Game Contest Server

Alex Sjoberg, Phil Brocker, Justin Southworth, David Nicholls, some other dudes sometimes
Department of Computer Science & Engineering, Taylor University, Upland, Indiana 46989

Introduction

The Computer Science and Engineering Game Contest Server is an ongoing department project started in the fall of 2013. The goal of the project is to create a web site that allows professors to host contests for various games such as checkers, chess, and battleships. Students will then be able to develop and submit automated players for these contests so that the professor can play them against each other and determine who can program the best player. The Game Contest Server was written using Ruby on Rails and has the flexibility to run a variety games written in multiple languages.

The original idea for the project came from Dr. Geisler and was begun in the fall of 2013 by students in his Multi-tier Web Application Development class. Basic site design and interaction with the necessary database tables, along with a suite of tests, were completed by the end of the fall.

January Goals

Our team picked up the project in January of 2014. Our overall goals were to continue improving the front end UI and begin developing a working back-end. More specifically, our goals were to:

1. Develop the back end so that we would be able to run matches and gather the results.
2. Add an administration interface.
3. Improve the general look and flow of the site.
4. Add pagination to index pages.
5. Implement search functionality for the major database tables.
6. Continue to expand the test suite.

We decided to implement the back-end using Ruby to maximize compatibility with the front-end. We considered using C++ for efficiency and better access to low-level system calls, but decided Ruby development would be faster. Parts of the team developed using Nitrous, a web app for quickly setting up development environments. We also set up a virtual machine to serve as a more stable long-term server platform. We developed the system in Linux and used Github for version control. We also used Trello for keeping track of goals and tasks and who was working on what.

Back-end Development

We decided to start small by getting a single match between two players running properly and reporting results. This was the lowest level of interaction with the players and was critical for defining what players and referees needed to be able to do to interact with our system. Single matches would later be used as the building blocks for running more complicated tournaments.

We decided to have the players and referees communicate both with our server and with each other using sockets. This allowed for games to be written in any language and kept the game as independent from the server as possible. It also left open the possibility of games being played over a network.

Running Tournaments

In the original design, there was no flexibility within a contest to exclude certain players or run multiple tournaments without uploading a whole new set of players. We thus added a new tournaments table to the database as well as front-end support for creating tournaments within a contest and selecting the participants

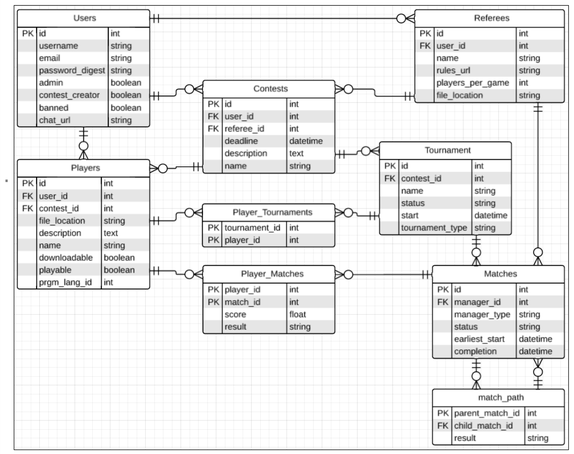

Figure 1: Entity relationship diagram

Once a tournament is created, a daemon running on the server will pick it up and run a script that creates the necessary match entries in the database for that tournament. These matches will be picked up by another script that runs the matches using the associated players and referees.

Adding single elimination

Initially, our only tournament-type option was round robin, which required a simple routine to organize matches between each player and every other player. Single elimination, however, introduced dependency between matches, with the winner of one match moving on to the next, all the way to the final. There needed to be some way to represent and store each such dependency.

After considering various solutions involving the database and daemons, we decided to construct a new many-to-many table between matches and itself. This table had fields representing a parent match, a child match, and a condition (usually “Win”) on which a player would move from one to the other. While the concept seemed strange at first, it made more and more sense during implementation, and eventually proved successful.

A Python library for checkers

Most of our initial tournament work was done using a mock game written in Ruby, which essentially handed out a win to the first player to guess “w”. Using the same setup to run a much more complex game written in a different language altogether seemed daunting at first. In the end, however, the process was fairly straightforward.

First, we learned how to use sockets in Python by creating another version of our “Guess W” game, finding some important differences from Ruby during debugging. Next, we analyzed the communication between the checkers referee and player, discovering that their only point of communication was a call to a single function. Ultimately, all it took to get them working was some importing and finding a way to send the function arguments and return values over the network.

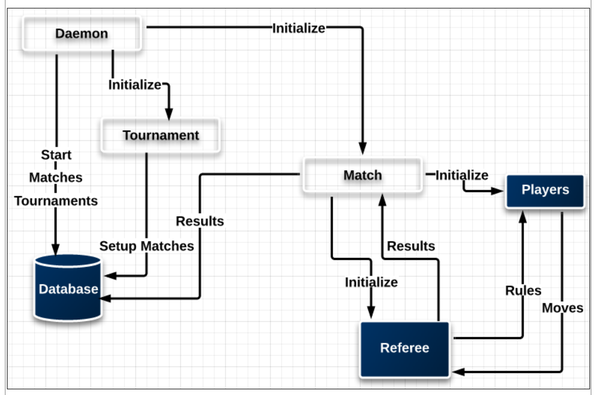

Figure 2: Back-end program flow

Front-end Improvements

Our first task for the front-end was improving the aesthetic appeal. We utilized Bootstrap (a front-end web development framework) in order to quickly implement the visual features that are presented on our web pages.

Another major task that we had to complete was implementing pagination. Pagination is concept of specifying how many items in a list are displayed on each page. Implementing the code for this feature was a fast process because many examples of how this process works could be found on the Internet. We implemented this stipulation on the referees, players, users, tournaments and contests pages.

Another section that we have produced for the front-end was links to the edit pages on the previously mentioned sections of the website. Our main focus for these links was ensuring that they only showed up for people who created the corresponding item so that it cannot be edited by someone who did not create it.

Search Functionality

We also implemented a search functionality, which was added to the users, referees, players contests, and tournaments pages. The implementation allows the users to find content depending on which page the search box is used. This functionality is fundamental to the website because as web-developers, it should not only be efficient for users to navigate through the website but also it should be located in a specific visible place so it can be easy used by users.

Most well-designed websites make the users take the least amount of clicks to navigate through the website so the they don’t get frustrated. The search functionality in the Game Contest Server allows partial word searching meaning that it is able find words even though only a few letters have been entered.

Error handling in this functionality is important because as web developers feedback is one of the most common and best ways to communicate with the users that either something went wrong and they have to fix it or everything is running smooth.

Testing for this functionality was long but good because I had to make search tests for every possible entry the user will think on inserting in the text field. Tests that were made for this functionality consist of the following:

- Partial search (displaying all words with the same letter inserted in the text field)
- Full word (results should display only the characters entered)
- Error handling (results when a symbol is inserted)

Testing

Test-driven development is a development technique where tests are written to describe what functionality should exist in an app before the functionality is added to the app. The tests initially fail, but as features are added, progress is determined by the amount of tests passing. Once all tests pass, we know that our code is sufficient.

Dr. Geisler used test-driven development in the all of 2013, and we chose to continue doing so. The difference is that now we were writing the tests rather than just having to pass the tests he had previously written.

We wrote tests using RSpec, which is a testing tool written specifically for Ruby. Though we did not have much experience writing tests for code and there was a learning curve to writing RSpec tests, the end result was worth the effort. While working in a group, it is very easy to mess up someone else’s functionality without knowing it. RSpec solved many problems by catching our errors and informing us when we had forgotten to update our database. It saved us time and frustration by making it hard to step on another group member’s toes while coding.

Tests were also written for use with the Selenium Webdriver, so that Javascript functionality could be checked. Unfortunately, because of database incompatibilities and time freezing requirements for other tests, these tests are unusable until fixes are made to make Selenium usable with our system.

Acknowledgements

- Dr. Jonathan Geisler for the original idea and requirements for the project and for his help in understanding the mysterious inner workings of Rails.
- Dr. Stefan Brandle for his endless knowledge of Unix and help understanding socket programming, as well as providing his Battleships code
- Nate White and Nathan Lickey for helping us set up our VM
- Dr. Art White for providing his checkers code