BookBrainz Developer Docs

Release 0.1

Ben Ockmore

Aug 05, 2022

Contents

This documentation is intended to act as a guide for developers working on or with the BookBrainz project, describing the system, its modules and functions.

For a description of BookBrainz and end-user oriented documentation, please see the BookBrainz User Guide.

CHAPTER 1

Contents of this documentation

1.1 Installation

So you've decided to contribute some code to BookBrainz? Fantastic! Thank you!

Here are some instruction to get your local coding environment set up.

1.1.1 Setting Up

BookBrainz depends on having PostgreSQL, Redis, Elasticsearch and NodeJS set up and running.

But first some basic configuration common to both docker and manual installation.

Cloning

To clone the repository and point the local HEAD to the latest commit in the stable branch, something like the following command should work

```
git clone --recurse-submodules https://github.com/metabrainz/bookbrainz-site.git
```

Since this project makes use of git submodules, you need to use `git clone --recurse-submodules` to clone it.

Alternatively, to manually initialize submodules, run these two commands

```
git submodule init
git submodule update
```

Currently we are using these submodules

- MonkeyDo/Lobes in `src/client/stylesheets/lobes`

Configuration

Create a copy of `config/config.json.example` and rename it to `config.json`. You can do this by running this command from the bookbrainz-site directory.

```
cp config/config.json.example config/config.json
```

If you want to be able to sign-up and edit, you will need to set up authentication under `musicbrainz`. To get the `clientID` and `clientSecret` tokens, head to MusicBrainz and register a new developer application.

Make sure to enter the callback URL as `http://localhost:<port>/cb` (port: 9099 by default).

If you want to review the entities, you will need to set up authentication under `critiquebrainz`. To get the `clientID` and `clientSecret` tokens, head to CritiqueBrainz and register a new developer application.

Make sure to enter the homepage URL as `http://localhost:<port>/` and callback URL as `http://localhost:<port>/external-service/critiquebrainz/callback` (port: 9099 by default).

You can then copy the tokens for that developer application and paste as strings in your `config/config.json`. The tokens and callback URL in your `config/config.json` needs to match exactly the one for the developer application.

1.1.2 Docker Installation

The easiest way to get a local developement server running is using Docker. This will save you a fair amount of set up.

You'll need to install Docker and Docker-compose on your development machine:

- Docker
- Docker-compose

Important: We recommended Windows users to docker-setup (preferbly WSL2) to avoid any compatibility issues.

When it is installed, follow the below instructions step by step.

Note: If you are using docker-toolbox you need to replace elasticsearch:9200 with ip address of your docker-machine in config.json. To get ip address of your docker machine use command `docker-machine ip default`

Database set-up

When you first start working with BookBrainz, you will need to perform some initialization for PostgreSQL and import the latest BookBrainz database dump.

Luckily, we have a script that does just that: from the command line, in the `bookbrainz-site` folder, type and run `./scripts/database-init-docker.sh`. The process may take a while as Docker downloads and builds the images. Let that run until the command returns.

Note: The latest database dump can be found here

Running Web server

If all went well, you will only need to run `./develop.sh` in the command line from the `bookbrainz-site` folder. Press `Ctrl+c` to stop the server.

Note: The dependencies (postgres, redis,...) will continue to run in the background. To stop them, run the command `./stop.sh`

Wait until the console output gets quiet and this line appears:

```
> cross-env node ./lib/server/app.js.
```

After a few seconds, you can then point your browser to localhost:9099.

Make changes to the code in the `src` folder and run `./develop.sh` again to rebuild and run the server.

Once you are done developing, you can stop the dependencies running inside docker in the background by running `./stop.sh`.

1.1.3 Search server setup

In order for searching to work on your local server, you will need to index the contents of the database.

1. first, ensure that Elasticsearch is running.
2. add your user name (if you haven't created a user yet, *now is the time!* <*https://musicbrainz.org/doc/How_to_Create_an_Account*>) to the array of `trustedUsers` in the `src/server/routes/search.js` file
3. **with that done and the server (re)started, navigate to `localhost:9099/search/reindex`**
   Reindexing will take a few minutes depending on your resources, and you can expect that the browser window will time out before the reindexing is done. However the process will continue in the background and after a little while the search indices will be created.
4. You can now try searching for an entity on the page `localhost:9099/search`

1.1.4 Advance Users

To improve your developer experience, here are some things we suggest you should do

Live Reload

You may want to use Webpack to build, watch files and inject rebuilt pages without having to refresh the page, keeping the application state intact, for the price of increased compilation time and resource usage (see note below).

If you are running the server manually, you can simply run `yarn run debug` in the command line.

If you're using Docker and our `./develop.sh` script, you will need to modify the `docker-compose.yml` file and change a few things on the `bookbrainz-site` service defined there

1. Change the bookbrainz-site command to

- `yarn run debug` if you only want to change client files (in `src/client`)
- `yarn run debug-watch-server` if you also want to modify server files (in `src/server`)

2. Mount the `src` folder to the bookbrainz-site service

For example:

```
services:
  bookbrainz-site:
  # 1. Change the command to run
    command: yarn run debug
    volumes:
      - "./config/config.json:/home/bookbrainz/bookbrainz-site/config/config.json:ro"
  # 2. Mount the src directory
      - "./src:/home/bookbrainz/bookbrainz-site/src"
```

Warning: Using Webpack watch mode (`yarn run debug`) results in more resource consumption (about ~1GB increased RAM usage) compared to running the standard web server.

Debugging with VSCode

You can use VSCode to run the server or API and take advantage of its debugger, an invaluable tool I highly recommend you learn to use.

This will allow you to put breakpoints to stop and inspect the code and variables during its execution, advance code execution line by line and step into function calls, instead of putting console.log calls everywhere.

Here is a good introduction to debugging javascript in VSCode.

There are VSCode configuration files (in the `.vscode` folder) for running both the server and the tests, useful in both cases to debug into the code and see what is happening as the code executes. Make sure the dependencies (postgres, redis, elasticsearch) are running, and you can just open the debugger tray in VSCode, select 'Launch Program' and click the button!

1.1.5 Testing

The test suite is built using Mocha and Chai. Before running the tests, you will need to set up a `bookbrainz_test` database in postgres. Here are the instructions to do so:

Run the following command to create and set up the `bookbrainz_test` database using Docker

```
docker-compose run --rm bookbrainz-site scripts/wait-for-postgres.sh scripts/create-
↪test-db.sh.
```

If you are running postgres manually outside of Docker, you can set some environment variables before running the script *scripts/create-test-db.sh* In particular `POSTGRES_HOST=localhost` but you can also set `POSTGRES_USER`, `POSTGRES_PASSWORD` and `POSTGRES_DB`.

Once your testing database is set up, you can run the test suite using

- To run in Docker

```
docker-compose run --rm bookbrainz-site yarn run test
```

- To run locally

```
yarn run test
```

Note: You may need to adjust your `config/test.json` file to match your setup.

See also:

if you face any issues, please refer to our *Troubleshooting* section.

1.2 Coming from MusicBrainz

This page describes the key differences between the BookBrainz and MusicBrainz schemas, for developers already familiar with the MusicBrainz schema.

1.2.1 Entities

There are 6 entity types in the BookBrainz world: Author, Work, Edition Group, Edition, Publisher and Series. For a better description of each entity see *Entities*.

1.2.2 Versioning

Entities in BookBrainz are versioned, which means that the entire editing history is stored in the database. To make this possible, we have a number of additional tables in the database. See a more complete description in *BookBrainz Schema*

In short, each entity type has its own `_header`, `_revision` and `_data` tables (i.e `author_header`, `author_revision`, `author_data`). The `_header` points to the latest `_revision``(each modification of one or more entities creates a new revision), and the _revision points to the ``_data` for that revision. An additional revision_parent table allows us to keep a tree of modifications.

In additions to the versioning system, "deleting" an entity in BookBrainz is only ever a "soft delete", meaning we mark the entity as deleted but keep all their editing history. This allows us to resurect 'deleted' entities if need be.

1.2.3 Aliases

Each entity has an Alias Set containing one or more aliases. An entity's 'name' as well as other names (authors' aliases, names in other languages and scripts, etc.) are all represented with the same `alias` table. For convenience and ease of access, an entity will store its "default" alias ID to avoid having to fetch the Alias Set.

1.2.4 Sets

Inside that data, some elements like aliases (names), relationships and identifiers come in `sets`. A set contains one or more element; any change to the elements contained in the set (modification, addition or deletion) creates a new set. That new set contains the unchanged elements as well as new elements for any modified item.

For example, let's imagine an Author has an Alias Set with ID 1 containing one alias with ID 123. If we modify this alias, a new Alias Set with ID 2 will be created, containing a new alias with ID 124. The new revision for this author will point to a new author_data row with AliasSet ID 2.

If we want to revert this change, we can simply create a new revision that either points to the previous author_data ID, or a new author_data row with AliasSet ID 1.

1.2.5 Merging

Merging in BookBrainz is very similar to regular edits: we create a new revision for each entity, one of which will contain the new aggregated data from the merge. The new revision will be marked as being a merge operation, as well as each author_revision (using Authors as an ecample) for each of the merge entity.

Let's say we're merging Author B and Author C into Author A. We select the correct value if there are any differences between entities, and the sets are combined (we add together the aliases, relationships, and identifiers). We create a new revision and three new author_revision, all marked as a merge operation.

Author A's author_revision entry will point to a new author_data ID with the combined data described above. Author B and C's revisions will have their author_data ID set to NULL, the equivalent of a soft delete. With the help of the revision_parent table, we can infer that Authors B and C previously existed but were merged into Author A. The versioning system also gives us all the data we would need to revert this merge operation.

1.3 BookBrainz Schema

1.3.1 Introduction

The BookBrainz schema describes how the data used by BookBrainz is stored. It's quite important to have a good idea of this before you look at any of our code. If you're coming from a MusicBrainz background, our schema is similar, but there are some key differences - see *Coming from MusicBrainz*.

1.3.2 Overview

In BookBrainz, an entity is a container for some data, associated with some globally unique identifiers (GIDs). BookBrainz represents a number of different objects as entities, and each of these has its own particular associated data. In the following section, we'll go through the database structures used to represent entities, and talk about each type of entity in more detail.

Entities

Fig. 1: There are 6 entity types in the BookBrainz world: Author, Work, Edition Group, Edition, Publisher and Series.

- **Author**

  An individual, group or collective that participates in the creative process of an artistic work.
  It also includes translators, illustrators, editors, etc.

- **Work**

  A distinct intellectual or artistic creation expressed in words and/or images.
  Here we are not talking, for example, about a physical book, but the introduction, story, illustrations, etc. it contains.
  Examples: novel, poem, translation, introduction & foreword, article, research paper, etc.

- **Edition** A published physical or digital version of one or more Works. Examples: book, anthology, comic book, magazine, leaflet .. note:: An Author can self-publish an Edition

- **Series**

  A set or sequence of related works, editions, authors, publishers or edition-groups.

Examples: a series of novels, a series of comics, etc.

- **Edition Group** A logical grouping of different Editions of the same book. Example: paperback, hardcover and e-book editions of a novel
- **Publisher** A publishing company or imprint

Entity and Entity Redirect

All entity GIDs are stored in a single table in the database, **entity**.

A second table, **entity_redirect**, allows redirection of GIDs. For example, if one entity was merged into a second entity, then a row would be created in the **entity_redirect** table to indicate a mapping to the merge target.

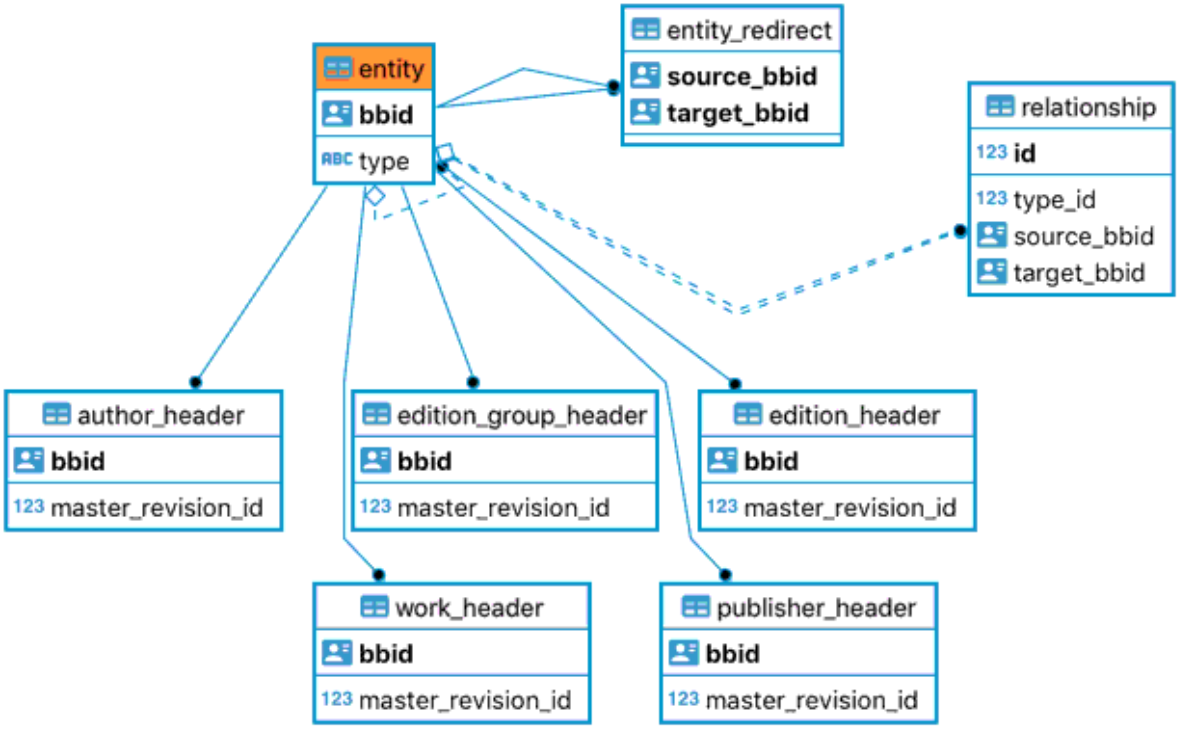


Fig. 2: The entity table and its relationships

Versioning

Entities in BookBrainz are versioned, which means that the entire entity history is stored in the database. To make this possible, we have a number of additional tables in the database. For an explanation, let's take the example of an Author entity.

The `entity` table, common to all entity types, describes the BBID (unique identifier, will never change) and type of an entity. Each entity type has its own `_header` and `_revision` tables, in this case `author_header` and `author_revision`. The `author_header` table tells us what is the master –or latest– revision of that Author entity

The `author_revision` entry points to the `author_data` table representing the state of that entity at that revision. Each time a user modifies one or more entities, a new revision is created for each entity modified. That includes adding or modifying a relationship between two entities.

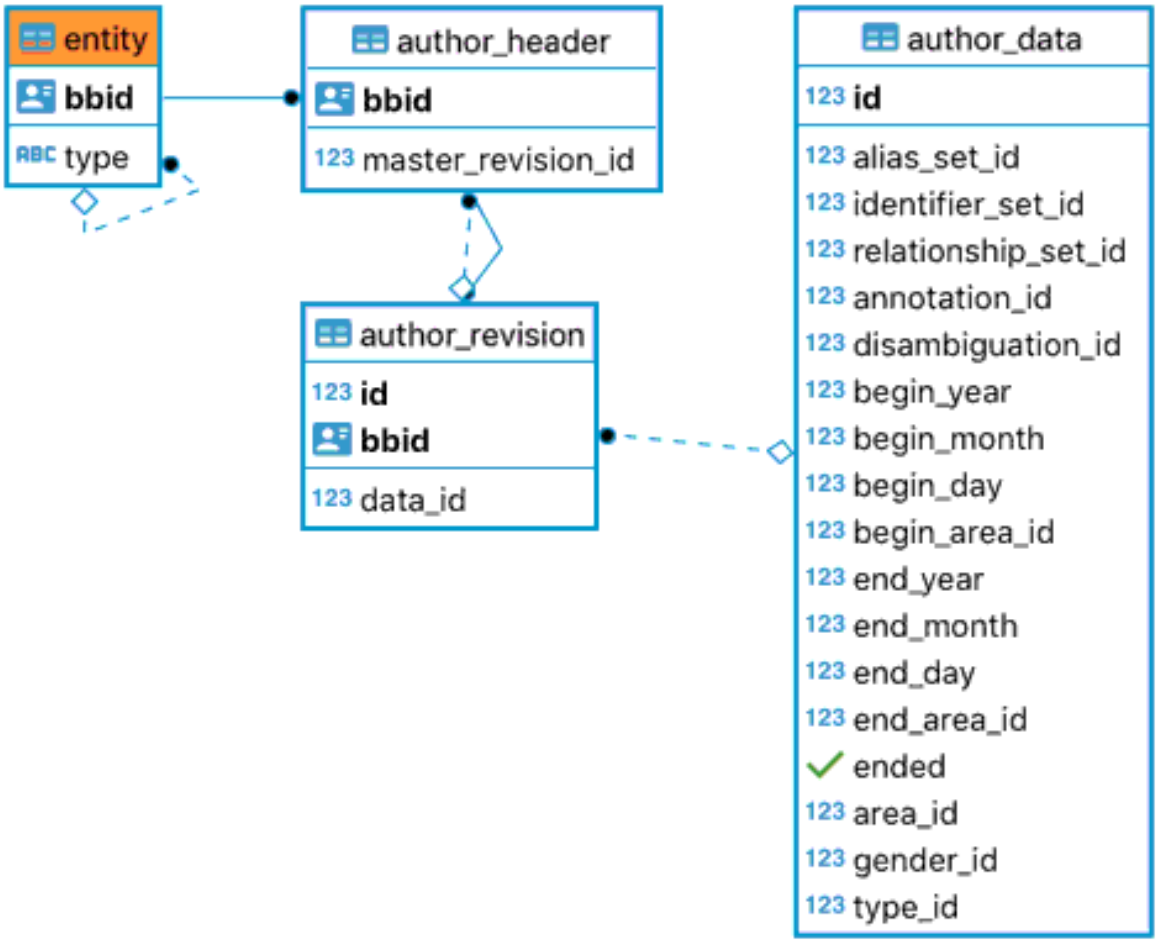


Fig. 3: The main tables for Author entities

1.3.3 Revisions

A revision represents a set of changes to one or more entities.

If you change an Author's date of birth, your revision will have the same ID as one single author_revision, which represents the state of that entity at that time.

If for example you create a relationship between an Author and a Work ("Author X wrote Work Y"), you will have a single revision but you will find an `author_revision` AND a `work_revision` with the same ID as the revision. This tells us that unit of change concerns the two entities, and each `*$entity*_revision` represents that entity's state after that change.

Following me so far?

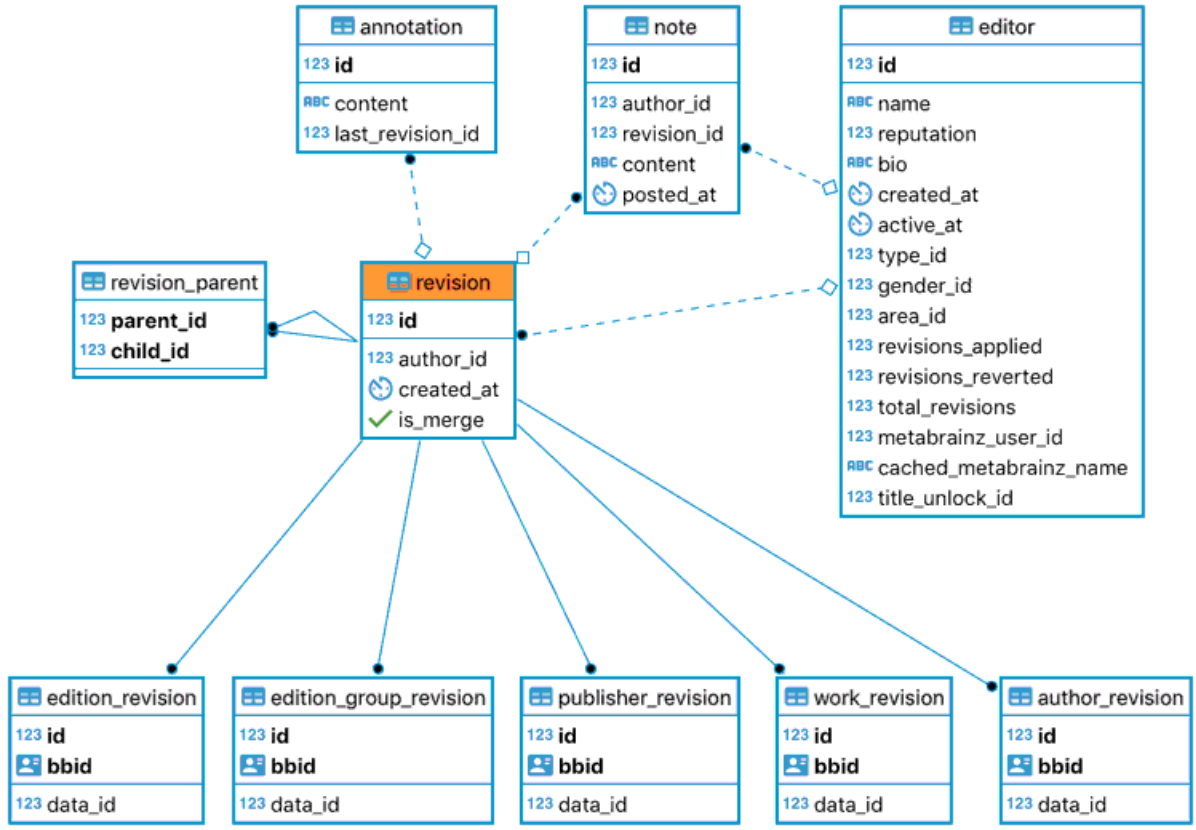


Fig. 4: The revision table and associated tables

Entity revision

As we saw above, a change to an Author will create a new author_revision which contains a data_id column. This ID links to an entry in the author_data table that describes the state of that entity.

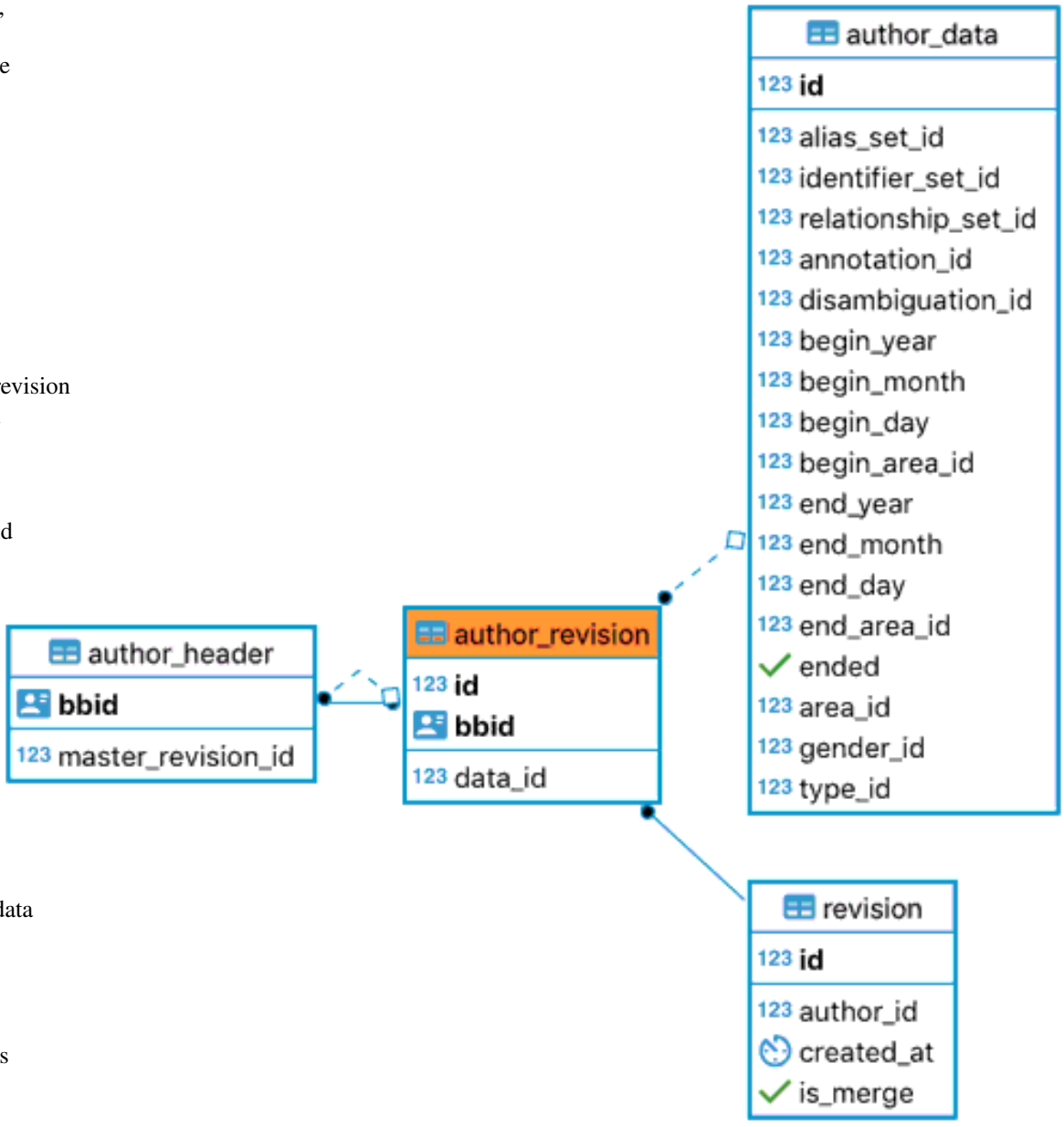


Fig. 5: An author_revision in action

Revision parent

The `revision_parent` table allows us to reconstruct the history of all the edits that have changed an entity, inspect it at any given time, and even revert past edits. When a new revision is created, it stores a reference to the previous

revision. For merge revisions, it will store a reference to the revisions of each entity being merged.

1.3.4 Entity data

Some of the information is stored directly in that author_data table, and some other information are stored separately.

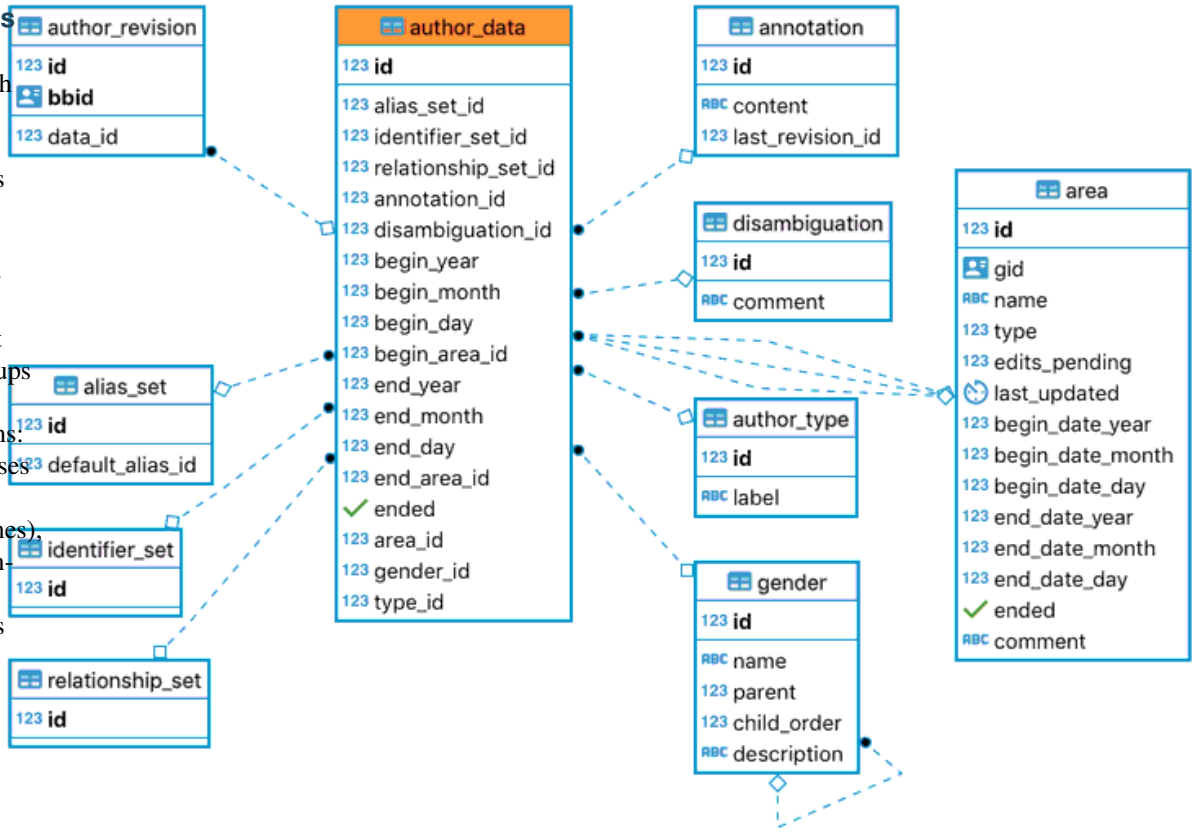


Fig. 6: Overview of the author_data associated tables

Sets

Each entity uses *sets* to represent groups of items: aliases (or names), identifiers (ID of that entity in other systems like wikidata), relationships, languages, etc. Sets allow us to modify some data while keeping the rest untouched, which is necessary for our versioning system. For example deleting an item from a set will create a new set but the removed element still exists and is still part of the previous set.

Each set type is comprised of three tables:

1. XXXXX: the table of elements of type XXXXX (for example `alias`)
2. XXXXX_set: the table of sets of type XXXXX (for example `alias_set`)
3. XXXXX_sets_XXXXX: the table that links elements to a specific set (for example `alias_set__alias`)

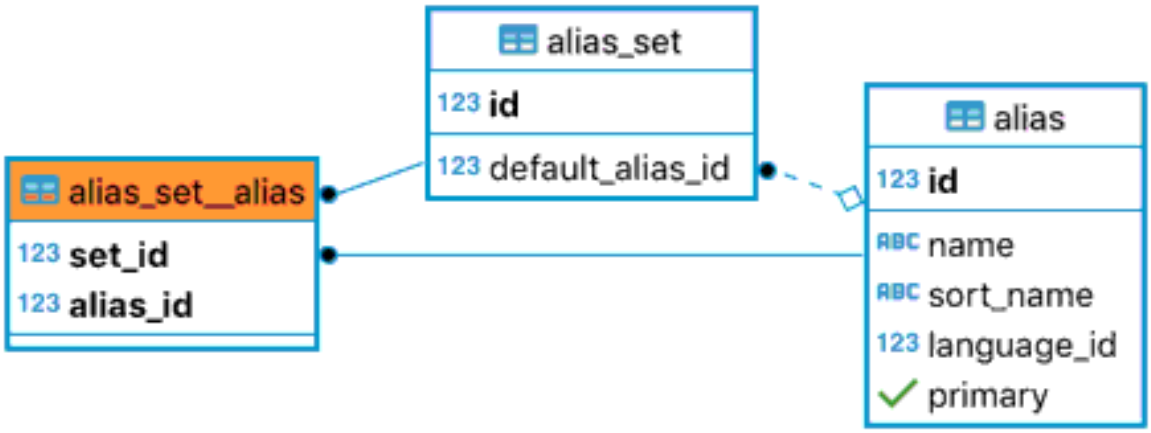


Fig. 7: An alias and the set it belongs to

Alias sets

Aliases are stored each separately, and combined to form an alias_set. This represents the various names of an entity. For example, an Author could have their birth name, a pen name, their name in various other languages, etc. There is a default_alias_id stored for each alias_set that points to one alias, as a shortcut if you only needs the main name of an entity.

When an alias is added to an entity, a new alias_set is created that will contain the previous unchanged aliases as well as the new alias.

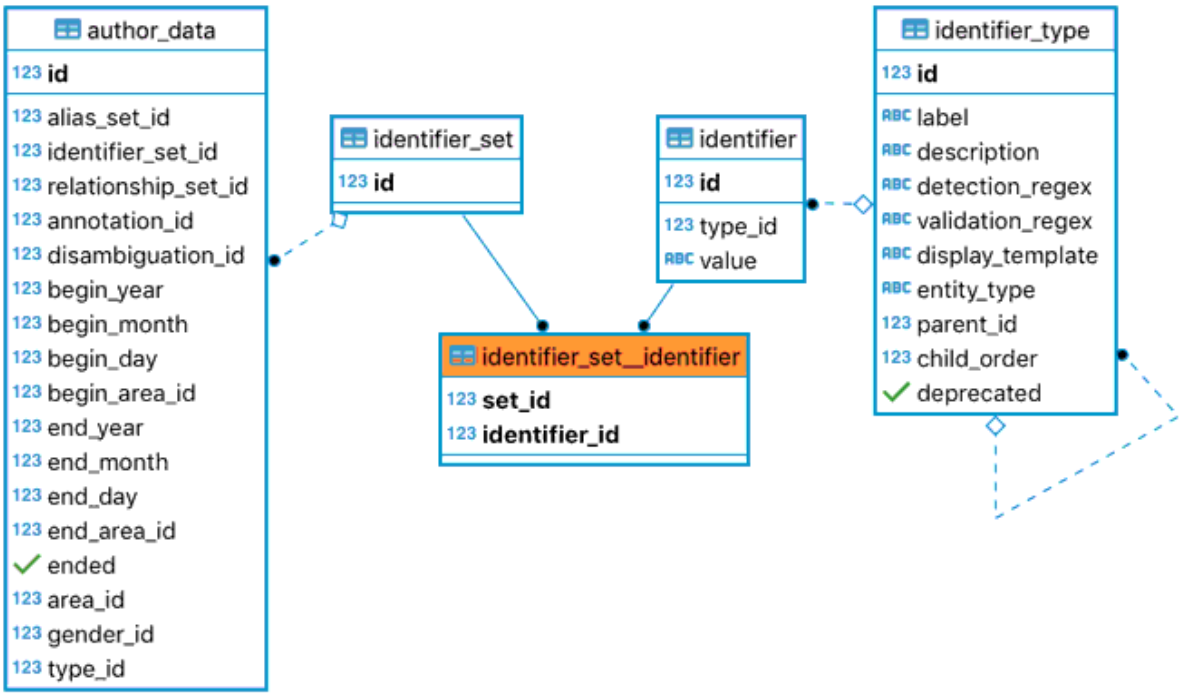
If an existing alias is modified, a new alias entry will be created as well as a new set containing that new alias entry. That means if we want to revert the change, the previous revision contains a reference to the previous alias_set that contains the original alias.

You will find the same structure for identifier sets and relationship sets.

Identifier sets

Identifiers represent the ID of the entity in another system (wikidata, musicbrainz, openlibrary, etc.). An entity can have an identifier set, represented by an id linking to row in the identifier_set__identifier table, which links a set to the identifiers that comprise it. That way, when adding a new identifier, a new set is created but the existing identifiers are not modified.

The set is comprised of identifiers each of a type_id that refers to the identifier_type table. The entity_type must correspond to the Entity's type. The other columns of identifier_type are used for detecting and displaying purposes on the front-end and API.



Relationship sets

Relationships are of a specific type (a relationship_type referred to by id) that describes the relationship, the entity type expected on either side of the relationship, and the phrases to use to represent the relationship from either direction (i.e: "Author X wrote Work Y" and "Work Y was written by AuthorX").

Publisher sets and release event sets

These sets are used solely for the Edition entities The publisher sets don't have an associated `publisher` table like other sets. Instead, the `publisher_set__publisher` table links a publisher_set to the BBID of a Publisher entity

Additional Tables

There are some additional tables related to all types of entity. We've already mentioned annotations and disambiguations, so let's talk a little more about those.

An **annotation** is a way of making notes about an entity, for other editors to read. It stores some content associated with an ID. Disambiguation comments, stored in the **disambiguation** table, have a similar data structure but are intended to contain a short description to allow editors to easily differentiate between similarly-named entities.

An **alias** represents a name or title. Each alias will store some text along with a language, and a couple of flags to indicate whether the alias is *primary* and whether it is *native*. An entity can only have one *native* alias, which indicates its original name. It can have many *primary* aliases, which give the most common names in particular languages. *Native* aliases will usually also be *primary*.

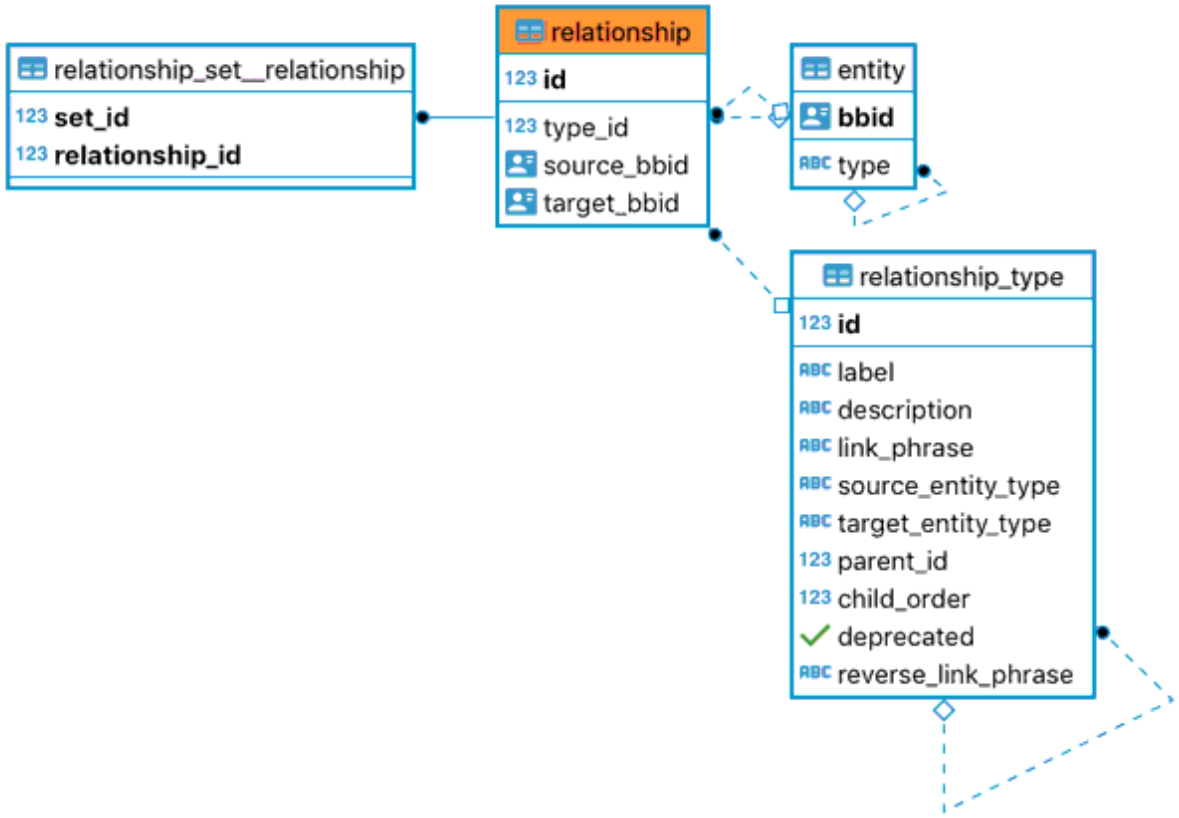


Fig. 8: A relationship set contains relationship entries

A **disambiguation** allows us to differenciate between two entities with the same name. All entities can have a disambiguation (although they are not required), referred to by a disambiguation_id, that points to the disambiguation table.

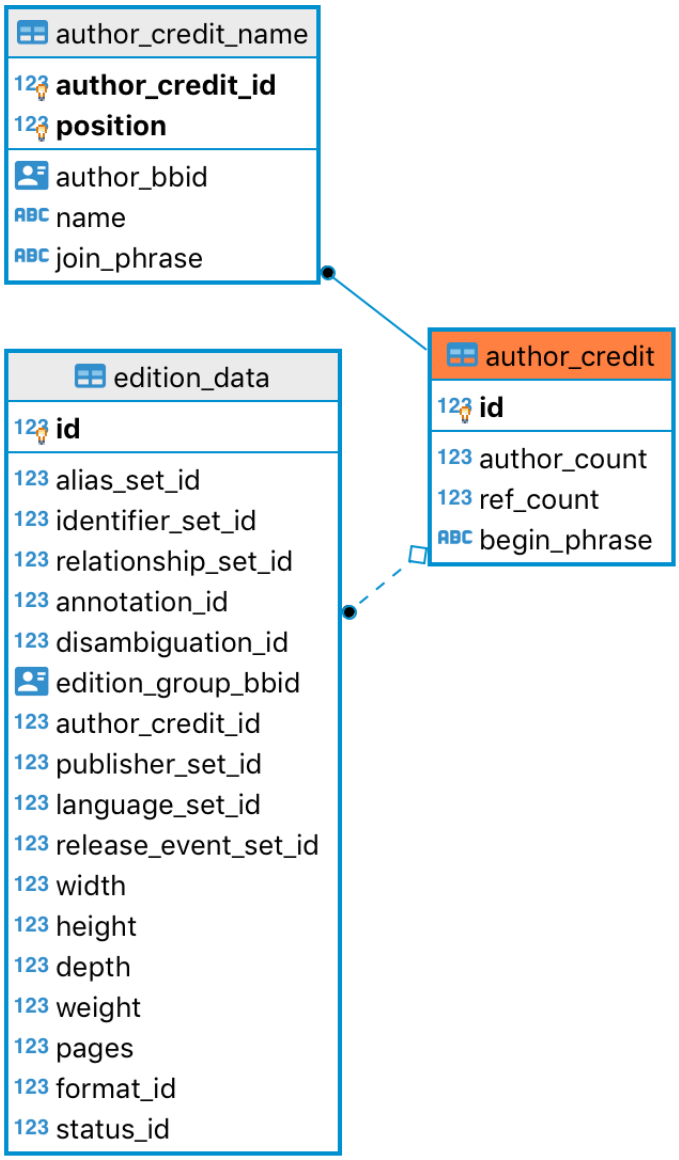
1.3.5 Author credits

Author credits allows us to define how authors are credited in an Edition (as on the book cover), without having to create separate Author entities for each pen name or name variation.

For example, Howard Phillips Lovecraft published under the names "H.P. Lovecraft" most notoriously, but you will find some Editions use the full name "Howard Phillips Lovecraft"

The author credits are composed of one or more authors. For each author, an author_credit_name is created with the author's BBID, name as credited and position, as well as a short phrase to join them with the next author.

For example if a book cover features "by John and Jane Doe", you would enter: Position 1: John Doe's BBID - name: "by John" - join phrase "and " Position 2: Jane Doe's BBID - name: "Jane Doe"



1.4 BookBrainz ORM

1.4.1 Overview

BookBrainz uses an ORM tool (Object-Relational Mapping) to interface with the database. What that means in practice is the we we import a separate Javascript package (hosted in this repository) in the webserver and use its methods to work with objects that represent the data rather than directly with raw database rows.

That allows us to load an entity with associated data (for example an Author entity and its aliases), and the ORM will compose the sequence of SQL queries to fetch the relevant data. The ORM will return an Author Javascript object with an array under `myAuthorObject.aliasSet.aliases`

The ORM we use is called Bookshelf.js (no relation to BookBrainz, the name similarity is just a coincidence!). If you ever get lost in the syntax of the ORM, you will find the documentation here. With it we define 6 main entity models (one per entity, see *Entities*) as well as a model for each of the schema concepts we need to work with (revisions, sets, relationships, author credits, etc.). In addition we export some utilities to deal with common operations to work with the models.

The ORM is set up, connected to the database, and then stored in the webserver application session at startup That way it is accessible at any time in the ExpressJS server requests (i.e. in each server route). In a standard ExpressJS route `router.get('/:id', (req, res, next) => {...` you will find it in `req.app.locals.orm`.

Whenever you see in our codebase an `orm` passed as an argument to a function, you can expect to find the whole BookshelfJS ORM as exported by the package. The list of models and components exported by the package can be found here.

1.4.2 Entities and their revisions

Following our schema, there are important concepts to understand when working with the ORM models. We have a generic Entity model which represents an entity of unknown type. It is described by a BBID (unique identifier) and the ID of the latest revision that describes its current info.

From that entity header we can get an entity type, and using the getEntityModelByType utility we can retrieve the corresponding ORM model.

There is also a generic Revision model as well as a more specific revision model for each entity type (i.e. AuthorRevision, WorkRevision, etc.) Neither of these directly contain the data associated with that revision, but they do provide a way to get from one to the other. The Revision holds information about who created the revision, notes associated with it, as well as the parent and children revisions that might come before and after it.

The AuthorRevision can be fetched along with the associated `data` and allows us to compare different AuthorRevisions

1.4.3 Entities and their data

When we merge entities, their BBIDs get redirected to the entity we merged into. To ensure we are targetting the correct entity, we can use the recursivelyGetRedirectBBID utility: `const redirectBbid = await orm.func.entity.recursivelyGetRedirectBBID(orm, relEntity.bbid, null);`

To get an entity's current data (i.e. at the last revision) using its BBID, we can forge a new instance of the entity model (let's take the Author model for example): `const myAuthor = await model.forge({bbid: redirectBbid}).fetch({withRelated: ['defaultAlias', 'disambiguation']});` The myAuthor object will have the content of the AuthorData model as well as the defaultAlias and disambiguation that we required.

The `orm.func.getEntity` method does just that for you!

1.4.4 Transactions

Sometimes, we want to apply a sequence of operations, and only apply them if all of the operations succeed. For example, when deleting entity A which has links to entity B, we want to modify the relationships of entity B to remove the A<->B relationship from its RelationshipSet. If deleting entity A fails for some reason, we want to abort all the operations. This concept of Atomicity in SQL uses transactions to commit or rollback operations. In the ORM, we can create a `transaction` object at pass it around when calling methods on the model to ensure atomicity. This is done using the `orm.bookshelf.transaction` method

1.5 BookBrainz Webservice/API

The BookBrainz API (currently in beta) provides developers with a way to make programs which use BookBrainz data. Documentation for the API is provided alongside the API itself: https://api.test.bookbrainz.org/1/docs/

1.6 The Entity Editor

1.6.1 Introduction

In Bookbrainz, a set of information about a particular author, book, publication, magazine, etc. is referred to as an *Entity*, and the form which we use to add/edit this information is called the **Entity-Editor**. There are 6 entity types in the BookBrainz world: Author, Work, Edition Group, Edition, Publisher and Series. For a better description of each entity, see *Entities*.

The code for the entity editor can be found in src/client/entity-editor. The entity-editor has been divided into different sections, and each of these sections have a folder within this directory.

We use Redux for state management. Each section has its own set of actions and reducers to handle state corresponding to the component. For more information on how Redux works, you can refer to this: Getting started with React-Redux.

Each section has its own directory containing the React components for editing and merging entities, along with the aforementioned Redux actions and reducers. You will also find a `common` folder with shared code and a `validators` folder containing the code used to validate the data entered in the form before submission.

1.6.2 Different sections of the Entity Editor

The Entity-Editor can be broken down into distinct mini-sections which are as follows:

Name Section This section is used to give a default alias to the entity. It also has fields which define the sort name, the language used to define the name of the entity, and an optional disambiguation field.

Alias Section This section helps us to define alternate aliases for the entity, such as an alternate spelling, or a different stylistic representation, etc.

Identifier Section This section allows us to add any identifier of the entity in some other databases and services such as an ISBN, MusicBrainz ID, etc.

Entity Section This deals with adding some entity specific information, such as a work might have a `workType`, an edition might have an Edition-Group, etc.

Relationship Section This section makes use of a modal called the relationship-editor, which allows us to make a logical connection between two entities.

Annotation Section This section contains a field which allows us to add additional freeform data that does not fit in any other section above.

Submission Section This section contains a field where the user can enter any edit notes. This also contains the Submit button which finalises our submission.

1.6.3 Adding an Entity

When we click on an Add Entity button, it takes us to a page with an url like this: `https://bookbrainz.org/{entityType}/create`. We make use of this `entityType` to generate the entity-specific markup for our entity-editor. The */create* routes are created separately for each entity type, each having a file in the src/server/routes/entity folder.

While creating the entity specific markup for our entity-editor, we also inject certain props into our entity-editor. We make use of router-level middleware to load `languageTypes`, `identifierTypes`, and `relationshipTypes` specific to the entity we are going to add. We might also need to load some entity-specific props, for example, in case of Work entity, we load `workTypes` too.

While creating the markup, it also creates a Redux store for our entity-editor, with a *rootReducer* which consists of a combination of reducers, each concerned with a particular section of the entity-editor namely:

- aliasEditorReducer
- annotationSectionReducer
- buttonBarReducer
- entityReducer
- identifierEditorReducer
- nameSectionReducer
- relationshipSectionReducer
- submissionSectionReducer

Let us take an example how we make use of actions and reducers in our entity-editor by describing the workflow for the **Name-Section** of the entity editor.

You can take a look at the initial state of a particular section by looking at the reducer in its directory.

The initial state for Name-section can be seen by looking at the reducer file in src/client/entity-editor/name-section/reducer.js. The initial state looks like this:

```
state = Immutable.Map({
            disambiguation: '',
            exactMatches: null,
            language: null,
            name: '',
            searchResults: null,
            sortName: ''
        })
```

Here, the fields `disambiguation`, `name`, `sortName` and `language` are self-explanatory. Whenever we start entering any name in the Name field, we use the onChange event handler to fire off 4 different actions:

- `onNameChange`: this updates the *nameValue*.
- `onNameChangeSearchName`: as we enter the name, we try to seach the nameValue in our database in order to let the user see whether the entity already exists in our database. This updates the search term and adds the results in *searchResults*.
- `onNameChangeCheckIfExists`: if we find an entity whose name matches exactly with the entity name we entered, we try to display a warning so as to avoid the user from making a duplicate entry. This updates the content of *exactMatches*.
- `searchForMatchindEditionGroups`: Search for Edition Groups that match the name, if the entity is an Edition.

Similarly, appropriate eventHandlers and actions are present for updating the value of *Sort Name* field, *Language*, and *Disambiguation*.

Hence, a similar pattern is followed on all the other sections of the entity-editor, where we make use of onChange event handlers for a particular field, to fire off an action with the help of the dispatch function to update its corresponding field in the state.

The Relationship Editor

Any type of connection between two entities in BookBrainz is called a relationship. We use the relationship editor component to add relationships to the entity we are creating.

The relationship section is concerned with two main tasks:

- Providing an **Add Relationship** button to open a Modal which acts as our relationship-editor. The relevant code for this is present in src/client/entity-editor/relationship-editor/relationship-editor.tsx.
- Rendering a list of already added relationships. This is done with the help of *RelationshipList* component present in src/client/entity-editor/relationship-editor/relationship-section.tsx.

We make use of `relationshipEditorVisible` flag to toggle the Relationship editor modal. Within the relationship editor modal, there are two fields:

Entity Select field : The *renderEntitySelect* function deals with this field. Here `baseEntity` is the entity which is being edited. The `EntitySearchFieldOption` allows us to search for any existing entity which we would like to link to our current *baseEntity*.

We can apply some additional filters to our search, so as to the optimize search results. For example, in case of a Series entity of type X, we dont need to display search results with entities which are not of the same type. When we select an entity from the Search results, it gets stored as `targetEntity`.

RelationshipType Select field : After selecting a targetEntity, we make use of a function called `generateRelationshipSelection` which takes our relationshipTypes object which was passed as a prop to our entity-editor, the baseEntity, and the targetEntity. This function returns all combinations of relationship types which are valid between the two entities. We can then select the Relationship Type for our entity using the RelationshipSelect field in the editor. This sets the value of `relationship` and `relationsipType` of our state.

When we click on Add, we pass the `relationship` object to the following action:

```
let nextRowID = 0;
export function addRelationship(data: Relationship): Action {
    return {
        payload: {data, rowID: `n${nextRowID++}`},
        type: ADD_RELATIONSHIP
    };
}
```

Here the data is the `relationship` object we passed from our Relationship Editor props. In the payload for this action, we also pass a `rowID`, which is the index of the relationship in the array of relationships for that entity. This is then added to the `relationships` object, with the `rowId` acting as a key for the mapping.

As we keep on adding relationships, they are rendered as a list on the entity-editor with the help of the aforementioned RelationshipList component. We can still edit and remove these relationships from the list. If we click the edit button next to a particular relationship, it opens up the relationship modal with the `relationship` object passed as prop to the relationshipEditor.

Submission Section

When we click on submit, the entire state of the form(rootState) is sent to an appropriate *createHandler* (`{entityType}/create/handler`). This route makes use of a utility function makeEntityCreateOrEditHandler which returns a handler function `handleCreateOrEditEntity` adapted to the entity type.

The *rootState* is then validated using the validators for that entity type. After this, the *rootState* is manipulated to be in a certain format by the `transformNewForm` function in each entity's route definitions file (`src/server/routes/entity/{entityType}`).

This formatted form is then passed to the aforementioned `handleCreateOrEditEntity` function which takes care of creating and modifying the required ORM models [*BookBrainz ORM*] (including other entities i.e. for relationships) and persisting the changes to the database.

If all the above steps are successful, the user is redirected to the Display Page of the newly created entity.

1.7 Troubleshooting

1.7.1 General

- `[Error: EACCES: permission denied, scandir '/root/.npm/_logs']`

  There is some incompatibility with Docker for Windows The solution is to **enable WSL2** (Windows Subsystem for Linux v2) and the problem goes away.

- `Can't open input file latest.sql.bz2: No such file or directory`

  After downloading the data dumps, you may realize that an attempt to uncompress it using the command `bzip2 -d latest.sql.bz2` doesn't work and gives the above error.

  It can be solved by giving the *absolute* path of the latest.sql.bz2 file in place of the file name.

```
bzip2 -d path/to/file
```

- `fatal: unable to access 'https://github.com/path/to/repo.git/': gnutls_handshake() failed: Error in the pull function`

  Check your internet connection and make sure you are not working behind any proxy.

- No Css Styles?

  Try running `yarn run build-less` or `npm run build-less` from the project directory.

1.7.2 Elasticsearch

- Elasticsearch taking too much time?

  let ElasticSearch to run on its own terminal, and proceed the building process by making another window of terminal.

- `Waiting for elasticsearch:9200 .elasticsearch: forward host lookup failed: Unknown host`

  The cause could be the docker-machine's memory limits. you can inspect this with the command:

```
docker-machine inspect machine-name
```

  To diagnose this problem, try taking a look at the logs with the command:

```
docker-compose logs elasticsearch
```

  And if you see an error within the logs along the lines of:

```
# There is insufficient memory for the Java Runtime Environment to continue.
# Native memory allocation (mmap) failed to map 2060255232 bytes for committing␣
↪reserved memory.
```

Please try recreating the default docker machine by:

1. Remove default docker-machine with the command:

```
docker-machine rm default
```

2. Create a new default machine with the command:

```
docker-machine create -d virtualbox --virtualbox-cpu-count=2 --virtualbox-
↪memory=4096 --virtualbox-disk-size=50000 default
```

3. Restart your docker environment with the commands:

```
docker-machine stop
exit
```

1.7.3 Redis

- `port 6379 already in use`

  This is because redis server is already on and you need to stop it first so that it can restart. So to get rid of this issue simply run the below command

```
/etc/init.d/redis-server stop
```

1.7.4 Node/npm/yarn

- When filling out the requirements of BookBrainz, you'll encounter an error that says you'll need to install postgresql-server-dev-X.Y for building a server-side extension or `libpq-dev` for building a client-side application To solve this problem, please install `libpq-dev` and `node-gyp`

  for ubuntu

```
sudo apt install -y node-gyp libpq-dev
```

CHAPTER 2

Our repositories

Our codebase is divided in a few repositories:

- bookbrainz-site:the main repository for the website and API
- bookbrainz-data-js: the ORM package we use to interface with the database
- bookbrainz-dev-docs: these very same docs you're reading !
- bookbrainz-user-guide: the user-facing guide for using the website
- bookbrainz-utils: database tools and scripts