Internet Video Game Database

Pirates Of Si Valley

Noah Cho, Doug Goldstein, Robert Hammond,

Arash Ghoreyshi, Cristhian Escobar, Richard Lage

Introduction

Our project will be based on video game developers, along with the games they have developed and the platforms they develop for. After forming our group, we decided the best way to pick a topic would be to split up on our own and brainstorm topics we thought would be unique, and easy to find information on. When we re-convened at the next class we decided to go with Richard Lage's idea to create a Nintendo Video Game Database, with Employees, Games, and Developers as classes. However, we later decided as a group Nintendo was too specific of a topic, so we expanded our project to include non-Nintendo developers. Ultimately, we decided to remove the Employee class, and instead used a Platform class. If you are not a big gamer you may wonder what use cases may result from our database so here are a few examples:

- Buying a game at the store may seem like a fun time, but how can you find a game that will guarantee your money is well spent before even opening the box?
- Maybe you are an aspiring game developer and want to know what companies have made all of your favorite types of games so that you can track down your dream job.
- Gamers of all types are passionate about their platform of choice to the point that most generations of consoles involved having "console wars", in which these consoles and their following compete against each other to come out on top of the market.

Factors that might influence these different concerns could be developers, platforms, publishers,

dates in which these products are released, ESRB ratings, genres and much more. We feel that it would be useful to have a web interface with a database to sift through all these varying factors and quickly find your favorite in each field that results in which game you would like to play, which developer you would like to work for, or which platform you would like to buy.

API Blueprint

REST

We began the task of developing a video game database after deliberation and agreement. We started work on our plan by building of a RESTful API and Django Models. For our RESTful API we used Apiary.io witch is a type of testing environment and debugger for our API. We were able to attach Apiary to our Github repo through the settings so all commits from apiary.apib were processed through Apiary.io. Once we started making our RESTful API it was a bit hazy at first since we had trouble understanding exactly the purpose of it. Some of us gathered that it would work as a form of Unit Testing for our project. Our Unit Testing theory would make sense because we were building it before our actual working code and according to "Extreme Programming Installed" this should be the proper first step for our group. Later on this would turn out not to be the case but why?

> *REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. ~Fielding, Roy Thomas*

So in actuality what we were doing with our REST is building our API. RESTful is just a type of API that we can use in the instance of our particular database. We use the REST to really focus on

the classes that we are building and think about how they are going to interact with one another.

Apiary

With no Apiary experience in the group, we first needed to learn the API Blueprint Language. After reading through the provided tutorial, we found that the language itself was simple, but we were unsure what Apiary exactly did. We understood that Django would be creating the database, and Heroku would be hosting the project, but how did Apiary connect and interact with the other tools? After further investing into Apiary, we concluded that it didn't really interact with either of the tools. Apiary will act as form of documentation for the API, showing the valid calls, and the expected response. So what we concluded from this learning process is that Apiary is just a debugger and unit tester for RESTful API code that compiles as you type the code. Apiary really is a Blueprint of our API since it doesn't have a live server actually handling database requests. There are tools that we can use to extract our work from Apiary later when we build our real API so that our apiary code isn't useless.

Blueprint

Our first approach at an API was minimalist. We tried to combine the GETs for each table into one call, using a parameter. However, we quickly discovered this plan was misguided because the get call to the Developer class would expect a different set of attributes then a call to the Game or Platform classes. Instead, each table required its own set of seven API calls. Each table has a GET call, which returns a list of the existing objects, as well as a POST, which creates a new object. Using the primary key as a parameter, we threw API calls on a specific object in a table. These calls are a GET, PUT, and DELETE, which returns the object, updates the object, and deletes the object respectively. The last two calls are GETs, which returns a list of related

objects in the other two tables. We want to note that we had an issue with single-quotes vs. double-quotes in JSON. Apparently, JSON files are not quite as flexible as Python when it comes to quotation marks, so we switched from using single-quotes to using double-quotes.

Django

Getting Started

After figuring out what our RESTful API was, we then knew why we had to deliberate on the UML design of our classes and their attributes. What we figured out reading the Django Documentation is that we could use our UML models as a guide for building essential portions of our Django App. Django in its entirety is a high-level Python Web framework designed for journalists and software engineering teams to make clean elegant web applications quickly. In Django's Documentation we are told that Django applications have a package that follows a specific convention that conveniently for us was already set up by Twitter Bootstrap. The package requirements for Django are an application file with __init__.py, admin.py, models.py, tests.py, and views.py. You can see in figure 1 our application file is a bit more bloated but still contains the required files other than tests.py which we keep in the parent directory cs373-idb.

```
cs373-idb/idb
        html
        __init__.py
        admin.py
        models.py
        resources.py
        setings.py
        urls.py
        views.py
```

wsgi.py

Figure 1

UML

The Django models are actual code representations of the UML models we built. Our models in the file models.py is an information source to be used by our API as a directory of information about our classes. Each of our models is a Python class that subclasses django.db.models.Model. Before writing our models, we had trouble figuring out what we want to be our primary key across all classes. Names and titles were decided to be the best primary keys across our classes. We negotiated what we wanted to be our three class types and settled on Developer, Platform, and Game. Then decided what attributes each class would contain. When we finished writing out our models by hand, we were left with an image that resembles the UML diagram in Figure 2 below:

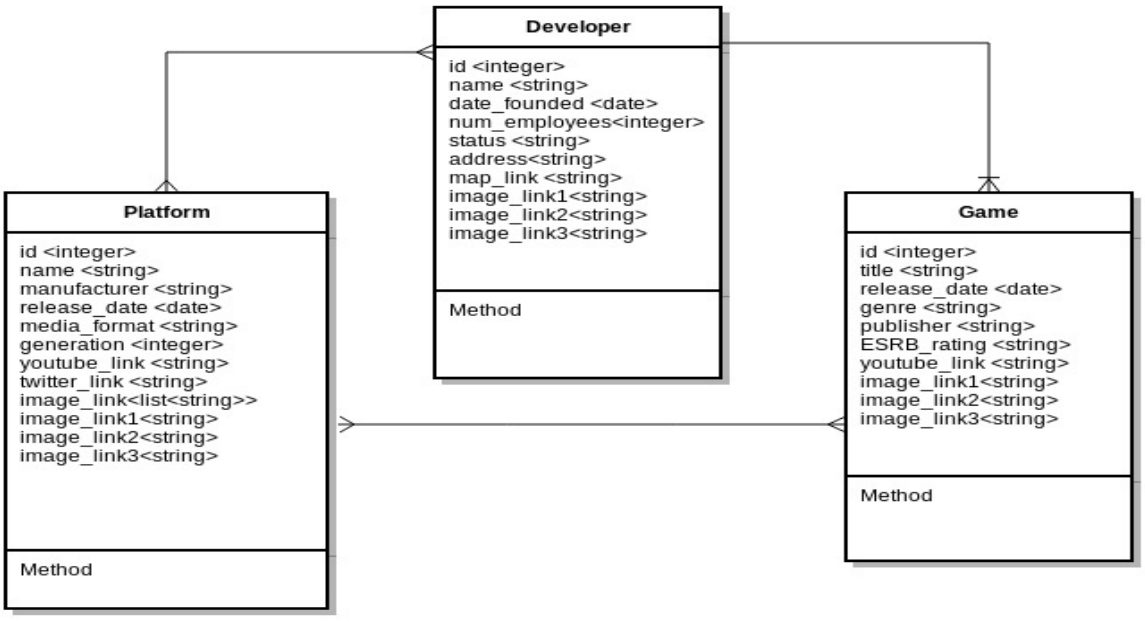

Figure 2

Models

Relations

We decided to make the relation from Game to Platform many-to-many since one game can be on many platforms as we can see with Guitar Hero in Figure 3. From Platform to Game the relation is many-to-many as well since we can expect any single platform competing in the marketplace would need to have many games. For the Platform and Developer classes we also notice a symmetric relation of many-to-many. This is understandable since a platform certainly has multiple developers at any time making games for them. The symmetry of the many-to-many relation from Developer to Platform is apparent as well since developers can make game across many platforms which we see in Figure 3. In the case of Developer and Game, we also have a many-to-many relation. A developer can create multiple games, and if they don't create at least one then they are not actual developers. The relationship between Game and Developer is unique since it is the only one-to-many relationship in our schema. It is possible for a game to have multiple developers working on it but credit for creating the game usually goes to one main developer. Each class we are about to delve into also includes a definition for __unicode__() which returns the Primary Key of each. Our function name should have been __str__() according to the Django Documentation but after checking Stackoverflow it appears that __str__() was the old formalism that returns bytes and the new prefered method to get characters is our function name __unicode__(). After our definition in each class there is a Meta class that places the ordering by primary key and gives us a plural version of the class name for calling the class.

Figure 3

Class Name: Platform

- Name - a character field with a maximum length of 255; primary key
- Manufacturer - a character field with a maximum length of 255
- US Release Date - a date field that is the US release date
- Generation - an integer field
- Media Format - a character field with a maximum length of 255
- Youtube Link - a character field with a maximum length of 255
- Twitter Link - a character field with a maximum length of 255
- Image Link(3 for each)- a character field with a maximum length of 255
- Developers - a many-to-many field of class type Developer; foreign key
- Games - a many-to-many field of class type Game; foreign key

As you can see we have the name field which will contain Nintendo Wii U, DS, Game Boy Advance, Gamecube, Wii, and 64. Other platforms are Xbox 360, Playstation 2 and 3, and

the MSX. The different manufacturers are Nintendo, Microsoft, and Sony. The Release Date field is represented by a date expressed in the form “mm-dd-yyyy”. Generation is just an integer field representing the period of time each console was released in. For example, we are currently in the 8th generation of consoles. The Media Format field is just a string describing what type of media is used for the software, be it physical or digital followed by what type of physical media if so. Our Youtube Link and Twitter Link attributes are just a url of type string. Image Link is an attribute represented as a string of a JPEG url that shows a picture of the system. The Map Link is an embedded url link of type string. The Developer and Game fields both showcase the many-to-many relationships the Platform class as shown in Figure 2.

Class Name: Developer

- Name - a character field with a max length of 255; primary key
- Date Founded - a date field representing the year it was established
- Num Employees - an integer field
- Status - a character field with a maximum length of 255
- Image Link(3 for each) - a character field with a maximum length of 255
- Map Link - a character field with a maximum length of 255
- Platforms - a many-to-many field of class type Platform; foreign key
- Games - a many-to-many field of class type Game; foreign key

For developers we start by the name where we chose Platinum Games, Infinity Ward, Cing, Rockstar Games, Dimps, HAL Laboratories, Monolith Soft, Konami, Midway Games, and Square Enix. These companies had to be founded at some time so we have the Date Founded field to express that in the form “yyyy”. Number of employees will just be an integer. The Status

attribute allows us to know if the company still exists since many of these companies might have dissolved or are still going strong, so we chose the strings “Active” and “Defunct” to represent them. Our Image Link attribute is just a JPEG url represented in string form that is most likely of the developer logo. Map Link attributes are embedded urls of type string that show the location of the development offices. The Platforms and Games attributes are needed so that we can have the many-to-many relation represented between these classes, which we express in Figure 2.

Class Name: Game

- Title - a character field with a maximum length of 255; primary key
- US Release Date - a date field representing the US release date
- Genre - a character field with a maximum length of 255
- Publisher - a character field with a maximum length of 255
- ESRB Rating - a character field with a maximum length of 255
- Youtube Link - a character field with a maximum length of 255
- Image Link(3 for each) - a character field with a maximum length of 255
- Developed by - a one-to-many field of class type Developer, foreign key
- Platforms - a many-to-many field of class type Platform; foreign key

Our Game class starts with the Title attribute which we chose “The Wonderful 101”, ”Call of Duty 4: Modern Warfare”, ”Hotel Dusk: Room 215”, ”Grand Theft Auto III”, ”Sonic Advance”, ”Super Smash Bros. Melee”, ”Xenoblade Chronicles”, ”Metal Gear”, ”Doom 64”, and ”Final Fantasy XIII”. For each game we have a release date represented as “mm-dd-yyyy”. Genre is what type of experience the game entails. For the Genre attribute we have Action, First-person shooter, Point-and-click adventure, Open-world action-adventure, Platformer, Fighting, Action

role-playing, Turn-based role-playing, and Stealth. The next field is a bit tricky since there is a bit of overlapping between the Name attribute in the Developer class and Publisher attribute here. We talked about this when we met up and decided that there really isn't overlap between these two since some companies, like Nintendo, actually encompass both entities but in different respects since a company like Nintendo, for example, will publish games in one branch of their company and have another branch working on programming a new video game for that platform. So even though Developers and Publishers can have the same name it doesn't mean that they are the same branch of the company. Our Publisher field currently has Nintendo, Activision, Rockstar Games, THQ, Konami, Midway Games, and Square Enix. The ESRB Rating acts as an age rating system for gamers intending to buy the game. Our ESRB Ratings are "T" for Teen, "M" for Mature, "E" for Everyone, and "N/A" for Not Applicable (which will refer to games that got released before the ESRB was established). The Youtube Link attribute is a url represented in string form that is quite likely a short demo of game play. Our Image Link attribute is a JPEG url that can be of either the game packaging, characters, or the actual media device. This model is also unique because it is the only class that includes a one-to-many relationship with the other classes. For this specific case, it is the Developer class. There is also a many-to-many relationship for the Platform class in correlation with our UML models represented in Figure 2.

API

Tastypie

Once we began phase two of our project we ran into a dilemma. We already have our working API Blueprint in Apiary but now we need to make our Blueprint an actual working API.

The hard headed way would be to code it all by hand, but why do that when we have a beautifully developed RESTful API in Apiary. Luckily Alex Cooman helped us on Piazza and notified us of Tastypie. There are more tools out there other than Tastypie that will do the same job, maybe even better according to Tastypie Documentation, it seems though that we would choose this particular tool since:

- We are working with an API that is RESTful
- It saves us time on having to re-write our Django serializer to make our output right
- This framework is direct with no magic sauce and flexible
- We might want to use JSON serialization with our data.json file
- The ability to test our API locally until we are ready to push to Heroku

Tastypie is a web service API framework for Django that acts as a reusable app and provides an API to any application without having to modify the sources application. Tastypie in a nutshell does all the heavy lifting of converting over your RESTful API in Apiary to any application you like, in our case Heroku. Tastypie isn't unique to Apiary as a starting point because the software actually comes with a ton of "hooks" for overriding and extending its functionality. However, Tastypie is unique to Django which makes it unique to Python as well. Also, Tastypie only works on providing a REST-style API which is perfect in our case.

Preparation for a Tastypie

1. We added Tastypie to our INSTALLED_APPS
2. Create an API directory in our app with an empty __init__.py
3. In the API directory make a resources.py file like ours in figure 4

```
from tastypie import fields
from tastypie.authorization import Authorization
```

```
from tastypie.resources import ModelResource, ALL, ALL_WITH_RELATIONS
from idb.models import Platform, Developer, Game

class PlatformResource(ModelResource):
  class Meta:
    queryset = Platform.objects.all()
    resource_name = 'platforms'
    authorization = Authorization()
    allowed_methods = ['get', 'post', 'put']
    filtering = {
      'name': ['exact', 'starstwith', 'endswith', 'contains'],
      'manufacturer': ['exact', 'starstwith', 'endswith', 'contains'],
      'release_date': ['exact', 'lt', 'gt', 'lte', 'gte', 'range'],
      'media_format': ['exact', 'starstwith', 'endswith', 'contains'],
      'generation': ['exact', 'gt', 'gte', 'lt', 'lte', 'range'],
    }
class DeveloperResource(ModelResource):
  platforms = fields.ToManyField(PlatformResource, 'platforms')
  class Meta:
    queryset = Developer.objects.all()
    resource_name = 'developers'
    authorization = Authorization()
    allowed_methods = ['get', 'post', 'put']
    filtering = {
      'name' : ['exact', 'starstwith', 'endswith', 'contains'],
      'date_founded' :['exact', 'lt', 'lte', 'gt', 'gte', 'range'],
      'num_employees' : ['exact', 'lt', 'lte', 'gt', 'gte', 'range'],
      'status' : ['exact'],
      'address' : ['exact', 'contains', 'startswith', 'endswith'],
    }
class GameResource(ModelResource):
  developer = fields.ForeignKey(DeveloperResource, 'developers')
  platforms = fields.ToManyField(PlatformResource, 'platforms')
  class Meta:
    queryset = Game.objects.all()
    resource_name = 'games'
    authorization = Authorization()
    allowed_methods = ['get', 'post', 'put', 'delete']
    filtering = {
      'title' : ['exact', 'starstwith', 'endswith', 'contains'],
      'release_date' : ['exact', 'lt', 'lte', 'gt', 'gte', 'range'],
      'genre' : ['exact', 'starstwith', 'endswith', 'contains'],
      'publisher' : ['exact', 'starstwith', 'endswith', 'contains'],
      'ESRB_rating' : ['exact'],}
```

Figure 4

4. Finally we need to specify our URLconf bits in our API directories urls.py file ours is below in figure 5

```
from django.conf.urls import patterns, include, url
from django.views.generic.base iport TemplateView
from django.contrib import admin
from tastypie.api import Api
from idb.resources import PlatformResource, DeveloperResource, GameResource

admin.autodiscover()
v1_api = Api(api_name='v1')
v1_api.register(PlatformResource())
v1_api.register(DeveloperResource())
v1_api.register(GameResource())

urlpatterns = patterns('',
  url(r'^$', 'idb.views.home', name='home'),
  url(r'^game/(?P<game_id>\d+)/$', 'idb.views.game', name='game'),
  url(r'^platform/(?P<platform_id>\d+)/$', 'idb.views.platform', name='platform'),
  url(r'^developer/(?P<developer_id>\d+)/$', 'idb.views.developer', name='developer'),
  url(r'^admin/', include(admin.site.urls)),
  url(r'^api/', include(v1_api.urls)),)
```

Figure 5

5. Now put http://localhost:8000/api/vi/?format=json in a browser and voila

Resources

The second step for starting Tastypie was easy for us since we already had an API directory thanks to Twitter Bootstrap. The contents of our Twitter Bootstrap idb file became even more clear in this phase since Tastypie required the same contents in reguard to resources.py, urls.py, and the same __init__.py that Django used. Our resources.py file represented by figure 4 ended up having three resources which represented each of our database classes. The interesting thing we found in resources.py was that we can control a lot of the abstraction easily with our Meta class. Setting the filtering variable allowed us to provide a list of fields that the resource will accept

client filtering on. The allowed_methods option let us control what list and detail REST methods our resources should respond to by building a simple assignment of a list containing our prefered API commands. Authorization classes for a resource were controlled with authorization, these are set to read only by default, meaning that only GET calls would work properly. To override this, we made it so that authorization happens automatically by calling the Authorization constructor. We also set resource_name on each to its plural class name. At the top of each Meta class in figure 4 we assign a queryset that provides its resource with Django models to respond with. Other than our Meta classes, our actual resource classes contained a many-to-many relation with Platform for Developer Resource and Game Resource. Game Resource also contained a Foreign Key for Developer, representing its one-to-many relation with that class.

URL Configurations

With our Entry Resource done with we were now able to connect it to our URL configuration. You can see in figure 5 our file starts with the needed imports from Tastypie, Django, and our Resources. After that we use the automatic admin interface to read metadata in our model and provide a production-ready interface so we can add content to the site. As you can see below our autodiscover() call in figure 5 we have a line for each Resource in a function call of the form v1_api.register(). register() is actually registering an instance of the Resource subclass with the API here. The last block of code for figure 5 is our urlpatterns assignment statement where we set up the patterns for home, game, platform, developer, admin, and api.

Testing - Unit Tests

Looking at our unit tests we approached each class (Game, Developer, and Platform) by

testing each of the API calls GET, POST, PUT, and DELETE. One of the ways we refactored our test was by creating dictionaries at the top of our Python file so we wouldn't have to hardcode it into the method every time. We had to improve our unit tests after our grader on this project told us to not just test the return codes from our RESTful API, but to actually check the contents of our JSON file to validate it. Our JSON test code can be found at the bottom of each of our testing methods as the response_content assignment line and our == assertion. We also included a few extra tests to ensure that our set-up with Tastypie was working properly. These tests are within the last three classes of tests.py.

Conclusion

In retrospect, we chose a topic for our database that has three useful real world implications at minimum. Before starting, we created our unit tests that passed at zero percent till we got them to one hundred percent passing. We had to repair our unit tests mid-project due to our trivial testing cases so we could actually verify our JSON files. Once we made our first round of unit tests we had to figure out what our RESTful API was doing and it turned out to just be a class of API that conveniently works with the tools Django, Apiary, and Tastypie. In order to think out our RESTful API we first had to reach a final decision on our Django Models as a group. The Django Models would be used to form our artificial API host till we perfected it and could move it to our Heroku host. The Django Models were important because they would regulate how our classes would relate to one another in our API Blueprint and actual API later. Apiary was a testing and debugging tool that we used to build our API safely. After we were done with Apiary we could then use Tastypie to integrate our Django models and API into Entry Resources and URL

configurations so that we could generate local API's that we could work on. Once satisfied with our entire application we can then post it to Heroku to host a our working web application.

Works Cited

- Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
- Jeffries, Ron E.; Anderson, Ann; Hendrickson, Chet. Extreme Programming Installed. Text Book, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA ©2000 .
- Django (Version 1.5) [Computer Software]. (2013). Retrieved from https://djangoproject.com.
- Lindsley, Daniel; Tastypie core team. Tastypie (Version 0.11.0)[Computer Software].(© 2010-2013). Retrieved from http://django-tastypie.readthedocs.org/en/latest/index.html