Specification for Development Contract

Ian W. Scott

Start date: July 7th, 2014
End date: July 18th, 2014
Total paid hours alotted: 24

Overview

This contract is to produce five data visualizations embedded in a web2py view, along with the back-end python functions to serve the data. The developer may choose a charting library to use, though d3.js is already included in the application and so would be simplest to implement.

The web2py application for which the work is required is called Paideia. It is an open-source language-learning game engine currently being implemented to teach ancient Greek at http://ianwscott.webfactional.com/paideia. The game has some rpg elements as students move around a fictional town and learn the language through conversations with the town's inhabitants.

note: All file paths below are relative to the paideia application directory ('web2py/applications/paideia/')

Application jargon

It will help to understand that in the application one question-answer user interaction is called a "step." These "steps" are grouped into sequences called "paths" which resemble short "quests" in a typical rpg.

Users work through a series of "badges" divided into "badge sets" which function much like levels in a typical game. Behind the scenes, these badges correspond to "tags" which are assigned to various "steps".

Data source

The required data is stored in a sqlite database and accessible through the abstracted model described in models/paideia.py. The relevant tables are

- db.tags: The tags which categorize "steps" for evaluation purposes.
- db.badges: The "badges" which have a 1:1 relationship with tags and represent those tags to the user. Users never see tag names or ids but rather see the names of the "badges" associated with the tags. (Yes, I know these tables should really be combined. It's a historical anomaly.)
- db.steps: This table includes list:reference fields providing a many:many relationship between steps and db.tags.
- db.attempt_log: The raw record of student performance. Each row represents the outcome of a single user attempt.
- db.badges_begun: A record of when users attain each new achievement level (cat1, cat2, etc.) with a given tag. (Although the table name refers to badges the data is linked to db.tags.)

Data visualizations

The charts should be added to the view in views/default/info.load. They should be added as new tabs in the tabbed interface of that view. The charts may each be assigned their own tab or they may be grouped in tabs for related data. This view provides users and administrators with information on the performance and progress of one user.

(Note: although d3.js is present in the application in plugins/plugin_d3, the files must still be exposed to web2py in the relevant controller function using the response.files.append() syntax.)

The required visualizations are:

vis1: User's highest badge set over time

This should be a line (or bar?) graph of the highest badge set reached each day since the user has been active. At present this information must be calculated dynamically from the 'tag' and 'cat1' fields of db.badges_begun and the 'tag_position' field of db.tags. db.tags provides information on the set to which each tag belongs ('tag_position') and db.badges_begun provides the date on which each tag became active ('cat1'). The visualization will group tags by 'tag_position' and then track which of these groups had active 'cat1' status on a given day.

vis2: Number of right and wrong answers per day

This should be a stacked area graph showing the number of right and wrong answers given by the student each day since s/he has been active. This data can be drawn from db.attempt_log.score and db.attempt_log.dt_attempted. Any score of 1.0 or more should be considered "right" and any score of less than 1.0 should be considered wrong.

vis3: performance on any (user-selectable) specific "step" over time

This should be like vis2 but restricted to a single "step". The step for each attempt in db.attempt_log can be identified using the db.attempt_log.step field. The visualization should include a widget to select any "step" the user has tried (i.e., any step with a db.attempt_log row for this user).

vis4: performance on any (user-selectable) particular "badge" over time

This should be like vis3 but restricted to a single "tag." The tag name and id should not be part of the visualization. Instead it should be represented by the name of the badge (db.badges.badge_label) linked to it by the db.badges.tag reference field. The performance displayed should be the aggregate total of all attempts for all steps tagged with the selected tag/badge.

vis5: performance on the user's ten worst "steps" over time

This should be either a stacked area chart or overlapping line graphs. Each area/line should represent the ratio of wrong answers/right answers for one "step." The worst ten steps should be chosen based on this ratio (i.e., the 10 steps with the lowest right/wrong ratio). Here I would like the time period to be user-selectable. The "worst" steps should ideally be re-evaluated based on the time period selected.

Organization of the html and js

The info.load view is a web2py template file. It is a plain html document with python code embedded between double curly braces {{ }}. Where python variables are to be inserted into the final html, this is expressed by an equals sign followed by the variable or expression whose value should be included. E.g., {{=myvalue}}

Python code in the view has automatic access to any values from the dictionary returned by the controller. Instead of using dict notation, these values are treated as if the dict keys were variable names.

One other idiosyncrasy of web2py views is their handline of indentation in embedded python. Since it is difficult to maintain consistent indentation, whitespace is NOT meaningful within the embedded python code. As a result conditions and other indented blocks must be "closed" by the keyword {{pass}}.

The necessary js may be included at the bottom of the info.load view or may be added in a separate file under static/js. In the latter case the js file will need to be exposed to web2py in the controller function (see below) using the response.files.append() API.

Data analysis

The necessary data is supplied to the view by the controller function info() in the controller file controllers/default.py. This controller simply calls on business logic from the module modules/paideia_stats.py.

The existing Stats class is quite inefficient and some of its code is now deprecated. My suggestion is that you begin a fresh class in that same file with the business logic for these visualizations. Later on I can integrate the two classes.

Speed and efficiency in the data analysis is a major concern, since this data will need to be calculated every time a user or admin views the profile. With this in mind I created the table db.user_stats which is meant to aggregate each week's data for a given user in one row. The idea was for the Stats class to store computed data for later recall. I have not gone far, though, in implementing this, so you may use that table, adapt it, or leave it alone as you choose.