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 decide on 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 fans) compete against each other to come out on top.

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.

Design

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. 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 since we are 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, but not actually doing any of the work yet. 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 before actually doing the implementation.

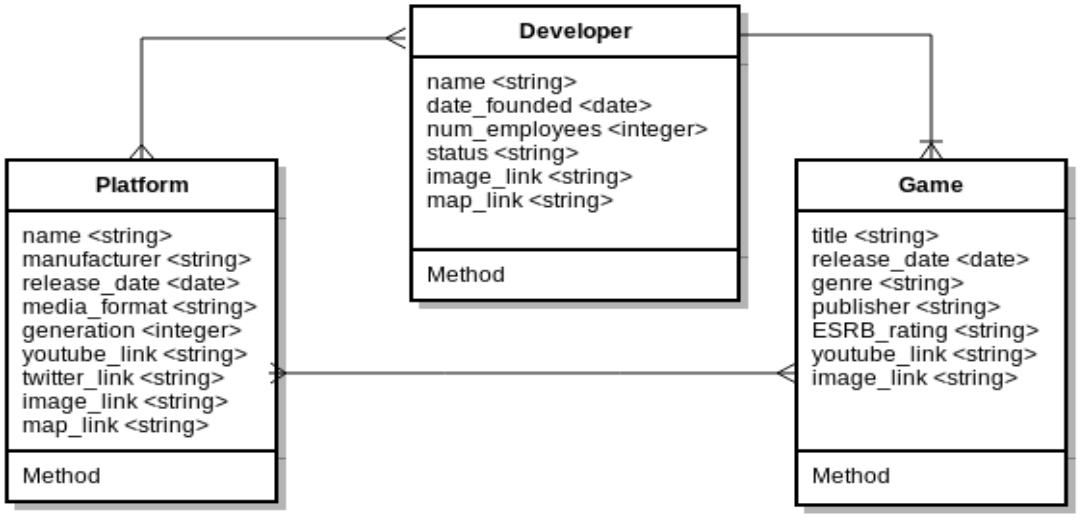

Figure 1

After figuring out REST we then knew why we had to deliberate on our classes and their attributes for our Django Models. The Django Models are pretty much an information source to be used by our API as a source 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 exactly what we were going to want to be our primary key across all classes. Names, and titles in the case of Games, were decided to be the best primary keys. We negotiated what we wanted to be our three class types since originally we had a Publisher class instead of a Platform class, then decided it would get too haphazard with both a Developer class and a Publisher class. So we settled on our three classes and then decided what attributes each class would contain. When we finished writing out our models by hand, we were left an image that resembles Figure 1 above.

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 2. 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 2 below.

Figure 2

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 as it is the only one-to-many relationship in our schema. Although it is possible for a game to have multiple developers working on it, it is not only uncommon but credit for creating the game usually goes to one main developer.

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 - a character field with a maximum length of 255
- Map Link - 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 1.

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 - 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 1.

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 - 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 1.

Each of our Classes 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.

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 expect response. The website also proved to be useful in creating unit tests, but more on that in the next section.

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.

Tests

We are required to write unit tests for our RESTful API which clued us in that we were thinking of it all wrong. Upon further inspection of the matter its seems that our RESTful API will act more as a set of ground rules for the interaction of our classes. We encountered a little bit of this during a brainstorming session the other day in that we had to decide the relations of our classes before actually writing the RESTful API. Apparently, our RESTful API will act as a scaffolding getting built around a building's construction site before the building – our working API – is actually built. We can't just decide after putting up our scaffolding and starting construction that we want to add on another room because there will be no scaffolding to build it from. In the same way we describe with our scaffolding example, we can't just write our RESTful API and then later on decide that the classes don't work and that we need a new class to handle our problem because our RESTful API won't support it. Luckily we saw this early on by talking over our class relations and deciding what we needed.

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

testing each of the API calls that we have already described. 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 (which we did at first), 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.

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. We set down the ground rules for our API by implementing our RESTful API that is the basis for our API structure. In order to think out our RESTful API we first had to reach a final decision on our Django Models as a group.

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.