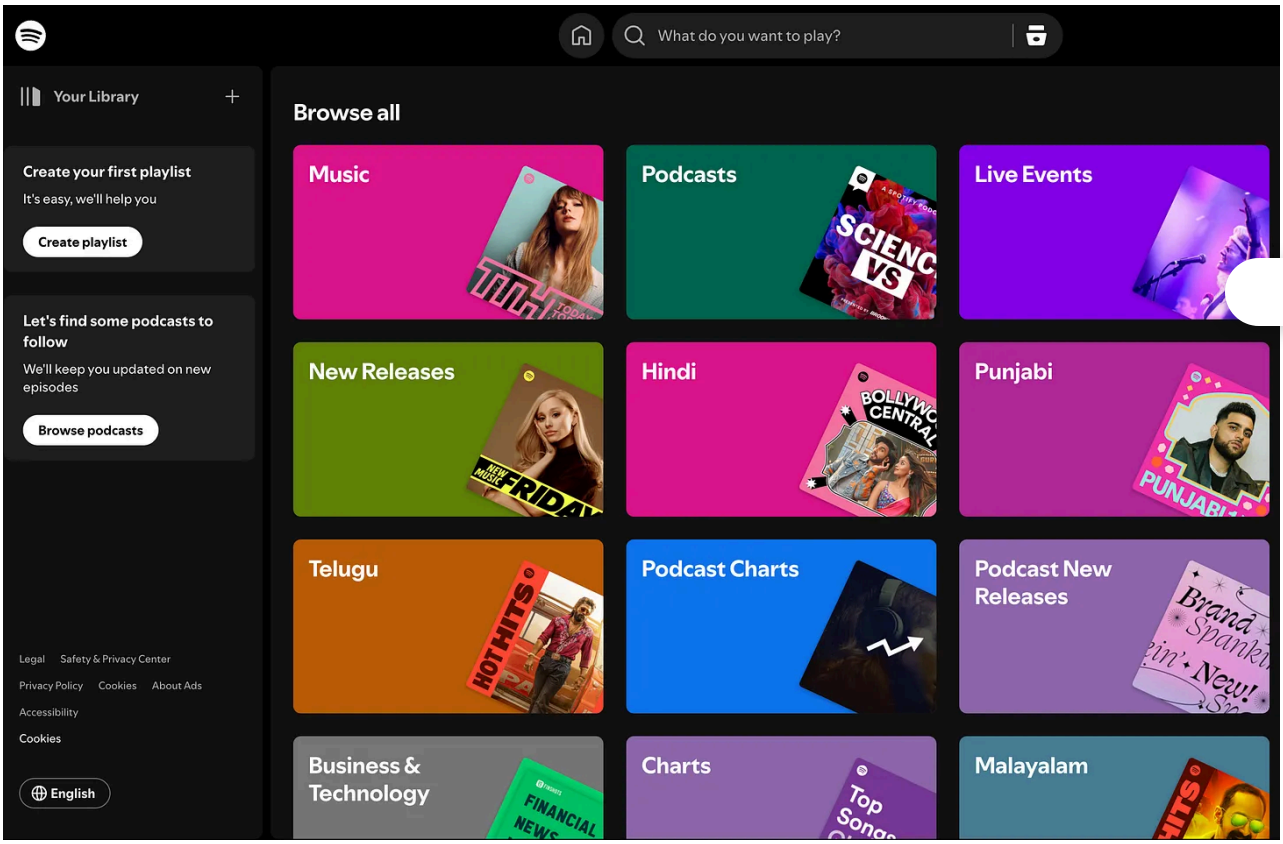
Design Spotify

ASHISH PRATAP SINGH
MAY 02, 2025 • PAID

Share

Spotify is the most popular music streaming platform in the world, with over **600 million** monthly active users (MAU) and **200 million** paid users.



In this article, we will learn how to design a **music streaming service** like **Spotify** tl can handle **100s of millions of users** and **billions of music streams** every day ensuri **low latency** and **high availability**.

1. Requirements Gathering

Before diving into the design, lets outline the functional and non-functional requirements.

1.1 Functional Requirements:

- **Search**: Users can search for songs, artists, albums, and playlists.
- **Music Streaming**: Users can stream songs in real time.
- **Playlists**: Users can create, share, and modify playlists.
- **Music Recommendations**: Users receive song recommendations based on thei listening history and preferences.
- **Ad-Supported Model**: Free-tier users will encounter ads after a few songs.

1.2 Non-Functional Requirements:

1. **Scalability**: The system should handle 100s of millions of users globally and the ability to stream millions of songs concurrently.
2. **Low Latency**: Real-time streaming must have low latency for a seamless user experience.
3. **High Availability**: The system must be available at all times with minimal downtime.
4. **Global Reach**: Support users from different geographic regions, potentially leveraging CDNs to serve audio files faster.

2. Capacity Estimation

Let’s assume the following traffic characteristics:

> **User Base:**
>
> - Total active users: 500 million
> - Daily active users: 100 million
>
> **Average streams per user per day:** 10
>
> **Average song size:** 5 MBs
>
> **Average song duration:** 4 minutes
>
> **Song catalog size:** 100 million songs

2.1 Network Bandwidth Estimation

- Daily song streams = 100M users × 10 songs = 1 billion streams/day.
- Data transfer per day = 1 billion × 5 MB = 5 petabytes/day.
- Data transfer per second = 5 petabytes / 86400 = 58 gigabytes/second

2.2 Storage Estimation

Total storage for music = 100 million songs × 5 MB/song = **500 terabytes.**

Assuming 2 KB of metadata per song and 10 KB of metadata per user (user details, preferences, playlists etc..)

- Total song metadata storage = 100 million songs × 2 KB = **200 GB**.
- Total storage for 500 million users = 500 million × (10 KB) = **5 TB**.

2.3 Caching Estimation

Caching plays a significant role in reducing the load on the storage system and ensuring low latency for popular content.

Frequently played song metadata can be cached in memory.

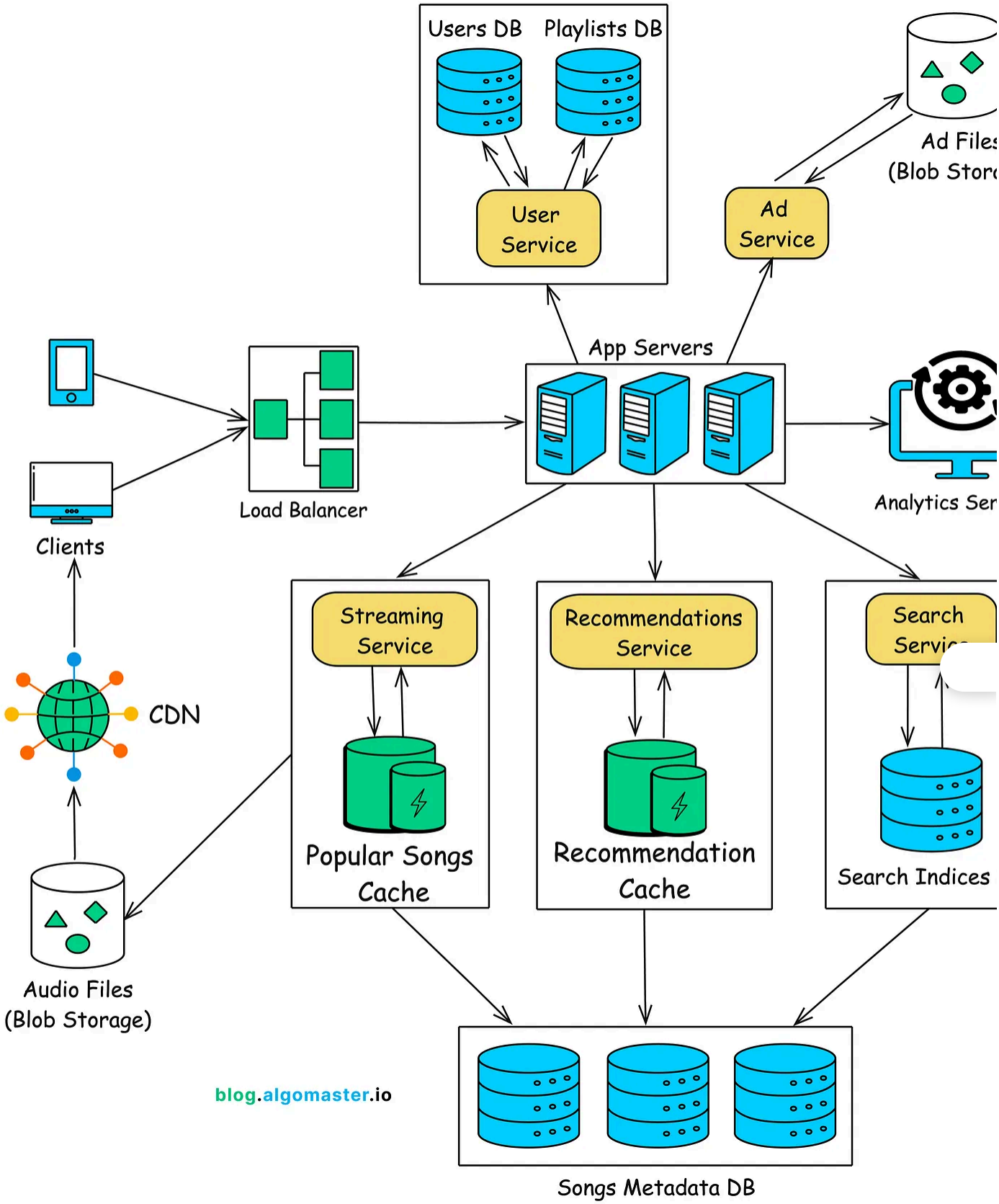
Lets assume top 20% songs contribute to 80% of the requests.

Assuming Spotify has 100 million songs and the top 20% are cached.

- Cache size = 20 million songs × 2 KB/song = **40 GB**.

3. High Level Design

The system architecture of Spotify can be broken down into several high-level components:



3.1 Client Application

The **client application** consists of the **mobile**, **desktop**, and **web** versions of Spotify which provides a clean and intuitive UI to interact with the service.

It communicates with backend APIs for search, streaming, playlists, and recommendations and supports offline listening by caching music on the device (downloaded content).

3.2 Load Balancers

The **Load Balancer** is the entry point for all client requests.

It distributes incoming client requests evenly across multiple instances of backend services, preventing overload on any single server.

3.3 App Servers

Receives incoming requests from load balancer and re-directs the request to the appropriate service.

3.4 Services

- **Streaming Service**: Handles streaming of music from the storage system to use device in real-time.
- **Search Service**: Handles searching of songs, artists, albums and playlists.
- **Recommendations Service**: Provides personalized music recommendations bas on user behavior, such as listening history, likes, and playlist creation.
- **Ad Service**: Handles the delivery of advertisements for free-tier users.
- **Users Service:** Stores and manages user profiles, including personal informatio subscription type, and preferences. Manages user playlist, allowing users to cre modify and share them.

3.5 Storage

- **Databases**: Stores user profiles, playlists, songs metadata and search indices.
- **Blob Storage**: A distributed storage system (e.g., AWS S3) for handling large-sc storage of audio and ad files.
- **Content Delivery Network** (**CDN**): Used to deliver large audio files efficiently t users across the globe with minimal latency.
- **Caches**: Caches frequently accessed data such as popular songs and recommendations to improve performance and reduce the load on the storage a database systems.

3.6 Analytics Service

The **Analytics and Monitoring** service tracks user engagement, system performanc and logs system health.

It generates alerts when issues are detected and logs all system activities for troubleshooting.

4. Database Design

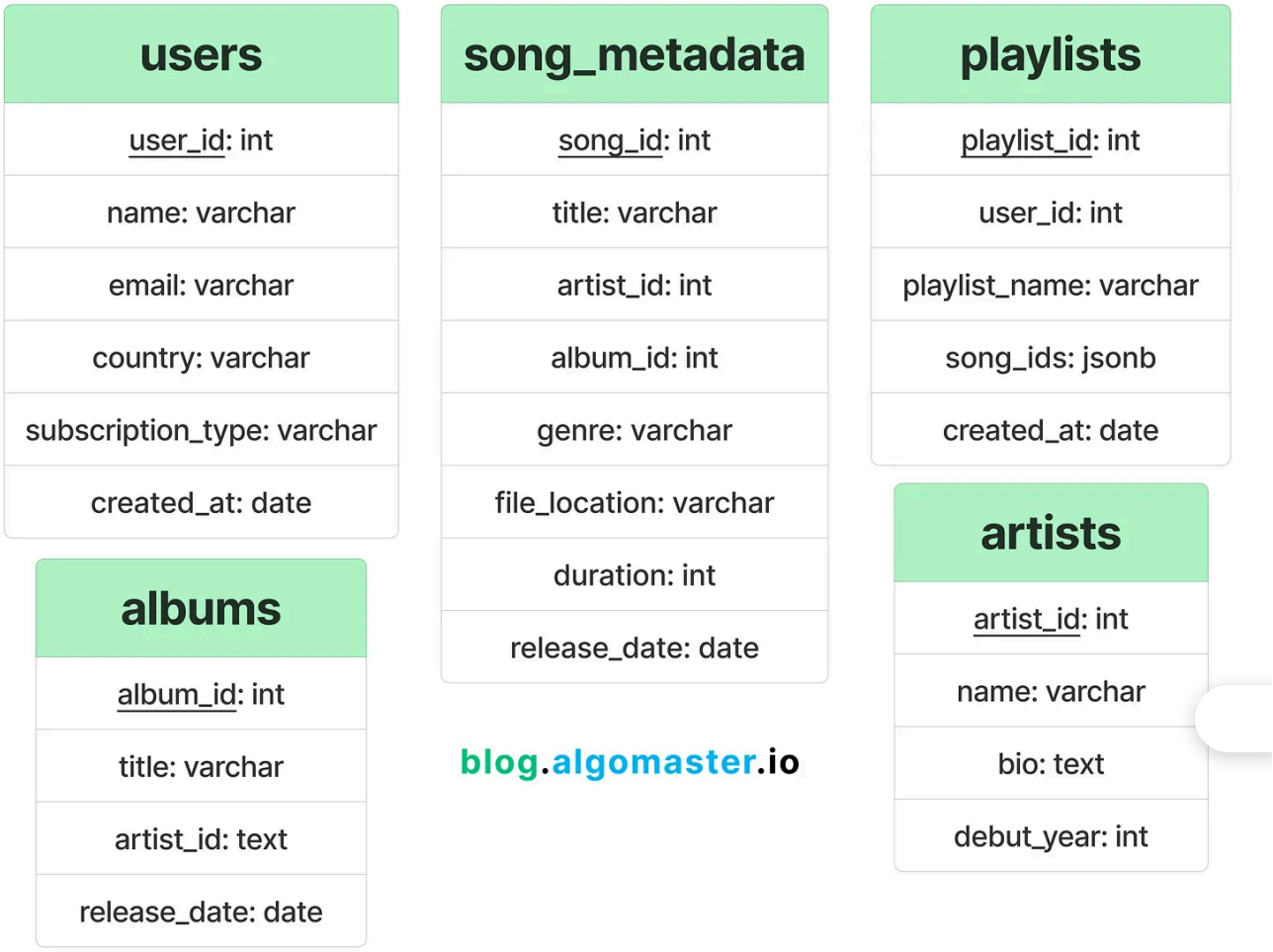
Here are the key entities we need to store in our database:

`Users, Songs, Artists, Albums, Playlists, Streams, Search Ind and Recommendations.`

Given the diverse types of data and high query demands, we use a combination of **relational databases**, **NoSQL databases**, and **distributed storage systems**.

4.1 Relational Databases for Structured Data

To store highly structured data like **user profiles**, **playlists**, **songs metadata**, **artists** and **albums**, we can use a relational databases like **PostgreSQL** or **MySQL**.



- **subscription_type**: Plan type (Free, Premium, Family, etc.).
- **file_location**: URL of the song file in storage (e.g., AWS S3).
- **duration**: Length of the song in seconds.

4.2 NoSQL Databases for Unstructured Data

To store unstructured and semi-structured data, we can use **NoSQL databases** like **MongoDB**, **Cassandra**, or **DynamoDB**.

NoSQL databases provide flexibility and scalability, making them ideal for handling highly **dynamic data** such as recommendations, and search indices.

Recommendations Table

Spotify generates recommendations for users based on their listening behavior and this data is updated frequently.

Example Record:

```
{
    "user_id": "56789",
    "recommended_songs": [
        {"song_id": "12345", "score": 0.95},
        {"song_id": "67890", "score": 0.89}
    ],
    "generated_at": "2024-01-01T10:00:00Z"
}
```

Search Indices

Search indices are stored in NoSQL databases like **Elasticsearch** to allow quick, fuz search queries across songs, artists, and albums.

These indices are continuously updated as new content is added.

Example Record:

```
{
    "song_id": "12345",
    "title": "Shape of You",
    "artist": "Ed Sheeran",
    "album": "Divide",
    "genre": "Pop",
    "release_date": "2017-01-06"
}
```

4.3 Distributed Storage System

To store large volumes of audio and ad files, we can use a **distributed storage syste** like **AWS S3.**

S3 ensures high durability and availability, making it an ideal storage solution f serving large static files.

Example Storage Object:

- **File**: `https://s3.amazonaws.com/spotify/songs/blinding_lights.mp3`
- **Metadata**: File size: 4 MB, Bitrate: 128 kbps, Format: MP3

4.4 Content Delivery Network (CDN)

We use a **Content Delivery Network** (CDN) for distributing large audio files (songs) users globally with minimal latency.

By serving music from CDN edge servers, Spotify ensures **low-latency** music streaming experiences for users across the world, minimizing buffering times and

reducing load on the central storage system.

Original music files are stored in a distributed storage system (e.g., **AWS S3**). The C pulls from this origin storage when a song is requested for the first time and caches for future requests.

4.5 Caching Layer

Caching frequently accessed data like user preferences, popular songs, or recommendations can improve performance.

A caching layer like **Redis** can be used to store this data temporarily.

Examples of Cached Data:

- **Search Queries**: Cache popular search queries to avoid hitting the search index repeatedly.
- **Popular Songs**: Frequently streamed songs can be cached to reduce database queries.
- **User Preferences**: Store the user's liked songs and playlists in the cache for fast retrieval.

Example - SET/GET queries for User Preferences in cache:

```
SET user:preferences:12345 "{liked_songs: [1, 2, 3], playlists: [10, 11
12]}"

GET user:preferences:12345
```

4.6 Analytics and Monitoring Data (Data Warehousing)

Analytics and monitoring data are critical for tracking user engagement, system performance, and identifying potential issues.

Data is aggregated and processed in a **data warehouse** or **distributed data stores** (e. Hadoop, BigQuery).

Key Use Cases for Analytics:

- **User Engagement**: Data on streams, skips, and playlist additions are used to generate insights into user behavior.
- **System Monitoring**: Logs from various services are used to monitor system hea detect anomalies, and perform performance tuning.
- **Royalty Calculations**: Streaming data is used to calculate payments for artists based on song plays and geographic reach.

Stream Log Example:

```
{
    "user_id": "56789",
    "song_id": "12345",
    "device": "iPhone",
    "stream_duration": 180,
    "stream_time": "2024-01-01T10:00:00Z"
}
```

5. API Design

We'll design RESTful APIs that are intuitive, efficient, and scalable.

Let's break down our API design into several key endpoints:

5.1 Search API

The **Search API** allows users to search for songs, artists, albums, or playlists. The search results are ranked based on relevance, popularity, and user preferences.

Endpoints

`GET /search`

Query Parameters:

- `query`: The search term (e.g., "Blinding Lights").
- `type`: The type of resource to search for (`song`, `artist`, `album`, `playlist`).
- `limit`: Maximum number of results to return (default: 20).
- `offset`: For pagination (default: 0).

Response:

```
{
  "results": [
    {
      "type": "song",
      "id": "12345",
      "title": "Blinding Lights",
      "artist": "The Weeknd",
      "album": "After Hours"
    },
    {
      "type": "artist",
      "id": "67890",
      "name": "The Weeknd"
    }
```

```
    ]
}
```

5.2 Music Streaming API

The **Streaming API** handles the delivery of music files from the backend or CDN to the user's device.

Endpoints

`GET /stream/{song_id}`

Response:

- **HTTP 302 Redirect** to the CDN URL where the song is hosted:

```
{
  "url": "https://cdn.spotify.com/song/12345"
}
```

5.3 Recommendations API

The **Recommendations API** provides personalized song suggestions based on the user's listening history, preferences, and likes.

Endpoints

`GET /recommendations/{user_id}`

Query Parameters:

- `limit`: Number of recommendations to return (default: 10).

Response:

```
{
  "recommendations": [
    {
      "song_id": "12345",
      "title": "Blinding Lights",
      "artist": "The Weeknd",
      "score": 0.98
    },
    {
      "song_id": "67890",
      "title": "Can't Feel My Face",
      "artist": "The Weeknd",
      "score": 0.95
    }
  ]
}
```

5.4 Ad Delivery API

For free-tier users, Spotify injects advertisements into their listening experience.

The **Ad Delivery API** fetches and serves personalized ads based on user preferences and demographics.

Endpoints

`GET /ads/{user_id}`

Fetch ads for a user to be played during music streaming.

Response:

```
{
  "ad_id": "ad12345",
  "ad_url": "https://cdn.spotify.com/ads/ad12345.mp3",
```

```
  "duration": 30
}
```

6. Diving Deep into Key Components

6.1 Music Streaming Service

The **Streaming Service** is at the heart of Spotify's architecture, responsible for delivering music content efficiently, securely, and reliably to millions of users in rea time.

The actual delivery of music files is managed by a **Content Delivery Networks** (Cloudflare, AWS CloudFront). This ensures that music is served from geographical distributed servers close to the user, minimizing latency and bandwidth consumpti

Request Workflow:

1. Client sends a streaming request (e.g., `/stream/{song_id}`).
2. The App server authenticates the user and routes the request to the Streaming Service.
3. If the song is not in the CDN, the Streaming Service retrieves the audio file's location (from the blob storage) and pushes the file to the nearest CDN edge server. The CDN returns a URL to the streaming service to stream the audio.
4. The CDN URL is returned to the client, allowing the client to stream the audio

6.2 Recommendation Service

The recommendation system analyzes the user's listening habits, likes, and playlists uses a combination of collaborative filtering (based on users with similar preferenc

and content-based filtering (based on song metadata).

Collaborative Filtering

Collaborative filtering is one of the most commonly used techniques in recommendation systems. This method leverages the behavior of users with similar music tastes to generate recommendations.

- **User-Based Collaborative Filtering**: This technique groups users based on their listening history. For example, if User A and User B both frequently listen to the same set of artists and songs, the system may recommend songs that User A has listened to but User B hasn't.
- **Item-Based Collaborative Filtering**: In this technique, songs are recommended based on their similarity to songs the user has previously liked. If many users who like Song X also like Song Y, the system recommends Song Y to users who have listened to Song X.

Content-Based Filtering

Content-based filtering focuses on the properties of songs, such as **genre**, **artist**, **album**, **tempo**, and **instrumentation**, to recommend similar songs to users.

- **Song Attributes**: Spotify collects metadata on each song, including genre, tempo, mood, and instruments. This data is used to recommend songs with similar attributes to what the user has already liked or listened to.
- **Artist Similarity**: If a user listens to multiple songs from a particular artist, the system may recommend songs from similar artists, based on shared attributes (e.g., genre, style).

6.3 Search Service

The **Search Service** in Spotify allows users to find songs, artists, albums, playlists, and podcasts from a vast catalog efficiently.

The architecture of Search Service can be broken down into the following key components:

1. **Query Parser**: Interprets and normalizes the user's search query.
2. **Search Index**: A dynamically updated index that contains metadata for all song artists, albums, and playlists. A search engine like **Elasticsearch** or **Apache Sol** can be used to build and manage this index.
3. **Ranking Engine**: Once the search index returns matching results, the **Ranking Engine** sorts the results to ensure that the most relevant results appear at the t
4. **Personalization Layer**: Customizes search results based on the user's listening history, preferences, and demographic information.
5. **Search Autocomplete**: Provides users with suggestions as they type their queri speeding up the search process.
6. **Cache Layer**: Caches frequently searched queries to improve performance and reduce the load on the backend.
7. **Search Index Updater**: Ensures that the search index stays up to date with new content being added to Spotify's catalog.

7. Addressing Key Issues and Bottleneck

7.1 Scalability

- **Sharding**: To scale the SQL databases and distribute the load evenly, implemen sharding for large tables like user, playlist and song metadata.
- **Indexes**: Add indexes on frequently accessed fields like user_id and playlist_id improve query performance.

- **Partitioning**: NoSQL databases can use partitioning strategies to distribute da across multiple nodes, ensuring low-latency access even at large scales.
- **TTL (Time to Live)**: Cached data is given a TTL to ensure that stale data is regularly invalidated.

7.2 Reliability

To ensure high availability, Spotify should implement fault-tolerant systems:

- **Replicated Databases**: Replicate user, song and playlists data across multiple d centers to prevent data loss.
- **Cache Replication**: Redis can be configured to replicate cached data across multiple instances for fault tolerance.
- **Auto-scaling**: Automatically scale the number of servers based on traffic load.
- **Graceful Failover**: If a server fails, traffic is rerouted to another server wit service interruption.
- **Monitoring and Alerting:** Implement comprehensive monitoring and alerting systems.

7.3 Security

Spotify handles sensitive data such as user profiles and payment information.

- **Authentication**: Use OAuth 2.0 for secure user authentication.
- **Encryption**: Encrypt all sensitive data in transit and at rest.
- **Rate Limiting:** Rate limit users to ensure that excessive API requests from a si client are throttled to protect the system.
- **Data Privacy**: Implement strong access controls to ensure user data is not leak or misused.

Thank you for reading!

I hope you have a lovely day!

See you soon,
Ashish

© 2025 Ashish Pratap Singh · Privacy · Terms · Collection notice
Substack is the home for great culture