Git Last updated: Nov 10, 2022

Pathfinder Schema Style Guide

This describes how the Pathfinder schema should be constructed, with concrete examples on how the schema can look like.
Start by skimming over the whole documentation to get a feel for what we want to achieve and then please refer to specific rules headings for details. We also have a “how to” heading that describes common schema design problems and their solutions.

Prerequisites

You should know the basics of GraphQL schema design before you read this, eg. what is a type, difference between input type and regular type, how to use directives etc., a good place to start learning from would be the official graphql.org tutorial. Another thing to familiarize yourself with is apollo apollo-federation specifications which Pathfinder is based on.

Why are we doing this

As Pathfinder adds more and more federated services, it becomes more important to have a set of rules on how all the federated service schemas should be constructed. We want the clients to have a consistent experience when they are constructing their queries against the composite Pathfinder schema. These are the kind of things that we want to avoid:

- Arguments and fields that have different case rules depending on where they are defined in the schema.
- Fields that basically have the same meaning are named differently depending on in which type the field is defined in.
- General concepts like pagination is handled differently in different parts of the schema
- Names of fields and types are cryptic and requires domain knowledge to decode and use correctly
- Error handling work differently in different parts of the schema

Updating this document

This is a live documentation, and we expect rules to be refined and further clarified. If you have comments, questions and/or suggestions: send your comments to the #pathfinder-schema slack channel, where also all contributors can monitor or be notified about any modification in the rules.

Table of Contents

Conventions for examples

We use “…” to indicate that some parts of the schema text have been removed, in order to not bloat the documentation. The examples may differ from the actual schema, either to clarify a point or because these examples are more up to date.
We have also removed the comments from the schema types and fields to keep the examples short.

The basics

The basic rules are generally applicable to any kind of GraphQL Schema, not specific to Pathfinder.

Case rules

We use camelCase throughout the schema for everything except enum fields. Example with types, fields and parameters:

```
type Chapter {
 audio(limit: Int = 10, offset: Int = 0): AudioPage!
 ...
 sharingInfo(customData: [KeyValueInput], utmParams: UtmParamsInput): SharingInfo
}
```

Declaring directives should be at the beginning of the schema and configuring directives should be the same across the federated services, for example:

- `@key` This directive is the key for retrieving the entity data across the different services
  `directive @key(fields: _FieldSet!) repeatable on OBJECT | INTERFACE`
- `@complexity` This is the complexity weight of the field when running the query, if the complexity of the query is too high, the query will be prevented from running.
  `directive @complexity(weight: Int!) on FIELD_DEFINITION`
- `@deprecated` This is used to deprecate any field, argument, input field or enum value.
  `directive @deprecated(reason: String = "<The reason for deprecation>") on FIELD_DEFINITION | ARGUMENT_DEFINITION | ENUM_VALUE | INPUT_FIELD_DEFINITION`

All fields start with a lowercase letter; and types, unions, interfaces and enum types start with an uppercase letter.

Enum values are all uppercase letters and use underscore as a separator:

```
enum ShowType {
 SHOW_TYPE_ADAPTATION
 SHOW_TYPE_EXCLUSIVE
 SHOW_TYPE_ORIGINAL
}
```

There is one exception, when naming fields that are not meant for the client to read directly, they should start with an underscore. These fields are for internal use, the most common example of this are our wrappers[1], if you're not sure what whether to add _ in the field or not you can consult #Bassline

[1] These fields should really be completely hidden from the client when they make queries, and the next version of the GraphQL Federated Spec will have a directive to support this.

```
type ArtistResponseWrapper
@key(fields: "_uri")
{
 _uri: ID!
 data: ArtistResponse!
}
```

Recommended Sorting

We strongly recommend sorting the graphQL schema with the Maven SortGraphQL plugin, by adding the plugin to the pom file.

It will do the following:

- Sort all types, all fields and all enum values in alphabetical order.
- Sort sections of the schema in the following order:

1. All Directives first
2. The Query type
3. The Mutation type (should you have one)
4. The Subscription type (not applicable for Pathfinder)
5. All Scalars
6. All Interfaces
7. All Unions
8. All Input types
9. All Types (beside Query, Mutation and Subscription)
10. All Enums

Types, enums, unions, interfaces should be sorted alphabetically; or in an order that describes the intent of the schema[2].

Conventions for string content in fields

We use the following conventions:

- A field called `uri` should contain a Spotify URI (eg. `spotify:concert:2g5H1retVYFj2oJR5UknLZ`). If the field is not a Spotify URI, it should have a different name with descriptive prefix or suffix.
- A field called `url` usually contains a full web link (eg. `https://i.scdn.co/image/ab6761e0202`).
- A date should be represented as a `Date` type, otherwise it **must** have the full date format documented as field description.
- String fields that have a specific format **must** have the format documented as field description, eg. `# Client time zone, e.g. "Europe/Stockholm", "America/New_York".`
- Use the scalar `ID` for fields that contain id values, as this makes for an easy identifier indicator.

---

[2] Describing the intent is having the root type(s) towards the top and its **child types towards the bottom**. Bassline started with this kind of ordering, but realized that a purely alphabetical order was easier to browse and maintain.

Pagination

All lists should use pagination types[3]. It is far easier to include pagination from the start than to try to retrofit it later if the client only needs a small number of items. The pagination should look like this:

```
type ArtistVisuals {
 gallery(offset: Int = 0, limit: Int = 10): ImagePage!
}

"A paged view of Image objects"
type ImagePage {
 items: [Image!]!
 "Info about the paged view and its surroundings"
 pagingInfo: PagingInfo!
 "Total number of items"
 totalCount: Int!
}

type Image { ... }
```

- The field that uses the pagination has two parameters, offset and limit. Decide on a reasonable value for the default limit[4]. Note that clients should not use the default value for limit (they should specify what they want), but the default will give a hint to the client what a reasonable value can be. This also means that you as a service provider must be prepared to handle all kinds of values.
- The Page type has the suffix "`Page`" and it contains exactly the fields `items`, `pagingInfo` and `totalCount` (the comments can be reused as well). The prefix in the Page type is singular, e.g. no `ImagesPage`
- PagingInfo is a predefined type that can be copied from most other federated services
- The pathfinder-federation library has java helper classes for the implementation of the pagination logic
- `totalCount` isn't nullable and the `items` as a field and each element in the array are not nullable.

The current type of pagination is offset-based and we have neither defined style guides for token-based pagination nor implemented one yet. If you would need token-based, then contact #Bassline to cooperate on how that would look.

If the list is not paginated (for some reason), it should still be a non-null list containing non-null items in it. It is not recommended to have lists which contain null items, and it is better to return an empty list than returning null as the list value.

---

[3] There are cases when the list size is static and we never see that it would change in the future, it that case it may be ok not to have a paginated list

[4] Sometimes the limit and offset parameters are placed on the parent list owning type, more about that in the how-to heading

When not to use pagination

If the returned list matches the size of the input list, pagination is unnecessary. The expected number of entities in the response should be equal to the number of entities in the request.

Common type name suffixes and prefixes

We encourage prefixing feature (or domain) types with the same prefix in order to avoid name collisions. Mutation input types and response types are an exception to this rule, since they are named after the mutation operation itself, see Mutations.

Some examples are:

- All types, unions, enums in the oxygen-home service are prefixed with “Home”
- The prefix “Browse” is used for the federated service browse-graphql.
- Oxygen-Library uses the prefix “UserLibrary” for most types, with the exception of mutation input and response types (more about those later in the Mutations heading).

Unions that represent the result of decoration or results from queries should have names that end with “Response”. This is commonly used for federated decoration with wrappers, ie. when a uri-producing service populates a wrapper and a decorating service is later looking up data for that wrapper. The response part is that the decoration could result in states like “not found” or “restricted content” depending on the result of the upstream call. Example:

```
type TrackResponseWrapper
@key(fields: "_uri")
{
 _uri: ID!
 data: TrackResponse!
}

union TrackResponse = GenericError | NotFound | Track
```

Other unions and also interfaces should have names that indicate an abstract representation of possible types. They do **not** need any “Union” or “Interface” suffix.
The “Page” suffix has been described in the Pagination heading. The “Wrapper” suffix will be further explained in the Wrappers heading. “V2” suffix will be explained in the Versioning heading.

We have an ADR that explains more about the reasoning behind Naming Conventions.

Entities vs Value types (Shared Type)

Federated types can be either Entities or Value types (aka. Shared types).

Entities are owned by a federated service and have one or several key fields. Adding a prefix where the shared type is, helps identify where the ownership lies (ex. Browse…, or Home…).

The key declaration can be nested, ie. using the field of a child entity as key[5]. An entity can be extended with additional fields by another federated service. Example where artist type is extended with goods field:

```
# Specified in one service
type Artist
@key(fields: "uri")
{
 discography: ArtistDiscography!
 profile: ArtistProfile!
 stats: ArtistStats!
 ...
 uri: ID!
 visuals: ArtistVisuals!
}
```

—

```
# Extended in another service
type Artist
@extends
@key(fields: "uri")
{
 "Physical goods that pertain to the artist"
 goods: ArtistGoods!
 uri: ID! @external
}
```

Another common example of entities are our wrappers, described under the Wrappers heading.

The Value type is a generic type that is shared across services and they must have identical definitions in all services that use them, otherwise the schema validation will fail. Example:

```
type Date {
 day: Int
 hour: Int
 ...
 year: Int!
}
```

In case you'd like to change an existing value type, it's important to inform all the teams who use it, if you're stuck you can consult #Bassline.

When should you use one or the other?

If the type has a clear key field, then set the key directive to it. It will make the type work better in a federated environment, since other services can add further functionality to the type by adding custom fields to it. For example, oxygen-artist-goods add fields to the existing type artist:

---

[5] Too many levels of nesting will not work, see heading about complex keys

```
type Artist
@extends
@key(fields: "uri")
{
 "Physical goods that pertain to the artist"
 goods: ArtistGoods!
 uri: ID! @external
}
```

If the type does not have a clear key or it is intended to be generic, then omit the key directive to make it a value type. Note that a feature specific value type must have the feature prefix to its name to avoid future name conflicts.
Collision scenario: you create a DateFormat type because you need it for album releases and another service also starts to use it for copyright date formatting. This means that neither of the services can modify the type, since any change would be rejected by the registry (the types must be identical)[6]. Using two different value types with separate prefixes would have made the intention clear instead and make the type safe for future changes.

New non-domain specific value types can of course be added to the schema, and that should be done together with Bassline.

We have an ADR that explains more about the reasoning behind Shared types (Value types).

Comments and Descriptions

Pathfinder supports both comments and descriptions in the federated service schemas.

Comments are meant for the developer of the service and they will not be shown to the client. They are prefixed with a hash sign. Example:

```
# This enum corresponds to the upstream protobuf definition, see file xxx.proto
enum Things {
   SOMETHING,
   OR_ANOTHER
}
```

Descriptions are meant for all the clients of Pathfinder and they will be visible in different client tools, such as the pathfinder-explorer site (pathfinder.spotify.net) or the GraphiQL tool. Multi-line descriptions are surrounded with triple double-quotes """ and single line descriptions are surrounded by a single double-quote ". Example:

```
"definition of union representing a network call response"
union EpisodeResponse = Episode | GenericError | NotFound | RestrictedContent
```

[6] This can be resolved by modifying both schemas at the same time, with additional pinning of both services and composed schema (aka. tedious manual handling with headache!).

```
type Album  {
   """
   List of the countries in which the album is available, identified by ISO 3166-1 alpha-2 code.
   An album is considered available in a market when at least 1 of its tracks is available.
   """
   availableMarkets(limit: Int = 200, offset: Int = 0): AvailableMarketPage!
}
```

Unlike most examples in this document, you should try to add a description to every type, union, interface and enum. Field descriptions are also very useful for avoiding client misunderstanding what your fields actually do. If a string field has some specific formatting (eg. timestamp), then you **must** provide documentation for that.

We don't recommend mixing comments and descriptions of the type or field. If you need both, you must place the comment above the description otherwise the description will disappear:

```
# This type corresponds to upstream definition xxx
"A speaker represents a device ..."
type Speaker {
 uri: ID!
}
```

We have an ADR that explains more about the reasoning behind Comments & Descriptions.

Mutations

Our way to represent mutations is to start with an action verb first; then the name of the entity that the action is being applied to; and finally the domain (or feature) of the entity.
Format: verb + entity + (To/In/For/At etc.) + domain

Examples of correct naming:

```
type Mutation {
 addItemsToPlaylist(input: AddItemsToPlaylistInput!): AddItemsToPlaylistResponse!
 addItemsInUserLibrary(input: AddItemsInUserLibraryInput!): AddItemsInUserLibraryResponse!
 moveItemsInPlaylist(input: MoveItemsInPlaylistInput!): MoveItemsInPlaylistResponse!
 removeItemsFromPlaylist(input: RemoveItemsFromPlaylistInput!): RemoveItemsFromPlaylistResponse!
 removeItemsInUserLibrary(input: RemoveItemsInUserLibraryInput!):
    RemoveItemsInUserLibraryResponse!
}
```

A suggestion for creating a mutation type is, to use a single non-null input type for it instead of listing the parameters directly in the mutation. This can make the schema design easily extendable (and easily documented) by adding more fields to the input object rather than updating the whole structure of the mutation to include more fields. Similarly, you can deprecate unnecessary fields in your input object.

Try to always use default values in input fields, as this will make the input type more backwards (and forward) compatible. It also encourages clients to use sensible values in queries. Try to keep the fields nullable, except when the field is critical to the mutation. Example:

```
input MoveItemsInPlaylistInput {
 newPosition: PlaylistItemMoveType = TOP_OF_PLAYLIST
 playlistUri: String!
 uids: [String!]!
}
```

The return type of the mutation is a non-null Response union that should return the new state of the entity or the domain. This offers the option to clients to get the updated state in the same call instead of initiating a new one to get it. The return value can be paired with a success flag if needed. Example:

```
union MoveItemsInPlaylistResponse = GenericError | MoveItemsInPlaylistPayload | NotFound

type MoveItemsInPlaylistPayload {
 playlist: PlaylistResponseWrapper
 itemSuccessfullyMoved: boolean
}
```

Both input and response to mutations should be named after the mutation operation with suffix “Input” and “Response” respectively. We do not recommend reusing input or result types, as this could cause conflicts as the mutations evolve over time.

The intermediate

These points are still quite easy to implement, but are mostly specific to how we do things in Pathfinder

Wrappers between services

A producing service generally returns lists of uris, eg. oxygen-search
A decorating service will typically take a uri and populate data for that type, eg.oxygen-podcasts
You can also mix when necessary, eg. the service oxygen-library which decorates some fields as well as produces uris

During decoration, entities are decorated with data, but we also need a way to return to the client that the uri could not be decorated due to restrictions, network errors or missing data. Just returning a null is not enough as it doesn’t give the client understanding what went wrong, but a type like RestrictedContent would surely help, for more info check out this ADR.

We use wrappers to pass uris to the decorating service and give the decorating service the chance to return different states back to the client.

All decorations of complete entries use wrappers when decoration is done in another service. A wrapper has a non-null key field (usually the spotify uri) and a non-null data field that is a union between different responses such as NotFound, GenericError, RestrictedContent; and the concrete type that should be returned. Example:

```
#as written in decorating service
type AlbumResponseWrapper
@key(fields: "_uri")
{
 _uri: ID!
 data: AlbumResponse!
}

union AlbumResponse = Album | GenericError | NotFound

---

"as written (and used) in producing service"
type AlbumResponseWrapper
@extends
@key(fields: "_uri")
{
 _uri: ID! @external
}
```

Given that the producing service populates the _uri key field, the decorating service will populate the contents of the data field.

A decoration of a single field in an existing entity will usually not use wrappers, but the consequences are that the field will simply return null if it cannot be decorated. One example of this is oxygen-artist-goods adding the field goods to the type artist.

We have ADRs that explain more about the reasoning behind Error responses and Wrappers. The union pattern is inspired from this blog post.

The name of the wrapper should correspond to the concrete type that it is returning, not the type of the key.

If a service is returning a union of entities, eg. PinnedItem, then all items in that union should be wrappers (even if the type is decorated locally in the same producing service). This is partly for consistency (client perspective) and partly for future migrations where local decorating functionality may move to a new service.

Nullability of fields

Maintainers of the schema prefer nullable fields and clients prefer non-null fields. Nullability can be an issue for pathfinder clients because it forces them to write boilerplate null-checks in their code. On the other hand, nullability makes it easier to break apart fields to different services, works better with a deprecation process and lets us handle errors in upstream data in a better way.

When you try to return a null value in one of the federated services for a non-null field, the null result bubbles up to the nearest nullable parent (from apollographql blog), which means the client can miss a lot of data, but might still get some other data, so you can just ask yourself for each field: do we return the other data that we have together with a null value, if the upstream somehow would return a null in the field?

Upstream data could be corrupt, or network issues may give us a null value. Setting null in a non-nullable field throws an exception and returns null on the whole query response, It would actually return null in the closest nullable parent field, which in practice usually means that the whole response becomes null.

In some cases the answer is “Yes!” and we make the field non-null, for instance on name or id values or if Pathfinder can return a sensible default value. Most of the time the answer is “No” and we make the field nullable.

Lists should mostly be non-null with non-null content, because returning an empty list can be the outcome of the non-null array field, but not of the non-null content. Example:

```
type AlbumGroupPage {
 # Non-null list with non-null content
 items: [AlbumGroup!]!
 # Required non-null for the AlbumGroupPage type to work
 pagingInfo: PagingInfo!
 # Non-null, since default value is the number items if real value is null
 totalCount: Int!
}
```

We have ADRs that explain more about the reasoning behind Nullability in unions and Nullability in other fields

Versioning and deprecation

We try to be backwards compatible whenever we can, but sometimes an existing field or type cannot be modified. You cannot just replace old fields with new types, since that will lead to breaking clients and angry client developers. Inventing new names for the same type of information will create definition confusion, so we suffix a “V” and version number to the new type or field instead. Examples:

```
@deprecated(reason: "Use PinnedItemV2 since it uses the wrapper to return error info")
union PinnedItem = Album | ... | UnknownType
union PinnedItemV2 = AlbumResponseWrapper | ... | UnknownType

@deprecated(reason: "Use ArtistPageV2 since it uses the wrapper to return error info")
type ArtistPage
@key(fields: "items{uri}")
{
 items: [Artist!]!
 ...
}

type ArtistPageV2
@key(fields: "items{_uri}")
{
 items: [ArtistResponseWrapper!]!
 ...
}

type ArtistPick {
 item: PinnedItem! @deprecated(reason: "Use itemV2 since it has better error handling.")
 itemV2: PinnedItemV2!
 ...
}
```

We have a few query methods that are named like `entityUnion(uri: ID!)` but these names are not the best[7], they should have been called `entityV2(uri: ID!)` instead.

Whenever a type or a field is replaced by a new version, the old field should have a deprecated annotation. The annotation must have a reference to the new type or field, for the client to easily update to newer structures. If you also add a date to the description when the field was deprecated, you get a gold star.
Sometimes fields actually change names because they also change semantics and then you would create a new field name. Example:

```
type TrackLyrics {
 kind: LyricsAnnotationKind! @deprecated(reason: "since 2021-03-02, use syncType instead")
 syncType: LyricsSyncType
}
```

Value types

Commonly defined value types are great since the client will have a consistent experience. The downside is that the value types must look exactly the same in every service which makes them hard to update. We have the following value types that can be used by any service:

- Date, to represent a date and time

[7] This naming happened early in the history of Pathfinder and made a lot of sense at the time

- Instant, to represent a number of milliseconds
- PagingInfo, to contain pagination information
- Copyright, to convey copyright information
- Duration, which is identical to Instant
- Language, which contains a language code
- Market, which contains a country code
- Playability, Information about a track's or an album's playability

We also have some common error types:

- NotFound, resource cannot be found
- RestrictedContent, resource is restricted for the end user
- GenericError, resource could not fetched or transformed for some reason
- UnknownType, the key (usually spotify uri) could not be resolved to a type that the service can handle

Adding new error types to existing unions should not be a breaking change, but remember that old deployed clients will not be able to get this new information and will most likely fallback to not showing anything.

Some value objects may belong to a domain. Contact the owner team directly or Bassline before you start using them in your service to see if the owning service is prepared to share it. Non-exclusive list includes:

- Price, generic way to handle prices used by audiobooks
- ContentRating, content rating used by episodes and chapters

Enums are inherently shared across services, but they are not always suitable for reusing between services. Treat enums as service specific and refrain from reusing them. Caution should be taken then sharing unions, as it will be much harder to change the union in the future if they are shared.

We have an ADR that explains more about the reasoning behind Shared types (aka value types).

Design for the client

One of the most important goals for Pathfinder is to make life better for our clients so that they can easily consume data. When we are creating or modifying schemas, we should try to adopt the client's point of view when it comes to naming.
Contact the potential clients during the design of the schema and ask them what terminology they use. Use that terminology, if possible, when naming types and fields. Adjust field data values to client expectations.

Do not expose the proto-file or db schema directly as types and fields, as this would probably expose too much internal structure. Another consequence is that refactoring in the upstream data will mean breaking changes for the client.

The advanced

These style guide points may be up for debate on how to design them in the best way. There may not be just one correct answer, but only the best answer according to the situation.

Using parent uri or using child uris

Scenario: you have two entities that have a parent - child relationship and these two are decorated in two different services (eg. podcasts and episodes). There are now two ways to model the decoration between those services:

Parent has a list of child wrappers

```
type Podcast
@key(fields: "uri")
{
 episodes(limit: Int = 5, offset: Int = 0): [EpisodeResponseWrapper!]!
 ...
 uri: ID!
}
```

The parent makes a query to resolve all the child uris and adds them to a list of wrappers. The child service does normal decoration of the wrappers.
This is a good way to design the schemas if the parent has direct access to the child uris when querying for the parent data.
A prerequisite is that the parent is responsible for any filtering of the children list, since the child service is unable to perform any actions on the list itself (it only sees the individual wrappers).

Child decorates parent with list of children

```
type Podcast
@extends
@key(fields: "uri")
{
 episodesV2(types: [EpisodeType] = [UNKNOWN], offset: Int = 0, limit: Int = 5):
ContextEpisodeResponse
 uri: ID! @external
}
```

The child service extends the parent type with a list and populates the child entities using the parent uri.
This is a good way to design the schema if the child service has access to a method where they can resolve child data given a parent uri. It is less good if the child first must resolve a parent

query and then resolve a secondary child data query, since that would put parent logic into the child service (less separation of concern). This design does give the child service opportunity to do more custom filtering of child entities, since it owns the list of children itself.

A third option is to keep the child and parent decoration in the same service (if they belong to the same domain).

Where to put parameters

Parameters for offset and limit should be placed as close as possible to the list that should be limited. However, sometimes the upstream puts limitations on where the parameters can be placed. For example the upstream of oxygen-search limits several lists at the same time, so the parameters are placed in a parent level:

```
type Query {
 # This parameters affect all the lists in SearchResultV2
 searchV2(query: String!, limit: Int = 5, offset: Int = 0): SearchResultV2
}

type SearchResultV2 {
 # You would expect that the parameters are placed on this level
 albums: AlbumPageV2 @provides(fields: "pagingInfo totalCount")
 artists: ArtistPageV2 @provides(fields: "pagingInfo totalCount")
 audiobooks: AudiobookPage @provides(fields: "pagingInfo totalCount")
 ...
}
```

Custom Error Types

The default error value types (GenericError, NotFound etc.) should be enough for most services, since the clients tend to only care about the occurrence of an error and not the specific details. If the clients have specific needs to process custom error types, then those types **must** `implements` the Error type and you are free to add any custom fields with the proper documentation.

Complexity annotations

TBD.
(It probably has something to do with increasing complexity, with a standard value, for each service (and each upstream client) that is needed to resolve the query and also some special handling for fields that we know are problematic. Perhaps add a complexity value to each data field in our wrappers.)

Common problems and how to solve them

Key is too complex to pass

We have had one case where the key itself was too complex to pass between federated service, and that was in oxygen-translations. The key had a multi-level tree structure and the gateway just silently removed some of the leaf levels when it passed the key.

Our solution for this problem was to put the whole key structure in a JSON string and pass that in a single key string field. The downside of this solution is that you create a tight coupling between services where serialization and deserialization of the JSON must always match, so you lose the GraphQL type safety.

Same Spotify URI Kind is reused for different types

We base most of our wrappers on spotify uris and sometimes the same spotify uri kind is used for two different entity types, such as Podcasts and Audiobooks. The producing service would just provide a spotify uri without specifying the concrete type and the decorating service would need to find out what the type actually is.

Our solution to this problem was to create a wrapper which can return both concrete types and let the decorating service make an extra call to find out what the spotify uri is and how it should be decorated:

```
type PodcastOrAudiobookResponseWrapper
@key(fields: "_uri")
{
 _uri: ID!
 data: PodcastResponseV2!
}

union PodcastResponseV2 = Audiobook | GenericError | NotFound | Podcast | RestrictedContent
```

The data response can be resolved to both an Audiobook and a Podcast. The reason for this design is that most producing services (pinned item, playlist, library etc) will not provide the concrete type so that logic must be in the decorating service.
The name of the wrapper is very ugly, and in hindsight `ShowResponseWrapper` would probably have been better. On the other hand `PodcastOrAudiobookResponseWrapper` clearly states the intent of the wrapper.

Using @provides in the schema

The “provides” directive gives a producing service[8] a chance to send additional data back to the gateway without resolving all fields with the decorating service. Our page types are entities, but our producing services want to populate more than just the key field. This example shows how the search producing service is using ArtistPage:

```
type SearchResultV2 {
 artists: ArtistPageV2 @provides(fields: "pagingInfo totalCount")
 ...
}


type ArtistPageV2
@extends
@key(fields: "items{_uri}")
{
 items: [ArtistResponseWrapper!]! @external
 pagingInfo: PagingInfo! @external
 totalCount: Int! @external
}
```

The “provides” directive indicates that the search service should populate the pagingInfo and totalCount without the need for the owning service (of `ArtistPageV2`) to populate those fields. In hindsight, it might have been better that the `ArtistPageV2` would have been a value type.

The “provides” directive could be used more extensively if the producing service already has some of the data that the decorating service should provide. The downside is that we would spread the decoration logic between two services.

Checklist for schema additions/changes

- ☐ All fields and parameters are in camelCase
- ☐ All enum values are in UPPERCASE_SNAKECASE
- ☐ SortGraphQL plugin is added or schema is manually sorted
- ☐ Any strings with special formatting is documented
- ☐ All key fields use type ID!
- ☐ All [lists] are paginated (excluding cases outlined in When not to use pagination)
- ☐ All types, that have some sort of identifier, use the key directive
- ☐ All new types in the schema have a feature (or domain) prefix
- ☐ Almost all types have ”descriptions”
- ☐ Most important fields have ”descriptions”
- ☐ Clients have approved the design of the schema

[8] In reality, it is any service which extends a entity for another service

- [ ] Mutations are named verb + entity + (To/In/For/At etc.) + domain
- [ ] Mutations use dedicated input type and return full domain data in response
- [ ] Wrappers are used for type decorations.
- [ ] Wrappers have non-null fields and the data field references a Response union
- [ ] Most fields in your types are nullable and only critical fields are non-null
- [ ] Fields replacing old fields use V2 suffix. Same for replacing old types.
- [ ] Old fields use deprecated annotation and clearly states the name of the new field
- [ ] No new common value types have been introduced
- [ ] Parameters for list limitation is placed close to list itself
- [ ] No new custom errors have been introduced
- [ ] No other special design has been introduced

Note: try to re-use as much as possible from what's already in the schema, for example: use the federated entity instead of exposing the fields directly