OlympiData – Software Documentation

Contents

DB scheme structure

Our database has two major types of tables:

1) Tables that hold the data that was retrieved from DBPedia.
2) Tables that hold the information about the questions that were added to the trivia questions pull.

The tables that hold the data from DBPedia have information about the olympic games:

1) **OlympicGame(game_id, year, season, city, comment)**:
   Holds the year and season of each olympic game and if available also the city where it took place as well as the comment text taken from DBPedia.
   Each Olympic game also has a unique id, which is taken from DBPedia's WikiPageID attribute, which is a unique identifier for every page on Wikipedia. This is the primary key of the table.
   The (year, season) tuple is unique, since it represents a specific Olympic game.

2) **Athlete(athlete_id, dbp_label, birth_date, comment, image_url):**
   Holds the athlete label (which is his name, sometimes with additional info), as taken from DBPedia label attribute, their birth date, and if available also the comment and image_url, as taken from DBPedia.
   Each Athlete also has a unique id, which is taken from DBPedia's WikiPageID attribute. This is the primary key of the table.
   The dbp_label is unique, since it's taken from the DBPedia unique attribute 'label'.
   We only took athletes who participated in at least one Olympic game, according to DBPedia.

3) **OlympicSportField(field_id, field_name):**
   Holds the different types of unique sport field names that were in at least one Olympic game.
   Each sport field has a unique id which is an auto-increment field. This is the primary key for the table.
   The field_name attribute is unique.

4) **AthleteOlympicSportFields(athlete_id, field_id):**
   Holds information about the sport field of every athlete, in tuples of athlete_id and field_id (there may have been athletes who participated in more than one sport field at the Olympic games).
   The (athlete_id, field_id) is the primary key for this table.

5) **AthleteGames(athlete_id, game_id)**
   Holds information about the games each athlete participated in. The table consists of unique tuples of athlete id and game id.
   The (athlete_id, game_id) is the primary key for this table.

6) **CompetitionType(competition_id, competition_name)**
   Holds the different types of competition names that were in at least one Olympic game.

Each competition type has a unique id which is an auto-increment field. This is the primary key for the table.
The competition_name attribute is unique.

7) **AthleteMedals(athlete_id, game_id, competition_id, medal_color):**
Holds information about each athlete medals (gold, silver and bronze), and in which competition type of a specific Olympic game did they win them.
The information is stored by athlete id, game id, competition id and medal color, each such tuple is the primary key of the table.

The tables that holds the information about the questions are separated to a table per question type. Each question type table holds a tuple for each question that was added to the pool for the specific question type. The attributes for each table consist of one/two attribute that hold the information needed to build the question, and they are the unique identifiers for each table. Additionally, there are two attributes for user statistics for each question – how many users answered the question correctly (num_correct), and how many were wrong (num_wrong). The answers to the questions are not saved in the database - they are generated once the question is selected for the quiz, using appropriate queries. This is because the data in DBPedia may change, and that may affect the correct answer.

DB optimizations performed

We used several optimizations in our database:

Using foreign keys between table:

- The game_id is used to identify each game in the OlympicGame table. It is used as a foreign key in different tables – AthleteGames, AthleteMedals, Question_type1 and Question_type5. For that, the game_id column in those tables is indexed.
- The athlete_id is used to identify each athlete in the Athlete table. It is used as a foreign key in different tables – AthleteGames, AthleteMedals, AthleteOlympicSportFields and Question_type 2, 4, 6, and therefore this column is indexed in those tables.
- The field_id is used to identify each Olympic sport field in the OlympicSportField table. It is used as a foreign key in different tables – AthleteOlympicSportFields and Question_type3, and therefore this column is indexed in those tables.
- The competition_id is used to identify the competition type and is used as a foreign key in the AthleteMedals table, and therefore this column is indexed in the AthleteMedals table.

Additionally, the dbp_label of the Athlete table is also indexed, since we perform many queries that uses it.

We split the tables, so that we don't have unnecessary duplicate data. For example, we created the AthleteOlympicSportField table to save the athletes' sport fields, instead of putting it in the Athlete table, since each athlete may have several sport fields.

Code structure

Our code is divided into 3 major parts, and here we will elaborate on each one:

1) Front End directory – here we have all the code that is related to the front end of the web app. In the top directory we have the different html files for each of the pages in

our webapp – the main page, quiz page (which also uses a generic template html), add question page and update DB page.
Additionally, we have the css and javascript files in separate directories, one file for each module of the app, if needed (for example, the main page doesn't use js).
We also have an images directory for the images we use in the webapp.

2) Back End directory – here we have all the PHP files that are used in our website, and are called by the different parts of the app. There is also a general PHP file that is included by most other PHP files, and is used for creating the database connection and has general functions for the preparation and execution of SQL queries.
3) The admin directory – here we have a PHP script that is called by the front end. This PHP script runs a Python script, found at the same directory. The Python script collects the data from DBPedia, updates out DB, and prints its logs to stdout. The PHP script sends the output to the front end, which displays the Python run results to the user.

There is also a PHP file in the top level of the source directory which redirects to the main page.

Description of the API used

We used DBPedia as the API from which we retrieve the data for our web app.

In order to retrieve the data, we use a Python script, which connects to the DBPedia SPARQL server (http://dbpedia.org/sparql) and run queries to retrieve the data we need. After running each query, we parse the info when needed, and then put it in the appropriate tables in our MySQL DB.

Sometimes the connection to the DBPedia server is slow, and the query times out. In order to avoid data loss, we have a retry count of 10 for each query.

We built our MySQL database scheme with unique attributes and keys, so that when we insert the data from DBPedia to our tables, we won't have duplicate values. For example, the athlete_id is unique and is taken from DBPedia's WikiPage attribute, and we don't allow adding the same id twice to the athlete table. This also applies to the game id. Another example is the Olympic field, where we don't use a WikiPageId value, but use the field name as unique, in order to avoid duplication.

On the other hand, since DBPedia data may change or increment, our script enables updating or adding some of the values when running it in the future. For example the birth date, comment or image url of an athlete, or the city and comment of an Olympic game. And of course if new data is added to DBPedia, it will be added to our DB by this script.

In order to get the Olympic games' data, we query all objects of type Olympics from DBPedia, which are filtered by a certain regex. We get the host city, and comment, and from the label attribute we extract the year and season.

In order to get the Olympic athletes' data, we query for objects of type athlete in DBPedia, which also have an attribute of subject that fits to a certain regex, from which we can tell they participated in at least one Olympic game. We use a similar query afterwards to get all these athletes subjects (which match the regex) in order to update the athlete games and sport field, and also get the different sport fields into the OlympicSportField table.

In order to get the athlete medals and competitions, we query all the objects of type {gold, silver, bronze} Medalist, that match a certain regex, to get only those who won a medal at the Olympic games in a certain competition. With the info, we update the competitionsType and AthleteMedals tables. We use the wikiPageID attribute to link them to the Athlete table.

Finally, we have a part in the script that fills two questions for each question type table (if there are less than two for each table), from the data we now have in our database, so that the users can immediately start playing.

External packages/libraries used

- AngularJS

  Combined with html and css, we used AngularJS to build the front-end of this project. AngularJS helped us create the all functionality we needed in our website.

  All the logic of the buttons, drop down menus, pop up messages and other components, was created with AngularJS, as well as saving data in variables and transferring data to and from the back-end.

  We used the various components of AngularJS, like controllers and directives, its services like $http, and mechanisms like the promise mechanism.

- MySQLdb Python library
  We used MySQLdb Python library for updating the DB in the data retrieval Python script.

- SPARQLWrapper Python library
  We used SPARQLWrapper Python library in the data retrieval script in order connect to http://dbpedia.org/sparql, run the sparql queries and get the results in JSON format.

General flow of the application

We divided our application several parts, considering the different pages and functionalities we needed in our project and website.

The first unit is the welcome page, from which the user can navigate to the two sections of our website - answering the quiz or creating a new question.

The second unit is the quiz section in the website. We get the questions from our back-end using the $http get method. It calls a PHP script that generates 10 random trivia questions from the existing questions in the DB, and generates the correct and wrong answers, as well as more info and images (if available) for each question, using appropriate queries. All the data is passed to the front end in JSON format.

On the front end we use a directive component inside an AngularJS app in order to present the questions to the users. We use a factory mechanism to produce the various questions in every quiz.

We send back the statistics about the user's performance in the quiz (which question were correct/wrong) using the $http post method. It sends the needed data in JSON format to a PHP script that updates the question tables for the specific questions that the user was asked. We used the promise mechanism to synchronize between the front-end and back-end functionality.

The third unit is the ask-question section of our website, where the users can create questions from various formats we provide them. In this part we used a controller inside an AngularJS app. The controller contains several functions which synchronize between the data that comes from our DB through our back-end, and the data that is being displayed in the drop down menus of our website. Similar to the quiz section, here we also use the $http get and post services to call the appropriate PHP scripts to get and send information and use the promise mechanism of AngularJS to make this synchronization. We first get the list of question formats. Once the user picked one, another PHP script is invoked to get the list of the available option for the first argument (which are not used yet). We use a similar procedure for the second argument (if exists). Once the question is ready, the question type and its arguments are being sent to a PHP script in JSON format, and are updated in the appropriate question table.

The forth part is the update page, from which the DB can be updated with the latest data from DBPedia. In this part we used a controller inside an AngularJS app. We used the post function of the $http service in AngularJS to call a PHP script, which first check if the Python update script runs on the server, and if not, it runs it. The PHP script sends the Python script's result back to the front end once it's done running. We used the promise mechanism to synchronize the termination of the update action and the messages presented to the user when it terminates.

General comments:

- When we run the SQL queries in the PHP and Python scripts, we use prepared statements in order to prevent the reparsing of SQL statements.
- In any place where there is an error in the back end/running SQL queries, the PHP script will return a http return code, and an error message will be shown.