Importing Data into BookBrainz

Shivam Tripathi
Email: shivamtripathi0108@gmail.com
IRCNick: bukwurm
GitHub: shivam-tripathi
Blog: shivam-tripathi.github.io

1 Overview

This proposal is aimed at importing data into the BookBrainz database from third party sources, while at the same time ensuring that the quality of the data is maintained.

1.1 Data Sources

Data can be possibly imported into BookBrainz in the following two major ways:

- Mass import the data using the third party data dumps/APIs
- Manual imports by users from various sources like on-line bookstores, review sites etc.

In this project, I will mass import the data from openlibrary.org , and if there is time left in the import phase I will work on Library of Congress dumps. The licensing information regarding the dumps can be found here and here. To aid the manual imports, I will write userscripts for goodreads.com and bookdepository.com and if time permits, for amazon.com (or any other source MetaBrainz community prefers).

1.2 Maintaining the quality of the imported data

To ensure that the quality of the data is maintained, each imported value will have to be validated by the BookBrainz editors. To do this, the imported data is not added right away into the **entity** object in the database, but instead saved as an **import** object. Unlike the **entity** object, the **import** object cannot be revised - and can only be either promoted to a corresponding **entity** type or deleted. This promotion to an **entity** object is done after an approval from the editors. Similarly, it is deleted if discarded by the editors. Also, the import object will not contain any relationship with other imports, as it is meant only to serve as a temporary vessel for the data until it is upgraded or discarded by the BookBrainz editors.

1.3 Proposed data flow

The data imported travels from source to finally become one of the entities in the BookBrainz database. This data life cycle has following steps:

- **Data sources**
  These are the data dumps and the various websites that have the data in the raw form, which needs to be cleaned up and moulded according to the BookBrainz database.
- **The import object**
  Once cleaned up, the data will be inserted into the database as an import object. This object cannot be revised, and simply serves as a temporary store of data before it is either accepted or discarded.
- **The entities object**
  The imports data is presented to the user on the BookBrainz site. The editor now makes one of the following choices:
  - **Approves the import**
    This then creates a new entity of the corresponding type in the database.
  - **Discards the import**
    In such a case the import data is deleted from database.

– **Decides to make some changes and then approve**
In such a case, the editor is presented a form with pre-filled details, using which s/he can edit the details of the import object. Once satisfied with the details, the editor can either promote the import to an entity or cancel the operation to return back.

The completion of the third step marks the end of the data flow. The proposed data flow could be better understood by Figure 1.

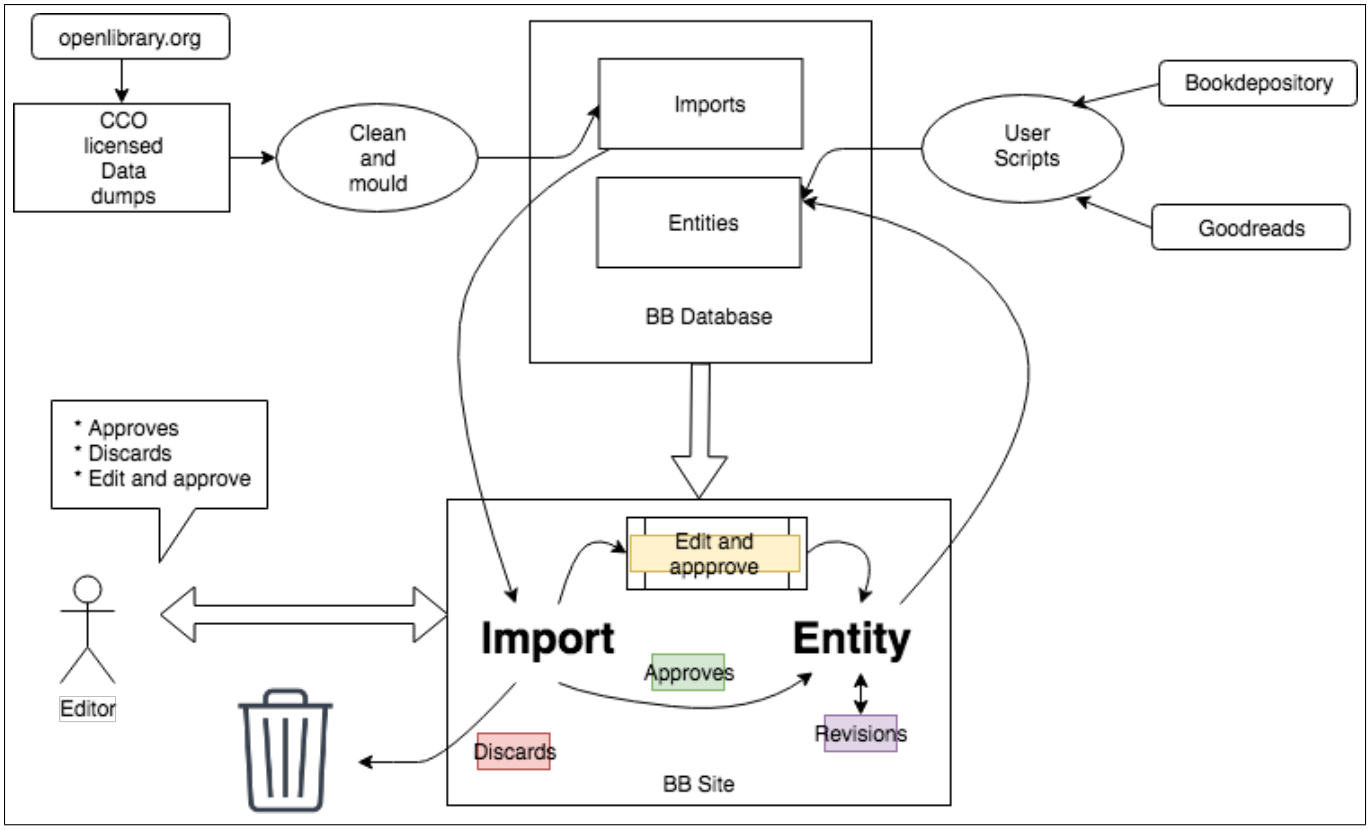

Figure 1: The imported data life-cycle

2 Implementation

The data to be imported has two attributes:

- **Structure**
  How the data would reside inside the database.
- **Behavior**
  How the data would flow, interact with the editors and how it's states would change from import to an entity .

2.1 Milestone 1

Milestone 1 would involve working on definition of the structure of the data that is imported. This involves properly working out the schema design and then making relevant changes to **bookbrainz-sql** and **bookbrainz-data** code. The proposed **import** object is similar to the entities, except the fact that it cannot be revised and can only be upgraded to the status of entities or deleted from the database. Some of the attributes the import object may hold are:

- Id
- Votes by the editors to discard/approve the imported data

- Source of the imported data (for example, goodreads.com or openlibrary.com)
- Date added

Presently each entity has **entitytype_header**, **entitytype_revision** and **entitytype_data** table, which respectively store the relevant information about the present master revision, all entity revisions and the complete entity data per revision. The proposed new **import** object cannot be revised, so it will not need **entitytype_revision** table. Also, there can be no master revision, so **entitytype_header** table is also not required by the import object. Only the **entitytype_data** table is required, and the existing tables can be used for the same.

Therefore, the changes to the schema roughly could be:

1. Addition of a table **import** with the following fields:
   - id: Incremental index
   - type: The type (string) of the imported entity, for example creator, work, publisher etc.
   - date_added: Date of import
   - source: Source of data (string)
2. Addition of tables **entitytype_import** , where entitytype signifies the type of the entity (namely creator, work, edition, publisher and publication) with the following fields:
   - import_id : The foreign key to the import(id) table
   - data_id : The foreign key to the entitytype_data(id) table
3. Addition of table **discard_votes** which stores all the votes cast to discard an entity.
   - import_id: Id of the import type
   - editor_id: Id of the editor who has voted to discard the import
   - date_voted: Date of casting the vote

The **discard_vote** table ensures that no two votes cast to discard the import are made by the same editor. This is done by making the tuple **(import_id, editor_id)** the primary key. The use of **discard_votes** table is based on the conservative approach to changing the state of an imported data which I will be explaining in detail later in this proposal.

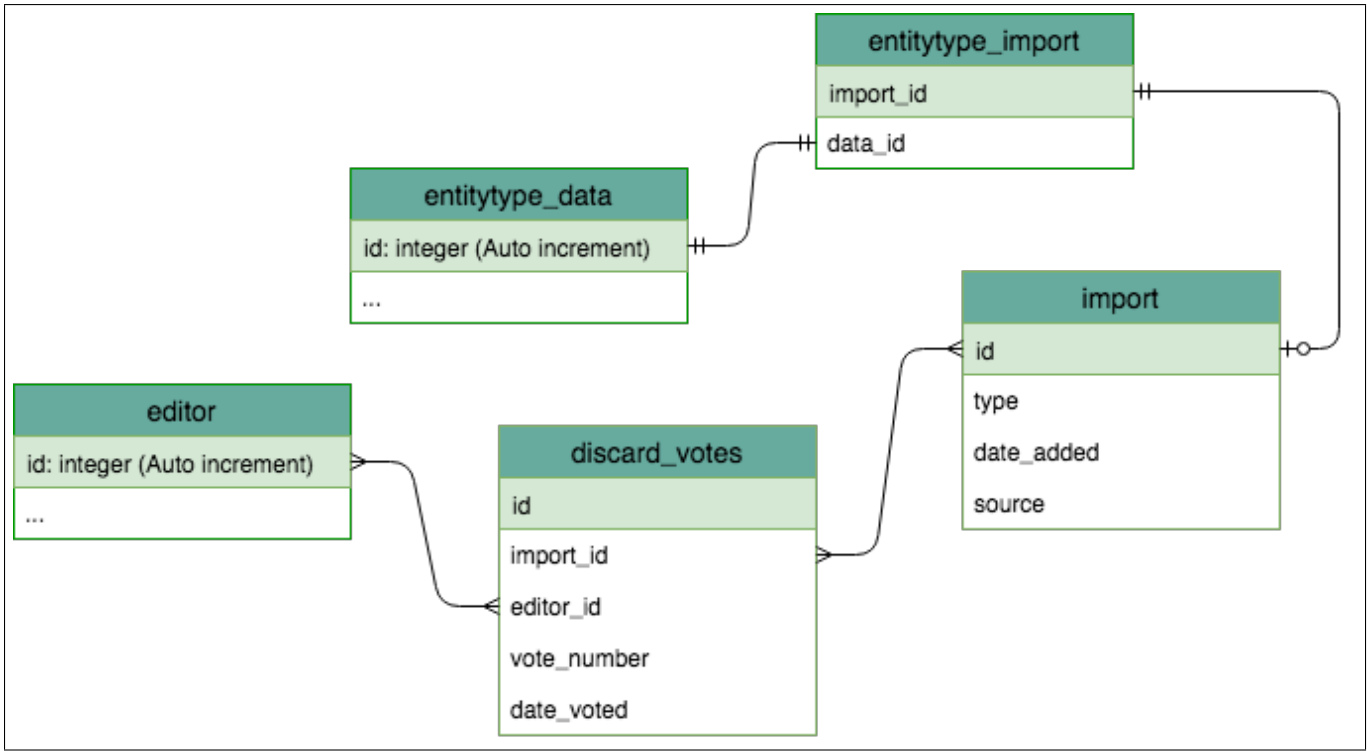

Figure 2: Proposed schema change

2.1.1 Code structure

For the schema change, most of the code would reside inside the bookbrainz- sql/schemas/bookbrainz.sql file. Commands to do this can be (creating tables and views):

```
CREATE TABLE bookbrainz.import (
        id SERIAL PRIMARY KEY,
        type bookbrainz.entity_type NOT NULL, date_added DATE,
        source TEXT NOT NULL DEFAULT '',
        voter_list
);

/* Commands for creator import */
CREATE TABLE bookbrainz.import_creator (
        import_id INT PRIMARY KEY,
        data_id INT
);

/* Commands for voting table */
CREATE TABLE bookbrainz.discard_votes ( import_id INT,
    editor_id INT,
    date_voted DATE,
    PRIMARY KEY (
        import_id,
        editor_id
    )
);

/* Foreign keys */
ALTER TABLE bookbrainz.import_creator ADD FOREIGN KEY (import _id)
        REFERENCES bookbrainz.import (id);
ALTER TABLE bookbrainz.import_creator ADD FOREIGN KEY (data_i d)
        REFERENCES bookbrainz.creator_data (id);
ALTER TABLE bookbrainz.discard_votes ADD FOREIGN KEY (import_ id)
        REFERENCES bookbrainz.import (id);
ALTER TABLE bookbrainz.discard_votes ADD FOREIGN KEY (editor_id)
        REFERENCES bookbrainz.editor (id);

/* Sample view for creator */
CREATE VIEW bookbrainz.import_creator_view AS SELECT
i.id, cd.id AS data_id, cd.annotation_id, cd.disambiguation_id,
        als.default_alias_id, cd.begin_year, cd.begin_month, cd.begin_day,
    cd.begin_area_id, cd.end_year, cd.end_month, cd.end_day,
    cd.end_area_id, cd.ended, cd.area_id, cd.gender_ id,
    cd.type_id, cd.alias_set_id, cd.identifier_set_id,
    cd.relationship_set_id, i.type, i.date_added, i.source, i.discard_votes
FROM bookbrainz.import_creator ci
LEFT JOIN bookbrainz.import i ON i.id = ci.import_id
LEFT JOIN bookbrainz.creator_data cd ON ci.data_id = cd.id
LEFT JOIN bookbrainz.alias_set als ON cd.alias_set_id = a
ls.id
WHERE i.type = 'Creator';
```

2.1.2 Case of edition_data

The edition table includes a field containing BBID of it's publication marked as compulsory. As imports have no UUID and relationships associated with them, this would require some rewiring or creating a separate table to store

imported data of edition type. There can be two possible solutions to the problem:

- Make *publication_bbid* field null for all imports, and enforce that it has to be not-null in the code for entities. This essentially means that the imported editions will not have publications attached. The editors can add them when upgrading to entity in the edit and approve page.
- Use UUIDs for imports as well, and add a separate column in the edition_data table to mark the type of entry - acting as a flag for whether it's an import or it's an entity. Upon up gradation, we will then assign the same UUID of the import to the entity as well, and update all publication_object_type to entity from import. This method would preserve the relationship between an edition and a publication.

2.1.3 Keeping other relevant data

While importing the data, we may come across some relevant information about the objects which we cannot introduce (yet) into the import data in the database. For example, import objects will have no relationships (as proposed). The imports will exist as single standalone objects, which will gain all the privileges once they are upgraded to the entity. Any form of relationship exiting in the dump object will be lost. Also, when upgrading an import we may be unable to introduce some information collected into the entity object as well due to the design of the present schema. Therefore, we need to store the meta data in such a format that if in future we decide to change the existing schema to add some field whose value already exists in the collected data, we are able to easily extract it and update the fields. There must also be a way to search the imports/entities based on this metadata. There can be two possible solutions to this:

- Clear up the meta data obtained and store it in the existing annotations table in the form of a json (or some other format) which could be read and parsed easily to extract relevant information.
- Construct a separate table which will hold all the metadata extracted out from the dump in form of a json field. Postgres 9.4 has the ability to store JSON as "Binary JSON" (or JSONB), which strips out insignificant whitespace (with a tiny bit of overhead when inserting data) but provides a huge benefit of indexing which helps when we query the data. This way, we avoid parsing each time and are directly able to search and extract the relevant part of the metadata. It needs to be decided how to keep this metadata table linked to both import and after upgradation to the entity object during it's life cycle. An example could be:

```
CREATE TABLE import_metadata (
        import_id INT, /* For imports */
        entity_id UUID, /* For promoted entities */
                import_id integer NOT NULL,
        data jsonb
);

/* Querying */
SELECT data->>'publisher' AS name FROM import_metadata;
/* Output */
name
------------------
IGC Code
International Code of Signals
IAMSAR Manual Volume III
Nautical Charts & Publications
(4 rows)
```

2.1.4 Interacting with the import data

The **bookbrainz-data** module will have to be updated with functions to access and modify the import objects. As the present direction of the module is to write query specific functions which return Immutable.js objects, I will follow that approach. This will also aid in testing, as results of each function could then be easily tested. Also, it will help to write pure SQL queries to interact with database. Roughly, functions for accessing different import objects, upgrading an import object, deleting an import object and adding a new import object will be required. Apart from that, handling additional tables like discard_votes etc. will have to be managed.

2.2 Milestone 2

The second step would be to import actual data into the BookBrainz database. I will be utilizing openlibrary.org and Library of Congress data dumps for this purpose. To do, I will write scripts that read the data, clean it up and then execute SQL queries to insert the data from the dumps into the database. To facilitate bulk insertion of data, I will consider the best practices to populate the database. I will cover some of them in more detail later in this proposal.

2.2.1 Handling large dump files

Dumps upon unzipping can easily acquire enormous sizes. For example, edition data from OL is a TSV file which exceeds 25GB upon expanding. To manage this, I propose to break down each file into small manageable chunks (of sizes around 250MB to 500MB) and run the code separately on each of them. As each line contains details of different entities which are completely independent of the other lines, we can split the entire document easily line by line. This can be done easily in the UNIX system using the following commands:

```
split -l [number of lines] source_file dest_file
```

This will split the source_file into multiple files with dest_file prefix. Once converted into manageable chunks, we can extract the data from them one by one, transform them into the desired format and then load them into the database.

2.2.2 Approaching dumps

One of the first thing to do would be to know what all fields exist in a particular dump. Once clear about the various fields in the dumps, it could then be figured out how to classify the various data fields and how to fit them into the BookBrainz data model. This would also help in understanding the edge cases and different forms of data that could creep in. This would also be of great help in validating the data, and testing the validation functions.

2.2.3 Structure of the dumps

In this section I will analyze the structure of the various dumps.

- Open Library The dumps are generated monthly, and have been divided into the following three subparts for convenience.
  - ol_dump_editions.txt
    - Size: 27 GB
    - Relevant json in the data dump (after being cleaned): edition_dump_sample.json
  - ol_dump_works.txt
    - Size: 9.5 GB
    - Relevant json in the data dump (after being cleaned): works_dump_sample.json
  - old_dump_authors.txt
    - Size: 2.23 GB
    - Relevant json in the data dump (after being cleaned): authors_dump_sample.json
- Library of Congress
  - Books All (A series of 41 dumps)
    - Size: 500 MB to 800 MB per dump
    - Relevant xml in the data dump: books_all.xml
  - Classification
  - ... Other dumps

2.2.4 Code structure

The functionality provided by the tools to import must consist of the following two parts:

1. **bookbrainz.import.generate_import**
   Data dump specific tools, which cleans up the data into a predetermined format. It's role can be summarized as follows:
   - Read each text files of the specific dump.
   - Extract relevant data field from the text file
   - Construct a generic object from the field and pass it to sub-module of bookbrainz.import.push_import to construct the import object.
2. **bookbrainz.import.push_import**
   A tool which works on a predetermined data format, validates it and imports the data into the database. Once developed, it could readily be reused for importing other dumps in future also. It's role can be summarized as follows:
   - Validate individual data fields
   - Encapsulate the data field into data objects
   - Validate the data objects
   - Push the data object into the database

I propose developing bookbrainz.import.push_import within the **bookbrainz-data** already existing as a npm module written in **javascript**, as a command line tool which could then be used for reading and pushing data into the BB database the data generated by bookbrainz.import.generate_import.
Another case can be to write a separate module in javascript or any other language which could be used by bookbrainz.import.generate_import separately.

The module structure **bookbrainz.import.push_import** could be:

- validators
  - common (including for encapsulated objects)
  - validators specific to each import type
- create_objects
  - create import objects of each type using the validators per field
- push_objects
  - push the constructed valid object into the database

The module structure of **bookbrainz.import.generate_import** could be:

- read data
  Run the script to input each text file.
- extract data
  Extract the required data fields from the objects construct a generic object: Construct the generic object to be used to contruct the import object.
- construct and push import object
  Call the relevant submodules from the push_data module to construct the relevant import object and push it into the database.

2.2.5 Validators and testing

The **bookbrainz.import.push_import** module involves validating the individual data to form a complete valid import entity. This requires the validation functions to be robust to all type of data thrown at them. Thus, thorough testing needs to be done to ensure that it can handle any form of data we throw at it. Writing validators will be greatly helped by the existing *bookbrainz-site/src/client/entity-editor/validators* module. All these validators would require unit testing to check if they perform as they are intended to. Secondly, we need to check also if the insertion of imports object occurs as required. This would require populating the **bookbrainz-test** database using sample input data, and then querying it to check if the final value inside the database is as expected. Similarly, other unit tests can be written to ensure that the system is robust to different forms of data and performs as it's intended to. One of the most necessary part of testing the validators would be the sample prototype dataset being used as arguments. This dataset is to contain all the corner cases so as to ensure that the validators are thoroughly tested and they do not pass any invalid data into the system. It could be built by sampling and adding some extra records which cover all different forms of values the data can take. It is necessary to work on this thoroughly, as the tests written presently would not just work on dumps presently being using to import, but also the future ones.

2.2.6 Bulk populating the database

Some techniques to bulk import data can be:

1. Creating a batch sequence of sql queries and executing them after a fixed batch size. In this case, I will disable autocommits and commit at the end of the batch. In case there is some error, the entire batch will be redone.

2. Write the validated imported objects to a flat file per dump subset, and then use COPY. Since it is a single command, it does not need to disable autocommit.

3. Before COPYing the data, dropping foreign key constraints and indexes and then importing the data leads to faster data load. Later the contraints can be added back. It would be ensured in the code that the data being imported satisfies the constraints while creating the import object. The recreation of foreign key constraints and indexing can be greatly sped up by increasing maintainance_work_mem configuration variable.

4. Run ANALYZE after altering the data contents of a table. ANALYZE (or VACUUM ANALYZE) ensures that the planner has up-to-date statistics about the table.

2.2.7 Import Endpoint

I will add an import endpoint route to the site, which will come in handy when adding the data via HTTP POST requests. Later if an API is developed, this end point can be removed. This will be crucial with respect to the implementation of various ways by which one tries to send the data object to this end point, requiring the data sent to become an import instead of an entity. Building it would involve designing a fixed data object which encapsulates all the necessary details for building any one or multiple import object. This would require changes in *bookbrainz-site/src/server/routes* and *bookbrainz-site/src/server/helpers* modules.

2.3 Milestone 3

The third milestone will be to add the behaviour of the data. Once inside the database, the import objects have to travel to the editors via the bookbrainz-site so that they could reach the next stage of it's life cycle. I propose to do this via two means:

- Show up the imported data in the search results

- Separately allow the users to see and choose to review what all data has been imported and is waiting for review.

2.3.1 Search results

Presently, BookBrainz uses Elasticsearch to facilitate search. I will update the Elasticsearch indexing so that the imported objects appear in the search results on the bookbrainz-site. The front-end on the website will be modified to reflect these changes, so that upon search the users of the site would know if the search result is one added by another editor or if it has been automatically added and awaits review. One of the plausible ways to do this can be:

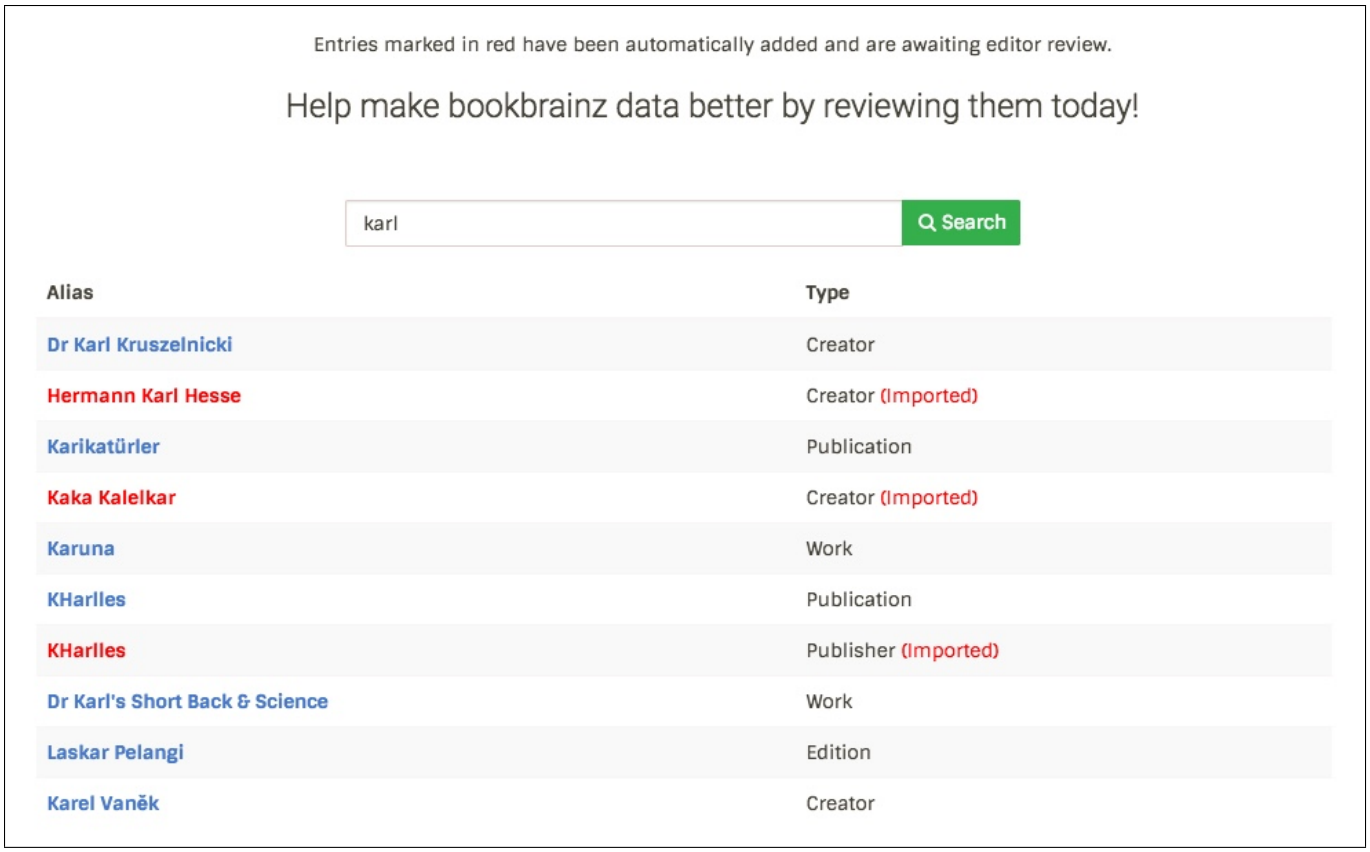

Figure 3: Proposed search results page with imported data highlighted

Upon clicking an imported data, the user will be directed to the imported entity page. Updating the Elasticsearch indexing would need updating the *generateIndex* function in *bookbrainz-site/src/server/helpers/search.js* which would add the imports when indexing the ES after a restart. Also, separate functions would be required to insert new imports and delete the approved imports from the Elasticsearch index. The function to insert new imports into the index would also be useful for add_import_endpoint, so that as soon as an import is added to the database, it is also added in the ES index.

2.3.2 Imported entity page

This is the landing page of the imported data. It will be very similar to a regular entity page, except with the following options instead:

- Approve
- Edit and approve
- Discard
- Data source

One possible way to implement the page can be:

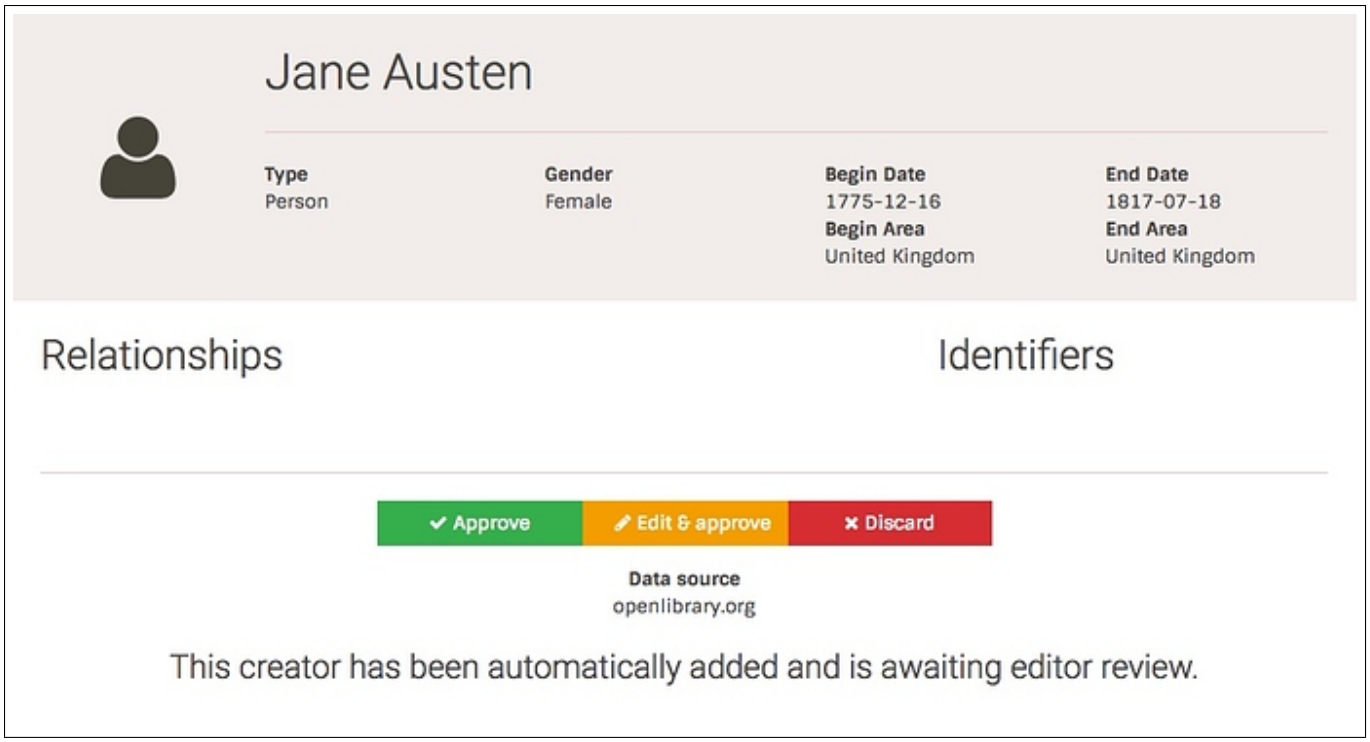

Figure 4: The proposed 'Imported entity' page

2.3.3 Changing state of an import

As importing of data takes time and effort and discarding an import leads to permanent deletion, I propose keeping a conservative approach towards discarding an import, but being liberal while upgrading, i.e. we make it easy for an import to get included in the database as an entity, but difficult to be removed. I propose that we attach a field 'votes against' which is incremented every time an editor says that the import should discarded, and discard an import only if the number of votes reach a threshold. However, we immediately promote an import if an editor marks it approved. Whenever the import object is deleted or upgraded, we delete the search index to ensure that it doesn't turn up in the results. Upon upgradation, the new entity also needs to be inserted into the ES index.

2.3.4 Review page

Another way for the editors to interact with the imported data is to provide them with a page that displays only the imported data. That way, the editors could choose to review imported data from the list. This would include adding a review button at the top, next to 'create' button. Upon clicking the 'review' button, the user would be directed to the page containing a list imported data, with a limit on imported data shown per page. The user can browse through the list and can choose which import s/he intends to review. Upon clicking any item in the list, the user would be directed to the imported entity page of that particular import.

Figure 5: Addition of the 'Review' option

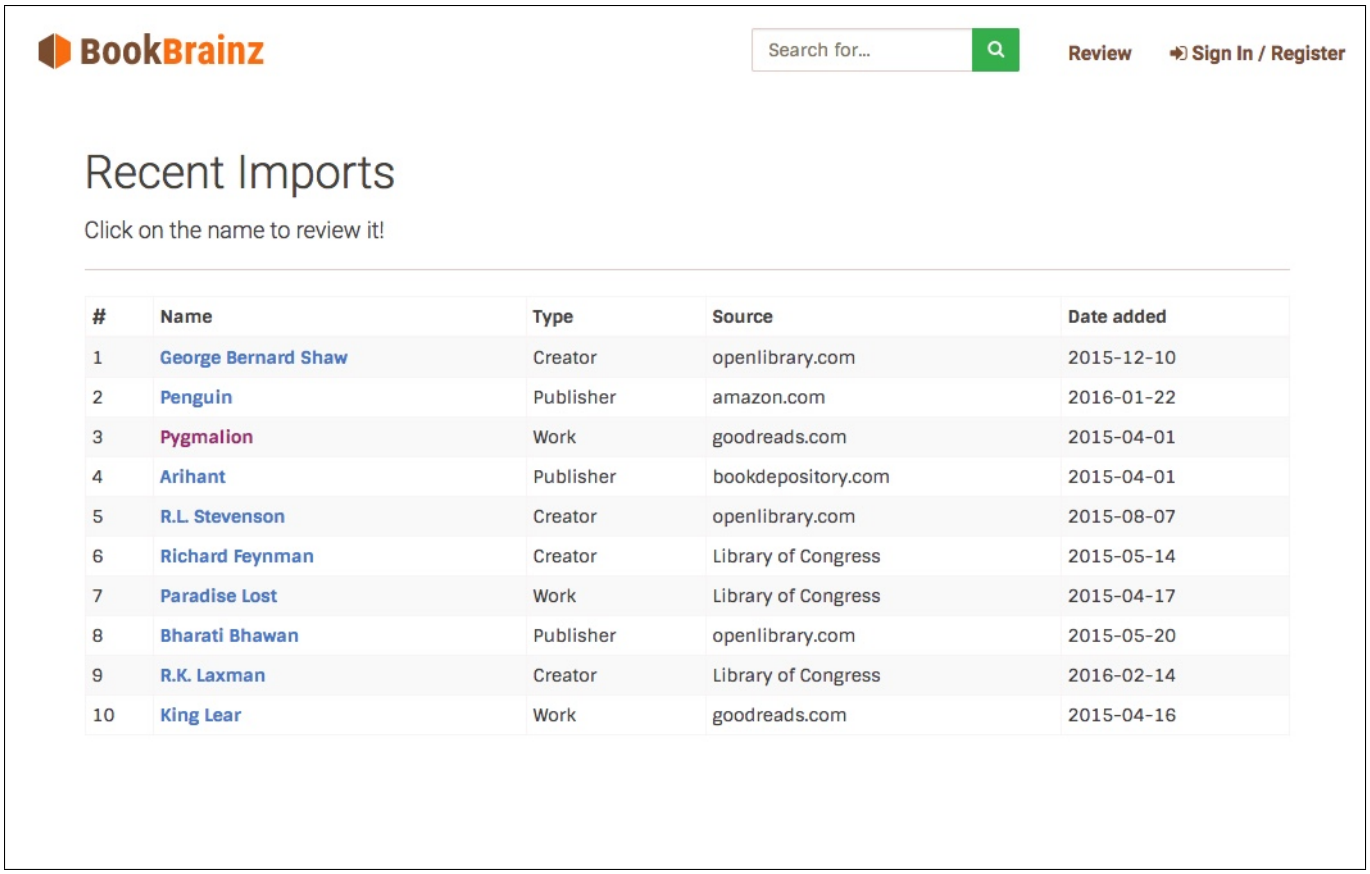

Figure 6: The 'Review' page

2.3.5 Imported data and non-editors

Presently, the users who are not logged in can search the BookBrainz database and view the results. Adding the imported data can now effect this search in two ways:

- Not allow the display of imported data to a user who is not an editor in the search result.
- Allow display of all the data BookBrainz has in the search result. Upon clicking the result, the user will be led to log in page. Only after logging in can user visit the imported entity page and approve/discard/edit and approve the import.

I would like to go with the latter implementation, unless decided otherwise.

2.4 Milestone 4

The last step would be to write userscripts to facilitate seeding of data from third party websites like goodreads, amazon, bookdepository etc. The userscripts written would be run using the tampermonkey or greasemonkey extensions in the user browser.

2.4.1 Functionality

Aim of this tool is to reduce the effort of an editor when s/he decides to add an entity's information into the database by directly sending over the data to the BookBrainz database. If the user intends to make changes, s/he can do so at any point of time at the bookbrainz-site.

- When a user opens up a site containing data about books (for example, bookdepository.com), the browser fetches the website content.

- If the URL of the website is the one on which the userscript has to become active, the userscript starts and collects relevant data from the page (by analyzing the DOM).
- The userscript then constructs relevant import objects from the collected data.
- The userscript also adds a Import to BB button to the page at any location easily visible.
- Upon clicking the button, a modal opens up containing a list of all the entities (creator, work, publisher, publication etc.) which could be imported from the page followed by a button to search the entities on the bookbrainz-site and a button to import the entity.
- Upon clicking any one search button, a new tab opens up of bookbrainz-site search page with query of the entity's name. Another alternative would be to display a warning if the entity exists in the database.
- Upon clicking any import button, the site sends the data directly to the BookBrainz-site.

The userscripts would be written separately in a different repository with name **bookbrainz-scripts** and made available to the users with a single click.

2.4.2 Code structure

Roughly, overall code can be broken into following two parts for every userscript:

1. The part which returns UI components (button, form etc.) to be inserted into the relevant URL. This is common to all userscripts.
2. The part specific to each URL - where to inject the button, from where to get data, how to construct the data object etc.

The first part will be implemented separately. It can be included in all the all the userscripts either by bundling them together, or *// @require* them in the code. I intend to use React to build the relevant components. It is unnecessary to use any bundler, but it could be beneficial in case of writing more modular ES6 format code.

More specifically, the code can be split up as:

- bookbrainz-userscripts
  - src
    - lib
      - UI Components
      - Application's business logic to wrap up the extracted data according to the end point
    - All other userscripts

This design is inspired from *musicbrainz userscripts*. It is robust to future additions, as the *lib* contains the generic userscripts including the UI components and other data cleaning and encapsulation methods, which are used by other scripts. The userscripts are written alongside the lib, and can be directly included in a CDN like manner using *// @require lib/{script_name}.js* tag in the header, if *// @namespace* is also mentioned pointing to the source code hosted online. Any new userscript can be written along side the lib package is it aims at providing site specific tools, or else add it in the lib package which hold all the common case scripts.

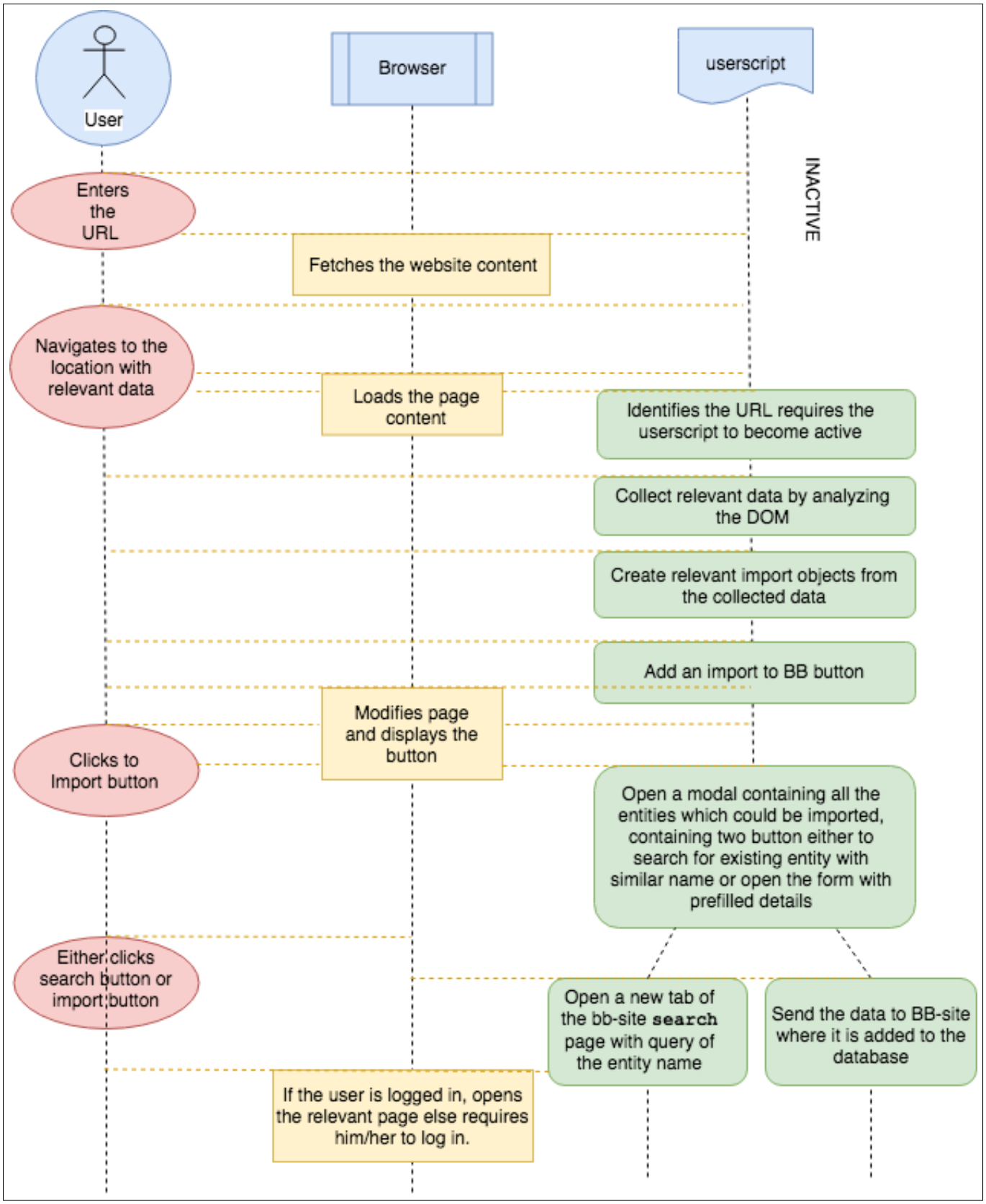

Figure 7: The sequence diagram on a timeline for the userscripts

3 Documentation

Schema changes would require proper documentation for future purposes, which I will write in the official developer docs. Most of the remaining work would be in javascript, which will be covered by JSDocs. Additionally, in every buffer week I will spend time to record all the work carried out in the previous phase, which will cover the developer side of the documentation. Addition of imported objects is a huge overhaul in the present state of site, and some users (old and new) might easily become confused about how to use the new features. To tackle this, I plan on writing a concise but complete explanation of the new import object in the website in a user friendly manner, which could be added on the website itself (with links to it lying all over the website) or on read-the-docs.
Writing proper user documentation is as useful as addition of the new feature, as if the editors themselves are unaware of how to use it, the addition would be rendered useless. One of the ways to easily integrate FAQs into the site with search option embedded could be by using gitbook static site generator, which reads developer friendly markdown to convert it to a static site. Additionally, I will open up multiple questions on the metabrainz community which answer some common FAQs regarding the new features and imports. I will also work on documenting some of the present features.

4 Timeline

4.1 Community Bonding (April 24 – May 13)

In this period I aim to familiarize myself more with MetaBrainz community and it's projects. I will talk with the BookBrainz users and MetaBrainz community members and know what all they would like to see improved or additionally implemented in my GSoC project. Additionally, I will write some documentation and FAQs for the project and edit entries in BookBrainz and MusicBrainz data. I will discuss with the mentor and confirm the implementation details, based on which I will make relevant changes to the proposed plan, adding, removing or improving as required.

4.2 Phase 1 (May 14 – June 11)

4.2.1 Week 1

Start working on Milestone 1. Write SQL queries reflecting schema changes in bookbrainz-sql and write relevant functions to access the newly created import object in bookbrainz-data.

4.2.2 Week 2

Start working on Milestone 2. Prune the dumps to a manageable size, while maintaining diversity. Draw a basic layout of the data dumps and propose a generic data object to be developed from the dumps to be used later by import script.

4.2.3 Week 3

Write scripts to pull data out of openlibrary.org and Library of Congress dumps and create previously decided data object.

4.2.4 Week 4

Write scripts to feed the generated data object to the database. Develop the import endpoint at bookbrainz-site.

4.3 Phase 2 (June 12 – July 9)

4.3.1 Week 1

Buffer week. Catch up if the data import is lagging behind, document code, fix bugs, write tests and clean up the code. Take reviews from the mentor and make relevant changes. Unit tests will be written using mocha.js and chai.js.

4.3.2 Week 2

Start working on Milestone 3. Update the elasticsearch indexing to make the imports appear in the search results. Start working on the review page.

4.3.3 Week 3

Finish off the review page. Finish the imported entity page.

4.3.4 Week 4

Start working on the 'edit and approve' page of the imported entities. This will include issuing warnings to the user if the name of imported entity already exists.

4.4 Phase 3 (July 10 – August 6)

4.4.1 Week 1

Buffer week. Catch up if the milestone 3 is lagging behind, fix bugs, document code, write tests and clean up the code. Take reviews from the mentor and make relevant changes. Unit tests will be written using chai.js and mocha.js, as already being used in bookbrainz-site.

4.4.2 Week 2

Start working on Milestone 4. Set up and configure the project (transpiler, linting, bundler etc.). Write the UI rendering modules.

4.4.3 Week 3

Start working on userscripts for goodreads.com, and if time permits for bookdepository.com and amazon.com.

4.4.4 Week 4

Clean up the code and write documentation. Discuss with the mentor relevant changes before the final submission of the work.

4.5 Post GSoC

I will continue to contribute to the BookBrainz project, and also try my hands at other *brainz projects. I wish to learn from the MetaBrainz projects all about "the art of data hacking".

5 Detailed information about yourself

I am an undergraduate student of Computer Science and Engineering at the Indian Institute of Information Technology, Una. I learned about GSoC from a friend.

Tell us about the computer(s) you have available for working on your SoC project!
I own an Apple MacBook Air Core i5 5th Gen (8 GB/128 GB SSD) running as of now Mac OS High Sierra 10.13.1. To handle the data dumps, I have ordered Seagate 1 TB Wired External Hard Disk Drive. It will arrive before the program begins.

When did you first start programming?
I started writing code in Java in my secondary school. I picked up some Python and C++ in my freshman year. Recently I have started working in javascript.

What type of music do you listen to? (Please list a series of MBIDs as examples.) If applying for a BookBrainz project: what type of books do you read? (Please list a series of BBIDs as examples. (And feel free to also list music you listen to!))
The choice of music mostly depends on the mood. However, I do listen a lot of pop (especially 80's/90's), house (progressive and deep mostly), sometimes hip-hop and a little rock here and there. I love to read. Some of my favourite books are GBS's Pygmalion, GO's 1984 and CD's A tale of two cities. I also follow comics, American Vampire being one of my favorites.

What aspects of the project you're applying for (e.g., MusicBrainz, AcousticBrainz, etc.) interest you the most?
As I mentioned, I love to read; so BB is the obvious choice. On top of that, I am a staunch believer of open data for inclusive education.
Have you ever used MusicBrainz to tag your files?
I often use Picard.

Have you contributed to other Open Source projects? If so, which projects and can we see some of your code?
I have contributed to GNOME photos and OpenDataKit in the past.

What sorts of programming projects have you done on your own time?
I have written a multi-threaded event driven server using libevent with self designed application layer protocol to communicate and transfer files.
I have worked for a joint research program undertaken by IIT Kanpur and Nissan-Renault to develop a 360 degree vision driver assistance tool. It was written in C++. My role was to integrate and display the data (Ip-camera feed and other relevant data) into the existing system, which could be later used for different forms of analysis.
I recently wrote a very simple cryptocurrency in node using express.js framework with a simple HTTP API.

How much time do you have available, and how would you plan to use it?
I plan on putting in at least 40 hours a week on this project.

Do you plan to have a job or study during the summer in conjunction with Summer of Code?
I have no other obligations during the period.