Prototype Music Distribution System

In this talk, I present my repo, *ice-exercise*

I'll outline:

- the overall approach
- the domain objects
- principal design and implementation decisions
- what would be needed for a production implementation

The project is about 600 lines of Scala.

The assignment

To summarise the spec:

The requirement is to write the core of a Music Distribution System
as the initial base of what could eventually become a
production app capable of managing the back-end distribution,
management and billing for a platform such as Spotify.

There are 9 required "behaviours" to be implemented as features, e.g.

```
- artist adds songs to a release
- system records the streaming of a song
```

The size of the assignment: It's intended to take no more than a day or two.

Overall approach

I considered test-driving the code. This seemed infeasible given the time frame.

So it seemed best to have a "main driver" app which can be demonstrated by running these commands to check out the repo and run it using SBT:

```
git clone https://github.com/fdilke/ice-exercise.git
cd ice-exercise
sbt run
```

This will run the "driver" program MainDriver which sets up a *MusicDistributionService*, invokes the desired behaviours, and prints the results.

In this presentation, I focus on the higher level concerns, then drill down into the detailed assumptions in an appendix.

Modelling the domain

From analysing the requirements, I extracted these business domain objects:

```
Artist: a legal entity (e.g. a band) who produce and release songs

Song: a named datum of music. Streamable, has a length.

Streaming: an instance of somebody listening to all or part of a song.
    If <= 30sec, it doesn't count towards monetization, but is still recorded for reporting purposes.

Release: an ordered collection of songs. Typically an EP or album.
    A release can be removed from distribution.
```

These are all modelled as discrete value objects and held in local storage.

Modelling the domain, continued

These additional concepts are part of the business vocabulary, but not modelled as domain objects:

```
Report: a list of streamed songs.

    For simplicity, this is modelled as CRLF-delimited text.

File for payment: an artist recording a request to be paid
    for the songs played since the last payment date.

    I store this information as part of the data for the artist.
```

Service classes

I have two service interface classes:

- *MusicDistributionSystem*
- *MusicStorageService*

These are separated from their respective implementations:

- *PrototypeMusicDistributionSystem*
- *LocalMusicStorageService*

The *PrototypeMusicDistributionSystem* takes a *MusicStorageService* interface as a construction argument.

Storage

The *LocalMusicStorageService* implementation stores all the domain objects in memory behind its service interface.

The objects are stored only while the application is running, which seemed acceptable for a prototype.

In a production implementation, this would likely be backed by a cloud-based database instead.

Also some of the operations (e.g. searching for songs) are currently efficient only for a small dataset, and would likely be replaced by optimized SQL queries.

Referring to domain objects

I use the convention that IDs are generated for all domain object instances, and these are used as permanent references for the object.

The APIs allow you to load data structures (e.g. Artist, Release, etc) representing them, but those are intended as transitory data.

The intention is that a database would store all of these IDs in its tables as foreign keys.

It's also possible the database is sharded across multiple locations and only provides eventual consistency, rather than being entirely locally consistent during all transactions.

The main driver program

With this architecture in place, the main driver program instantiates and manipulates domain objects, referred to by their IDs, using the specified behaviours.

It registers an artist, a fictitious band called The Clueless Tea Boys, then registers two releases for them, an EP and an album, and populates these with songs.

For the album "One Lump or Two?" a release date is proposed and approved by the record company.

No such dates are set up for the EP, "Afternoon Picnic".

Producing reports

The driver program generates a report of all releases, showing their streamability.

We can see both releases, the album being streamable and the EP not.

Another report shows all the songs with an indication of streamability.

We can see that on the unreleased EP, no songs are streamable, while all the songs from the approved and released album are now streamable.

The logic considers a song streamable if it's included in a release which is streamable.

Searching for songs

The driver program does a search for the pattern "Madeira", specifying that the top three results should be returned.

This correctly finds the streamable song "Madeira Cake".

In the current search implementation, the logic runs over the name of every streamable song in the entire database, calculating the Levenshtein distance of each one from the search pattern.

Again, this is OK for a prototype using a small data set, but a production version would need to be more sophisticated.

(Perhaps a trie tree or some other data structure would enable a more efficient search. Alternatively, maybe the database understands Levenshtein natively, or one could write a PL/SQL method to calculate it.)

Note on implementing Levenshtein

This seems to be a standard algorithm, which I wasn't previously familiar with.

I was surprised to learn that (as with Fibonacci numbers) a straightforward recursive algorithm isn't very efficient, even for small datasets as used here, and some form of memoization is needed.

I experimented with several algorithms from open-source listings (see notes in the README) and adapted one which seems fast enough for this demo.

APPENDIX: More detail on assumptions

These are all documented in the repo, but I describe them in a little more detail here.

Several may not be quite right as my understanding of the domain is incomplete. It seemed better to write them up explicitly and continue, rather than stop for guidance.

Assumptions: references and IDs

We refer to domain objects by IDs which are just wrapped strings.
The storage service allocates unique IDs which are just sequence numbers
with a prefix, e.g. "artist0".

The intention is that the domain objects will mostly be referred to by their IDs,
which will be passed around for reference, and occasionally have fields loaded
or modified using the storage service.

As implemented, there is only one ongoing sequence number,
so the labelled IDs for different types of domain object will all have
different numbers as well as different prefixes across types.
This was not strictly necessary but seems OK as an additional safeguard
that also saves memory.

Assumptions: artists

For artists, we just store the name (typically a band name) and categories, which are just keywords indicating the type of music.

For each artist, we record the dates on which payments are requested and made.

Assumptions: songs

For now there is no need to store any music or actually stream anything for the songs. We just record their names and how much was streamed.

In production, the songs would be stored in the storage service, which would also be responsible for streaming them.

I'll assume that when the system is notified of a streaming for a song, the song has been released, i.e. is included in a release with an agreed release date lying in the past.

Assumptions: releases

Releases have a name, description, and optional proposed/confirmed release dates.
They also store the ID of the associated artist, which I'd expect to map to a
foreign key in a database backed storage implementation.

We store the artist ID for a release, but not for a song. Conceivably, songs
could appear as part of multiple releases for multiple artists.

We'll store the list of song ids in the data for a release.
They are in a specified order, as for a CD.

There is no provision for separating blocks of songs within a release,
as might be useful for sides of an LP, or for naming composite pieces.

Assumptions: releases (2)

When a release is removed from distribution, then rather than remove it from the storage service, I set an additional flag so that it is not considered streamable and won't appear in searches.

This makes the process reversible and ensures compatibility with other APIs such as the streaming report.

Assumptions: streamings

For each streaming, we record the song and how long it was played for.
We won't record anything about the users.

The spec seems to indicate that we should keep track of ALL streamings
(even short ones) and indicate the monetizable ones (> 30 sec)
in the reports.

I'm assuming that each streaming event for an individual song includes a
number of seconds, which is truncated to an integer.

Also that these aren't to be coalesced or accumulated, so if you listen
to lots of tiny snippets of a song, the artist isn't credited.

Assumptions: searches

A search will return up to a specified number of songs whose name matches a given text pattern, ordered by Levenshtein distance.

Assumptions: payments

Since we have a concept of payments being requested and made for streamings in a given time period (between two dates), we must record the date of each streaming.

I'm assuming this level of resolution for the streaming time is enough.

Although it's not explicitly requested, I added an API for the record company to update these when they make a payment, as this audit record would likely be of interest to both parties.

Assumptions: reports

For a prototype, the "streamed songs report" for an artist can be a CRLF-delimited string.
I considered returning a more elaborate data structure where you could step through the rows, but a simple text report seems adequate.

I'm interpreting the spec for the report to list all streamed songs for the specified artist, with an indicator whether they're monetizable or not.
I used a £ sign to indicate they are, else a dash.

I include all songs in the report which appear in a release for the specified artist.
In a production system, possibly some extra logic should be added here to take account of releases which are removed or not yet confirmed for release.
It's possible also that a song could be included in releases for multiple artists.

Handling dates

A release can have an optional proposed release date, and an agreed release date. The intention is that at most one of these is defined, but that isn't enforced.

For simplicity I'm storing dates (such as proposed release dates) as *LocalDate*, so for the purposes of this exercise I'm glossing over the management of timezones, or just assuming that all dates are rendered locally to wherever they are being processed.

For a production app operating globally, this would need to be more sophisticated.

Additional code details

In the API I include convenient methods withRelease(), updateRelease(), etc. The repetition here could be eliminated, at the cost of some additional complexity and type trickery. For now, it doesn't seem worth it.

I note that Levenshtein distance on its own might not be the best calculation; a search for "Crumpets" didn't match very well with "Crumpets (Disco Remix)", suggesting a more sophisticated algorithm that takes account of prefixes.

For consistency I named all the "store" methods the same way: storeSong(), storeRelease(), etc. I briefly considered calling the one for streamings trackStreaming(), to emphasise something about the way we record streamings. Consistency seemed better.

Additional code details (2)

Many of the APIs can simply be delegated to the MusicStorageService, to such an extent that the MusicDistributionSystem implementation is really quite a thin layer. It still seems a worthwhile separation.
In a production system I'd expect that the MDS would need to include further interactions with other services such as payment gateways.

I use a class *Id[<CLASS>]* to tag the IDs, so we can type-safely have *Id[Song]* etc without repeating any code.