Super Bowl Internet Database

Inglorious Bashers

Trent Kan, Sam Tipton, Sophia Hernandez, Richard Salazar, John Schindler, Yoan Chinique

Introduction

What is the Super Bowl?

The Super Bowl is the annual championship game of the National Football League (NFL) and takes place on the first Sunday in February of every year. The game features the two best teams from each conference, the American Football Conference and the National Football Conference, who square off to earn the title of “World Champions” while hoisting the famed Lombardi Trophy. The event is watched by millions of Americans each year and has become a *de facto* holiday in the United States. It has become so popular that many of the world’s most successful and well-known companies pay millions of dollars for a thirty second advertising segment during the game.

The Problem

Throughout the web there are plenty of sites which offer information and statistical analysis of Super Bowl history. This includes information about the Super Bowl, information about the team that competed in that super bowl and information about the Most Valuable Player (MVP) of that game. However, these sites fail to provide a cohesive set of data and a positive user experience. For example, many sites will provide the competing teams of each Super Bowl but fail to list the Most Valuable Player of the game. Furthermore, many of these sites have advertisements that distract from the main content provided by the website as well as track visitor's online activity without permission. In addition, some of the data provided by these sites are hidden behind pay walls that force a user to pay for a subscription to the website.

Our Solution

The goal of the Super Bowl Internet Database (SBIDB) is to present our data on Super Bowls in an elegant and accessible manner. The SBIDB features a complete history of all forty-eight Super Bowls with information about the competing teams and MVP. The SBIDB also provides its users with fan generated twitter content focused on the Super Bowl, competing teams, and players. In addition, the SBIDB provides a means of social interaction with NFL players and teams by linking their respective Facebook pages. We also sought to provide YouTube highlight videos of each Super Bowl, team, and player. Lastly, we provide our data via a RESTful API to any external party under no constraints or conditions.

Architecture

The SBIDB is designed as a server-side Model-View-Controller (MVC) application that serves HTML5 pages. We use a Object-Relational mapper (ORM) to model and bind our database to objects automatically. Additionally, we provide a REST API for our data which can be consumed by other entities (including us). By using AJAX we can integrate our content with other platforms such as Twitter, Facebook, YouTube, and Google Maps.

UI

The user will arrive at a splash page and will be given the opportunity navigate to one of the three main categories (Super Bowls, Teams, Players) via the navigation bar or the buttons located on the bottom of the page. The Super Bowl page will present the user a list of all the different Super Bowls to choose from along with

links to the two teams involved. If the user chooses a Super Bowl they will be taken to the Super Bowl game page with the score, a photo and link to the MVP, highlights, photos and links to the teams involved, and social media elements. If the user chooses a team page, they will be presented with team logo, social media elements, number of Super Bowl victories, history of MVPS, and other team information. Lastly, if a user chooses the MVP from either the Super Bowl game page or the team page, they will be taken to that player's page consisting of a photo of the player, personal information, his team and Super Bowl history, video highlights, and social media concerning the player.

Tools

A wide variety of tools have been utilized in creating this web application due to complexity of this project.

Front End

At its core the SBIDB is written in HTML, while utilizing Twitter Bootstrap for CSS styling and JavaScript modules. This gives the unified look and feel of each page in the SBIDB.

Back End

The SBIDB is served by Heroku, a cloud based web application host recommended to us for its free small scale offerings, which suited our particular needs perfectly.

For persistent data storage we are using the embedded SQLite database offered within the Django framework. In the future, we will migrate to a PostgreSQL

database for better scalability, and ease of maintenance.

Framework

The Django framework is a good choice for our architecture because it provides a server-side MVC framework with a built-in ORM. Another advantage of using Django is that allows us to easily deploy and update our application on the Heroku application platform.

Git Workflow

The primary repository for the code is kept at https://github.com/samtipton/cs373-idb. The master branch contains the latest, stable state of the codebase with milestones idb1, idb2 and idb3 marking the release versions of the codebase. Experimental features and changes are stored in other branches until they are ready to be merged into the master branch. Merging into the master branch is done using a Github pull request since it allows us to code review the changes and make adjustments.

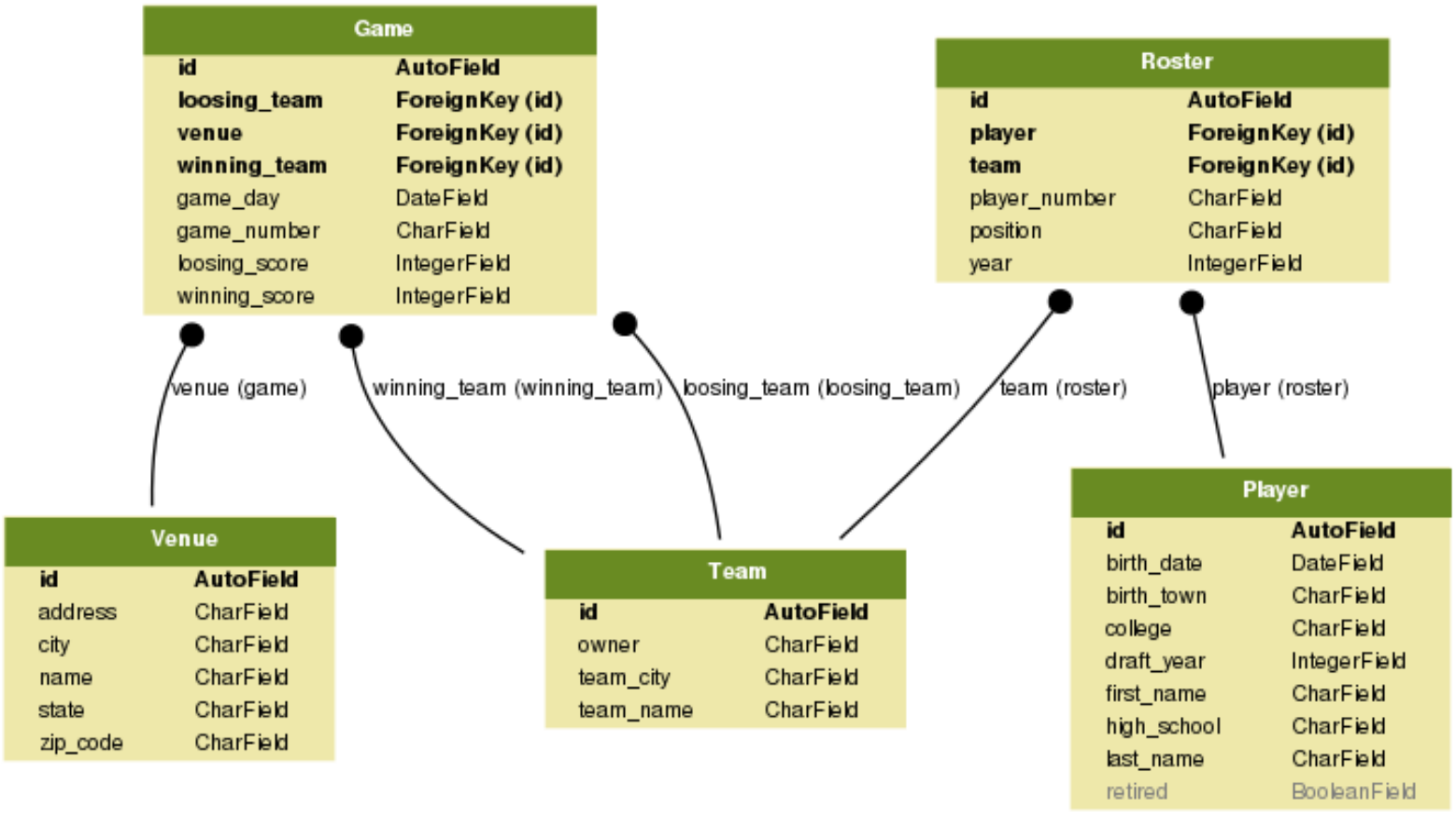
Models

Our models represent the underlying SQL database in a way that we can programmatically access with Python classes. This capability is provided for us by the Django framework through it’s ORM magic, allowing us to concentrate on the manipulation of data and the relationships between data.
The models we chose for this iteration of SBIDB are: Game, Team, Player, Venue and Roster.

Relationships

We chose each property of the models to fundamentally describe each model while offering reasonable relationships between them. Each instance of the Game model uniquely represents a Super Bowl game in the history of the NFL by encapsulating the pertinent relative data. A Game instance describes a one-to-one relationship between itself and two distinct Team instances, and a one-to-one relationship to a Venue Instance. The Roster model connects Player instances to Team instances along with providing additional information about the player. This relationship allows us to build a roster for a team for any given year. This also helps to provide a transitive relationship between the MVP and the Super Bowl, while reducing duplication of information within the models. Moreover, by abstracting the Venue as a separate model from the Game model, we are provided the ability to maintain referential integrity between the stadiums and the games in which they were played.

Relationship Diagram


Game
id AutoField
loosing_team ForeignKey (id)
venue ForeignKey (id)
winning_team ForeignKey (id)
game_day DateField
game_number CharField
loosing_score IntegerField
winning_score IntegerField
Roster
id AutoField
player ForeignKey (id)
team ForeignKey (id)
player_number CharField
position CharField
year IntegerField
venue (game)
winning_team (winning_team)
loosing_team (loosing_team)
team (roster)
player (roster)
Venue
id AutoField
address CharField
city CharField
name CharField
state CharField
zip_code CharField
Team
id AutoField
owner CharField
team_city CharField
team_name CharField
Player
id AutoField
birth_date DateField
birth_town CharField
college CharField
draft_year IntegerField
first_name CharField
high_school CharField
last_name CharField
retired BooleanField

API

Representational State Transfer (REST) is an architectural pattern for exposing web resources as an Application Programming Interface (API) using the HyperText Transfer Protocol (HTTP).

Implementation Patterns

SBIDB has three entities Games, Teams and Players. We provide five types of API operations on these entities using the four HTTP verbs: GET, POST, PUT and DELETE.

For each of the three entities, we provide:

- A GET call to obtain a list of all data for each entry within a single entity.
- A POST call to create a new entry in a single entity.

For each entity, we provide separate calls to access specific entries:

- A GET call to obtain all data for a single entry.
- A PUT call to update data for a single entry.
- A DELETE call to remove a single entry from the parent entity list.

Motivations

The intent is to provide this API such that others can easily and freely access and update the dataset. Using REST allows us to expose our API to a variety of languages and platforms in addition to allowing our web front-end to use it. Additional benefits of building a REST interface over HTTP, we gain the advantage of using existing tools and technologies to easily scale the API to multiple consumers.

Documentation

The documentation for our API can be found at: http://docs.idb.apiary.io/

Testing

We developed a set of unit tests to test each API call from our REST API. We utilized the Django's django.test module to execute our unit tests.

Testing the GET call

For our GET requests we obtained a response from Apiary using Python's urllib library function urlopen which returned an HTTP response object. We checked the response code with the Python function assertEqual to make sure that the response code matched to what we expected. For the GET call, we expected a HTTP response code of 200. We then called the Python readall function to read the body of the HTTP response object. Next we called the decode function to produce a utf-8 character string representation of the response body. We called the loads function from the JSON library on this string to produce a JSON object that we compared to our expected output.

Testing the DELETE call

For the DELETE requests we created a request object using the from urllib module. We then called the get_method on the request object and assigned it the DELETE call. Next we used the urlopen function with our request object and obtained a response. We then checked the response code to ensure it was 204. Our DELETE call does not return any other content to test.

Testing the POST call

We used the dumps function from the JSON library to create a dummy JSON formatted string. By dummy, we mean that we made up some data that would simulate a POST call. This was necessary to encode the string as a utf-8 string and create a request with our data and the header "Content-Type": "application/json". We used this request to open the URL (using the urlopen function) and made a POST call. Once the call was made, we verified that the HTTP response code was 201 and proceeded to test the content of the POST call. We tested the content in a similar manner to how we tested the GET call's content. We read the HTTP response object using the readall function and decoded the object into a utf-8 character string. We took that string and created a JSON object using the loads function. To ensure a correct output, we compared the JSON object to our expected output, a dictionary with 'id' as the key and the integer 1 as the value.

Testing the PUT call

We tested the PUT call in a similar manner to how we tested the POST call since they have a very similar behavior. Once again we used the dumps function to create a dummy JSON formatted string and encoded this string as a utf-8 string. We used this encoded string as our data along with the header "Content-Type" : "application/json" to create a request. We then called the get_method function on the request object and assigned it the PUT call. This was similar to how we used get_method in the DELETE call tests because both calls return no content. We used the request to open the URL (using the urlopen function) and made a PUT

call. As stated earlier, the PUT call returns no content, so we just checked that the HTTP status code matched the expected status code, which was 204.

Testing failure cases

We included additional test cases that were set up to deliberately fail. We did this to ensure that our tests were correct. For example, we changed some our content to make sure that our tests were comparing data and not type of the content. This would ensure us that our API calls were performing and responding correctly.