December 2015

Scalability In Web APIs

Michael Joseph Perrone
Worcester Polytechnic Institute

Ryan Daniel Baker
Worcester Polytechnic Institute

Worcester Polytechnic Institute

Scalability in Web APIs

Ryan Baker
Mike Perrone

Advised by:
George T. Heineman

1 Introduction

Game developers, especially those that make social games, undertake a large amount of work to create them. There are many technical considerations and ways of building games, from designing a game to deploying the first minimum viable product. Developers often implement a number of generic services within their games, such as friend networks, map editors, achievements, and leaderboards.There is an opportunity for game developers to instead use services (Freeman, 2004).

Students enrolled in WPI's CS 3733 (Software Engineering) course have just this opportunity. They have seven weeks to create a game as a small team using agile development methods, and they do not have time to implement these generic services within their game. In this project, we will design and implement a **leaderboard** service that can be easily integrated into CS 3733 group projects. This will allow the the development teams to focus on their own game while also learning how to integrate with third party web services.

Facebook (Facebook Inc., 2015b) and Playtomic (Lowry, 2013) are two examples of services that currently exist to help game developers in exactly this manner. They provide services via an Application Programming Interface (API) as well as provide code libraries to access this API with methods that send requests to the API. Creating standalone services like these provides an interesting research opportunity for understanding the technologies that will enable them to handle a large number of concurrent requests using the least amount of resources, while also allowing for easy scaling as demand grows.

The goal of this project is to implement a leaderboard service to game developers. The service will allow developers to register a new game, and subsequently allow them to use the service. The service will report on the usage and analytics in an administrator console that is available to the service owners. Most notably, the service will be resilient, easy to debug (as far as we can tell), and easily scalable. At the end of this project, this service will be in production and easily usable for the students of CS 3733 and it would be easy to shift this project available into another context, such as one for consumer use.

2 Background

In this chapter we discuss research we conducted to make informed decisions about how to design and implement our service. We intend to mimic how other platforms provide the services to be consistent with industry standards. This research also includes the problem statement and our solution and design for the features our service offers.

2.1 Problem Statement

Video game developers have a lot of work to do in order to create a game. Typically they are pressed for deadlines to complete required features. This project reduces the effort of game developers by creating a leaderboard service. This will allow the game developers to focus less on some of the peripheral features of their game and instead focus on creating the logic and mechanics of the game. Naturally, the service we offer will be scalable, functional, and extensively tested.

2.2 Game Services and Tools

Many tools, frameworks, and libraries already exist that decrease the amount of work game developers need to put into a game to make it modern and fully featured.

2.2.1 Graphics Engine

Almost every modern game is a graphical game (as opposed to textual games). In order to render objects to the screen, these games have to interface with graphics cards in some way. Graphics engines solve this problem. Most graphical games also have objects that relate to and interact with other objects. A physics engine is typically used to govern these interactions. For example, Unity is a game engine that provides a rendering engine and a physics engine, as well as other useful features for a graphical game like networking capabilities (Unity Technologies, 2015). Unity and other game engines are powerful and provide a high level functionality for building in-game functionality.

2.2.2 Map Editor

Most games will have some kind of terrain or environment where players move around to accomplish some goal, often called a 'map'. The task of building maps can be difficult if it's done by directly writing it into code. Some services have built tools around making this task easier for game developers. For example, Tiled is a graphical editor for building environments for two dimensional video games (Lindeijer, T., 2015). It outputs the description of the map to a standardized XML document, which allows for use in many types of games.

2.2.3 Friend Network

Many multiplayer games will have some way of codifying social relationships between the humans behind players. This usually involves saving pairs or groups of players that enjoy playing with each other, to make it easier for them to play together again. This functionality is usually called a friend or clan network. Some services exist to try to solve this problem. For example, Facebook's Game APIs include ways to invite friends to join in a game (Facebook, Inc., 2015b).

2.2.4 Achievements

Multiplayer and single player games both often have some kind of achievement system. When a player performs a particular action or passes some sort of milestone (sometimes separate from the main narrative goal), the game rewards the player with an in-game or out of game reward in the form of some badge or ability. Playtomic is a service that manages these achievements, providing an API for creation and filtered viewing of achievements (Lowry, 2013).

2.2.5 Leaderboards

Many games that have some sort of scoring or competitive structure also have a way for players to see how well other players are doing. This usually manifests itself as a high score list or a leaderboard that can be viewed in game or in a web browser. Both Playtomic and Facebook provide an API for adding entries to a leaderboard, as well as listing different filtered views of the leaderboard (Facebook, Inc., 2015a; Lowry, 2013).

2.3 Our Service Definition

This section will discuss how we intend to implement leaderboards. For more details on on why we chose to build a leaderboard service, see Appendix A.

2.3.1 Leaderboards

A leaderboard is defined as a set of *entries* for a specific game. An *entry* is typically an association between a user and a score for that game. The user association can simply be a user id or it can contain metadata about the user. Likewise, a score can simply be a primitive representation of a score for a game or it can also contain metadata about the game score.

There are multiple services that already implement leaderboards in a way that help game developers focus on creating the logic and mechanics of their game. In this section, we will discuss how these services implement leaderboards as well as how our features will compare to their services.

Facebook offers many services to game developers to help them with features they may want for their game (Facebook Inc., 2015b). Their services for games range from push notifications to monetization via a subscription service. We are primarily interested in Facebook's *Scores API*. The *Scores API* defines functions for accessing the scores of a game. A developer has read access for:

- A game's scores. Asking for this will return a list of user/score associations.
- A user's score. Asking for this will return a list with the user's score in a sorted list with any friends of the user that also play this game.

A developer has create and update access for:

- A user's score. A user is allowed a single score per game, any request after the initial create is an update for the user's score

A developer has delete access for:

- A user's score. This will remove a user/score association from the leaderboard.
- All scores in a game. This will delete all scores associated with the game.

For more information on Facebook's API for game developers, see Appendix B.

Playtomic is another solution for hosting leaderboards (Lowry, 2013). While Playtomic has many features, we will discuss Playtomic's leaderboards features. Playtomic's leaderboards seem to be much more verbose than Facebook's Scores API. A developer has read access for:

- A game's scores. This has many filters including a player's id, a friend's list, a time span (such as the last 7 days, last 30 days, etc.), pagination (page number and results per page), and a few other filters of that nature

A developer has create access for:

- A user's score. A user is allowed to have multiple scores per game based on an "allow duplicates" flag. An entry on the leaderboard can also have an arbitrary "fields" hashmap, saving things like "character_type" or "level_name" to the score. Playtomic allows filtering on these saved fields when querying the leaderboard.

A developer can also do a create and list function at once, simultaneously posting to the leaderboard and receiving a leaderboard with that score in return. For more information on Playtomic's API, see Appendix C.

For our leaderboard service we have decided to implement a few features from both Facebook and Playtomic, as well as some of our own features. Here is a table that maps the service to the features offered by the service:

Feature	**Playtomic**	**Facebook**	**Our Service**
Read a game's top N scores	☐		☐
Read a user's scores	☐	☐	☐
Scores pagination	☐		☐
Score tags	☐		☐
Score deletion		☐	☐
Multiple scores per user per game	☐		☐
Scores + radius queries			☐
Filter scores with a time granularity	☐		☐

Externally hosted		□	□
Friend Network		□	
Self-hosted	□		

As can be seen from the table, we actively decided against implementing any notion of a friend network, instead leaving that to the client, if they so desire. The reason for this is that many games would already choose to use the existing Facebook social network to represent the friends of the users. We also decided against making the service a self-hosted one. This means that we, as the service owners, would be responsible for any service level agreements of our clients and we would have to develop a subscription based model to fund and support the project, instead of a software licensing model. The following list defines some features whose definitions may not be obvious from the above table:

- top N scores: given a number, N, return the top N scores for this game
- scores pagination: allow scores for a game to be paginated based on a “scores per page” and “page numbers” variables
- score tags: allow a tag to be associated with a leaderboard entry, this may indicate the game’s “level” or other information about a leaderboard entry
- scores + radius queries: given a score’s id and a radius, r, return a list of length 2r+1 that contains r scores above and r scores below the score associated with the specified id

2.4 Service Requirements

We make the following requirements of our leaderboard service.
This section will evaluate these requirements in terms of (a) ease of administration; (b) security of service; and (c) scalability of service.

2.4.1 Administrative Ease

An administrator of this service should be able to modify or restart the service. This section will outline the different features of this service that will make it easy for an administrator of this service to do her job.

Server restarting: The service we use for our cloud servers should enable a fast and simple server restart, should that be required. There are cases where this is necessary, as it will wipe the cache for the server and start the service with a “clean slate” so to say. This could help by closing any bad database connections, dropping any hung client requests, and things of that nature.

Application deployment: The service should be easy to deploy. Given a new version of the codebase it should be easy to drop the code on the server and start the service running in the new version, given that it has the same startup or restart commands.

2.4.2 Security

Clients of this service should not be concerned with how data is handled and the security of the service. Certain things, however, are out of control of the service in terms of security (such as a client storing their API key in a public GitHub repository).

The only concern in terms of security right now is how the API key will be sent from the client to the service in an HTTP request. The most simple solution we found for this is to register the server with an HTTPS certificate. This will allow asymmetric encryption of the HTTPS request. The HTTPS protocol encodes the entire HTTP payload over the encrypted TCP connection, which is where the API key is specified. Putting the API key in the headers of the request will keep the API key from being read by man in the middle attacks.

2.4.3 Scalability

The service should be able to scale to support more traffic as a game’s user base increases. This means that as there are more players in a game, there will naturally be more requests to the service. As our service gets more requests, it should not crash or hang, but should rather scale as needed. This should either be an automatic process (which is something enabled by the cloud computing service provider) or it should be a small amount of requests by an administrator.

Because scalability in web services is a primary focus of this project, we intend to make sure that this service can handle as many requests as possible before having to scale either horizontally (adding more servers to our infrastructure) or vertically (making our current servers able to handle more load). This means that there will be testing done to develop a reasonable estimate for how many requests can be simultaneously handled on a specific server with specific hardware.

2.5 Internal Service Decisions

To build this service, there are two basic things we will need. We need software (an application server) to handle requests and build responses. We need hardware on which to run this software.

2.5.1 Application Framework

We designed the application server more easily using the building blocks of an application framework. Web application frameworks are software structures that abstract many of the low level concerns of making a web application and allow high level programming of web concepts, without which, we would have many hours of labor to write handlers for the HTTP requests. One example of a common feature of web application frameworks is a templating system. Instead of directly writing HTML, many frameworks will allow you to just write a template in an HTML shorthand, give it some arguments, and it will dynamically render the template into HTML using the arguments. Other common features include: caching, URL routing, object-relational mapping, session management, and argument parsing. For more information, see Knupp's blog post on web frameworks (Knupp, 2014).

Some popular web application frameworks include: Django (Python), Ruby on Rails (Ruby), Flask (Python), Bottle (Python), Sinatra (Ruby), CakePHP (PHP), Node.js (JavaScript), Meteor.js (JavaScript), and JSF (Java). There are many more, and the differences between each span larger distances than even the distance between each of the languages in which they are implemented.

2.5.2 Cloud Computing

It used to be the case that if you wanted to run software, you needed to get the hardware yourself. Now the physical computers and most of the administration around maintaining them can be taken care of for you by renting virtual resources with a cloud computing provider. We want to use one of these providers so we don't have to manage the physical hardware ourselves.

Cloud computing removes the need to buy and maintain physical hardware on which to run software. Instead, all of the hardware level management is taken care of by the cloud computing provider, and one just buys virtual computing resources on which to run software. There are two levels of granularity to these cloud computing services: Infrastructure as a Service, and Platform as a Service.

First there is infrastructure as a service (IaaS). IaaS allows users to create and monitor remote resources like servers, storage, and networking. Users can easily add or remove resources with their variable needs. They have a lot of control over the the resources. Popular IaaS providers include Amazon Web Services (AWS), Google Compute Engine, and Windows Azure.

Second is platform as a service (PaaS). PaaS abstracts most parts of the system that IaaS requires explicit management, with the exception of the actual application. PaaS effectively handles all of the specifics of IaaS such as scaling, load balancing and high availability. Popular PaaS providers include AWS Elastic Beanstalk, Heroku, and Google App Engine.

3 Methodology

Starting from the initial design of this project there are important decisions we had to make about how the system would work internally as well as how it would handle external requests. Along with that, there are decisions we made around how we would build this software and the platforms, frameworks, and services we would use to help us. This chapter discusses the decisions we made revolving around how we would theoretically implement the features we needed, and how we implemented those features in practice.

3.1 Decisions of Design and Architecture

This section describes the system architecture and design and how we arrived at those decisions. While it seems broad, this section will focus on more abstract decisions that we faced with how the system should work internally as well as how it should handle external requests.

3.1.1 Leaderboards

To determine how our leaderboard would work, we compared how other services designed their leaderboards as well as which model of leaderboard would be best suited for our use case. We decided to make our leaderboard granularity on the level of allowing multiple scores on the

leaderboard per game per user per tag where a score is defined as an integer. Here is what a simple leaderboard looks like:

http://blog.spivi.com/wp-content/uploads/2014/08/leaderboard_1.jpg

To give an illustration of how a game using our leaderboard might work, let's take Pac-Man as an example game that might use our service. The user would not want to see their score on the granularity of the whole game, because the score for one level might be the top score for that entire level across all users, whereas their entire game score might not make the top 100 scores. The tag for these scores would then map to the different levels that a user could play. Games that only had one level, however, would be able to get around this by only creating scores on the leaderboard with the same tag.

A score is going to be represented as an integer to be able to easily order the scores. Ordering is important because all of the different ways to query the leaderboard will return an ordered list of scores. We considered using a JSON string representation (or really any arbitrary data structure) for scores to allow greater flexibility, but this would make it very difficult to order and compare scores, as the leaderboard would also have to introduce a scoring function to order scores. Instead, the service will work with integers for the scores and if the developer wants their scores to be based on something more complicated, they will have to create some sort of encoding and decoding logic themselves.

Tags are represented as flat strings. We considered having a hierarchy or multiple dimensions of tags. This would allow a client to specify the level, the difficulty, the items equipped, and whatever else they might want to arbitrarily filter on later. Allowing an arbitrary data format for a tag makes it much more difficult to store and query in most databases, and adds a lot of complexity. We decided to stick with the simpler model of flat strings for tags. We also considered making tags an enumeration, having clients define which tags are valid when they

sign up. This imposes another burden of validating the tags on every new score addition. It also would add extra work for clients that want to add new valid tags after scores were already submitted. We decided to trust the clients to take care of validating tags on their end.

We chose not to make the leaderboard score global (that is, accessible across all games) because comparing scores between different games would not make sense. Even if two development teams made the same game, they could try experimenting with a different scoring metric. This would render comparisons between the two games uneven.

For the purpose of leaderboard querying we decided to expose three different ways to get data. The first would be a simple query, asking for the top *N* scores of a game and tag pairing. The *N* value would have a default if no value was passed, as well as a limit if a value that was too large was passed. Second, the client can query the leaderboard with a list of *user_ids*. This would return the *user_id*, *score* tuples for those users in ascending order by score. Having this capability would integrate easily with a friends network and would delegate that responsibility to the game developer. The third way is by asking for adjacent scores when adding a new one. Along with the information for the new score, a client would pass in a *radius* variable. The *radius* describes how many scores would be listed above and below the new score. This query has a maximum radius limit. We decided that an ordinal radius makes sense for this type of querying, rather than a score radius. This ensures that the user knows exactly how many scores they're getting back (2*radius + 1) as well as having the score that was just created be the median score in the returned list. The disadvantage of this approach is that developers would not be able to construct a score-dependent radius based on the game.

We considered whether or not we would keep track of the timestamp of a score. We decided to include time as a factor in our leaderboard. This would allow, externally, for requesting scores on a granularity of time. A client could request the top scores for the past day, month, week, or all time Along with that it would allow, internally, metrics and reporting of the usage of our service and how often and frequently it was used. We decided that this would prove very useful both internally and externally, so we decided to keep track of timestamps.

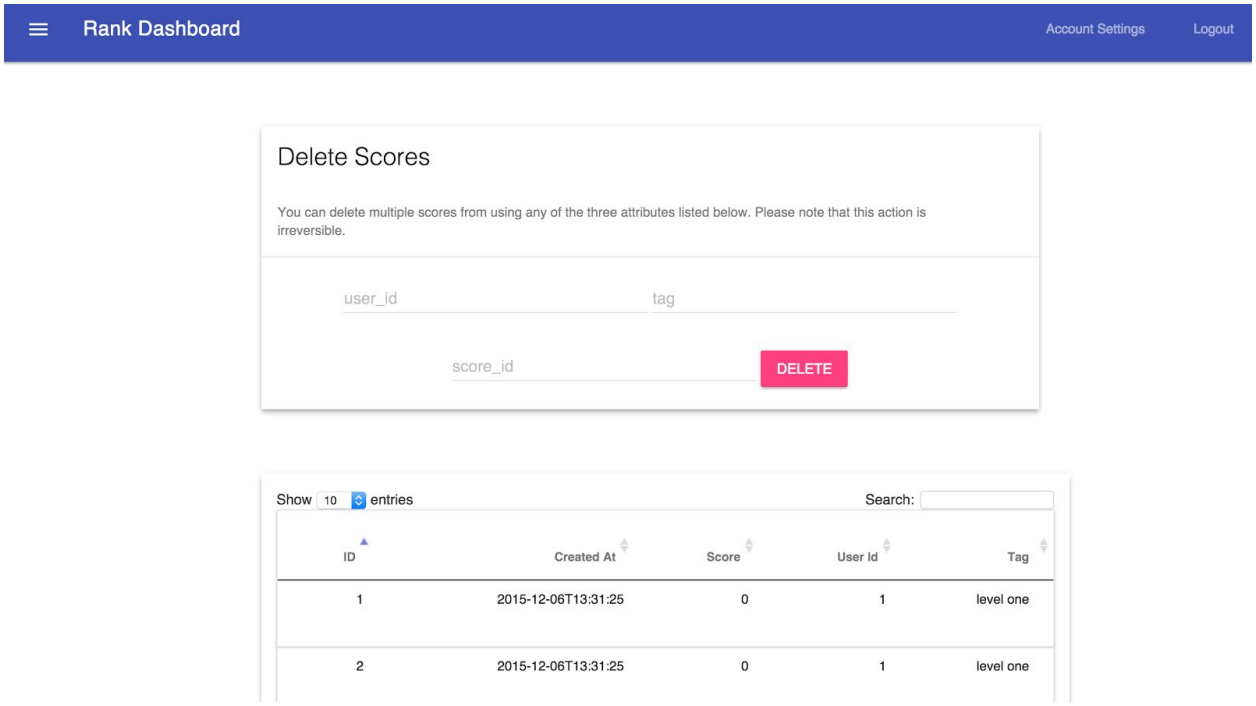
We considered allowing anyone to read any leaderboard. This would make it easier for third parties to make custom leaderboard views outside of the game itself. Unfortunately, there might be some games that don't want that kind of transparency. There might be a game that's focused on weight loss that uses this service, and a player may not want their scores shared with anyone outside of the game, or anyone outside of who the game allows to see it. To get the best of both worlds we considered allowing user or game level permissions. We decided that the added complexity wasn't worth the benefit of the feature, and will simply make everything private to the game.

3.1.2 API Documentation

The purpose of the API documentation (shown here: http://tmwild.com/static/docs/index.html) is to make it easier for developers to integrate with the API and to understand the peculiarities of the interface. We wrote the documentation using the slate template which provides a beautiful view of example code and text in parallel annotating the code. The documentation covers using the HTTP API and using the Java Client Library. It is intended for developers to be able to more easily integrate Rank into their game.

3.1.3 Developer Console

The developer console exists to allow client game developers to register their game. When the developers register themselves, they will receive an API key along with a request limit. This request limit will indicate how many requests they can send over a 24 hour period before becoming throttled. Once they hit the ceiling for that 24 hour period, all incoming requests will be rejected with an HTTP status code of 429 to indicate that the client has sent too many requests (IETF, 2012). The developer console will have three primary views: **overview**, **settings**, and **manage**.



*A screenshot from the **manage** page.*

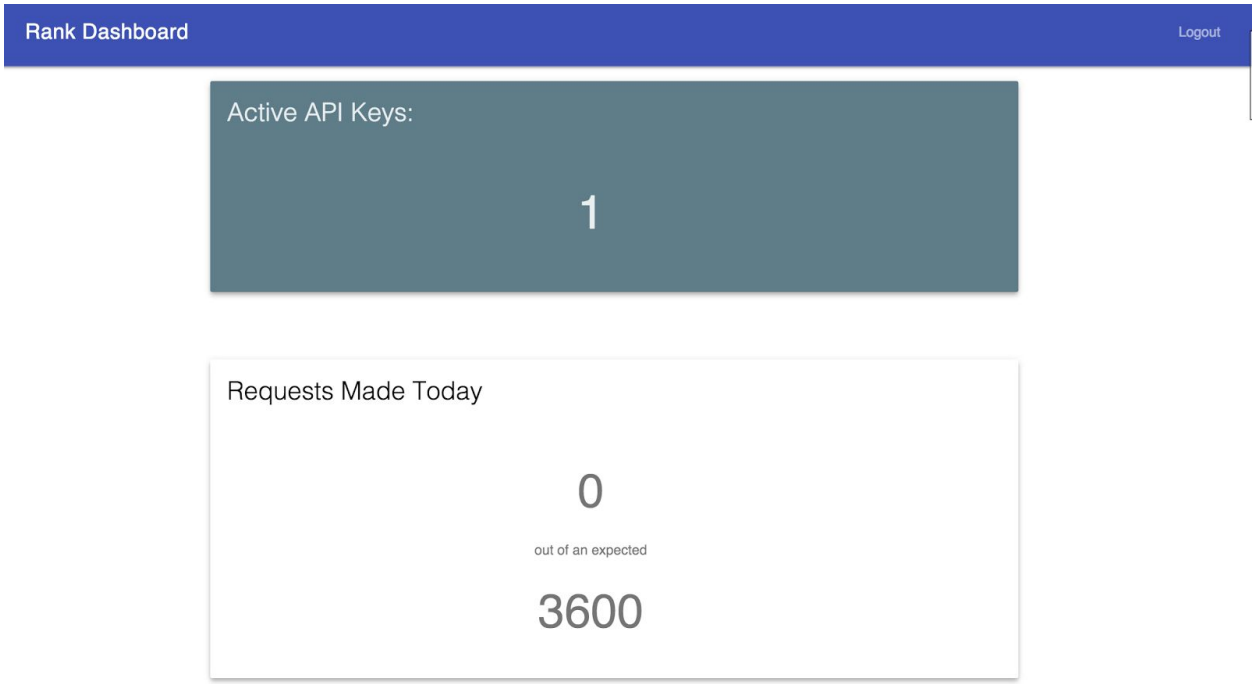
From the **overview**, the developer will be able to see how many requests their games have made in the current day starting at midnight. It will show the requests made compared to their allotted limit, this will include the requests that were not processed due to exceeding their limit. The developer will also be able to see the requests made over the period of the day starting at midnight and see the requests made every day over the past week in a line graph.

From the **settings** view the developer will be able to invalidate an API key and request a new one. This can be necessary in the case of the API key being compromised or obtained by some malicious third party. This will render the API key to be invalid for **all** requests and only allow the developer requests using the new API key that will be supplied to them. In addition to this, the developer will also be able to close their account from this console. This means that the API key will be invalidated and they will be unable to request a new one. Since the account is closed, all data that belongs to that account will be unable to be reached by any external user and it can be deleted. From this view the developer will also be able to close their account. This would mean that all record of the client would be eradicated from the application.

From the **manage** view the developer will be able to view and delete specific entries based on different things. All entries in their database will be exposed in a table. In this table they will be able to delete entries based on a `tag`, `user_id`, or `entry_id`. This will be useful for developers that are testing from a single user account, or developers that test with a tag called “test” or something of that nature. This will also be useful if, for example, a user found some vulnerability in their game and that user’s scores needed to be deleted. The developer will also be able to see a graph of the requests from their API key over different time granularities.

3.1.4 Admin Console

The admin console presents analytical insights to the service. An admin in this sense is an owner of this service. The admin console has three views: **overview**, **manage clients**, and **metrics**.



*A screenshot from the **overview** page.*

The **overview** exposes to the admin how many API keys are currently active. The overview page will also show how many requests have been made in respect to the total expected amount based on the throttling limit of all the users together. This allows the service administrator to evaluate what is needed in terms of server resources.

The **manage clients** view exposes the admin insights to the different users and allows the admin to manage the users. One of the features of managing the users include “freezing” their account. This means that the selected client cannot make requests with their API key, returning a 403 (Forbidden) HTTP status response instead of processing the request and returning any normally expected response. The admin console will also allow an admin to “impersonate” a client. This means that the admin will be able to see the client’s developer console and have just as many permissions as the game developer would have.

The **metrics** view shows the currently logged in admin how many requests have been made in the past week. This line graph will show an aggregation of data over all the clients of the service. There will also be a link to Amazon Web Services’ CloudWatch feature. In the AWS console this feature shows resource usage of the servers that are currently being used for the service. This will allow the service admin to see how the infrastructure is doing in terms of CPU and memory usage.

3.1.5 Java Client Package

We planned to integrate the Rank API into an example game that is written in Java. The Software Engineering students who will be using the API will similarly integrate the Rank API into their Java games. We can significantly reduce the difficulty of that task by creating a Rank Java Client Library. This library uses Java objects to wrap each of the API calls. It allows developers to work directly with Score objects instead of having to deal with making HTTP requests and parsing JSON responses.

The only major decision we made in the process was which Java HTTP request library to use. Apache's library and Google's library seemed to be the most popular. We chose to use Google's library because the example code seemed more simple. Other than that, creating the library was a matter of having methods cover all of the endpoints.

There are detailed instructions on how to use the library in the API documentation: http://tmwild.com/static/docs/index.html

3.1.6 Logging

Due to the nature of the developer console and the admin console, we needed some way to inform the users and system administrators of their usage and general usage of the service. To do this we built a logger that reports metrics of usage at both an infrastructure level and an application level.

We decided that we needed logging because we needed a way to provide metrics of usage and system health to both the service administrators and the service users. We need to expose usage to our users to show them how close they are to surpassing their daily allotment of requests. We, as service administrators, need to see how many users there are, how many requests are coming in, and how our hardware resources are handling user traffic.

We decided to log every request both to a database and a local file. The log files are rotated on a daily basis. Every request that is logged contains the following information:

- timestamp
- http verb
- url route
- response code
- game id

A link to AWS CloudWatch is also supplied. CloudWatch is a service that reports on the health of any AWS resource, including servers and databases.

3.2 Decisions of Implementation

Whereas the previous section discussed how the service would work at an abstract level, this section will detail how the service works on a lower level and how we decided to implement the features we decided on having in the previous section.

3.2.1 Enterprise vs Public

Facebook's game API was centralized; everyone using it sent requests to the same set of servers. Playtomic's API was a specialized enterprise solution, meaning that each game development group has to host a copy of the API on their own servers. We had to choose between an enterprise solution or the public solution.

The enterprise solution is better because it allows the service to be contained in the user's internal network, isolated from the rest of the Internet. It also allows the user to have much more control over uptime of the service, and it makes it impossible for the service creator to do anything nefarious with the data given to the service, as the service creator doesn't have access to the data. It also creates some additional overhead for version upgrades, requiring the enterprise customer to download the new version and redeploy with that upgraded version. This could have serious implications for how frequently bugs could be patched.

The public solution is better for a simpler client in that it doesn't require any overhead or system administration for users. Users just use it directly. This requires less work for the service developers. This also allows faster iteration because the development teams don’t have to worry about maintenance of the service.

Since we don't think that our service will have incredibly sensitive data that game developers wouldn't want us to be able to theoretically access, and it will take more time than we have to complete an enterprise solution, we are going with the public solution.

3.2.2 Front End Implementation

The front end of the application is composed of multiple components: the API documentation (shown here: http://tmwild.com/static/docs/index.html), the service admin console, and the developer console. These three components have multiple libraries to help implement them. The libraries and other details are discussed in the following paragraph.

The API documentation (shown here: http://tmwild.com/static/docs/index.html) is served as a static page that is compiled from written HTML files that are compiled by Slate (Lord, R., 2013). The static pages and static assets that make up the developer and administration consoles are served from Flask. The HTML pages are styled using Material Design Lite (Google, 2015c). All of the JavaScript is written first in Coffeescript then converted to JavaScript (Coffeescript, 2015). This is done through a process of transpiling, whereby the Coffeescript code is transpiled into JavaScript so the browser can run it. JQuery helps developers to manipulate the DOM of the web page by exposing utility functions that make DOM related functions easier (The jQuery Foundation, 2015). We use DataTables for rendering easily searchable tables. DataTables helps by easily styling and exposing functionality for HTML tables (Spry Media Ltd, 2015).

The application framework we used came pre-loaded with the bootstrap library. Bootstrap is a user interface library that is very standard among many websites that are created in today's software climate (Twitter, 2015). The developer and admin consoles use a different user interface library, however. This is because we wanted the user experiences to feel different. While we felt the home and about pages should feel like a normal website, we wanted the console to have a stark difference and feel more like a web application. For this reason we used Material Design Lite, which is an implementation of the Material Design Specification (Google, 2015b). Material Design is a design specification intended to make the elements of the web page look and feel like "material". Many of the specifications in the document include rules around implementation that seem very intuitive, e.g. material casts shadows, material cannot pass through each other, etc. The design specification is currently under active development by Google.

3.2.3 Cloud Computing Provider (AWS)

We considered two popular cloud computing services: Amazon Web Services, and Digital Ocean. Both are sufficient to be able to build the API, but they differ in level of control and abstraction. Digital Ocean allows the user to make machine images, deploy those images to a virtual server, manage domain names, and manage IP addresses. AWS allows all of those things and more. It allows for specialized database servers to be spun up with little configuration. AWS also provides easy load balancing, automatic scaling, a hugely scalable key value store, and server metrics, all of which would have to be set up manually with Digital Ocean.

Each of those services is necessary for the Rank API, so since AWS takes away the burden of managing or developing those key services, we decided to go with AWS. We initially started using just AWS's Elastic Beanstalk service, but decided it wasn't the best option for our use case and time restraints. For more details on this decision, see Appendix F.

3.2.4 Web Application Framework Implementation (Flask)

With the multitude of web application frameworks available, it is nearly impossible to understand all of them and compare them to make a decision on which is right for a particular task. We narrowed the search space to a few that we personally know are being used in industry production software: Ruby on Rails, Django, Flask, and Sinatra (Hamlett, 2015; Twitter, 2011). Of these we chose Flask. This analysis is necessarily shallow, as the differences between even these four frameworks could be a considerable research project itself.

Rails is a heavy framework built on top of Ruby that emphasises convention-over-configuration; it does a lot of things for the user behind the scenes on its own (Hansson, 2015). It has a model-view-controller structure, which allows for a clean separation between the application data and its presentation.

Django is written in Python, is similarly as heavy as Rails, but emphasises explicitness over implicitness (Pires, 2014; Makai, 2015). This means spelling out all of the configurations that Rails has by default. It, too, uses the model-view-controller structure.

Flask is a light microframework in Python. To get all of the features of Django or Rails, it requires extensions. Fortunately, Flask has an active extension ecosystem (Hamlett, 2015). For smaller projects, it is much easier to hit the ground running with Flask than Django or Rails. Sinatra is a lightweight DSL on top of Ruby for creating web apps. It, too, has many extensions. Sinatra is very similar to Flask in that it is also easy to develop small projects (Jones, 2012).

Between these options, we have chosen Flask. We don't think the leaderboards API will be complicated enough to warrant using the more powerful Django or Rails frameworks. Between Flask and Sinatra, the choice isn't clear. Since we both have much more experience in Python than Ruby, we chose to go with Flask.

3.2.5 Continuous Integration Service

For our continuous integration (CI) service we have narrowed the choice down to be either Travis CI, or Drone(drone.io, 2013; Travis CI, Gmbh, 2015b). We chose these because we did not want to focus too much effort on our CI tool, because we can instead spend that time to develop the core functionality of the service, which we find to be more important.

We chose to use Travis CI for a couple reasons. The main reason we decided to use Travis CI was because of its Build Matrix feature (Travis CI, Gmbh, 2015a). This feature enables the CI build to be run for many combinations of different versions of dependencies. For example, we

could specify to build with Python versions 2.5, 2.6, and 2.7 along with Flask versions 0.9.0, 0.10.0, and 0.10.2 and it would run the build with every combination. In a Quora posting, the primary author of drone.io indicated numerous features that drone.io and Travis CI share, as well as some features in which they differ. As of the posting (November 2014) and the writing of this document (September 2015), Travis CI has more features that are relevant and useful for this project (Rydzewski, B., 2014). Our current build runs on Travis CI even though we don't currently use it to the full extent.

3.2.6 API

When building APIs, it is typically regarded as good practice to make a RESTful API (Searchsoa, 2015). For the actions on our API, however, we felt the best way to represent them was without following REST. One requirement of a RESTful API is that its actions on its resources are informed by HTTP verbs on its universal resource identifiers (URIs). For example, retrieving a list of scores would be the HTTP verb GET and the URI might be /scores. In our case, we separate our leaderboard resource into a few URIs based on their actions and only allow specific HTTP verbs to be used on them (attempted use of other verbs will return a 405, method not allowed error). Here are our routes and what they do:

HTTP Verb	URI	Action
GET	/api/leaderboards	This URI returns a leaderboard for the game specified by the API key used in the request. This URI takes multiple different parameters and filters the leaderboard on them.
POST	/api/add_score	This URI adds a new score to the database.
POST	/api/add_score_and_list	This URI adds a new score and returns a list of the new score with a radius of scores around it.
GET	/status	This URI returns the status of the API and the current version.

These are the only endpoints that are exposed in the API. Since the API was small (there was only one resource) we found it appropriate to not follow RESTful recommendations. For more information on the API endpoints and documentation, take a look at our API documentation page: http://tmwild.com/static/docs/index.html

To ensure a request is coming from an authorized user, the service checks the headers on each incoming request. The header the service looks for is "Authorization" and the value of the

header must be “Bearer<space>“ then the client’s API key. If the header is not present, the request is rejected with an HTTP 401 code. If the API key is invalid, the request is also rejected with an HTTP 401 code. If the header is present and the API key correctly maps to a client in our database, but the client has used their allotted amount of requests for that day, then the request is rejected with an HTTP 429 (too many requests) code.

To send requests to and from the server, there should be some standard method of communication and message encoding. We chose to use JSON because it’s currently adopted as the industry standard and most languages and web frameworks (including Flask) have libraries to handle interacting with JSON (JSON, 2015). All API endpoints in the service accept JSON in the request data and return JSON in the response data.

3.2.7 Logging

Since we used Python with the Flask framework, we had to understand best practices for logging with that technology stack. As it turns out, Flask has no special logger; it uses Python’s logging under the hood. We created some new Python loggers that record the details of the requests we want to keep track of and registered them to log after every request. They log to a local file that is rotated every midnight and also to the database for the purpose of reporting aggregate statistics.

The Python loggers have four components: the Logger, the Handler, the Filter, and the Formatter (The Python Software Foundation, 2015). The service uses no logger, instead it attaches a handler to Flask’s logger so that on every log, the log record will be sent to the service’s handler. The service’s handler will send the logs to the file “logs/<environment>_log.json” where the “environment” is either “dev” or “prod”. Every midnight, the log will be rotated out, marked with the current date and replaced by a new log file for the new day’s requests. The service attached a filter to this handler that filters out (refuses to log) any request that did not hit any route path that is prefixed with “/api”. After those requests are filtered out, the logger passes the log record to the formatter. The formatter will format the log record by passing parameters into an ordered dictionary (an ordered hash map) and convert that to JSON. During this process, the formatter will also create a new instance of a UserRequest class. The UserRequest is used to measure how many requests have been sent by a specific client, for the purpose of rejecting the requests after the allotted amount of requests in the 24-hour period has been exceeded. The log record’s message is then set to the JSONified version of the ordered dictionary. The log record is then passed to the parent formatter’s “format” method, so the record’s message can be logged to the local file.

3.2.8 Database Schema

The database has a limited number of tables, since the service is not currently complex. The database tables consist of `Users`, `Scores`, `Games`, and `UserRequests`. The `User` table consists of all the users that sign up for the service, including the service administrators. The fields include an `ID`, a `username`, a (hashed) `password`, a `created_at` date, an `is_admin` boolean, and a `game_id` foreign key. The `Score` table consists of all the scores created by any game (or user). A score instance consists of a `user_id` (not a user's foreign key), a `score`, a `tag`, a `created_at` date, and a `game_id` foreign key. The `Game` table consists of all the games associated with every user (1-to-1 relation, created every time a new user is created). A single game instance consists of an `api_key` and a `frozen` boolean, which determines whether or not the `api_key` can be used to make any requests on behalf of that game. Finally, the `UserRequests` table is made up of all requests to the API from any game's `api_key`. A `UserRequest` instance consists of a `time_requested` date, a `game_id` foreign key, an `http_verb`, a `uri`, and a `status`. All of these fields are the same fields that are logged on each request.

4 Success Metrics

We'll be evaluating this service from two different perspectives: resiliency and design. Resiliency testing will deal with simulated traffic, system scalability, and how it handles load, among other things. Design, on the other hand, deals with how easily this service can be used and how easily the service can be maintained as needed.

4.1 Resiliency

While it's important that the service works, it's arguably equally important that the service is robust in the face of a large influx of traffic, while requiring little manual intervention from any system administrator. This section will discuss how well the service does when evaluated in terms of how resilient and robust it is. The two topics of resiliency discussed are the simulated traffic of the system and load testing & scalability. Both of these criteria are defined in their respective sections.

4.1.1 Simulated Traffic

After a simple client integration for a Java game, we found that, on average, 35 requests were made per hour by a single user. Based on our estimates, we can handle roughly 780 of these users simultaneously on a single server instance. Based on these metrics, we can safely conclude that any game developer (with an API key) will run out of requests well before their expected 24 hour request period counter is reset. After a couple of minutes of this much traffic (discussed more in detail in sections 4.1.2 and 4.2.3), the Amazon Web Services auto-scaling policy will take effect and create another server instance and put that server instance behind the Amazon Web Services' Elastic Load Balancer (ELB). While it is important to decide on a good balance of time between high CPU load on a server and creating a new server instance, we unfortunately did not have enough time to thoroughly investigate what that ideal time (for our service) would be.

The simulated (single user) traffic for this is based on a naïve implementation of a Rank service integration for a Java game. The Java game sends a request to the Rank service to create a new score after a game has been finished. It also has a scoreboard open that will update the scores every few minutes. Ideally, a new score would be added to the leaderboard upon a win condition when a user finishes a game, and not a lose condition (for most games, anyway). This would decrease the amount of requests per hour that are being sent to the service. Another way to decrease the amount of requests sent to the service would be to only display the scoreboard after a game is finished or if a user specifically asks to see the game's scoreboard.

4.1.2 Load Testing and Scalability

We tested the capacity and scalability of the Rank service by hitting it with a large number of concurrent users and monitoring the behavior. We used Locust (http://locust.io/), an open source load testing tool, to simulate many concurrent users.

We created a simple model of a user, who starts by requesting a filtered score list, posting a new score, and listing again. The user waits randomly between 5 and 10 seconds between each action sequence, then repeats. Then we slowly scaled up the number of users until a request failed. We defined a request failure either as returning a 500 error, or by taking more than 1 second to return a response. Here is the data from that load testing:

servers	1	2	4	8	16	...
failed after # concurrent users	781	1532	3174	6518	13492	...

Second, we tested the scalability of the system by measuring the **time to action** of the autoscaling service. As above, we hit the service with many concurrent simulated users, and recorded when a request failed. We then monitored the system to see how long after the failure the system would auto-scale itself to be able to handle the increased load. The average time to action was 136 seconds.

This means that the system will scale itself with no human interaction within 4 minutes of a problematic load. Since this is automatic and relatively quickly self-repairing, we consider our autoscaling endeavor a success.

4.2 Design

It's important that our service is functional. It's also very important that the functionality of the service is easy to use and understand. This section will evaluate how well designed the service is. First, in terms of a client perspective, to see how easy it is to integrate the service into an existing Java-based game with the Java client as well as how understandable the developer console is on the web application. Second, in terms of an administrator perspective. It is important to determine how easy it is to deploy and scale the application, as well as how understandable the admin console is.

4.2.1 Client Perspective

We wanted it to be very easy for a client (game developer) to add the use of our API to their game, and manage their account through the developer console. To evaluate this goal, we integrated our API with a game. We documented the troubles we had, and this was a rough qualitative assessment for success or failure.

Since one member of our team worked alone to develop the Java client package, it made sense that the other member followed the documentation provided to integrate the package into the standalone Java project. In following the documentation that was provided, we found that we were missing a couple details on how to get the project working well with the Java client. We also found and fixed a few small problems with the Java client to make sure that the user of the Java client would better understand the errors they were receiving.

We also evaluated how well the developer console was implemented. The core functionality of the developer console was very easy to use. Signing up followed the natural user sign up flow. The default view showed request rate and it was easy to navigate to account deletion, API key cycling, and the score management page. Since each of the actions a developer can take are relatively simple, and the UI is very clean and intuitive, the developer console provides a very easy way to manage the game account.

4.2.3 Admin Perspective

We want it to be very easy for us and future developers to deploy the API, scale it as needed, and extend its functionality. This section will touch on the ease of deployment (refer to the deployment steps for the deployment instructions in the source code of the project) and the ease of scalability.

The steps for deploying the service was set up by one of us and the other of us followed the steps to deploy the service. The deployment has clear and discrete steps for deployment and, though it was found to be missing some information on the first pass, that was corrected in an update to the deployment steps. The only potential problem arises in the fact that the steps for the Amazon Web Services web console depends on the version of the web console implemented as of December 2015. If a new version of the web site is released and major things are changed, then the instructions may no longer suffice. Going through the entire process, from registering an account on Amazon Web Services to having a production server running serving the code to the website to external users took roughly one hour.

The service scales automatically. This is done by telling Amazon Web Services that after a single server reaches a 70% CPU load, another server should be added behind the load

balancer. This only happens if the server maintains that CPU load for at least two minutes. Once another server is added, all traffic from the load balancer cycles round-robin style between all servers that are currently in service. Once there is very little traffic for another period of time, Amazon Web Services will automatically start terminating server instances that it does not need. The maximum number of servers that will be put behind the load balancer is five.

5 Conclusions & Future Work

This chapter will discuss how this project turned out after completion. It will describe the results of our work from a client perspective, an administrator perspective, then future work that is recommended, with recommendations on how to start going about some of that work.

5.1 Client Conclusions

After finishing this project and conducting a postmortem, we drew some conclusions about how the application works from a client's perspective. Specifically, the only things that a client touches are the developer console, the API documentation (shown here: http://tmwild.com/static/docs/index.html), and the Java client library. The implementation of these three things directly informs a client's experience with using the Rank service for their game.

The developer console allows many functions to the client of the Rank service. It's user interface uses a library that is based on Google's Material Design specification. As such, it has the look and feel of a modern web application. The only functionality that the web application does not offer is the possibility to modify the existing data in the leaderboard. Deleting and viewing the data is available to the client users.

The API documentation (shown here: http://tmwild.com/static/docs/index.html) is built with a new library that was developed by TripIt for a slick and clean API documentation design. While the library doesn't allow the documentation pages to make on-the-fly example calls to the API, it does have other features, such as showing how to make API calls in different languages based on the client libraries that the developers might have created to aid the clients in making calls to the API's endpoints. Since the documentation tool allows for this functionality, it is used as the documentation tool for the Java library as well. While this is sufficient as far as documentation goes, it lacks things like exception catching.

The Java client library is a standalone .jar file (which needs its dependent libraries installed) to help clients integrate their Java projects into Rank's service. It's functions work and return what is expected to be returned. The dependencies that it uses are libraries that are used and developed by large corporations (Google). From a client perspective, the only downside is that all of the existing documentation lives inside of the API documentation itself.

5.2 Administrator Conclusions

Likewise, we found that after conducting the project's postmortem, there were only a few things from an administrator perspective that needed evaluation. Those three things are the administrator console, the deployment documentation (shown here: https://github.com/Rdbaker/Rank/blob/master/deploy/readme.md), and the code repository. These three things will directly influence a Rank service administrator's experience.

The administrator console allows many functions over the clients of the Rank service. The administrator can see how many requests from the expected requests were hit today as well as the number of active clients. The administrator can manage the clients by viewing their data, freezing their API key (rendering their API requests useless), deleting a client's account, or promoting the user to an administrator. Finally the administrator can see the aggregate count over time of the number of requests that has come into the service as well as a link to the Amazon Web Services' Cloudwatch information to see the health of the service's resources.

The deployment documentation (shown here: https://github.com/Rdbaker/Rank/blob/master/deploy/readme.md) for a service administrator can be found in the code repository's "deploy" folder (shown here: https://github.com/Rdbaker/Rank/blob/master/deploy/readme.md). It has a discrete number of detailed steps that the service administrator must follow to set the service up using Amazon Web Services. All steps from creating an account to stopping a running server instance (which halts any cost incurrence for that server instance) are detailed in the documentation. During the project development, one member took on the primary role of setting up the infrastructure. The other project member, towards the end of the project, followed the step-by-step guide to create an account, create the resources, and deploy and run the service. At a leisurely pace, this took a little under one hour (around 55 minutes).

The code repository for the Rank service is available and open sourced on GitHub (https://github.com/Rdbaker/Rank). The same is true for the Rank Java client library (https://github.com/mjperrone/RankJavaClientLibrary). The repositories themselves are made to be readable and encapsulated through each of the subdirectories within the repository. The "skate" folder has the API documentation information. The "migrations" folder has all information on the database migrations that need to be run in order to make sure the models used by the object-relational mapping are up to date. The "rank" folder contains all the information used by the service. The two main folders inside the "rank" folder are "api" and "user". The logic for all of the endpoints of the API and the dashboard are contained in those two folders, respectively. Much of the code in the repository follows PEP8 guidelines (https://www.python.org/dev/peps/pep-0008/). The entire service was scaffolded from the Flask-Cookiecutter project template (https://github.com/sloria/cookiecutter-flask). This allows the code base to be standardized across many different Flask projects and ensures readability.

5.3 The Future

While working on this project we amassed a list of features we wished we could have added, ranging from UI/UX improvements to cron jobs for the purpose of purging the database of unnecessary data The paragraphs that follow discuss things that would be good to add to this projects, as well as an idea on how to implement some of them. Obviously, the details of implementation choice are left out for the sake of brevity, but please contact the authors if you wish to discuss implementation details.

HTTPS: While it was one of our original goals, our service does not currently have HTTPS support. The service we were investigating and invested a large amount of time into did not end up natively supporting our type of infrastructure. We intended to certify our servers from the load balancers, but the service we were investigating requires a single server, and does not support a load balancer. This is something that should be investigated soon to ensure proper encryption and protection during the transportation of client data. Using HTTPS would give the service SSL encryption on traffic to and from the website. This means that our clients wouldn't have to worry about the security concerns of the data that was being sent over the internet traffic.

Client package implementations in other languages: As the number of potential users grows, so does the possibility that the users will be using languages other than Java. While the implementation of a client library will depend on the language itself (and the community behind it), it is a good idea to have a client library because it will wrap the HTTP requests and decouple the HTTP routes from the specific actions that they might represent. This is especially important in cases where there is no specific pattern to the API routes, such as a non-RESTful API (like Rank).

Global users: As the number of clients grows, it will be more likely that different clients will be using different games, but the same user base. For that reason, it may be worthwhile to investigate the possibility of having global users. Currently, Rank only implements users in the form of an integer (representing a user's id). It might be found worthwhile to investigate different use cases that a client might have for persisting a user entity between different games (with different API keys).

Database purging: Every time a user makes a request to any API endpoint of the service, the request that was made is logged to the database. The use case for this is a quick lookup to either see if the API key has surpassed the allotted amount of requests in a day and for usage metrics and reporting of the service. While these requests are logged to the database, they are also logged to the local file system. After one week, there are currently no uses for viewing the requests. For this reason, it would be useful to periodically purge the database of requests that are older than one week.

Log forwarding: Since the API requests are logged locally on each server that receives requests, it would be good to forward the logs to some sort of more permanent storage. There are services and libraries that exist that periodically send local log files to a remote storage, such as AWS' S3.

6 References

Ashkenas, J. (2015). BackboneJS. Retrieved September 12, 2015, from: http://backbonejs.org/

Apiary Inc. (2015). Apiary. Retrieved September 13, 2015, from: https://apiary.io/

Coffeescript (2015). Coffeescript. Retrieved November 21, 2015, from: http://coffeescript.org/

drone.io (2013). Drone. Retrieved September 12, 2015, from: https://drone.io/

Facebook, Inc. (2015a). React. Retrieved September 12, 2015, from: http://facebook.github.io/react/

Facebook, Inc. (2015b). Scores API. Retrieved September 2, 2015, from: https://developers.facebook.com/docs/games/scores

Freeman E., Robson, E., Sierra K., & Bates B. (2004). *Head First Design Patterns*. Sebastopol: O'Reilly Media.

Google, Inc. (2015a). AngularJS. Retrieved September 12, 2015, from: https://angularjs.org/

Google, Inc. (2015b). Material Design. Retrieved November 24, 2015, from: https://www.google.com/design/spec/material-design/introduction.html

Google, Inc. (2015c). Material Design Lite. Retrieved November 21, 2015, from: http://www.getmdl.io/

Hamlett, A. (2015). Pirates Use Flask, The Navy Uses Django. Retrieved September 4, 2015, from: https://wakatime.com/blog/25-pirates-use-flask-the-navy-uses-django

Hansson, D. H. (2015). Ruby On Rails. Retrieved September 4, 2015, from: http://rubyonrails.org/

Internet Engineering Task Force (2012). RFC 6585. Retrieved October 18, 2015, from: https://tools.ietf.org/html/rfc6585

Jones, D. (2012). Rails or Sinatra: The Best of Both Worlds? Retrieved September 5, 2015, from: http://www.sitepoint.com/rails-or-sinatra-the-best-of-both-worlds/

JSON (2006). Introducing JSON. Retrieved November 24, 2015, from: http://www.json.org/

Knupp, J. (2014). What Is A Web Framework? Retrieved September 6, 2015, from: https://www.jeffknupp.com/blog/2014/03/03/what-is-a-web-framework/

Lindeijer, Thorbjørn (2015). Tiled Map Editor. Retrieved Septermber 22, 2015, from: http://www.mapeditor.org/

Lord, R. (2013). Slate. Retrieved September 13, 2015, from: https://github.com/tripit/slate

Lowry, B. (2013). Playtomic. Retrieved September 4, 2015, from: http://playtomic.org/

Makai, M. (2015). Full Stack Python. Retrieved September 4, 2015, from: http://www.fullstackpython.com/web-frameworks.html

Marionette (2015). Marionette.js. Retrieved September 12, 2015, from: http://marionettejs.com/

Nygard, M. T. (2007). *Release It!: Design and Deploy Production-Ready Software*. Raleigh: Pragmatic Bookshelf.

Pires, B (2014). Rails Vs. Django: An In-Depth Technical Comparison. Retrieved September 4, 2015, from: https://bernardopires.com/2014/03/rails-vs-django-an-in-depth-technical-comparison/

Rydzewski, B. (2014). What are the core differences between Travis CI and drone.io? Retrieved September 12, 2015 from: http://qr.ae/RHjLMU

Searchsoa (2015). REST (representational state transfer) definition. Retrieved November 22, 2015, from: http://searchsoa.techtarget.com/definition/REST

Solano Labs, Inc. (2015). Solano. Retrieved September 12, 2015, from: https://www.solanolabs.com/

Spry Media Ltd. (2015). DataTables. Retrieved November 22, 2015, from: https://www.datatables.net/

Swagger (2015). Swagger. Retrieved September 13, 2015, from: http://swagger.io/

The jQuery Foundation (2015). jQuery. Retrieved November 22, 2015, from: https://jquery.com/

The Python Software Foundation (2015). Logging facility for Python. Retrieved November 24, 2015, from: https://docs.python.org/2/library/logging.html

ThoughtWorks, Inc. (2015a). Continuous Integration. Retrieved September 12, 2015, from: https://www.thoughtworks.com/continuous-integration

ThoughtWorks, Inc. (2015b). Snap. Retrieved September 12, 2015, from: https://snap-ci.com/

Tilde, Inc. (2015). Ember.js. Retrieved September 12, 2015, from: http://emberjs.com/

Travis CI, Gmbh (2015a). Customizing The Build. Retrieved September 12, 2015, from: http://docs.travis-ci.com/user/customizing-the-build/

Travis CI, Gmbh (2015b). Travis CI. Retrieved September 12, 2015, from: https://travis-ci.org/

Twitter, Inc. (2015). Bootstrap. Retrieved November 24, 2015, from: http://getbootstrap.com/

Twitter, Inc. (2011). The Engineering Behind Twitter's New Search Experience. Retrieved September 5, 2015, from: https://blog.twitter.com/2011/engineering-behind-twitter%E2%80%99s-new-search-experience

Unity Technologies (2015). Unity. Retrieved November 12, 2015, from: http://unity3d.com/unity

7 Appendix

A - Why we chose Leaderboards

We want to build one of the options listed in the above section, so we considered each of them.

Game engines are extremely complicated systems that require both low level hardware knowledge and intricate designs for building a sensible interface. This is far out of the scope of what we can accomplish during this project.

A map editor is possibly contained enough to have enough time to build it, but it would not raise any concerns about scalable design, which is a research goal of this project, so we chose not to pursue this option.

A friend or clan network is a simple concept at first glance, but it becomes a bit more complicated when you add privacy and abuse concerns. Data about who is friends with whom can be considered personal information. Drafting terms of service for how we store, track, and possibly sell data is out of the scope of this project. Direct user to user interaction like with a friend network opens up a lot of potential abuse vectors like harassment, stalking, and friend request spam. We don't want to have to deal with those problems, since they are of a more social science nature and less of a technical nature.

Achievements

A leaderboard service is much simpler than the other options, and since it needs to be centralized (instead of existing just on the game client), it is subject to scaling concerns. This is well contained, and can be generalized to be useful for many different kind of games. For these reasons, it fits the project objective (learn about scalable web systems) reasonably well.

B - Facebook's Game Development API

Facebook offers many services to game developers to help them with features they may want for their game (Facebook Inc., 2015b). Their services for games range from push notifications to monetization via a subscription service. This section will focus on their Scores API, game requests, and achievements.

The Scores API for Facebook associates a user and an app with one another via a "scores" , much like we intend to do with a leaderboard. Facebook matches users to game scores on the granularity of one user score per game. This can be created, deleted, and updated. Another interesting point to note about Facebook's Scores API is that it is from the perspective of a user, and not from a game. All routes are RESTfully accessed from the USER_ID entity, and all requests are sent with a developer token. This makes sense for Facebook's use case, because Facebook is user centered, whereas our service is game centered.

The Scores API allows read access of a user's scores with a GET request to /USER_ID/scores. This returns an array of objects with information about the user, the app, and the score. The API allows update/create access of a user's scores with a POST request to /USER_ID/scores. It requires a payload of { "score" : <integer> } and returns with a boolean indicating whether or not the create/update succeeded. Lastly, on the granularity of a user, the API allows deleting the score for a play for the app by sending a DELETE request to /USER_ID/scores. It returns a boolean indicating whether or not the operation was successful. Finally,the API allows deleting all scores for a game by sending a DELETE request to /APP_ID/scores. It returns a boolean indicating whether or not the operation was successful.

Facebook's game requests are categorized into three different scenarios:

1. Sending to a friend who has not played before (i.e. inviting a friend to join a game)
2. Sending to a friend who has played before (i.e. turn-based notifications, gifting and asking for help)
3. Sending to a non-friend who has played before (i.e. match-making)

The game request works by asking the USER ID (or IDs) to send the request to, the type of request it is, and a message for the recipient to see. Facebook offers many client libraries and SDKs to aid in development. Instead of exposing the url, they ask the developers to use the functions in the SDK.

Lastly, Facebook has an Achievements API. Facebook enforces a total 1000 point limit between the sum of all achievements, distributable across no more than 1000 achievements. They have a recommended scoring policy for the achievements (50 - hard; 25 - medium; 10 - easy), and they require that an achievement is never deleted, save for testing purposes. When developers create a new achievement, Facebook requires certain specified properties: the title, the

description (which should indicate how the player earns the achievement), the url, the image url, the point value, and the app id. They tell developers to send that information to the /APP_ID/achievements url. There is no specification on how a player earns that achievement, so I imagine that it is built into the rules of the game.

C - Playtomic's API

Playtomic is an open source, hosted solution to a few problems game developers may have (Lowry, 2013). It provides APIs for leaderboards, achievements, user content, dynamic content, and newsletters.

The leaderboard API allows for many different filters to create custom leaderboards. It allows filtering by a user's friends so that one can see how well she is doing compared to her friends. It allows filtering just by one player id, so one can see how well he or a friend has progressed over time. Playtomic also allows to filter based on recency; it can show the top scores for today, the last week, the last month, and all time. You can also add custom fields to every score, and then filter on them later. This would allow for many independent leaderboards, perhaps one per level of the game, or one per class of player. Finally, it allows a configuration option to indicate whether a lower score is better (like golf) or a higher score is better (like baseball).

Playtomic's achievements API provides a way to award and search for achievements that players can unlock by performing specific in-game actions. The user content API provides an easy way to allow users to produce and share in game content such as custom maps. The dynamic content API allows for quick modifications to your game post distribution via config variables that live on the server. The newsletters API provides an easy way to subscribe users to a mailing list for updates.

Playtomic does not host these APIs for you. Instead, it is preconfigured for heroku to deploy these servers with one step. In order to get the server working, one needs to just have a heroku account and point heroku at the playtomic git repository. Scaling up is simple, just select heroku dynamos with more power. Scaling out is also simple, just add more dynamos. The only danger to this is that the data is only eventually consistent; when there is a write to one dynamo, it does not update other dynamos immediately.

D - Front End Tooling Decision

Front end web applications have been on a trend to becoming more mature and powerful, with a wise community behind them. There are multiple different tools we may consider using in building this service. There are frameworks for building front-end heavy web applications, such as AngularJS and Ember.js (Google, 2015a; Tilde Inc., 2015). Frameworks typically enforce a paradigm and a structure to use the functionality the framework provides. There are also libraries to aid in building user experiences and interfacing with a web API, such as React, Backbone.js, and Marionette.js (Ashkenas, J., 2015; Facebook Inc., 2015a; Marionette, 2015). A library will provide utility functions without necessarily enforcing a paradigm or structure on the code using the library's functionality.

Our use case lends itself better to taking advantage of smaller, more composable libraries, because the service is not focused very heavily on user interactions. For this reason, it is likely that some combination of the aforementioned libraries (Backbone.js, React, and Marionette.js) will be used, if at all.

React is an open source project created and maintained by Facebook (Facebook Inc., 2015a). The tagline for the library is: "A JavaScript library for building user interfaces." Most of React's functionality comes from allowing the developer to define and reuse web components that are made for different situations. React allows the developer to create these components then act based on the user interactions on these web components.

Backbone.js is a library that provides models, collections, and views for the developer to use (Ashkenas, J., 2015). It was originally intended to model interactions on a RESTful API, to create user interactions that made sense for the uses of the data. It follows a paradigm of MVVM (model-view-viewmodel). The view handles the user interactions and rendering (or re-rendering) the model, while the model handles the business logic of the application. When needed the model can "sync" or "fetch". These functions send information to the API to either get the model's data or update the model's data via HTTP requests.

Marionette.js is a library that is built on top of Backbone.js (Marionette, 2015). The purpose of Marionette.js is to provide useful pieces of functionality to the views of a Backbone.js application. Backbone.js' strong features come primarily from the model interactions with the API, whereas Marionette.js' strong features come from providing Backbone.js views that already have many helper functions in them.

Despite our research into this area, when it came time to build the front end of the web application, we did not feel it was large enough to merit using any type of front end application framework or library.

E - API Documentation Tool

API documentation is important primarily to keep the service users up to date on what services are available and how to use them. In our research, we have identified multiple different tools that help to document APIs. The tools we will consider include Swagger, Apiary, and Slate (Swagger, 2015; Apiary Inc. 2015; Lord, 2013).

Swagger markets itself as a powerful yet simple tool to use for describing APIs (Swagger, 2015). Swagger has a tool called Swagger UI, which is the collection of HTML, CSS, and JavaScriptcode that take a Swagger compliant JSON document and renders the actions made available by the API. A sample of how Swagger UI works and looks can be found at Swagger's example: the Swagger Pet Store (http://petstore.swagger.io/). The API developers can either write the JSON document that describes the API actions themselves or they can find a tool that will automatically generate the JSON document by reading their files. Multiple tools of this nature exist for different languages and web frameworks. Swagger UI allows the user to send a request straight to the API by interacting with the web page.

Apiary aims at an enterprise level target market by claiming to offer a solution to inconsistent API design across a company (Apiary Inc., 2015). If everybody has the same API specs written on Apiary, then there will be one central place to know what actions are permitted on a given service. Apiary works by writing the API description in the online editor and saving it online. It can hook into a GitHub repository to publish the API document to the repository. To view the resulting markdown you must have a link to the Apiary API document. Self-hosted solutions are also available for enterprise level customers.

Slate offers a way to write a markdown description of the API (Lord, 2013). Unlike Apiary and Swagger, Slate is not interactive, meaning that the user cannot send a request to the API from the documentation. Slate does, however, support multiple different ways of displaying how to send a request to the service for a specific endpoint. This feature is useful if the API developer would like to describe how to use client libraries in different languages, as well as how to send a raw HTTP request from the bash shell, for example. Slate seems to offer the least overhead in setting it up to work inside a project, as it is just a static file served from the project server; but it also seems to require the most work, as it seems to require substantially more writing than either Apiary and Swagger.

F - Elastic Beanstalk

Elastic Beanstalk is Amazon Web Services' platform as a service which allows you to point it to a python app, and it deploys it to servers with a load balancer, configures the firewall settings, and sets up other services like that. We initially expected that this would cut down on the time it took us to deploy the app, but it ended up increasing the effort. Since Elastic Beanstalk does so much for you, it is much harder to tweak it. This made it hard to add HTTPS, and hard to control the autoscaling properties. After we hit that barrier, we backtracked and just directly managed the AWS resources ourselves.