

# A Case Study on Spotify

I have always been interested in delivering music to people, and have always admired Spotify for sharing that passion, and in particular:

- How they are able to cater to 320 million monthly active users.
- How they are able to tailor the experience to each and every user.
- How they can provide such a large and dynamic library of over 60 million tracks to its user base.

# R1: Research and describe the technologies used to deliver services or products of the chosen organization.

## Presentation Layer

- **JavaScript:** The primary Spotify desktop web player interface for desktop is comprised of various proprietary elements that combine to form the full interface. Each of these elements is authored by a dedicated team and written in vanilla JavaScript.
- **React:** In order to reconcile the various JavaScript elements forming the application, Spotify adopted the React framework in 2016. This came with a vast performance improvement as well as enabled interoperability between JavaScript elements.
- **Swift:** All Spotify applications on the iOS platform are written natively in Swift.

## Application Layer

- **Apollo:** Spotify has written its own application for handling web requests called Apollo. It is designed to handle routing and the standard HTTP methods. All server-side code, whether it be a web server or a back end microservice, make use of this technology.
- **The Spotify API:** The back end of Spotify's architecture is comprised of a multitude of microservices, which now are increasingly being accessed through a publicly available API. This is enabling the core functionality of Spotify to be accessed through a centralized point, as well as encourage open-source development and extensibility. According to an inspection of the Web Player's HTML, the official API is used amongst a few other endpoints, such as the web player gateway and custom playlist generation endpoints.
- **Google Compute Engines:** Spotify makes extensive use of GCE virtual machines to host web servers, as run the various microservices on the back end of the infrastructure. These machines are managed via a Regional Instance Group, which runs multiple machines as a single logical entity and dynamically scale based on load.
- **Docker:** Docker containers provide a lightweight means to generate and deploy environments on virtual machines. According to the article cited above, docker

containers set up to run Apollo instances are what is deployed on Google Compute Engines.

- **Java/Scala:** The predominant coding language at Spotify is Scala, which is both a superset of Java and a build tool. Apollo is written in Scala, as is Scio ([see below](#)).
- **Python:** Spotify exposes a framework called [Klio](#), which enables developers who are developing machine learning pipelines and audio processing algorithms at Spotify to perform batch operations on the entire music library. This plays a major role in providing streaming quality options and music preferences to the end-user.

## Data Access Layer

- **Google Bigtable / BigQuery:** In 2015, Spotify made the decision to move from using in-house architecture to Google's Cloud Platform. Bigtable is a NoSQL database that is designed to handle large amounts of data as well as dynamically scale in terms of storage space. Paired with this is BigQuery, which is a propriety technology which is designed to run SQL style queries against petabytes of data. [Source - Jon Hearsch](#)
  - **Google Pub / Sub:** Every time a user performs an action on Spotify, such as logging in or playing a song, an '**event**' is triggered which creates a cascade of other operations through the Spotify architecture. Pub / Sub allows for the decoupling between the original events and the events that should take place as a consequence.
  - **Scio:** The processing of events involves the transformation of data as it passes through different points of the architecture. For interoperability with the above technologies, Google's Dataflow platform is used. However, in order to integrate further with other technologies such as Hadoop, Spotify has developed [Scio](#), an API written on top of Dataflow.
  - **S3 / Google Cloud Storage:** For storing media files, Spotify uses a mix of AWS and Google storage platforms.
  - **Fastly:** In order to improve performance, Spotify uses Fastly, a company that acts as an intermediary between the application layer and the media storage units. It also caches media, so that more in demand files are duplicated in memory so that users can be adequately served.
-

## R2: Identify one application stack utilized by an organization to deliver a product or service. Describe the role of each technology used with an application stack and how it relates to other applications/technologies in a stack.

At its essence, Spotify is acting as a broker between the listener (customer) and the artist (vendor). Spotify grants its subscribers unlimited access to all songs in its catalog for an ongoing fee and passes on an atomic payment to the vendor, based on the popularity of the track. It is this transaction that occurs between the customer and the artist that will form the basis of the application stack for this discussion.

- **Streaming Process:** The streaming of music to the end-user is largely a front-end mechanism, where an embedded JavaScript player is initialized with an OAuth token. The player object exposes listeners that monitor the player's state (paused/playing, track location, etc). The listeners are consumed by the graphical Web Player widget. The player object can be manipulated with direct function calls or accessed indirectly through the Spotify API.

The player delegates to Fastly, a third-party service to interact with Spotify's S3 and GCS storage units to provide streaming optimization and availability via caching. The streaming process also uses Encrypted Media Extensions (EMEs) to implement digital rights management.

The embedded player will also publish events (such as when a song ends) to Google Pub / Sub, presumably via a separate microservice.

Source: [Jose M. Perez](#)

- **Google Pub / Sub:** In order for Spotify to pay its artists, it needs to be able to track what the end-users are listening to in order to calculate the appropriate remuneration. When it comes to a specific track, Spotify needs to know:
  - When a song was selected to be played.
  - How long the song was listened to.
  - Whether the end-user finished listening to the complete song.

Each of the above points is considered to be an event in Spotify, which need to be recorded in persistent storage for record-keeping. According to its engineering blogs, there are over three hundred unique events that can be created, and 100 billion individual events are triggered daily. Google's Pub / Sub

service provides a mechanism for the separation of concerns from the origin of the event, and the data streaming process that occurs when the event is published.

Some events are more common than others, therefore each event type is consumed by an auto-scaled cluster of workers running an Apollo service. The job of the worker is to write the events into files on an intermediate storage unit known as a bucket. Each bucket will contain an hour's worth of a single event type.

A completionist service will monitor the state of the bucket and send an acknowledgment back to the Pub / Sub service once all of the hourly events have been aggregated. To ensure data integrity, a deduplication service will batch process the bucket upon completion to remove any duplicate events that may have been generated.

- **Scio / Google Dataflow:** The data that is collected in these buckets will then need to be batch processed. There are multiple implementations of data processing and storage at Spotify, principally Hadoop / HDFS and Dataflow / GCS, and events pertaining to artist royalties could be either or a mix of technologies.

In order to streamline the coding environment amongst all of the backend coding teams, Spotify authored Scio, an SDK written in Scala which provides a polymorphic interface for data processing. As well as data processing, Scio also incorporates endpoints for BigQuery and Bigtable, Pub / Sub, and other parts of the Spotify ecosystem.

Finally, when an automatic payment to an artist is due to be made, the microservice that is triggered which queries aggregated data collected from event-driven data processing will also have been written using the Scio SDK.

**Source:** [Igor Maravic](#)

---

### **R3: Research the hardware/cloud platform(s) that are utilized by the organization and describe what hardware is required.**

The prime focus of the shift towards the Google Cloud platform from traditional infrastructure was triggered by system outages as a result of the volume of Spotify's event-driven data processes. To address this bottleneck, Google's Pub / Sub was implemented due to:

- Automatic scaling, Pub / Sub is effectively a server-less technology.
- Extensibility, Pub / Sub relies on HTTP requests for event delivery meaning that Spotify could write its own wrapper implementation (what would eventually be Scio) tying in with its mantra of writing custom unified interfaces.

As a flow-on effect of adopting Pub / Sub, Spotify then opted to adopt the full range of Google's Cloud Platform services. With that and the more recent adoption of Fastly for content delivery, Spotify retired all of its own physical hardware in 2018. This has now enabled Spotify to serve its 60 million tracks to its 320 monthly active users.

Spotify has an R&D team of approximately 2000 members, who are running multitudes of projects from data analysis to machine learning audio processing algorithms. Additionally, its event-driven data processes are constantly resizing its resources based on usage. Therefore, it is impractical (and perhaps unhelpful) to try and analyze Spotify's Cloud usage based on individual hardware components.

It is clear that Spotify makes every effort to dynamically size its requirements in order to optimize cost vs performance. It is perhaps epitomized by its authoring of its own Bigtable "auto scaler" which reflectively analyzes the load on a Bigtable cluster and adjusts the number of nodes as necessary.

However, what emerges from the advent of operational code becoming decoupled from the hardware itself is a reduced ability to monitor usage and therefore the overall cost of cloud resources. To combat this, Spotify released a publicly available cost insight tool, which allows its software engineers to fine-tune the hardware requirements for their respective projects therefore optimizing operation cost.

When making a final determination on Spotify's cloud resource utilization, the monetary cost is possibly the most useful benchmark. According to this article from 2018, Spotify spends approximately 150 million USD on Google Cloud Platform services per year. Whilst a large figure in absolute terms, when cross-referenced

against Spotify's operational costs in the most recent quarter in 2020, a rough calculation suggests that GCP's costs account for about 2% of operational expenses.

SUMMARY USER AND FINANCIAL METRICS					
				% Change	
USERS (M)	Q3 2019	Q2 2020	Q3 2020	Y/Y	Q/Q
Total Monthly Active Users ("MAUs")	248	299	320	29%	7%
Premium Subscribers	113	138	144	27%	5%
Ad-Supported MAUs	141	170	185	31%	9%
FINANCIALS (€M)					
Premium	1,561	1,758	1,790	15%	2%
Ad-Supported	170	131	185	9%	41%
Total Revenue	1,731	1,889	1,975	14%	5%
Gross Profit	441	479	489	11%	2%
Gross Margin	25.5%	25.4%	24.8%	--	--
Operating Income/(Loss)	54	(167)	(40)	--	--
Operating Margin	3.1%	(8.8%)	(2.0%)	--	--
Net cash flows from operating activities	71	39	122	72%	--
Free Cash Flow <sup>1</sup>	48	27	103	115%	--

// Calculation logic, figures above in millions of Euro  
 150 Mil USD per year @ 0.82 EUR = ~31M Euros per quarter  
 Operational Costs for Q3 2020: 1975M - 489M = 1486M Euros  
 Therefore GCP costs as percentage = 31 / 1486 = ~ 2%

## R4: Research the data model of an organization by looking at their API documentation and any other sources of publicly available information and describe the organizational functions that are possible based on the API.

Spotify has a [publicly available API](#) that provides most of the core functionality of the application from the listener or consumer's perspective. Most of the API's functionality requires an OAuth token, and there are seventeen scopes available for authorization.

### Playback

```
ROUTE(S):  
- /player/*
```

Spotify exposes a Player object as a model for controlling basic player functions (such as play/pause, seek, next/previous) for existing players. The queue is a list of tracks for the player to automatically play. The API allows for tracks to be added to the queue, but for no other manipulation. Any track, album, or playlist can be submitted to a Player for immediate playback. A user can have multiple instances of Players running simultaneously, but only one instance can be **'active'**, meaning that it can produce playback.

In order for a player to be created the user needs to initialize the Spotify application. Alternatively, Spotify provides an SDK which provides a headless player client for 3rd party developers. A sample project is provided on their [developer website](#).

### Object Retrieval

```
ROUTE(S):  
- /audio-features/*  
- /tracks/*  
- /playlists/*  
- /episodes/*  
- /shows/*  
- /albums/*  
- /artists/*
```



The basic objects that exist in the Spotify framework are tracks, albums, playlists, artists, shows and episodes. All of these objects can be retrieved by their id attribute (which is presumably their primary key) so that their attributes can be viewed. It is not possible to create or delete content on the Spotify API. Rather, this is done via [Spotify for Artists](#), which does not have a public API.

The basic objects are:

- **Tracks:** A single song or audio file that can be played.
- **Artist:** The creator of the track.
- **Album:** A conceptual grouping of tracks by the artist, which are released simultaneously. This preserves the physical media paradigm of the music industry.
- **Shows:** A non-musical visual production (such as a podcast), a new entity now being offered by Spotify.
- **Episode:** A single, serialized unit of a show.
- **Playlist:** A custom, ordered array of tracks that can span any combination of albums or artists.

Playlists are the exception for object manipulation in that they are a truly RESTful resource. Playlists are generated by both end-users and automatically by Spotify's recommendation algorithms.

## Exploration

```
ROUTES(S):  
- /search/*  
- /browse/*  
- /recommendations/
```

Given that Spotify has a library of over 60 million tracks, it makes use of functions that can recommend and filter content on a user-to-user basis. These mechanisms are available in two broad categories; searching and browsing.

Searching is for when the user has a firm idea of the content they wish to view, so provide keywords to cross-reference against the object types listed above. Additionally, the desired object type of the return value can also be specified.

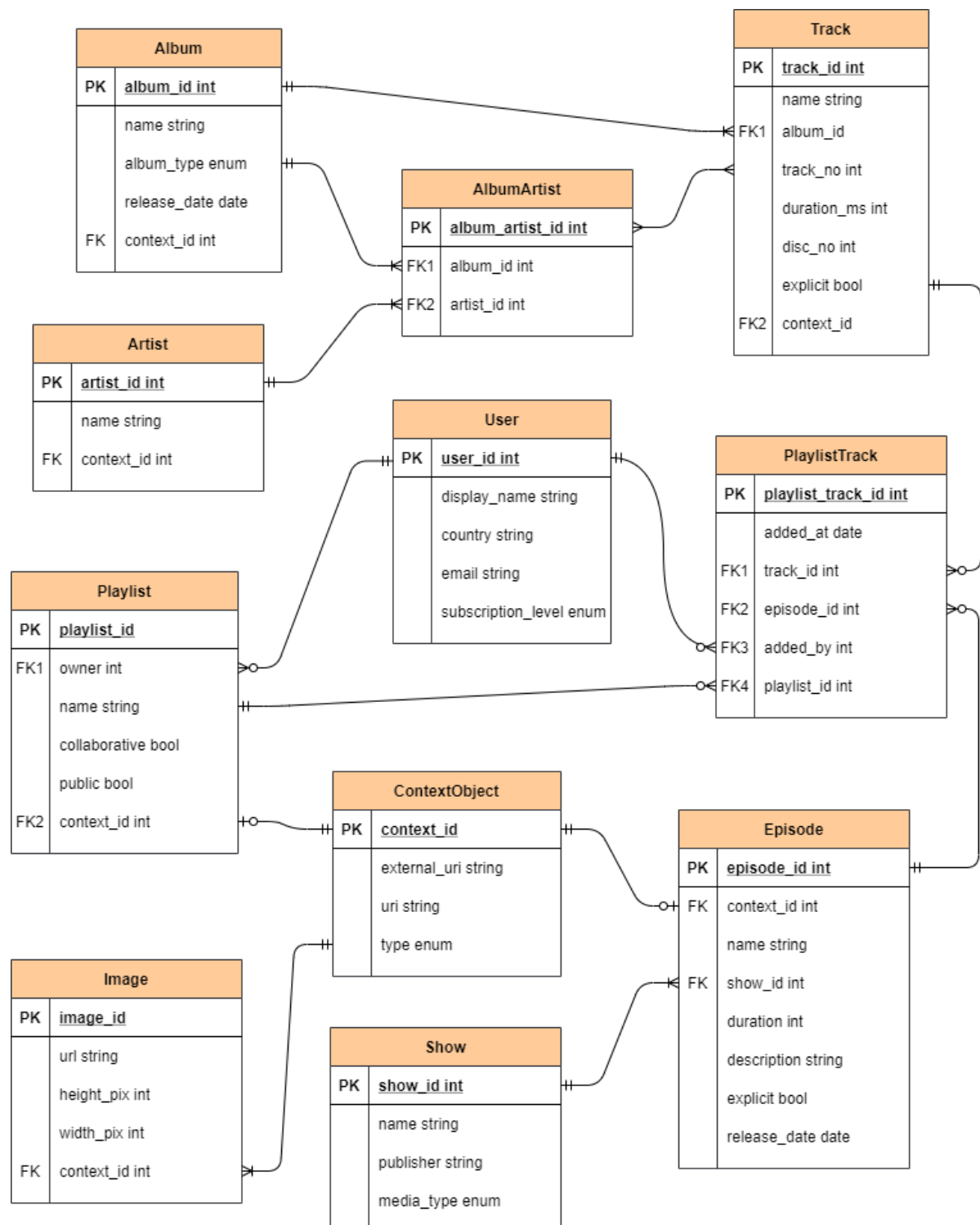
Browsing is where the user provides a more generic parameter (such as a genre) and Spotify algorithmically returns a list of playback items based on the profile of the

user as well as popular trends. The recommendations function is effectively a hybrid between browsing and searching. Where keywords can be provided that act as a 'seed' for the algorithm rather than a conventional search.


---

**R5: Create an entity-relationship diagram that represents entities used in a product or service (or part of) and the relationships between them.**

API Reference



## Enum Values

<u>Aa</u> Name	 Possible Values
<u>album_type</u>	album, compilation, single
<u>type</u>	playlist, track, album, artist, episode, show (possibly others)
<u>media_type</u>	audio, video
<u>Untitled</u>	

## The Context Object

The context object is the supertype of multiple entities in the Spotify architecture. Any entity which extends from Context Object can be passed along with a device ID to a Player endpoint for immediate playback. It also groups attributes (notably URLs) to keep entities DRY.

In order to increase readability, not all context object relations with their subtypes have been indicated, however, they all conform to an optional one-to-one relationship.

## Schemas VS Entities

Many objects that are returned by the API contain both simple and complete versions. Often this is to reduce verbosity when returning nested entities. I'm assuming that these entities are kept whole in persistence and making use of schema dumping when returning JSON data to the user. The same logic has been applied to the Private and Public User objects.

---

## R6: Describe two processes for the input and output of data based on the company's API and how they achieve organizational objectives.

### A) Search on the Web Player ([Link](#))

```
GET <https://api.spotify.com/v1/search>
```

#### Process

1. As the user types a query in the search box, the web application is making calls to the API. The user text will be submitted as a query parameter 'q' and converted to be HTML compliant. Spotify will request that every type of media be returned, but set a rather small limit of results. If the user clicks on a 'see more {object}' button, the limit will be raised, the type will be specified, and offset parameters will also be included to implement pagination.
2. It is assumed the API then calls a microservice to populate the results, using Google BigQuery. Whilst a proprietary algorithm, it would be matching for exactness as well as popularity of the results. This service would be run for each context object type specified.
3. The return search results are returned as JSON, as well as published on Google Pub / Sub. The event will be asynchronously consumed for permanent storage, caching, and recalculating the popularity attribute of the entity.
4. The React web application manipulates the DOM of the web page, grouping the search results by `context type`. It will pull in the images via the `image_url` attribute of the objects.

#### Objectives

- The user is more effectively able to negotiate the large library of items to select from, serving them with the content that they wish to consume.
- Spotify is able to gain an insight into popular trends such that they can feature specific content on the home page and well as implementing User Personalization.

## B) 3rd Party Playlist Provider

```
GET <https://api.spotify.com/v1/playlists/{playlist_id}/followers/contains>  
PUT <https://api.spotify.com/v1/playlists/{playlist_id}/followers>
```

### Scenario

A self-help guru as part of their teachings wants to offer their followers a service on their website where they will be automatically subscribed to different Spotify playlists based on mood ( Study, Meditative, Energised, etc).

### Process

Note that similar processes to the first scenario take place here too, but have been omitted for brevity.

1. The user clicks a link on the 3rd party website to subscribe.
2. The 3rd party app uses an SDK (such as Spotipy), or manually orchestrates an authorization flow on their back end server to generate an OAuth token. Both the "playlist-modify-public" and "playlist-modify-private" scopes will be requested.
3. Upon authorization, the app then runs an AWS Lambda function from an HTTP endpoint, which retrieves a list of playlist ids from an Amazon DynamoDB instance.
4. It then uses a combination of both endpoints above to ensure that the user is subscribed to the self-help guru's playlists. This time, the playlist id is passed as a path rather than a query parameter. The token will be passed on every occasion as it is required by the request.
5. The token is then stored in the DynamoDB instance, for future use when playlists are added to the service.
6. When the user next logs in and views their library, their playlists will be updated.

### Objectives

- Spotify actively promotes development through third parties through their development platform, and this scenario ties in with philosophy.
  - Spotify is able to potentially attract new customers via a third party.
-

## R7: Develop an extension or modification of the existing data model to improve an organizational function. Explain how these changes will lead to an improvement.

### The Premise



***Revise the Tracks, Artists, and Albums data structure such that it would incorporate classical music in a way that more accurately reflects the genre.***

### How Classical Music is Different

- There are many types of different types of artists, such as performers, conductors, and composers. This is very different from the modern context where creation is normally attributed to a single entity or person (the artist).
- Classical music has a multitude of sub-genres, resulting in the length of an individual piece ranging from a couple of minutes to several hours. The consequence of this is that many pieces will span multiple tracks, and there may be multiple works on multiple works on a single album.
- Works will have multiple performances across multiple albums but share identical information.
- Works have distinct sub-sections that are consistent with the work. For example, *symphonies* contain *movements*, and *operas* contain *acts*, which in turn may contain *numbers*. Again this information will consistent between performances.

### A Proposed Model

- Introduce a **Work** model, which would contain information such as the **composer**, **composition\_date**, **sub\_section\_type** (movements/acts), **work\_type** (symphony/opera etc) and so on.
- Introduce the **SubSection** model, which represents the individual units of a **Work**. It would contain attributes like **name**, and the **ordinal\_number** in its parent. This is very similar to the relationship that a **Track** has with its **Album**.
- Allow for **SubSections** to be infinitely nested.

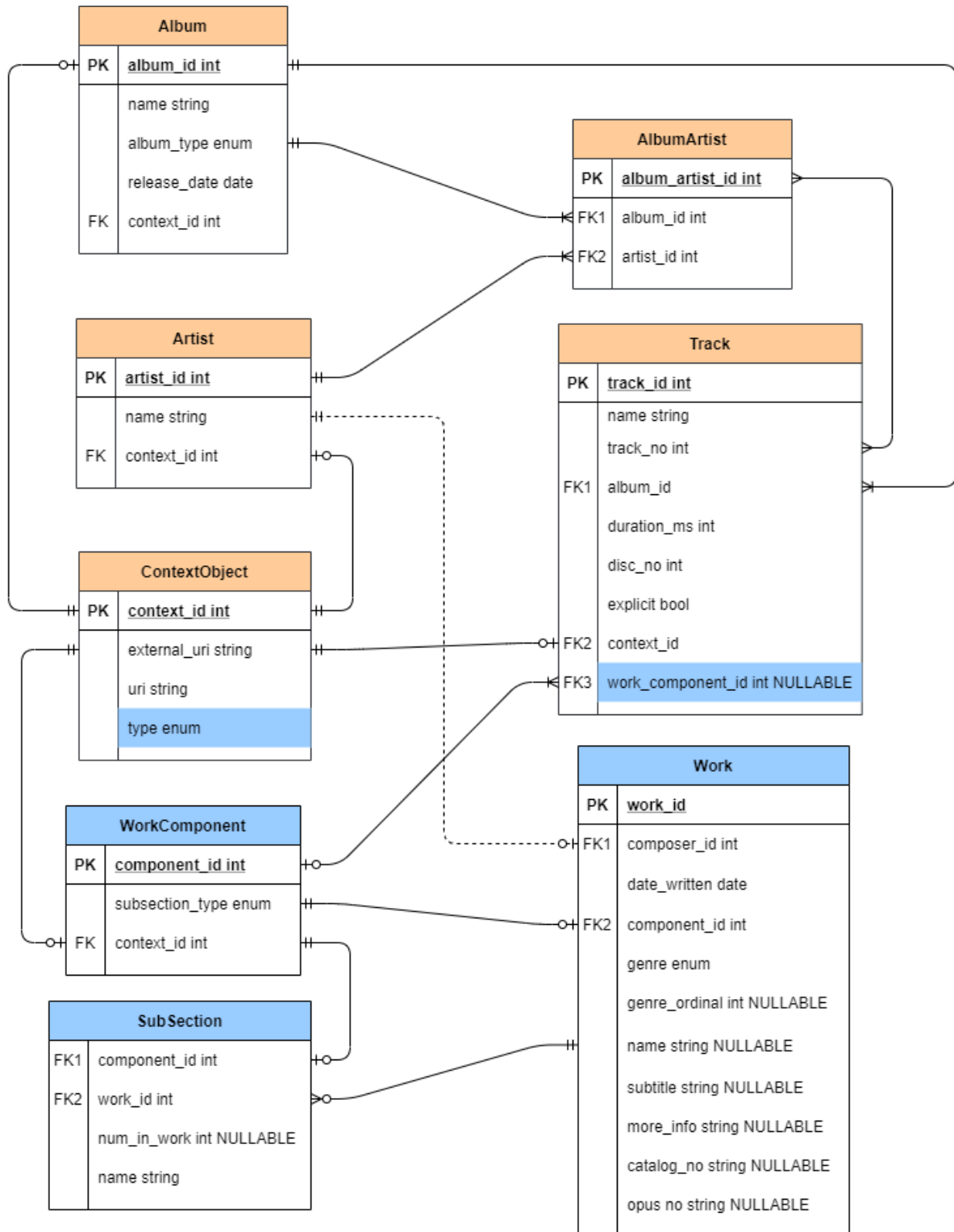
- Wrap Work and SubSection objects under a single entity supertype called WorkComponent . A WorkComponent will be a subtype of ContextObject .
- Allow tracks to contain a reference to a WorkComponent .
- Create a new API endpoint called /works/ which will allow for GET requests similar to other ContextObject types.
- Add an attribute to the Artist model ( artist\_type ) that contains values such as "composer" or "conductor".
- When using the /search/ and /browse/ endpoints, users will be able to declare work , composer , performer

## The Benefits


- **UI Improvement:** Information about the currently playing track could convey the information clearer to the user rather than jamming multiple fields on the track and artist fields.
  - **Search:** Users can search by work (which is the most common approach by consumers of classical music) rather than by album which is hit and miss. Search by the artist sub-types will also be possible.
  - **Context:** As a work entity is shared across tracks, more context about the work, such as trivia and historical information can be provided to the user, enriching the user experience.
  - **Analysis:** Insight into popularity by classical work will now be possible, information like the most popular performance of a particular work, and recommendations for other works will now be possible to obtain
-



**R8: Create an entity-relationship diagram(s) that describe the entities and relationships you propose to extend or modify the system.**



## Enum Values

<u>Aa</u> Name	 Possible Values
<u>type</u>	composer, conductor, soloist, ensemble (in addition to the others)
<u>genre</u>	symphony, concerto, suite, opera, tone_poem etc
<u>subsection_type</u>	movement, act, number etc

## Notes

- Modified attributes and added entities are indicated by blue.
- The `subtitle` variable is the means by which you can specify things like instrumentation or other miscellaneous information. For example, "for Piano" might be used when specifying Beethoven's Piano Concerto #5.
- The `catalog_no` and `opus_no` fields must be strings as they take forms like 'Opus 3a' and 'w.o.o. 13/2'.
- Sometimes a section of a work is split into multiple tracks, hence the many-to-many relationship. This recording is one such example.
- The `num_in_work` field indicates the position within the larger framework of the piece. For example, if this represented the second movement in a symphony, its value would be 2.
- The `ordinal_num` refers to the ordinal number of the work relative to how many pieces the composer has written of the same type. For example, Beethoven's fifth symphony would contain the value 5.

## R9: Justify the technical and operational feasibility of your improvement or extension to an organizational function.

### Rollout

Given the size and complexity of the Spotify organization, the process of adopting the proposal should be as incremental as possible, whilst also keeping the destructive operations on the underlying data structures to a minimum. Fortunately, there should be relatively few migration issues as the only modifications to the data involve adding a nullable column and widening the scope of a constraint.

The next task would be creating the various work data structures so that the tracks can provide a reference to them. Spotify has spent a great deal of time sanitizing the track name information for classical music. This is evident by the fact that I was able a couple of months ago to display track information in a more classical-friendly way in [this project](#). The algorithm for converting track information is [documented here](#). It would be realistic to think that with the vast amount of resources Spotify has that it could algorithmically generate `Work` entities from the track information and reference them correctly. It would then only be a matter of performing some fine-tuning manually after the fact.

From this point, new events could be designed and implemented to enhance the user personalization of the application. The API would then be able to be extended to include the modified `/search` and brand new `/work` endpoints.

Once the API is extended, the front end would be able to take advantage of the updated API, where it could create extra lines in the currently playing box, as well as generate extra rows in the search results page dynamically for the composer, work, conductor, etc.

### Benefit to Stakeholders

The idea to make a streaming service more tailored to classical music is not new, and there are a few players on the market that are [already capitalizing on this idea](#). By implementing this proposal Spotify could potentially win back the market share that it is losing to these types of competitors.

There will also be the potential for collaboration with performing arts companies. The information that could be extracted from Spotify's data analytics would be of high value to any classical music institution. This has increased this year with many

companies turning to streaming platforms due to the coronavirus pandemic, with user personalization becoming highly desirable as a result. Additionally, since many arts companies are already utilizing a platform such as Tessitura, there would be many opportunities for an exchange of data.

## **Cost**

The largest cost associated with this exercise would be the time required in development to execute it. Fortunately, the initial phases and user interface tweaks would take relatively much less time than honing the data analytics component. Ultimately, improving the user experience is where most of the benefit would be in terms of an increased user base.

There would also likely be an increased cost in the form of cloud resources. However, because only text-based data is being added, and classical music forms the minority of content on the platform, it's hard to imagine much of an increase of cost incurred from this.

## R10: Select two technologies in the organization's stack and provide an alternative solution/technology and provide a comparison.

### Streaming Optimization

Currently, Spotify primarily uses Fastly as part of its Content Delivery Network (CDN). However, it is curious that the company opted to go down this path, given its primary cloud service provider Google provides Google CDN as an ostensible alternative. Furthermore, as well as a more unified ecosystem, there appear to be quite favourable benchmarks for Google's CDN according to a few benchmarks. Additionally, according to some reviews, the ease of use with Fastly is not as pronounced as it is with other providers. This is indicated as well by a private consulting service that Fastly offers in order to fine-tune their service on a client-by-client basis.

On consideration of this last point, this actually plays into the strengths of an organization like Spotify, because they are a company that prefers granular, manual configuration over their technologies. Therefore, it would be reasonable to assume that Spotify is able to achieve a much higher benchmark than the average Fastly consumer, thereby not justifying the costs of migration.

**Benchmarks:** Pingdom, CDNPerf

### Event Handling

Google Pub / Sub is at the center of Spotify's event publishing and data streaming mechanism, it is essentially the nervous system of the data access layer. One other hypothetical solution to this business problem could be the adoption of AWS Lambda. The two are not exactly alike, Pub / Sub's paradigm is to allow different event types to be published and consumed whereas Lambda is simply a single unit of code that gets called via an HTTP endpoint.

However, there is nothing to prevent Spotify from potentially using a single Lambda gateway, that serves as an entry point function that delegates to one of many specific event-handling Lambda functions. There would also be little disturbance to the rest of Spotify's application stack. Since Lambda functions can run Scala code (Lambda supports the running of Scala) the entire stack could be accessed from within a Lambda function. A similar case study is the marketing company Nielsen,

which claim that they process 250 billion events per day using Lambda at the core of their service. This is approximately 2.5 times the load that Spotify currently serves.

However, migrating is always a costly exercise, and it perhaps would only be a move worth considering should there be an unsolvable scaling issue in their current infrastructure with a growing user base.