# Super Bowl Internet Database

Inglorious Bashers

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

## Table of Contents

# I. 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 eleven 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,twitter feeds. 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 utilizes the Django framework to serve 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, Franchises, MVPs) 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 with links to the two teams involved and the MVP. 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, an embedded google map of the stadium the game was played in, a highlights video, 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, photos of the Owner, General Manager, and Head Coach, links to former 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, a link to the team he received the MVP with and links to past Super Bowl MVP titles, video highlights, and social media concerning the player.

## II. 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 utilizing Twitter Bootstrap for CSS styling and JavaScript modules. This gives the unified look and feel of each page in the SBIDB. Every page besides the contacts page is an extension of a root template HTML5 file.

The root template (template.html) includes the boilerplate <!DOCTYPE html>, <html>, <head>, and <body> HTML tags within which every child template (*-template.html) page extends to include their specific content. The root template includes links to our style and script files in the <head> element and the html for the navigation bar (seen at the top of every page) in the body. After the navigation bar begins the django template syntax for block content ({{% block content %}}). This is where html from the child template htmls is inserted. Lastly, every page of SBIDB includes a footer <div> element with group member names.

The children templates (*-template.html) represent each individual type of page of our site (/superbowls/, /superbowls/id, /franchises/, /franchises/id, /mvps/, /mvps/id)  and include the logic for django to include content data from the database into the html, delimited by { } or {{ }}.

Every page in the SBIDB references the same stylesheets (*.css) for organizing and formatting html elements in the pages: bootstrap.css and justified-nav.css ( located in /static/bootstrap/css/). Bootstrap.css is the entire distribution of Twitter Bootstrap-3.1.1 that includes rules for many of the base elements of the page including paragraphs, headers, rows, tables, and most importantly, the logic that makes these elements resize in response to changing screen size or the type of screen medium (i.e., mobile or desktop). Justified-nav.css includes styles we desired beyond those provided in bootstrap.css, principally, the styling of the navigation bar.

A javascript file (script.js), linked into every page via the root template, provides logic to correctly style the navigation buttons to provide the appearance that a button is depressed while on that specific page or when a user moves his mouse cursor over a button.

## 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.

## III. 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 our first  iteration of SBIDB were: Super Bowl, Team, Player, Venue and Roster. We decided to overhaul our models in the second iteration changing to SuperBowl, Franchise and MVP. These respectively correspond to Game, Team and Player, giving us simplified relationships.

Each instance of the SuperBowl model uniquely represents a Super Bowl game by encapsulating the pertinent relative data that we used to create our dynamic web pages. Furthermore, each SuperBowl instance relates to two Franchise instances and one MVP instance. In addition to the general Super Bowl information, we decided to add fields for various social media IDs, latitude and longitude. We pulled the general Super Bowl information from our database to upload text and images onto our page. The social media IDs were used to upload external media onto our webpages. The embedded YouTube videos required the URL, so we entered the unique path of each video in the corresponding ID field. From there, we could upload the unique path into the template and generate the correct media for each page. The Twitter widget was a bit more complicated. Each widget has a unique ID that is generated when you create the widget. Thus, we had to create each widget and extract this ID to get the correct Twitter feeds. The latitude and longitude fields were FloatFields and were entered into the GoogleMap script to generate the desired location on our embedded maps.

Each instance of the Franchise model uniquely represents the Super Bowl history of a specific organization. We decided that a Franchise model would capture the history of the Super Bowl in a more efficient manner. For example, the Pittsburgh Steelers have appeared in three Super Bowls. In our original design, each team that appeared in the Super Bowl (ie: the '06, '09, '11, Steelers would have all had their separate pages). In our new design, the pages for the '06, '09, '11 Super Bowls will all be connected to the Pittsburgh Steelers franchise, resulting in a more efficient display of Super Bowl History. Each instance of a Franchise relates to the players who have won the Super Bowl MVP for that respective franchise. The Franchise model used the same social media fields as the SuperBowl model, but we added a Facebook plug-in to each team. The Facebook plug-in required a similar ID as our YouTube videos, so we extracted the unique path and used that for the Facebook ID.

Lastly, each instance of the MVP model uniquely represents the player who won the MVP and encapsulates unique data for that individual. The MVP model used the same social media fields as our Franchise object, where each field was geared towards the player rather than the franchise.

## Relationships

### SuperBowl - Franchise

We chose each property of the models to fundamentally describe each model while offering reasonable relationships between them. Each SuperBowl instance is linked to two franchises, the winning franchise and the losing franchise. Both SuperBowl - Franchise relationships follow a many-to-one relationship, where one franchise is related to many Super Bowls. This allows us to keep an extensive network of franchises and the Super Bowls they've played in regardless of the outcome. For example, consider the New York Giants, who have played in two of the last ten Super Bowls. To add the Giants franchise to our database, we need to relate them to both Super Bowls they played in. Thus a many-to-one association is ideal for this relationship. We modeled this relationship using the ForeignKey field. This relationship was included under the SuperBowl model due to the syntax of Django's ForeignKey field.

### SuperBowl - MVP

A SuperBowl instance also describes a many-to-one relationship between instances of a SuperBowl and a MVP. Each SuperBowl instance is only linked to one MVP. However, each MVP is linked to many SuperBowl instances. Consider Eli Manning, a two-time Super Bowl MVP. If we wanted to add Eli Manning to our database, we would want to connect him to both Super Bowls where he was crowned as the MVP. Therefore, a many-to-one relationship is ideal for the SuperBowl - MVP relationship. Like the SuperBowl - Franchise relationship, the SuperBowl - MVP relationship was modeled using Django’s ForeignKey field. We put this relationship under the SuperBowl model due to the syntax of Django’s ForeignKey field.

### Franchise - MVP

A Franchise instance describes a many-to-many relationship between instances of a Franchise and instances of a MVP. We chose a many-to-many relationship because a player can win multiple MVPs for multiple Franchises. Although this has not happened in the history of the Super Bowl to date, we left it as a many-to-many relationship in case it does happen in the near future. On the other hand, a Franchise can have multiple MVPs depending on its number of Super Bowl appearances. This is more common and seen consistently throughout the history of the Super Bowl.

## IV. 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). The Django framework provides the ability to implement the API using a single Python file, api.py.

### Implementation Patterns

SBIDB has three entities SuperBowl, Franchise and MVP. 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/

## V. Testing

We developed a set of unit tests to test each API call from our REST API. Django provides a test module, aptly named django.test to run these tests.

Additionally, Django’s Client class allows for us to create a local testing database, run a client, and test the calls to the API through the running client. We created helper functions in the test class to handle the basic API calls. These functions take in the specific type of entity which we are creating the call against, and an optional id for the specific calls, as well as an optional request body for POST and PUT calls. The helper function then will create the actual request through the client module and then return the response back to the calling function in the form of the status code and response payload. The response payload is returned to us as a JSON object which we decode and load using an imported json module. Our API is set to return a response payload with two keys, a boolean of success, as well as a boolean with the data. The data key contains the value for the response content which we check for validity in each test.

## Testing the GET call

For our GET requests we use testing helper methods to formulate a get call for each of the entities in our database/models. These helper methods return both the response payload, as well as the response code from the call. Using an external helper method we are able to replicate the expected response from the GET call, and check that against the actual response we received. For GET calls to retrieve an id specific response we use the ‘get’ helper method, but simply provide an id in the call, and follow the same procedure as with the generalized form.

## Testing the POST call

POST calls are tested for each entity type in the Django models, and hence actual database. For each POST call we create a request by mimicking the format of an entity in our testing database with which to enter a new entity into the database. We place this request into a Python dictionary and send the request to the 'post' helper method which then makes the call via the client to the API. This again returns the status code as well as response payload which we are able to check for the correctly returned expected values. For POST we expect a status code of 201 for successful cases. The response payload that is received will indicate that a new id and url have been created for this new entity. After checking that the status code and response payload are returned as expected we then check that the newly added entity is present in the testing database. This shows that both the API is returning what we expect as well as modifying the database as it should be.

## Testing the PUT call

PUT calls are similar to POST calls by nature, except that we expect to modify the values that currently exist in the database. Following a the pattern of POST, we create a Python dictionary this time with an object that mimics an entity in our current testing database, and we change some values in the fields. When we make the request to the API, we expect to receive back a response payload that shows that we have changed values. A successful POST call will return to us a status code of 200. After this, we again test that the changes to our testing database are evident.

### Testing the DELETE call

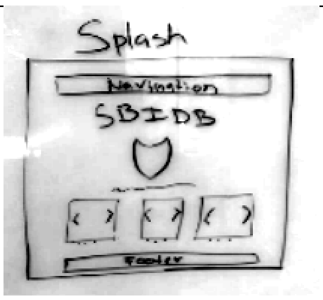
DELETE calls are created using the delete helper methods in the testing class. We send the id of an entity which we would like to delete, along with its type to the helper method, and expect to receive a success code of 200 as well as a response payload that includes a null value for the data field.

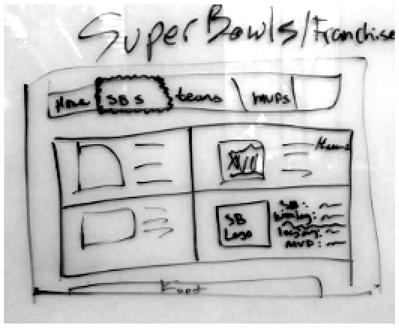
### Testing failure cases

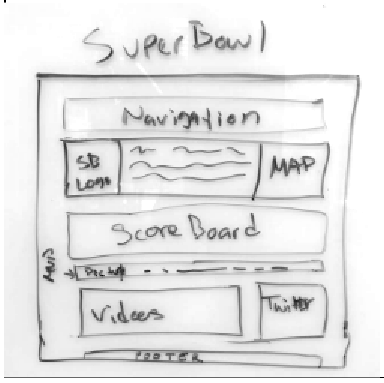
We included additional test cases that were set up to deliberately fail. We did this to ensure that our tests were correct. For example, we changed some our content to make sure that our tests were comparing data and not type of the content. This would ensure us that our API calls were performing and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

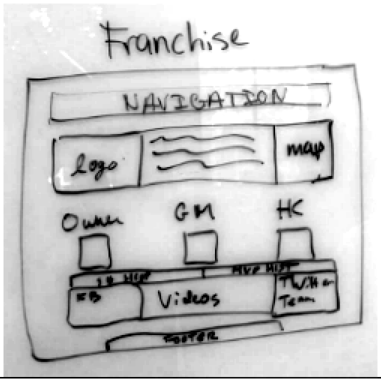
## VI. Reference Images

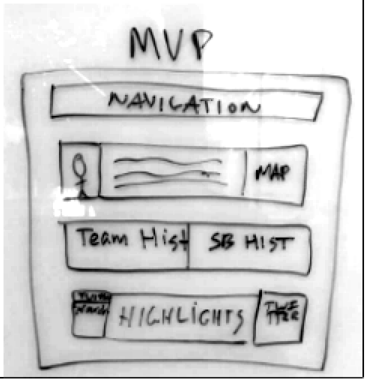
### UI Mockups


Splash
Navigation
SBIDB
Footer

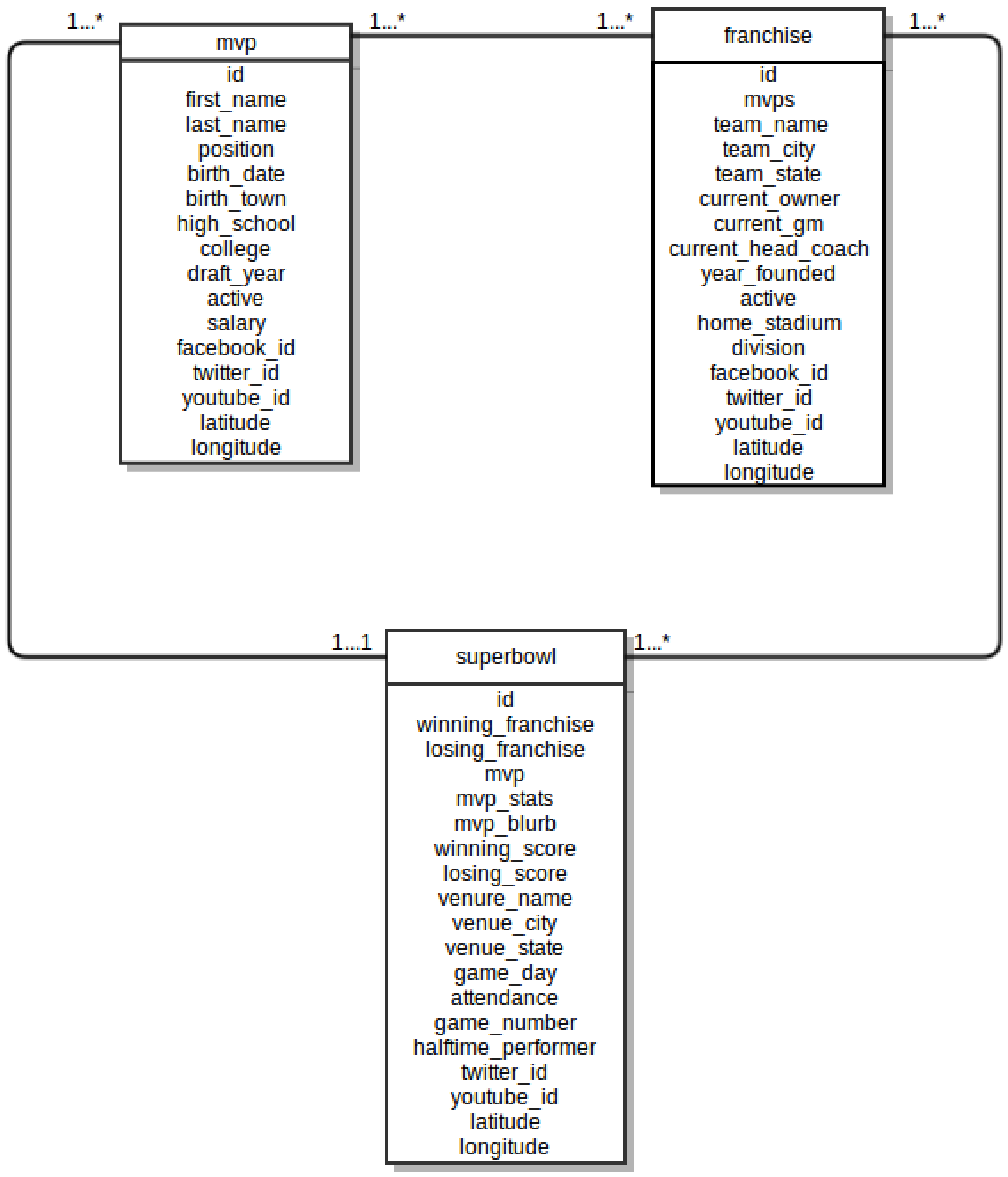

SuperBowls/Franchise
Home
SBs
teams
mvps
SB Logo
MVP
Foot


SuperBowl
Navigation
SB Logo
MAP
Score Board
Videos
Twitter
FOOTER

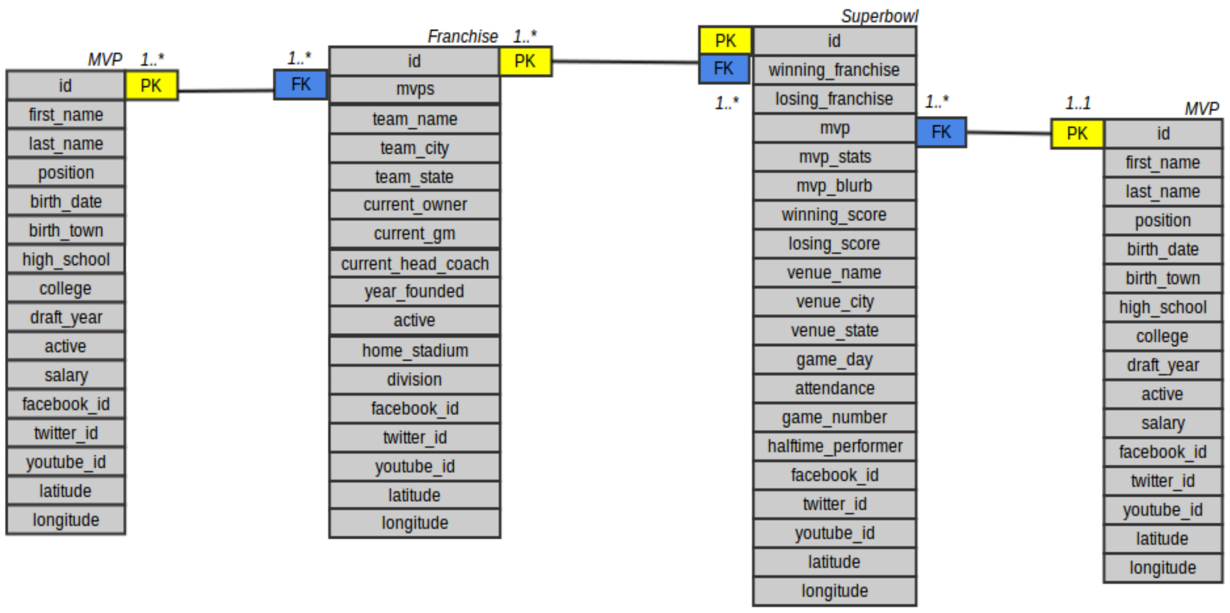

Franchise
NAVIGATION
logo
map
Owner
GM
HC
SB HIST
MVP HIST
SB
Videos
Twitter Team
FOOTER


MVP
NAVIGATION
MAP
Team Hist
SB HIST
HIGHLIGHTS

## UML Class Diagram



## Relationship Diagram


MVP
1..*
id
PK
first_name
last_name
position
birth_date
birth_town
high_school
college
draft_year
active
salary
facebook_id
twitter_id
youtube_id
latitude
longitude
1..*
FK
Franchise
1..*
id
PK
mvps
team_name
team_city
team_state
current_owner
current_gm
current_head_coach
year_founded
active
home_stadium
division
facebook_id
twitter_id
youtube_id
latitude
longitude
Superbowl
PK
FK
1..*
id
winning_franchise
losing_franchise
mvp
mvp_stats
mvp_blurb
winning_score
losing_score
venue_name
venue_city
venue_state
game_day
attendance
game_number
halftime_performer
facebook_id
twitter_id
youtube_id
latitude
longitude
1..*
FK
1..1
PK
MVP
id
first_name
last_name
position
birth_date
birth_town
high_school
college
draft_year
active
salary
facebook_id
twitter_id
youtube_id
latitude
longitude