gamingDB

Motivation

Video games are a beloved childhood pastime that most people continue to enjoy well into adulthood. As a team, each of us grew up playing video games. Video games not only build friendships but were a great way to experience a rich story with deep characters. There is nothing quite like getting a new game and playing it for the first time. Immersing yourself in a new game is an amazing feeling, but it can be hard to find new video games that you like. We created GamingDB so that people have a place to go to find a new game to play by providing information about everything that comes together to make a video game. From the developers who created the game to the major characters in the game, our site will provide users all the information they need about every aspect of a video game. We will provide information about video games, gaming platforms, major characters, and companies that develop video games. With these models, we will be able to create a great web of information that people can browse through whenever they want to find a new game to play.

Use Cases

We think that anyone looking to find a new game to play will find our site useful. It will also be useful to anyone with an interest in video games. From the home page, users will be able to see not only all recently released games but also be able to see the highest rated video games ever. If they are ever looking for a new game to pick up, they can just consult the home page. If they are interested in a specific game or gaming genre, they will be able to go to the Games page and search for a specific game or filter by a certain game genre. Users will definitely find GamingDB useful when looking for new games to play.

Users will also be able to look up information about individual platforms as well. If they got to the Platforms page, they can see all major platforms ever created. From there, they can click on any platform and see all video games ever created for the given platform. Users will be able to use this information to see what games are available for their consoles.

If users want to find out more information about a certain video game character, they will be able to go to the Characters page where they will find a list of all video game characters. From there, they will be able to click on any character and find out all kind of information about them including all games that they are in, a character summary, and other character details. Users can use this information to find new games to play with their favorite characters.

Video games would not exist without companies spending time and money to develop them. From the Developers page, users can choose a developer and see all the games they have created. From here, users can use this information to find new games to play created by their favorite developers, as well as other developer details.

RESTful API

Based on our use cases mentioned above, we wanted to create simple APIs to allow us to provide the data necessary for each page, while also allowing utility as far as listing all the elements to the users. Our basic GET requests are as follows:

List Games

Get Game

List Characters

Get Character

List Platforms

Get Platform

List Developers

Get Developer

An example of a response to a request:


```
Get Game [GET]

+ Response 200 (application/json)

        {
          "characters": [],
          "description": "Cuphead is a classic run and gun action game heavily
focused on boss battles. Inspired by cartoons of the 1930s, the visuals and
audio are painstakingly created with the same techniques of the era, i.e.
traditional cel animation (hand drawn & hand inked!), watercolor backgrounds,
and original jazz recordings. Play as Cuphead or Mugman (in single player or
co-op) as you traverse strange worlds, acquire new weapons, learn powerful
super moves, and discover hidden secrets. Cuphead is all action, all the
time.",
          "developers": [
            {
              "average_rating": null,
              "description": "StudioMDHR (Studio Moldenhauer) is an
independent video game company founded by two brothers: Chad &amp;amp; Jared
Moldenhauer.\nBased in Oakville, ON / Regina, SK",
              "developer_id": 254,
              "image_url":
"http://images.igdb.com/igdb/image/upload/qrhnhn54o6gepip9oo1y.jpg",
              "location": "United States",
              "name": "Studio MDHR",
```

```
              "website": "http://studiomdhr.com/"
            }
          ],
          "game_id": 13,
          "genres": [],
          "image_url":
"http://images.igdb.com/igdb/image/upload/gtzfjc9pipa6s7v7m68g.jpg",
          "platforms": [
            {
              "description": "Windows XP, the successor to Windows 2000 and
Windows ME, was the first consumer-oriented operating system produced by
Microsoft to be built on the Windows NT kernel. Windows XP was released
worldwide for retail sale on October 25, 2001, and over 400 million copies
were in use in January 2006",
              "image_url":
"http://images.igdb.com/igdb/image/upload/e9w12ei09dljpsiwz7pv.jpg",
              "name": "PC (Microsoft Windows)",
              "platform_id": 4,
              "release_date": "2001-10-25",
              "website": "http://windows.microsoft.com/"
            },
            {
              "description": null,
              "image_url": null,
              "name": "Xbox One",
              "platform_id": 41,
              "release_date": "2013-11-22",
              "website": "http://www.xbox.com/en-US/xbox-one"
            }
          ],
          "rating": 80.7018744810862,
          "related_game_ids": [
            103,
            65,
            45
          ],
          "release_date": "2017-09-29",
          "screenshot_urls": [

"http://images.igdb.com/igdb/image/upload/sngyjwqwzzlciy0ko0sq.jpg",

"http://images.igdb.com/igdb/image/upload/eulkrcocd1zyuvvcv7k0.jpg",

"http://images.igdb.com/igdb/image/upload/ec5tsfhl7wjabwjnsshs.jpg",

"http://images.igdb.com/igdb/image/upload/r9zt66wdgqohmhuukiir.jpg",
```

```
"http://images.igdb.com/igdb/image/upload/sqho6e7tv9verg6j1tvv.jpg"
          ],
          "title": "Cuphead"
        }
```


The purpose of these two specific cases for each model, getting an instance and listing all instances satisfies our use cases. On the main page of a model, Games, for instance, we want to display pages of games to our users. Listing all examples of this with the information provided in each object allows us to display an image of the game, the title of the game, genres associated with the game, platforms the game can be played on, and the companies who made it.

In the case where a user wants to get more information about a game, they would be taken to the unique page of the game filled in with the data from the Get Game request. This request provides us with all of the other attributes about the game. This pattern is followed by the APIs for the other models as well.

Pagination was easily implemented using SQLAlchemy's API manager. One of this objects allows us to specify a table on which we wish to call a GET request using our API and handles that request for us. Using the keyword `results_per_page` we were able to be explicit about the amount of rows that we wanted in our JSON for each page. This instance of the object notation also include the total number of elements, total number of pages, and the current page number, to be used in subsequent requests or other frontend constructs.

We would also like to include filtering. This would allow certain attributes to be taken into account when listing elements, narrowing the search results for the user. This would be included with the request of a given API, only showing results that match the filter passed into the request.

We believe the use of these APIs should allow to provide as much information necessary to the user in a timely manner and is also organized in such a way that it is simple to access whatever is data needed for specific pages.

Models

The models chosen for the database are *Games*, *Companies*, *Platforms*, and *Characters*. These were not arbitrary decisions; a deliberate process was employed in order to narrow down the vast number of possibilities available to make a video game database. Both technical and subjective perspectives were employed in the discussions. Next, a description of this process, the specifics for each of these models and their attributes, and the ways in which they are related to one another will be provided.

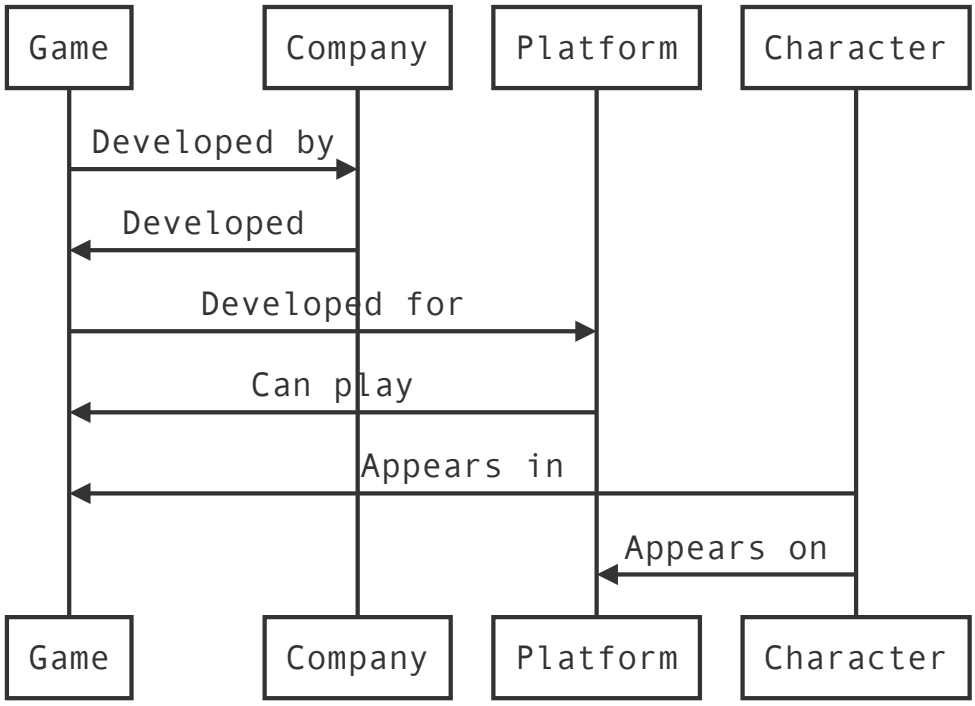
Immediately after the decision to use video games was taken, an ample list of elements present in this ambit was proposed. In addition to the ones that were ultimately picked, locations , genres, voice actors, artistic styles, soundtracks, franchises, critics, DLC, release date, and a surplus of other factors were considered but finally discarded due to the impossibility of getting them reliably from available APIs or the lack of relevance in terms of the connectivity that they would provide to the

relational graph of models. While some of them were completely ignored beyond this point, others were included as attributes of the chosen models. For instance, *genre* became an attribute for the *Game* model, as well as *release date* which was included also included in the *Platform* model. It was noted that this combination would enrich both the density of the graph and the relevance of the nodes. The latter was decided by considering the needs of potential users who would be interested in the contents for different models.

The next step was to decide which attributes to enumerate in each model. As mentioned earlier, some were extrapolated from the previous brainstorming session, but a new process was necessary to achieve the optimal quantity and quality of attributes. Again, a comprehensive list of possibilities was compiled and compared against the available APIs. The set difference between the considered options and the availability of these in the initial APIs was then submitted to another process of scrutiny that checked if it was possible to use alternatives to obtain these attributes. A lack of consistent sources and excessive technical difficulties ruled out most of the options in the set difference. After the list of attributes for each model was definitive, the attributes were analyzed to determine which were the most relevant (i.e. would be displayed promptly to the users) and which were secondary. The final product looks as follows:

Model	**Attributes**
Game	Title , Image, Array of Platforms, Description, Ratings, Array of Developers, Major Characters, Array of Game Engines, Average User Rating, Number of User Ratings, Average Critic Rating, Number of Critic Ratings, Time to beat, Player Perspective, Number of Players, Genre, Release Date, Array of Screenshots, Array of Videos, ESRB Rating, PEGI Rating, Array of DLCS, Array of Expansions, Related Games.
Company	Name, Logo, Developer/Publisher/Both, Location, Highest Rated Game, Description of developer, Games they have produced, Developer Website, Social Media Profile.
Platform	Name of the platform, Image, Website, Summary, Alternative Names, Games on the Platform, Developers on this platform, Release Date.
Character	Name, Image, Game(s), Platform, Company, Species, Gender, Birthday, Description, Enemies, Friends.

Additionally, we can observe how the models are to be connected to the database:



DB

The database for our site is based on PostgreSQL's relational structure. In addition to that, in order to populate our instance (which is hosted in Google Cloud Platform), we used SQLAlchemy in conjuction with Flask. The former is a Python library that allows easy management of SQL tables. In essence, our code creates the Flask app with information to log into our Postgres instance and is used in the construction of an abstract `db` made available by SQLAlchemy. Next, we utilize the class `Model` provided by the library to create our own classes that define the models described above through inheritance. This allows the underlyting implementation of the library to grab these newly defined classes and generate tables containing columns for attributes that we chose during the definition.

After this point, our tables are not necessarily connected with each other. To achieve this connection, and since most of our models are connected in a many-to-many fashion, we use the concept of association tables made available by SQLAlchemy. Essentially, we create one table for each many-to-many relationship between our models and initialize these relationships using the `backref` attribute; the latter ensures that, when a change is made to one of the tables in the relationship, the change is relfected in the table that shares this relationship. For instance, if I add a platform to a game, the game will also be listed in the `.games` attribute of the platform.

Other miscellaneus advantages to this combination of tools motivated our choice. SQLAlchemy for example, handles the auto-incrementation of the `id`s we use as our primary keys. It also handles the creation of RESTful API requests if specified. On the other hand, Postgres allows us to use arrays of elements as attributes in our models, making the implementation of our `screenshot_urls` trivial.

Lastly, when populating our database we follow these steps:

1. Drop any current tables and information in our database in order to avoid conflicts and achieve idempotence in our operations.
2. Create every table specified by our models.
3. Parse the JSONs scraped from our data sources to create rows, at each point checking for relationships and adding those as well.

Getting the data

To populate our database with information we are building json files that have the information from our sources, that are then read into our database. To do this we wrote scripts that will run and scrape relevant information from other sources, and save that formatted data to json files. Once we have downloaded this information, we have it saved so that we don't have to scrape those sources again if we need to reload our database. This allows us to merge and build the information we want into a format that works with our database. We also then don't need to frequently make requests to the other sources, since once we download it once, we have it.

We also scraped the images that we need to display and are hosting them ourselves on Cloudinary. This way we have access to the tools provided by the image hosting site, allowing us to request the images with trasformations already applied. This means if the website simply needs a small thumbnail, it will only need to request a small version of the image instead of downloading the entire large file and then shrinking it. This makes the website faster, and less costly to use.

Issues

Deploying to GCP

When first deploying to GCP we were getting cryptic errors that gave us trouble. We were originally timing out because the database didn't have enough time to build itself successfully. After we adjusted the timeouts accordingly, it worked fine. However, it took time to diagnose.

Differing Image Types

Though we were able to retrieve many images from our APIs, we initially had issues handling .png files. In order to accomodate both file types, we needed to host all of our images on Cloudinary. This allowed us to perform transformations on the file type, so instead of needed to request specifically a .jpeg or .png, we can now always request .jpeg and Cloudinary will accomodate our request. We also gained the functionality of being able to apply other transformations such as height, width, and gravity (where the transformation starts).

API Load Times

We were receiving extended load times which gave our frontend a delay in requesting the information. In order to provide a visual signal to the client, we decided to add a loading animation to ensure it is clear the data is being loaded in. Moving forward, we would like to eliminate this latency entirely to provide a better user experience.

Sorting and Filtering

Both features to organize the data we display start with the API endpoints our backend provides. Flask's API manager and the capability of providing arguments it provides. In essence, we were able to specify multiple ways to filter and organize the responses through JSON objects. For instance, the following request gets every game with a rating between 50 and 100 and orders them in descending order based on rating again:

```
http://gamingdb.info/api/game?q={"filters":
[{"name":"rating","op":"ge","val":50},
{"name":"rating","op":"le","val":100}],"order_by":
[{"field":"rating","direction":"desc"}]}
```

Given the previously mentioned pagination functionallity specified in the creation of our endpoints, these results are also paginated. This is a very simple example, but the amount of filters that we can add is not bounded.

On the frontend, we allow for sorting in ascending in descending order. Every time the sorting or filtering changes, we reconstruct the url to fetch the data from the correct endpoint. Because the backend is paginated, we can just change the page of the contructed url to keep the sorting and filtering the same as you go through the pages for the model.

Search

From a backend perspective, the utilization of Flask's API manager was extremely helpful and time-saving once again. It allows a certain set of arguments to be provided to HTTP requests in order to filter the results when querying a table. Specifically, an example of a query like this would be:

```
http://gamingdb.info/api/game?q={"filters":[{"or":
[{"name":"title","op":"ilike","val":"%Mario Kart%"},
{"name":"description","op":"ilike","val":"%Mario Kart%"},
{"name":"genres","op":"any","val":{"name":"name","op":"ilike","val":"%Mario
Kart%"}}]}]}
```

This request would return a JSON object with all game objects in our database that contain the string `"Mario Kart"` in their title, description, or genre, which are all of the string searchable attributes we have. The percentage signs used inside the strings are a SQL construct to allow zero or more characters.

On the frontend, we dynamically build the filters for each model type depending on the data available for each model. The format for the filter is different for numbers and strings, so we also had to format the filter differently depending on the query. Everytime the query changes, we rebuild the filters for each model, construct the url for the endpoint, and fetch the data. Because the backend is already paginated and split up the search results based on the model, we did not have to further paginate the results.

Hosting

We are using the Google Cloud Platform to host our website. The Google Cloud Platform is a powerful back-end service that has many capabilities useful for running applications. It ranges from just giving you an empty Linux virtual machine where you can run your application however you want, to using something like the App Engine which handles many of the common hurdles you normally have to overcome with a back-end service. Because of the many capabilities the App Engine provides, this is the service we are using to run our web application. It has easy commands for deploying a website, can auto-scale if the website suddenly starts getting a lot of traffic, and provides useful analytics on the requests made to our application.

An additional benefit of the Google Cloud Platform is that since it is a Google service, using it gives us access to many of Google's powerful resources. If needed, we can rent out numerous cores in order to handle large requests in only seconds. Depending on the nature of our application, we can gain access to special hardware that will optimize our runtime. There are also multiple applications such as a visual processing tool that utilize machine learning models Google has trained, allowing us to take advantage of technologies that would normally be fairly difficult to use or gain access to.

Using the service, while seemingly complex due to all the tabs and options, is actually made fairly simple. Once you know which platform you are going to use, you mostly will just interact with the options available for that one. To start with the App Engine, you simply go to the App Engine tab and create a new project. This project will be the workspace for your application, and once created you will be able to see a dashboard with all the relevant information. While there are some ways to interact with your application through the menus, the easiest way to handle things is through the command line shell that exists in the browser. This shell will connect with your current application and essentially gives you control to the machine you are working with. Once there, deploying your application is fairly simple.

The first step to deploying your application is pulling down the code base onto the Google machine. There are ways to set up the Google Cloud SDK on your own device, but handling it all through their interface makes everything fairly simple. Once you have your project's code, the next thing to do is create an app.yaml and a requirements.txt file. These files will specify to the service how to run your application. They have plenty of documentation on how to set this up depending on your application type. In our case, since we are using a Python back-end, we also included a file called appengine_config.py to give additional instructions for loading our Python code. Once you have these files added to the root of your project, all you have to do is run:

```
gcloud app deploy
```

Deploying can take a couple of minutes, but once it has finished your application will now be available at its created domain. By default, Google will generate a URL that includes your project's name, but you can also link your project to a URL that you own in the settings.

One thing we have noticed with deploying changes to code is that sometimes certain files may be cached, and so their changes won't take effect for a while after a deploy.

That is all it takes to get a project up and running using the Google Cloud Platform.

We also needed to get our project to use a custom domain. We purchased gamingdb.info from Google Domains, Google's domain selling service. From there, we went to our Google Cloud Platform console under App Engine Settings and added our custom domain to the project. From there, all we had left to do was configure the DNS correctly so that GCP could generate the SSL certificates and link the domain to our project. To configure the DNS, all we had to do was go to Google Domains and add custom resource records using the information provided by GCP when adding your custom domain. After those couple steps, we had our project up and running on a custom domain.

Tools & Technologies

Front-end

- **Bootstrap**
  - Front-end style framework that supplies pre-built CSS styles for a web application.
- **SASS**
  - "Syntactically Awesome Style Sheets" a scripting language that compiles down to CSS. SASS provides many useful tools that make creating UI's better and easier, like: inheritance, nesting, variables, mixins, and more.
- **HTML**
  - The markup language for creating the DOM for a web page.
- **React**
  - React is a Javascript framework written by Facebook. It allows for dynamic SPA's that can reload components without a re-render on the entire page. It recently came under the MIT open-source license.

Back-end

- **Flask**
  - A microframework for handling rendering and navigation for the web page.
- **SQLAlchemy**
  - Python library that aids in the generation of SQL databases.
- **PostgreSQL**
  - Object-relational database system.
- **Cloudinary**
  - An image hosting website. Provides tools for requesting images with transformations already applied.

Development Tools

- **Git**

    - Git is a version control system that we are using to keep track of our code base and enable parallel workflows.
- **Webpack**
    - Webpack is a bundler for web applications. It allows the application to be brought together into a single bundled Javascript file. This allows much of the deployment and development lifecycle to be taken care of on developer hardware.

Documentation

- **Apiary**
    - We are documenting our applications API using a service called Apiary. Apiary allows us to create a 'blueprint' for our API that is used to roadmap our development and allow others to interact with our service.

Misc

- **Google Cloud Platform**
    - A service that allows for cloud hosting of content with easy to learn setup, rapid releases, and automatic scalability.
- **Slack**
    - A team-based communication tool that can be linked to multiple other services for good collaboration practices.
- **Github**
    - Cloud-based origin host for Git repositories, with a web client for viewing codebases and other things related to Git.
- **Trello**
    - An agile story management tool for keeping track of a backlog and active issues in the project.
- **Google Domains**
    - DNS provider for getting custom domain names.

Datasources

- **IGDB & GiantBomb**
    - Two verbose libraries of videogame-related data formatted to be RESTful API's. They contain a wealth of information about games, platforms, developers, characters, genres, publishers, and much more.

The tools have been very helpful in getting through this first project. Trello allows much greater visualization of issues, comments associated with them, and what stage of completion they are in. Google Cloud Platform has allowed for frictionless integration between our site and our repository and allows for updates to appear seamlessly after they are completed.