Internet Video Game Database

Pirates Of Si Valley

Noah Cho, Doug Goldstein, Robert Hammond,

Arash Ghoreyshi, Cristhian Escobar, Richard Lage

Introduction

Video Game Database

Our project is 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.

Group Responsibilities

Every member of the group was required to carry their own weight in the project by accomplishing an assigned tasks in the same way extreme programming assigns stories to each programmer to build value. On top of our individual responsibilities we were expected as a group to work on the following:

- Repository maintenance - All of us have committed to the repository and made suggestions on how to handle issues or posted issues ourselves.
- Model design - We worked on this together in project one before we started with Apiary.
- Website design - We all contributed in many ways to the design of the site throughout all three projects by sharing with one another what we think about its look and functionality.
- Data collection - At one point in project one we all were in the lab looking up data for our static pages that we would later use for the dynamic page. We also each contributed data in later phases of the project.
- The research and understanding of tools used - We have all either used the tools in each phase of the project directly or have pair programed over the shoulder of another partner using the tools while researching their use and advising our "driving" partners.

Individual Responsibilities

We decided, and were required to, pick a group leader for each project to ensure that there was a manager type roll to check that we stay on track and accomplish all requirements from our project specifications. For the first project our leader was Doug Goldstein, followed by Noah Cho for project two, then Arash Ghoreyshi for part three. Outside of our management assignments we found it necessary to allow each member of the group to pick a few of the requirements/stories to ensure that somebody is making sure some part of the total package is completed to the best of our ability as a group. It may seem that we just split up the work, but that is not the case. Since we often worked as a group in the lab and communicated via Facebook messenger while working alone, everyone has asked for advice on their personal responsibilities and given advice or helped with other members responsibilities. Here is how we split up our personal responsibilities for the project:

<u>Doug Goldstein</u>

Doug worked on everything Apiary especially the actual RESTful API and its Tests. He also handled the brunt of our Django Testing as well as the list pages. Cristhian and Doug did most of the work on our Queries page for SQL.

<u>Noah Cho</u>

Noah handled our Twitter Bootstrap coding and later when we brought our API to Heroku he implemented our Tastypie code. Noah was one of the constant coders in our group since if he wasn't actually coding something he would be deep into documentation trying to help another team member.

<u>Arash Ghoreyshi</u>

Arash was invaluable to our group since he was the expert on Django. He focused on a lot of Django code and helping others with it, as well as cranking out a large portion of our dynamic web pages. Arash also handled the Dynamic page on our web site to finish out our acceptance tests.

Cristhian Escobar

Cristhian actually created our UML designs according to UML standard. Cristhian also was the other half to Noah in making sure our Bootstrap code was finished and worked well. With Dougs help Cristhian wrote the Queries page. He also was highly active in all the parts of the project including managerial tasks like getting us to plan time to work in lab.

Robert Hammond

Robert focused mainly on our Technical Report which involves gaining a firm understanding of all that was going on in the project since the report is expected to be a summarization of that. He orchestrated presentation tactics with the team as well as making our Slid.es file.

Richard Lage

Richard was our database subject guru since he has a vast knowledge of video games and everything related. With respect to Extreme Programming he played the role of customer which was essential in our approach to completing the project. He also worked on our Tastypie testing and the JSON file we used for storage. Richard also wrote the SQL that Doug and Cristhian coded.

Approach and Results

The key components to our approach to the project and the results that we saw were Agile development methods, Continuous Integration, and Extreme Programming. We had to adhere to the following guidelines for Continuous Integration: maintain a code repository, have an automated build, make sure our build is self testing, pass the build and tests at each commit. For Continuous Integration we maintained our code repository on Github and pushed/pulled using git commands from our terminal.

Our unit tests and actual code were both automated since we had a makefile to handle their build so we could do so on each commit to make sure everything was working. Many of the other requirements of Continuous Integration were handled by Github like with deliverables being accessible once committed or access to latest builds. A few of the requirements of Extreme Programming overlapped with Continuous Integrations needs so all we had to do extra to adhere to Extreme methods was: fine scale feedback, shared understanding, refactoring, customer availability, turn based integration, test before debugging, and acceptance tests to mention a few. We adhered to fine scale feedback by pair programming as mentioned in group responsibilities. Planning and working as a group as mentioned in our introduction also adds credit to our fine scale feedback. We constantly were editing each other's code over one anothers shoulders, like Doug in figure 1 below.

Figure 1

Richard, to the right of figure 1 in the yellow shirt, played our ever present customer since he knew the most about gaming and gave us the most thorough feedback. Because of our group work we all had a shared understanding of what was going on. The creation of our client end acceptance tests was a requirement for phase three. To be honest testing wasn't always used before finding a bug if it was blaring to the coder or another group member looking over their shoulder, but if our group work safety net failed we would always test. We referred to our lazy shortcut as a fifteen second rule with bad smells in code. Our turn based integration was pretty much a ground rule from the start of phase one since we didn't want the headache of any collisions. In the event that a partner was working on code without the group we would communicate via Facebook Messenger and they would commit outside of the group meets to ensure no collisions. There are twelve principles to the Agile Manifesto, but since most overlap with Continuous Integration and Extreme Programming we will only address the three traits that are unique to Agile. Regular adaptation to changing requirements and circumstances was required for phase two since we had no clue we had to post our API to Heroku until two days before the due date (which was extended one day). Sustainable development was followed by our group since all assignments were turned in before the due date with passing unit tests and working code. Simplicity was followed in particular when we had trouble in phase two with hydration and dehydration of bundles. We wanted to be able to remove api/v1/id from our queries and just use id but ran into problems when we tried hydration. So we decided since it wasn't a requirement, we could save it for an optimization at the end, and everything was already working fine then why do it. In regards to limitations we faced during the project there weren't many. Time constraints had to be our only problem since anything we attempted we were able to finally master, yet obviously we could tweak our application even more if we wanted to.

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 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, POST, and DELETE. These calls return the object, updates the object, place an object in our database server, and delete 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 2 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 2

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 3 below:

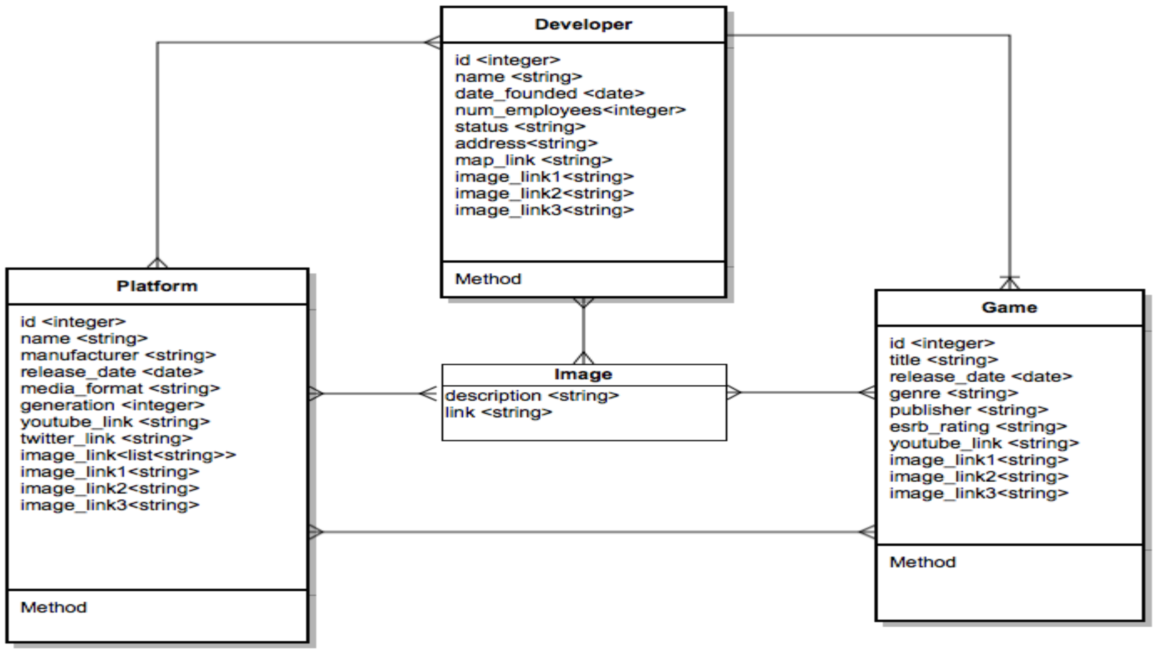

figure 3

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 4. 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 4. 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 Metaclass that places the ordering by primary key and gives us a plural version of the class name for calling the class.

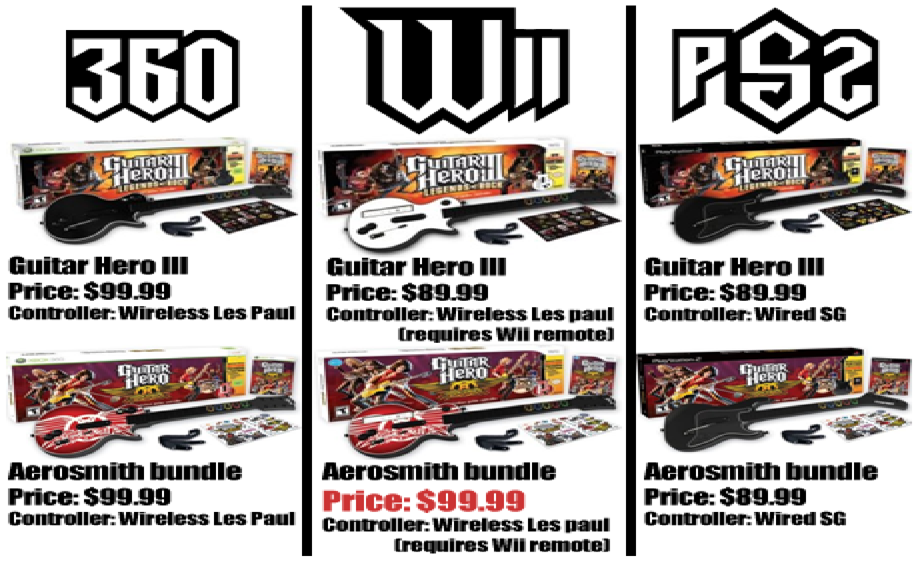

figure 4

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 3.

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 3.

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 3.

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 a 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 5

```
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 5

4. Finally we need to specify our URLconf bits in our API directories urls.py file ours is below in figure 6

```
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 6

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 5 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 5 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 6 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 6 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 6 is our urlpatterns assignment statement where we set up the patterns for home, game, platform, developer, admin, and api.

Design

User Interface and Design Decisions

The hard work coding really paid off in our web-application. The scaling is smooth thanks to Twitter Bootstraps grid utility. We did have a bit of trouble with grid at first but Doug and Noah worked out a solution that looks good on mobile devices too. Our page bar at the top and our pages, the jumbotron, static navigation tables, Dynamic Queries, and SQL pages all adhere to the University of Texas primary/secondary color scheme. The static class pages with lists of links have fitting thumbnails to the left and a nice jump to section alphabet bar atop the links. Regrettably due to corner cases some of our instances of our classes have null data fields since any sort of reason like "we don't have a location on the map because we got shut out of business." But we did test some of these in our SQL queries on our site and thats just the problem with real world objects. To remedy the lack of some data points we instead added more data types to our project like more images, maps, twitter feeds, and others. Since we added more images we didn't want each object of a classes page to become cluttered so now every page starts with a smooth three image carousel that rolls through each picture. Our page includes a dynamic lookup page that unlike the search field it had a series of user interfaces that have broken down all our fields attributes into a list of selections. So all you have to do on our dynamic page is select some traits to see a summary of row objects with those attributes. You can also winnow down the results by using yet another search field that we have next to the table. Our search is detailed and it not only returns objects by the primary key, and substring of that, but it even returns places by address or other attributes in a flash. Our search also has a counter field on top for how many classes match our query. The main button in our interface that isn't a field switch is our learn more button on the splash page inside of our jumbotron that links to the assignment specifications.

Special Features

One of the clever simple features of our web-application is our embedded linkedin profile links from our names. The jumbotron in the splash page has a list of each member and if you click our name it will show our profile for job interviews. Our search <type up exactly how code works/ use and example> And last but not least our carousel that we used from twitter bootstraps JavaScript section helped us to beautifully display each page with all the data.

Frameworks

Twitter Bootstrap

When developing a website it is a pain to develop every page from an empty HTML, CSS, jQuery, or JavaScript page. The industry best practice to solve this problem is to use a web design framework. For this projects concerns that framework is Twitter Bootstrap. Twitter Bootstrap is a front end framework for developing responsive, mobile first projects on the web. Bootstrap has global CSS settings and an advanced grid system for quick use. In our project we used this for buttons, the grid system so our pages scale well, tables for our query results, forms for client tests and search, image utilities were used for our data, and helper classes were used to beautify our sites layout. Bootstrap also has tons of components that we could have used for many things in the site but we mostly used them for thumbnails in our class results and the jumbotron on our splash page. The JavaScript section of Bootstrap also has many uses but we mostly needed the Carousel feature so we could have multiple images without everything looking cluttered. These traits and more are what made our web development fairly fast using Bootstrap but this is only on the front end of our site so in order to develop for our back end we needed to find a new framework that could easily be incorporated to our already present Bootstrap architecture and easily work with Django.

AngularJS

Twitter Bootstrap was great for designing a beautiful user interface, but Bootstrap was the wrong tool when it came to developing our client side acceptance tests. We needed a framework that could declare dynamic views in web-applications. AngularJS was the perfect tool for our belt since not only did it extend our Bootstrap HTML code for our application simply, but it had a quick and expressive development environment. What attributes itself to Angulars speed is the fact that in order to make a dynamic portion of code in jQuery, for example, you would need to specify a command including a function that most likely will be more than three lines. With Angular however you only need the following:

1. Find the part of the code you want to make dynamic
2. Place a ng-model in the tag
3. Name that model anything you want like for example ng-model = "foo"
4. Bind the model to the static data by using {{foo}}! where foo is the name for your model
5. Then bootstrap the application in the headers html tag section with ng-app

So here we are able to do something that takes lines of code in jQuery with just three words and a little over five special characters. AngularJS is even faster since you can just extend the code you already have instead of having to flip through different code segments like jQuery, HTML, JavaScript, or CSS because the Angular is right there in the HTML and you are done. You dont have to see whats going on on two pages since the code on the static page expresses exactly where the dynamic Angular is by the model tags and their bindings. AngularJS also allows you to add control, wire up backends, create components, or embed and inject. For purposes of phase three of our project we needed to create a client side acceptance test to adhere to the Extreme Programing rule of client receiving fast automated tests. The test we ended up writing was just an SQL query that tested these corner cases:

- All video games that do not have an ESRB rating of M, but have another rating
- The video games that appear on more than one platform
- Video games developed by defunct developers
- Which platforms only one developer has worked on
- Developers that created a game during the sixth generation of consoles

In order to implement our client side tests we needed to perform a get on each of our corner cases. What we actually ended up doing was just a big GET that grabbed them all but the actual code is simple and obvious thanks to AngularJS having nice functionality. The Tutorial for Angular actually has an entire section on REST and custom services that a majority of is about controllers like our ng-body block controler at the top of our dynamic.html file. Actually getting Angular to work with Django was simple thanks to the step by step Django-Angular documentation. Django-Angular's documentation has ways to integrate AngularJS with Django, integrate a Django form with an AngularJS model, validate Django forms using AngularJS, CRUD operations, dispatching Ajax requests from Angular controllers, forgery protection, Django URL management for AngularJS, and three way binding.

Other Tools and Processes

We have already mentioned many of the tools used for the project and there are as follows.

- Apiary - For our backend blueprint for the site
- Django - For testing our API in a realistic environment before posting to Heroku
- Tastypie - For bringing our API from Django to the actual web on Heroku
- Twitter Bootstrap - For designing a beautiful web pages on the front end
- AngularJS - For simply integrating backend tests into our HTML code so we can provide client end acceptance tests
- GitHub - For continuous integration, Agile development, and Extreme Programming needs with respect to our repository and its issue tracker
- Git - For pull requests workflow in our terminal
- Facebook Messenger - For planning of meetups and remote correlation
- Python - Language used for actually coding our Django architecture
- RESTful - API style used to make our backend functionality
- UML - Discipline used in developing our classes to keep track of our class relations

Other than these tools and languages we also used Heroku, epydoc, Standard Query Language, Google Docs, and Slid.es. Heroku provided us with the tools needed to extend the application we could build locally on to actually being hosted on a cloud service without worrying about infrastructure. We used Tastypie to actually do this with our code but Heroku was the final destination. All we have do to push to Heroku is

```
$ heroku create ivgdb
Creating ivgdb... done
http://ivgdb.herokuapp.com/ | git@heroku.com:ivgdb.git

$ git push heroku master
----> Heroku receiving push
----> Python app detected
----> Compiled slug size is #.#MB
```

http://ivgdb.herokuapp.com deployed to Heroku

Figure 7

Standard Query Language was used to test some corner cases on the client side of our web-application but we mention that in Testing. We used Google Docs so that we could easily edit the technical report at the same time from a browser anywhere. We could have done it in the repository but instead we just posted a Google Doc url in our readme so that any of the members could pop over to it and make changes at their own leisure. Epydoc was used to generate API documentation for our Django models file models.py before turnin. And finaly Slid.es was a requirement in phase three to make our presentation slides. Slid.es has been good in some places so far but we have had problems with the font functionality since it is very weak and the positioning is kind of weird. Another gripe we had about using Slid.es was that it wont let you manipulate the HTML code even though it seems that you should be able to. Slid.es is a good idea and a needed tool for everyone but it seems like it still needs some work after using it thus far.

Testing

How We Tested

Due to the multiple tools and frameworks we used in this project we needed a few more that just one set of tests to make sure that everything was always working. This is why we had actual client side tests on our website so the client can verify functionality of our backend while accessing it through the front end of the site thanks to AngularJS. Our Unit Tests were mostly used to tests the API on our end while we coded. In the event that we ran into problems while coding to the next commit we could know by running our quick automated unit tests that also showed us runtime of program on top of correctness.

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.

Acceptance Tests

Our performance tests were actually two of the pages on our website. Our Dynamic page and our Queries pages are for the most part our acceptance tests for the client. Extreme Programming requires that we develop easy to use quick acceptance tests for the client to know that their product is working when we deploy the web-application. The first part of our acceptance tests is the Queries page which develops these SQL Queries:

1.
   ```
   select title, esrb_rating
   from idb_game
   where esrb_rating != 'M' and ESRB_rating != 'N/A';
   ```
2.
   ```
   select title, count(*)
    from idb_game inner join idb_game_platforms
   ```

```
 on idb_game.id = idb_game_platforms.game_id
group by idb_game.id
having count(*) > 1;
```

3.

```
select title, genre, publisher
 from idb_game
 where developer_id in
    (select idb_developer.id
        from idb_developer
        where status = 'Defunct');
```

4.

```
select name, manufacturer
 from idb_platform
 where id in
    (select platform_id
        from idb_developer_platforms
          group by platform_id
          having count(platform_id) = 1);
```

5.

```
select name, status
 from idb_developer
 where id in
    (select developer_id
        from idb_developer_platforms
where platform_id in
    (select id
        from idb_platform
        where generation = 6));
```

Each of these Queries is a particular corner case as described on page 23. With these particular queries passing we still need to ensure our customer that the rest of the thing works too so we provided our Dynamic page. The Dynamic page has a tab for each class where we can see all the objects of that class. Our client can further more specify particular queries in our dynamic page by using our class search field that checks against the attributes of each class object. The client also is able to search class object by key attributes with a menu for manufacturers in platforms, and a menu for genres in games. Developers don't have this menu functionality other than the same sorting menu that all three classes use according to size, new to old, old to new, or alphabetical by name.

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. For the final phase we didn't add too much other than the search functionality and our acceptance tests. We did do a bit of beautifying since we had the spare time too but thanks to Twitter Bootstrap and AngularJS that part was a breeze.

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

- MIT. AngularJS (v1.3.0)[Computer Software].(2010-2014). Retrieved from http://docs.angularjs.org/tutorial/step_11
- Rief, Jacob; Read the Docs. Django-Angular (Version latest 4-17-2014)[Software Documentation]. (©2014). Retrieved from http://django-angular.readthedocs.org/en/latest/
- Heroku Inc. Heroku (Version latest)[Computer Software].(©2011) Retrieved from https://www.heroku.com
- University of Texas in Austin. Primary Colors (2014) Retrieved from http://www.utexas.edu/brand-guidelines/brand-identity/primary-colors