PROJECT PROPOSAL: RETRO VIDEO GAMES

TECHNICAL REPORT - IDB.1

GROUP NAME: ITSAFEATURE

TEAM MEMBERS:

- Christopher Word
- Ethan Poulter
- Andrew Solis
- Yousup Lee
- Mitch Stephan
- Andres Lugo-Reyes

http://retro-video-games-373.herokuapp.com/

Note: These are old static pages and no longer exist.

http://retro-video-games-373.herokuapp.com/metroid/

http://retro-video-games-373.herokuapp.com/sonic/

http://retro-video-games-373.herokuapp.com/crash_bandicoot/

http://retro-video-games-373.herokuapp.com/yoshio_sakamoto/

http://retro-video-games-373.herokuapp.com/andy_gavin/

http://retro-video-games-373.herokuapp.com/naoto_oshima/

http://retro-video-games-373.herokuapp.com/nintendo/

http://retro-video-games-373.herokuapp.com/naughtydog/

http://retro-video-games-373.herokuapp.com/sega/

TABLE OF CONTENTS

INTRODUCTION

Retrogaming or “old-school” gaming is the activity of collecting (relatively) older pc, arcade, and console video games and of course, playing them. Video games, as a fairly new digital medium of art, are building a rich and interesting technological history. Video games contribute to many forms of art including painting, writing, music, cinematography, and storytelling. Even the Smithsonian has held exhibits on video games. Furthermore, the development of video games has also contributed to many technological advances. For example, motion sensing technology is advancing at a rapid pace thanks to the influence of research on newly developed consoles and games. The website designed by the ITAFEATURE team aims to preserve relevant data concerning the rich history of video games in a quickly accessible and eventually dynamic manner.

Use cases are created from a user visiting the site. From an archivist interested in information about old games to a collector trying to gauge the rarity of a game, our database provides quick access to obscure data. When a user visits, one could easily be curious to know about the founding of some of the most innovative video game companies like Nintendo. Or, perhaps one might be curious about the reasons behind the rise and fall of Sega. One could even find and compare the overall sales of some of the most popular retro

games like Super Mario or Metroid.

API DESIGN - Description

For the Games (Crisis equivalent) we decided on the following data fields for the API: name, system, release_date, genre, synopsis, copies_sold, images, videos, People, Companies. For the People (People equivalent) we decided on the following data fields for the API: name, DOB, location, job, description, images, Games, Companies. For the Companies (Organization equivalent) we decided on the following data fields for the API: name, founded, location, description, images, maps, external_links, Games, and People. We separated our HTML returns and our JSON returns by prepending /api, similar to Facebook's graph API and the example API. For example: API endpoint for '/people' would be '/api/people'. POST requests are used to create, while PUT requests are used to update endpoints.

We first thought a lot about the kinds of interactions we would want to have with the data. When we first envisioned we decided we would have 3 major elements, games, people, and companies. This means our API will need to supply access to 3 types of things. We also decided for each of the 3 major collection types we would probably want the same interactions. There seemed to be 4 places we would query: the collection as a whole, an individual, and an individuals

intersection with each of the other 2 types of collections. We decided we should have 2 types of responses, brief and verbose. It seemed that responses that involved many values should be brief while the rest should be verbose. When talking to a whole collection we would want to get a list of all members or add something to the collection. When dealing with an individual member we would need to do 3 things: get info, modify the it, or delete it. This breaks down as follows:

API DESIGN - Methods

- collection/ <- interact with the collection as a whole
    - get all - list of brief responses
    - add
- collection/{id} <- interact with an element of the collection
    - get - verbose
    - modify
    - remove
- collection/{id}/otherCollection <- intersection 2 collections
    - get - list of brief responses

API TESTING - Phase 1

To test the Apiary blueprint we created, we used libraries

native to python 3. Thus, no additional libraries need to be downloaded or installed to run our unit tests. The four libraries we used are unittest (http://docs.python.org/3/library/unittest.html), json.dumps to turn the JSON dictionaries into a format that we could send as HTTP requests (http://docs.python.org/3/library/json.html?highlight=json#json.dumps), urllib.request library to facilitate the communication with Apiary (http://docs.python.org/3.0/library/urllib.request.html) and ast.literal_eval to convert the string responces from the HTTP requests into JSON (http://docs.python.org/3/library/ast.html#ast.literal_eval). To test the Apiary blueprint, we sent HTTP requests using urllib.request with JSON data (if applicable) to each endpoint. Then we tested to make sure the the response code and JSON data (if applicable) we received from the HTTP request matched the Apiary blueprint. We ran into an issue with Apiary when we tried making HTTP requests using python 3's urllib.request library. Apiary was receiving plain/text rather than application/json even though we were passing the correct header. However, we were able to use a different service called RequestBin that correctly displayed the JSON and so we concluded that the issue is only with Apiary displaying the JSON it received incorrectly. Because this was the only flaw we noticed in their

system, our unit tests were still able to hit each endpoint and receive the expected response.

API TESTING - Phase 2

For phase 2 we wrote 81 unit tests using the Django test framework. We made sure to test valid input (of every variety) and tested all bad input individually. We did this to ensure that nobody could use our api incorrectly and we could use it to insert, update and delete entities out of our database. Our GET all methods were straightforward as it's pretty hard to fail a get request with no input. We tested our GET by id and GET intersection methods for correct input and all types of bad input. We used the same philosophy for our DELETE methods as we tested a successful delete, and an attempted DELETE at a non-existent game. The vast majority of our tests are focused on our POST and PUT methods. Allowing users to enter bad data can cause our project to become useless. We decided that every object must have an image for representation purposes. We also decided that everything must have a name, and that it not only be white-space. Each type of object has different thresholds of validity depending on the model and we check each.

API IMPLEMENTATION (Phase 2)

When implementing the API we first set up each endpoint in the url file. Then in our view file, we created methods for each endpoint and type of request and merely echoed the request as the response to make sure we understood how to connect the endpoint with a method. We decided that the front-end should not do any DB queries and made it so that they interact with the DB through our api/ calls. This decision was made so that the front-end only depended on our already tested methods and does not communicate with the database at all. We did not want to test both their view end-points and ours. The implementation was relatively straight forward except for the POST and PUTS. The circular dependency nature of games having companies that have people that have games, that have genres, that have systems, etc. added a lot of complexity to our POST and PUT requests. The way our database is designed, the order new objects are created matter a lot. To add a Game a user must first create the related company if it doesn't exist. Next, you must add the people if they don’t exist followed by the genre and system if they don’t exist. Once all these elements are in place then you can add a game. The logic to do this from a single POST on a game was tricky, especially since you can fail at any one of the adds. We try to catch any problems early with our data validation, but rely on the DB throwing

exceptions to handle the hard to validate problems. The PUTs had the same complexity due to validation.

DJANGO MODELS - Description

The first thing we wanted to represent in our model were the three main collections. In our project these are Game (Crisis equivalent), Person (Person equivalent) and Company (Organization equivalent). We analyzed the collections and tried to separate all entries that could be repeated. We created separate tables to represent the data that is repeated such as what system a game is on, jobs a person has, or the genre of a game. We then decided to create a media table, and inherited from it to create an Image and Video table. Because django does not support inheritance we instead had to make them instances of the Media table. The documentation of our models is at /ModelsDoc/index.html

Phase 2

For phase 2 we had to make some minor changes to our models. We discovered that we did not have the skill nor the time to implement the kind of relationship needed to track a persons job over many games. We decided to get around this limitation by having a list of the major players in the description for each game. We also decided

to add twitter feeds to both people and companies. We also changed our maps to be embedded, because we discovered that what we were really doing was just linking to the location. Slightly different but still significant.

WEBPAGE DESIGN - Description

For our game template we created our three static web pages by extending our base.html page, and then placing the data for each of our games into the web page. For our webpages we have a background image that is a repeating tile design of a picture of the game the link is representing. This takes up the full background and is in a fixed position as well so that user's will be able to slide up and down without interfering with the placement of the background image. They are able to see this image in the next 400px right below the navigation bar, which takes up 75px, and then below that is the rest of the webpage with a background color of solid gray background color with the information of each game. For this part of the webpage it is split into three parts column wise across the web page: one for the attributes of the game, one for the links to the companies associated with that game, and one for the people associated with that game.

For the attributes area we have the info from the data model related to each game shown here. This includes the name of the game,

the system it was released on, the release date of the game, genre(s) associated with the game, a small synopsis describing the story of the game, the number of copies sold of the game, links to the images associated with that game, a link to the games gameFAQ web page, and videos associated with that game. We have decided to not include the id associated with the game, as this is more to be used behind the website, such as for database reasons, and feel that this is unnecessary knowledge for the user to see. The next column over contains a header for companies, and displays all companies associated with that game. The names of these companies are themselves links to the web pages for that particular company. The last column of people represents a list of people associated with that particular game, with each name being a link to that person's respective web page as well.

For the companies web pages we again extend the base.html web page and then placing the data for each company into the web page. the top 75 px of the webpage represents a banner for our web site. The next 400 px of the web page represent a tiled image of the company the web page is associated with the image being fixed so that scrolling does not interfere with the placement of the background image. The rest of the bottom of the web page is broken up into three columns: one representing the attributes associated with the company,

one that lists the people associated with the company, and one listing the games that are associated with the company, with padding on the top of each column so that the companies name is shown. For the attributes column we portray data that is pertinent to our companies information. This includes the name of the company, the year the company was founded, the Location of the company's current headquarters, a description of the company’s information, links to images representing the company, external links to the companys website, and an embedded google map whose location is pinpointed to the companies headquarters. The next column over is called people which contains links to the people that are associated with the company. The final column of data is used to list off games associated with the company.

The pages for each person also extend the base.html web page format. The people pages also make use of the three (col-md-4) sized columns that span the webpage. Also, similarly to the pages for Games and Companies, the left-most column for the People pages is the one that displays the individual attributes for that specific person. The attributes (similar, but not identical to those of Games and Companies) are as follows: We have the name of the person (i.e “Yoshio Sakamoto), first and last in a single string, the ID (primary key, implied), the date of birth (DOB, in YYYY-MM-DD) format. This

should be consistent with the DateTimeField used in our models), the location (where they are currently based, i.e, Kyoto), the role they played for the development for each game they are involved in (i.e {{game_id} : “Director”}), the description of the person (a short bio), any image links stored as a collection, any embedded videos, a collection of games that they worked on (indexed by foreign keys), and lastly the company(ies) that they work for in collection format. Unlike the Companies page, we have chosen not to embed the map of the location each person is based in because that just came off as a bit creepy to us.

Two other types of pages we have in our app are the splash/home page and our three different index pages. Unlike the pages for the individual games, companies, and people, these pages do not follow a three column col-md-6 format. The splash page follows a two column col-md-6 format while the index pages as of now have a single col-md-6 column offsetted 5. The left column in our splash page displays the group members’ names while the right column in the splash page holds the links for every unique page in our app (for purposes of phase 1). The index pages (games_index.html, companies_index.html, and people_index.html) have a single column that displays all the pages for the unique thing they are indexing. Every link in the app is currently active, but we plan on shuffling

around our templates and making them more dynamic for our future phases.

WEBPAGE DESIGN - Phase 2

In phase 2, we decided that we wanted to switch to a bootstrap template to make things easier. The template we used is http://startbootstrap.com/landing-page. This enabled us to focus on the back end and api and not worry about creating a design that looked good.

DATABASE - Collection

For each unit of our three class types, game, person, and company, we collected ten unique examples, each linked to the other two classes inside the model. For most units(rows), basic data such as name, location, date-of-birth, and descriptions, was retrieved through wikipedia. Video media data was exclusively retrieved through youtube. Google image search was used for all image data including maps. Certain data links, like the ones for maps, needed to be shortened to be accepted by our PostGreSQL database. To do this, we used tinyurl to shorten links and fit them into the database. To integrate media feeds like twitter, we created a unique feed widget for each individual person and company. Note: Not all people or

companies had media feeds because either a company no longer exists or the person decided to opt out of twitter (good for them).

DATABASE - Configuration

In order to store our information we used Django's manage.py file to create a Sqlite3 database using the command

```
python manage.py syncdb
```

This command effectively uses the compiled version of models.py to create a database based off the classes we defined in our model. For each class defined, Django created a table for it. The specifics of how each table is related to one another is described in the DJANGO MODELS section of the report. After the Database was created we had to find an effective way to populate it given our model's dependencies. In order to actually begin populating our newly created database, we had to first enter the database shell using the command:

```
python manage.py shell
```

This command allowed us to use python code to directly interact with our database. We then had to import all of our tables using:

```
from idb.models.videogames import *
```

Without running the above command every time we enter the shell, we would be dealing with a virgin database.

Some other useful commands were:

ClassName.objects.get(name=’text’) - retrieve an object based off its name

ClassName.objects.values() - display all records in the table for ‘ClassName’

We quickly learned that we had to first insert Company objects considering that all of their attributes consisted of CharFields, UrlFields, and DateTimeFields. To create a Company object, we used the command:

```
Company(name='text', founded='YYYY-MM-DD', twitter=’text,
description='text', location='text', mapimage='text',
webpage='text')
```

After creating the Object, it then had to be saved to the database which was done by appending .save() to the creation of the object. .save() does the job of actually updating the database whether we are creating a new object or updating an existing object

After we created all of the companies, we were then able to create Person objects. Since Person objects had an attribute that had to be assigned to a Company object, we decided it was best to create Persons after Companies. However, it is worth noting that Person objects CAN still be created without companies. This led to incomplete records that would need to be changed later. For the sake of being thorough, we decided to just make all the Company objects

first so that Person objects could be complete on creation. To create a Person object we used a similar syntax as the Company Objects:

```
Person(name='text', DOB='YYYY-MM-DD', description='TEXT',
twitter='text', residence='text').save()
```

Note that the companies field has been ignored so far. This is because the 'companies' attribute is a ManyToManyField. These have to be added with their own special code of the form:

```
Object.ManyToManyFieldName.add(attr1, attr2,
attr3,...,attrN).save()
```

The final staple table we needed to populate was Game. This table was unique in that it contained two ForeignKey attributes, System, and Companies. It is IMPOSSIBLE to create record with these two attributes being set to null. Since Company records had already been created, it was just a matter of creating System objects using:

```
System.objects.create(platform='text').save()
```

We then assigned the ForeignKey attributes to variables so that we can add them to the Game objects on creation:

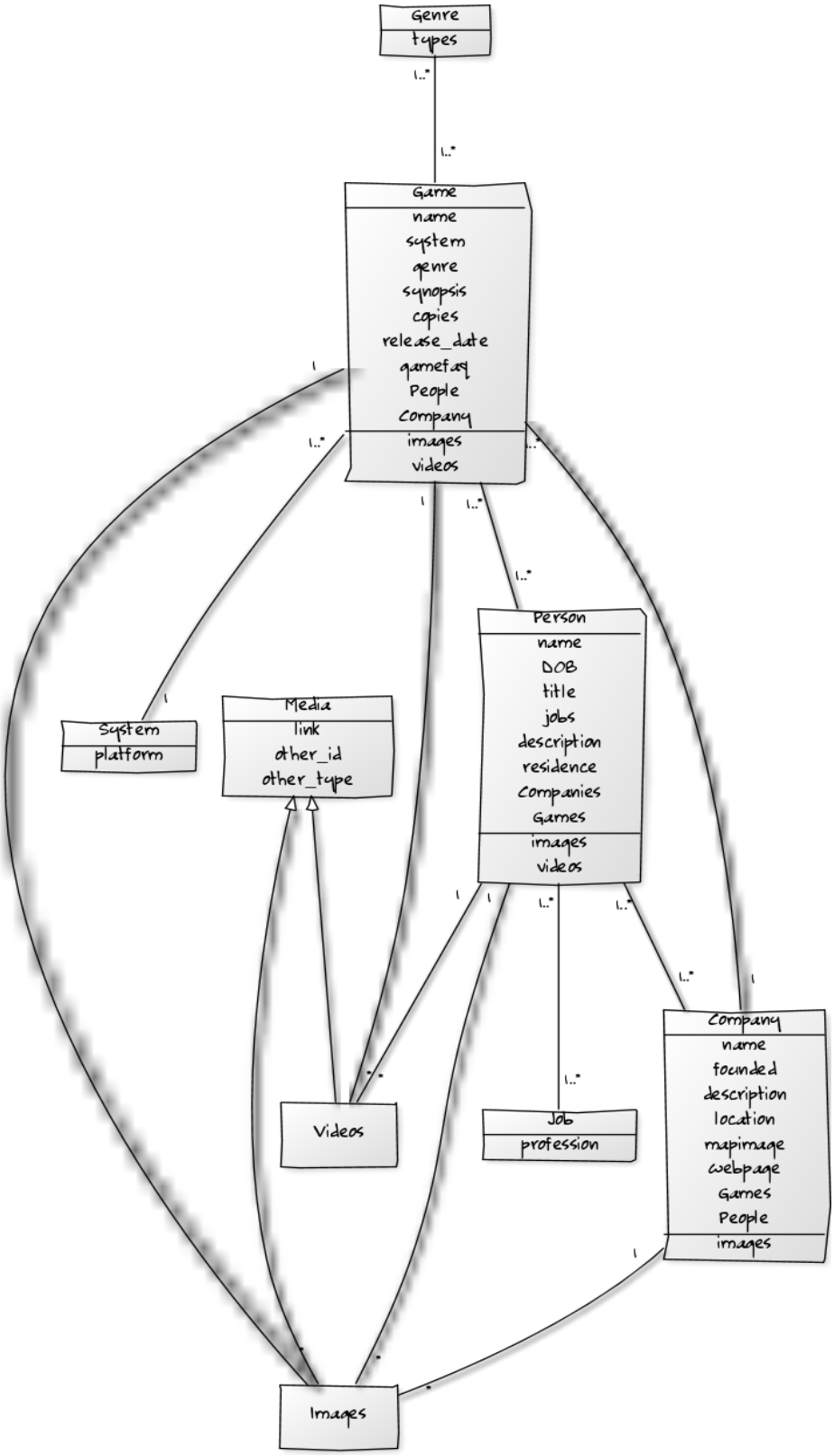
```
Game(name='text', system=System Object, synopsis='text',
copies=int, release_date='YYYY-MM-DD', company=Company Object,
gamefaq='text').save()
```

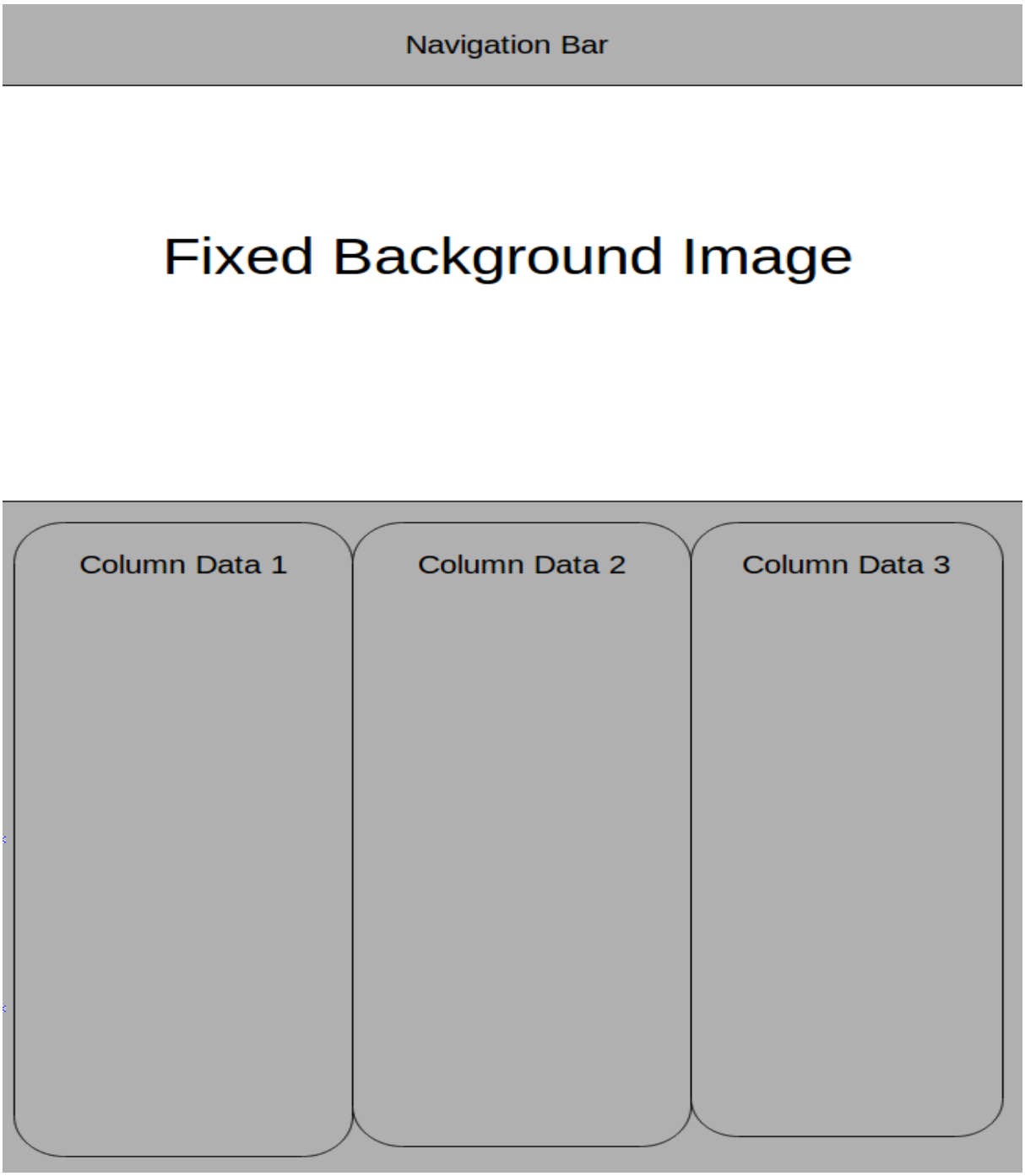
Finally we add the Game objects ManyToManyField, Genre and People in the same manner as when we added companies to Person objects. Genre

objects are created similar to System objects:

```
Genre.objects.create(types='text').save()
```

Calling exit() would leave the shell so that we could then push all of our changes to the database to our git and heroku repositories.


Genre
types
Game
name
system
genre
synopsis
copies
release_date
gamefaq
People
Company
images
videos
System
platform
Media
link
other_id
other_type
Person
name
DOB
title
jobs
description
residence
Companies
Games
images
videos
Videos
Job
profession
Company
name
founded
description
location
mapimage
webpage
Games
People
images
Images


Navigation Bar
Fixed Background Image
Column Data 1
Column Data 2
Column Data 3