

Database Connection for the CPS Facilitator Tool

Jos Kraaijeveld

January 13, 2012

Contents

| | | |
|----------|--|----------|
| 1 | Changelog | 3 |
| 2 | Introduction | 3 |
| 3 | How to use | 3 |
| 3.1 | Initialization | 3 |
| 3.2 | Basic queries | 3 |
| 3.2.1 | Retrieving data | 3 |
| 3.2.2 | Storing data | 4 |
| 3.3 | Arguments | 5 |
| 4 | Specification | 7 |
| 4.1 | Models | 7 |
| 4.2 | Static functions, non-static functions | 7 |
| 4.3 | Adding to the system | 7 |
| 4.4 | PHPDoc | 7 |
| 5 | Future Work | 7 |
| 5.1 | Deal with invalid or missing values | 7 |
| A | Class diagram of the models | 8 |

1 Changelog

13-01-2012: Fixed the issue with Open-Closed, updated documentation accordingly.

2 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 (3). 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 4. The areas I suggest improving upon first are described in section 5.

3 How to use

3.1 Initialization

Before the database can be used, you should remember to import master.php on every page:

```
require 'classes/master.php';
```

3.2 Basic queries

3.2.1 Retrieving data

Getting data is done by calling the get method of the class for which you want to retrieve data. The specification for this method is the same for every class and is as follows.

| Function get() | |
|------------------------------|----------------|
| Second argument: \$arguments | Array |
| Return type: | Array/Variable |

The return type is an Array of objects with the type corresponding to the class you called it on. For instance, if you call User::get() you will only get User objects. Below are a few examples showing the usage of get(). A description of what arguments are possible per type is given in section 3.3.

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.

```
$questions = User::get(array());
foreach ($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 = User::get(array("name" => "Jos"));
//Assuming there is a result, the UID is:
$josUID = $userResults[0]->uid;
//Now to get the requested surveys:
$surveys = Survey::get(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);
    }
}
```

3.2.2 Storing data

Storing data is easy, as long as you stick to using the given PHP Classes. Retrieve these objects by using the `get()` function, and call the `save()` function of the objects.:

| Function <code>save()</code> | |
|---------------------------------|-----------|
| Returns: <code>\$boolean</code> | succeeded |

If the save function returns false, there is an invalid references in that object which needs to be resolved before saving. This ensures the database is always valid. 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 = Question::get(array());
$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, $question, array("12345", "four"));
//Save the answer in the database
if($answer->save())
    echo "Success";
```

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

```
//Get the survey
$surveyResults = Survey::get(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
$question->save(); $survey->save();
```

3.3 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 creationdate 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 |
| SessionInstance | uid title location facilitator starttime endtime notes session resultset | String String String String String String Array of Strings String String | UID of the user that facilitated the session Unix Timestamp. No way to search on intervals (yet). Unix Timestamp. No way to search on intervals (yet). UID of the Session this Instance refers to. UID of the ResultSet this Instance has. |
| 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.

4 Specification

4.1 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.

4.2 Static functions, non-static functions

It is important to note that the `get()` function of each object is static, whilst the `save()` function is not. This is to provide a better saving mechanism in the future: tracking when an object gets changed and saving it automatically. The `get()` function is static because there is no reason for it to be linked to a specific Object, only to the type it describes as a whole.

4.3 Adding to the system

Adding support for new datatypes is easy and can be done by extending the `ResearchToolObject` class and ensuring the `get()` and `save()` methods are implemented correctly. Simply look at the existing classes and stick to the same idea unless you want to radically change the infrastructure.

4.4 PHPDoc

PHPDoc for the database classes can be found at SVN.

5 Future Work

5.1 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 compatability of certain .rdf files. Secondly, it assumes that fields containing a UUID to a different entry actually point to a valid entry. For instance, a Survey evaluation will try to get a User object as its creator value from the database as well, assuming it exists. It will return false if this user object does not exist.

A Class diagram of the models

