Cardiff University

Final Report

Final Year Project

CM3203 – 40 Credits

BibTeX Web Database

Author
Mr Miles Budden

Supervisor
Prof. Frank Langbein

June 5, 2020

Abstract

When researching for a paper, report or other written piece, the need for a place to store references is universal. There are many existing solutions for such a problem, however, many fall short with required features. Be these multi-user collaboration, the ability to self-host, or accurate parsing of sources into references, few solutions fill all of these requirements. This project aims to produce a solution that can fulfil these requirements.

Contents

List of Figures

1 Introduction

1.1 Aims

The initial description given to this project was as follows:

"BibTeX is a system to keep track of references and include them in TeX/LaTeX documents. It keeps the information about the references in a single file. This is problematic when more than one person wants to edit the file and keep entries synchronised. The idea of this project is to create a web interface to edit a BibTeX database, where individual entries are stored in a NoSQL database. An interface to LaTeX editing systems (to create the BibTeX database for a specific document), and searching and annotating the references in the database are essential. Interfaces to bibliography databases, journals, browser extensions to enter references, etc. are also an option to consider, to simplify editing and populating the database. Ideally, the database would also hold links to papers or the actual PDF files of the paper and can extract the citation information from those files directly. Potentially bibliography information could be extracted directly from PDFs or related web resources into the database instead of manually entering these."

From this initial brief, several initial aims can be determined:

- The project should include a web interface.
- The user should be able to edit the BibTeX entries from this interface.
- Individual entries should be stored in a NoSQL database.
- The project should interface with LaTeX through the use of BibTeX.
- The user must be able to search and annotate the entries in the database.
- There could be a browser extension to enter data into the database.
- The system should hold references to the PDFs or store the PDF.
- The system could extract information from the PDFs.

1.2 Audience

Due to the technologies that this project will integrate with, and the specificity of the need to manage citations and references, the target audience for this project is relatively small. There are three main groups of people who require the use of reference management software: students, researchers, and academics. Of the members of these groups, the proposed project would only be of good use to those who use LaTeX and therefore BibTeX. On average, 26.8% of papers submitted to journals are typeset in LaTeX, although this value is significantly higher for fields such as mathematics (96.9%) and physics (74.0%) [1].

Despite the relatively small audience, this is a tool that is vital to collating and collaborating on research and would, therefore, see extensive use from any individuals from the specified demographic.

1.3 Scope

The scope of the proposed project is limited by three factors. The first is what is needed by the audience. The brief clearly outlines the expectations of the project and therefore the aims. The aims outline project of considerable complexity and it would, therefore, be out of scope to add features and functionality that are not present in these aims.

The second factor is time. As shown in the initial plan, there is a limited time frame for this project. Included in this time frame is the report. Therefore, the number of weeks available for the project is limited which thus limits the scope of what can be achieved.

The final factor that limits scope is my technical ability. Although I regularly use and am experienced with back end technologies, I have not used front end technologies and frameworks in recent years and therefore my knowledge is out of date and limited. This limits the scope of the front end as the development will be slower and involve more out of date technologies.

Through these limitations, I expect to be able to fulfil all of the aims specified in the required time frame.

1.4 Assumptions

The main assumption that I make for this project is that the end-user is using BibTeX to manage their references. This is due to the fact that it is specified that the website should export its references at BibTeX and would, therefore, be difficult to use with other reference management tools such as those built into MS Word. It would be possible to integrate the project with Word, however, this is beyond the scope of the project.

The second assumption is that the only type of reference that the user wants to be automatically parsed is an academic paper. I made this assumption as if it was any type of reference, then the metadata parsing would have to be significantly more complex and therefore beyond the scope of the project.

1.5 Summary of Important Outcomes

The brief outlines a tool that would be valuable to researchers and students alike. Even if not all of the aims are met by the deadline, it is important that a usable piece of software is produced that has the basic functionality to improve the research flow. From the aims, the core functionality can be defined as the user being able to add references to the database and to be able to access them in such a way as to be easily addable to a BibTeX database for use in a LaTeX document.

Asides from this requirement, the other features mentioned, although they would add to the user experience, are not required to use the product and are therefore secondary to the aforementioned core functionality.

2 Background

2.1 Existing Solutions

BibTeX has existed for many years [2] and therefore it is natural that there are many tools to interact with it [3]. Although many exist, there are few that firstly, have a web interface, and secondly, are open source and are therefore available to self-host. Of those listed, three offer up to date web interfaces: Mendeley, EndNote, and Zotero. Of these, only Zotero is FOSS (free and open-source software) and therefore available to self-host. Although the source is available [4], there is no documentation and therefore it would not be feasible to easily deploy this. For the purposes of identifying important features of existing solutions, I shall be examining Mendeley as it is the most fully featured and contrasting it against my proposed solution.

2.1.1 Metadata extraction

The most useful feature of a reference management system over a different form of storing references such as a text file is the ability to add references from unstructured data such as a PDF or web-page automatically. Mendeley supports this with four different methods of inputting data. The first is the most simple. This is presenting the user with a form that allows them to manually enter the data. This provides little advantage to entering references manually such as in a text field. The only advantage that this beings is that the reference management system can provide templates for common reference types such

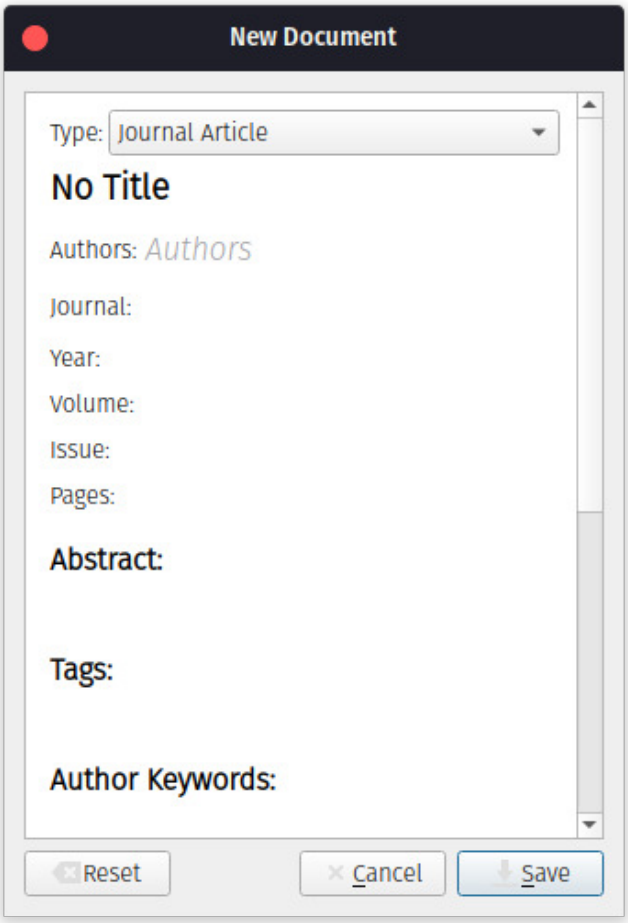

Figure 1: Manually adding a reference in Mendeley

as an article or book. Mendeley provides this feature on their desktop application as can be seen in Figure 1. This project will implement similar templates. The templates will be able to be specified by an admin and presented to the user as options when they wish to enter a reference manually.

The second method is to import existing structured forms of data such as BibTeX files. Mendeley supports the importing of several data types such as BibTeX, EndNote, and Research Information Systems (RIS) files. This project will allow the user to upload BibTeX files. It will not, however, allow other types of files. This is because the specification provided does not require this. This feature could be easily added however.

The third way to enter a reference would be through uploading a PDF. Mendeley supports this in a variety of ways. The user can either use the browser extension, the website, or the desktop application to do this. The desktop application allows for multiple PDFs from a directory to be added at the same time to streamline the uploading process. This project will allow PDFs to be uploaded from the website. This will only allow one PDF to be uploaded at a time but could be changed to allow the upload of multiple PDFs.

The final way is to enter a URL of a paper. Mendeley supports this by adding a URL using the browser extension, however, the desktop and browser clients do not allow the user to enter a URL. This project will allow the user to enter a URL in two ways. The first is through the browser extension. When the user is on the page they wish to add, they can use the extension to enter the page as a reference. In the main application, one of the options for adding a reference is to add a URL.

2.1.2 Interface

Mendeley is available on four platforms. These are web, desktop, browser extension, and MS Word extension. This project will only be available as two: web page and browser extension. Due to the platforms that Mendeley is available on results in a potentially better user experience as it can leverage native user interface elements as well as access to the local file system. This project will only be available through a web interface and therefore suffers some reduced user experience as a result.

2.1.3 Self-hosting

As it is closed source, Mendeley is not available to self host, even as a pre-packaged binary. The closest feature it offers is the desktop client which can store some data locally. This project, how-

ever, allows the user to host their own instance and even compile the extension with a specific target.

2.1.4 Conclusion

Although Mendeley has many fully featured clients and more functionality than the proposed project, the limitation that it cannot be self hosted rules it out for many users. This project, although lacking in a few of the features, allows the user to self host and therefore fulfil a niche that Mendeley does not fulfil.

2.2 Problem Areas

The solutions specified in section 2.1 as well as the proposed solution can be decomposed into three separate parts: entry of data, storage of data, and retrieval of data.

Each of the solutions specified in section 2.1 allows the entry of data in three ways. The first is manual. This area will be relatively simple to implement as only a form is required. The second is through a given URL. The data at that URL will have to be parsed and a paper retrieved. This process is simplified by most papers providing metadata in a predefined format [5]. The final method for data entry is to provide a PDF to be parsed. This method raises several issues. The first is extracting raw data from the PDF. This raises such issues such as OCR (optical character recognition) for papers that do not contain embedded text as well as identifying contiguous sections of text [6].

Although the BibTeX specification has a base set of field types [7], users can add extra field types for use with BibTeX alternatives such as BibLaTeX. Therefore, in order to store the reference data in a structured format, the use of NoSQL document storage would be more suited than traditional relational storage as the schema of each reference is unknown.

As previously stated, the main way to retrieve data from the database will be in the BibTeX format. In order to locate specific papers, the user will need to search the database. The benefit of sorting the references as structured data and not as plain text is evident here as the user will be able to construct more specific queries with regards to differentiating between field names and values. As well as searching the references, the user should also be able to search the full-text of the referenced PDFs. This raises the aforementioned issues with extracting text from PDFs.

3 Specification and Design

3.1 Requirements

The requirements for this project are derived from the project brief, aims, and discussion with the supervisor. The requirements have been divided in two ways. They have been divided into functional and non-functional requirements. The functional requirements specify what features the project should have and then non-functional requirements specify how the project should run. They have also been divided into *must*, *should*, and *could* haves. The *must haves* are features are core features of the project and must be completed in order for the project to be considered a success. The *should have* features are features are ones that would benefit the project but are not essential. These features are within the scope of the project and are the difference between a basic and good finished project. The final type is the *could haves*. These would be great additional features to the project but are likely beyond the scope and timescale of this project.

3.1.1 Functional

- The system *must* be accessible from a browser.

- The system *must* be able to determine between different users.
- The user *must* be able to add references to the database.
- Multiple users *must* be able to edit the same reference database.
- The user *must* be able to view the references for all databases they are a member of.
- The user *must* be able to search for references.
- The user *must* be able to export a reference as a BibTeX file.

- The administrator *should* be able to approve new accounts before they are used.
- The administrator *should* be able to view and edit all data stored by the system.
- The user *should* be able to add a reference by uploading a PDF.
- The user *should* be able to upload a reference by providing a URL.
- The user *should* be able to upload a reference/s by providing an existing BibTeX file.
- The user *should* be able to add a user to a group they are a member of.
- The user *should* be able to filter search results to specific BibTeX fields.
- The user *should* be able to search the full-text of uploaded PDFs.
- The user *should* be able to edit the reference once uploaded.

- The user *could* be able to add URLs by the use of a browser extension.
- The user *could* annotate the PDFs in the browser.

3.1.2 Non-functional

- When running in production, the system must be available 99% of the time.
- The interface must be pleasant to view.
- The interface must be intuitive to use.
- Each page must load in no more than 1500ms (excluding loading of the first page).
- The application must be able to be accessed securely.
- The application must be able to be viewed in all modern browsers.
- The user should be aware of the state of the system upon error.

3.2 Architecture Overview

The project is decomposable into three different sections. These are the webserver, the PDF extractor, and the database. The separation of each of these services into their own services benefits the project in two ways. The first of these is stability. If one of the services was to fail, the others could continue to run. Although some services rely on others, if there is no dependency, the project could continue to function so some extent. For example, if the PDF extractor were to fail, the majority of the website will continue to function.

The second benefit is that the project will be modular. This allows sections of the project to be changed with greater ease. For example, if the database needed to be changed, a separate database module could be added and would only require small changes in the other modules.

3.3 Existing Tools

3.3.1 Database

In order to store the data, a document storage database would be more suited than a traditional

relational database for reasons stated in section 2.2. The most common NoSQL database is MongoDB. This would be suited to the storage of the BibTeX data although it has two shortfalls. The first is that although the project requires the BibTeX data to be stored in a NoSQL fashion, the project also needs to store other data such as user information that is more easily stored in a relational database. The second shortfall is that with default settings, it has been shown to have "*read skew, cyclic information flow, duplicate writes, and internal consistency violations*" [8] when using default settings.

A mature solution which supports both document storage and relations is PostgreSQL. As one of its field types, it supports JSONB [9]. This is a binary representation is JSON which supports other features such as GIN and GIST indexes.

3.3.2 PDF parsing and extraction

There are several methods for extracting the metadata from the plain-text of PDFs such as the use of a Multi-Layer Perceptron [10] or Hidden Markov Models [11]. An evaluation of common implementations of these methods as well as others [12] shows that an implementation of Conditional Random Fields called GROBID [13] results in the greatest accuracy of extraction of fields such as the title, authors, and date. Mendeley outperformed GROBID in some areas however the metadata extraction is not available outside of Mendeley.

GROBID is suitable for the modularity aspect of the project as it is entirely self-contained. Interaction with it is through an API it exposes over HTTP.

3.3.3 Front-end

Due to the simplicity of the front-end of this project, due to most of the interactivity coming from the dynamic back-end, Foundation is a suitable framework to use for front end development as is has many, easily themeable components.

In order to implement the interactive aspects of the front end, JQuery is suitable. Although it is now an old web technology, I have experience using it and for the scope of this project, it is suitable.

3.3.4 Server

In order for the project to serve different pages to different users, the website must be dynamic. There are many existing tools to do this. Django is a mature and fully featured web framework for Python that allows Python to interface with a web server through the WSGI [14] or ASGI [15]. In order to serve this, Uvicorn can be used. This is a fast web server designed for serving ASGI. As well as the dynamic content from Django, static content must also be served. Although Uvicorn can do this, Nginx can be used to take load off the Python server.

3.3.5 Architecture

In order to easily create isolated modules with identical behaviour in all environments, Docker is a tool to make containerisation on Linux systems easier. It provides the tooling to create containers that act as virtual machines for each of the sections of the project as well with less overhead than standard virtual machines as virtual networking to network them together in a secure way (see Figure 2). As well as this, there are several tools which make development and deployment of these *containers* simpler. Docker Compose allows for configuration files specifying which containers should be run and how they should behave. Docker Swarm and Kubernetes allowing for complex deployments by allowing the user to specify the resources needed and ability to run the containers over a cluster instead of a single host.

Each of the previously mentioned existing tools

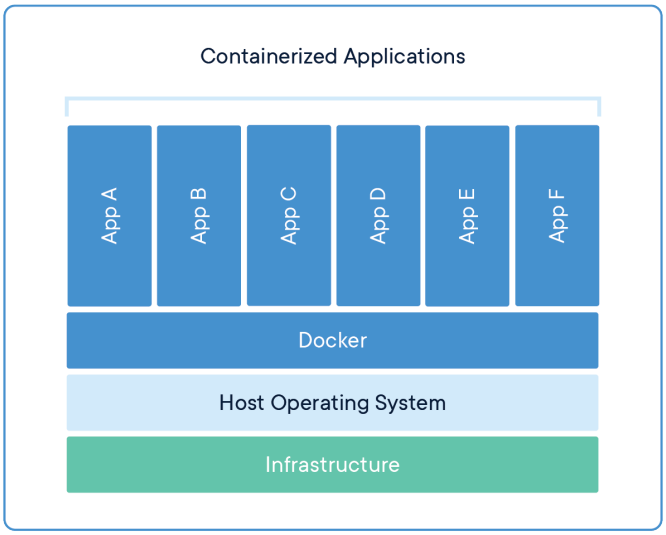

Figure 2: Docker architecture [16]

are easily used inside of these containers. Tools, such as GROBID and PostgreSQL, which usually have a complex deployment process are easily deployed using these containers as images for Docker already exist for these tools which can be used in containers, thus simplifying the configuration process.

3.3.6 Security

Although the project would work using the previously mentioned tooling, it lacks security. Træfik is a reverse proxy that adds HTTPS and HTTPs upgrade to HTTP requests. It is more suited to Docker than alternatives such as Nginx as it supports Docker auto-discover as well as automatic negotiation with Let's Encrypt, thus making deployment much simpler.

3.4 Data Flows

Within the project, there are 2 main actions that cause data to flow through the application. The first of these is when the user uploads a reference (see Figure 4). The system accepts three kinds of reference upload which makes the possible flow of data more complex than a single type of upload. As well as uploading data, the user must also be able to download it. This process is much simpler as there are only two methods (see Figure 3).

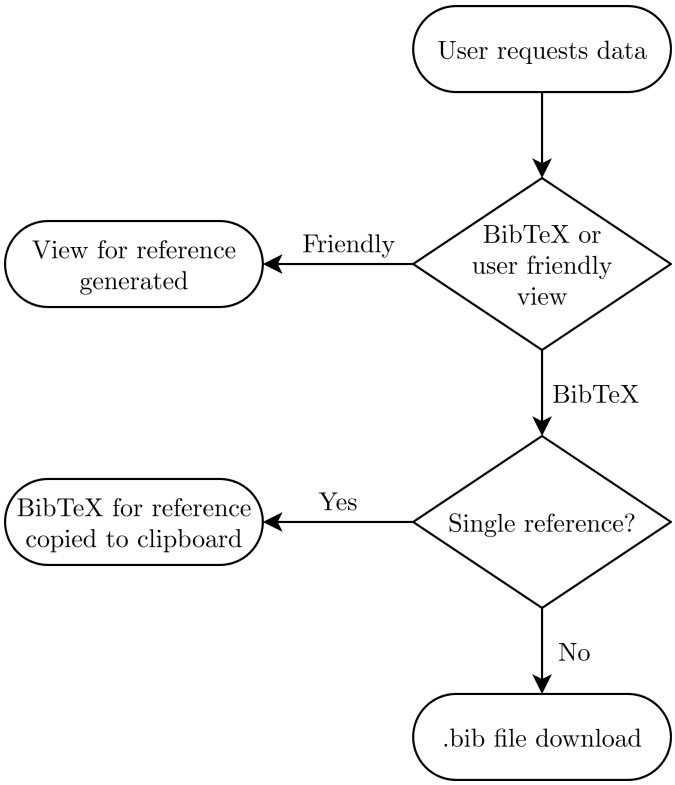

Figure 3: Data download

3.5 Database Design

As can be seen in Figure 5, the database for this project comprises of multiple tables. Although an in-depth, table-by-table breakdown can be found in Section 4.5, I will do a high level overview. To understand the database design, the design model for the project must be understood. The most basic object in the project is the user. This object is unique for each user. There are two types of user: the admin and the standard user. The admin user will also have access to the admin views (see Section 4.2.1). Groups represent a single BibTeX database. There is a many-to-many relationship between users and groups. This many-to-many relationship is implemented using a third table that stores users' memberships to groups. Each group can store multiple references. These refer-

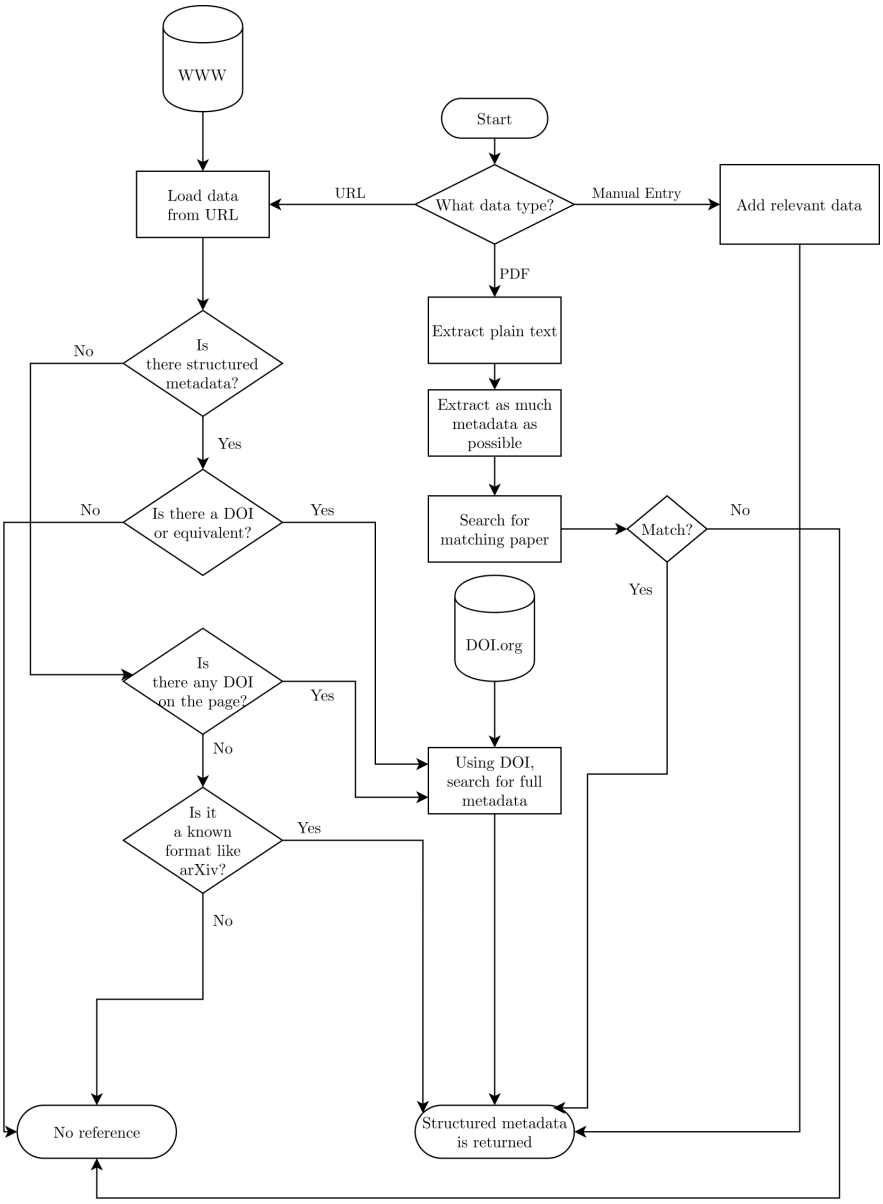

Figure 4: Data upload

ences are specific to a group. Each reference can also have files associated with them.

Separate to the core reference storage and access objects are the reference templates. These are stored by having a many-to-one relationship between many fields to one reference type.

4 Implementation

4.1 Metadata Extraction

The extraction of the metadata is divided into two parts. These are the extraction of partial metadata from the PDF and consolidation of the paper's full metadata. This division into two parts came as a result of two factors. Firstly, extraction of the metadata is not always accurate, so by consolidating it against an external source means that any small mistakes can be corrected. Secondly, not all of the data required to form a full

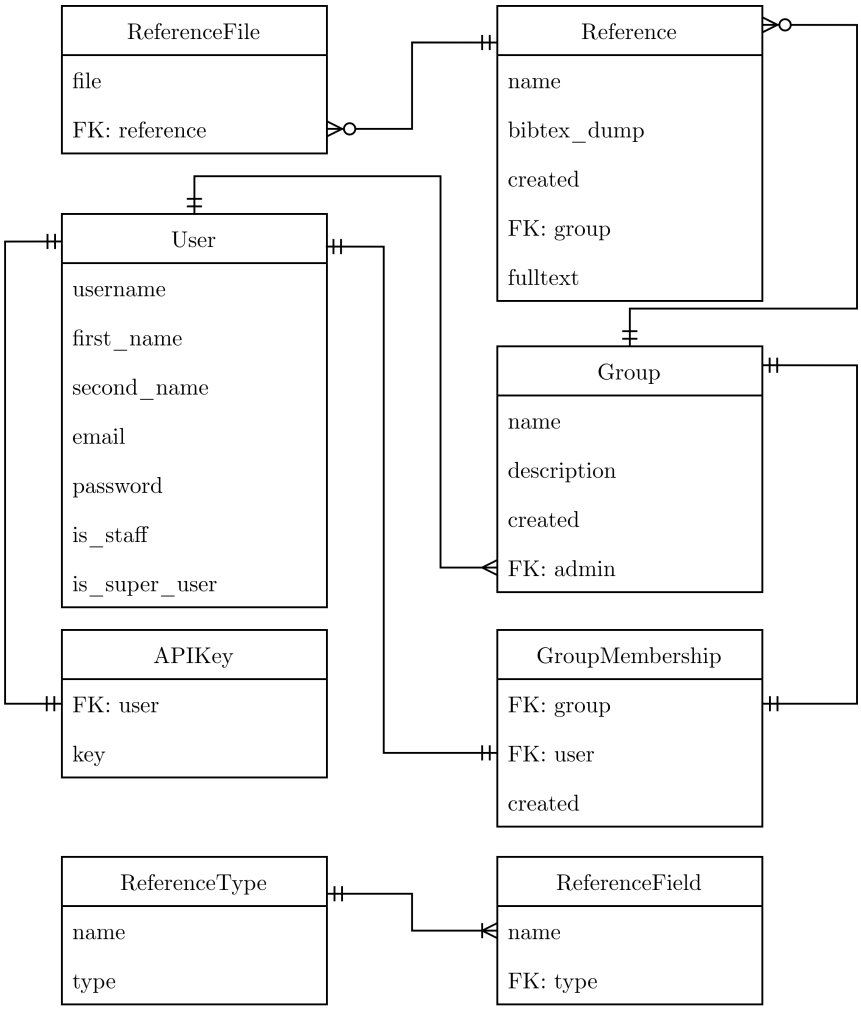

Figure 5: Database Design

citation is present in the PDF. Examples of these are the DOI, pages in the journal, and other metadata that is determined after the paper is typeset.

4.1.1 PDF Extraction

The majority of this process is performed by the GROBID service. As shown in Figure 4, The process starts when the user uploads the PDF to the server by a POST request. At his point, the server performs two tasks. It saves the PDF according to the file handling method specified in `settings.py`. This configuration option allows for storage formats such as file system storage and S3 storage to be easily interchanged depending on the deployment situation. The second action that is performed is that the PDF is sent to the GROBID service. The GROBID service runs in a separate Docker container to the main web server. The GROBID container and the webserver are on a private, internal virtual network so that they can communicate securely. The GROBID service exposes an HTTP API to the webserver. The server POSTs the PDF to GROBID for data extraction. Specified in the POST request is that GROBID should return what metadata it can extract as well as the full-text of the PDF. The full-text is required for the search functionality.

If no error occurred, then GROBID will return the metadata as well as the full-text. The web server will save the full-text to the `fulltext` field of the `Reference` table in the database (see Figure 5).

4.1.2 PDF Metadata Consolidation

Once the server has received the metadata, it can begin to fill out the missing fields of the reference as well as check that a mistake was not made by GROBID when extraction the metadata.

The first step to this process is to query online academic paper repositories to determine the DOI or any other metadata. The first online source that gets queried is Crossref. Crossref is queried with the retrieved title of the paper as well as the first author. Only the first author is used as often, GROBID returns other names found on the cover of the paper such as anyone the paper thanks. The returned papers are iterated through. For each of the papers, it is checked as to whether the titles are similar. This is calculated by the use of `difflib` in the Python standard library. If the two titles have a similarity ratio of greater than 0.9, it is deemed that the titles matched. The similarity ratio is required as often, papers had marginally different titles on the PDF copy to those found on Crossref such as apostrophes. Once a result is determined, the DOI is retrieved. Using the DOI, the BibTeX can be found using the DOI.org DOI resolved [17]

If Crossref does not return any valid results, arXiv is queried. Crossref is queried before arXiv as is it significantly faster to respond as well as often having more complete citation data. A similar approach is applied to the results of arXiv as to those from Crossref. The method differs, however, when no DOI is found. Whereas results on Crossref are guaranteed to have a DOI, those from arXiv are not. If a matching paper is found but does not have a DOI, then a citation is generated from what metadata is returned.

In order for the database to be able to store these references in a way that can be effectively searched, the references returned from the aforementioned methods must be converted to a Python object so Django's ORM can write the values to JSONB in the database.

4.1.3 URL Metadata Extraction

The first stage of loading metadata from a URL is to load the web page. Many publishers forbid bots from accessing their content, however, as this is a single page load that must be actioned by a user, it is not a bot. In order to stop the publishers' bot prevention mechanisms, the user agent of Google Chrome is used for the request. Although when given the direct link to the page, the page is able to be loaded by sending a single request, when a redirect link provided by the publisher is used, a JavaScript redirect is used to prevent scraping. This is only an issue with testing as this is the only situation that commonly uses the redirect link provided by DOI.org instead of the direct link. In order to circumvent this, the JavaScript must be run to successfully redirect. Selenium and a headless version of Google Chrome are used to run this JavaScript. This, however, only occurs during testing as there is a large overhead associated with this and is unnecessary the majority of the time.

Once the HTML from the page has been fully retrieved, three methods are used to try and extract the data. The first is to test whether the page uses the Dublin Core Element Set [5] in the header. This is a known format of metadata tags that can be easily parsed. It is checked whether any tags start with `dc:`, which is the prefix used by Dublin Core. If this format is used, the DOI is located. Once located, it can be passed to the DOI.org resolver and the full reference retrieved.

If the Dublin Core Element Set is not used, the second method is to scan for a DOI in the page. An assumption is made that the first DOI found is that of the page as any references made would likely be made after the principle paper. To locate a DOI on the page, the following regular expression is used [18]:

```
10.d{4,9}/[-._()/:A-Z0-9]+
```

Finally, if no DOI is found, it is assumed that the paper does not have a DOI. In this case, an attempt to read any data from the meta tags is made. For each usual reference field, it is checked whether there is a corresponding meta tag. For example, is a meta tag has "*author*" as its name, the contents of the tag are written to the "*author*" field of the reference.

If none of the aforementioned methods work, then an error will be thrown and the user notified that there was an issue with the URL provided.

4.2 Server Back-end

The back-end of the project is implemented in Python using the Django framework. The code is split into two *apps* called *syncref* and *references*. Syncref is the working title for the application. The app contains the configuration of the project as well as the code to interface with the ASGI server Uvicorn. The majority of the code resides inside of the references app. Amongst the contents are the templates for the dynamic views, the static files required by the front-end, the code required for the metadata extraction, the admin interface for the project, the forms required by the project, the models for the database (see Section 4.5), a router for the application, and the dynamic views.

4.2.1 Admin Console

The admin console is generated by Django from the models you specify in `admin.py`. Once a model is specified, along with and configuration options, all instances of the model in the database can be edited. The admin interface is important for two functions in this project which cannot be done from the website. The first of these is approving users when they sign up. When a user registers, their account is disabled by default and needs to be activated. This allows the administrator control over who has an account if the website is accessible to the internet. The second function is the configuration of the field populator for the manual entry of references. When a user manually enters a reference, they can choose from a variety of standard reference types such as *article*, and the fields are auto-populated with fields standard to that reference type.

As well as these two functions, the admin console can be used for general user management, management of references, and files. Although most of this is possible from the website, the admin console allows these actions to be applied to multiple entities for bulk actions.

4.2.2 Forms

Django forms were not extensively used in this project in favour of manually POSTing data, however, to handle file upload, it is simpler to create a form. This is because the Django file form converts the byte stream into a filehandle which simplified the handling of files.

4.2.3 Router

There are two routers present in this project. The main router is in the *syncref* app. This is the first router that runs when a request is made to the server. This router is responsible for directing the request to one of two other routers: the admin router as part of the standard Django module, and the references router that handles routing for the rest of the application.

This router takes a request and maps the path specified to a view. During this mapping, it can take arguments from the path and pass those to the view as well.

4.2.4 Views

The initial view served to the client when they first load the website is the index view. This checks to see whether they are logged in. If they are, they are redirected to the main app page. If they are not logged in, then the website landing

page is served instead.

The sign-up view takes a POST request from the client and creates an account if the fields of the request are valid. Once an account is created, the user is notified that they must wait for an admin to activate the account if the `REQUIRE_ADMIN_AUTH` flag is set in the settings file.

The login view, similarly, takes a POST request and attempts to log the user in. It checks to see if the `is_active` field for the user is set to true. This field of false by default but must be set to true by an admin.

The next view of interest is the view to manually add a reference. It iterates over every key and value in the POST request. From this dictionary, a reference in the database is created with this information. Once the reference has been created, the user is redirected to the new reference. The edit reference view follows a similar structure. It differs, however, as when the key-values are submitted, the corresponding reference is updated to match the new values, as opposed to creating a new reference.

The next view is the PDF upload and parser handler. The user submits a form containing the PDF to the server. Using the aforementioned form, a file pointer is passed to the metadata extraction function. This function calls GROBID with the PDF and returns any extracted metadata as well as the full-text of the PDF. The metadata is then consolidated. This consolidated metadata is stored as a reference in the database. The PDF is also saved and linked to the reference. Once the PDF is parsed and saved, the user is redirected to the page of the reference. This view is similar to the `uploadPDFToReference` view. This view allows the user to upload a PDF but doesn't parse it. This is for the purpose of adding PDFs to existing references.

A similar view to the PDF view is the URL view. This allows the user to create a reference from a submitted URL. The view accepts a POST request. It then attempts to extract the metadata from the URL. If the extraction fails, then the user is notified. If it succeeds, then a new reference is created with the metadata and the user redirected to it. The next view is almost identical. It allows the user to submit a URL but is CSRF token exempt. CSRF tokens are used throughout the site to ensure that requests are made from the same origin as the tokens are only available from the server. This view is exempt from them as it is intended for use from the browser extension which does not have access to CSRF tokens. Instead of a CSRF token, this view uses an API key which can be accessed by the user from another view. This API key is randomly generated using the cryptographically secure pseudo-random number generator included in the Python standard library.

The export view allows the user to export a BibTeX file for a given group. The export process ensures that there are no references with the same identifier in the same BibTeX file as this would cause issues further on. The upload BibTeX view allows the user to upload a BibTeX database to a given group. This is significantly faster than adding references from their sources such as PDFs as the metadata is already structured and therefore does not need to extracted.

4.3 Front-end

The front-end is divided up into templates for each view. Each template inherits from the base template which means elements such as the navbar only need to be edited in one location for the change to reflect in all of the views.

When the user first loads the website, they are presented with the landing page as can be seen in Figure 6. From here, they can either log in or register. If they choose to register, they are presented with the create account page as can be seen in Figure 7. Each input control states the requirements that must be met by that element for

the form to be valid. If the user submits the form with any invalid data, then they will be notified at the top of the page along with the incorrect fields being highlighted. Upon successful submission, the user will be notified that they must wait for an administrator to verify their account and then be redirected to the home page. If the user chose to sign up, they will be directed to the login page. As can be seen in Figure 8, the standard login format of username and password is used. If there is an issue with credentials, the user will be notified of the error. If the credentials are correct, then the user will be redirected to the main page of the app.

As can be seen in Figure 9, the main page of the application shows the user a variety of data. On the left-hand column, the list of groups the user is a member of is shown. Each of these groups can either have just one member or if the user wishes to collaborate, multiple members. In the main column, all of the references from all of the user's groups are shown. If the user wishes to view any one of these references, they can click on it. As with the references, if the user wants to view any of the groups they are a member of, then they can click on the group's name.

When the user clicks on a group name, they are shown a list of all of the references that are stored in that group (see Figure 10). This is similar to the home view except for a few differences. Firstly, controls have appeared at the top of the list. These allow the user to either add a new reference to the group or to export all of the references from the group. If the user chooses to export the references, they will be presented with the browser download dialogue for a BibTeX file containing all of the references in that group as can be seen in Figure 11. As well as the controls, the breadcrumb reflects that the user is now viewing a group as well as which group they are viewing. Finally, there are new controls at the bottom of the list to edit the group as can be seen in Figure 12.

When a user wishes to view a specific reference, they are shown the view shown in Figure 13. They are shown the plain text data stored in the reference as well as the BibTeX its self. In the BibTeX code block, there is a button to allow the user to copy the contents to the clipboard. Below the BibTeX is the PDF if available. As can be seen in Figure 14, the browser's built-in PDF is used to show the corresponding PDF. The *Edit Citation* tabs allow the user to edit the reference. This tab is shown in Figure 15. Each of the fields is shown allowing the user to edit either the key or value. The user may also choose to delete fields or add more. As can be seen in Figure 16, the user may also choose to upload a PDF if one does not exist or if they wish to add additional PDFs. They may also delete the PDF.

In order to upload a reference, the user has three options. The first is to upload a PDF (as seen in Figure 17). Once uploaded, if successful, they are redirected to the reference with a notification saying it was successful (see Figure 18). If the metadata was extracted but no published paper found, the extracted metadata will be shown but a warning will be displayed.

The second option is to specify a URL as seen in Figure 19. If the URL is parsed successfully, then the user is redirected to the reference. If there was an error, the error is shown as in Figure 20.

The final option is to manually enter the fields as seen in Figure 21. The user can autofill the form from templates specified by the admin as in Figure 22.

The admin console is generated by Django from the models specified in `admin.py`. The main view for the admin console can be seen in Figure 23. Each of the models are shown. If the user wishes to edit any of the values stored in any of the models, they can choose the model and are shown a view with editing options as seen in Figure 24.

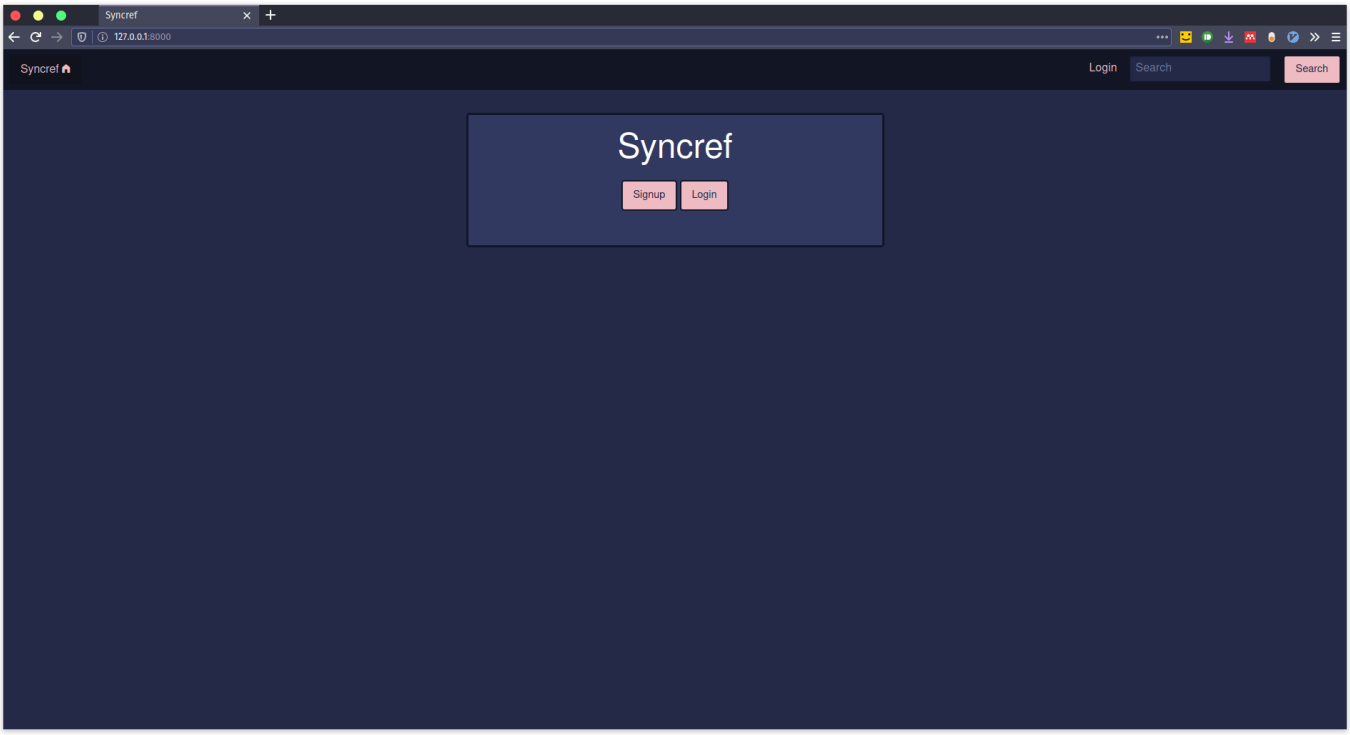

Figure 6: Landing page

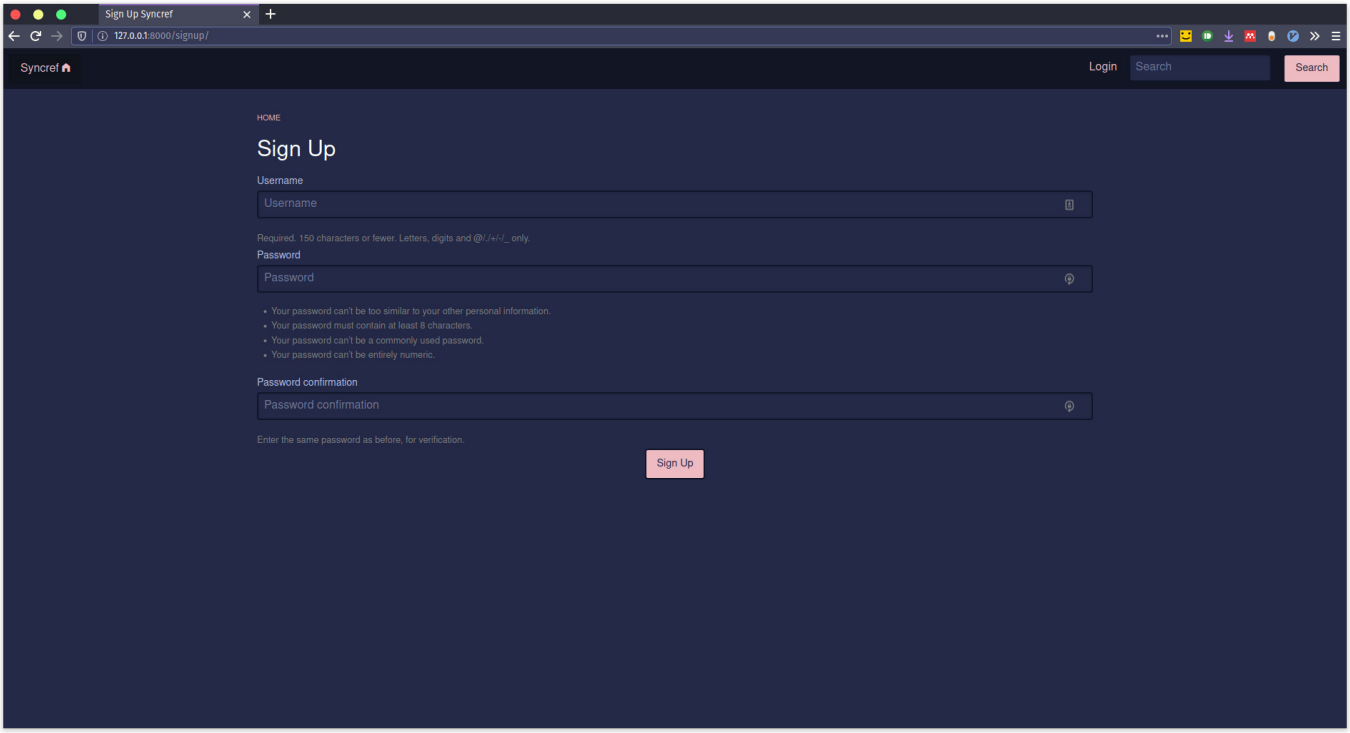

Figure 7: Account creation page

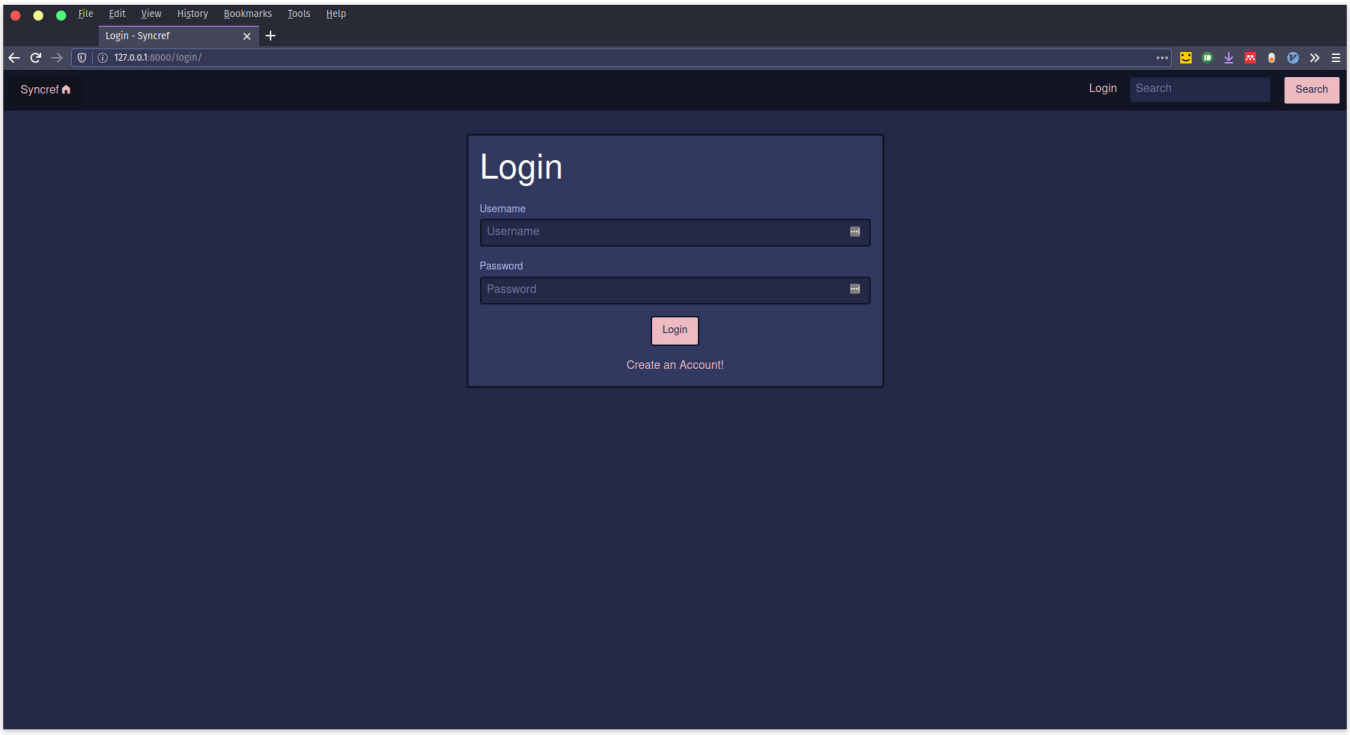

Figure 8: Login

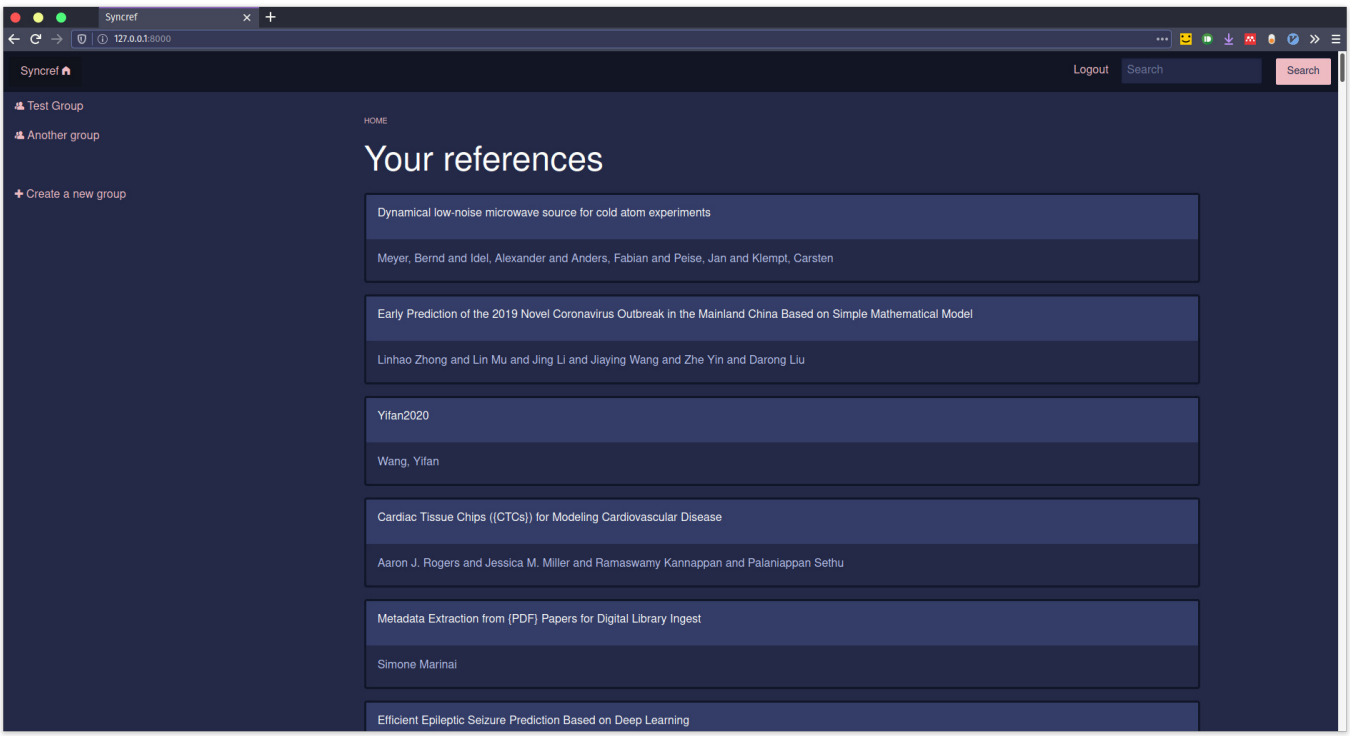

Figure 9: Application main page

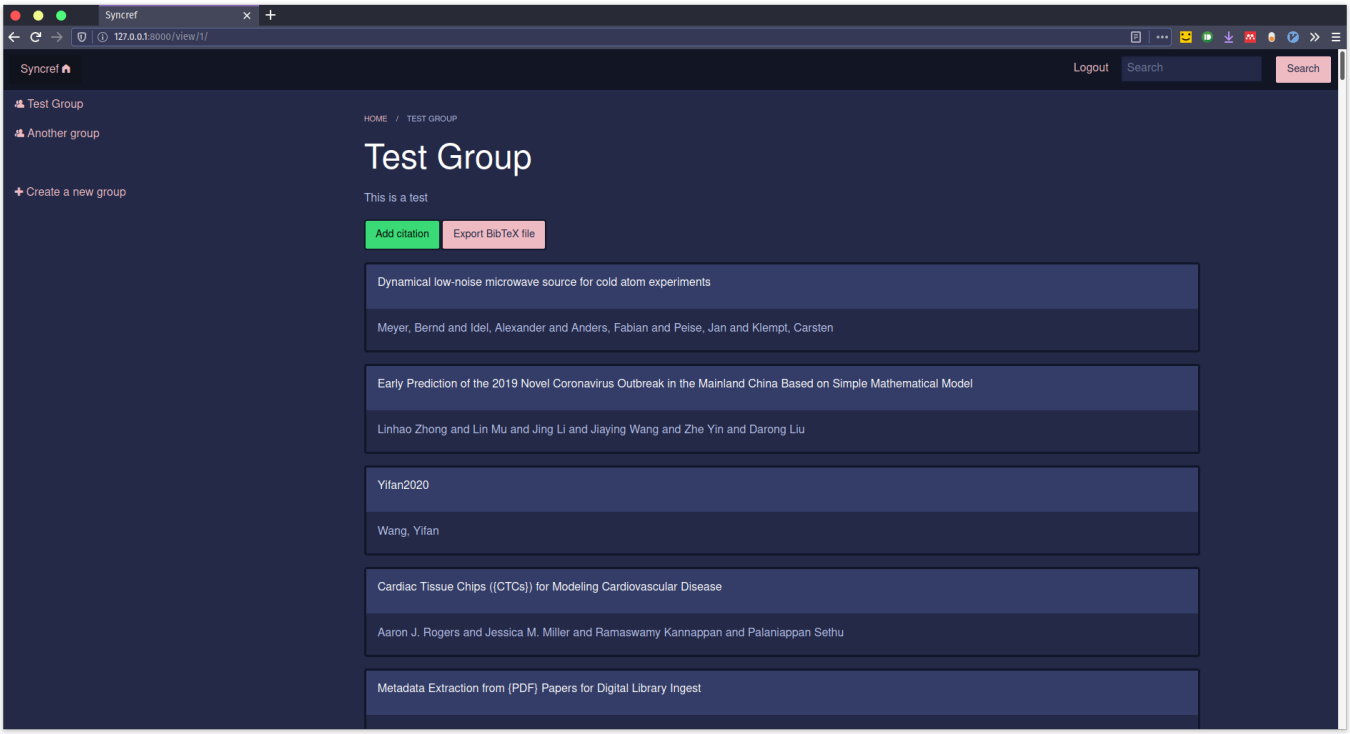

Figure 10: Group view

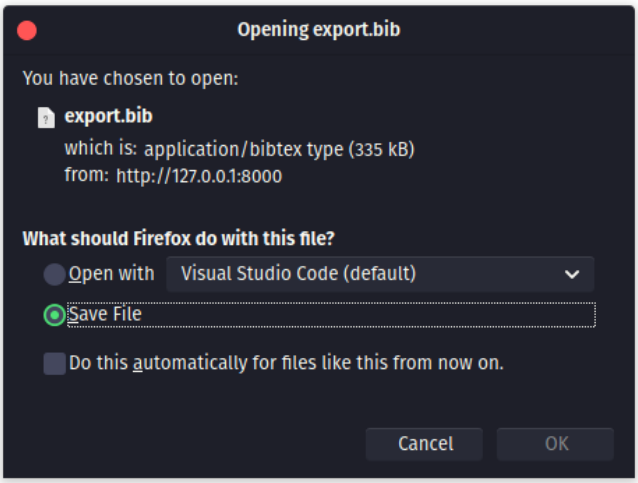

Figure 11: BibTeX file download dialogue

Figure 12: Group actions

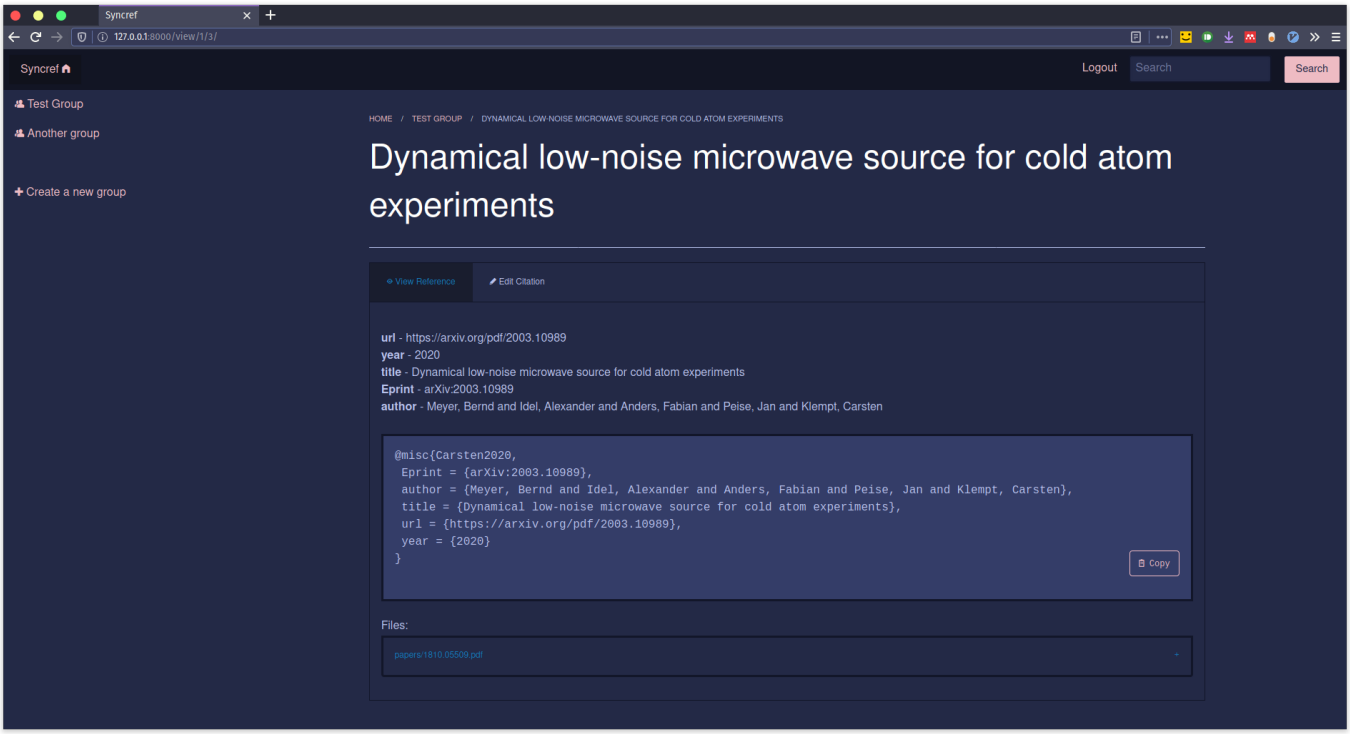

Figure 13: Reference view

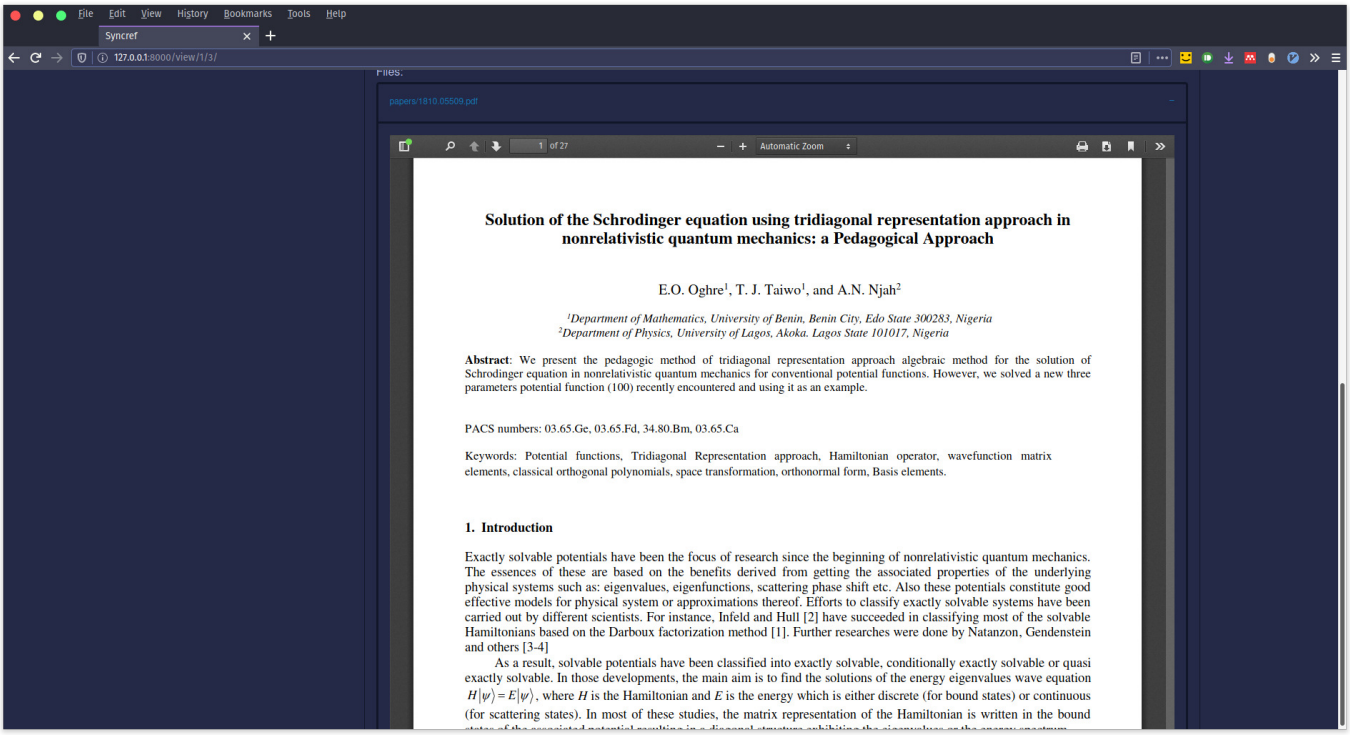

Figure 14: PDF viewer

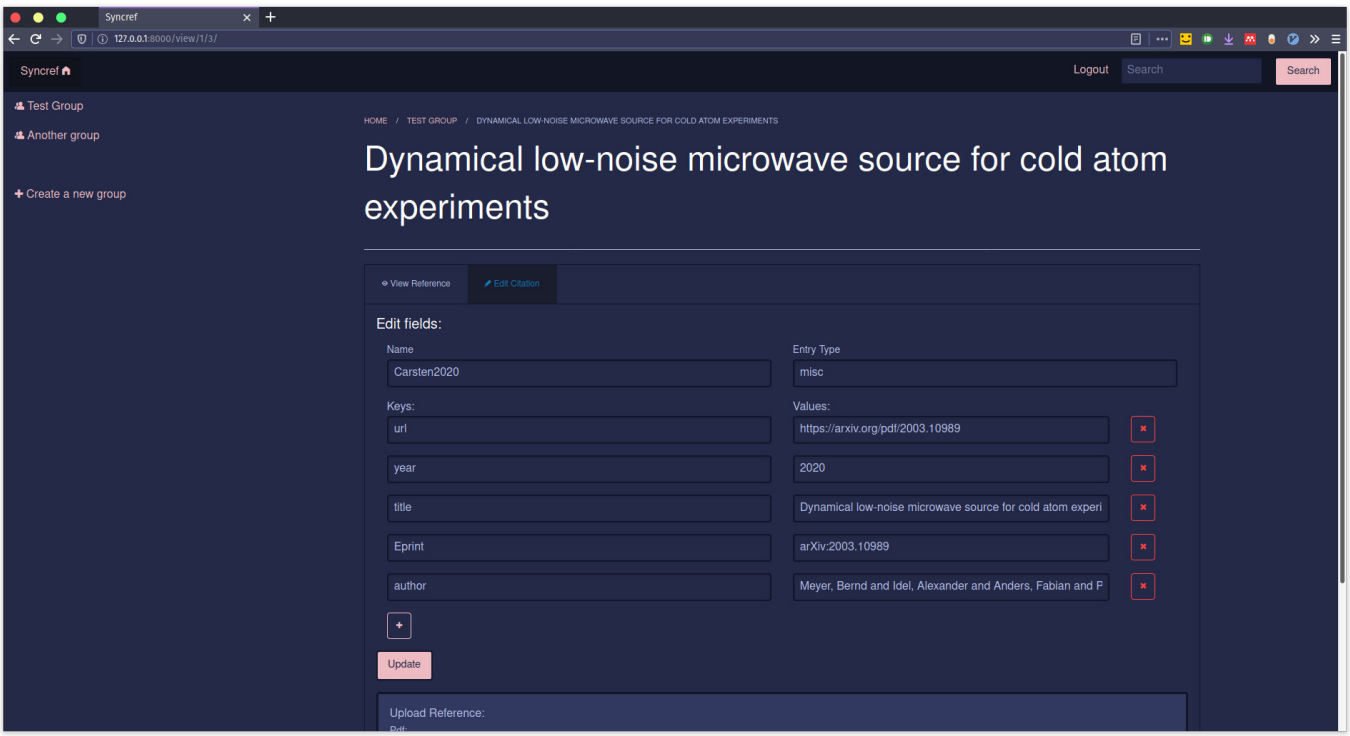

Figure 15: Editing a reference – 1

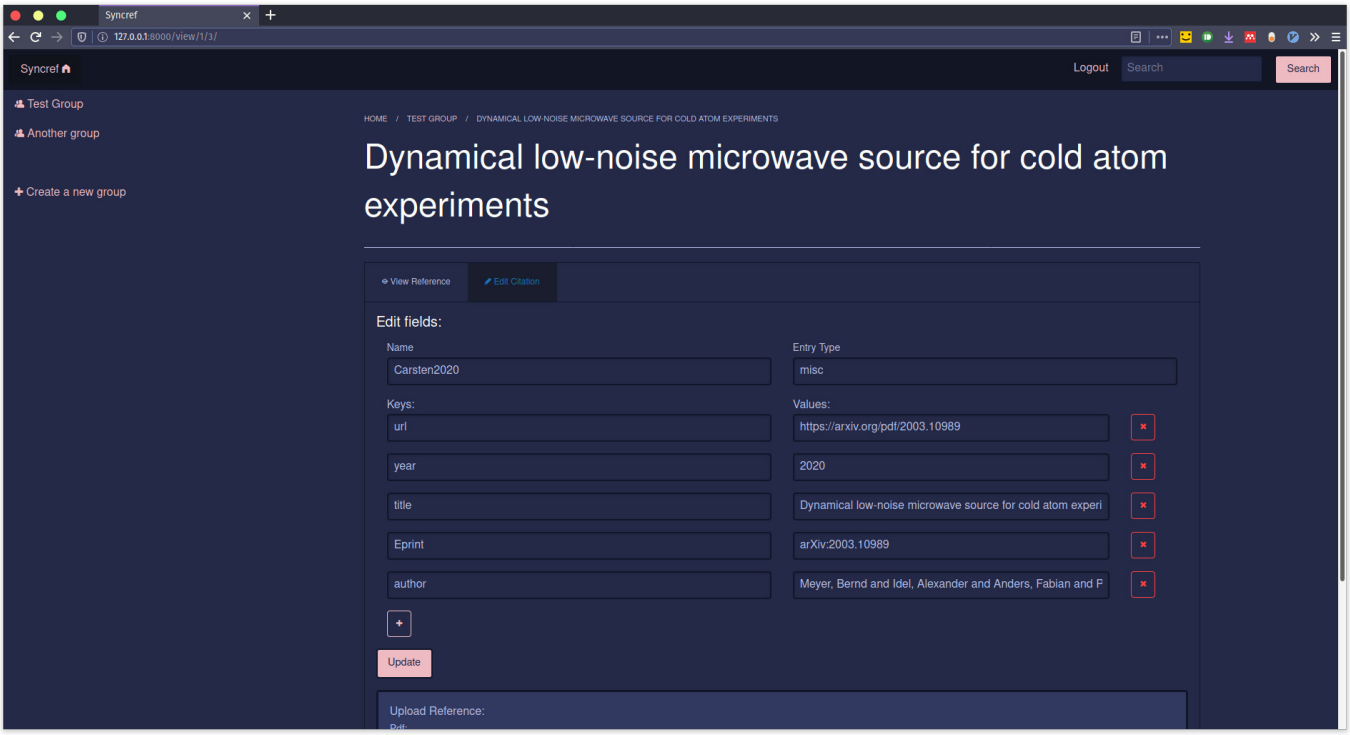

Figure 16: Editing a reference – 2

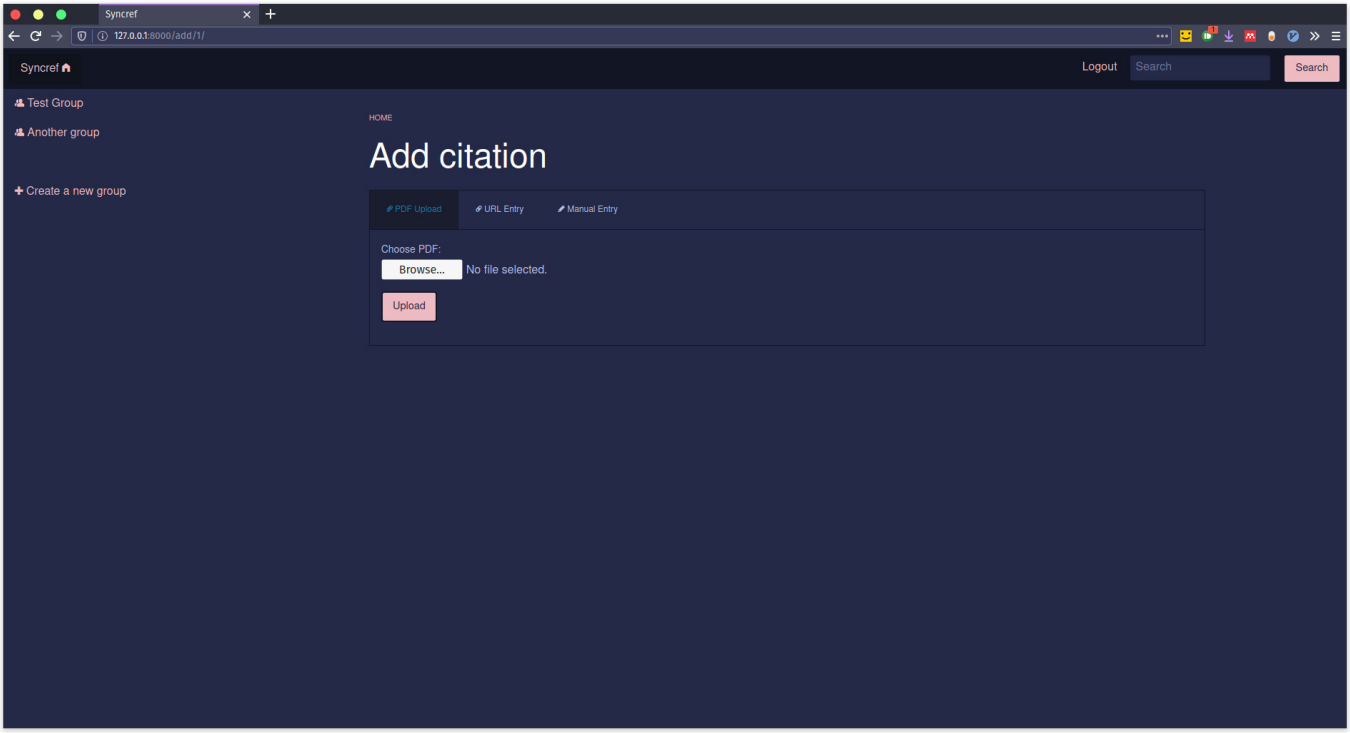

Figure 17: PDF upload

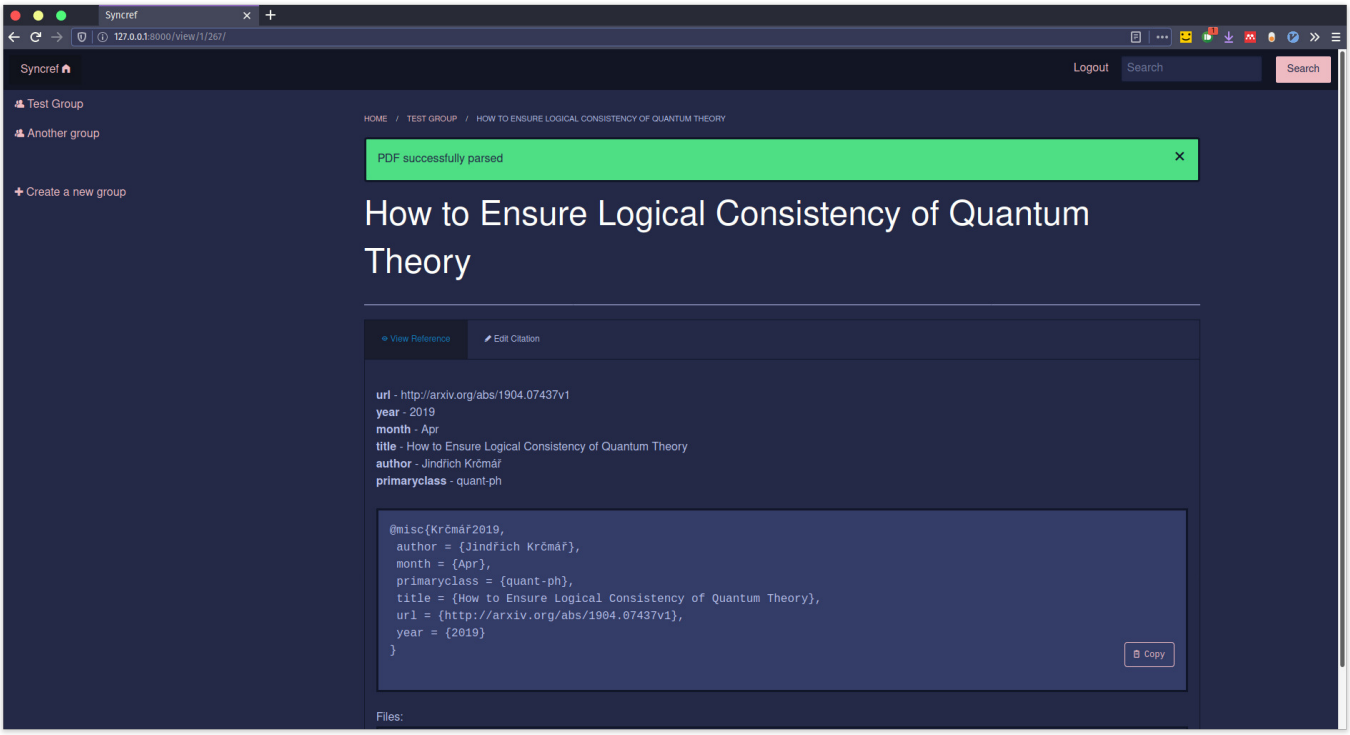

Figure 18: PDF upload success

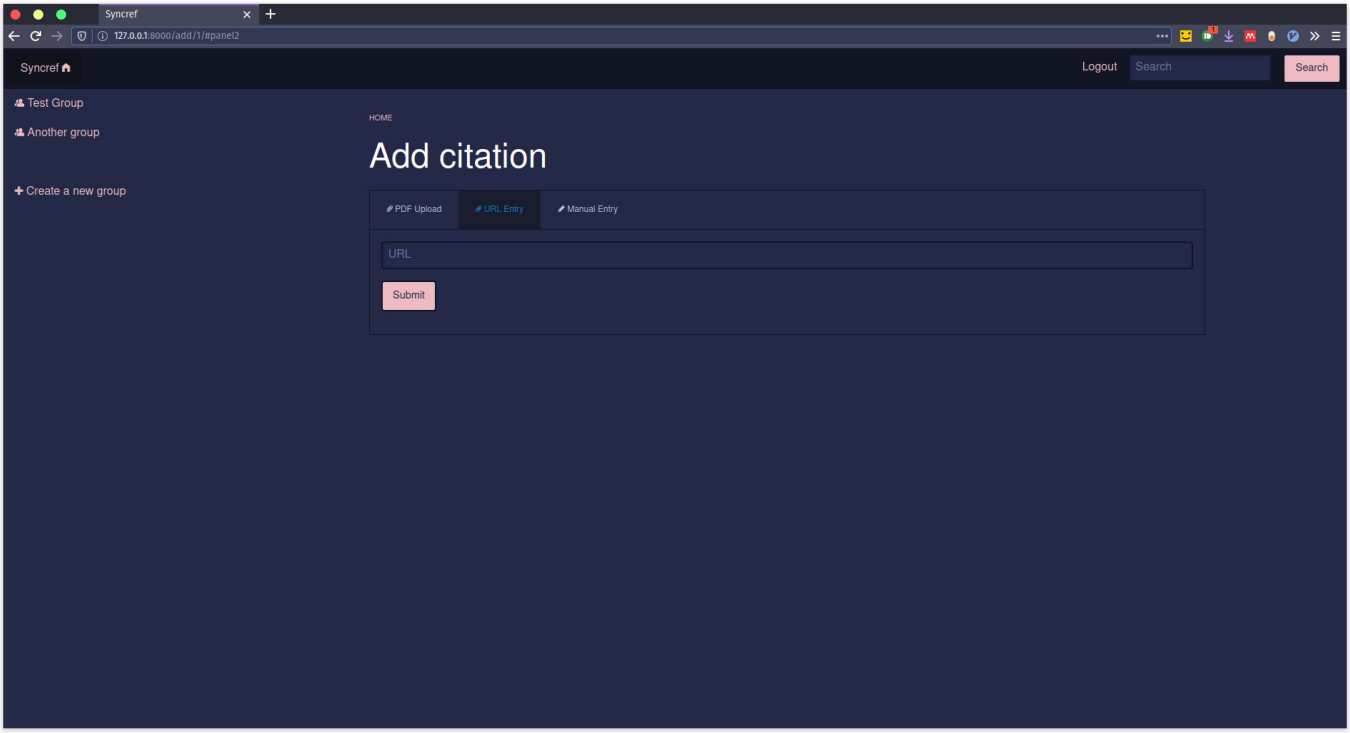

Figure 19: URL upload

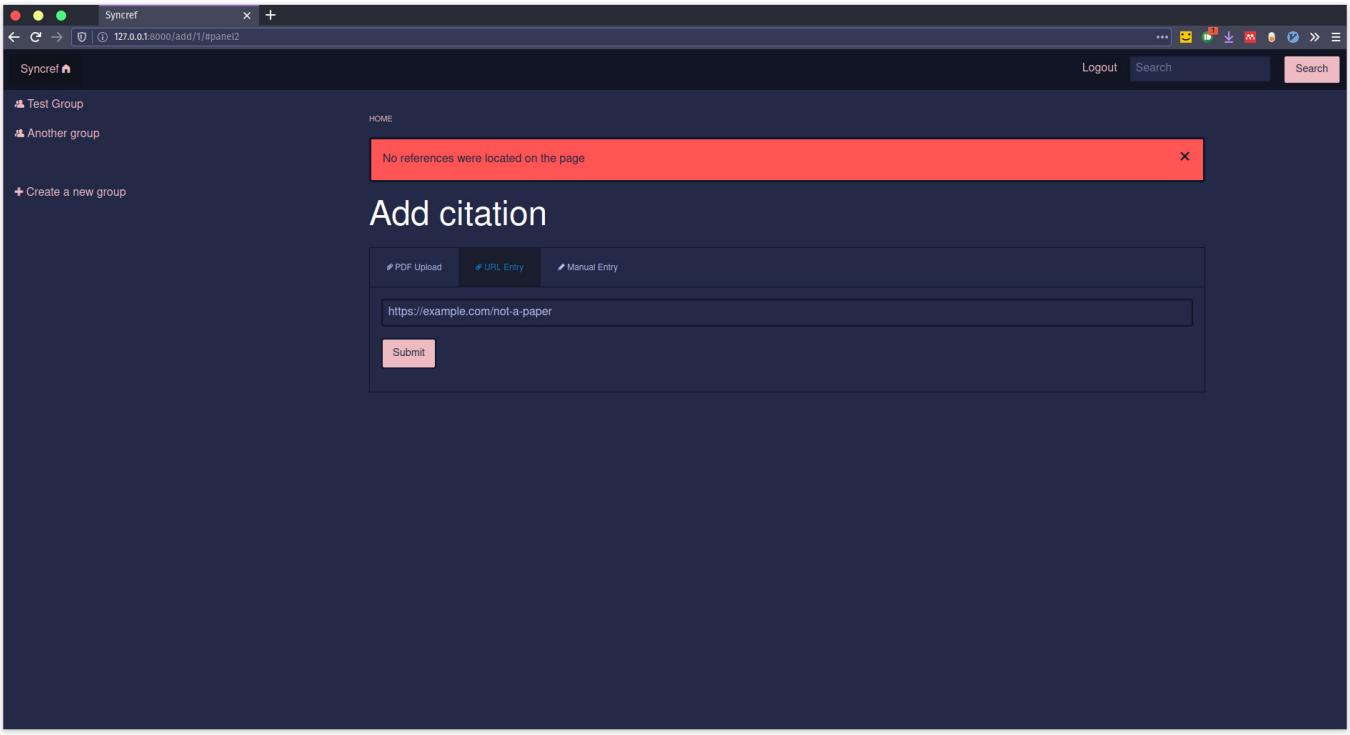

Figure 20: URL upload error

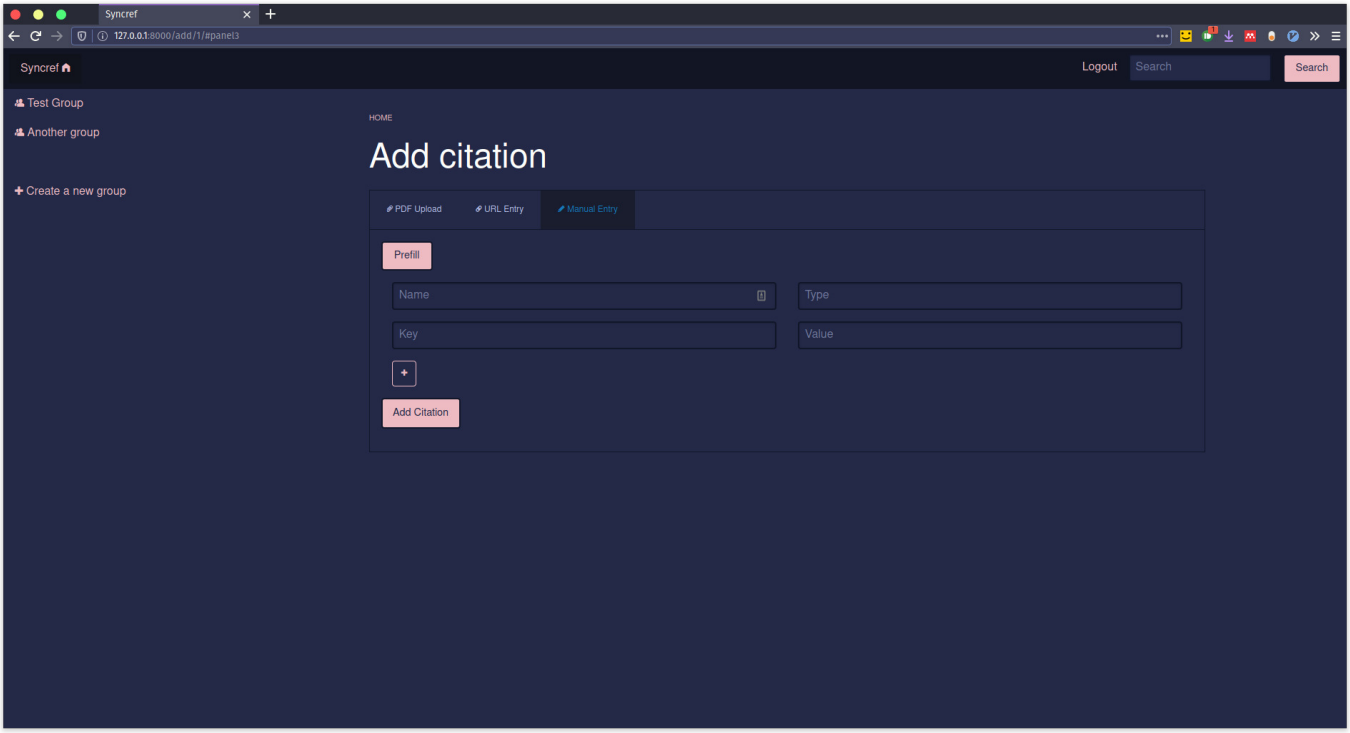

Figure 21: Manual upload

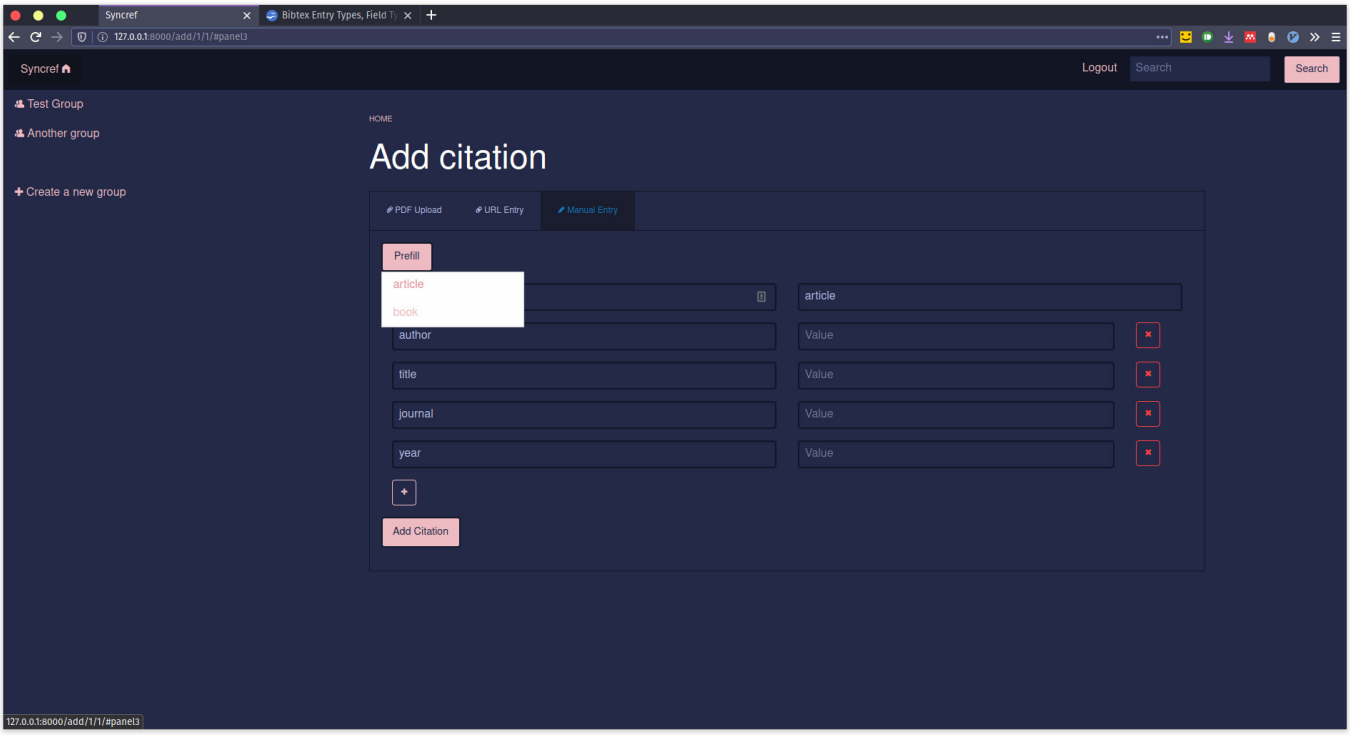

Figure 22: Autofilling manual upload fields

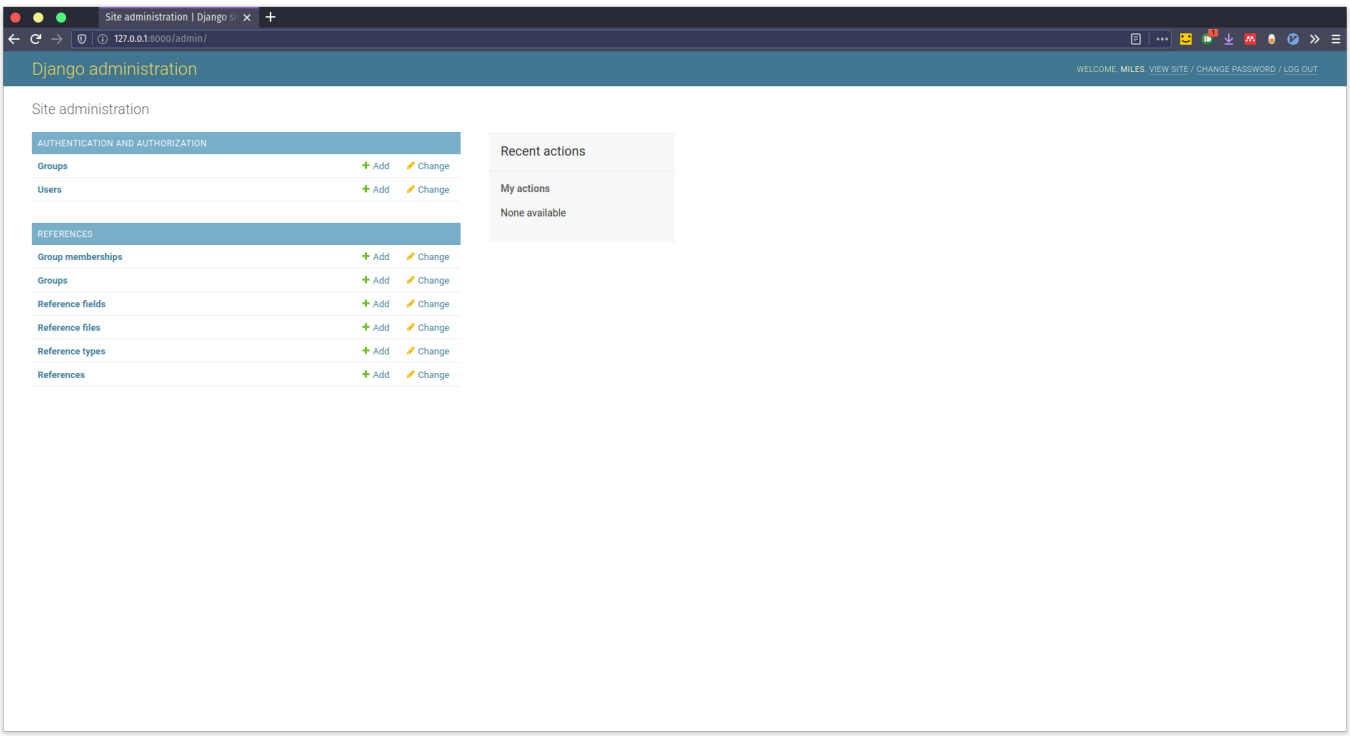

Figure 23: The admin console

4.4 Browser Extension

In order to reduce development time whilst maximising the available users, it was important for the extension to compile for multiple browsers. I found an existing build system to base the extension off that allows Chrome, Firefox, and Opera build targets [19]. The extension is split into four sections: the popup, the options page, the content script, and the background worker. The popup is the small window that appears when the user clicks the extension button at the top of the browser window. This popup provides the main controls of the extension as can be seen in Figure 25. The popup shows the user the page they are going to submit to the server as well as the button to submit it. A link to the options page can also be seen. If the user tries to submit the URL without specifying the root location of the syncref server and an API key, they will be notified. If the page doesn't contain any references, then the extension will display an error accordingly.

The options page contains two text fields: the syncref base URL and an API key (see Figure 26). These ensure that the extension is communicating with the correct instance of syncref and for the correct user. Due to the scope of the project, it was simpler to use API keys as opposed to a more conventional authentication method such as OAuth. Once the user has entered these, the extension will no longer error when attempting to submit a URL.

The popup and options pages are written using HTML, CSS, and JS and behaves live a web page. The third section is the content script. This is a piece of JavaScript that is injected into the page to extract the URL and title. The extension can't access information such as this so it needs to inject JS in order to access it. The injected JS communicates with the main extension by the message passing interface that is standard across browsers.

The final part is the background worker. This is used to send the request to the server as it can't be cancelled partway by the user. A message is passed from the popup to the worker with the information to send to the server. The worker attempts to send this to the server. Upon success or error, the worker passes a status message back to the popup to display to the user.

4.5 Database

Django interfaces with PostgreSQL through an ORM. The models are defined as classes in `models.py`. The models correspond to the tables defined in Figure 5. Each model is a class that extends Django's `django.db.models.Model`. Django's User model is also imported. This allows the models to use the user model used for authentication and session management.

The first model is `Group`. This represents a reference group. It contains a name, a description, a creation date, and an admin. The creation date is automatically populated when an object is created from the model. The admin field is a foreign key to the user table. The admin user has special privileges over the group.

The next model is the `GroupMembership` model. This is used to create a many-to-many relationship between users and groups. Although Django's ORM supports many-to-many, this is a more verbose way of defining the relationship. The model contains two foreign keys: a user, and a group. There is also a date created field that is automatically populated on creation.

The `Reference` model contains data about a specific reference. It contains the name, BibTeX data, a creation date, group it belongs to, and the full-text of the reference. The BibTeX field is a special data type specific to PostgreSQL: JSON field. This allows for the parsed BibTeX data to be stored in an indexable way.

The `ReferenceType` model is used for the autopopulation of fields when the user manually en-

ters a reference. The `ReferenceField` model uses this model as a foreign key. The `ReferenceType` model represents a type of reference such as "*article*" or "*book*". The `ReferenceField` model represents fields associated with those reference types such as "*title*" or "*author*".

The `ReferenceFile` model contains two fields: a file field and a foreign key to a reference. This model is used to store a file that is related to a reference. The Django file field automatically handles the file by the method defined in `settings.py`.

The final model is the `APIKey` model. This contains a foreign key to a user and a string of the API key. The string must be unique as if there was a duplicate, then it couldn't be determined which user it was if a user identified by their API key in the browser extension.

4.5.1 Search

All search functionality in the application is handled by one view. This includes features such as the search suggestions when the user is typing in the search box. The main search view has several advanced controls that change how the search function runs. These controls change flags in the POST request sent to the search view endpoint. The user is shown a list of all of the field names for all of the references they have access to (see Figure 27). This allows the user to quickly select a field they wish to query. They also have the option to choose whether to search the full-text of references and the group the reference is present in as well.

Django uses search vectors to construct searches. These vectors can be assigned weights to determine the weighting they have on the results. In the case of syncref search, full-text results are given a lower weighting than other vectors as it is more likely for any given query to exist in the full-text than in the metadata.

When a query is retrieved, several vectors are created. If the user has requested a full-text search, a vector for the full-text is created. If specific fields are requested by the user, then those fields are used. If none are specified, then the default fields specified by `settings.DEFAULT_SEARCH_TAGS` are used. These fields are sent to a function to return a search vector for each field. All of these vectors are combined into a final search vector. The `SearchRank` method is used to perform the actual query. This returns all of the matches and a similarity score to the original query. These results are ranked by similarity and results that have a very low similarity are filtered out. If the request specified a JSON response, then the results are returned as JSON, otherwise, they are rendered for the user.

4.6 Deployment

Deployment is handled by the Docker-compose configuration file and the project Dockerfile. The Dockerfile specifies steps to build an image of syncref. It installs the dependencies, copies the code to the container, and collects all of the static files and places them under a single directory.

The Docker-compose file defines several services that are run for the project to function correctly. The first service it defines is the Træfik service. The flags on how Træfik behaves are given here. Træfik is configured to accept any traffic on ports 80 and 443. It is also told where to store the Let's Encrypt certificates. The Docker socket (`/var/run/Docker.sock`) is also forwarded to the container. This allows Træfik to discover and control Docker containers running on the host. Træfik is given access to both the internal private network and the public-facing network. Both networks are required as it functions as the reverse proxy and forwards traffic from the public-facing network to the private one.

The next service defined is the syncref service. The location of the Dockerfile is given so Docker-compose can start the service. Docker-compose

runs a shell script contained in the container. This shell script waits until the database has started before starting the application. The directory that the static files were moved to is then mapped to a shared volume by all of the containers. Port 8000 is exposed so Træfik can forward requests to it. The Træfik rules are then defined for the service. options such as HTTP to HTTPS redirection are specified along with how to resolve the SSL certificate from Træfik. Environment variables are then defined. These are used to set up the administrator account when the server starts.

The next service is the Nginx service. This is given access to the static volume made available by syncref. Træfik forwards all requests to `/static/` to this service which can then serve the static files without the Django server serving them.

The penultimate service is the database. This has a secure configuration. It is not available for Træfik, meaning there is no risk of it being exposed to the public-facing network. A volume is defined that is mapped to the PostgreSQL data folder so that the container can be stopped and deleted with no risk of deleting the data.

The final service is the GROBID service. Like the database, this is only accessible on the internal network as the security of the container cannot be guaranteed as it is a community build container.

Finally, the aforementioned Docker-compose volumes and networks are defined so that they can be automatically created when the Docker-compose file is run for the first time.

This configuration allows for the project to be easily deployed to any hardware with now worry about dependencies. Docker-compose files can easily be converted into Kubernetes files for more advanced deployment across multiple hosts should the project require this.

5 Results and Evaluation

5.1 Accuracy of Metadata Extraction

5.1.1 Success criteria

To evaluate the accuracy of the metadata extraction from PDFs, a set of PDFs shall be passed through the metadata extraction system. For each PDF, it is determined if the metadata was successfully extracted by searching Crossref and arXiv for papers matching the extracted data. If a match is found, then it is deemed that the metadata was extracted successfully. If no match is found, it is assumed that the metadata was incorrectly extracted.

The main limitation of this method is that the paper could not exist on either repository. Due to the volume of papers stored, however, makes this unlikely. Unfortunately, there is not a better method within the scope of this project.

5.1.2 Data set

The PDFs being parsed is a set of 331 PDFs of scientific papers, the majority of which are from popular publications.

5.1.3 Results

After parsing the 331 PDFs, there was a success rate of 84.2% Although this is a satisfactory percentage, I looked into why the papers that failed did fail. The feature of a paper that had the greatest effect on the parsing outcome was whether the paper was a scan. Figure 28 shows a sample of text from a scanned PDF. Although OCR can usually recognise text like this, there are anomalies. The other most common causes of mal-parsing are unusual layouts, especially when there are other names on the first page, ear the top, that aren't authors.

5.2 URL extraction success rate

5.2.1 Success criteria

To evaluate the success rate of the URL data extraction, a URL is deemed successfully scraped if some metadata can be returned. Many URLs do not contain enough metadata to full identify the paper. In these cases, if some metadata such as the author and title are returned, then it is deemed a success. This compromise had to be made as many of the URLs present in the data set are not for papers.

5.2.2 Data set

The data set for this was a BibTeX database containing 435 references with URLs. The URLs are a variety of direct links, DOI.org links, and links to PDFs.

5.2.3 Results

After attempting to load all 435 URLs, there was a success rate of 76.6%. This is an acceptable value as the majority of failures are not due to not being able to parse a page containing a paper. The most common error was the `url` field of the reference was pointing to a PDF, and not the web page of the paper. The URL parser is not designed to handle PDFs; This role is reserved for the PDF upload feature. Another common problem was dead links or links that returned a 404 error. Although these links should be discounted, there is not way within the scope of the project to do that for a data set of this size. The final issue that many of the links were to websites that were not papers. The URL parser assumes that the user wishes to add a paper and is therefore not reliable at extracting metadata from web-pages that are not papers.

5.3 User Testing

In order to test the user experience of the project, I used the project (once in a usable state) to manage the references for this project. This allowed me to identify usability issues and useful features that were not specified in the requirements. Examples of these are the breadcrumbs at the top of every page. Although I did not initially include them in the design, the similarity between the home page and a group view, and the lack of feedback about which group is being viewed showed that there was a requirement for them. Another example is the copy to clipboard button by every code snippet. Although it is not required and was not in the original requirements, my workflow for writing this report caused me to copy references often, which got repetitive. The button was a small snippet of code which saved me time in the long run. Insights such as these are only available when the user is relying on the software as then they find the features or lack thereof that become repetitive for the user.

The other advantage to testing it myself is a very quick development cycle. If a bug was identified, I could fix it right away and continue testing and using the software.

5.4 Requirements

Of all of the requirements specified in Section 3.1, only one was not met, and this was only categorised as *could have*. This was to be able to annotate PDFs. After research into this area, there were no viable open-source methods to do this in a browser. Implementing this would have been possible but would have cut into the development time of other more important features.

Although the functional requirements were simple to identify whether they were fulfilled, the non-functional requirements are more subjective. Below are the requirements and my justification as to whether they were met:

- *When running in production, the system must be available 99% of the time.*

  Although this requirement is specific to long-term deployment which has not been tested, tools required to achieve this percentage of SLA have been integrated into the project. These include Docker which would allow multiple instances of the project to run in a cluster as well as Træfik which allows for health to be monitored.

- *The interface must be pleasant to view.*

  Although this dis subjective, the end user can easily edit the colour scheme of the final product by editing the `variables.scss` file. When compiled, the colour variables specified in this file as well as other stylistic choices will be applied to the project. Therefore, whatever the user defines as pleasant to view can be configured.

- *The interface must be intuitive to use.*

  This is another subjective requirement. The project had constant user testing throughout its development and as a result the interface is relatively streamlined. Further user testing would have to be carried out to verify this, but the project has been tested for usability and therefore it can be assumed that it is intuitive to use.

- *Each page must load in no more than 1500ms (excluding loading of the first page).*

  In testing, the average page load was 500ms. The loading of the core page was significantly faster, however, as the slowest part to load was the MathJax library used to display the LaTeX commands present in some references.

- *The application must be able to be accessed securely.*

  The use of Træfik to automatically negotiate SSL certificates means that all communication between the client and server is secure. Internal services that should not be exposed to the internet such as the database are also on separate networks to prevent external access.

- *The application must be able to be viewed in all modern browsers.*

  The front-end library used has been tested on and is supported by all modern browsers. The limitation of browsers comes from the use of flexbox in the project, meaning that there is no support for IE 9, however, this is not considered a modern browser.

- *The user should be aware of the state of the system upon error.*

  Django messages are used throughout so that upon error, or success, the user is presented with a notification box at the top of the page that notifies them of any system information that is relevant to them, such as a PDF failing to parse.

5.5 Development Retrospective

5.5.1 Positives

The choice to develop using a Docker based environment was greatly advantageous to the project. It reduced overheads associated with configuration throughout the project. For example, the two main environments this project was developed in was a Debian based environment and an Arch based environment. Although these are both Linux based, they have different package systems and repositories. Once Docker was installed on both, the build process was identical. It also prevented pollution of the host with tools and packages specific to the project. Developing in Docker also has the advantage that deploying using Docker is much simpler as all of the Dockerfiles are already written.

The use of Django as a back-end framework saved time in multiple areas. Creating and managing the database tables was made significantly easier thanks to the Django ORM. The ORM also comes with several predefined tables such as the user table which saves time implementing basic features such as authentication. Django also handled other basic features that would have take up development time to implement otherwise such as sessions, WSGI/ASGI, the admin console, and routing.

Along with Docker, Træfik was extremely helpful with configuring the deployment of the application. The automatic negotiation of Let's Encrypt certificates saved time on a task that can often be time consuming to configure. Træfik also allows an admin to monitor the health of all of the Docker containers which would be useful in production if a high SLA was of importance.

5.5.2 Negatives

Retrospectively, the largest negative during development was my knowledge of front-end technologies. I decided to use JQuery as there wasn't much interactivity required for the front end and I have experience using it. Although it did get the job done, it is a large network overhead that offers few features for managing a data dense front-end. A more suitable framework would have been React. This allows for much greater control over data and components in the DOM. I did not choose React as I did not anticipate the complexity of implementing a field editor for references in just JQuery.

Although Docker removed many development overheads, it complicated debugging. Although VS Code provides tooling to debug Django applications inside of containers, it required more configuration that debugging a Django application outside of a container. I initially started with no debugger which significantly reduced development speed as I was unable to get the Docker based debugger to work. Although this slowed development, Docker did reduce development time in the long run allowing for me to implement more features that were initially in question as to whether there would be time to implement them like the browser extension.

6 Future Work

The possible scope of this project makes it very viable to continue development into the future. There are many more features that can be added as well as solidifying the existing code base. In order to solidify the existing code base, a rewrite of the front-end in a more up to date technology such as the aforementioned React. This would create a more solid base to implement more features onto as the existing code grew organically and is not suited to extension in a modular format.

As well as solidifying the existing code base, more features can be added. For example, parsing of more types of academic site such as PubMed. This would not be too much more work as they are a set format. The parsing of non-academic sites could also be implemented. This would increase the functionality beyond competitors such as Mendeley if accurate parsing could be achieved.

7 Conclusion

In conclusion, this project was a success. There was a gap in the reference management software market for FOSS software that is up to date and feature rich. This project fills this niche and there is little else available that competes in this area. Of the requirements given, all bar one were met and the requirement not met was not a core one. Although I have identified issues with the development process and tooling, the development of the project on the whole was successful and produced not just a working result, but a useful

one.

8 Reflection on Learning

This project was a steep learning curve of project management and self management skills over a long period of time. This was by far the longest project I have undertaken alone and therefore required new skills to manage. The differentiating factor between this project and other projects I have done is that this one had to have a constant level of work throughout due to the scope. With other shorter projects, If there was a period of less progress, the project was of small enough scope that all outstanding work could be completed shortly before the deadline. With this project, if there was a period of reduced work, the volume of features and functionality required meant that it would be very difficult. In order to circumvent this, I tired to adhere roughly to the Gantt chart that was part of the initial plan. Although the time estimated for some sections was not accurate, it helped greatly to ensure that I was not falling behind with where I should have been at that point in time.

Another differentiating factor between this project and others that I have undertaken that required a change in approach was this this project was to a brief. For other projects I have worked on, they have either been of my own choosing and therefore could go I whatever direction I wanted, or they had a rigid outcome. For this project, I had to learn how to approach a much less rigid set of guidelines without getting sidetracked into working against the brief.

If I were to carry out a project like this again, I would change how I structured my time with regards to what had already been achieved. For the first weeks of the project, the work I had scheduled was simple and therefore I completed it quickly. I should have therefore edited my schedule accordingly. Instead, I just waited until my next scheduled piece of work. Although in a perfect world, this approach would work, I failed to acknowledge that outside influences would affect the speed at which I could carry out work such as exams.

As well as time management skills, I have learned from mistakes made with the development process. The first mistake that I made was testing. Although there was constant time I did not implement any unit testing or testing at a higher level. Unit testing would have allowed me to refactor code more easily and safely which would help greatly with future development. User testing would have helped as well as although my small feedback loop of testing in the development phase allowed for rapid change to the user experience, as I was the one building it, I knew how it worked and therefore it was difficult to test intuitiveness as I already knew how it worked.

This project was also a perfect opportunity to learn a new technology/language/framework. I however, picked the safe option of a language and framework I have already used before. Although this was beneficial to the project, I did not learn any new technical skills as the ones I used I already knew. This also had a detrimental effect on the front end as my knowledge or relevant technologies was out of date. If I had of learned a new skill for the front end, then the benefits would have been twofold.

This use of the safe option extended into the use of new frameworks specific to this project. For example, for the metadata extraction from PDFs, I did research into the effectiveness of existing solutions but not how effective my own implementation could have been. Although it would be unlikely that my implementation would have yielded better results, I did not pursue this opportunity and therefore did not find out. This has the two negative effects of firstly, me relying on black-box technologies that I do not fully understand, and secondly that the project could have been better if I had of succeeded on a more accurate imple-

mentation.

Through the completion of this project, I have learned new skills but more importantly, it has shown me which skills I can learn after completing the project.

9 Appendix

The full code-base can be cloned from `git@github.com:pbexe/syncref.git` and `git@github.com:pbexe/syncref-browser-extension.git` .

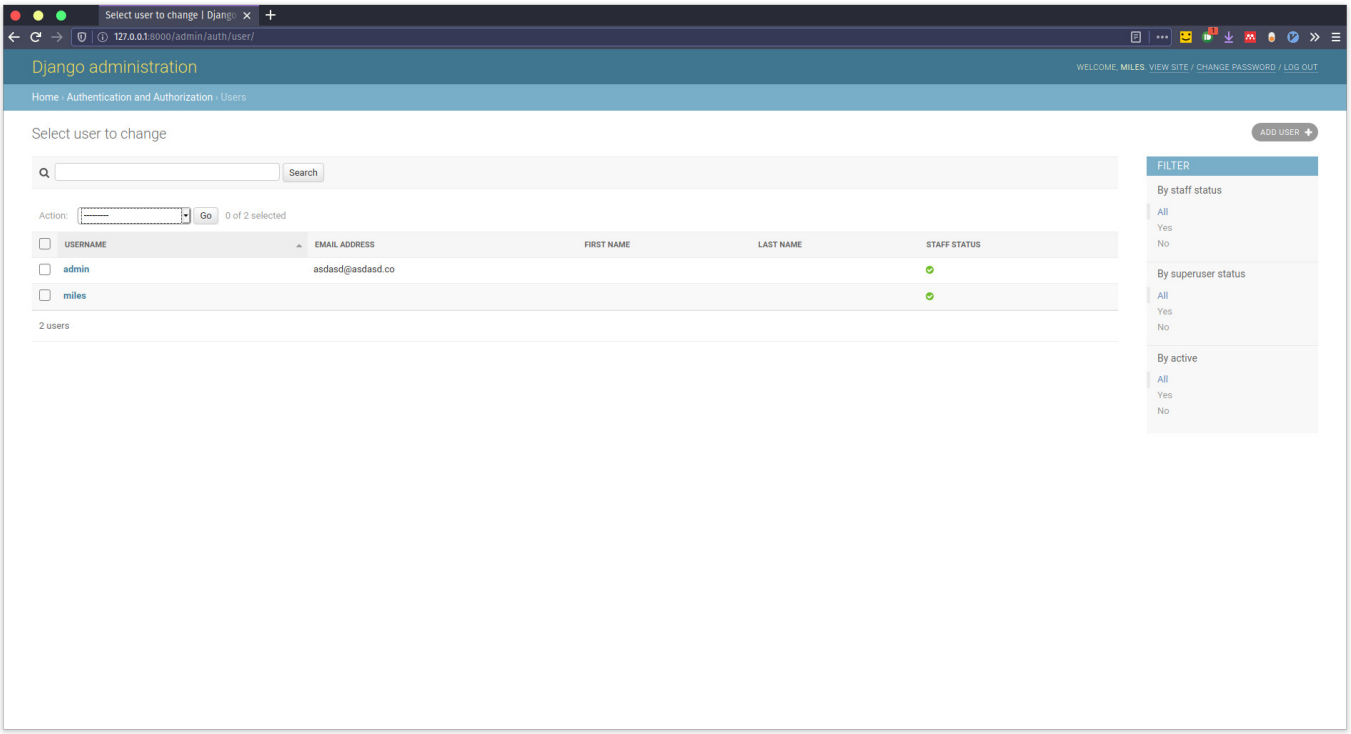

Figure 24: Editing fields in the admin console

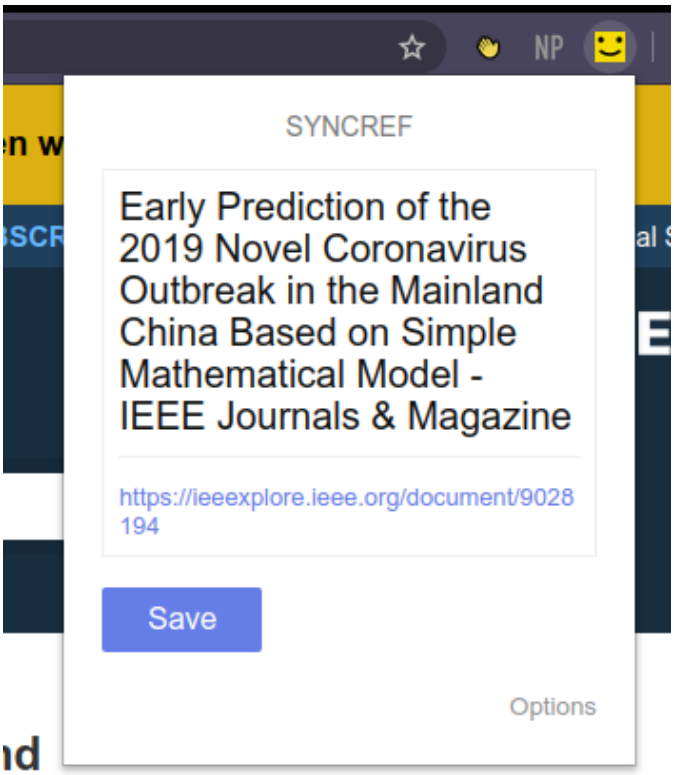

Figure 25: Extension popup

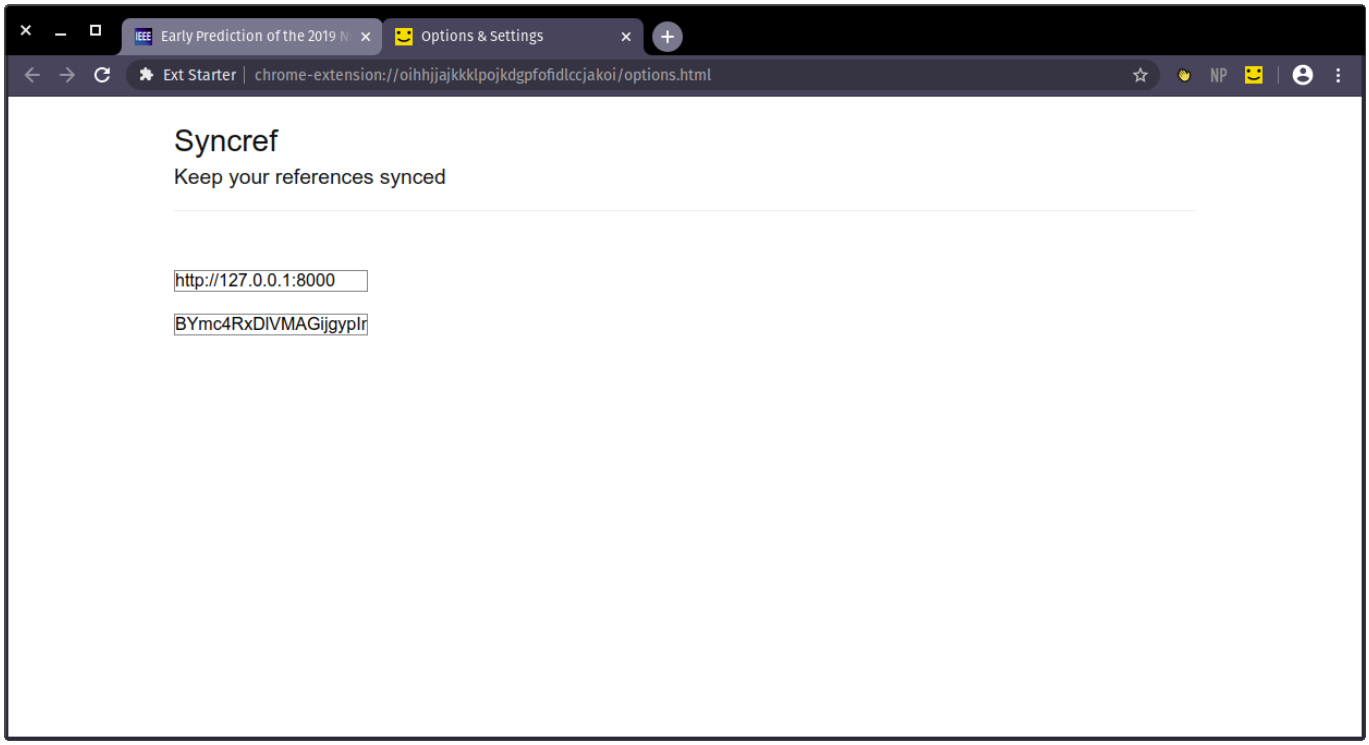

Figure 26: Extension options

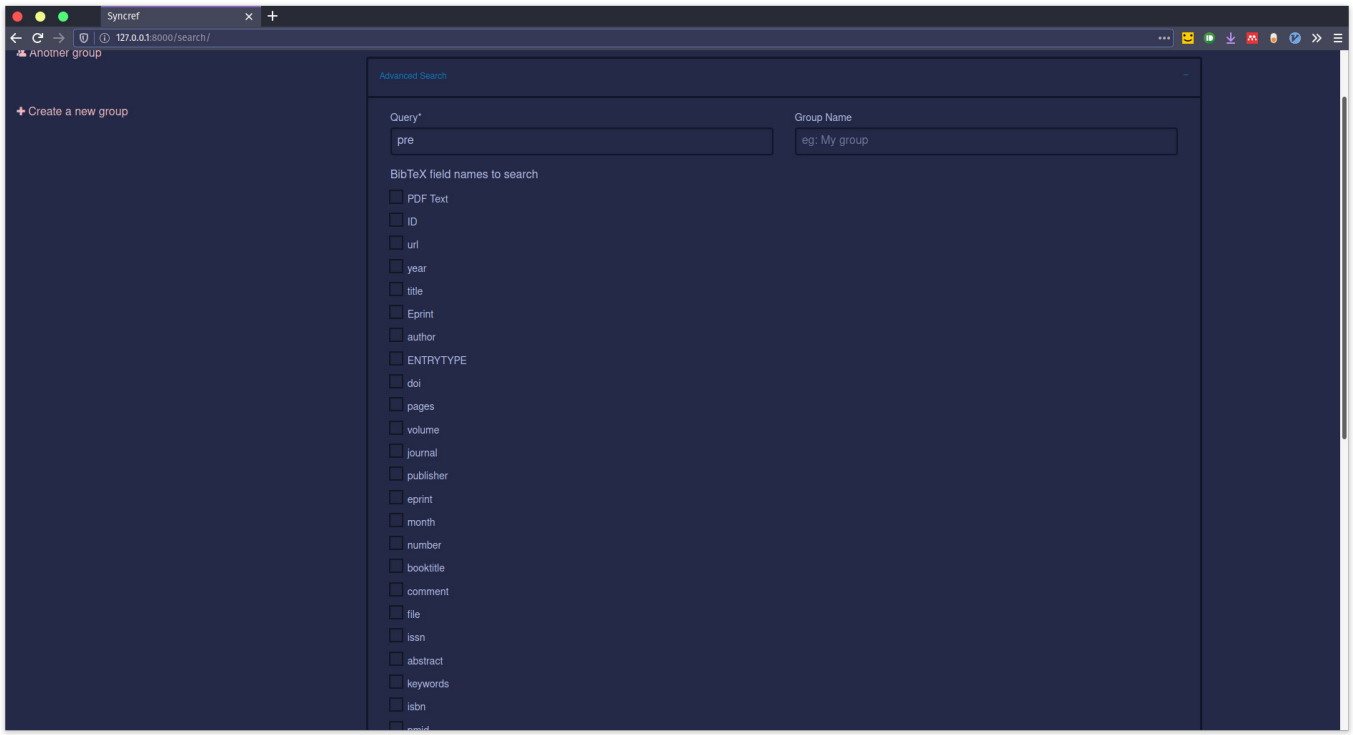

Figure 27: Search filters

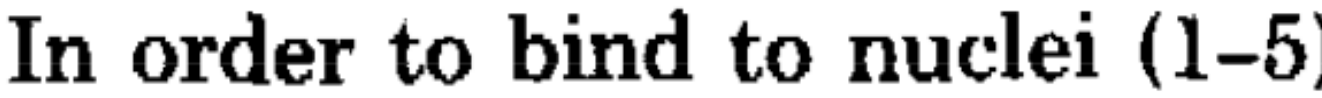

Figure 28: Scanned PDF

References

[1] F. Brischoux and P. Legagneux, "Don't format manuscripts," *The Scientist*, vol. 23, no. 7, p. 24, 2009.

[2] O. Patashnik, *Bibtex.web*, Sep. 2011. [Online]. Available: https://web.archive.org/web/20110927042356/http://www.tex.ac.uk/tex-archive/bibliography/bibtex/base/bibtex.web (visited on 05/25/2020).

[3] *Comparison of reference management software*. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_reference_management_software (visited on 05/25/2020).

[4] D. Stillman, *Zotero Data Server*. [Online]. Available: https://github.com/zotero/dataserver (visited on 05/25/2020).

[5] *DCMI: Dublin Core Element Set, v 1.0: Reference Description*. [Online]. Available: https://www.dublincore.org/specifications/dublin-core/dces/1998-09-01/ (visited on 11/04/2019).

[6] C. Ramakrishnan, A. Patnia, E. Hovy, and G. A. Burns, *Layout-aware text extraction from full-text PDF of scientific articles*, May 2012. DOI: 10.1186/1751-0473-7-7.

[7] O. Patashnik, "BibTeXing," pp. 9–11, 1988. [Online]. Available: http://www.ctan.org/tex-archive/biblio/bibtex/contrib/doc/.

[8] K. Kingsbury, *MongoDB 4.2.6*, May 2020. [Online]. Available: http://jepsen.io/analyses/mongodb-4.2.6 (visited on 05/25/2020).

[9] *PostgreSQL Documentation: 12: 8.14. JSON Types*, 2020. [Online]. Available: https://www.postgresql.org/docs/current/datatype-json.html (visited on 05/27/2020).

[10] S. Marinai, "Metadata extraction from PDF papers for digital library ingest," in *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*, 2009, pp. 251–255, ISBN: 9780769537252. DOI: 10.1109/ICDAR.2009.232.

[11] B. G. Cui and X. Chen, "An improved hidden markov model for literature metadata extraction," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6215 LNCS, Springer, Berlin, Heidelberg, 2010, pp. 205–212, ISBN: 3642149219. DOI: 10.1007/978-3-642-14922-1_26.

[12] M. Lipinski, K. Yao, C. Breitinger, J. Beel, and B. Gipp, "Evaluation of header metadata extraction approaches and tools for scientific PDF documents," in *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries*, New York, New York, USA: ACM Press, 2013, pp. 385–386, ISBN: 9781450320764. DOI: 10.1145/2467696.2467753. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2467696.2467753.

[13] *Deep Learning models*. [Online]. Available: https://grobid.readthedocs.io/en/latest/Deep-Learning-models/ (visited on 05/27/2020).

[14] *How to deploy with WSGI*. [Online]. Available: https://docs.djangoproject.com/en/3.0/howto/deployment/wsgi/ (visited on 05/27/2020).

[15] *Specifications — ASGI 2.0 documentation*, 2018. [Online]. Available: https://asgi.readthedocs.io/en/latest/specs/index.html (visited on 05/27/2020).

[16] *App Containerization*. [Online]. Available: https://www.docker.com/resources/what-container (visited on 05/27/2020).

[17] *DOI Content Negotiation*. [Online]. Available: https://citation.crosscite.org/docs.html%20http://citation.crosscite.org/docs.html (visited on 11/03/2019).

[18] A. Gilmartin, *DOIs and matching regular expressions*, Aug. 2015. [Online]. Available: https://www.crossref.org/blog/dois-and-matching-regular-expressions/ (visited on 05/29/2020).

[19] Bharani, *A template for building cross browser extensions for Chrome, Opera & Firefox.* 2019. [Online]. Available: https://github.com/EmailThis/extension-boilerplate (visited on 05/31/2020).