CS 411: Team Globex Final Project

[dalbrech|arawlin2|pcurry2]@uiuc.edu

December 6, 2007

1 Functionality

In this section, we discuss the technology and features of our application.

1.1 Technologies used

Our group originally planned to use Linux, Apache, MySQL, and Mason [5] to develop our site.

Before choosing a platform, we realized that web applications embody many seperate concerns: application logic, display, database access, etc. which, for ease of development and design, are best dealt with seperately. It became clear that using a scripting language directly made seperating theses concerns much more difficult, so we opted to use a framework to construct our application. After examining several frameworks (Django [4], Ruby on Rails [6], etc.) we chose Catalyst [2], a Perl-based framework which uses the MVC pattern (Model, View, Controller) to organize a site's concerns. Although full discussion of Catalyst's benefits and features is beyond the scope of this document, Catalyst's ability to leverage existing Perl code (since it is nothing more than a Perl module itself) makes it an outstanding candidate for rapid application development. We used Perl's Session::State for session management, DBIx::Simple for SQL access, and HTML:Mason for templating.

The finished product used MySQL as its storage engine. We didn't end up using Apache because Catalyst has its own Perl-based HTTP server. Catalyst's server provides rich event logging for debugging, but Apache is the preferred choice for production-scale deployment.

1.2 Application Functionality

Our finished product is a web-based application for managing a single-elimination tournament. The final product is accessible at http://dra.gotdns.org/, a Linux server David maintains in his apartment.

User authentication and session management is our site's first major functional area. When a user first connects to our site, he is greeted with a friendly login prompt which requests the user's email and password. If the site user doesn't yet have an account, he can click on the "create new account" link at the bottom of the site. Site users are classified into two roles: players, who participate in our tournament, and judges, who adjudicate and score the tournament. These two roles differ in their functionality; we discuss the player role first.

When a player logs in, he is greeted with a home page which summarizes the subset of site data relevant to the particular user. The teams listing, at the bottom of the page, summarizes the player's team membership. If the user wishes to join or leave a team, he can do so on the Teams page. Using the Teams interface, a user can also create a new team, of which he is considered the "captain", the team's *de facto* owner. Since the captain owns the team, he alone has the power to delete the team. If ever a team is deleted, its players are all removed from membership on the deleted team. Moving back to the home page, a user's teams will be involved in several matches. The home page shows the matches for the player's current teams, and shows the status of completed matches.

When a judge logs in, he is greeted with a different page which summarizes his current open matches. The judge adjudicates the match in favor the winning team by clicking on the victorious team's name on his home page. When adjudication occurs, new matches are created based on the results of the judge's action, and the players will have access to their new pairings as soon as they reload their pages.

A user logs out of the site using the Logout link provided at the bottom of the page.

1.3 Advanced Features

1.3.1 Security

SQL injection and cross-site scripting (XSS) are the biggest threats to database-backed web applications. Using careful input validation and defense in depth [1], we believe we have made our site safe from both forms of attack.

Our first layer of defense arises from the way our application communicates with the database. As described in [1.1], Catalyst allows use of any Perl SQL module for database communication. Although more sophisticated backends (e.g. DBIx::Class) were available, we used DBIx::Simple because of its ability to issue SQL commands directly to a database server. (DBIx::Class, among others, implement turnkey Object-Relational Mapping functionality which abstracts away all primary/foreign keys, joins, etc., which was not permitted in the project.) To issue a query in DBIx::Simple, the programmer must first compile the query without arguments, and then insert (possibly user-supplied) values into the query after it is compiled. By the way it operates, DBIx::Simple makes SQL injection very difficult because user input has no effect on the query plan whatsoever. (Compare to other frameworks such as PHP, where the dominant mode of query construction is still string concatenation.) Additionally, DBIx::Simple is aware of database semantics and automatically escapes all control characters; doing so is easy because of the rigid seperation between the query's structure and arguments.

Our application's second layer of defense is per-parameter validation. When the user submits a form, the application's controller passes the submitted data to the model, which ensures that each and every parameter matches a predefined regular expression. As a rule, we have based all of our application's security decisions on access, rather than exclusion, meaning that errors are far more likely to incorrectly deny access rather than allow it [1].

Finally, our application's MySQL user credentials specify that all database accesses take place under the identity cs411_user. cs411_user is a non-root entity whose rights are restricted to Insert, Update, Delete, and Select on the tables in our database. Compromise of cs411_user would at worst breach the security of our application, not the database host itself.

We believe that our site is likewise immune to XSS attacks. Although DBIx::Simple does not automatically immunize the site against XSS, we believe our parameter validation to be sufficiently strong to reject any malicious tags (e.g. <script>) which form the basis of XSS attacks.

While we are not experts in SQL injection techniques we were able to attempt some basic attacks, which would be made even easier by our knowledge of all the table names in the database. By researching various example of SQL injection attacks, we were able to construct some user entries that may cause issues. One such technique is to make a condition what will always evaluate to true. For example, an entry like ' or 1=1 would evaluate to true, and retrieve all rows in the table. In a naively designed login page, this could result in anyone being able to login. However, with our security based on access and DBIx properly escaping characters, there is a measure of protection against SQL injection.

1.3.2 Role-based interface

As described in [1.2], our users are either Judges or Players. The application presents different views and privileges to judges and players on a "least-privilege" basis, i.e., it gives each class of user the minimum set of rights necessary to perform their duties.

1.3.3 Advanced SQL

Most of the SQL queries in our application relatively simple, but a few go beyond select-from-one table. In particular, when selecting or inserting from many-to-many relationships a JOIN may be required. One example is retrieving games that a player participates in. In the function get_games,

```
SELECT gid, gtime, place, outcome, win-
ner, team_one, team_two, parent
FROM Users
NATURAL JOIN PlayerTeam JOIN Game
ON PlayerTeam.tid=Game.team_two;
```

1.3.4 Round-pairing algorithm

Our application manages single-elimination tournaments. A tree-like "bracket" defines these tournaments, so the first step in our round pairing algorithm generates the overall structure of the bracket. The algorithm must account for the number of teams (which usually isn't a power of two, so it awards byes) and make provisions for various kind of fairness.

After bracket generation, paired teams participate in matches. For each match, one team advances; the database reflects this by updating a team's next round (we call this relationship "parent", in the manner of a binary tree), indicating that the winning team from one side of the tree will face the winning team from the other side of the tree.

We feel this feature deserves consideration as an "additional cool feature" due to the difficulties of storing a binary tree in a relational database. The way in which we read and updated the tree throughout its lifetime required intelligent algorithms for tree serialization and deserialization.

1.4 Request/Response Example

Suppose a logged-in player attempts to create a new team "hax0rs". The following series of steps would take place to fulfill this request:

1. The player's browser issues an HTTP GET for /teams/create, which contains the form for creating a new user.

2. After completing the form, the player uses an HTTP POST request to /teams/docreate.

3. Catalyst's URI dispatcher sees the incoming POST operation. The dispatcher creates a new instance of the TEAMS controller, and invokes its DOCREATE() method.

4. DOCREATE() ensures the user is logged in; if he is not, the user is redirected to /auth/login, which renders the login prompt. If the user is logged in, DOCREATE() proceeds.

5. The controller creates a new instance of the model class TEAM (distinct from the Team controller). The controller populates the new TEAM's name parameter from the form, and its captain parameter (the current user's id) from the session (set at login).

6. The controller calls TEAM->COMMIT(), which attempts to persists the newly created TEAM object into the database.

7. TEAM->COMMIT() establishes a connection to the database using DBIX::SIMPLE's CONNECT() method.

8. TEAM->COMMIT() decides if the object is already in the database by checking the object's IN_DB flag.

   (a) If the object is already in the database, TEAM->COMMIT() issues the SQL call UPDATE Team SET name=?, capt=? WHERE tid=?, populating the call's arguments using the object's member data.

   (b) If the object is not in the database, TEAM->COMMIT() issues the SQL call INSERT INTO Team (name, capt) VALUES(?, ?). Next, TEAM->COMMIT() issues INSERT INTO PlayerTeam (tid,usid) VALUES(?,?) to add the captain (the current user) as the team's only member. Finally, TEAM->COMMIT() issues SELECT last_insert_id(), a special call which returns the auto-incremented value the database assigned to the last inserted object (read back to keep the object's copy of its ID in the database current).

9. TEAM->COMMIT() returns to the controller. The controller redirects the browser to /teams, which presents the teams page including the newly-created team to the user.

2 Project Management Concerns

2.1 Plan vs. Reality

We deviated from our plan in two significant ways:

1. We changed our target platform after submitting our "final decision" in stage3. [1.1] explains our final choice of platform in detail.

2. We cut out some functionality we had planned to offer due to time constraints. Most notably, the finished product makes no provision for multiple tournaments (it can only handle one tournament at a time), and does not include very sophisticated reporting.

Apart from the two deviations noted above, our application's implementation did not differ appreciably from our plan.

2.2 Division of Labor

1. David: System administrator and overall project architect

   (a) Researched platform choices and decided on application platform for project

   (b) Installed Catalyst framework and modules on privately-owned server

   (c) Wrote most controller modules

   (d) Designed site's interaction pattern and templates (view)

2. Andrew: Database/model developer

   (a) Developed all model classes

   (b) Specified and implemented project's SQL

   (c) Developed and tested regular expressions for field verification

3. Paul

   (a) Tournament pairing and seeing algorithms.

2.3 Technical Challenge

The biggest challenge of this project involved learning a new framework (Catalyst). Once past the initial learning curve, our technical challanges were relatively minor. We encountered one "interesting quirk" in Mason.

To see his Teams page, the user issues a HTTP GET on /teams. The Catalyst dispatcher creates an instance of the Teams controller and passes control to its index method. The index method creates several model classes which query the database and in turn determine the user's affiliated teams.

A user is generally affiliated with multiple teams; as such, we chose an array to convey the user's affiliations from the controller to the Mason template (the model). For some reason, iterating over the set of teams with a FOREACH loop consistently produced a list of only one team. After some study, we determined that one cannot pass arrays into Mason directly; the accepted practice is to pass a scalar containing a reference into the array in place of the array itself [3]. Using a scalar reference is slightly more efficient because the data structure is not recopied (compare to passing by value vs. reference in C), but it was not immediately obvious that such behavior would not produce the desired result.

3 References

1. Bishop, Matt. *Computer Security: Art and Science*. Addison-Wesley Professional, 2003.
2. Catalyst Home Page. http://catalyst.perl.org/
3. Catalyst::View::Mason. http://www.cpan.org/modules/by-module/LWP/FLORA/Catalyst-View-Mason-0.15.readme
4. Django. http://www.djangoproject.com/
5. Mason. http://www.masonhq.com/
6. Ruby on Rails. http://www.rubyonrails.org/