

Banana-Fish

Video Game Database

Group Members

Ruben Baeza

Andy Tang

Nico Vasquez

Alexander Ventura

Steven Zvonek

Introduction

BananaFish is a Gaming database used to find relevant information pertaining to the gaming industry. It focuses on three pillars: games, companies that create games, and jobs in the gaming industry. All of these pillars relate to each other in some way.

Problem

There are many websites on the web that offer reviews about games and give relevant information about new releases, genres, platforms. The problem is that they don't show openings for jobs in the gaming industry and allow you to sort the games by company.

Solution

Our goal is to create a friendly user experience for gamers that allows them to navigate through game titles, read reviews on the games they like, and have an opportunity to connect with the companies that make these games and apply for a job there. We give the users a simple web design so that they can parse through information that links to other information in a tightly connected graph. For instance, the user may see a game they like called World of Warcraft and see that the game is made by Blizzard Entertainment. If they click on this video game developer, they will be taken to their company page where they can find relevant information on where they are located and what games they produce. If they are still interested they may choose to click on jobs openings and apply to become a software engineer or a graphic designer. Users will also be able to browse the job opening pillar and if they see a job opening they are interested in, they can find useful information about the company and the games they produce before they apply so they can know what to expect in the interview.

Design

Technologies

Multiple technologies were used to build BananaFish, many of them being new to most of the developers involved. It was a good learning experience for us that didn't know what some of the technologies were.

Back-end

BananaFish is hosted on a Rackspace virtual machine on an Ubuntu Operating System. We used PostgreSQL as our database, and used Flask models to represent the three pillars.

Front-end

The BananaFish website is written using HTML, while also using Twitter Bootstrap and AngularJS to style the website to have a consistent theme across each pillar. The website is organized into three pillars: individual games, game companies, and jobs with those companies.

RESTful API

We provide an API for our database so that we and others can have an easy way to access and update information on our server. For each of the entities: Game, Platform, Company, Genre, and Jobs, we provide an API to create an entry using the POST HTTP protocol. We also provide get and get all using the GET HTTP protocol. Using the GET call, one can obtain all the information about one entry for an entity or they can obtain all of the entries for an entity.

Game API calls

Lookup a Game (GET HTTP request) - Get the game name, image, original release date, deck(a shortened summary), a full description, company id, and game id using game id as the required number parameter.

Add a Game (POST HTTP request) - Add a game to the database using the game name, game release date, game deck, game description, the company that created it, and the company id using required integer as a parameter. It returns a response with everything and the game id it creates.

Get a list of all the Games (GET HTTP request) - Returns a json response containing a dictionary containing other dictionaries for each game with attributes being the same as looking up a single game.

Company API calls

Lookup a Company (GET HTTP request) - Returns a response with the company id, name, deck, description, image/logo, address, city, state, country, phone number, data founded, and website url using a required number parameter.

Add a Company (POST HTTP request) - Add a company to the database using the company name, deck, description, image/logo, address, city, state, country, phone number, data founded, and website url with a new company id number being the required number parameter. It returns json along with a company id.

Get a list of all Companies (GET HTTP request) - Returns a json dictionary containing a dictionary for each company containing the company id, name, deck, description, company image/logo, address, city, state, country, phone number, data founded, and website url.

Job API calls


Lookup a Job (GET HTTP request) - Returns a response with required number parameter of job id and company id to get the result of a company id, name, job id, position, description, location, and source url.

Add a Job (POST HTTP request) - Add a job to the database with a parameter of a company id and a job id while requesting company id, name, job position, description, location and url. Returns json along with other job ids.


Get a list of all Jobs (GET HTTP request) - Returns a json dictionary containing a dictionary for each job with the company id, name, job id, position, description, location and source url.

Documentation

The documentation for BananaFish API: <http://docs.bananafish1.apiary.io/#>

**Banana-Fish**
Steven Zvonek • bananafish1

Documentation Inspector Editor ?

Andy 

API Blueprint Syntax Tutorial

Valid API Blueprint Preview **On** Save & Publish

Fork on GitHub

INTRODUCTION

REFERENCE

Game

Company

Job

Banana-Fish

INTRODUCTION

BananaFish is an API that allows people to find games, companies, and jobs in the gaming industry.

REFERENCE

Game

API calls related to resources of the **Games API**

Games

Attributes belonging to a **game**

- Game ID
- Name
- Image
- Original Release Date
- Deck
- Description
- Company ID

Lookup a Game

Add a Game

Get all Games

Get a list of all Games

Company

User may search up companies of interest to look up basic information about them.

Companies

Attributes belonging to a **company**

- Company ID

Switch to Console

Game Games **Lookup a Game**

GET `http://private-e4a6a-bananafish1.apiary-mock.com/models/Game.py/game_id`

Parameters

game_id	ID of the Game in form of an integer Example: <code>1</code>	Number
----------------	--	--------

Request

Mock Server

Raw

Try

Response

200

Content-Type: application/json

```
1 {
2   "game_id": 1,
3   "name": "League of Legends",
4   "image": "image.jpg",
5   "original_release_date": "10-27-2009",
6   "deck": "MOBA",
7   "description": "League of Legends is a fast-paced,
8   "company_id": 1
9 }
```

Models

Our models are Company, Game, Job, Game_Genre, Game_Platform, Genre, and Platform. These models allow easy access to the tables' data without having to worry about writing the most efficient sql queries ourselves.

Company: This model contains basic information about a company. It has a one-to-many relationship with the Game model as well as the Job Model. Companies are uniquely identified by a company_id.

Job: This model contains information about jobs for a company and has a many-to-one relationship with the Company model.

Game: This model contains information about video games. It has a many-to-one relationship with the Company model, as well as a one-to-many relationship with the Game_Genre and Game_Platform models. Each game is uniquely identified by a game_id.

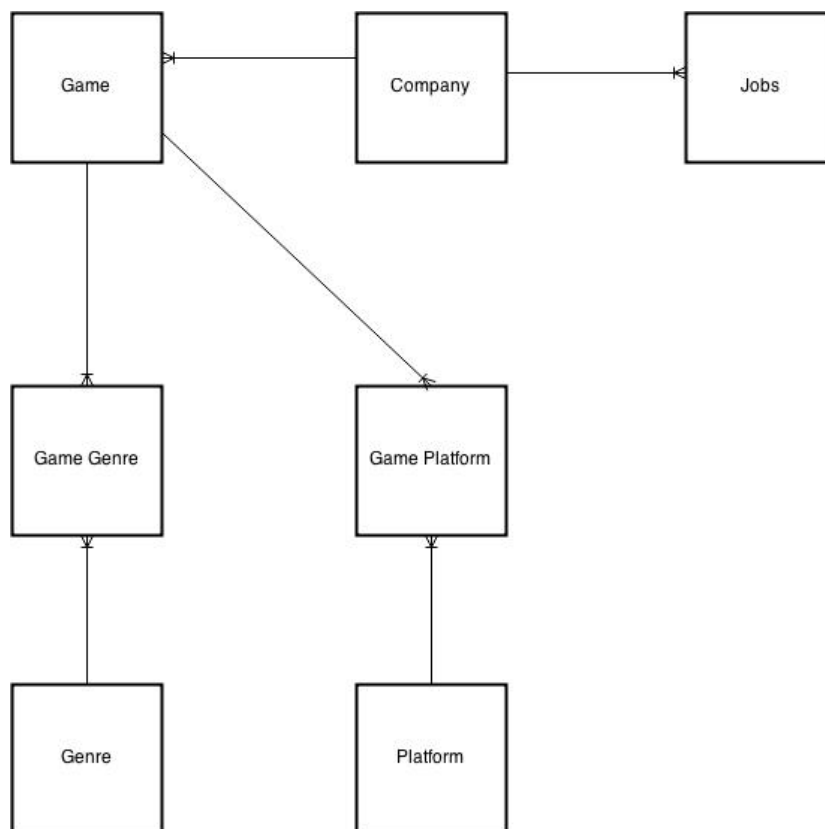
Game_Genre: This model simply contains game_id - genre_id pairs. It has a many-to-one relationship with the Game and Genre models. Having this model along with the Genre model, which contains the genre's name, reduces the space required for genre information, as well as allows us to alter a specific genre by editing only one row in the Genre model.

Genre: This model contains a genre_id and the genre's name. Having this enum model allows us to save time when editing genres, as well as space required for all genre information.

Game_Platform: This is identical to the Game_Platform model except that it links games to platforms. It contains pairs of game_id's and platform_ids. It provides the same benefits as the Game_Genre model.

Platform: This model contains information about a platform. Having this enum model allows us to save time when editing platforms, if needed, as well as space required for all platform information.

UML Diagram



Tests

Testing of the SQLAlchemy models uses a sqlite database. These tests confirm the desired behavior when creating new instances of the different models under varying conditions. Before each test the setup method is called. This creates a fresh tables and allows each test to be completely isolated from the others. After each test a call to `teardown()` is made. This resets the database contents and is another step in isolating the tests from one another.

For the **Company** model there are two types of tests. One type creates new model instances, adds them to the database, and confirms they can be queried for. Another type of test ensures that companies with duplicate `company_ids` can't be inserted.

For the **Game** model there are three types of tests. One type creates new games, adds, and queries for them, just like for the Company model. Another type ensures that games with duplicate `game_ids` can't be inserted. A third type ensures that games can't be added if they reference a company that doesn't exist.

For the **Genre** and **Platform** models there are two types of tests. One makes sure they can be created, added, and queried for. Another ensures that they are unique based on their `genre_id` and `platform_id`, respectively.

For the **Game_Genre** model tests ensure they can be created, added, and queried for. Tests also ensure that when creating a `game_genre` the game exists as well as the genre.

For the **Game_Platform** model tests ensure they can be created, added, and queried for. Tests also ensure that when creating a game_platform the game exists as well as the platform.