a POST call to the server with the containing JSON string.

Call:

```
POST /log?token=<token>
```

Get logs:
To get logs from the Data Logger you create a JSON string containing the values of a LogEntryFilter. The following LogEntryFilter instance would request all log entries created for user "Joe" by the application "Heads or Tails" with tags matching "defeat", "victory" or "stalemate" in a given timeframe:

e.g.

```
{
 username: "Joe",
 applicationId: "Heads or Tails",
 tags:["defeat", "victory", "stalemate"],
 timestart: 2013-01-11 03:14:07,
 timeend: 2013-01-11 03:16:33
}
```

Then you make a GET call to the server with the JSON object in the request body.

Call:

```
GET /log?token=<token>&data=<json>
```

Response:
You will get logs back in JSON according on how your filter was built up.

```
{
  result: [
   {
      username: "Joe",
      applicationId: "Heads or Tails",
      value: "You won the game",
      tag: "victory"
      timestamp: 2012-05-14
  },
    … (more logs)
 ]
}
```

The Data Logger would be used by the applications corresponding to the following sequence diagram:

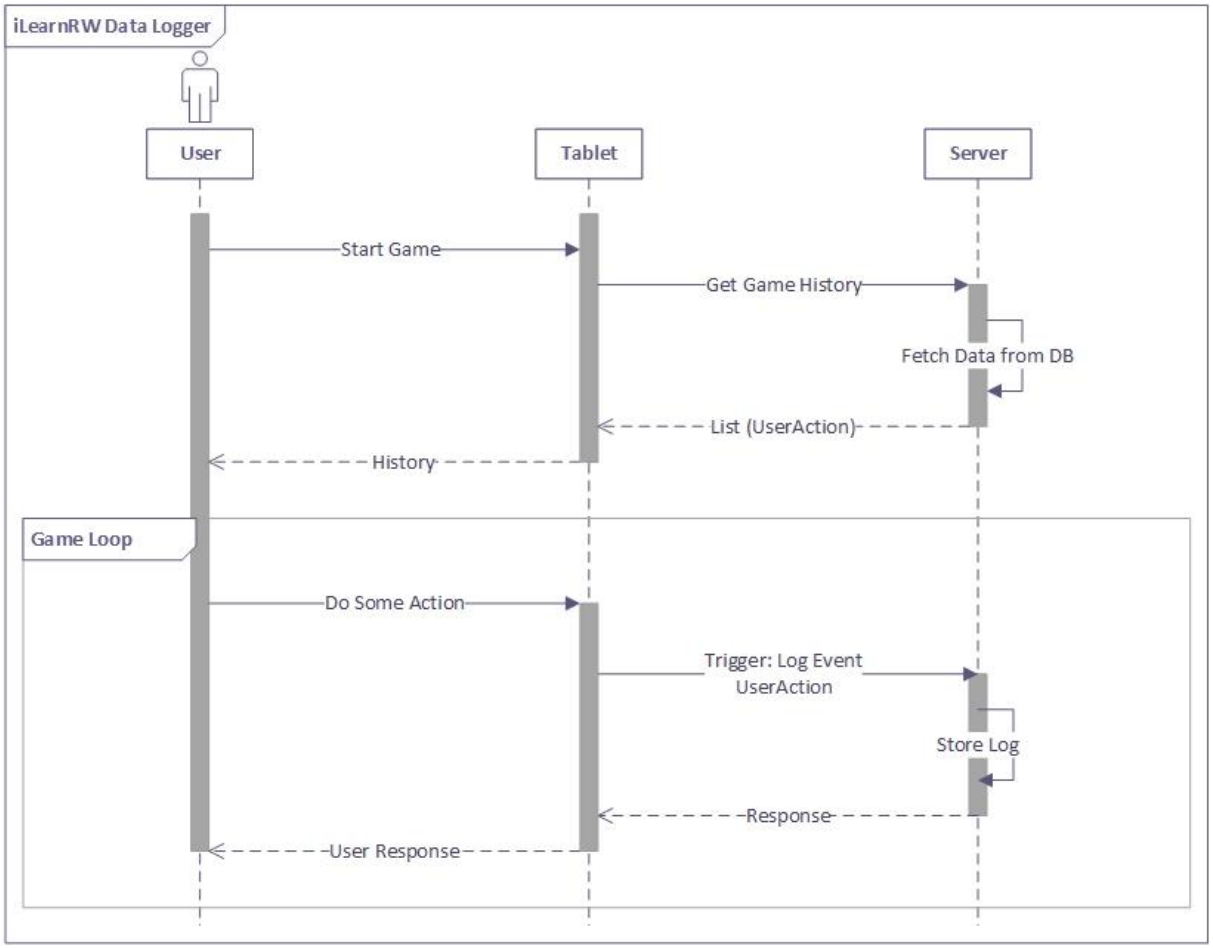

Figure 4 Sequence Diagram for Data Logger Component

3.7.2 Retrieving statistical data

The API developed for retrieving statistical data is built taken into considerations the main requirements identified from the use cases described in chapter **Error! Reference source not found.**. The implemented API is thus tailored for these use cases, and is therefore probably not covering all future requirements, since it is extremely difficult to cover all data present in the Cube Model using a generic RESTful representation.

All API for retrieving statistical data have the following in common:

- They are all mapped to the GET HTTP verb. Other HTTP verbs are not supported.
- They return a list of items, specific to the addressed resource.

- All methods can receive an additional HTTP GET parameter, called *count*, which, if set to true, will return only the count of items, not all items. Other aggregations like sum, average, etc. are not currently implemented.
- All methods can receive additional HTTP GET parameters for defining the time frame of the facts to be retrieved. The parameters are called *time_start* and *time_end.*

```
/{username}/sessions/{session_type}/
```

⇨ Returns a list of all sessions for the given user, filtered by time

```
/session/{id}
```

⇨ Returns the details of the given session.

```
/{username}/words?status={failed|success}
```

⇨ Returns the words that a given user managed to handle correctly or didn't manage to handle correctly

```
/{username}/problems
```

⇨ List of difficult problems to overcome, in descendent order based on the difficulty to overcome

```
/{username}/{problem_category}/{problem_index}/words
```

⇨ List of words used by the child for addressing a given problem.

```
/problems/{age}/{gender}
```

⇨ List of problem areas the children have the most difficulties with, ordered based on the difficulty and grouped by the age of the children having difficulties with it.

```
/words?status={failed|success}
```

⇨ All words ranked by their difficulty

```
/children/{age}/{gender}/words?status={failed|success}
```

⇨ All words ranked by their difficulty, grouped by children's age.

```
/apps/words?status={failed|success}
```

⇨ All words, grouped by applications

```
/app/{id}/facts
```

⇨ Return all facts for a given application.

```
/facts/{id}/
```

⇨ Returns the fact identified by {id} with all its dimensions denormalized

4 Implementation

The Data Logger is implemented as a Web Service using the Spring MVC[3] framework (Java language). The Spring Web MVC framework is part of the general purpose Spring framework. The Spring MVC framework allows a direct decoupling of the data (the Model), the visual representation (the View) and the logic binding them together (the Controller).

4.1 Data

Access to the data is done using JDBC[4] and MySQL. JDBC is the industry standard for database-independent connectivity between the Java programming language and a wide range of SQL databases and other tabular data sources, such as spread-sheets or flat files. The Java-based web services rely on JDBC to retrieve data from MySQL using SQL queries. This is performed via JdbcTemplate class, which executes SQL queries or updates, allowing the developer to focus on providing the SQL and processing the results.

The database accessing code is decoupled from the Data Loggers main functionality by Service interfaces. These Service interfaces expose methods for handling data from the Model. The Service implementation will pass over the functionality to Data Access Objects (DAO). The DAO classes will execute the SQL queries and return Model objects. The Service classes are thus loosely coupled to the Data Logger's main code, so that Dependency Injection and Inversion of Control can be used, for instance to change the Service with a mockup implementation, used for Unit Tests.

4.2 Representation

Spring MVC allows processing incoming and outgoing data to different formats. This is done through Content Negotiation of the HTTP protocol, Request Mappings and Data Serializers which transform data to different formats, like JSON or XML.

The Data Logger is implemented to support JSON and XML representations, with the default set to JSON.

[3] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[4] http://www.oracle.com/technetwork/java/javase/jdbc/index.html

4.3 Application logic

The application logic is stored in the Controller classes of the Spring MVC framework. A controller is a class exposing several methods mapped to incoming HTTP requests. Each controller method has its own parameters and is session-less and stateless (i.e. it does not store Client state).

Based on the incoming parameters, the controller method has to check permissions (i.e. if the user is authenticated or not) and invoke the appropriate Service class. If data must be aggregated from different Services, then the controller method will have to call all involved Services and gather information which it will then present as a result.

5 Other considerations

5.1 Performance

The performance of the Data Logger has two main aspects: performance when storing data (i.e. how fast and which volume of data can the Data Logger store in a given timeframe) and performance when retrieving data (i.e. how fast is it able to retrieve the data for a given set of filtering parameters).

When storing data, the Data Logger is influenced by the performance of the web server it runs on (how many HTTP requests can it process) and by the performance of the database server into which data is stored. Since we do not expect a huge load, both cases should be sufficiently covered by the chosen hardware and software infrastructure. On the database side, the solution is optimal in the sense that only one table is used, which makes writing easier, no transactions or blocking operations are needed.

When retrieving data, the amount of data existing in the table as well as the amount of data requested makes retrieving more or less fast. Since there is only one table there is no need for cross-table joins and indexing on the proper columns should improve the overall query performance. If the table should grow very large in size, an option available is to move old data (e.g. older than two months) into a different table.

5.2 Extending the OLAP functionality

The API exposed by the Data Logger (described in 3.7.2) is tailored to the general needs of the ILearnRW system. In order to better analyse the data, OLAP clients can be used to fetch, aggregate, display and navigate the OLAP data. Such OLAP clients generally make use of the MDX language[5] to query OLAP databases and to return multidimensional data so that it can be displayed properly. Supporting OLAP clients is beyond the goal of the Data Logger, but, using a software solution like *Mondrian*[6], we would have the possibility to reuse exactly the same MySQL database and the Cube Model. Mondrian would then map to this model via configuration and allow different applications to execute MDX queries upon this model. MDX queries are similar to SQL queries, but they allow multiple dimensions and measures to be specified.

[5] http://en.wikipedia.org/wiki/MultiDimensional_eXpressions

[6] http://mondrian.pentaho.com/documentation/olap.php

5.3 Asynchronous calls

In order to keep track of the usage (i.e. storing data into the DataLogger), the application will have to make asynchronous calls to the service on the server. If the action to be written can't be stored on the server (e.g. in case of a connectivity problem), then there are two options:

- Stop the application workflow and display a message like "Internet connection is needed in order to continue. Please wait…"
- Store the actions in a buffer and periodically try to push them to the server.

It should be up to the applications to implement one of these options, whichever suits the needs of the application better.

6 Conclusions

In this deliverable, we presented the Data Usage Logging Mechanism of iLearnRW. While in the initial design of the project it was indented that the data-logging mechanism will only serve the serious game, during the design of the iLearnRW Architecture it became evident that the data-logging mechanism could find a project-wide use. Details on the implementation are also provided. It is anticipated that during the usage of the data logging mechanism by the iLearnRW applications (serious game, learning activities, statistics, etc.) minor changes of the tagging system may be required. If deemed necessary, a revised version of the deliverable will be issued.