

Database Connection for the CPS Facilitator Tool

Jos Kraaijeveld

December 16, 2011

Contents

1	Introduction	3
2	How to use	3
2.1	Initialization	3
2.2	Basic queries	3
2.2.1	Retrieving data	3
2.2.2	Storing data	4
2.3	Arguments	5
3	Specification	6
3.1	Models	6
3.2	Connectors	7
3.3	PHPDoc	7
4	Future Work	7
4.1	Adhere to Open-Closed Principle	7
4.2	Deal with invalid or missing values	7
4.3	Optimize queries	7
A	Class diagram of the models	8

1 Introduction

This document describes the database design and implementation for the CPS Facilitator Tool. The goal is to make sure everyone can communicate with the RDF-based database without having to write queries. The DBInterface is written in PHP and requires slight PHP knowledge before use. It is heavily based on initial work by Bas van Nuland.

If you use this framework, you can skip to the ‘How to use’-part of this document (2). If you are planning to improve this framework, I recommend reading the RDF Primer¹ and SPARQL Query Language for RDF². A full specification of the current implementation can be found in section 3. The areas I suggest improving upon first are described in section 4.

2 How to use

2.1 Initialization

Before the database can be used, you should initialize a *DatabaseInterface* object. This is the only database class you will use. This is done calling the DatabaseInterface constructor.

```
$db = new DatabaseInterface();
```

2.2 Basic queries

2.2.1 Retrieving data

Getting data is done by calling the get method of the DatabaseInterface class. The specification for this method is as follows.

Function get()	
First argument: \$type	String
Second argument: \$arguments	Array
Return type:	Array/Variable

The return type is an Array of objects with the type specified as the first argument. For instance, if the first argument is “user”, the function will return an array of User objects. The second argument can be left out if you want to retrieve all entries of a specific type. Below are a few examples showing the usage of get(). A description of what arguments are possible per type is given in section 2.3.

¹<http://www.w3.org/TR/rdf-primer/>

²<http://www.w3.org/TR/rdf-sparql-query/>

Figure 1: Retrieving all questions and echo their title.

```
$questions = $db->get("question");
foreach ($questions as $question)
{
    echo $question->title;
}
```

Figure 2: Retrieving all surveys containing questions with IDs q1 and q2, created by the user Jos, printing all questions in those surveys

```
//To get all surveys created by the user Jos, we need his UID.
//We first query the database for the User object belonging to Jos.
$userResults = $db->get("user", array("name" => "Jos"));
//Assuming there is a result, the UID is:
$josUID = $userResults[0]->uid;
//Now to get the requested surveys:
$surveys = $db->get("survey", array("questions" => array("q1", "q2"),
                                     "creator" => $josUID));
//And to print all the questions in these surveys:
foreach($surveys as $survey)
{
    echo "All the questions in " . $survey->name;
    foreach($survey->questions as $question)
    {
        print_r($question);
    }
}
```

2.2.2 Storing data

Storing data is easy, as long as you stick to using the given PHP Classes. Retrieve these classes by using the `get()` function, and pass them as a parameter to the `set()` function of the `DatabaseInterface`. This `set()` function will determine what type the object is and store it at the correct location. The method overview is as follows:

Function <code>set()</code>	
First argument: <code>\$rToolObject</code>	Variable

Another important thing to note is that currently it will overwrite a previous object with the same UID

in the database. The following examples show how to create new objects and save them, as well as edit old objects and save them.

Figure 3: Creating a new Answer object and storing this in the database.

```
//For this example, I choose a random question to answer
$questions = $db->get("question");
$question = $questions[2];
//Note two things:
//1 - If you pass 'null' as first argument when creating any object
//      A new UID will be generated, indicating a new object.
//2 - Depending on the question, there can be multiple answers
//      This means the values-argument (third argument) is an array.
$answer = new Answer(null, $question, array("12345", "four"));
//Save the answer in the database
$db->set($answer);
```

Figure 4: Getting a Survey object from the database and removing the first question. Also alter this question.

```
//Get the survey
$surveyResults = $db->get("survey", array("uid" => "b91d39e8667372e220bb861b3f94b5bd"));
$survey = surveyResults[0];
//Remove the question
$question = $survey->questions[0];
unset($survey->questions[0]);
//Change the question
$question->title = "New Title";
//Save the survey and question
$db->batchSet(array($survey, $question));
```

2.3 Arguments

The arguments you can supply when calling the `get()` function differ greatly per type. Unfortunately, there is no way around this, so here is a complete overview of arguments per type.

Type	Argument name	Argument type	Extra notes
Answer	uid question values	String String Array of Strings	UID of the question
AnswerSet	uid survey respondent answers	String String String Array of Strings	UID of the Survey UID of the Respondent UIDs of the Answers
Application	uid title description style	String String String String	
Question	code title type description category definedanswers	String String String String String Array of Strings	String values of possible answers
Respondent/User	uid name password	String String String	Use hashes to store passwords
Session	uid title creator datetime applications surveys answersets	String String String String Array of Strings Array of Strings Array of Strings	UID of the User Unix Timestamp. No way to search on intervals (yet). UIDs of the Applications UIDs of the Surveys UIDs of the AnswerSets
Survey	uid title description creator questions	String String String String Array of Strings	UID of the user that created this survey UIDs of the Questions

Whenever an Array has to be supplied, it will match for results that satisfy *all* the constraints given in the array.

3 Specification

3.1 Models

The framework heavily relies on the Object Oriented Programming paradigm, and only allows you to create, edit and store data through instances of precreated classes. All these classes inherit from a global `ResearchToolObject` class. A UML class diagram of the model classes can be found in Appendix A. The most important thing to note is that references to other objects are not evaluated immediately. Rather, these classes with references have to override the `evaluate()` function, which in turn tries to query the database and resolve those UUIDs to model objects. This gives us two advantages: initial queries do not scale exponentially because of the 'lazy' evaluation, and the boolean return value notifies us when there exists an incorrect reference. By evaluating before saving an object, we ensure that there cannot exist invalid values in the database.

3.2 Connectors

Although the front end developer only uses one general `DatabaseInterface`, the different files for the different datatypes are accessed by separate connectors. A connector has to implement the `IConnector` interface, which mainly enforces the `get()` and `set()` methods to ensure a connector can perform as expected. The rest of the methods are implemented in the baseclass `Connector`, which should be extended. In every connector, the `get()` and `set()` methods are different. A `get()` method builds the querystring based on the given arguments, then retrieves the data and performs other necessary queries (like retrieving a set of fields of unspecified length) to ensure the PHP DataModel is complete. A `set()` method has to store the data accordingly.

3.3 PHPDoc

PHPDoc for the database classes can be found at SVN.

4 Future Work

4.1 Adhere to Open-Closed Principle

At the moment, *the DatabaseInterface.php* class violates the Open-Closed principle. Every new connector, four lines need to be added in this class. This is unwanted if new RDF datatypes are added. The same holds for the method `createArguments` in this class: extra cases have to be added if more types of arguments become possible. Adhering to the Open-Closed principle will allow for easier extension and maintenance of the database-related classes.

4.2 Deal with invalid or missing values

This goes in two parts. Firstly, the database now assumes that given an entry, all the fields for its type are there and filled in. This is unwanted since it can cause issues with backwards compatibility of certain .rdf files. Secondly, it assumes that fields containing a UID to a different entry actually point to a valid entry. For instance, a Survey query will try to get a User object as its creator value from the database as well, assuming it exists. It will break if this user object does not exist.

4.3 Optimize queries

Queries as they are at the moment can be a bottleneck in performance if the sets get really large. This is because of a couple of things. For one, there are duplicate queries being executed: if a Session object is retrieved, it also retrieves all the corresponding AnswerSets, which in turn retrieves the corresponding Questions. However, the Session also retrieves the Surveys related to it, which afterwards also retrieve the same questions. There has been no benchmarking of performance so I have no clue at what size of the .rdf files this will become an issue.

I have a gut feeling that the SPARQL queries can be optimized some as well. This is regarding entries with zero or more elements of a specific type, like Survey. Now, there is a second query for getting all the IDs for all the questions, but there must be a way to introduce this into the first query without getting the exponential growth of the RDF 'different tree options'.

A Class diagram of the models

