eSportGuru
Persia Ghaffari
Thien Vo
Rachel Beale
Eric Liu
Kevin Zhang

Introduction

Over the last decade, the market for competitive gaming has exploded, producing an almost insatiable demand for high quality competitive multiplayer video games and highly skilled players. The competitive gaming scene, coined as “eSports,” has raked over five hundred million dollars in revenue in 2017 alone. Financial news institutions, such as the Business Insider and Forbes, have projected that eSports will become the next multibillion dollar industry within the next two years. The exponential demand for highly competitive gaming has been fueled by new emerging technologies, such as online streaming through Twitch or YouTube.

While the video game industry has always supported competitive local multiplayer options, developers didn’t have a reason to produce highly competitive multiplayer video games with an emphasis on broadcast quality presentation or production, until now. Not only have developers and publishers created these types of video games to meet consumer demands, they’ve also promoted and advertised them through sponsorships. Publishers sponsor big tournaments by adding enormous amounts of prize money as a means to maintain interest of their games, as well as to increase their popularity. Developer sponsorship draws in an enormous amount of people to play and view tournaments, simply because eSports is one of the few activities that anyone can become good in, regardless of background.

Without a doubt, this decade in video games has been be defined by rise and dominance of eSports and eSports culture. However, eSports has also been slowly creeping its way into mainstream media. For instance, ESPN has started broadcasting huge League of Legends, Dota 2, and Hearthstone tournaments, and Disney has been experimenting with broadcasting Super Smash Bros. tournaments. The fact that these companies have been testing eSports viewership and seeing favorable results is proof that eSports will find its place in mainstream entertainment, alongside traditional sports such as football or baseball. The major consequence of this growth is that the general population will be exposed to a complicated world with many different facets, with an overload of information that will pique their interest.

Motivation

There are several databases on the internet devoted to eSports, however they either cover one game or a few major games in depth. For instance, gosugamers.net is a major website dedicated to providing ranking information for the top eight competitive PC games, such as Dota 2, League of Legends, and Hearthstone. While this information is highly useful to those already initiated to these games, it would be very hard for casual viewers to absorb of all their information. These websites are overloaded with statistics and rankings that would mostly make sense to people already familiar with the ranking system.

The goal of eSportGuru is to create a general but universal database for all competitive games. The general information included would be basic information over players, tournaments, teams/sponsors, and the games themselves. While other websites dive into great detail with in-depth statistics over specific matches, teams or players, our site is aimed at presenting data in an aesthetic fashion that, by nature, is wide in scope. What our site would lack in depth would make up for in scope, and this would allow viewers new to eSports to quickly find general data that could serve as a stepping stone in their research over the game and its players. While the developers of eSportGuru would love to add rankings and useful statistics for each player and team, we do not have the resources to do so with the given scope; it would cost too much money to maintain.

Use Cases

Users can query our database through four main models: players, teams, games, and tournaments. If a viewer tunes into a live stream of a prestigious event for the first time they may find themselves asking questions such as: What game are they playing? How popular is that game? Is this tournament a big deal and who's the favorite to win? What's the prize pool and who's on what team? The answer to these questions are not always immediately apparent from the stream itself, so the viewer can make a query on our database to find out.

If the viewer wants to find out more about a tournament, they can simply search for the tournament and will receive information about the winner of the tournament or the current standings, date(s) of the tournament, the game(s) played, as well as what teams and players were involved. These ids in the teams and players are initially foreign keys to players and teams in our database that our API will handle and join multiple tables to provide the user with the information in a simple format that can be easily ingested. For a particular game, the viewer can see when the game was released, who published it, what genre(s) it falls under, and relevant links to the game's website. For a given team, a viewer will receive the acronym for the team, the current roster of the team, and the video game(s) the team is involved in. Lastly, the viewer can query a player and see their full name, hometown, their role (if any), current team (if any), and current video game. Furthermore, all of our models are interconnected. For example, if a viewer queries a player they're interested in, and see that the player has a team, they can immediately click the team to find out more information. This type of interconnectivity can be found throughout the website and increases the ease of access for the user, allowing them to bypass the act of researching. This interconnectivity also ties into the philosophy that esportsGuru should be a very user friendly site to visitors who have little to no eSports experience, allowing the user to access a slew of data with minimum interruptions.

RESTful API

The majority of our preliminary data was scraped from pandascore.co. Their API allowed us to request information from any game, player, team, or tournament, either requesting all of a type of model, or allowing us to request a model based on their ID in the PandaScore database. All of our models can be filtered or sorted by their respective attributes. For example, we can send a request for all players with the hometown "Los Angeles", or we can request all players but sort them alphabetically by first or last name. This same functionality applies to all of the data models. Below is an example of a player entry returned by the PandaScore API:


```
{
    "id": 7659,
    "slug": "q1",
    "name": "q1",
    "first_name": null,
```

```
  "last_name": null,
  "role": null,
  "bio": null,
  "hometown": null,
  "image_url": null,
  "current_team": {
    "id": 583,
    "name": "DAN Gaming",
    "acronym": "DAN",
    "image_url": null
  },
  "current_videogame": {
    "id": 1,
    "name": "LoL",
    "slug": "league-of-legends"
  }
}
```


For the games League of Legends and Dota2 in particular, PandaScore API can also return details of particular matches from a player or team, such as gold earned, heroes or champions selected, and the level and kills of each player in the match. While Pandascore had a plethora of information regarding the professionals performing at the highest echelons of skill of several games, it did not provide enough information about the game itself.

To mediate this drought of information, we turned to another useful resource, igdb.com. This website provided the data to populate our game models. This game database allowed us to search for game statistics based on a given ID, or allowed us to filter games by genre or by rating. The igdb database also provided sorting functionality such as ordering by popularity. It also provided much more in-depth statistics than Pandascore for a given game. For example, for League of Legends we can see the MetaCritic rating, the genre and the description, as well as links to the official website and social media outlets. It also provides links to members of the community currently live streamers, reviews of the game, as well as recommendations to games that are similar. This features were more descriptive and gave a more accurate description of the game to the casual player, which again complimented the philosophy that eSportsGuru should be a site with appeal to the newfound sports viewer, providing a vast ocean of information, lacking depth but wide in scope.

The API we are currently developing will be similar to the PandaScore API. We will be able to provide all of our players, teams, tournaments, or games, as well as the capability to search for specific models by their id. If the user wants to filter by tag or name to find a particular player in a given game, they must use the Get All Players functionality and filter it there, since there can be multiple people across different games with the same name/gamer tag. For example, if there are two players named “kevin”, and one plays League of Legends and the other plays Dota2, then requesting a player named “kevin” through the Get Player functionality would prove problematic and instead the user should request all players through the Get All Players functionality and filter only by those named “kevin”.

Our API will also provide a very handy additional functionality. The concept of interconnectivity is seen through the idea that players or teams or other models have foreign keys to other models. When calling on a player, who is a part of a team, and plays a specific game

or games, a simple call to the DB will return those player properties with the foreign key to other tables. However, we choose to incorporate this table joining into our API and when getting a particular player, the API will also retrieve the team entry in the TEAM table using the FK as a key into the correct table. This allows the frontend to reduce queries and thus run more quickly. This design choice arises from the philosophy that front end should simply receive the correct data and produce it in a digestible and aesthetic format, whereas backend should be handling everything occurring under the hood.

Models

Our database is composed of four main models: players, teams, tournaments, and games. The player has attributes specific to itself, such as name, and hometown, as well as attributes connecting it to other models. For example, the player “Doublelift” has the first name “Yiliang” and last name “Peng”, but also is a part of the team “Team SoloMid”, so his model will contain his attributes unique to himself, and a foreign key to the TSM team model in our database.

Our player model has a private ID as a unique identifier, a name (equivalent to their gamer tag or in game name), a first and last name, a role, a hometown, an image, a current team, and a current video game. The current team and current video game attributes are foreign keys to other models in our database. Our team model has a private ID as a unique identifier, a name, an acronym, an image, a list of players, and a current video game. The players attribute and the current video game attribute are foreign keys to other models in our database. Our tournament model has a private ID as a unique identifier, a name, a beginning and an end date, a video game played, and a list of teams involved. The video game played attribute and the list of teams attribute are foreign keys to other models in our database. Our games model has a private ID as a unique identifier, a name, a developer, a genre, a release date, a website, and a logo. The games model has foreign keys associating it with recent tournaments that have taken place within the game as well as top teams that exist within the game.

We made a decision to only host four games: Overwatch, Hearthstone, League of Legends, and Counter-Strike: Global Offensive. This decision was primarily based on the fact that there was not enough data from other games. Including other games would lead to sparse trees and low amount of interconnectivity. We believed that user experience would be negatively impacted and thus chose not to include sparse games. This decision also stems from the philosophy that showing a bad experience (small, un-populated game) with a good experience (well populated game) is a worse experience than just showing a good experience. One of the reasons that we believe this leads to a bad experience is that a low level of interconnectivity correlates to a low level of immersiveness, and in that way these unpopulated games provide a bad experience.

Database

We are making use of Google’s Compute Engine to host our own mySQL database. The reason we went with mySQL is that it was faster and easier to use, given that all of our information for our four models fits nicely into schemas and well defined structures. Initially, we decided to use GCP’s CloudSQL service, however it ended up being too pricy and we would have quickly run out of the GCP credits. Since money was a factor for us, we decided to use Compute Engine as a cheaper alternative. The trade off was that we had to explicitly handle

our own database. In a real life scenario, that means that we would also have to manage our own security, and perhaps worry about basic techniques like hashing or salting, but that's not an issue for us here currently. Presumably, GCP's CloudSQL would be able to handle all of that for us implicitly. An issue we actually ran into was that some random connection from China was actually taking our database down. We had to write a script to auto reboot the database every time it crashed so that user experience was unaffected. Just to reiterate over the earlier point, had we chosen to go with CloudSQL, this all would have been implicitly done for us. Now, these are things that we have to take into account. If this were a real company, these attacks or disruptions would be a serious concern and we'd have to look into the trade-off of hiring a security expert or just paying GCP to do it for us.

For our database, we have four tables, one for each of our models: PLAYER, TEAM, GAME, and TOURNEY. Each table contains a corresponding model that was listed above. For example, all of the entries in the PLAYER table will all be player models. The player entry itself, will contain the foreign key team and game to the entry in the TEAM and GAME table, respectively. However, the Each of these has properties described in our models stated above. For foreign keys and API calls, instead of returning the entry with the foreign key(s), we choose to substitute those foreign keys for the model.

Search

To setup search, we originally tried going with Google's Custom Search Engine. Since we were generating pages dynamically, we had to provide the CSE with a sitemap and give it a couple of days to process the information on our pages so that it would be able to display results. However, we ran into a few issues. The results were sparse and the search results were extremely slow, ranging anywhere from five to fifteen seconds. We decided that this search would be a negative experience on the user and drive them away from the platform, rather than attract new viewers and provide useful functionality.

On the frontend side, we created a component for the search page and captured the fields that the user was searching for from the search bar. We then dynamically routed their search query to the url /search/query and displayed those results. We displayed the results from our search similar to a Google search, where the title of the entry is the link to its page, and all of its relevant information is right below it for the user to easily access and read. This functionality of displaying contextualization is a big benefit to the target users of our platform, who do not need to go through and check out every entry one at a time, page by page, to absorb information. This allows them to easily process desired information or skim over entries that aren't relevant to their search. We also created a default no results found page to indicate to a user that their query is bad. It could be the case that the user may think that our database simply does not have the information they are searching for, and this case prompted us to distinguish those two scenarios so that the user can be more well informed.

The frontend simply makes a call to our API which searches for the captured query from the user. The backend took this query, and searched through the database using the query as a regex. This search was through all attributes of all the entries of all of our models, but did not include foreign key attributes (which were simply ids or meaningless numbers to the user). The backend then returned a nicely packed JSON, which essentially was a list of lists of models. The first list indexed into four separate other lists, one for each of our models, and those four separate other lists contained all the entries that contained values matching the users search query. The frontend then parsed this data accordingly. Then, four new

components were created as a means of displaying the results, one for each type of model. The components were then created dynamically, as one entry was parsed it was sent to the components which handled that information and visualization. We used a different component for each type of model because our models all had their own unique attributes.

Tools

To set up our website we used a couple of tools. We used React and react-routing, Bootstrap, Apiary, Github, Flask, SQLalchemy, Google Cloud Platform, Slack, Trello, PlanItPoker, and Travis CI. React and Bootstrap were key tools used in allowing us to develop the aesthetic and visual features of the website. Some of the key aspects that were implemented using React and Bootstrap were the navigation bar, which allows users to traverse the website to pages or topics that they are interested in, as well as the carousal on the front page, a circular queue of images that changes on user input. React routing was allowed us to handle the routing for frontend dynamically generate the URLs. Apiary was a tool that allowed us to document the APIs in a presentable and clean fashion. This tool was useful when creating the web scraper, as well as when storing the information from the web scraper into the database. We could see the exact JSON format from what we would receive from the API's we scraped, as well as the exact JSON format that our SQL database required when storing our information. Github was used for version control and on-going development of different features. We made different parts of the website on different branches then proceeded to merge the final product after going through review by at least one teammate. Since everyone had different roles, we would all be working on our own separate development branches, so Github allowed us to manage these separate features and functionalities and eventually combine them into a functioning website. Flask is a micro framework for Python, and handled our routing for backend. Google Cloud Platform was used to host our website and used to store our SQL database.

Slack was used for communication between team members. It also notified us when new branches, pushes, or pull requests were made on Github and also show when something was being done on Trello. Trello was also an organizational tool that allowed us to keep track of projects, whether they were currently on going, waiting for reviewed, or yet to be started. It allowed us to keep our priorities focused and see what features were close to being finished or what features needed more attention. This helped keep us organized and showed who was doing what and what needed to be done. PlanItPoker was a tool used to decide the difficulty in implementing user stories and features for the website. We used this to create stories or consumer experiences that we thought should be a functionality in our website. We then each anonymously voted for the difficulty, ranging anywhere from 0 to 5 in determining the estimated cost of implementing said functionality. This tool allowed us to determine what could be easily added on to the website and what would require more time, which, in turn, logically paved the development pathway and dictated what features could be finished in a timely manner.

Another tool we used was Travis CI, for continuous integration practices. The idea was to have something in place to ensure that new or changed code wouldn't break the website. Initially, the website would be quite easy to check and we could manually tell if anything broke, but as it grew and grew and more features were added on, it would become harder as we couldn't possibly foresee all the possibilities. With Travis CI, we could avoid this inevitable disaster and have a system in place to automatically check for us. Slack, Trello, and PlanItPoker were mainly used to communicate between team members while BootStrap and React

were used for frontend and Apiary and Github were used for back-end and Flask, SQLalchemy, Google Cloud Platform, and Travis CI were mainly used for platform related purposes.

Hosting

We used Google Cloud Platform's App Engine to host our web server. The main choice of going with the App Engine over GCP's Compute Engine was simplicity. Since the web app is not too large, we chose to use App Engine because it gave us quick deployment and we don't have to worry as much about hosting and configuration. The idea is that for App Engine we can simply deploy our code and the platform will do everything else for us, whereas with Compute Engine, we create and configure our own virtual machine instances. Compute engine gives us more flexibility and choice, but it also gives us more responsibility in managing our apps and virtual machines.

However, for hosting our API, we decided to go with GCP's compute engine, as flexibility here was an upside and being able to specify configurations was favorable. However, this meant that we separated our domain into two different subdomains, leading to an individual frontend and backend. This meant that both were essentially two different projects that communicated with each other. Our API makes use of flask and SQLalchemy and runs on an apache2 server. Also noteworthy is that our API doesn't reflect our database directly, as mentioned above. Our API joins multiple tables to retrieve all relevant and related data by substituting foreign keys for corresponding models from other tables. Thus, all the significant data can be retrieved in one query. For example, a call to a tournament would return a list of teams. However, in the database, the list of teams are just a foreign key into the TEAM table, and thus our API will will handle the joining and substitute those foreign keys for TEAM models. This allows our frontend to run faster as it makes less queries.

For our SQL database, we also decided to go with GCP's Compute Engine. Since we have a clear scheme in mind with levels of interconnectivity, we wanted to use an SQL-based database. Using postgres provides more robust table operations that something like mySQL would lack, leading to faster read/write speeds for database warehousing. We also decided to go with two databases, a development and production database. The philosophy behind this was that we valued having a workspace where we could test changes in the development database without negatively affecting the user experience. The consumer can receive these changes once they are polished and aesthetic. The main tradeoff to this approach is that, data in production may become stale or inaccurate. However, the upside is that we can really deliver changes and adapt or tweak until we're satisfied that viewers and consumers will be content with the changes. At the end of the day, the only thing that matters is the user experience, and being able to push changes to production that are more polished is more of a benefit than stale data is a detriment.