Entangled Philosophies

Group 1

Summer 2021

Entangled Philosophies is a collaborative effort between Mitchell Wise, Zachary Sutrich, Blaze Wiseman, Isabelle D'Oleo and Ahmad Barhamje.

Contents

1 Project Overview

1.1 Executive Summary

As the world continues to develop, the study of humanities is constantly undergoing fundamental changes and it is becoming increasingly important for historians to approach their studies in a global context, rather than a European context. In the field of philosophy, specific and efficient research on a global scale is becoming more challenging for philosophers to handle, especially as the scope of their research continues to expand. While there are many resources that cater to learning about different philosophies from around the world, there doesn't exist an efficient way to catalog and compare these varying ideas in a way that would be beneficial to a studying philosopher.

This is where the University of Hildesheim begins their work to modernize the study of philosophy and create an environment that makes global philosophies easily accessible. Currently, they have a large database of cataloged papers, but their organization is very linear and more akin to a file hierarchy than a robust query system.

Our goal is to help them revamp their entire database by creating a robust tagging system that allows newly cataloged papers to be easily indexed. With this new system, researchers will be able to add new texts, efficiently filter papers by multiple categories (language, time, author, etc.), and graphically visualize the data (through word clouds, heat maps, network of texts based on their tags, etc.).

1.2 Statements of Motivation

I find the field of philosophy to be fascinating. How ideas develop, travel and change with people, countries, identities - how these ideas can be grouped into schools of thought and categorized. Seeing a project like this, where we're directly in the center of this categorization system, was incredibly attractive to me. I was motivated to select and work on this project because I want to be the driving force behind that. To see how these philosophies entangle and intertwined, and to lay the groundwork for future researchers to find new connections and groupings.

I think this system can do wonders for that. Visualizing how all these connect, and even helping researchers find new papers to review that are closely related to the topic they were already looking for.

- *Zachary Sutrich*

What initially made me interested in Entangled Philosophies was how those involved would be working to help maintain an organized catalogue of multicultural philosophies, which would greatly assist researchers. As someone who is interested in the different languages and cultures of the world, it caught my attention.

People's perspectives are impacted by their cultures as well as the languages they speak. From speaking and interacting with people in different languages, it's apparent that people's values and the sense of worldview are impacted in large by the societies they live in. This is why this project is so important, as there is a need to archive the history of world philosophies.

I am motivated to help in the creation of an application that gives users access to search through a range of philosophies, and see just how different cultures think about the same topics. While no one philosopher is a monolith for their culture, it's good to see how different cultures have differing perspectives. This is one of the reasons I'm motivated to help create an efficient querying and tagging system, so as to make the process of comparing these philosophies simpler.

- *Isabelle D'Oleo*

Philosophy (and history in general) has always been a topic that seemed interesting to learn about since it involved detailed discussions about seeking the fundamental truths of the world and how the many parts of the world interact with each other. However, it's not surprising that most of the philosophical ideas that we're introduced to through standard education only include a Eurocentric point of view (and even then, we mainly only cover people like Aristotle). So, getting to help design a system that intends to organize and emphasize different philosophical ideas from around the world is a great opportunity to expand the field of philosophy.

From a more technical standpoint, after doing years of programming competitions and being exposed to various data structures, seeing how computer scientists organize and query data has always been very interesting to me. In this project, it's almost as if we get to create our own data structure and come up with an innovative way to help researchers query their own data. Overall, there's a lot to learn about and design in the coming months and it'll be exciting to see it all come together in a completed website.

- *Ahmad Barhamje*

My main motivation to choose and work on this project was seeing how human languages were involved. I have a passion for studying languages, how they connect, and how it influences the cultures of the people speaking it or vice versa. Seeing that the website and database were meant to have multilingual support and also having articles in other languages about other cultures, I knew it meant interacting with other languages and that enough seemed to get me interested. In the end I would be creating something that others could use to learn about other cultures and therefore also how the language could interconnect with it. The philosophies show how those cultures can have different ways of thinking influenced by native language which has been an area of interesting study for me.

- *Blaze Wiseman*

Over a long period of time, we have accumulated a massive amount of data, about how the inner workings of the world work. Some of the hardest problems solved to date, have been solved by philosophers from various cultures and different upbringings. Wouldn't it be amazing if we could find out how famous philosophers were related and connected by different categories, then use that data to learn all kinds of useful information. This would be nice, although there is no relational database that connects all the philosophers together for researching purposes.

My motivation for this project came from me wanting to tackle the problem of not being able to visualize patterns within complicated data. Some of the great philosophers of the world such as Aristotle, Plato, Thomas Aquinas, etc. Are unique individuals and it would be difficult to figure out how they were connected and related within many categories. I have a passion for traveling and learning about history and culture, so this was a perfect fit for fueling my motivation to make significant contributions to Entangled Philosophies.

- *Mitchell Wise*

1.3 Goals and Objectives

Our goal for this project is to convert a static WordPress (non-relational database) site that contains textual information about all the philosophers over time. Into a relational database based website that can be logged into by users/admins, which then can choose which language they want to visualize the websites information in, as well as create sophisticated queries by using a GUI for researching purposes. This querying system within Entangled Philosophies would be completed by using a tagging software approach

on the different categories for each philosopher, to figure out how they were related or not to one another.

We will decide on software/hardware that will be used with our group/sponsor. For version control we will use Github and for communication/meetings we will use Zoom and Discord. We will evaluate progress of the project twice a week and meet with our sponsor biweekly to go over questions, concerns and progress. Every group member will be assigned different tasks to complete and we will use a Trello board and burndown/velocity charts to measure our personal contributions as well as progress to make sure we are following the project timeline.

1.4 Broader Impacts

The impact from Entangled Philosophies will help contribute to research within the STEM community by using a relational database that has all the philosophers in the world, within a flexible and scalable website that will be able to achieve complicated queries for researchers. This will help researchers find out various details about these philosophers and can make a difference on a global scale by understanding the relations/differences from the different categories such as Religion, Country, Race, Gender, etc. Entangled Philosophies can also help on a local scale as well by providing students with ways to make their own searches.

1.5 Constraints

The time to complete the project is a constraint as we have a specific short amount of time it has to be completed by. This is even more a constraint because of the project being in the summer semester so it must be done sooner and during a shorter semester. There is also the constraint of the LAMP stack being predetermined, as well as the server being limited in what it can handle.

This means we can’t choose what languages we will use and might have more to research before we can start. Money can also be a limiting factor for what we could have available to us, as explained in the budgeting section. The final constraint is that we are working with the prototype as the final site cannot be completed and released until a grant is received later on.

1.6 Legal, Ethical, and Privacy Issues

For legal issues, we will consult with only textual documentation from philosophers that are approved by our sponsor. We can get in trouble and face possible charges if we are a website that states to have true information about philosophers when we do not. Our sponsor being a University has ethics which we must follow that abides to their specific university. For privacy issues we will make sure that user/administrator accounts have security in place that they trust and will be able to recover their account if they need. While not every paper will include the full text, we have been requested to include that capability, and for that we must be careful. Any paper with its full text published online without consent of the author is a violation of copyright in many countries and could result in fines and jail time.

1.7 Project Budgeting and Finance

As the server and domain name are already being paid for by the sponsor, we would not need to pay for anything for this project. There is the possibility of paying for graphic design which would have a recommendation budget of around $400, but this would likely not be needed.

2 Specifications and Requirements

2.1 Front End

- Main site should be in English and German
- Users should be able to create accounts. Should have admin and user accounts
- Users should have a password recovery system.
- User can choose language on home screen (drop down maybe)
- GUI search friendly feature (query). User GUI queries support (AND, OR, NOT) relationships
- Users can save queries that they've made for future reference.
- Data entry for a single bibliography: information like author, time written, location
- Administrators would have the ability to upload a spreadsheet with the data for several papers and have them automatically imported (with tags pulled from the spreadsheet). Should also be error handling for improperly formatted spreadsheets.
- Users can tag individual papers with their own personal tags that they can then use for queries (only they should be able to view their personal tags).
- Data visualizations: word clouds, heat map (for time periods,areas, etc.).

2.2 Back End

- MySQL Database.
- Website/database needs to process non-english characters.
- Website should run on a LAMP stack.

- API endpoint for adding new papers (properly stores them in database and updates the relevant tags).

2.3 Tagging System

- Admin accounts should be able to create new public tags for papers

- Tags for papers should be in multiple languages but should reference the same set of papers.

- Tags names should be standardized. We shouldn't have similar tags like "renaissance" and "Renaissance era" be considered separate tags.

2.4 Stretch Goals

- Site basically just googles the paper to find a place to read it (or agree on a specific site from the sponsor).

- The website can support seamless addition of new languages.

- Topic modeling software to find tags common to papers in the system (Mallet).

- Add an auto-complete feature to the upload and search pages when the user has to manually input a tag.

3 Division of Labor

3.1 Mitchell

- Project management: tracking the progress of the project
- Front end development
- Database design
- CSS design
- UI Design

3.2 Zachary

- Front end development
- Language swapping
- Search System (Javascript)
- UI Design

3.3 Blaze

- Back end development
- Email API
- Account endpoints
- Paper endpoints
- Designing tagging system
- Database design

3.4 Isabelle

- Front end development
- Data visualization
- D3JS
- Account security
- UI design

3.5 Ahmad

- Back end development
- Designing tagging system
- Tag endpoints
- Search endpoints
- Database design

4 Research

4.1 Framework

For our framework, we will be using React on a LAMP stack. Because of the limitations of the host server, it cannot handle anything big or complex, so the LAMP stack is a main requirement for the project given by the sponsor. We are, however, still able to use React as a way of generating the front end and have been encouraged to do so. We will be using React to generate the HTML, and then transfer the build to the server. The rest of the stack remains as a normal LAMP stack. We can make use of some small node modules since they are integrated with React, but will not be using a back end based in Node.js.

4.1.1 Benefits of React

React.js is a JavaScript library and framework that is simple to use and keeps everything in one place. It has easy integration with other JavaScript libraries such as Node.js and Express.js. It can be used for web app or mobile development and makes it easy to write new components and elements [1]. We will be using it because of the group's familiarity, encouragement from the sponsor, project structure, and because its access to Node.js libraries gives us functionality we had planned on using such as Nodemailer.

4.1.2 Why we cannot use Node.js for the back end

Using Node along with React starts moving too close to a MERN stack and far from the original LAMP stack which was required to be used. The webapp is being developed to be run on servers which can't handle much and should be limited to something like LAMP. We know that React will work and benefit us, but any more and it is taking away from the simplicity of the LAMP stack we started with. However the node package manager and its modules can still be used and have been very helpful for building and testing the project.

4.1.3 How we can still integrate React without the MERN stack

As a React App is essentially just generating HTML, it is possibly to create the project and then move it onto the Apache web server [2]. React is compatible with any kind of back end, which means that we can still use PHP [3], and will send the data as JSON for ease of use and compatibility with the JWT. This means that a full LAMP stack can still be used with a React App as the base for the HTML on the web server.

First, the React app was built locally using create-react-app from the npm. All of the work and testing could be done here and run using npm start. The package.json file must be updated to specify the homepage, which is where the app will be moved to (Figure 1). Once ready to be put on the server, a production build is created with npm run build. This folder is then copied on to the Apache server. The API is made up of only
php files, and is stored in a separate API folder on the server that is accessed by the React app's JavaScript.

Figure 1. Default package.json as generated by
create-react-app, with homepage added

As a React App is essentially just generating HTML, it is possibly to create the project and then move it onto the Apache web server [2]. React is compatible with any kind of back end, which means that we can still use PHP [3], and will send the data as JSON for ease of use and compatibility with the JWT. This means that a full LAMP stack can still be used with a React App as the base for the HTML on the web server.

First, the React app was built locally using create-react-app from the npm. All of the work and testing could be done here and run using npm start. The package.json file must be updated to specify the homepage, which is where the app will be moved to (Figure 1). Once ready to be put on the server, a production build is created with npm run build. This folder is then copied on to the Apache server. The API is made up of only php files, and is stored in a separate API folder on the server that is accessed by the React app's JavaScript.

4.2 Email API

The use of email API allows the webapp to access email functions such as for sending and receiving emails [4]. We need this functionality for two key features of the webapp: verifying an email address and password recovery. When a user creates a new account using an email address, an email will be send to that address containing a link to update the is_verified value in the database under that user. For password recovery or changing your password, another link will be emailed to the same address to reset the password in the database. Because these emails can be requested by the user on the front end but alter the database, this functionality is put in the API.

4.2.1 Why use an access token

In order to create a more secure API, an access token is used to authorize the access of information through the API, without revealing or transferring private information such as the user's credentials [6]. This can be used in situations where confidential information is needed to perform an API request and should be secure. For the example of the email API, the user's email, old password, new password, and other information transferred in this API should be secured, and therefore should use an access token.

4.2.2 JWT access tokens

JWT stands for JSON Web Token, which means that all the information including the header, payload, and signature are contained in a JSON. The benefits to this are that it is completely self-contained, and less verbose than other web token options such as ones in XML [5]. JWT also has a smaller size as it is compact in its own JSON, making it easier to pass through the HTML of the webapp (Figure 2).

4.2.3 Benefits of Nodemailer

Nodemailer is a single JavaScript module with zero dependencies, and integrates perfectly into a React App. It has a heavy focus on security and uses OAuth2 authentication, which includes a JWT access token. It is also a generally powerful module for email API because of its additional features such as attachments and unicode support [7]. It can be installed easily using Node.js and npm (node package manager). This module meets all our requirements and more.

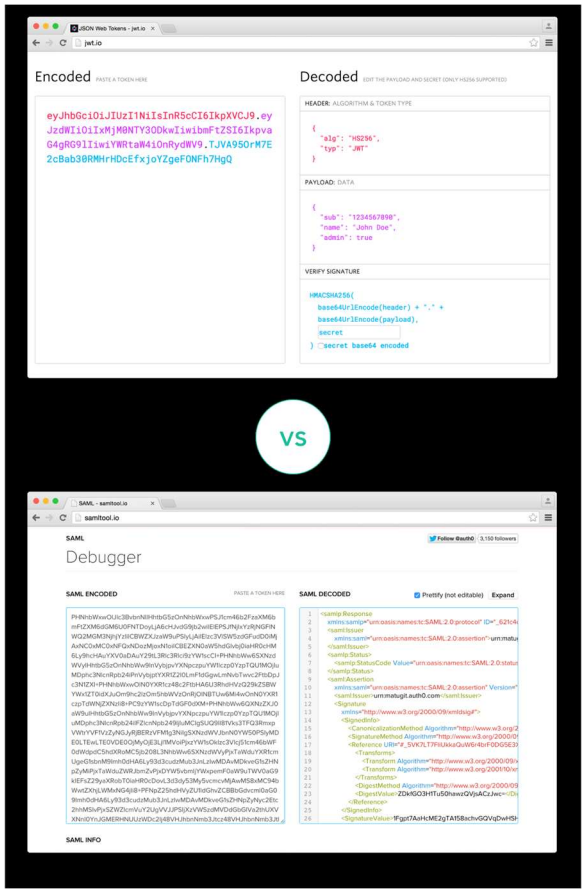


Figure 2. Comparison of the size of JSON based JWT and XML based SAML [5]

4.2.4 Email API without Node.js back end

React comes along with node modules that are lightweight and easy to install through npm. Integrating React into our LAMP stack allows us to make use of some great functionality from npm. With a simple command like ‘npm install nodemailer’ in the project directory, the module is installed immediately for use. It can be used easily with the server because of its small size and how light weight it is. Nodemailer specifically as stated before, is a single module with zero dependencies, making its use very simple.

Unfortunately from testing it seemed that these modules were unable to be used without something like Express.js for API in place of PHP.

4.2.5 Alternate options to using PHP as a back end

While PHP is more limited than using the great features of Nodemailer, we can still use it just for the basic functionality we mainly need. PHP has a built-in mail() function that can be used to send emails in a very simple way [8]. Making complex emails with images, in depth formatting, attachments, etc. is not as easy as just using Nodemailer, however creating a simple email such as for email verification or password resetting, PHP's mail() function can actually make the process much simpler. Using this approach would keep the whole back end to only PHP and not need to rely on node modules.

Nodemailer requires you to set up an SMTP (Simple Mail Transmission Protocol) to send the mail, while PHP has one already set up for you. The PHP function comes pre-installed and has very few required parameters, such as the email to send to, and the message contents. This simplicity is perfect for what we need to do and the main benefit of it. Nodemailer however is more powerful after some setup, and is an easily integrated node module which work automatically with React.

4.3 Connecting the PHP back end

4.3.1 Connecting to the MySQL database using PHP

```php
<?php
$servername = "chdr.cs.ucf.edu";
$username = "entangled_philosophy";
$password = "GjOIYa1O1OpwQR8X";
$dbname = "entangled_philosophy";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn -> connect_error) {
        die("Connection failed: " . $conn -> connect_error);
}

echo "Connected successfully";

mysqli_close($conn);
?>
```

Figure 3. Basic connection test to the database

The first step in connecting the API was to create the PHP files themselves as a standalone file which can query the MySQL database. The first immediate problem was that the database could not be logged into remotely using PHP. The front end React App was able to be created and tested locally before being transferred onto the server, but the API endpoints could not access the database unless they were hosted on the web server. However, this was not an issue because the API was meant to be standalone. We can still test the endpoints locally using SwaggerHub or on our local React app, by accessing the URL they are stored in on the server. Once completed, the API can be hosted on this server and accessed by the rest of the app whenever needed.

A sample connection test code was created to first establish the connection, and placed on the server (Figure 3). Upon viewing this file on the web server, the output of “Connected successfully” indicated the PHP now had access to the database tables and can be queried using MySQL. The connection information such as the username and password were stored in a separate database.php file and included at the top of each file that needed a connection.

Below there is an expanded test of the PHP file including the querying of the database and basic filtering (Figure 4). The output is then displayed, showing that the PHP can successfully access or alter the database (Figure 5). This output could then be encoded to JSON format, instead of the cleaner format used here for testing. Once the front end is able to access the JSON text output by an echo command on this endpoint, the database can be queried directly from the front end.

```php
$sql1 = "SELECT ID, Data FROM TestTable";
$sql2 = "SELECT Data FROM TestTable WHERE ID = '1'";
$result1 = $conn->query($sql1);
$result2 = $conn->query($sql2);

echo "<br><br>Test 1: Output all data in table";
if ($result1->num_rows > 0) {
    while ($row = $result1->fetch_assoc()) {
        echo "<br>ID: " . $row["ID"] . " - Data: " . $row["Data"];
    }
} else {
    echo "0 results";
}

echo "<br><br>Test 2: Look up ID 1's data<br>";
if ($result2->num_rows > 0) {
    $row = $result2->fetch_assoc();
    echo $row["Data"];
} else {
    echo "0 results";
}
```

Figure 4. Two tests querying the database from PHP to show they are connected

```
Connected successfully

Test 1: Output all data in table
ID: 1 - Data: This is a test
ID: 2 - Data: This is a second test
ID: 7 - Data: This is a 3rd test

Test 2: Look up ID 1's data
This is a test
```

Figure 5. The output of the previous test, before JSON encoding

4.3.2 Connecting to the API with XHR

Once the API could communicate with the database, the endpoints can be completed and stored on the API directory of the server, and tested independently even without a front end. Using SwaggerHub, we can access all the endpoints, and therefore alter the database from outside of it. This represents the working back end, and the next step is to actually integrate it into React in place of Node.js.

We can use XHR to send a request from the JavaScript to the PHP. The request is opened on the URL where the endpoint is stored on the server. A request header is set to be JSON format, and includes the data received from the front end encoded manually to JSON in the JavaScript with String concatenation (Figure 6).

```
var urlBase = 'http://chdr.cs.ucf.edu/~a1657032/api';

export function getData(i) {
    var id = parseInt(i);

    var jsonPayload = '{"id":' + id + '}';
    var url = urlBase + '/getDataById.php';

    var xhr = new XMLHttpRequest();
    xhr.open("POST", url, false);
    xhr.setRequestHeader("Content-type", "application/json; charset=UTF-8");
    try {
        xhr.send(jsonPayload);
        var jsonObject = JSON.parse(xhr.responseText);
        document.getElementById("answer").innerHTML = ("Answer: \"" + jsonObject.data + "\"");
    } catch (err) {
        document.getElementById("answer").innerHTML = err.message;
    }

}
```

Figure 6. Example JavaScript function to call an API endpoint with XHR

The API endpoint located at the given URL is then run given the input header. An example of an endpoint that might be accessed is the example PHP and MySQL connection from the previous section. The main changes from the example in order to make this next

connection is the query being based on the decoded JSON data from the input header (Figure 7), and the echo'd output being encoded back to JSON data (Figure 8). The input JSON is converted to an associative array, which means PHP can access different values associated with another value, such as a String, instead of the index of that element. This works similarly to accessing data from a JSON object, where each element is associated with its name.

```
$inData = json_decode(file_get_contents('php://input'), true);
$sql = "SELECT data FROM TestTable WHERE id = " . $inData["id"];
$result = $conn->query($sql);
```

Figure 7. Decoding input header from JSON to associative array for database query

```
echo '{"data":"' . $row["data"] . '"}';
```

Figure 8. Encoding the output data to JSON

4.3.3 Connecting everything to the user

Everything is now functional, but all done behind the scenes. There needs to be a way to access it all at the touch of a button, from outside perspective of the user on the website. To test this, some simple HTML was created on top of the default made by create-react-app. A basic text input and submit button were added, along with an output text area (Figure 10). The most import part was that the JavaScript function shown off in the previous section that sends the XHR to the back end, is imported from the other file into React's App.js, and then called when the button is clicked (Figure 9). The returned JSON from the API is used to update the answer area in the HTML (Figure 6) and completing the full circle of connections. Every part of the stack can communicate to and from each other. The React App HTML and JavaScript is build and transferred to the server, and the API is hosted separately on the server to be accessed. Any part can now be tested for a particular job without needing to understand the other parts, and the user can see a clean webpage to access database functions.

```
<body className="App-body">
  <input type="text" id="id" placeholder="Enter ID"/>
  <button type="button" id="submit" onClick={() => getData(document.getElementById("id").value)}>Submit</button>
  <div id="answer">Answer: </div>
</body>
```

Figure 9. JavaScript arrow function calling a function on click of the button

Figure 10. Final result of full circle connectoin test for the LAMP stack with React

4.4 Adding papers to the database

With everything hooked up properly and communicating, work can begin on adding the API endpoints, populating the database, and creating the website. One of the major requirements of the website that should be done first is to create a page where specifically admin level users can enter information about a paper along with an upload of that paper. These papers should all be stored in the database to be queried by the user later such as in searches using the tagging system. Before any testing can be done to test the functionality of the database and making an API that interacts with it, the database should be populated with papers to work with. For this an endpoint will be created that takes in all the data about the paper, and adds a row to the database. Then the page on the website can be created later that accesses this endpoint to enter the data about the paper.

4.4.1 Database paper table structure

Before data for papers can be added, a table to store that data should be added to the database. For this a Papers table will be created with a schema that allows many different kinds of data about the paper to be stored. Some of these will be entered by the user such as the title, while others will be created automatically such as time uploaded and ID.

More columns can be added an expanded as these features are added, so for testing just some basic fields will be created (Figure 11). The paper itself that will be uploaded will have its URL stored in this table as well.

This table will have links to the other tables such as the admin user that created the paper, or the philosopher that wrote it, but those other tables are not created yet. It also should be noted here that tagging system will be involved with this table, but is complex and will be added later. So it was also excluded from the first test.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	**id** 🔑	int			No	*None*		AUTO_INCREMENT
2	**title**	varchar(100)	utf8mb4_0900_ai_ci		No	*None*		
3	**author**	varchar(100)	utf8mb4_0900_ai_ci		No	*None*		
4	**url**	varchar(256)	utf8mb4_0900_ai_ci		No	*None*		
5	**created_at**	timestamp			No	CURRENT_TIMESTAMP		DEFAULT_GENERATED

Figure 11. Test table for papers in the database

4.4.2 Add paper API endpoint

Now that the table is set up, we can alter it using PHP. A file can be created for this that takes the title, author, and URL as input, and adds a row to the table using those three values. The ID and created_at timestamp will be created automatically upon adding this row to the table.

This endpoint differs from the query test because it is adding something new to the database. For this ‘INSERT INTO’ is used instead of ‘SELECT’. Rather than returning a table of results, INSERT simply returns True or False on success or error respectively. Since we have no data to return from this, the resulting JSON will just return the status of the operation, based on whether the result was True or False (Figure 12). With just this, the endpoint will add a new entry to the papers table based on currently three inputs, but can be easily expanded to include more inputs later.

```php
<?php
include "database.php";

$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$inData = json_decode(file_get_contents('php://input'), true);
$sql = 'INSERT INTO papers (title, author, url) VALUES ("' .
    $inData["title"] . '", "' . $inData["author"] . '", "' . $inData["url"] . '")';
$result = $conn->query($sql);

if ($result) {
    echo '{"status":"success"}';
} else {
    echo '{"status":"error"}';
}
?>
```

Figure 12. FIrst version of add paper API endpoint

4.4.3 Uploading a file

The URL of the uploaded file is how the database is able to store access to the paper's file, however for this URL to exist in the first place, it must be put on the server after being uploaded by the admin user. A separate directory on the server will be created for storing uploaded papers. The API endpoint described in the previous section can work properly on its own, but just needs to be given a valid URL. For that reason, the uploading of the file can be done in JavaScript in the React App using XHR in the same place and way as the XHR connecting to the API. Rather than posting a JSON to the given URL, it posts the file there to be stored.

This is done in combination with FormData API to get the user to select a file and store the data [9]. Then that data can be sent to the server (Figure 13). First the file is selected with the querySelector, and the data is appended to the formData. This data is posted to the given URL and therefore stored on the server [10]. The URL it is stored in is a part of the JSON data returned from a successful upload, so this can be sent to the PHP endpoint that eventually adds the paper. This method will be used to implement file uploading to a URL, to keep consistent with the XHRs we are already using for our API.

```
// collect files
const files = document.querySelector('[name=file]').files;
const formData = new FormData();
formData.append('avatar', files[0]);

// post form data
const xhr = new XMLHttpRequest();

// log response
xhr.onload = () => {
    console.log(xhr.responseText);
};

// create and send the reqeust
xhr.open('POST', url);
xhr.send(formData);
```

Figure 13. Example usage of FormData API and uploading a file with XHR

4.4.4 Front end access to the endpoint

The front end webpage will then have a page available to enter this data for the paper, uploading the file and adding it to the server and database. For now, the previous test page was appended to include this functionality for testing (Figure 14). Hitting the submit button runs another JavaScript function connected to the API that takes the values in the two text fields, along with the URL for the file location of the uploaded paper (Figure 15). The ID and created_at timestamp are both generated automatically and all of this data is added to a new row of the papers table in the database.

Eventually the JavaScript code will handle the uploading of the file as described before, but for the purpose of back end testing, a placeholder URL was used. This demonstrates the PHP code to be accurate and connection to the database to be working, as it correctly adds the placeholder URL which will eventually lead to the location the file is actually stored. As stated in the database section for this endpoint, only the basic fields were set up for this test (Figure 16), but can be easily added to in the same methods shown in the previous sections.

Figure 14. Simple text input adding paper functionality, later to become full page

```
export function addPaper() {
    var title = document.getElementById("title").value;
    var author = document.getElementById("author").value;
    var uploadLocation = "http://chdr.cs.ucf.edu/~al657032/paper/test";

    var jsonPayload = '{"title":"' + title + '", "author":"' + author + '", "url":"' + uploadLocation + '"}';
    var url = urlBase + '/addPaper.php';

    var xhr = new XMLHttpRequest();
    xhr.open("POST", url, false);
    xhr.setRequestHeader("Content-type", "application/json; charset=UTF-8");
    try {
        xhr.send(jsonPayload);
        var jsonObject = JSON.parse(xhr.responseText);
        if (jsonObject.status === "success") {
            document.getElementById("answer").innerHTML = ("Answer: Success");
        } else {
            document.getElementById("answer").innerHTML = ("Answer: Error");
        }
    } catch (err) {
        document.getElementById("answer").innerHTML = err.message;
    }
}
```

Figure 15. JavaScript function that will upload the file and generate the URL

id	title	author	url	created_at
1	TestT	TestA	http://chdr.cs.ucf.edu/~al657032/paper/test	2021-03-15 22:19:58
2	Test Title	Test Author	http://chdr.cs.ucf.edu/~al657032/paper/test	2021-03-15 22:28:05

Figure 16. Example rows in the papers table showing a successful insertion

4.5 Secure Account Creation

4.5.1 Secure Account Creation Summary

For our Senior Design project, we will be designing a platform for cataloguing different philosophical works. In order to access these works, as well as search through these works, users would need the ability to create an account. One of the many responsibilities we have as the students working on this project is to make sure that account creation is as simple for users to navigate as it is secure. For users accessing the website, as well as the owners of the website, they should be able to trust that we can create an application that is fully functioning as well as secure. We can create this by keeping personal information such as emails and passwords safely stored into our databases. Whenever one is logged into a website or application, one trusts the website to keep all of their information secure, therefore it is our job to make sure we also abide by these expectations.

4.5.2 Password Hashing

For security reasons, one wouldn't want to take the passwords submitted to an application and store them in a database completely in plaintext. If this were the case, anyone with access to the database could be able to see every password submitted and stored, and immediately know each of the passwords for a given account. This is the equivalent to writing all of your passwords on a piece of paper, anyone with access to it can clearly see all of your passwords.

This is why we will have to hash passwords before entering into the database. This way, even if one were to gain unauthorized access to the database, they would only see the passwords in an encrypted form, which would be unusable for them if they wouldn't know how to decrypt it. What hashing does is create a one-way transformation onto any password being submitted during the account creation or password changing process, turning the password into another string before it then gets stored into the database. What is being stored into the database would then be a hashed password.

These types of hashes however, are not enough to meet our needs. We would also need a password hasher that can add salt for extra protection. Salting adds more data to the password in order to make the passwords less vulnerable to different attacks, and thus make it more secure. This is important because if two users in the website submit their

passwords, and they just happen to be the same password, an unsalted hash would have the same password hash stored in the database. If someone were to gain unauthorized access to the database, they would be able to see the two identical password hashes and then deduce that the passwords are unsalted, which would therefore make these passwords easier to crack.

In order to mitigate these types of attacks, we would need a password hasher that can add salt to the passwords stored in the database. For that, we have chosen to use bcrypt for our needs.

4.5.3 Bcrypt

As stated above, the reason as to why we will be using bcrypt for our password hashing needs is because bcrypt will allow us to add salt to our passwords to prevent common attacks. Bcrypt is based on the Blowfish block cypher cryptomatic algorithm and is also cross platform. The fact that Bcrypt can be used across different platforms will help us as we are using a React.js front-end on a PHP server. We are also figuring out that we can still use some node packages for React, despite not using Node.js for our stack. This is important because that means there could be two different ways we could implement bcrypt into the Entangled Philosophies website application. We could either try using bcrypt for Node or bcrypt for PHP, which we will detail below.

Bcrypt for Node.js

On the occasion that we could get Node.js or some node packages to work on the Entangled Philosophies server, we could install Bcrypt for Node.js using JavaScript. First, we would have to install bcrypt using npm.

```
npm install bcrypt
```

After installing bcrypt, it can now be added to our files in order to hash passwords. The following code is from the bcrypt's npm package page. [11] The recommended method is to use the async functions provided, which I will be showing first.

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const myPlaintextPassword = 's0/\/\P4$$w0rD';
```

```
const someOtherPlaintextPassword = 'not_bacon';
```

First, we create a variable for bcrypt and add it to our file. We can also create a variable for how many salt rounds we would want to add to our hash. The plaintext password could also be assigned to a variable as long as this plaintext variable is not stored into our database.

Bcrypt here gives us the option to either hash our passwords and salt them separately, or do these in two separate functions. Here we have the first technique where we do each of these steps separately.

```
bcrypt.genSalt(saltRounds, function(err, salt) {
    bcrypt.hash(myPlaintextPassword, salt, function(err, hash) {
        // Store hash in your password DB.
    });
});
```

Here is the technique where we have the hashing and salting done in the same function.

```
bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
    // Store hash in your password DB.
});
```

Ultimately both of these functions do the same exact thing, asynchronously. We could also do these synchronously, however it is not recommended. The reasoning for why the asynchronous approach is recommended over the synchronous approach is because hashing is CPU intensive. The synchronous version will block the event loop and prevent the Entangled Philosophies website application from handling other requests until it finishes. The synchronous functions look like this when they're separated to handle the hashing and salting:

```
const salt = bcrypt.genSaltSync(saltRounds);
const hash = bcrypt.hashSync(myPlaintextPassword, salt);
// Store hash in your password DB.
```

An example when they're both handled in the same function would be:

```
const hash = bcrypt.hashSync(myPlaintextPassword, saltRounds);
// Store hash in your password DB.
```

Password checking can also be done synchronously or asynchronously, here is how we can use the password checking both ways. First, asynchronously:

```
// Load hash from your password DB.
bcrypt.compare(myPlaintextPassword, hash, function(err, result) {
    // result == true
});
bcrypt.compare(someOtherPlaintextPassword, hash, function(err,
result) {
    // result == false
});
```

The example of password checking in a synchronous manner is also shown here:

```
// Load hash from your password DB.
bcrypt.compareSync(myPlaintextPassword, hash); // true
bcrypt.compareSync(someOtherPlaintextPassword, hash); // false
```

Note that both of these still have the same properties as the async and sync functions for creating a hash, meaning that the async functions are still favored because of time costs.

Bcrypt for PHP

In the case that we do not have access to the node packages when using React.js for our front-end, PHP still allows us to use bcrypt hashing and salting. There is a password_hash() function that is already built into PHP that can be used to create a new password hash. The password_hash() function also has a setting in it that allows support for bcrypt. The way we can use the password_hash() function to meet our needs is as follows. The following code has been taken from the PHP manual.

```
<?php
$options = [
    'cost' => 12,
];
```

```
echo password_hash("rasmuslerdorf", PASSWORD_BCRYPT, $options);
?>
```

This function returns the hashed password or false if the operation fails to happen. Here the cost is set to twelve. The cost decontes the algorithmic cost that should be used. If the cost takes too long to perform, one could set a time limit and adjust the cost accordingly. The code for this can be seen below. Again, the code for this has been taken directly from the PHP manual. [12]

The following diagram shows the format of a returned value using the password_hash() function: [13]

Figure 17. Example return value for password_hash()

In the following code, we create a function to evaluate the appropriate cost for the allotted time we would like the function to take. Note that this is only for finding out what cost should be used, and that this function doesn't create the hash in that allotted time frame.

```
<?php
/**
 * This code will benchmark your server to determine how high of a
cost you can
 * afford. You want to set the highest cost that you can without
slowing down
 * your server too much. 8-10 is a good baseline, and more is good if
your servers
 * are fast enough. The code below aims for ≤ 50 milliseconds stretching
time,
 * which is a good baseline for systems handling interactive logins.
 */
$timeTarget = 0.05; // 50 milliseconds
```

```
$cost = 8;
do {
    $cost++;
    $start = microtime(true);
    password_hash("test", PASSWORD_BCRYPT, ["cost" => $cost]);
    $end = microtime(true);
} while (($end - $start) < $timeTarget);

echo "Appropriate Cost Found: " . $cost;
?>
```

Using this function, we can use the cost to complete the hashing process. The hashing function also automatically creates its own secure salt.

Now that the hash is created, we'll also need a function that can take the hash and compare it to the user's password whenever they want to log in. PHP also includes this function, and the function is called password_verify(). In order to use password_verify() we will need the hash and the users inputted password. Example usage for password_verify() function is provided by the PHP manual and can be shown below [14].

```
<?php
// See the password_hash() example to see where this came from.
$hash =
'$2y$07$BCryptRequires22Chrcte/VlQH0piJtjXl.0t1XkA8pw9dMXTpOq';

if (password_verify('rasmuslerdorf', $hash)) {
    echo 'Password is valid!';
} else {
    echo 'Invalid password.';
}
?>
```

This function returns, "Password is valid!".

As shown, bcrypt's ability to help us hash and salt in a simple manner is why we can use it for Entangled Philosophies. Since it is adapted for PHP as well as Node, its flexibility will help us when we start implementing the website itself. This hashing system is also

secure and trusted by many, which is also a reason as to why it's a good candidate for our website application.

4.6 Search System

4.6.1 Basic Search

The first part of our search system is the most intuitive and basic functionality – a standard search bar. This will be able to search through for exact and partial matches through various paper metadata, as one would expect of any search bar encountered on the internet. The goal is intuitiveness of use over power.

First, we start with the text container for the search term.

```
<h5>Paper Search</h5>
<form>
     <input
     id="papersearch"
      name="papersearch"
      autoComplete="off"
      placeholder="Search Term"
     />
     <div>
          <button
           type=”button”
           onClick={handleSearch}
           type=”submit”>
               Search
          </button>
     </div>
</form>
```

A basic text box and button to submit. An option worth considering is using an *onChange* function, allowing the results to update in real time. We will be running performance tests to see the viability of different methods to achieve this given the large amount of data this would be sorting through.

Next, we'll work on the *handleSearch* function.

```
const handleSearch = async e => {
```

```
e.preventDefault();
var term = document.getElementById("papersearch").value;
document.getElementById("papersearch").innerHTML = "";
var jsonPayload = '{"Term" : "' + term + '"}';
fetch('http://192.168.56.1:5000/api/search', {
     method: 'POST',
     headers: {
          'Content-type': 'application/json',
     },
     body: jsonPayload,
})
.then(response => response.json())
.then(data => {
     console.log('Success:', data);
})
.catch((error) => {
     console.error('Error:', error);
     return;
});
};
```

This will take the text entered in the field and construct a JSON payload out to the API to handle. Now, we need to be able to display this data somewhere on the page.
React allows us to create generic arrays of HTML code, making it easy to procedurally add complex elements to the page.

We start by creating a prop to store the results at the beginning of this component.

```
export default class SearchPage extends React.Component {
     state = { results: null };
```

Then, we edit the API response catch to put the data inside the results. For now we'll keep the console logging for debug purposes, though we will remove it upon release.

```
.then(data => {
console.log('Success:', data);
if (data.error == "") {
     this.setState({ results: data });
}
})
```

Now we can take those results and use them to construct an HTML component. For now we'll just display the title of the paper and make it a link to a full page of details, working out a more complex display at a later time.

```
var data = this.state.results;
const papers = []
if (data != null) {
     papers.push(
          <thead className="header">
               <tr>
                    <th>Title</th>
               </tr>
          </thead>
     )
     for (var i = 0; i < data.results.length; i++)
     {
          linker = "/paper?name=" + data.results[i].Title;
          papers.push(
               <tbody key={i}>
                    <tr key={i}>
                         <td id={i} key={i}>
                              <Link to={linker}>
                               <li>{data.results[i].Title}</li>
                              </Link>
                         </td>
                    </tr>
               </tbody>
          )
     }
}
const resultTable = []
if (data != null)
{
     resultTable.push(
          <table className=”resultTable”>
               {papers}
          </table>
     )
}
```

Since React updates components on the fly, the above if statement will read in the null and do nothing until that *data* variable changes – namely, when we get some data to display. Now all we must do is throw this component in our render function.

```
<div className=”container”>
     <h5>Paper Search</h5>
     <form>
          <input
          id="papersearch"
           name="papersearch"
           autoComplete="off"
           placeholder="Search Term"
          />
          <div>
               <button
                type=”button”
                onClick={handleSearch}
                type=”submit”>
                    Search
               </button>
          </div>
     </form>
</div>
<div className=”container”>
     {resultTable}
</div>
```

4.6.2 Advanced Search

Using 5e.tools as a reference for functionality, we can create a filter button that allows more advanced searches within a GUI framework. It’s a middle road between usability and power, meant for more advanced users.

First we’ll plop down a few buttons – for this example, two tags and a submit. For the tags, we’ll place them in a separate function to create local behavior variance, and then put everything in their own component.

```
function filterTag(name, id) {
     function handleClick(e) {
```

```
            e.preventDefault();
            console.log (‘Successful click.’);
      }
      return (
            <div className=”tag”>
                  <button
                   type=”button”
                   onClick={handleClick)>
                        {name}
                  </button>
            </div>
      )
}
```

Then, going back up to build our filter page component.

```
export default class FilterPage extends React.Component {
      render() {
            return (
                  <div>
                        <form>
                              <filterTag(‘20th Century Philosophy’) />
                              <filterTag(‘German Philosophers’) />
                              <button
                               type=”button”
                               onClick={handleClick}
                               type=”submit”>
                                    Save
                              </button>
                        </form>
                  </div>
            );
      }
}
```

Now that the buttons are in place, we need to make them do something. One of the interesting aspects of the model we’re using is that rather than just being on/off, each tag can be in one of three different states – include, exclude or ignore.
When searching through papers, any papers returned MUST have the included tags (category overlaps – such as German Philosophy and Russian Philosophy - will be OR

instead of AND) and the MUST NOT have any excluded tags. Tags set to ignore will not have any special treatment.

We'll assign this to a basic integer where 0 = ignore, 1 = include, 2 = exclude.

```
function filterTag(name, id) {
     var setting = 0;
     function handleClick(e) {
          e.preventDefault();
          console.log ('Successful click.');
          setting++;
          if (setting > 2)
               setting = 0;
          this.props.onTagClick(name, setting);
     }
```

This loops around to a function on the SearchPage component.

```
state = { results: null, include: null, exclude: null };
render() {
     const onTagClick(name, setting) = async e => {
          e.preventDefault();
          if (setting == 0)
          {
               const index = exclude.indexOf(name);
               if (index > -1)
                    exclude.splice(index, 1);
          }
          else if (setting == 1)
          {
               include.push(name);
          }
          else if (setting == 2)
          {
               const index = include.indexOf(name);
               if (index > -1)
                    include.splice(index, 1);
               exclude.push(name);
          }
     }
```

Now that we have an array of included terms and excluded terms, we just need to modify the search button to account for them.

```
const handleSearch = async e => {
     e.preventDefault();
     var term = document.getElementById("handleSearch").value;
     document.getElementById("recipename").innerHTML = "";
     var jsonPayload = '{"Term" : "' + term + `", "include" :
[`;
     var i;
     for (i = 0; i < include.length; i++) {
          jsonPayload += `"` + include[i] + `"`;
          if (i < include.length - 1)
               jsonPayload += `, `;
     }
     jsonPayload += `], "exclude" : [`;
     for (i = 0; i < exclude.length; i++) {
          jsonPayload += `"` + exclude[i] + `"`;
          if (i < exclude.length - 1)
               jsonPayload += `, `;
     }
     jsonPayload += `]}`;
     fetch('http://192.168.56.1:5000/api/search', {
          method: 'POST',
          headers: {
               'Content-type': 'application/json',
          },
          body: jsonPayload,
     })
     .then(response => response.json())
     .then(data => {
          console.log('Success:', data);
          if (data.error == "") {
               this.setState({ results: data });
          }
     }).catch((error) => {
          console.error('Error:', error);
          return;
     });
```

```
};
```

4.6.3 Custom Search

Now that we have our GUI searches done, we can add the most complicated and most powerful functions – a custom search. This allows researchers to create their own database search queries by hand rather than navigating a GUI to do it. It ends up being a much simpler implementation on the front end. First, again, the HTML code.

```
<h5>Custom Search</h5>
<form>
     <textarea
      id="customsearch"
      name="customsearch"
      autoComplete="off"
      placeholder="Database Input"
     />
     <div>
          <button
           type=”button”
           onClick={handleSearch}
           type=”submit”>
               Search
          </button>
     </div>
</form>
```

Then the handleSearch function.

```
const handleSearch = async e => {
     e.preventDefault();
     var query = document.getElementById("customsearch").value;
     document.getElementById("customsearch").innerHTML = "";
     var jsonPayload = '{"Query" : "' + query + '"}';
     .then(response => response.json())
     .then(data => {
          console.log('Success:', data);
          if (data.error == “”) {
               this.setState({ results: data });
          }
```

```
    }).catch((error) => {
        console.error('Error:', error);
        return;
    });
};
```

Allowing users to do this safely will require some major input sanitizing on the back end. Given the audience, it is worth the trouble to support this ability.

4.6.4 Final Search

What we ended up implementing these ideas is first loading every paper in the database to the user end. For a database of reasonably large size (~10,000 papers) we estimated this to take less than five seconds and a negligible amount of memory, though there are alternative implementations should performance become an issue.

The filter button opens a new window with the options listed earlier. All the selections on here form a database query to draw from instead of grabbing all papers, thus forming the pool to display. This pool is what's used for the visualization.

The simple search bar acts as a quick client-side filter for the pool of results gathered, rather than forming its own query. This allows typed searches to be done in real time.

4.7 Language Swapping

4.7.1 Language Selection

The front end of the website is required to support multiple languages – English and German to start, with the ability to easily add new languages in the future at the university's discretion. The first step of this is the user selecting the language, which gives us a couple options:

· Including an opening page which prompts the user to select a language – could include flags of countries and text below them, English and Deutsche with the flags for the US and Germany.
· Accessing the user's IP and tracing it to a country, automatically selecting the dominant language of the country for the website.

The first option adds another click to the user experience, which we consider to be a negative. The second option is prone to a couple errors:

· What if the user prefers a language that is not dominant in their IP location? For example, a researcher visiting another country for a limited time or a recent immigrant.

· What if the user is using a VPN, and the proxy is in a country whose dominant language they do not speak?

To resolve these questions, we could include a button on the main ribbon to manually select another language. The button could be written as: Language/Sprache, adding a new word for each supported language in the future. This would add a click for those niche cases and reduce a click for most users. However, it would add a small extra step for adding new supported languages and could clutter the UI in the future.

In researching this, and researching with the team, I discovered a third option:

· Accessing the user's language preference as set in their browser.

This would reduce the niche inconvenient cases down to negligible, and we could still offer a manual selection button as a backup.

So, how does this work?

Most of this information comes from this StackOverflow discussion. The key lies in the property "navigator.language", though how exactly this property works has changed as web browsers developed. I found a snippet of javascript code adapted from http://angular-translate.github.io/, source code here [15]:

```
function getFirstBrowserLanguage() {
     var nav = window.navigator,
          browserLanguagePropertyKeys = ['language',
'browserLanguage', 'systemLanguage', 'userLanguage'],
          i,
          language,
          len,
          shortLanguage = null;

     // support for HTML 5.1 "navigator.languages"
     if (Array.isArray(nav.languages)) {
          for (i = 0; i < nav.languages.length; i++) {
               language = nav.languages[i];
               len = language.length;
               if (!shortLanguage && len) {
                    shortLanguage = language;
               }
```

```
                if (language && len>2) {
                     return language;
                }
           }
      }

      // support for other well known properties in browsers
      for (i = 0; i < browserLanguagePropertyKeys.length; i++) {
           language = nav[browserLanguagePropertyKeys[i]];
           //skip this loop iteration if property is
null/undefined.  IE11 fix.
           if (language == null) { continue; }
           len = language.length;
           if (!shortLanguage && len) {
                shortLanguage = language;
           }
           if (language && len > 2) {
                return language;
           }
      }

      return shortLanguage;
      }

console.log(getFirstBrowserLanguage());
```

This supports the current versions of Edge, Internet Explorer, Firefox and Chrome, with support for older versions and other common browser properties.
As not every language will be supported, a default language will need to be established during implementation. English is spoken by most countries in the world, so that will be the default.

This operation will return a *language code*, which will be implemented using a list of codes found at https://www.andiamo.co.uk/resources/iso-language-codes/.
As requested by our sponsor, instead of detecting a browser language we have defaulted to English.

4.7.2 Language Selection Consistency

How do we keep that selection with the user while they pass from page to page? There are three options:

- Rerun the browser language detection code on every page.
- Store the language in a query string appended to the URL.
- Store the language in a cookie on the user's computer.

The first option is the easiest to implement but adds a performance delay and – most importantly – means the user cannot manually select a language at all.

The second option would allow storing a manual selection without using cookies, but if a user wishes to manually select a language they would have to do so each time they started a new session on the website. Additionally, the preferred language of the user would be carried over with any links they shared to another person, which could be unnecessarily frustrating.

While the recent push against storing cookies steers me away from the third option, it would allow a manual selection to persist across sessions and exist only for the user in question. For those reasons, we're going with this option.

In Javascript, utilizing cookies is straightforward. We'll need a name for the variable and an expiration date if we want it to last longer than when the browser is closed. We'll choose twelve months in the future for purposes of compliance.

Code has been adapted from https://www.w3schools.com/js/js_cookies.asp. To set the cookie [16]:

```
function setCookie(cname, cvalue, exdays) {
     var d = new Date();
     d.setTime(d.getTime() + (exdays*24*60*60*1000));
     var expires = "expires="+ d.toUTCString();
     document.cookie = cname + "=" + cvalue + ";" + expires +
";path=/";
     }
```

Then to call it:

```
var language = "en-us";
setCookie("lang", language, 364);
```

Then to read it:

```
function getCookie(cname) {
     var name = cname + "=";
```

```
     var decodedCookie = decodeURIComponent(document.cookie);
     var ca = decodedCookie.split(';');
     for (var i = 0; i < ca.length; i++) {
          var c = ca[i];
          while (c.charAt(0) == ' ') {
               c = c.substring(1);
          }
          if (c.indexOf(name) == 0) {
               return c.substring(name.length, c.length);
          }
     }
     return "";
}
```

And to call that:

```
getCookie("lang");
```

Once a user creates an account, we can store their language preference in the database and associate that with the account. This wouldn't have an expiration date, and the above code shouldn't have to run again.

4.7.3 Text Swapping

How do we swap the text on the page based on the language selected?
Since we're using React, we can create a file containing each individual component of text which we place in any section that requires text instead. The following code is what we ended up using:

```
import React from "react"
import { Link } from "react-router-dom"
export function dictionary(s, lang) = {
   switch(s) {
     case 1:
          if (lang==='eng') return "Languages";
          if (lang==='ger') return "Sprachen";
          break;
   }
}
```

Then, when calling a line of text, we write:

```
return
     <div>
          <h1 id=”languages”>{dictionary(1, userLang)}</h1>
     </div>
```

By keeping all the text together, it gives potential future translations an easier time being implemented.

4.8 Spreadsheet Uploading

4.8.1 File Selection

First we have to create a button which allows the user to select a file for upload. This is easily implemented. The following code is adapted from the guide at https://www.geeksforgeeks.org/file-uploading-in-react-js/. First, we must create a new state variable [18]:

```
state = {
      selectedFile: null
};

onFileChange = event => {
      this.setState({ selectedFile: event.target.files[0] });
};
```

Then we create a button to select the file:

```
<input type="file" onChange={this.onFileChange} />
<button onClick={this.onFileUpload}>
     Upload
</button>
```

Which creates the button. It triggers a standard OS “File open…” dialogue. Once a file is selected and confirmed, we can get the file name from:

```
this.state.selectedFile.name
```

And the file itself from:

```
this.state.selectedFile
```

We can use the file name to give feedback to the user that the selection was successful.

4.8.2 CSV Processing

Since we don't plan on just hosting user CSV files in the database, the front end script will have to parse the file before sending it on.
We have a couple different options for parsing:

- Manually created function to read through the file and translate it directly to our needs.
- Use the jQuery-CSV library and running a second manual pass through the resulting array.
- Find a CSV parser native to React and running a second manual pass through the resulting array.

For first option, CSV files can get rather complicated, and we do not plan on having complicated formatting requirements for data dumps as that would work against the entire point – saving time by entering multiple sets of data at once. Manually creating a function to parse the file and account for all edge cases would take more time than it is worth and more expertise than we currently have.

For the second option, the jQuery function is easy to use and has been widely tested for several years. However, we run into a problem with React – React is a modular and live updating DOM controller, and any changes made to the DOM outside of it create conflicts and errors. There are some ways to get around this, but it is not clean and we would rather have a function much more native to React.

Thankfully, there exists a npm install that works with react for parsing CSV files – Papaparse (https://www.npmjs.com/package/react-papaparse). This function requires the CSV file to be streamed in (like when requesting something from the database) so we access the file through a `fetch()` function. Code has been adapted from https://react-papaparse.js.org/ [19]:

```
import React, { Component } from 'react'

import { CSVReader } from 'react-papaparse'

export default class CSVReader extends Component {
  handleOnDrop = (data) => {
    console.log('---------------------------')
    console.log(data)
    console.log('---------------------------')
  }
```

```
  handleOnError = (err, file, inputElem, reason) => {
    console.log(err)
  }

  handleOnRemoveFile = (data) => {
    console.log('---------------------------')
    console.log(data)
    console.log('---------------------------')
  }

  render() {
     return (
      <CSVReader
        onDrop={this.handleOnDrop}
        onError={this.handleOnError}
        addRemoveButton
        onRemoveFile={this.handleOnRemoveFile}
     >
        <span>Drop CSV file here or click to upload.</span>
      </CSVReader>
     )
  }
}
```

This replaces our old upload code, as Papaparse's CSVReader needs to be in the center of that process [19].

Figure 18. Example of Papaparse's CSV Reader

As a bonus, it looks really nice. We can use `this.setState({ selectedFile: data })` to pass this data outside the component.

Certain datapoints for papers are required, and certain ones are optional. We cannot guarantee that the file the user is uploading includes the required data, or that the CSV parser grabbed everything as the user intended them to be. To this end, we can manually run through the resulting data file, present it to the user in a form and have them verify the information grabbed and prompt for any missing information.

To assist with CSV parsing, we will require standardized header names and prompt the user if any important headers cannot be found. This should be a quick enough adjustment on the user end to justify parsing CSVs in the first place.

We can parse the standardized CSV form through a for loop and use the data to autofill a submission form like this:

```
for (var i=0; i<data[0].length; i++) {
     if (data[0][i] == "Key")
          this.keyPos = i;
     if (data[0][i] == "Publication")
          this.pubPos = i;
     . . .
}
```

Then we reorder the data into a way we can map to the submission form.

```
for (var i=1; i<data.length; i++){
this.reorder[i].append(data[i][this.keyPos]);
this.reorder[i].append(data[i][this.pubPos]);
. . .
}
```

4.8.3 Uploading the Data

Using the CSV, we can dynamically render the input forms using the map() function. Code has been adapted from https://stackoverflow.com/questions/34321128/render-array-of-inputs-in-react and https://www.pluralsight.com/guides/display-multidimensional-array-data-in-react [20,21]:

```
return (
<div>
     {this.reorder.map((items,index) => {
          return (
               <ol>
                    {items.map((subItems, sIndex) => {
```

```
                        return (
                             <input key={index+"-"+sIndex}
type="text" value={subItems}/>
                        })}
                </ol>
           );
      })}
</div>
);
```

Once these are rendered, the user can change the .value field and attempt to submit. The submission button will verify all required fields have a value, and then use that to construct json package for the database and finally submit.

4.9 GDPR and ePR compliance

4.9.1 Requirements

GDPR (General Data Protection Regulation) and ePR (ePrivacy Directive) are two laws in the EU which pertain heavily to cookie usage on the internet. Since a large, if not majority of our users will be from the EU, it's very important to remain compliant with these regulations.

Here are rules notable for us in GDPR and ePR:

· Cookies should last no longer than 12 months.

· Strictly necessary cookies – cookies essential for website functionality – don't require consent, however do require explicit explanation and justification to the user.

· Preference cookies – cookies which remember choices the user made in the past, such as preferred language. These require explicit explanation and consent from the user before they are ever placed on the machine.

· Consent must be documented and stored.

· Users must be allowed access to the website even if they refuse the use of certain cookies.

· It must be as easy to withdraw consent as it was to give consent.

4.9.2 Cookie Banner

The most common method of remaining compliant with these laws is through use of what is called a Cookie Banner.

For ease of compliance, and especially ease of storing user consent data, we will be using a plugin named Cookiebot. Since the concept of the website requires less than a hundred subpages, this should remain free for us to use for the foreseeable future. More information can be found on: https://www.cookiebot.com/en/cookie-banner/#:~:text=the%20questions%20above.-,What%20is%20a%20cookie%20banner%3F,before%20their%20data%20is%20processed.

To implement Cookiebot, we must place this in the HEAD tag of the website:

```
<script id="Cookiebot" src="https://consent.cookiebot.com/uc.js"
data-cbid="f15ea496-8c04-4465-b574-13a289f4294e" data-
blockingmode="auto" type="text/javascript"></script>
```

And then place this where we want the banner to appear:

```
<script id="CookieDeclaration"
src="https://consent.cookiebot.com/f15ea496-8c04-4465-b574-
13a289f4294e/cd.js" type="text/javascript" async></script>
```

Note that the cbid will change once we get an account setup for the university itself.
This method of implementation, unfortunately, isn't so simple with React. In order to assist us with this, we will be using an npm package called react-cookiebot (https://www.npmjs.com/package/react-cookiebot). In order to place this on the site, we include the following code on the highest level component [22]:

```
import React, { useState } from 'react';
import { Button, View, Text } from 'react-native';
const domainGroupId = 'f15ea496-8c04-4465-b574-13a289f4294e';

function App() {
const [hasCookieBot, setHasCookieBot] = useState(undefined);
return (
          <View style={{ flex: 1, alignItems: 'center' }}>
               <CookieBot domainGroupId={domainGroupId
language={this.language}} />
```

```
                <Text>Click to test Cookiebot</Text>
                <Button title="TEST" onPress={() =>
setHasCookieBot(!!document.querySelector('#CookieBot'))} />
                <Text style={{ color: 'red', marginVertical: 10
}}>
        {hasCookieBot && `Has CookieBot:
${JSON.stringify(hasCookieBot)}`}
                </Text>
          </View>
     );
}

<App />
```

The groupDomainId is equivalent to the cbid from before. As a bonus, this adds functionality to use our own language selection code to implement for the banner.[23]:

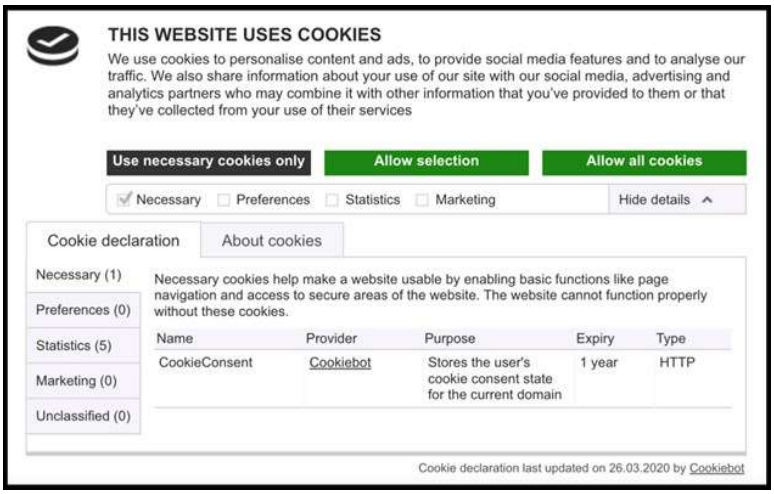


Figure 19. Example of a Cookiebot banner.

In the end, we ran into a single but important issue with Cookiebot:

In order to scan through a website's cookies to create an appropriate banner, the website needs to be live and active. Unfortunately, since our website was only located on a Chdr server and only the sponsor could make it go live, this ended up not being an option for us.

We've chosen to implement a simple banner as shown before, with accept or decline, and included an option on the Settings page to opt in or out at any point. In the future, the sponsor can choose to switch over to Cookiebot easily.

4.10 Storing Tags and Tag Hybrid Model

4.10.1 Tags Overview

The fundamental building blocks of this database will be how we store metadata on philosophical papers, tags about those papers, and the relationship between tags and papers. The most effective solution seems to be what some users call the “Toxi” solution [24].

Simply put, we keep a table for each tag (since tags are multilingual the structure of storing the tags is more complex than laid out here) in our database, a table for each paper, and then the relational table where each entity will have the primary keys of what tag and what paper it represents [25]. By describing our tables in this manner, we can easily find papers that correspond to our query by checking for entities in the relational table that have the queried tags and filter out the paper identification numbers that are in those found entities. The resulting list will just be the list of all papers that match the search criteria.

While this simplistic model does get the job done, we’ll see that if we stick with it, the querying process becomes significantly more convoluted and worsens the user experience. For example, say the user was querying a list of papers that had the following format:

```
19th century, China, Qin Dynasty
19th century, China, Zhou Dynasty
19th century, Germany
19th century, Country A
...
19th century, Country Z

19th century AND ((China AND (Qin OR Zhou)) OR Germany)
```

Figure 20. Example of when tag hierarchies are useful

If they wanted the result of their search query to contain only the first 3 results, the amount of detail that they would need to provide to their query is already quite involved. As the tags become more complex, the queries will continue to grow in complexity and make it more likely for the user to have multiple errors, thus significantly slowing their research process.

To remedy this situation, we will introduce the idea of a hierarchy of tags. This can simply be viewed as the equivalent to a file directory in the database. If users were able to organize the papers in the database as if they were in a file directory (the system that they currently have), then querying papers that are semi-related but only specific portions of that set will become significantly simpler.

As we can see, there are multiple benefits to keeping the university's old file system as well as benefits to organizing the tags in a loose manner that doesn't tie them to any hierarchy. But restricting the university to follow only one of these formats will mean that we either need to give up a lot of freedom (if we enforce a hierarchy) or force the user to have potentially complex queries when it comes to detailed research (if we enforce only the loose tag system). The solution will be to merge these two ideas and have a hybrid tagging system: one that allows loose tags and the ability to tag papers with custom hierarchies.

This hybrid system will allow the papers in the database to have an arbitrary number of basic tags. But additionally, the papers will be able to be tagged with a hierarchy of tags that will essentially be treated as separate tags. For example, a paper could have the tags of: **Racism**, **19th century**/**China**/**Qin Dynasty**. We can notice here that there are both single tags as well as a hierarchical tag. Now if a user were to query the hierarchy **19th century**, they would get our example paper as a result (as well as any other paper that has the 19th century hierarchy tag). A user would also be able to just query **China** as a single tag and as a result, they would get this example paper in their output.

As we can see, the user was able to tag their paper with a hierarchy containing China but even though they didn't specifically tag their paper with the singular tag of China, but tagging their paper with a hierarchy containing China will also simultaneously tag the paper with the singular version of the tag. By using this hybrid system, our users now have a large degree of freedom when it comes to how they wish to organize their papers. Maintaining the tags that are singular can still be done with the relational table we used initially, but maintaining hierarchies of tags and being able to build off of those hierarchies in the database will be significantly more challenging.

4.10.2 Hierarchy of Tags

Due to the need of the user to be able to give hierarchical tags to a philosophical paper, we need some sort of design structure that allows our database to handle a hierarchy of entities as a tag. With some basic experience in database design, it's easy to see that

tables are not explicitly designed to handle hierarchies; tables in a database are “flat” and have very direct relationships.
Nonetheless, we’ll need to come up with a way to incorporate a hierarchy into our tags in order to fully satisfy the user requirements. Currently, the competing models for a hierarchical table in a database are the Adjacency List Model and the Nested Set Model. We’ll give an overview of each model and then compare the two to see which one will best fit our needs.

4.10.3 Adjacency List Model

The key idea behind the Adjacency List Model is the inclusion of parent pointers for each entity in the table [26]. An entity’s ID parameter will simply be an integer and if an entity is not the root of a tree, then it will have a non-NULL parent pointer that tells us this entity’s parent.

This simple structure will define our tree but we need to ensure that we can perform some basic tree operations with this model.
Our main operations that we’ll need to perform on this tree are the following:

- Adding a new tag hierarchy to the tree
- Find the entity corresponding to a path in the tree.
- Removing a node from the tree. Removing may do 2 of the following:
    - Delete the entire subtree rooted at some node.
    - Delete the individual node and set the parent pointer of all its immediate children to be the original node’s parent.

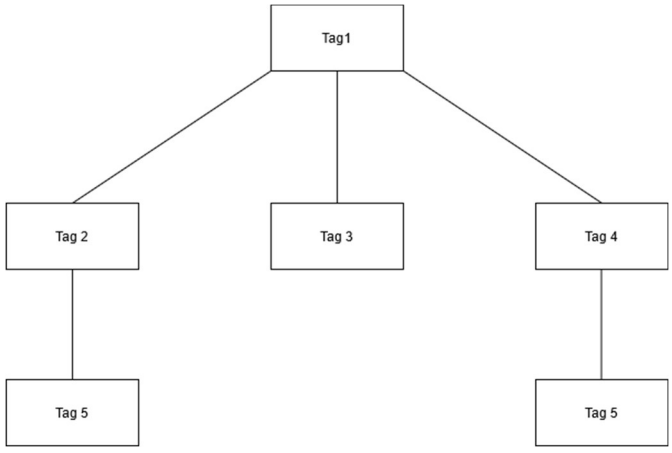


Figure 21. Depiction of Adjacency List Model as classic tree

The Adjacency List Model is simple in terms of implementing it in a database. However, the main issue lies in the complex SQL queries needed to handle these operations on a tree.

For example, querying the path in this tree will need to involve a JOIN statement for every depth level in the tree. Some sample code that allows us to retrieve an entity in the tree is as follows:

```
SELECT t1.name AS lev1, t2.name AS lev2, t3.name AS lev3, t4.name AS lev4
FROM category AS t1
LEFT JOIN category AS t2 ON t2.parent = t1.category_id
LEFT JOIN category AS t3 ON t3.parent = t2.category_id
LEFT JOIN category AS t4 ON t4.parent = t3.category_id
WHERE t1.name = 'PARENT' AND t4.name = 'CHILD';
```

Figure 22. SQL to show Adjacency List Model operations

As we can see, the main drawback to performance here will be the JOIN statements that are required for every level in the hierarchy. As the philosophers have more complex hierarchies that they're tagging their papers with, the performance will inevitably worsen. Even without the worsening performance, the complexity of the SQL queries as the depth level of the tree increases. So, while the Adjacency List Model is simple in its structure, it most likely won't be able to keep up with the scale of Entangled Philosophies.

4.10.4 Nested Set Model

The Nested Set Model takes a different approach to representing this tree structure that doesn't involve looking at it as a standard graph theory tree. Instead, we'll look at the structure as a group of nested containers.

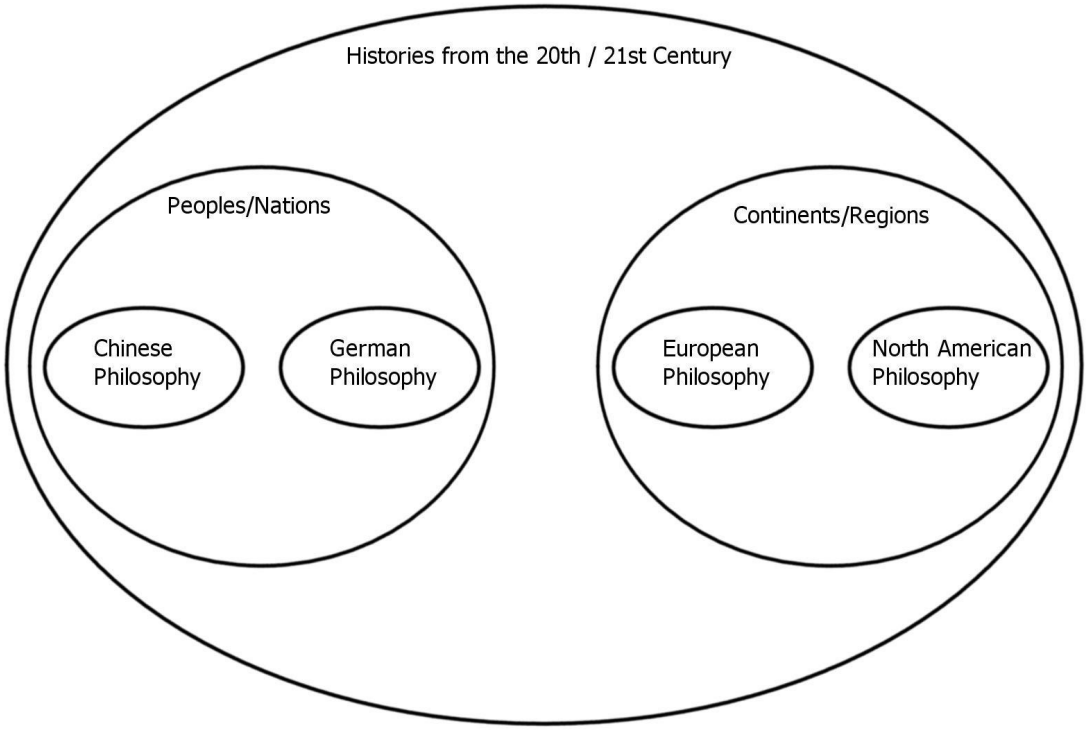


Figure 23. Example of nested hierarchies in Nested Set Model

We can represent this in our database by having each entity hold “left” and “right” pointer values. These left and right values are equivalent to a modified preorder tree traversal. If a node **A** exists in the subtree of another node **B** then the left and right values of the node **A** will contain the left and right values of the node **B**.

Using this representation of a tree, adding new nodes is equivalent to just adding 2 to the left and right pointer values for all nodes that have pointer values greater than the newly created node. Finding specific nodes and paths in this tree are also relatively simple SQL queries that don’t involve a JOIN statement for every depth level in the tree.

For example, if we wanted to find a single path ending at a specific node in this tree, we would simply need to query all entities in our table whose left and right pointers contain our chosen node. The code would look something like:

```
1. SELECT parent.name
2. FROM paths AS node, paths AS parent
3. WHERE node.lft BETWEEN parent.lft AND parent.rgt AND node.name = 'node_name'
4. ORDER BY parent.lft;
```

Figure 24. SQL example for Nested Set Model operations

As we can see, we'll begin by looking over two items from the paths table ("node" and "parent") where we will ensure that "node" will be a child of "parent" by making sure it lies between the parent's left and right pointer values. We can alternatively view these left and right pointer values as equivalent to flattening the tree in the database. By assigning these values, we essentially compress our tree down into a linear array which ends up making queries on the tree significantly simpler to translate into SQL.

Given the large amount of queries (as well as the potentially large depth in this tree structure), this model provides much simpler SQL queries that are a lot more efficient. The only remaining issue is the ability to delete nodes in these trees. Since this hierarchy is essentially equivalent to file storage, deleting a node in the tree can correspond to deleting a directory and so the removal of a node should constitute the removal of the entire subtree. Note that this deletion of a node in a hierarchical path does **not** correspond to deleting that tag from the database. But if a tag is deleted from the database, all paths in this tree that contain this tag should have their subtrees removed.

To summarize the importance of being able to represent a tree hierarchy of tags, the admins of Entangled Philosophies should be able to tag papers with a hierarchy of tags instead of only singular tags in order to expedite their queries. By tagging a paper with a specified hierarchy of tags, it allows the paper to appear as a result of any query that involves a subpath of its hierarchy but the paper can also appear as a result if a singular tag is queried that is involved with its hierarchy. The hierarchy system, when combined with the singular tag system, can make the queries a lot more flexible and specific.

4.11 User Created Tags

Another feature of Entangled Philosophies is the ability for all users to create their own custom tags that are available to only that specific user. You can almost think of this as something like user notes that are managed as tags in the database. When users create their own tags, they'll have all the functionality that regular admin tags have. That functionality includes making their own singular tags for papers and even creating their own hierarchies that mix and match between the admin-created tags and the user's own personal tags.

Implementing this in a database seems to have its own set of challenges. A possible approach would be to create a separate table in the database for tag relations between a paper and a user created tag. By organizing the tags this way, we can provide a clear divide between what tags should be available for a user to see by simply checking the separated tag table to see if they own the tag they're referencing.

But after further consideration, we can see that the previously described solution to this requirement doesn't actually need a separate table for user defined tags. Instead what we can do is add another field to the contents of a tag (and its multiple translations) that represents the owner of that tag. Owners of a tag will be classified into two states: admin-made or user-made. If a tag is created by an admin, this field flag will be set to true and thus will be available to all users regardless of who is making the query. However if a tag is user-made, then a special check will be made when that tag is being requested to be a part of a query. If the user making a query with a custom tag doesn't actually own the tag, that query wouldn't be accepted as valid.

And with this, users can successfully create and delete their own versions of tags and fully customize their researching experience. It should be noted that users will not be allowed to create custom tags that already exist in the database (regardless of language). But because our tagging system should also support a hierarchical structure of tags, we need to ensure that these custom tags do not interfere with the queries and storage of our hierarchies.

Recall that the Nested Set Model will allow us to efficiently create, maintain, and build off of a defined hierarchy of tags that can then be queried. When a user queries a hierarchy, every paper that falls within that hierarchy should be returned as a query result. However, if a user creates a custom hierarchy that initially starts with some public tags but then begins to use their own custom tags, our query to the nested set model will need to be able to filter out certain subtrees of the hierarchy depending on which user is querying the system. To clarify, consider the situation below:

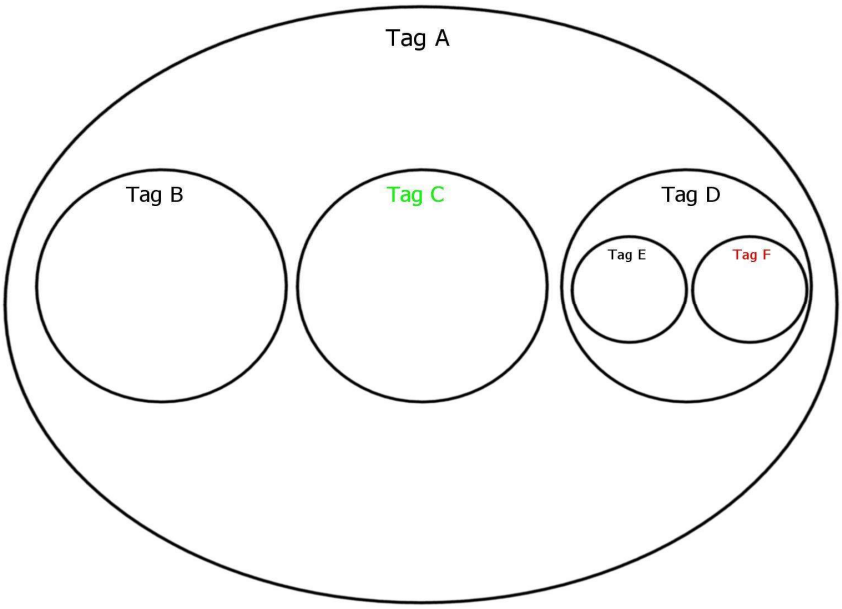


Figure 25. Nested Set Model with different user-created hierarchies

If we consider the hierarchies colored in red/green to be created by users (i.e. not public) but we decide that we want to intermingle user-specific hierarchies and admin-made hierarchies, then making a query with on the hierarchy “Tag A” will get us all papers stored in hierarchies that are labelled in black. When it comes to dealing with the singular tag entities where some are private, filtering out inaccessible tags is reasonably simple since we’ll just have an extra part to the SQL query that ensures we only pick out admin tags or tags that belong to the user making the query.

But when it comes to querying the hierarchy structure it could potentially be inefficient if we look over every single hierarchy in the database. Since there may be multiple hierarchy tags where we know the topmost level is a private tag, we can determine beforehand that any hierarchy tags nested inside this private hierarchy will also automatically be private. It’s evident that arbitrarily looking over each hierarchy (without any sort of order to it) to see if it’s valid/invalid can inevitably lead to performance issues as more users begin to make custom tags.

We can remedy this issue by processing the hierarchies in a query in an efficient order, namely in DFS order. If we begin a depth first search traversal from the node representing the queried hierarchy, we can continue calling this recursive procedure on a hierarchy’s children until we see a tag that is not accessible by the user making the query. Once we

see this forbidden hierarchy, we can immediately cut out of our search since we know any further processed hierarchy won't be valid.

Unfortunately, doing a DFS in this somewhat complicated manner in an SQL query can be quite challenging. There is an elegant solution around it however that involves adding an extra field to each hierarchical tag. We can notice that in a private hierarchical tag, there is exactly one owner (if there are private tags in a hierarchy then the only user that could have made this private hierarchy is the owner of the private tags). So, when a hierarchical tag is added to the database, we can just update this additional field with the owner. Then querying all subtrees of a hierarchy (but making sure to ignore private hierarchies) can be expressed with this SQL query:

```
1. SELECT paper.id
2. FROM paths AS node, paths AS root, papers AS paper, hierarchyRelation AS relation
3. WHERE root.id == subpath_id AND (node.id == 0 OR node.id == user_id)
4.  AND node.lft BETWEEN root.lft AND root.rgt
5.  AND relation.tag == node.id AND relation.paper == paper.id
6. ORDER BY paper.id
```

Figure 26. SQL query to get all sub-hierarchies that belong to the user

While it is a crowded query, we can see that we'll efficiently pick out any sub hierarchy of the requested hierarchy (we'll inject the id of the subpath directly into the query but in this snipper it's known as "subpath_id"). The query will start off by ensuring that the node that we're considering is either an admin-made node (denoted by the id 0) or made by the user that matches the injected user id (we call it "user_id"). As long as those conditions are satisfied, we'll then ensure that the node we're considering is in the root's range which will qualify as a valid hierarchy. Note that this query is a very basic access query to the hierarchical structure and that these SQL statements will be even more complex when we begin excluding subtrees or pulling data from multiple subtrees.

4.12 Multilingual Database Design

A major requirement of this service is the removal of language barriers between tags. This means that a singular tag (for example, "China") should be recognizable to the database in multiple languages and if 2 separate papers are given the same tag but in different languages, the database should treat it as if those 2 papers were given the same tag.

There are many different ways to go about making the tags in the database multilingual (some more efficient than others). An initial idea would be to have the user dynamically update all possible translations of an individual tag (let's call it tag A) in the tag edit page. Then, with the addition of a new tag translation, we can traverse the database and add this new version of tag A to each paper that had any previously existing translation of tag A. Now, all papers that have any version of tag A will actually be guaranteed to have all versions of tag A.

While the previously described idea does indeed work, it is incredibly inefficient. Adding a new version of a tag is an expensive operation because it now needs to traverse the entire database for all papers that contain the updated tag and then update each individual paper by adding the new version as a tag. Not only is traversing a database containing tens of thousands of entries for each new addition very slow, each paper will very quickly grow to have hundreds of tags depending on how many translations exist.

To get around this, we'll build a pointer system into our database. We'll have a master table that stores the actual entry for a tag (it'll simply have a tag id and no associated language with it). Then with a separate table, we'll store all possible translations for all tags in the database. Each entry in this translation table will have the literal string representing the word that users will use to search, an associated language with it, and then a parent id for what tag it actually corresponds to in the master table. Our database schema would look something like this:

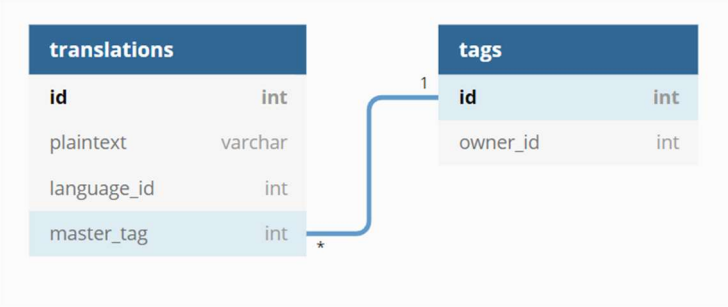


Figure 27. Example database schema for multi-language tags

With this solution idea, queries can be made using any language (since we'll allow the users to pick any tag that exists in the translation table) but when the query is passed over to the api endpoints, each tag will be converted into its canonical form (without a language association) which will then be used to query the database. Going with this

solution idea has many benefits since now users will be able to add an arbitrary number of translations without any performance issues as well as make sure that the tag tables do not blow up in size as multiple languages begin to get used.

4.13 Storing User Queries/Results

A key feature of the querying system in Entangled Philosophies will be that users will be able to save their queries and bring them back at arbitrary times to edit and refine their queries as well as re-run their queries to get updated results. Saving queries in the database and reloading them into the front end (when a user requests it) does pose some challenges depending on how we implement it.

To begin, it should be clarified that when a query is sent to the database, what will initially happen is that the tag values that were inputted by the user will be converted to their corresponding identifier values. There are actually two steps to this process since we will first need to reference the database to figure out what tag is being referenced (since we need to convert the corresponding plaintext into an entry in our translation table). Once that is known, we can follow the pointer back to the master tag table whose identifier will then be replaced in the query.

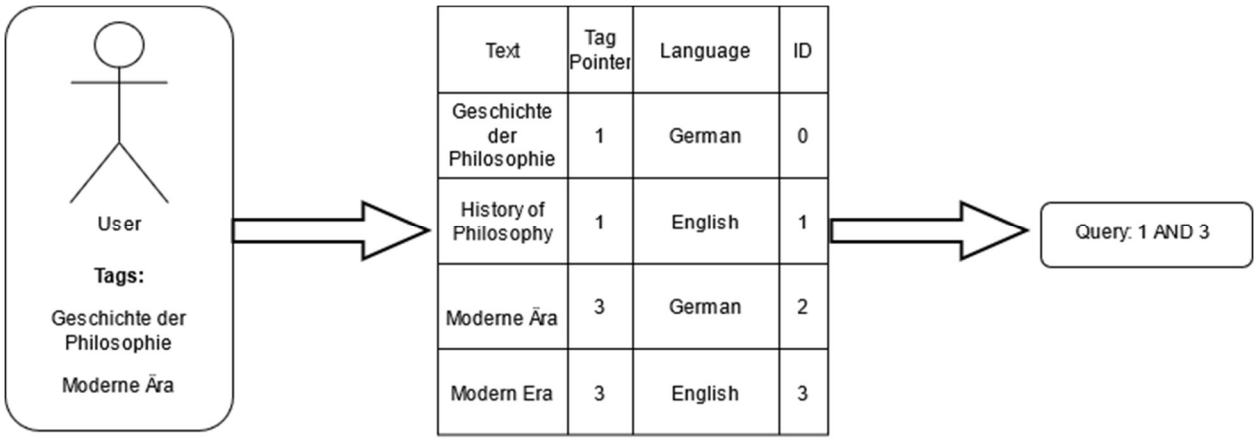


Figure 28. Visualization of converting user query into database query

Now this modified query is what is then parsed to create an SQL query but the query that we save that the user can then retrieve is actually the query that has yet to point back to the master tag. The reason the query we send to the database is actually slightly different from the query that will be saved for the user is that once a tag is translated from its

respective language into a database entry, there is no way to travel back to the original translation. To clarify, if we begin with a tag in German but point it back to the tag that it represents, there is no back edge to let us know that this tag was originally German. So, we have to make sure that the copy of the user query that we save actually retains its original languages so that the user will be able to pick up and edit their query where they left it off.

Now that we've clarified what query we're going to store, we need to consider how it actually gets stored in the database. The simplest approach would be to just store the plaintext of the query in a database table. This table will have 3 fields that describe the owner of the query, the name of the query (defined by the user) and then just a TEXT field that holds the actual query. Visually it will look like:

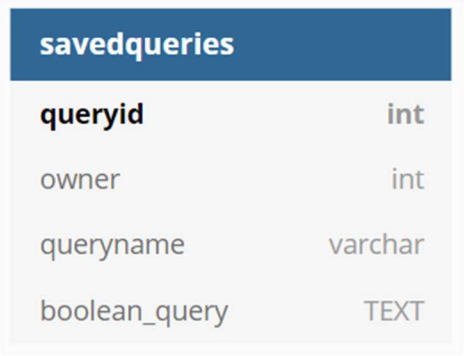


Figure 29. Example database schema for saved queries.

With this, we can easily store and access any queries made by a user and reload it into the front end for the user to edit. Another way to view this query storage process is to notice that it's possible to build a parse tree out of the boolean expression. Since we already have a way to store trees in a database (since we'll be using the Nested Set Model), we might consider storing the parse tree in our database and recovering it. While building this parse tree may actually be more complex for storing queries, it's very possible that we will be writing code to create a parse tree anyways when it comes to evaluating queries. In that regard, while it may not seem to be the most practical idea, converting a query into a tree and then reverting it when the user requests is also a valid solution path (although most likely not what we'll be going with).

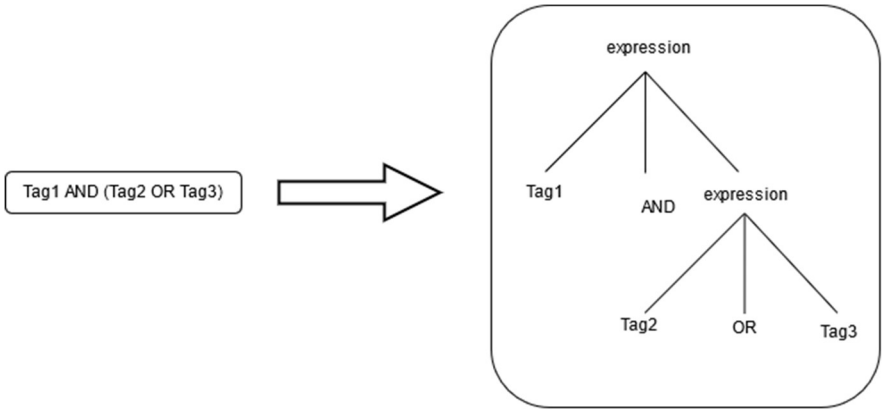


Figure 30. Example of what our parse trees would look like

Now that we can save queries, we also need to explain how to save the results of a query for the user. The simple approach becomes very easy now that we can save user queries: all we need to do to recover the results of the query is send the query to the database again and display those results. This approach requires little to no extra effort and doesn't place any extra strain on the database since we don't need to save any more information.

However, we can see that there may be some issues when we save results this way. If the user is only interested in the exact set of papers they see when they make the initial query, storing the results this way is problematic because rerunning their query at a later point may not return the same set of papers (i.e. if a new paper has been added since they've made their query). So, we will have to put in a little more work to restore the exact set of papers that the user has seen.

The first solution that was considered was creating something similar to the tag table but instead for relating query ids and paper ids. The created table will just store the relationship between a query and a paper and when a user wishes to recover the results of the query, we'll refer to this relational table. The memory issues with this will very quickly become a major concern. Since this database is intended to hold up to tens of thousands of records, allowing users to add up to ten thousand different table entries for each query can make the site vulnerable.

Another similar, but also vulnerable, solution idea would be to replace this idea of creating yet another relational table and make do with what we already have in the database. To clarify, instead of creating a table that will relate query ids and paper ids, we can instead just use the tagging system to save query results. By reserving a certain set of tag names (something the user wouldn't use) to be used specifically for saving queries, we can implement a batch tagging procedure that adds a specific tag to all papers satisfying some criteria. When a user intends to save a query, they simply provide the website a name

and that will prompt us to tag all papers in their results with this reserved tag name corresponding to their query. Once complete, whenever a user wants to recover the snapshot of their query results, the database will simply query for papers that have their special tag associated with a query. This solution idea is clever since we can take the pieces of the site built so far and use them to save query results.

But even with both of these proposed ideas, we run into the same memory issues: a malicious user can quickly fill up space for the database by making general queries and saving the thousands of results. In the case that we proceed with one of the previously described solution ideas, it would make sense to create a cap for each user that limits the total number of paper entries they could save. Admins would be able to manually adjust the cap for a user in case their needs are legitimate. This would address our concerns of malicious users but also restrict the freedom of the user quite a bit.

However, there exists another solution to allowing the user to take a snapshot of the database. The main concern at the moment is that simply re-querying the database with a user's saved query doesn't accurately reflect the results they saw since the database may include more papers that match their criteria. Assuming that they would explicitly want to ignore these newly added papers, it almost seems like we need to manually save whatever results they see in order to reliably recover their results. But there exists an even simpler solution that involves recording the date each paper is added and the date a query is made. The date and time a paper was added is simply a piece of metadata that we'll be tracking anyways, and we can use the date to simulate taking a snapshot of the user query's results.

Since the user query is already being saved, all we simply need to do to recover their results is to add a condition to their query that states that any returned paper should have been added at a date **before** they created the query. This way, the user will only see the query results that they had seen before, ignoring any newly added papers. So, we've arrived at a simple solution for simulating a snapshot of the database without expanding the memory consumption of the service.

Up until now, we've only been discussing solutions to avoid seeing newly added papers that the user may not want to see. The opposite situation, however, is significantly more challenging to deal with: what happens with papers and tags that are deleted from the database after the user makes their query? When it comes to dealing with removed paper entries from the database, there is a reasonably simple solution: adding a field to each paper that represents whether or not it's active in the database. This feature of deactivating a paper is useful in multiple ways but it'll also give us the admins a chance

to revert a deletion as well as let us inform a user that a paper saved by them has been removed.

A possible way of communicating to the user that a paper that they had saved was now in fact deleted, is to essentially just perform their query twice: one that tracks active papers and one that specifically tracks inactive papers. We would then present both to the user (most likely the inactive papers being listed at the bottom and its bibliographic information unviewable for the user). While inconvenient for the user, this feature allows the proper sequence of events to be explained to the user and thus open up the opportunity for them to contact an admin to reactivate said paper.

The most challenging aspect of saving user queries/query results will be merged/deleted tags. When tags are merged, it essentially means that the 2 tags represent the same content and so one redundant tag **A** should be "removed" from the database and all papers that were previously tagged with **A** should have that tag replaced with the real tag. Addressing the situation for when a user's query contained a tag that was merged is reasonably simple: for each merged tag, maintain in the database what tag it was merged into. With this information, we can determine the true identity of an old tag and modify the user's query to contain the accurate representation of their tag.

There are some concerns that need to be addressed with this solution for dealing with tags. Namely, what happens when there are multiple chain merges that need to be tracked? To clarify, if the tag **A** merges into tag **B** but eventually tag **B** merges into tag **C**, any saved queries that contain the tags **A** or **B** should be replaced with **C**. However if our records for tag merges simply state "**A** merged into **B**" then we'll end up replacing all occurrences of tag **A** with tag **B** who is similarly an inactive tag.

The structure of this problem resembles the Disjoint Set data structure. A Disjoint Set is a data structure that essentially maintains the relationship between a series of elements in a set. The relationship described in the Disjoint Set structure is transitive: if A is the same as B and B is the same as C, then A is the same as C also. In this structure, when 2 or more elements are the same, only one of them will be considered the representative of the entire set. It's sufficient to ignore everything except the representative of the set due to the nature of transitivity we've described. The similarity to the tag merging issue we laid out is very apparent: when a tag **A** is merged into a tag **B**, we're essentially saying **A** and **B** are the same and **B** becomes the representative of that set.

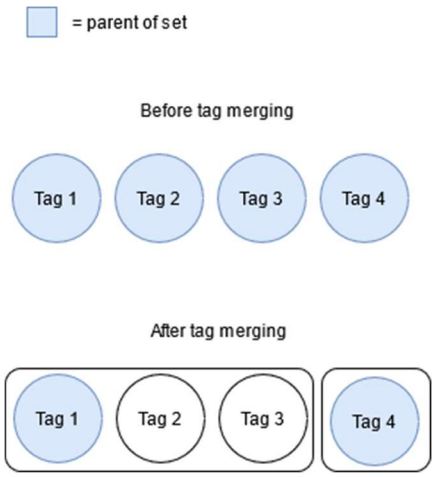


Figure 31. Visual depiction of Disjoint Sets for tags

So, we can represent merged tags using the disjoint find data structure (maintaining that in a database is simple) and edit queries that have inactive tags. There is still an issue that some might notice depending on the scale of the database. If we have a long chain of merged tags, replacing a single instance of that tag in a query will result in us having to traverse the chain every time and then lead to performance issues. We can refer to the Disjoint Set data structure yet again and make us of its path compression technique.

Essentially, when one element is merged into another (and is no longer the representative of its own group), its only function from this point on in the data structure is to redirect its children over to the group representative. When an element of the set **A** follows its parent links all the way up to the group representative, we can replace its parent to now be the current representative of the set instead of simply **A** (which we now know is not the representative of the set). So, what we're tracking with this parent pointer in the data structure is what this element currently believes is the representative of the set. If after following a node's parent links, we find out that the node did not have an accurate pointer to the real representative, we'll update it's pointer to the current representative. Doing this every time we query the representative of a node, we can ensure that the path to the representative is flattened and we don't make any useless link traversals (when we already know the outcome). This simple optimization will drastically improve our performance when we find out that an inactive tag needs to be replaced with the representative.

Now that we've dealt with handling saved queries that have inactive tags, we need to finally deal with the saved queries that have deleted tags. This issue is entirely different from everything we've discussed previously. Handling deleted papers wasn't too difficult since we can just set an inactive flag. But because the queries are so heavily tied in

recovering the saved set of papers, we can't simply set a flag and present it to the user. While there does exist a work around for dealing with deleted tags, we are currently under the assumption that if an admin deletes a tag, they would like that tag to no longer be queryable by anyone in the database and so we should notify the user that they're using a dead tag and not allow that query to be submitted to the backend for evaluation. If it's requested that we should still store information about removed tags and allow users to query it, it is possible to get that done by just adding extra flags to the relationships between tags and papers.

The last topic that needs to be addressed when it comes to dealing with deleted/merged tags is whether we update these saved queries all at once or on a need-by-need basis. When tags are changed, all relevant papers in the database are changed at the same time (our papers need to be consistent with the tags in the database). There is no choice in that regard but when it comes to saved queries, we don't necessarily need to process all saved queries and demand that the user edit them. Instead, we can simply mark a query as problematic after updating tags and once the user attempts to edit/search with that saved query, we can alert them that their query is currently invalid and needs to be changed. This way, we can ensure bad saved queries are known by the database but it won't bug the user about it unless necessary.

4.14 Database Management System

4.15.1 Choosing the right Database

Figure 32. MySQL logo

When choosing our Database Management System for Entangled Philosophies, we decided to go with a MySQL database. We chose a MySQL database because we needed a powerful, secure, stable, and easy to use relational database. We needed a relational database, so that we could put together complicated queries which had multiple logical operators (AND, OR, NOT, etc.). These queries would search through philosopher's papers for specific tags and return all kinds of metadata about the papers found that were related to the query. MySQL also runs on a wide range of operating systems (UNIX or Linux, Microsoft Windows, Apple Mac OS X) and others. We do not know which kind of operating system our users will be on, as well as the developers for Entangled Philosophies have a variety of different operating systems. This made MySQL perfect work well for Entangles Philosophies, since it runs on so many operating systems and has for many years.

MySQL supports standard SQL (Structured Query Language) which was created in 1979. SQL is a proven relational database programming language, which is easy to learn and can be very powerful when implemented correctly.

MySQL can be used as a great solution for both small and large applications. Here are a few large applications that are built with a MySQL database.

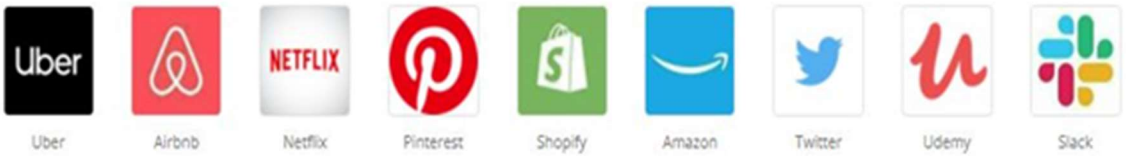


Figure 33. App examples that use a MySQL database

Since large companies like Uber, Netflix, Shopify, and Amazon use MySQL, we felt confident to use it as our Database Management System. Many of these multibillion-dollar companies have faith in using MySQL, since it is developed and distributed by one of the leading companies in the world for innovation, Oracle Corporation.

Figure 34. MySQL is design and developed by ORACLE

4.14.2 Structure of MySQL table

MySQL databases store data into tables like other relational databases. A table is a collection of related data and it is divided into rows and columns. Each row in a table represents a data record that is inherently connected to each other such as information related to a particular user, whereas each column represents a specific field such as id, first_name, last_name, email, etc.

The structure of a simple MySQL table that contains a user's general information may look something like this:

id	first_name	last_name	phone_number	email	password	is_admin	is_verified	created_at
1	Mitch	Wise	3212221029	mitchxwise@gmail.com	P@55sw0rd	1	1	2021-06-05 18:16:02
2	Joe	Smith	7049249304	joesmith43@yahoo.com	jj9dSsld#12	0	1	2021-04-05 20:26:01
3	Josh	Darth	3025559242	jdarth22@gmail.com	djw$j8OsZ	0	1	2020-02-03 07:22:06
4	Logan	Foster	2019450623	Loagdog@gmail.com	9dkf2D*&D	0	1	2021-11-02 12:04:11
5	Kevin	Gene	3212063233	KevinG@hotmail.com	Jojo123#jd	0	1	2020-10-06 08:29:09
6	Kaitlen	Jefferson	9540234571	CatKateJ@gmail.com	KJ203$#okl	0	1	2021-04-05 20:26:02
7	Scott	James	3212563278	James_Scott@gmail.com	Eastman32!	0	0	2021-04-05 20:26:02
8	Sarah	Conner	3219462929	Sarahhh32@yahoo.com	SnnerCon^1	0	1	2021-04-05 20:26:02

Figure 35. MySQL user table example

Here is another example of a MySQL table that contains metadata from philosophers' papers:

Key	Item Type	Publication Year	Author	Title
6FVFYQC7	webpage	2019	Pressestelle, Dr Jörg Diederich	Deutsche Philosophie
4TJ7YEU3	webpage	2020		ジーベルト氏最近独逸哲学史(名著綱要文学教育秘 ; [15]) | 書誌詳細 | 国立国会図書館オンライン
3UWUPRFM	book	2006		中国哲学史
EJPH5R4W	book	1984	コプルストン[著;小坂国継 [ほか]訳	ドイツ観念論の哲学
NLN7KMHB	book	1984	Copleston, Frederick C.	ドイツ観念論の哲学
8AGQAFBS	book	1934	Windelband, Wilhelm	十九世紀の独逸哲学
E2AW2IEC	book	1925	Windelband, Wilhelm	近代独逸人の精神生活の哲学
ZKSNUARS	book	1901	桑木, 厳翼; Siebert, Otto	ジーベルト氏最近独逸哲学史
VYAZUK7R	book	1932	Moog, Willy	ヘーゲル学派史

Figure 36. MySQL paper table example

As you can see from the tables above, MySQL tables organize data in a way that is easy to read. This allows developers to analyze data and make educated decisions on what code needs implemented to produce the desired results. There are many ways to sort data as well in the table, whether you want something in order by Author's, Item Types, etc. You Can also organize columns in ascending or descending order, that way you can see the most recent or older data first.

4.14.3 MySQL code examples

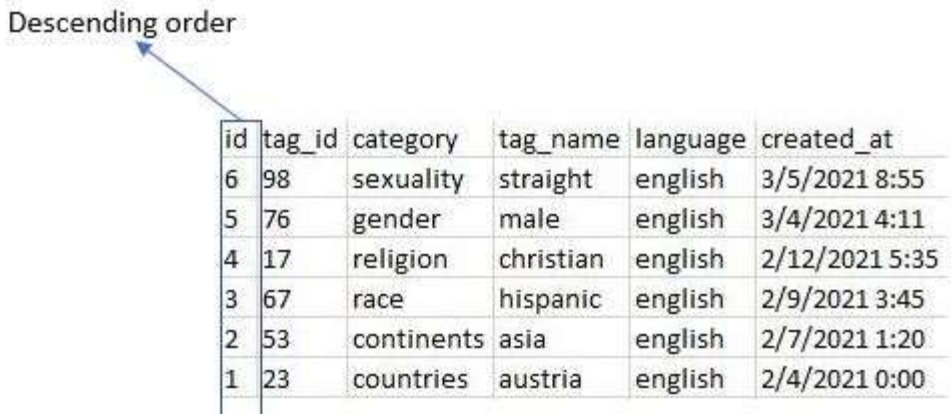
For Entangled Philosophies we had to make sure to choose a query language that had prebuilt functions that could accommodate the query at hand. Such as returning entire tables, parts of tales, sorting columns by ascending and descending order, as well as returning data in order by a specific column.

Below are some proven MySQL code snippets to show that MySQL can accomplish just that.

Here is a code example of retrieving data in descending order using MySQL:

```
SELECT * FROM tags ORDER BY id DESC;
```

Below are the results of the tags table in descending order by id:



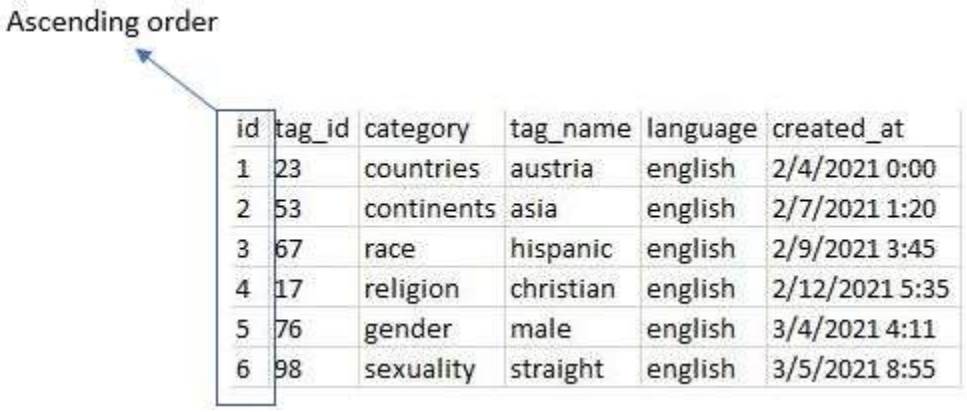
id	tag_id	category	tag_name	language	created_at
6	98	sexuality	straight	english	3/5/2021 8:55
5	76	gender	male	english	3/4/2021 4:11
4	17	religion	christian	english	2/12/2021 5:35
3	67	race	hispanic	english	2/9/2021 3:45
2	53	continents	asia	english	2/7/2021 1:20
1	23	countries	austria	english	2/4/2021 0:00

Figure 37. MySQL tags table in descending order

The same query in ascending order:

SELECT * FROM tags ORDER BY id ASC

Below are the results of the tags table in ascending order by id:



id	tag_id	category	tag_name	language	created_at
1	23	countries	austria	english	2/4/2021 0:00
2	53	continents	asia	english	2/7/2021 1:20
3	67	race	hispanic	english	2/9/2021 3:45
4	17	religion	christian	english	2/12/2021 5:35
5	76	gender	male	english	3/4/2021 4:11
6	98	sexuality	straight	english	3/5/2021 8:55

Figure 38. MySQL tags table in ascending order

Let’s say that you wanted to retrieve an entire table with MySQL, this can easily be done by doing the following:
SELECT * FROM `tags`;

All results from the tags table will be returned.

id	tag_id	category	tag_name	language	created_at
1	23	countries	austria	english	2/4/2021 0:00
2	53	continents	asia	english	2/7/2021 1:20
3	67	race	hispanic	english	2/9/2021 3:45
4	17	religion	christian	english	2/12/2021 5:35
5	76	gender	male	english	3/4/2021 4:11
6	98	sexuality	straight	english	3/5/2021 8:55
7	85	education	college	english	3/9/2021 5:35
8	53	continents	asia	english	4/1/2021 2:54
9	7	gender	straight	english	4/3/2021 4:44
10	4	religion	buddhism	english	4/8/2021 7:09

Figure 39. MySQL tags table returning all fields

Suppose we want to get a list that groups the query result set by using the tag_name field, we would use the script shown below.

```
SELECT * FROM `tags` GROUP BY `tag_name`;
```

Query returning all the data from the tags table grouped by tag_name.

id	tag_id	category	tag_name	language	created_at
2	53	continents	asia	english	2/7/2021 1:20
8	53	continents	asia	english	4/1/2021 2:54
1	23	countries	austria	english	2/4/2021 0:00
10	4	religion	buddhism	english	4/8/2021 7:09
4	17	religion	christian	english	2/12/2021 5:35
7	85	education	college	english	3/9/2021 5:35
3	67	race	hispanic	english	2/9/2021 3:45
5	76	gender	male	english	3/4/2021 4:11
6	98	sexuality	straight	english	3/5/2021 8:55
9	7	gender	straight	english	4/3/2021 4:44

Figure 40. MySQL tags table returning all fields, grouped by tag_name

4.15 Database Diagram

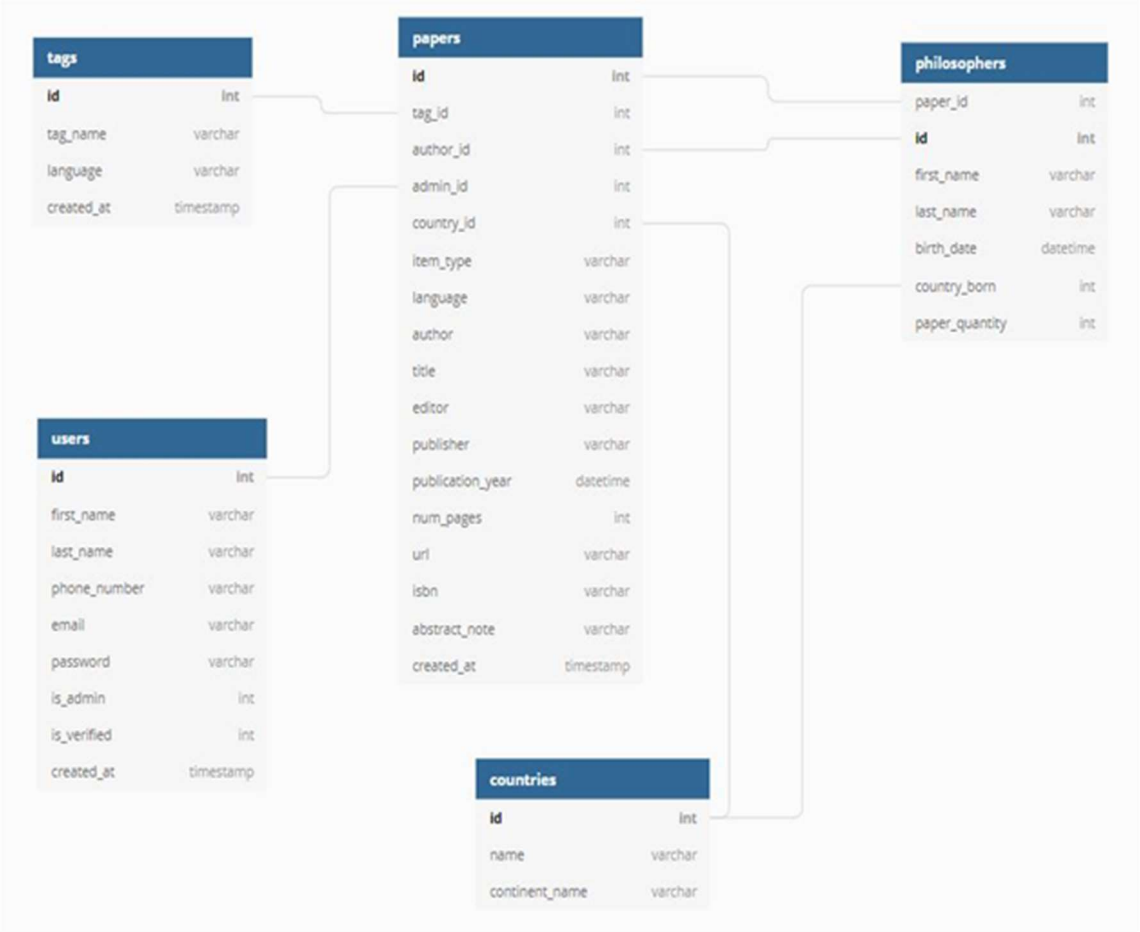


Figure 41. Database Diagram

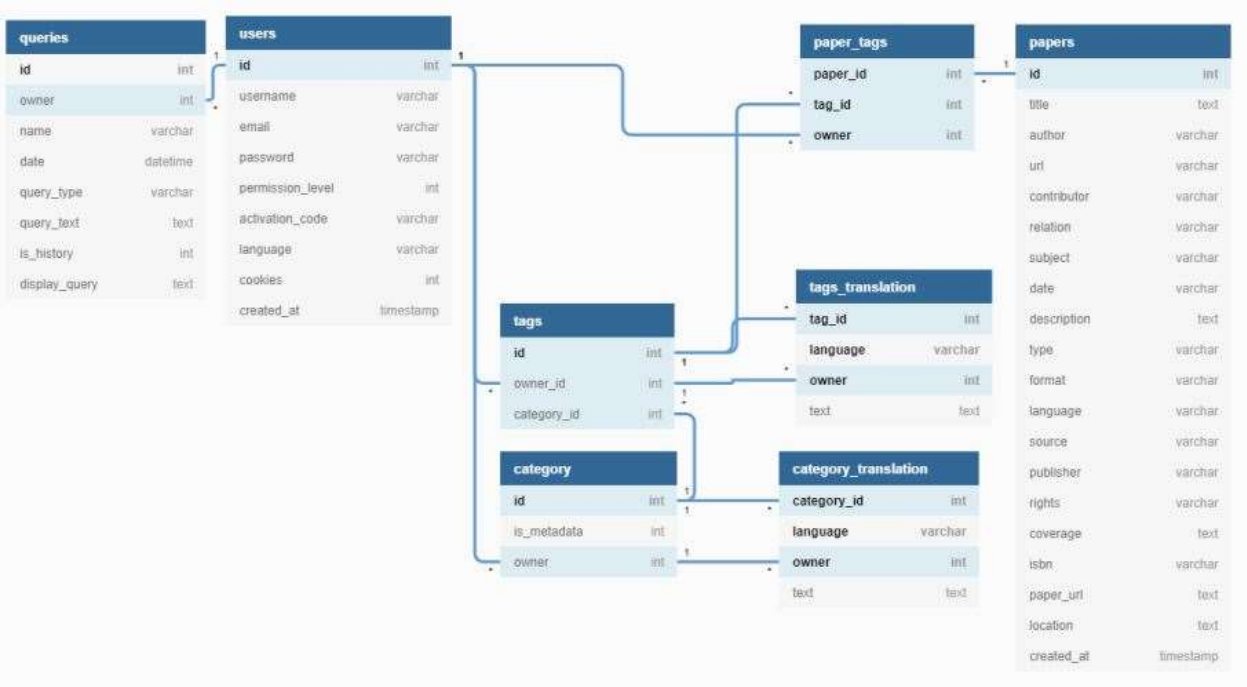
4.15.1 Database Diagram Overview

Entangled Philosophies database has 5 primary tables (Tags, Users, Papers, Countries and Philosophers). When coming up with a Database Diagram for Entangled Philosophies. We had to come up with a design that would allow users to perform complex queries. To perform sophisticated queries, we figured all 5 tables should be interconnected by the Papers table. We chose for the papers table to be the primary table that connects all the id's, because Users upload philosophers' papers onto the Entangled Philosophies website. Therefore, the Papers table was an easy decision for us to choose, to support all the relationships possible for all the tables. Each table has its own unique id as a primary key and the Paper's table has the foreign key which matches up with the corresponding primary key. This database design will allow users to make difficult queries and be able to access any table through joining on certain criteria.

4.15.2 Database Diagram Research Changes

After completing more research, the Entangled Philosophies Database Diagram has gone through some changes. We have decided to reduce the number of primary tables that are going to be used. We were originally using 5 primary tables (Tags, Users, Papers, Countries and Philosophers). Although, we have decided that it is going to be more efficient to remove the Countries table and Philosophers table. This will leave us with 3 primary tables (Tags, Users and Papers). Having fewer tables makes things easier when querying and storing data, as the variables from the Countries table and Philosophers table can be combined into the Papers table. We have also made some modifications to the database so that it complies to the Dublin Core metadata standard.

4.15.3 Database Diagram Final Changes



For the Final Database Diagram, we ended up using 8 tables, rather than the 5 tables we planned to use, during our research. The 8 tables that make up Entangled Philosophies database are (queries, users, tags, category, paper_tags, tags_translation, category_translation and papers). After trying to implement our researched design, we found out that this final design works much better, since there are a lot of moving parts with the tags/category translations, as well as saved queries. Most of our overall database

design stayed the same. The Countries and Philosophies table's were removed from our researched design. For our FInal Database Design, we added (queries, category, category_translation, paper_tags and tag_translation tables). With these newly added tables we can efficiently fulfill our goals and requirements for Entangled Philosophies.

4.16 Papers Table Database Diagram

Figure 42: Database table for storing papers

4.16.1 Papers Table Variable Description

The Papers tables fields will be generated from papers being uploaded by the admin users onto the Entangled Philosophies website. The Primary Key "id" is going to be a unique id that differentiates each paper from one another. This way the process of finding details about a philosopher's papers, will be much easier, by using the paper's primary key to pull out specific characteristics about the philosopher that wrote the paper or the paper itself. Each paper will also have a "tag_id", which is a foreign key for the Tags table's primary key "id". The "tag_id" within each paper, will give us a way to join the Papers table and the Tags table together to find out which tags are associated with each paper. One of the main goals for Entangled Philosophies is for users to be able to use

complex queries to find out details about philosophers that we would have never been able to know otherwise. With us being able to associate philosophers' papers with multiple tags, gives researchers a tool to learn more about these amazing philosophers.

The Papers table "author_id" is a foreign key, which matches up to the Philosophers table's primary key "id". With this relationship we will be able to tie which authors are connected to each paper. The foreign key "admin_id" is connected to the Users table's "id". This will let us know which admin user uploaded which paper. The foreign key "country_id" will link to the primary key "id" within the Countries table. This will help us create a heat map that shows where each paper is from. The heat map will be shaped like a globe and will be shaded based on where papers are located or not. "item_type" will be describing if the philosopher's text is from (webpage, book, or book's section). "language" will let us know which language the philosopher's text is in.

"author" is the philosopher who wrote the paper. "title" is the title of the book, article, or text. "editor" is who edited the paper for the philosopher. "publisher" is who published the paper. "publication_year" is what year the philosopher's paper was published. "num_pages" is how many pages the paper is. "url" will be the link to the philosopher's paper, or if the paper is from a book then will be NULL. "isbn" is the International Standard Book Number for each philosopher's paper. "abstract_note" will be any additional information that might want to be given, upon uploading the paper into the database. "created_at" is the exact time that the admin user uploaded the philosopher's paper into the database.

4.16.2 Papers Table Research Changes

Figure 43. Updated database table for storing papers

The Papers Table has had some changes after completing more research. We decided to merge the variables from the Countries Table and the Philosophers table into the Papers Table. We chose to do this because the Papers Table will be able to store the Countries and Philosophers tables variables and still be fluid without any misconception of coding practices with variable naming and how you would call something in a query as well as the information is still all coming down to being within the papers origin.

We also decided on using a Dublin Core metadata standard for Entangled Philosophies database, since it is the most simplest and widely used metadata standard used today. The variables added to the papers table that coincides with the Dublin Core metadata standard are Contributor (varchar), Relation (int), Subject (varchar), Format (varchar), Source (varchar), Rights (varchar) and Coverage (varchar).

4.16.2 Papers Table Final Changes

For our final design of the Papers table, we removed some of the metadata that we planned on using during our research. We chose to remove some of the metadata listed below because we wanted our papers metadata to mimic the Dublin Core metadata standard, to better standardize our papers data. Other than the items removed below, the Final Design and idea of the Papers stable stayed the same.

author_id -> removed from papers table
first_name -> removed from papers table
last_name -> removed from papers table
birth_date -> removed from papers table

admin_id -> removed from papers table
country -> location
country_born -> removed from papers table
continent -> removed from papers table
continent_born -> removed from papers table
item_type -> format
editor -> removed from papers table
publication_year -> removed from papers table
num_pages -> removed from papers table
abstract_note -> removed from papers table

4.17 Tags Table in Database Diagram

Figure 44: Database table for storing tags

4.17.1 Tags Table variable description

The Tags table will be populated by the researchers at the University of Hildesheim (Admin Users). Each paper will be associated with a set of tags. Each tag will have a unique primary key named “id”. We will use these primary keys to differentiate each tag from one another. The primary key “id” will be very useful in the database design since we have multiple languages within Entangled Philosophies. For example, the word “cat” in English is not the same spelling as “gato” in Spanish, although they do have the same meanings. Therefore, having a unique id for each word will eliminate the language barrier between tags.

The “language” field in the Tags table will describe which language the word/tag is in (English, Spanish, German, Chinese... etc.). The “tag_name” will be the name of the tag

and will be in whatever language it corresponds to. The “created_at” field will be a timestamp in the Tags table showing the exact time the tag was uploaded into the Entangled Philosophies database.

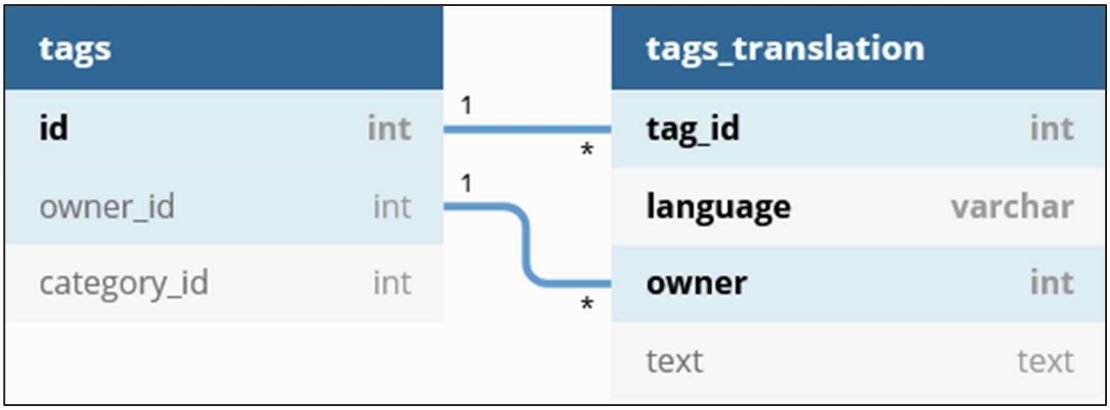
We are working on implementing an intermediate table or variable into our database that will flow between the Paper’s table and the Tag’s table. This intermediate table or variable will help us translate words that are queried by users into their primary key “id”. With this database design for the Tag’s table, users should be able to perform complex queries with AND, OR and NOT boolean logic statements to help with research within the philosophy domain.

4.17.2 Tags Table Research Changes

Figure 45. Updated database table for storing tags

The tags table has had some variables that we have added to it after doing more research. We have decided the id (int foreign key) will link to the papers table (id primary key) doing so let us return all of the tags within each paper. Next we chose to add tag_id (int) which will be a variable that describes a unique id for each specific tag. We have also added a category (varchar) which will be for describing which category the tag falls under. For example the tag “Egypt” will fall under the Countries category. Having categories within the tags table will also help us simplify the way that the results get returned to the users, based on their searches.

4.17.3 Tags Table Final Changes



Rather than just using a tags table, we decided to use a tags table that would work in conjunction with a tags_translation table. We chose to do this because of all the different translations of tags being in different languages, as well as users having the ability to make public and private tags, based on whether they were an admin, or a regular user. Some other changes were also made to our researched design for the tags table, which are listed below.

tag_id -> moved from tags table to tags_translation table
category -> category_id
tag_name -> renamed as "text" and was moved from tags table to trangs_translation table
language -> moved from tags table to tags_translation table
created_at -> removed from tags table

4.18 Users Table in Database Diagram

Figure 46. Database table for storing users

4.18.1 Users Table Variable Description

The Users table will contain all the users that register with Entangled Philosophies. The registration process will involve a user entering in their personal information, such as first name, last name, phone number, email, and password. Once the user clicks the “Register” button at the bottom of the registration form, they will have a code sent to their email to verify. After they click the verification link in their email, they will be able to log into Entangled Philosophies. The “id” is the primary key for the User table and will be used as a unique identifier for each user. This primary key will also help retrieve any data that is linked to that specific user. For example, maybe someone wants to retrieve all papers uploaded onto the Entangled Philosophies website by a certain user.

With this database design, that request would be possible, and all the papers uploaded by that user would be retrieved. The “first_name” variable is for the user's first name. The “last_name” variable is for the user’s last name. The “phone_number” variable is for the user’s phone number. The “email” field in the Users table is for the email address of the person registering an account. This email will be used to verify their account, to gain access to the system. The “email” is also used to log into their account.

The “password” variable will be for the user’s password and is one of the credentials to login to an account. The “is_admin” variable will be an integer value either 1 or 0. If “is_admin” is equal to 1, then the account has administrator rights and will have special privileges, such as uploading papers and tags onto Entangled Philosophies. Admin can ban users from logging into Entangled Philosophies, as well as create other admin users. If “is_admin” is equal to 0, then the user will be classified as a regular user and can only use the website with limited privileges, primarily for queries. “is_verified” will be for telling if the user verified their email code yet or not. The variable “created_at” is a timestamp for when the user or admin account is registered with Entangled Philosophies.

4.18.2 Users Table research changes

Figure 47. Updated database table for storing users

The users table has not undergone any research changes.

4.19 Philosophers Table in Database Diagram

Figure 48. Database table for storing philosophers

4.19.1 Philosophers Table Variable Description

The Philosophers table contains a primary key “id”. Which will be a unique identifier of each philosopher. “first_name” will be the first name of the philosopher’s name associated with a paper that gets uploaded onto Entangled Philosophies. “last_name” will be the last name of the philosopher’s paper. “birth_date” will be the day that the philosopher was born. “country_born” is the country that the philosophers were born in. “paper_quantity” will be the number of papers that a specific philosopher has on Entangled Philosophies.

4.19.2 Philosophers Table Research Changes

The Philosophers Table has been removed and the variables in the Philosophers table have been added into the Papers Table.

4.20 Countries Table in Database Diagram

Figure 49. Database table for storing countries

4.20.1 Countries Table Variable Description

The Countries table has a primary key “id” which will be linked to which country the paper is from. This primary key will help with a heat map that will be implemented into Entangled Philosophies. The “name” variable will tie to the name of the country that the paper is from. The “continent_name” field will be which continent that the paper is from.

4.20.2 Countries Table Research Changes

The Countries Table has been removed and the variables in the Countries table has been added into the Papers Table.

4.21 Metadata Standard

4.21.1 What is metadata?

Metadata described in simple terms is basically “data about data.” It’s tracking pieces of information about cataloged resources and using it to analyze trends in data and is helpful when sharing data with other parties. There are 3 overarching categories when it comes to metadata:

- Descriptive metadata: this type of information is geared towards identifying resources and will include data such as the title, author, targeted keywords, etc.
- Structural metadata: this type of metadata is helpful for describing how objects are organized and structured.
- Administrative metadata: this is generally helpful for managing resources by tracking information such as its file type, when/who created it, etc.

Regarding Entangled Philosophies, we're mainly interested in tracking descriptive and administrative metadata. It's worth noting that in most resources that we track, we will probably be maintaining the table of contents for it and while that would fall under the category of structural metadata, that information won't actually be searchable and will instead just be an add-on file that administrators can upload to a paper to make it easier for their research.

We've broken up metadata into 3 general categories but in reality, there are dozens of different categories that we can define a metadata standard for and hundreds of different types of standards overall [27].

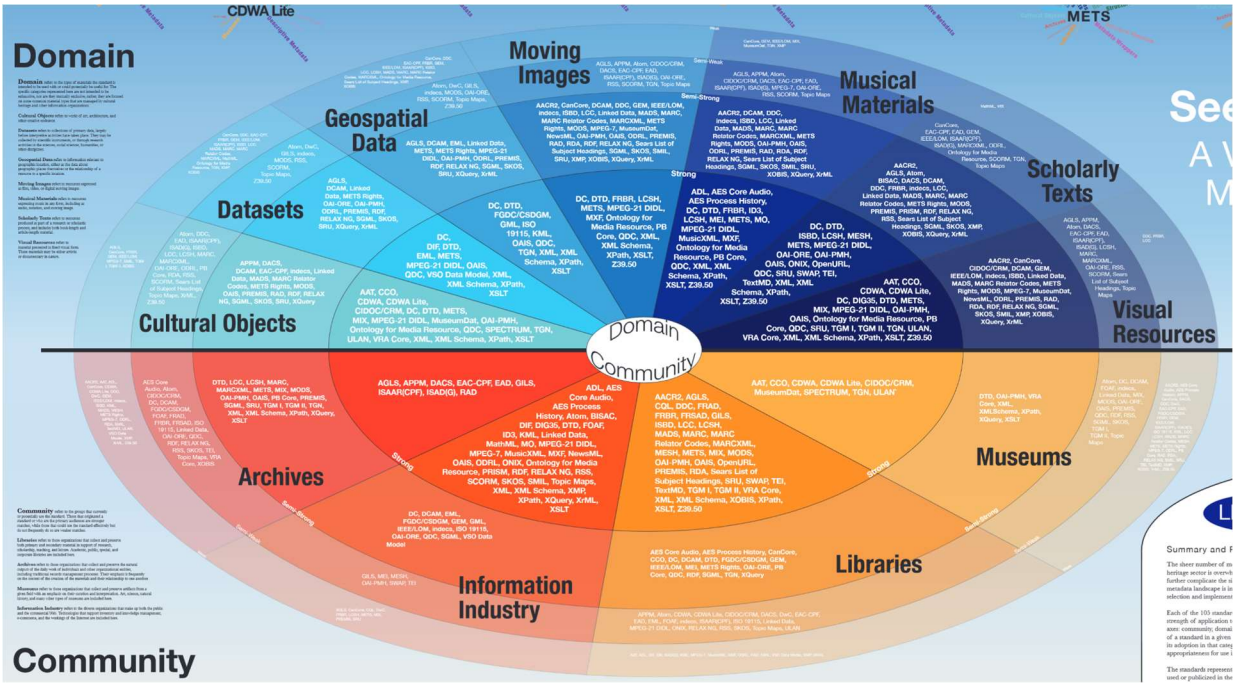


Figure 50. The range of what metadata standards there are for different mediums

4.21.2 Importance of metadata

In general, metadata is extremely important to track whenever you're interested in querying or viewing any sort of data. Tracking the right type of information (using some metadata standard that is designed to fit your data) makes it accessible to your users, makes it easy to integrate information about physical and electronic resources (which is becoming increasingly important as our society technologically progresses), and by

following a known standard (or some derivation of it) makes it easy to share information with other parties.

While it's clear that organizing your data according to some metadata standard is extremely important, picking the right standard to follow is critical for the overall functionality of the service and for its future (in case our sponsor decides to merge their data with another party).When deciding on a standard, we'll be looking for one that is well tailored to the product we're building and also fits the data that our sponsors are currently tracking (which was information already given to us). When picking a metadata standard, it's important to compare and contrast several standards in order to properly determine which will be best for our product.

4.21.3 Dublin Core

The Dublin Core standard is a metadata standard that is focused around tracking 15 pieces of optional data relating to a resource and is capable of tracking digital and physical resources. Initially the standard consisted of 2 levels of complexity: Simple and Qualified. The simple level had the established 15 categories of data to track, but the Qualified level had introduced 3 additional categories (Audience, Provenance, and Rights Holders).

However, in 2012, the standard has merged these 2 levels into a standardized 15 [28]:

1. Title: a given name that succinctly describes the resource being cataloged.
2. Identifier: a unique id value given to this resource so that it may be identified in the space it's being stored.
3. Creator: the name of the individual responsible for the creation of the resource (the author in our case).
4. Contributor: in case there was more than one individual/group that helped create the resource, they would be recognized here.
5. Relation: the unique identifier of any related resource that may be relevant to the current resource.
6. Subject: a few words describing what type of content is talked about in this resource.
7. Date: a date (or potentially a range of dates) where the resource was originally created/discovered.
8. Description: a more formal and long description of the resource (an abstract, book summary, etc.).
9. Type: a genre associated with the resource (philosophy, politics, etc.).

10. Format: the physical description of the resource (file type and size if it's digital, size of the resource if it's physical).
11. Language: the language corresponding to the version of the resource we're cataloging.
12. Source: an identifier for a potential resource that inspired the current one.
13. Publisher: the entity that is responsible for making the resource publicly available.
14. Rights: Information regarding the legal rights of the resource (copyright claims if it's physical for example).
15. Coverage: a topic describing under which circumstances this resource could be relevant.

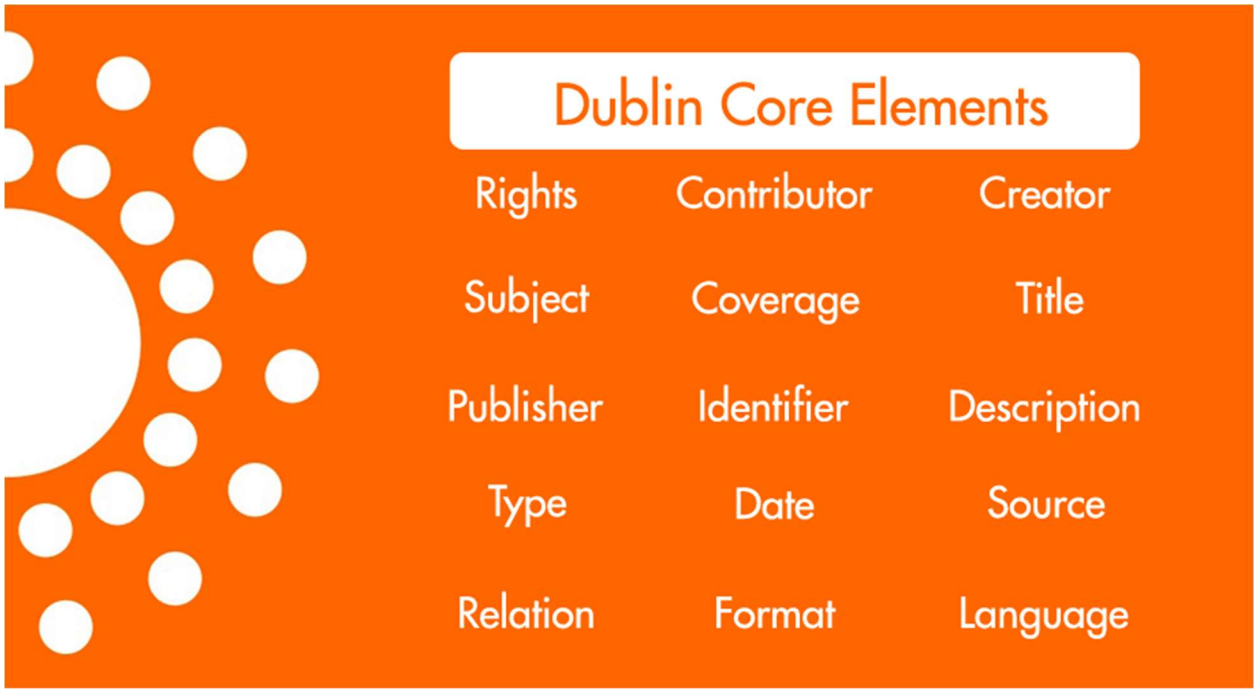


Figure 51. Dublin Core Element Summary

It's worth noting that each of these pieces of information are considered optional but in the context of Entangled Philosophies, the information we'll require will be the Title and Identifier elements (the user provides a title and we'll supply the identifier). While the elements described here are informative, there are still missing elements that our sponsors track and that Dublin Core does not. Some of those elements include ISBN numbers, URLs, and more descriptive elements for contributors such as translators, short titles, and editors.

To expand on why the contributor element is incomplete, for most of the resources tracked by our client, there are editors, translators, and co-authors which are all elements that they track. To remedy this, if we end up using Dublin Core, we'll most likely need to expand the Contributor element into the stated categories and in addition, add the elements that correspond to ISBN and URL. It's important to emphasize that while we do want to choose a standard to follow in order to make the data being tracked more structured and easily shareable, it's unlikely that we'll find a standard that is exactly what we're looking for. In response, it'll be fine to add our own expansions to a given standard as long as we follow the core elements of our standard.

4.21.4 MARC XML

The MARC (machine-readable cataloging) standards have been around since the 1960s and have mainly existed to describe physical books. Even though the mediums we'll end up tracking in Entangled Philosophies are not strictly physical, it's very important to understand what exactly makes the MARC standards useful and why it's still used regularly today.

Very important to Entangled Philosophies, the MARC standards are focused on tracking the bibliographic information for books. Some of the fields described are: title, author, publisher, language, date, and media type. At the time of creation, computer space was a more limited resource and so the MARC standards introduced a 3 digit identifier for each field (the author is denoted by the identifier 100). This simple and defined format makes it very easy to store and communicate metadata.

In the early 2000s, the Library of Congress moved towards an XML description of the MARC standard in order to make it easier for evolving computer systems to parse this bibliographic information.

```
<?xml version="1.0" encoding="UTF-8"?>
- <marc:collection xsi:schemaLocation="htt p://www.loc.gov/MARC21/slim
http://www.loc.gov/standards/marcxml/schema/MARC21slim.x sd"
xmlns:xsi="http://www.w3.org/2001/XMLSchemaOnstance"
xmlns:marc="http://www.loc.gov/MARC21/slim">
  - <marc:record>
      <marc:leader>00957 a2200145 4500</marc:leader>
    - <marc:datafield ind2=" " ind1="7" tag="024">
        <marc:subfield code="a">10.1515/jmc.2007.011</marc:subfield>
        <marc:subfield code="2">doi</marc:subfield>
      </marc:datafield>
    - <marc:datafield ind2=" " ind1=" " tag="041">
        <marc:subfield code="a">eng</marc:subfield>
      </marc:datafield>
    - <marc:datafield ind2=" " ind1="1" tag="100">
        <marc:subfield code="a">Joan Daemen</marc:subfield>
      </marc:datafield>
    - <marc:datafield ind2=" " ind1="1" tag="700">
        <marc:subfield code="a">Vincent Rijmen</marc:subfield>
      </marc:datafield>
    - <marc:datafield ind2="0" ind1="0" tag="245">
        <marc:subfield code="a">Probability Distributions of Correlation
          and Differentials in Block Ciphers.</marc:subfield>
```

Figure 52. MARC XML example

Using MARC XML is definitely not an option for Entangled Philosophies (since our metadata will be stored inside of database entries and not XML files) but because the MARC standards are geared towards bibliographic information, researching its progress over time can give key insight for how we should be designing/modifying the metadata standard we'll end up using.

In fact, a key advantage of MARX XML is that there is a defined standard for how exported/imported information should look. Because XML is such a well known format, any bibliographic data stored using MARC XML becomes very easy to share with other worldwide organizations that may be interested in working with Entangled Philosophies. While we can't necessarily use this standard for our own database, we can clearly see the value and importance of having our data be easily exportable to some international standard for easy collaboration. A requested feature of Entangled Philosophies is that administrators will be able to upload spreadsheets of bibliographic information for efficient batch uploads of data they already have. As an extension to that feature, it would be convenient if we offered the reverse: export a certain subset of the database entries into the same spreadsheet format that we allow for imports. As a result, while we may not be explicitly using MARC XML as our metadata standard, we may definitely take inspiration from some of its field categories/data representation and work that into Entangled Philosophies.

4.21.5 MODS

MODS (Metadata Object Description Schema) is essentially a hybrid of the Dublin Core and MARC standards. It combines the flexibility of formatting data as XML and the simplicity of Dublin Core's language-based tags in order to efficiently describe bibliographic information for both physical and digital mediums.

A key benefit of using MODS is that its structure is multi-level and does not have a flat representation. What that means is that there can be multiple layers of sub-fields for each defined field unlike MARC XML which only allowed one layer of sub-fields. By having this tree-like structure, MODS becomes significantly more flexible in the level of detail that it can represent.

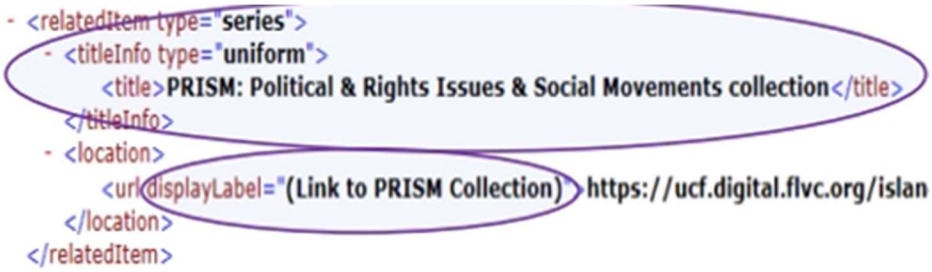


Figure 53. MODS example

As we can see in the picture above, we have the ability to add multiple layers to the field *relatedItem* which are *titleInfo* and *location* which both have their own subfields as well. By having this sort of structure and being able to add local fields that aren't strictly defined by the standard (unlike MARC) we can incorporate Dublin Core's key categories into an XML format and then build off of it arbitrarily.

It's evident that the amount of freedom and flexibility offered by MODS seems to be superior to Dublin Core and MARC but MODS may still not be the best metadata standard that fits Entangled Philosophies. While MODS does provide us the tools to expand the way we represent our data, not only is it still in XML (which is something we would need to convert to standard key:value pairs for the database entries) but the bibliographic information that we'll be tracking doesn't benefit from this extra freedom. The data we'll be storing for each entry is essentially flat (there may be a few categories that are related to each other) and so we don't necessarily need the multi-level structure and while being able to freely define local fields is beneficial, the scope of Entangled Philosophies is more restrictive and all tracked metadata needs to be defined beforehand. So, while MODS

seems to be a very powerful metadata standard, a lot of its offered features aren't useful to Entangled Philosophies and it would be better to pick a simpler standard.

4.21.6 TEI

TEI (Text Encoding Initiative) is a metadata standard designed for cataloging text from domains such as the humanities, social sciences, linguistics, etc. As per usual with metadata standards, the main format is in XML and this standard is useful for encoding bibliographic information about texts with a highly adaptive schema. By researching a standard like TEI, which one could say is a standard targeted towards the same kind of content we'll be storing in Entangled Philosophies, we can get a better idea of what type of fields might be relevant for our sponsor to consider including in the future.
For example, some of the fields that TEI is built to track include edition statements and publication statements which are often included in books about philosophy. While this information isn't something that our sponsors have communicated they would want stored in the database, by seeing what other standards may be used, we can design/modify our metadata standard in such a way that it doesn't rule out the possibility of eventually expanding to include these fields.

```
▼<sourceDesc>
  ▼<biblStruct>
    ▼<analytic>
       <author>Brown, Charles Brockden</author>
       <title level="a">Richard the Third and Perkin Warbeck</title>
     </analytic>
    ▼<monogr>
       <title level="j">The Literary Magazine and American Register</title>
      ▼<imprint>
         <pubPlace>Philadelphia</pubPlace>
         <publisher>John Conrad & Co.</publisher>
         <date when="1805-02">February 1805</date>
         <biblScope type="vol">III</biblScope>
         <biblScope type="issue">17</biblScope>
         <biblScope type="pp" from="108" to="110">108-110</biblScope>
       </imprint>
     </monogr>
   </biblStruct>
 </sourceDesc>
```

Figure 54. TEI example

As we can see by this snippet of XML, the level of detail that can be applied to the bibliographic information of a work isn't something we plan on handling in the scope of Entangled Philosophies, but if we can design our metadata standard in a way that's easy

for future users to modify, it can be expanded to include information about the imprint, bibliographic scope, etc.

4.21.7 ONIX

ONIX (Online Information Exchange) is a metadata standard that was initially designed to help communicate bibliographic information for physical books. Eventually, the standard expanded to allow support for eBooks making it a good standard for tracking publications of books in all forms. It's mainly employed by the publishing industry and has a high level of description about a book's information (contributor roles, illustrators, shipping information, etc.).

While ONIX may be a little too detailed for the uses of Entangled Philosophies, it's worth noting its flexibility and adaptability help it fit a wide variety of collections and be crosswalked to other popular standards (such as MARC). By familiarizing ourselves with the type of information that is tracked by the ONIX standard, we can better prepare our own metadata standard to at least have some common fields as well in order to help facilitate potential data merges with other databases in the future.

4.21.8 Metadata standard for Entangled Philosophies

After an extensive look at different metadata standards geared towards maintaining bibliographic information, there are a few key takeaway points that are worth taking into account when choosing a standard:

- Compatibility with information provided by the University of Hildesheim (our standard should be able to support fields that they're currently tracking)
- Cross compatibility with other metadata standards. If our sponsors ever decide to merge databases with another database stored using a different standard we should pick a standard that makes that transition easy.
- A single level description depth is enough. Given that this data will be stored as key:value pairs in database entries, we don't need to employ a multi-level metadata standard.

Given all these requirements, we'll be going with an expanded version of Dublin Core. Dublin Core's initial 15 elements are in line with the information we've received from the University of Hildesheim. We won't be removing any of the default fields but will instead be adding some additional ones such as ISBN numbers and URLs. And because the standard we're using will be simple and single level, exporting the information in the

database will end up being an easy task. Overall, learning about different metadata standards gives us a good idea of what to expect in order to fully accommodate our sponsors and any future endeavours they may have.

5 Overall System Design

5.1 Web App

5.1.1 UI and Design

5.1.1.1 Account Registration and Login

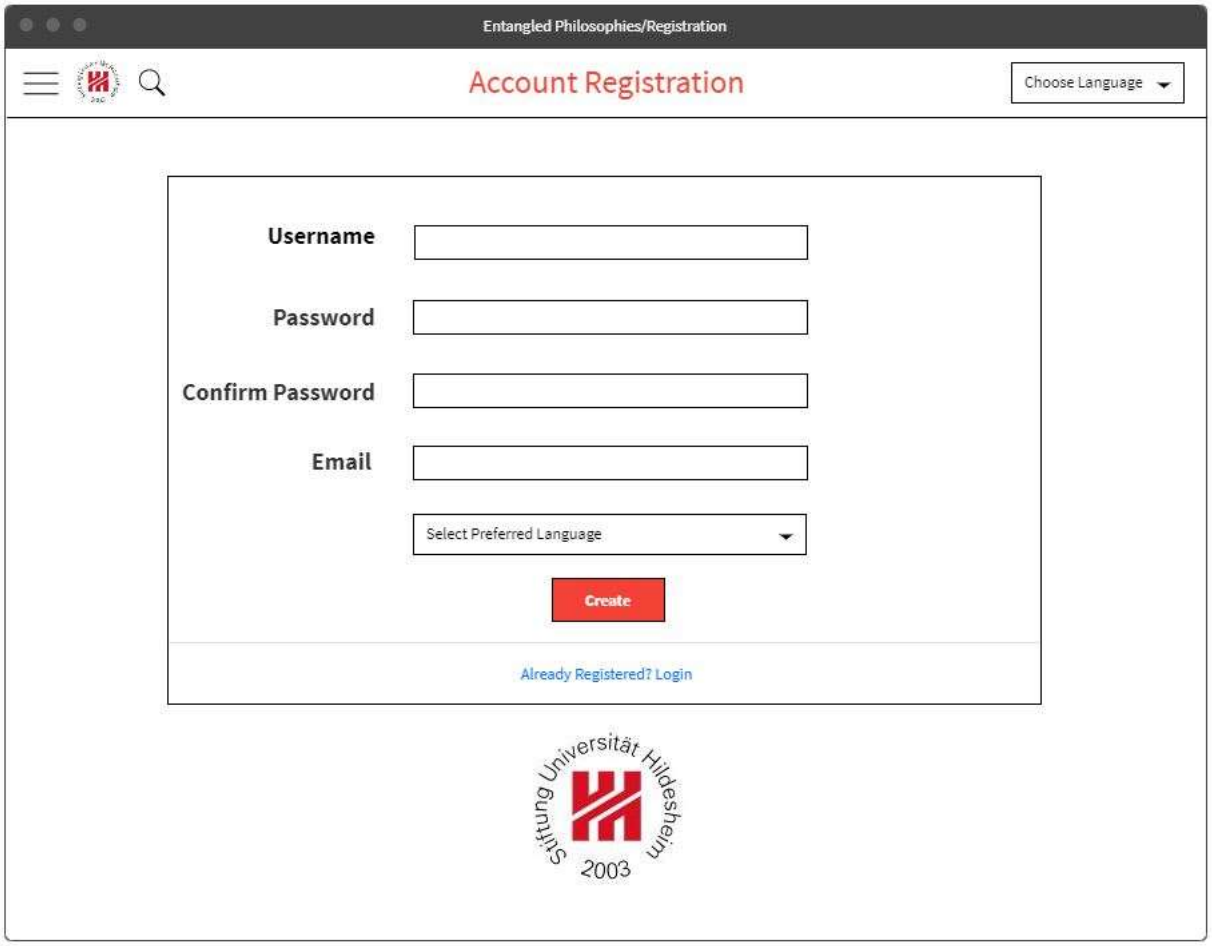


Figure 55: Registration page for registering accounts

The Account Registration Page has 5 fields to enter before you can create an account. The Username will have to be unique when chosen and must not be the same as any other Usernames in the Entangled Philosophies database. The Password will have to be more then 6 character's long and will have to have 1 uppercase and 1 special character. The password will also have to be confirmed when creating an account, so that you do

not accidently enter in two different passwords. The Email you enter will also have to be unique and will have to be an email that you can log into the verify your account, after you click the “Create” button at the bottom.

Depending on which language you prefer you can click the “Select preferred Language” drop down, to have a default language of your choice for when you log into Entangled Philosophies. If you were brought to the Account Registration screen and already have an account, you can click on the blue “Already Registered? Login”, which will take you to the Account Login Page, so that you can log into your Entangled Philosophies account. The items at the top of the Account Registration page are static and will be the same for pretty much every page on Entangled Philosophies. They will be discussed after the Account Login Page Photo below.

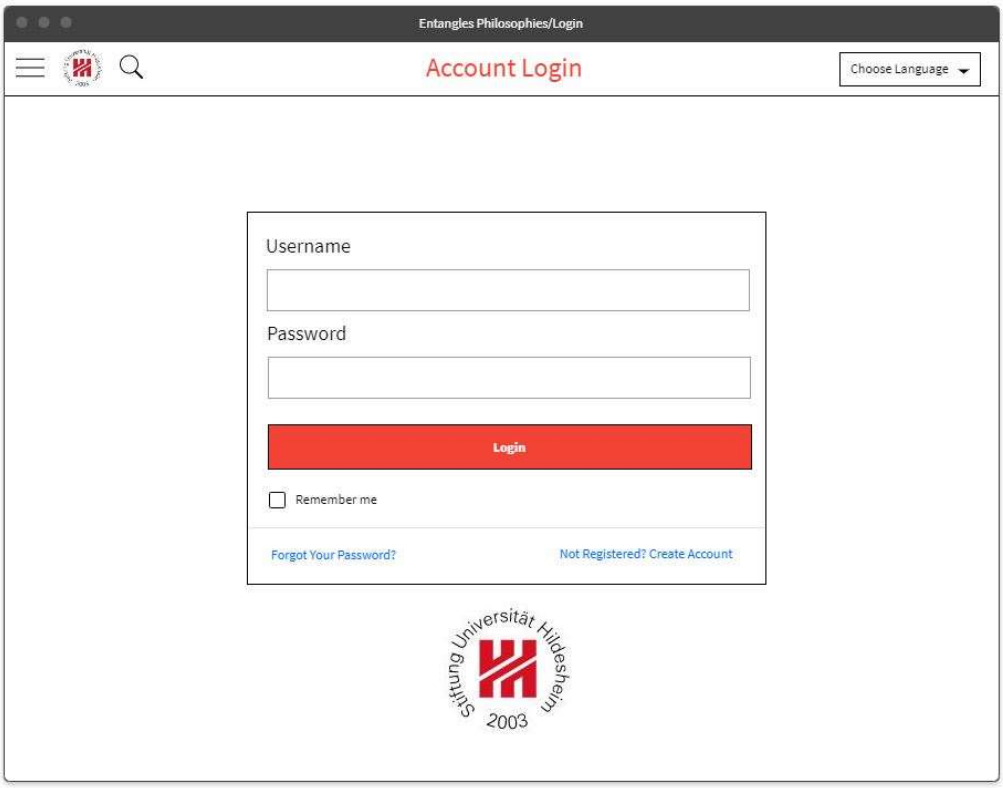


Figure 56. Account Login page for logging in

The Entangled Philosophies Account Login Page has fields to enter the user’s login credentials like Username and Password to get into their account. It Also has a “Remember me” checkbox that you can select, so that the next time that you visit the Account Login Page, that your login credentials are already saved. If you forgot your password to login, you can click the blue “Forgot Your Password?” text to reset your

password. To reset your password, you will have to type in your account's email address and then you will be emailed a link, which will allow you to create a new password for your account. If you are accidently on the Account Login Page and do not have an account yet, then you can click the "Not Registered? Create Account" text to go to the Account Creation Page. At the top of the Account Login Page there is a hamburger menu that will be at the top of every page of entangled philosophies, which will have various menu items, such as settings, about us, query screen (home), etc.

When clicking onto the hamburger menu, a rectangular side bar will come out and display the options to choose. The magnify glass at the top of the Account Creation Page will take you to the query page and will let you perform searches on Entangled Philosophies. If you do want to save queries, then you will have to have an account registered with Entangled Philosophies. The final item for the Entangled Philosophies Account Creation Page is the Choose Language drop down, which will allow you to choose from various languages like English, German, Spanish, etc. After the language is selected, the pages shown for the user will be in the language that was chosen.

5.1.1.2 Paper Upload

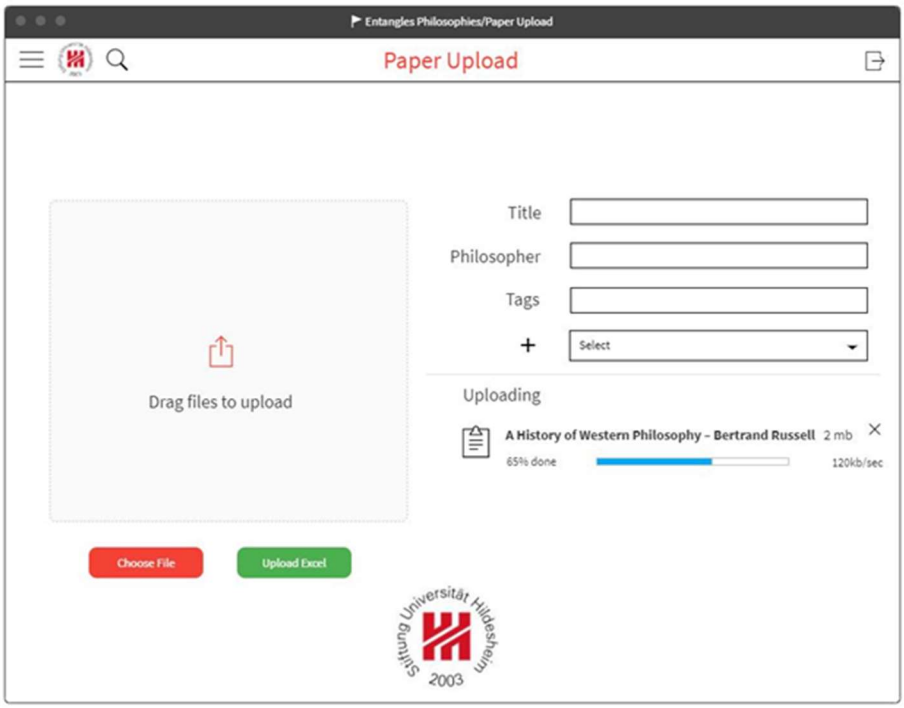


Figure 57. Paper Upload page for uploading papers

The Paper Upload Page is an important page for Entangled Philosophies. On the right side of the Paper Upload Page are boxes to enter the Paper's Title, Philosopher's name

as well as additional tag's that will be tied to each paper. There is a "+" symbol with a drop down to the right of it, which will allow you to continually add more details for each paper upload. The "+" symbol was integrated to give the admins control of which information needs to be added or not to each paper.

After you have filled in the right side with your needed details, and you are ready to upload a paper onto the website with your admin account logged in. You will select the "Choose File" button and select the paper you wish to upload. You can also upload an excel file for papers if needed. Once that is done you will see the paper of you selected being uploaded under the word "Uploading" and given an estimated time of completion. If you wish to cancel the paper upload, then you can select the "x" above the paper being upload and the paper upload will be canceled.

5.1.1.3 Edit Paper

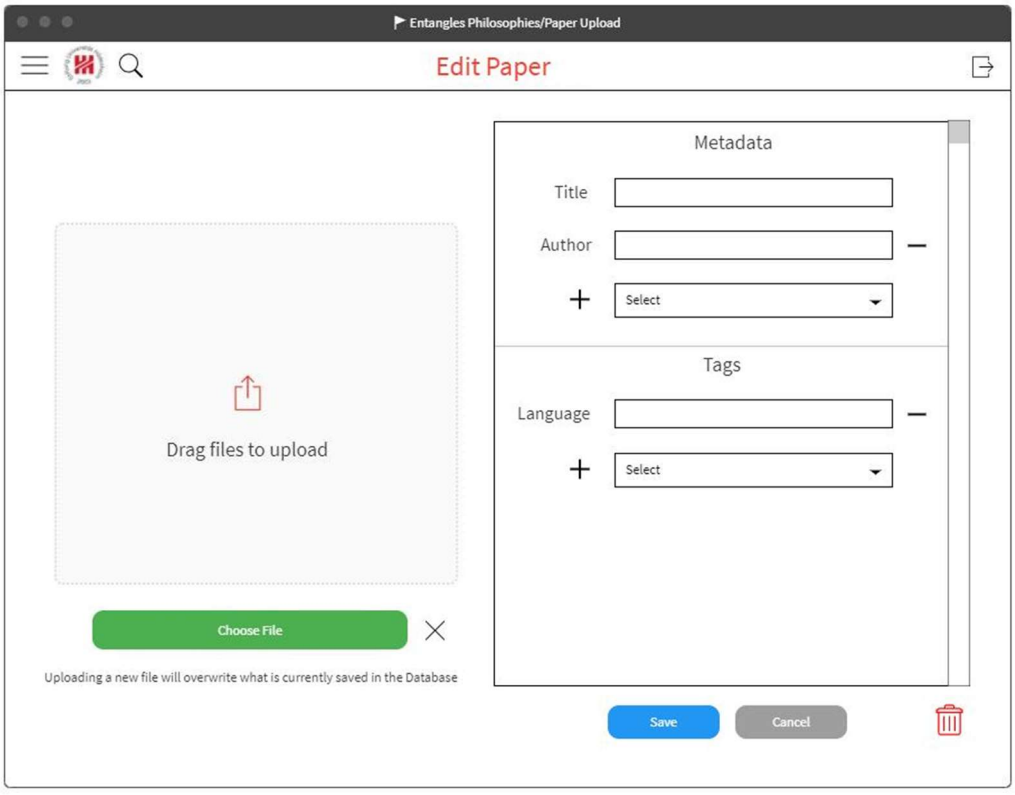


Figure 58. Front end design for editing an uploaded paper

When it came to designing the Edit Paper page for our Web App, we knew that it would essentially have to look like the initial Upload Paper page but with a lot of the information about the paper prefilled so that a user could edit it. This led to us doing a complete rework

of our initial Upload Paper page so that users would be able to add/edit the information in a clear and convenient way.

We started off by segmenting the Metadata and Tags section into 2 different areas of a scroll box. A key component of Entangled Philosophies is that while we can also search by the tags we've added to a paper, we should also be able to search by the metadata related to a paper as well. So, knowing that we're using Dublin Core for our metadata standard, users would also be able to search by the format of a paper, its language, etc. In the database, there's almost no reason to make a distinction between metadata and tags since both of these pieces of information are searchable and are related to a paper in the same way. But on the front end, it's very important to make a distinction between the two since regular users are not allowed to add their own custom information into the metadata of a paper. Maintaining this distinction in the database is as simple as just adding an extra field to columns indicating whether or not it's a metadata tag. This will allow us to easily make distinctions between the 2 when presenting the information about a paper to the user.

In each of the sections for adding metadata or new tags, we can see a simple drop-down box labelled "Select." When clicking this drop-down box for metadata, the administrator will be presented with all the metadata fields that have not been currently selected for the paper. Upon choosing one, a new row will be added to the metadata section with a label on the left hand side indicating what type of metadata this box represents. We can also notice a black minus sign on the right hand side which just allows the administrator to remove the selected field.

When it comes to editing tags, the presentation is slightly different. In our current designs, each tag belongs to some category (like "General" or "Philosophical idea") so that there is a clear subdivide between the types of tags to the user. So, when clicking on the "Select" drop-down box, the user will be presented with the entire list of possible categories they could add. Once selected, the user can input the tag they wish to add to the paper. The method of adding a tag has been a topic of discussion among us where we've been debating whether or not to add an auto-complete feature to this section to help facilitate adding tags. Currently, we're opting to not have an auto-complete feature and instead have the user find the tag they wish to add in the "Tags" page and have them copy and paste the text to avoid user errors. As a stretch goal, we would definitely consider adding auto-complete features to help with this tedious aspect.
At the bottom of the right side we can see some familiar options: a save and cancel button as well as a red trash can. The save and cancel buttons are there to save changes we've made to the paper (i.e. new/removed/edited tags and metadata) and to ignore any of

these changes we've made and not commit them to the database. The red trash can exists for when an administrator wishes to delete a paper from the database. Currently, when an administrator deletes a paper, it's gone for good with no chance to roll back those changes. Since deleting a paper is a big decision (it affects saved collections for users as well), they will be presented with a pop-up box where they need to confirm the deletion of the paper. As a stretch goal, we're considering having the ability to roll back these more permanent changes such as deleting tags/papers from the database. Because these decisions alter many pieces in the database (such as saved queries and collections of papers), we do not have plans to implement this roll back feature unless there is ample extra time.

And the last section of the "Edit Paper" page is related to the uploaded file associated with a paper. This optional file upload is generally used to describe any extra information about a paper that can't be described by our metadata standards (such as the Table of Contents) that a user may be interested in when viewing the paper. The administrator has the opportunity here to upload a new file associated with that paper (for security purposes, it'll most likely be limited to some text format document such as pdf or rtf) but should note that there is only allowed to be one file upload per paper in the database and so will replace any currently uploaded file. Similar to the other information, we will display what file is currently associated with that paper and allow the administrator to download and view it before uploading a change.

5.1.1.4 Search

The Search page is going to be the landing page after you login to the Entangled Philosophies website. Here you will be able to click the "Filter" button to do advanced searches, or type in things like "Germany" into the search bar to return all papers from Germany. Depending on how many results are returned the numbers will be displayed within the search bar. There is a next/previous text at the bottom of the results of your search that will allow you to go to the next or previous page.

The visualization button will bring up a few options for different visualization layouts that will show you your data in word clouds and a hierarchy view. Visualizing data like this can help people make crucial decisions about the information that they are looking for and give them another perspective. The Options button will allow you to select or deselect columns from the search results. After selecting the columns that you want to display you will only see those columns. The right side of the page will display which search result you click onto. Here will be information displayed about the paper selected.

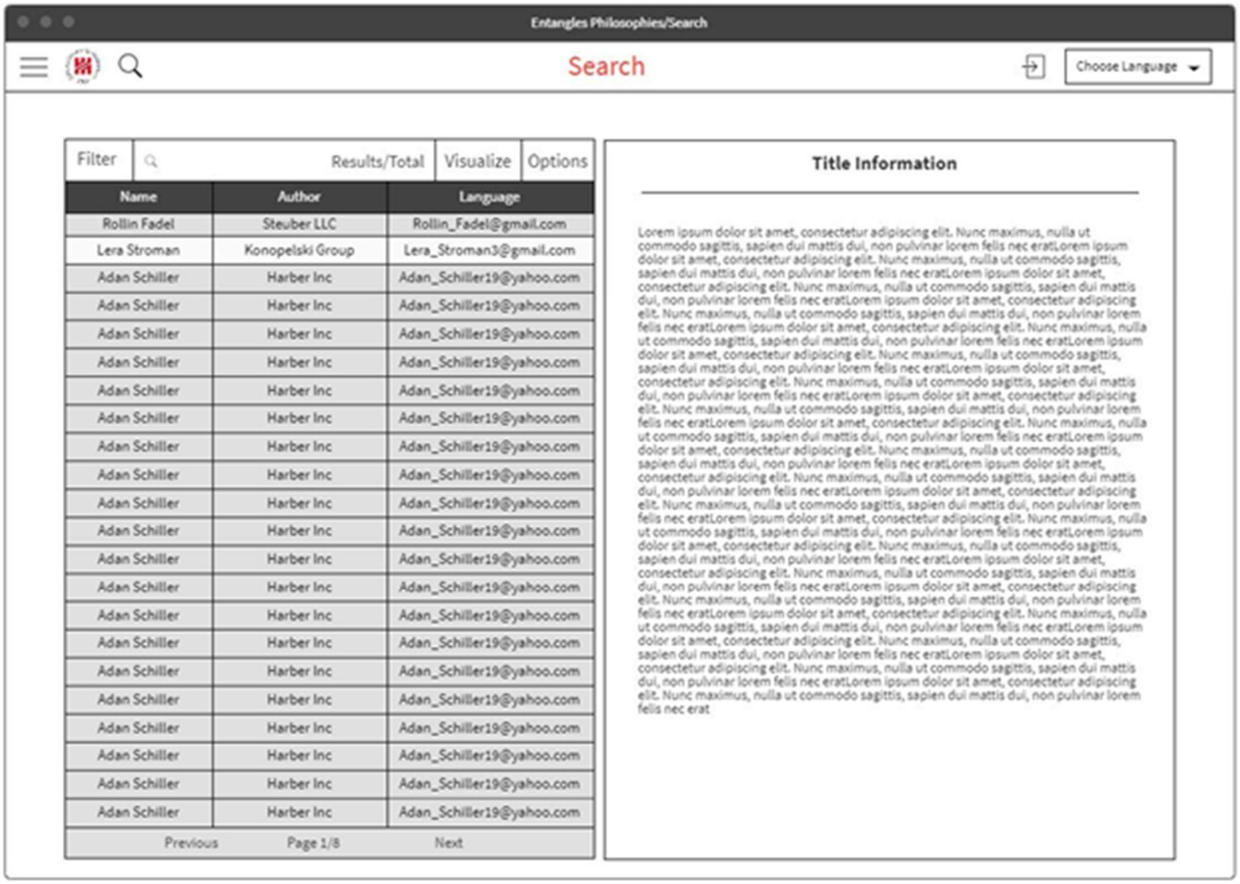


Figure 59. Search page

5.1.1.5 Advanced Search

One of the key features of Entangled Philosophies is our ability to make complex boolean logic search queries on papers we've tagged so far. While our extremely basic search page simply displayed results and allowed users to search by title, this Advanced Search page allows the user to specify complex AND/OR relationships between the tags they're interested in.

The layout of this advanced page follows a general style: display a tag category (Religion, Philosophical idea, etc.) and then the most relevant tags for that category (either based on most recently used or most used in general). Because there may eventually end up being hundreds of tags for a category, we'll pick some constant number to display initially for each category and give the user an option to expand a category to view all tags.

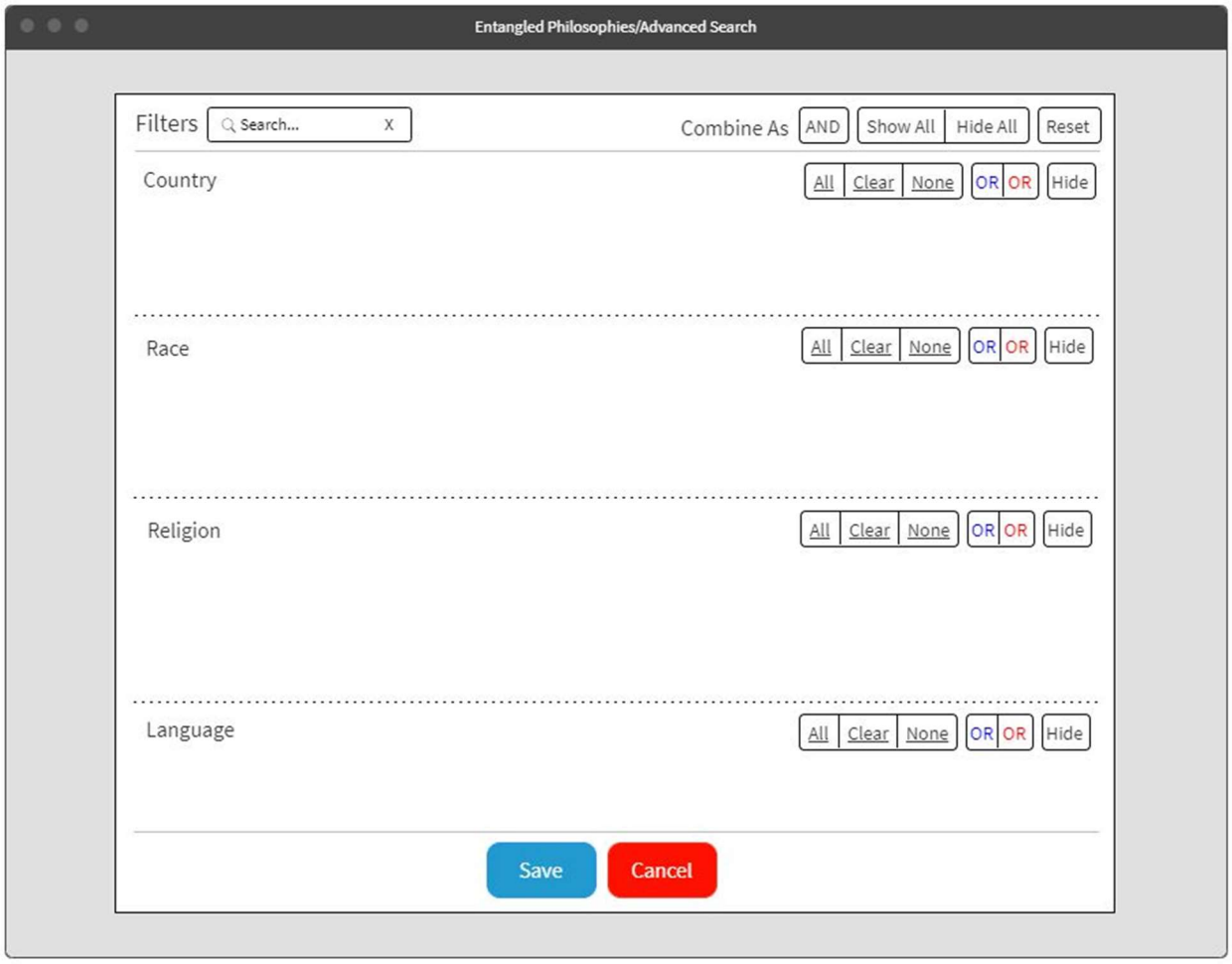


Figure 60. Advanced Search page for complex queries

Each individual tag can have 3 states when it comes to a search: include, exclude, and undefined. Clicking on a tag entry will make it cycle between these 3 states (the order being undefined, include, and then exclude). The “include” state is denoted by the color blue, the “exclude” state is denoted by red, and an undefined state will be a light gray. “Include” and “exclude” are simply whether we want papers with or without this tag but the undefined state for a tag will mean that we don’t need to specify a restriction for that tag (it can appear or not appear in a paper).

What you’ll notice when viewing each category of tags, is that there are 6 buttons on the upper-right corner that help the user further define what their query is looking for. The initial 3 buttons are reasonably simple: “All” which will set every tag in the section to be included (i.e. blue), “None” which will set every tag to be excluded, and “Clear” which will set all tags to be in their undefined state. The “Hide” button will simply compress the

category into a hidden row which will be useful if the user needs to scroll through many categories to find one they're interested in filtering.

The last 2 buttons are not entirely clear upon the first glance and so we'll most likely provide some type of description for it when a user hovers their mouse over one of them. These last 2 buttons describe the relationship between the included tags and the excluded tags. When a series of tags are marked as "include", there is still some ambiguity in what that actually means. Should we only return papers that have all "include" tags selected or should we return papers that have at least one of the "include" tags selected? These 2 buttons describe that kind of specification between tags that have the same state: should they all appear or should at least one of them appear? It's the same for all tags marked as "exclude". Either we return papers that don't have any of those tags or papers that at least have excluded one of them.

The last area to describe about the Advanced Search page would be the top bar that has a few extra options for users to consider. The first is a standard search bar which can help users find specific tags that they're looking for. The "Show All" and "Hide All" buttons in the upper right are convenient ways to minimize/maximize every category on the advanced search page (similar to the "Hide/Show" option on each individual category). The "Reset" button exists to set every tag on the page to an undefined state. And our "Combine As" option has a similar function to the previously described buttons but instead denotes the AND/OR relationship between categories of tags instead of the individual tags themselves. Combining the categories as AND implies that we only return papers that match all criteria defined for all categories and similarly, OR implies that we will return papers that match at least one of the criteria defined. And the last 2 buttons at the bottom of the page exist so that a user can apply the changes to their criteria that they made, or cancel it and leave their results page unchanged.

5.1.1.6 Tags

The Tags page gives both general users and administrators the option to add/edit/remove their tags. When an administrator makes edits to a tag, those are considered public edits and all users will be affected by those changes. General users also have the opportunity to make their own custom tags that are only viewable by them.

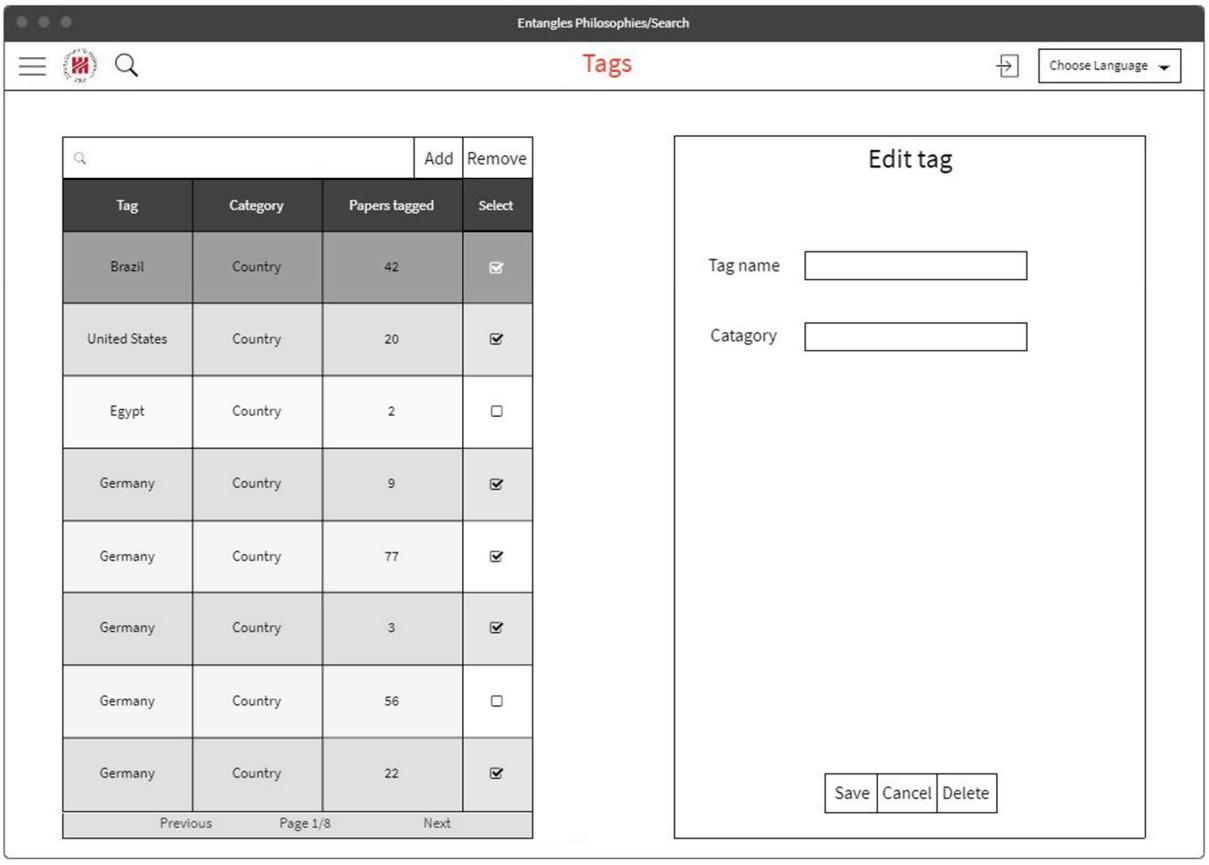


Figure 61. Tags page

The layout of the left side of the page is a simple table displaying all the current tags. Of course, since there may be hundreds of tags, these results are segmented by page and in the final design, we'll most likely display around 20 entries per page and give the users the option to cycle through these pages. At the top of the table, we have a search bar that allows the users to make a quick search for tags whose names match their input. It's currently undecided if this search bar will update on each change made to the search criteria or if the user will have some sort of search button that commits their search and loads matching results. Regardless, the user will quickly be able to find a specific tag if necessary. The 2 options on the right side of the search bar are also reasonably simple: the "Add" button will open up an input box on the right side where the user will add the necessary information about a tag, and the "Remove" button will delete all tags that are selected on the rightmost column of the table. Since deleting tags is a decision that can't be undone, users/administrators will be presented with some sort of warning dialog box that confirms their actions to them.

The result table itself has a lot of customizable options for the user to interact with. The 4 columns displayed are the tag name, its category, the number of papers it's tagged in, and whether or not it's being selected (for removal). The tag name that is displayed to the user is dependent on whether that tag was user-created or a public tag. If the tag is public, then that implies that there exists translations for that tag in all supported languages for the website. As a result, the translation that is presented to the user will be the translation that corresponds to their preferred language setting. If, however, the tag is a user-created tag, what will instead be displayed is the plaintext that they initially inputted when creating the tag.The category field will simply display the proper translation of the tag's category. The "Papers Tagged" field simply represents the amount of papers that a certain tag is involved with. Currently, our design does not intend to track that information manually in the database and will instead display it by simply making a query to the database table that handles tag-paper relationships to get the proper count.

A feature not displayed in the wire frame for this page is that each column is actually sortable in ascending or descending order (except the "Selected" column). When sorting columns that are displayed through plaintext, we'll simply sort them in lexicographical order and when sorting the "Papers Tagged" column, we'll sort them by ascending or descending numerically.

The final piece of the Tags page to analyze is the actual addition/edit of a tag. There are 2 different displays that a user will see depending on if they're an administrator or if they are a general user. What is displayed above is the view that a general user will have. When a general user is adding their own tag, the only information they need to fill out is the name of the tag and the category it belongs to (which can auto fill to "General" if they don't have a specification). Then all they need to do is simply add the tag by clicking the corresponding button at the bottom.

When administrators are adding tags, the view will change by a significant amount. In addition to specifying the category of the tag, administrators will have to provide all translations of a tag before adding it. In order to say that Entangled Philosophy fully supports a language, all tags and website information needs to be displayed in said language. That being said, it may be that at the time of adding a tag, not all translations are readily available to an administrator. In order to allow them to stall adding all translations and still get the tag into the database, they are allowed to simply fill in the unknown languages with a different language's translation. In general, we won't allow tag translations to have duplicate names (even user-created tags) but by allowing this we are introducing equivalent tag translations into the database. While this may seem troublesome on the back end, this will be the only instance of duplicate text in the

translations table and it won't actually break any of the endpoints that involve searching for tags. The endpoints that exist to search for proper tag translations will always take in the plaintext version of the translation as well as a language identifier for that tag meaning that there will still always be one result in the SQL query if a tag exists.

5.1.1.7 Saved Queries

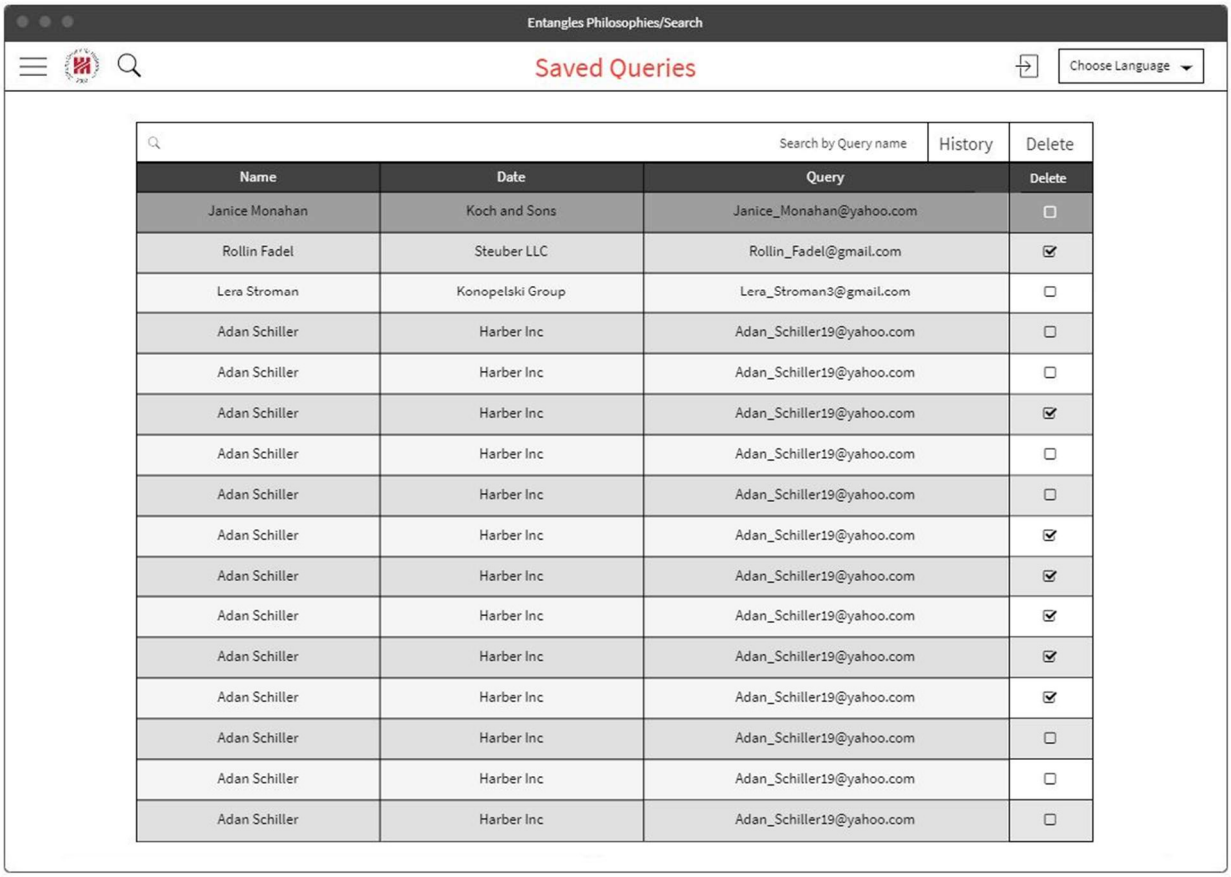


Figure 62: Saved Queries layout

One of the most useful features for a user in Entangled Philosophies is the ability to save their queries for quick re-edits and to bring up any updated results they may be interested in. Users can view their saved queries in this table shown above that has 3 main fields of information: the query's name, the date and time it was created, and a string of text representing their boolean expression. Similar to before, each of these columns are sortable by ascending and descending order. There also exists a delete checkbox column on the right hand side that users can use to speed up their deletion process.

The top of the layout provides a search bar, a delete button, and a toggle button to show their search history. The search bar is useful when a user knows which query they're interested in viewing again and so they can directly input a part of its name to get any matching results. The delete button is simply used for deleting all checked queries for quick deletion.

The history toggle button is there to allow a user to view a small snippet of their most recent search queries. In the database, a saved query and query history are actually stored in the same way. Since each saved query will have an identification number (which will serve as its primary key), a piece of search history is simply a saved query without a name. So, when a user toggles this history button, they'll be presented with a fixed amount of their most recent searches which will be displayed in the same way as saved queries except that the "Name" column will be omitted. Because we don't want to flood the database with what is essentially ignored data, we'll cap the search history at a certain number (most likely around 10 or 20 searches) and will delete the oldest search query when that limit is exceeded.

The last piece of functionality in the Saved Queries page is the ability to load any saved query into the search page. When a query is clicked on, it'll parse the information stored in the "Query" column (i.e. the actual boolean logic) and inject that information in the search page (and potentially Advanced Search page). While this is a challenging piece to implement (converting between a boolean expression and our own search format) and is potentially subject to change, we believe it's an integral part to making the user experience more friendly. Upon loading this query, the search page is populated and the user can modify any of the information displayed to them like a normal query. Once they've finished, if they would like to re-save this query, they can save it under a different name or delete the previously existing query and save it under its original name.

A piece of functionality that is still being discussed and may be added later on is the ability to save collections of papers. If we go forward with it, saving collections of papers will essentially be a combination of previous parts that we've already implemented: user tags and saved queries. To clarify, when a user wants to save a collection of papers to look at for later purposes, we can simply provide them with a button that will (after they've named their collection) automatically tag each paper with a user-created tag that signifies that it's a collection tag. Then we can display these saved collections in a similar format as the Saved Queries page and when we need to recall the saved papers, simply make an SQL query that only matches papers that have the unique collection tag. While this isn't a formal piece we intend to include yet, it should be a simple addition once all the other functionality is complete.

5.1.1.8 Add Administrator

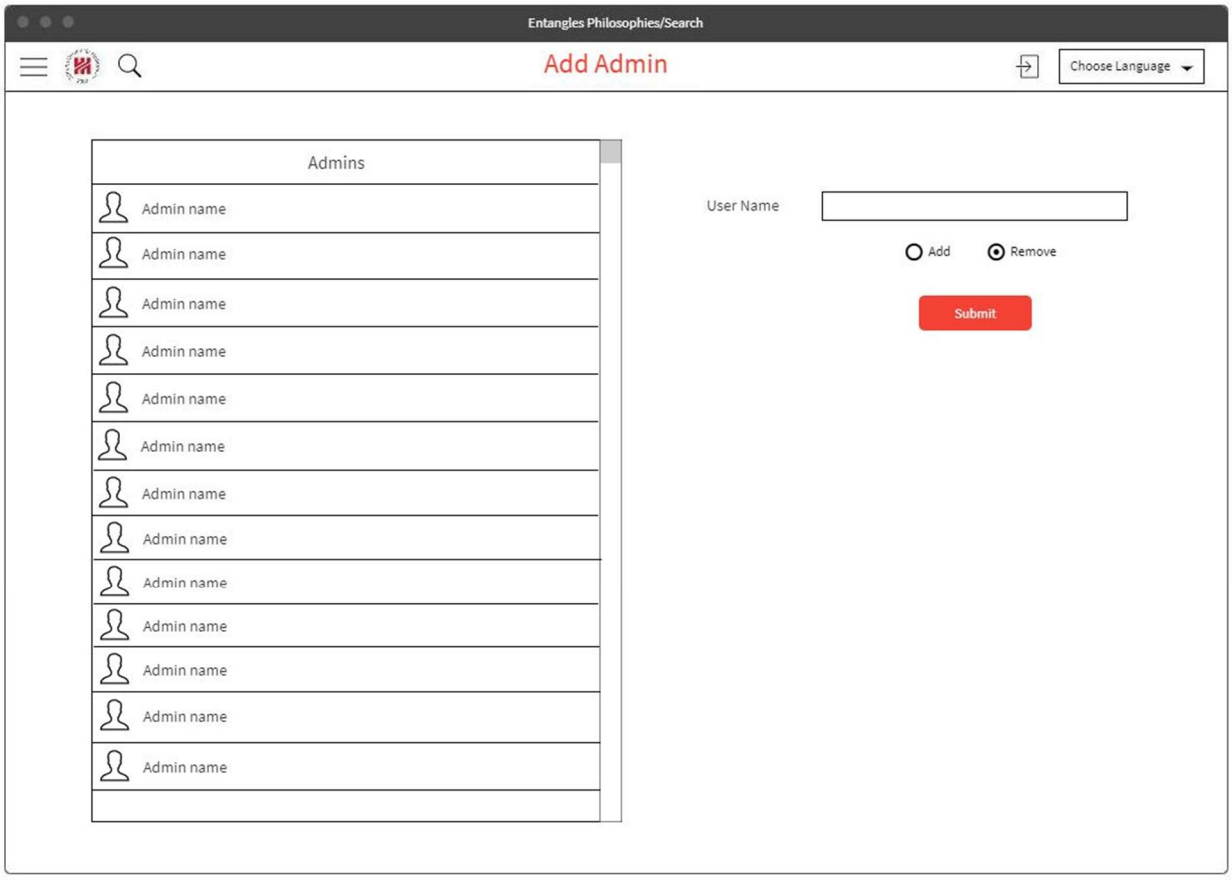


Figure 63. Adding new administrators to Entangled Philosophies

The users who can affect the data stored in Entangled Philosophies the most are, of course, the administrators. Adding and removing administrators is a necessary piece for our front end since we don't want our clients to have to manually edit the database anytime they need a new administrator and it would be unrealistic to expect only one administrator to handle all the paper uploads and tag assignments.

In the Add Administrator page, there are really only 2 pieces of functionality: adding/removing an admin and viewing the list of all current admins. If a current administrator wishes to add a new administrator, all they would need to do is input their username and select the "Add" option on the radio button below. Removing an administrator, however, can add an uncertainty to the integrity of the website. It would be bad if an administrator maliciously began deleting other administrators and they had no way of revoking administrator rights from such a user. To remedy this, we introduced 3

levels of permissions among the users: a super administrator, a regular administrator, and a general user. The super administrator will be the only user (the root use) who is allowed to remove anyone from administrator privileges while regular administrators only have the ability to add new ones.

5.1.1.9 Settings

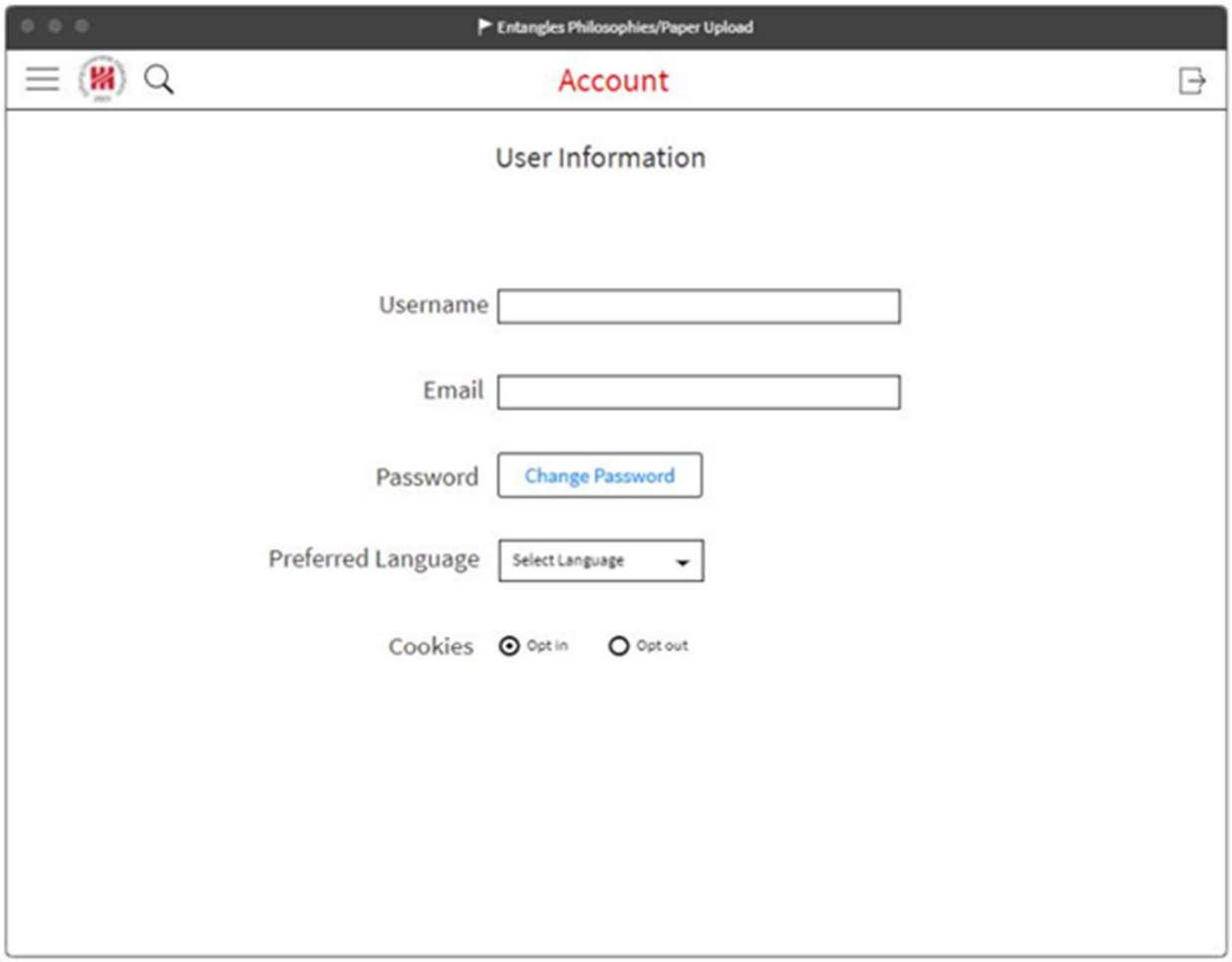


Figure 64. Account settings page

The Account Settings Page lets the users change their account information. If a user wants to change their username, then they can enter in the new username into the Username box. If a user wants to change their Email, then they can by entering the new email into the email box. The user also has an option to change their password, by clicking the “Change Password” button. Depending on if the user is happy or not with their Preferred language the user will have an option to change their preferred language. Changing your preferred language will make Entangled Philosophies website display in

that language when you log in from now on. Lastly you have the option to Opt in or Opt out of cookies.

5.1.2 Data Visualization

As a requirement for our Entangled Philosophies project, we will need a data visualization tool that will help us see all of the different tags contained in the database in word clouds, heat maps, etc. These tools will help us see how different tags relate to each other in many ways, whether by the country a certain tag is or to which hierarchy it belongs to.

For this aspect of the project we will be using D3JS, as this tool was made specifically to gather data and use this data to create visualizations such as the ones described above. We can populate D3JS with the data put into the database by the users of the Entangled Philosophies website application to keep visualizations up to date. This can be done by creating an API endpoint specifically made for data visualizations. The API endpoint can send each element of the database in JSON format, which can be easily read by D3JS.

5.1.3 Word Zones

The best way to explain how to use D3JS for our needs in this project is to bring up an existing project and explain why it works, and how we can modify this example to suit our needs. Here we will be using an example from the website ObservableHQ titled WordZones (Usable Alternative to Word Clouds) by Marti Hearst. [29]

Below we have “WordZones” taken from former president Obama’s State of the Union Address.

Figure 65. Word Zones

Each element, or word, in this “WordZone" is given a category value as well as a weight value. The category value will give each element a zone to populate, and the weight value will determine the word’s opacity as well as its size. The data for this “WordZone" is shown as follows:

```
sotuData = ▼Array(42) [
  0: ▶Object {word: "America", category: 1, weight: 10}
  1: ▶Object {word: "American", category: 1, weight: 8}
  2: ▶Object {word: "people", category: 1, weight: 7}
  3: ▶Object {word: "world", category: 1, weight: 4}
  4: ▶Object {word: "country", category: 1, weight: 4}
  5: ▶Object {word: "medical", category: 2, weight: 9}
  6: ▶Object {word: "cancer", category: 2, weight: 7}
  7: ▶Object {word: "research", category: 2, weight: 6}
  8: ▶Object {word: "cure", category: 2, weight: 4}
  9: ▶Object {word: "malaria", category: 2, weight: 4}
  10: ▶Object {word: "energy", category: 3, weight: 9}
  11: ▶Object {word: "climate", category: 3, weight: 8}
  12: ▶Object {word: "planet", category: 3, weight: 8}
  13: ▶Object {word: "oil", category: 3, weight: 5}
```

Figure 66. Word Zone Data

There are two words in the second category, “energy” and “oil” that have different weights. As we can see in the word cloud, these weights affect how visible these words appear. This “WordZones” example could help us visualize the hierarchical data that will be given to us for this Engangeld philosophies project. The weights could help us see the frequency of a given tag related to its location in a hierarchy for instance.

5.1.4 Heat Maps

We could also use heat maps to show where the locations in which more tags are more frequent than others, or how frequently users search for specific tags depending on which country they are from.

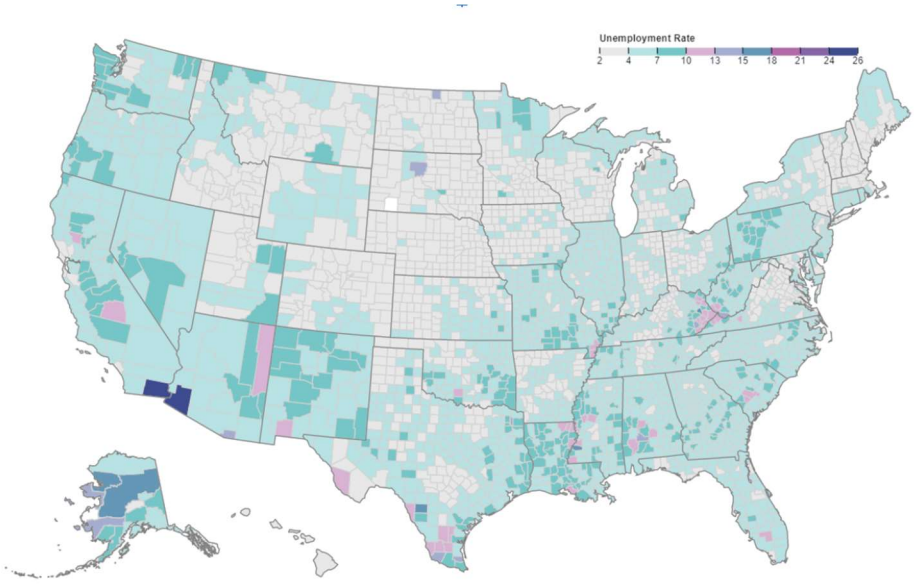


Figure 67. Heat Map

Here we have an example heat map also from ObservableHQ titled US County Heatmap by Chris Daly [30]. This heat map shows the unemployment rates across the United States, with a legend on the top right. On the raw data shown below, we can see how the rate is what is affecting each county's color on the heat map.


```
dataRaw = ▼Array(3219) [
  0: ▶Object {id: "01001", state: "Alabama", county: "Autauga County", rate: "5.1"}
  1: ▶Object {id: "01003", state: "Alabama", county: "Baldwin County", rate: "4.9"}
  2: ▶Object {id: "01005", state: "Alabama", county: "Barbour County", rate: "8.6"}
  3: ▶Object {id: "01007", state: "Alabama", county: "Bibb County", rate: "6.2"}
  4: ▶Object {id: "01009", state: "Alabama", county: "Blount County", rate: "5.1"}
  5: ▶Object {id: "01011", state: "Alabama", county: "Bullock County", rate: "7.1"}
  6: ▶Object {id: "01013", state: "Alabama", county: "Butler County", rate: "6.7"}
  7: ▶Object {id: "01015", state: "Alabama", county: "Calhoun County", rate: "6.1"}
  8: ▶Object {id: "01017", state: "Alabama", county: "Chambers County", rate: "5"}
  9: ▶Object {id: "01019", state: "Alabama", county: "Cherokee County", rate: "5"}
  10: ▶Object {id: "01021", state: "Alabama", county: "Chilton County", rate: "5.2"}
  11: ▶Object {id: "01023", state: "Alabama", county: "Choctaw County", rate: "7.9"}
  12: ▶Object {id: "01025", state: "Alabama", county: "Clarke County", rate: "11.1"}
  13: ▶Object {id: "01027", state: "Alabama", county: "Clay County", rate: "5.9"}
  14: ▶Object {id: "01029", state: "Alabama", county: "Cleburne County", rate: "5.5"}
  15: ▶Object {id: "01031", state: "Alabama", county: "Coffee County", rate: "5.6"}
  16: ▶Object {id: "01033", state: "Alabama", county: "Colbert County", rate: "6.5"}
  17: ▶Object {id: "01035", state: "Alabama", county: "Conecuh County", rate: "7.7"}
  18: ▶Object {id: "01037", state: "Alabama", county: "Coosa County", rate: "5.7"}
  19: ▶Object {id: "01039", state: "Alabama", county: "Covington County", rate: "6.7"}
  … more
]
```


Figure 68. Heat Map Data

This type of example is useful for us if we want to create a heat map of the globe to display the frequency of searches to a certain tag within a given country. The heat map we use does not have to be specific for the given county, it only would need to be as specific as the country.

For locating a user's location however, using the user's preferred language wouldn't be accurate enough for this type of heat map, as differing countries could have the same national language. This could be a problem when creating heat maps for English speaking countries such as Canada, USA, Australia, UK, etc. and for countries in Latin America excluding countries such as Brazil. A solution for this would be for an user to state their country of residence during the signup process. This would be something to consider going along with further implementation.

5.1.5 Word Clouds

For this Entangled Philosophies project, word clouds could be a great help in categorizing the frequency of certain words in a given text. In the instance where a philosopher were to upload a single document to the website, there could be word cloud functionality to help the philosopher find the appropriate tag given the frequency of the words in the text.

Here we have an example, again from Observable, titled World Cloud by Mike Bostock. [] Below is the given word cloud for the speech "I Have a Dream" by Martin Luther King Jr.

```
I am happy to join with you today in what will go down in history as the greatest demonstration for freedom in the history o
our nation.

Five score years ago, a great American, in whose symbolic shadow we stand today, signed the Emancipation Proclamation. This
momentous decree came as a great beacon light of hope to millions of Negro slaves who had been seared in the flames of
withering injustice. It came as a joyous daybreak to end the long night of their captivity.

But one hundred years later, the Negro still is not free. One hundred years later, the life of the Negro is still sadly
crippled by the manacles of segregation and the chains of discrimination. One hundred years later, the Negro lives on a
lonely island of poverty in the midst of a vast ocean of material prosperity. One hundred years later, the Negro is still
```

Figure 69. “I Have a Dream” Word Cloud

D3JS allows us to visualize every word in the “I Have a Dream” speech, with larger words being more frequent. This type of word cloud however seems to have quite a few drawbacks. For one, we have words appearing in the word cloud that have nothing to do with the topic of the speech. Words like “yes”, “sands”, “well” don’t give any context as to what the text fed to this algorithm is really about.

One way to mitigate this is to tell D3JS what frequent words aren’t meaningful for any word cloud. These words that don’t add any context to the text at hand are called stopping words. The problem with this solution is that even with stopping words, these words still make it into the word cloud.

```
stopwords = ▸ Set(183) {"i", "me", "my", "myself", "we", "us", "our", "ours", "ourselves", "you", "your", "yours", "yourself'
```

```
stopwords = new
Set("i,me,my,myself,we,us,our,ours,ourselves,you,your,yours,yourself,yourselves,he,him,his,himself,she,her,hers,herself,it
,its,itself,they,them,their,theirs,themselves,what,which,who,whom,whose,this,that,these,those,am,is,are,was,were,be,been,b
eing,have,has,had,having,do,does,did,doing,will,would,should,can,could,ought,i'm,you're,he's,she's,it's,we're,they're,i've
,you've,we've,they've,i'd,you'd,he'd,she'd,we'd,they'd,i'll,you'll,he'll,she'll,we'll,they'll,isn't,aren't,wasn't,weren't,
hasn't,haven't,hadn't,doesn't,don't,didn't,won't,wouldn't,shan't,shouldn't,can't,cannot,couldn't,mustn't,let's,that's,who'
s,what's,here's,there's,when's,where's,why's,how's,a,an,the,and,but,if,or,because,as,until,while,of,at,by,for,with,about,a
gainst,between,into,through,during,before,after,above,below,to,from,up,upon,down,in,out,on,off,over,under,again,further,th
en,once,here,there,when,where,why,how,all,any,both,each,few,more,most,other,some,such,no,nor,not,only,own,same,so,than,too
,very,say,says,said,shall".split(","))
```

Figure 70. Word Cloud stopping words

These are the stop words that were given to the “I Have a Dream” word cloud. This list of words however, is not long or extensive enough to include those words such as “yes” and “well”. All of these stop words would have to be added manually.

Another problem is that we’d also need stop words for any given language for any text being uploaded into the website. There are node packages that can give us lists of stop words however. The node package “stopword” for instance could be of use if we do decide to implement these types of word clouds into our finished project as “stopword” also supports 54 different languages. We would only need to tell users which languages are not supported.

The usage of this package is detailed as in the following code provided by the stopword package page:

```
sw = require('stopword')
const oldString = 'a really Interesting string with some words'.split('
')
const newString = sw.removeStopwords(oldString)
// newString is now [ 'really', 'Interesting', 'string', 'words' ]
```

In the “I Have a Dream” word cloud, words were defined using the following functions:

```
words = ▶ Array(811) ["happy", "join", "today", "go", "his
```

```
words = source.split(/[\s.]+/g)
  .map(w => w.replace(/^["‘"\-–()\[\]{}]+/g, ""))
  .map(w => w.replace(/[;:.!?()\[\]{},"'’”\-–]+$/g, ""))
  .map(w => w.replace(/['’]s$/g, ""))
  .map(w => w.substring(0, 30))
  .map(w => w.toLowerCase())
  .filter(w => w && !stopwords.has(w))
```

Figure 71. Stopping words manually

If we were to use the “stopword” package, this would not be necessary. Instead the variable for words would be:

```
words = sw.removeStopwords(source)
```

5.1.6 Radial Tidy Tree

One of the types of data visualization that was recommended by our sponsor to use for the project is the Radial Tidy Tree. An example was provided in the form of an ObservableHQ project titled “Radial Tidy Tree” by Mike Mostock. [31] D3 has its own tree layout which can help us easily create tree diagrams that we can visualize. This tree layout is useful as it is able to produce “tidy node-link diagrams of trees using the Reingold–Tilford “tidy” algorithm, improved to run in linear time by Buchheim et al”. [32]

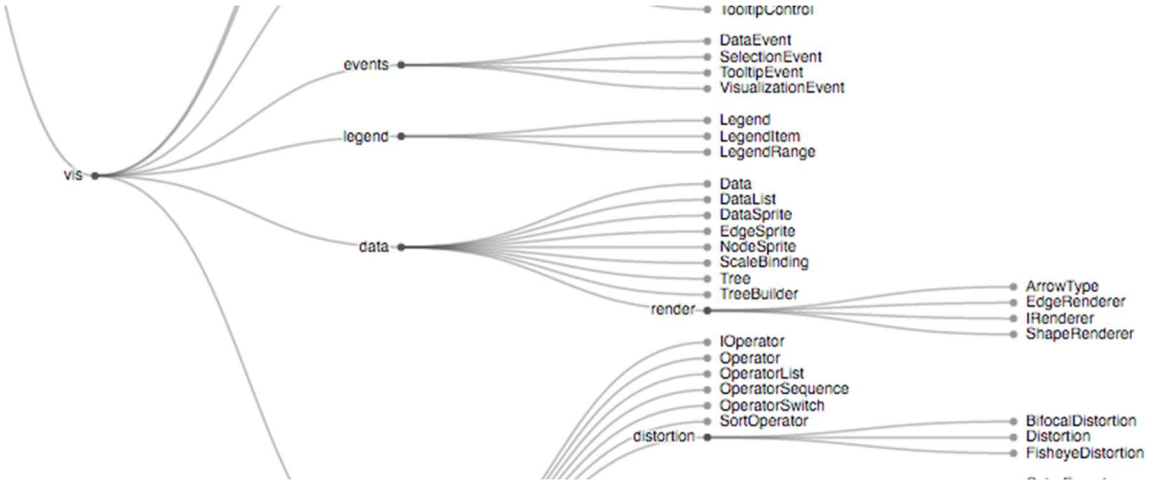


Figure 72. Listed Tidy Tree

The image above shows an example of a standard D3 tree. This type of visualization is useful to our needs as we will need a way to visualize tags in a manner that it shows all of the hierarchical information of any given tag stored into the database. The tree would have tags as a leaf node in the tree, and the category or subcategory belonging to this tag would therefore be the node's parent. This is a better way to visualize the hierarchy of tags stored in the database because we can see the ways tags and their categories are linked to each other in a concrete way, rather than the abstract type of linking that we see in Word Zones for instance. With Word Zones, one would be able to see how tags are related if they were put into a cluster, but one could not be able to visualize how these clusters relate to each other.

The radial version of the Tidy Tree would be able to save visual space for the user, as well as keep the information more compact so that it easier to follow all of the categories for the tags. To have a listed Tidy Tree would mean that if an user were to visualize a large amount of tags, they would have to scroll through the page, which makes it more difficult to keep track of what is the parent of a tag and how all of the categories might relate to each other.

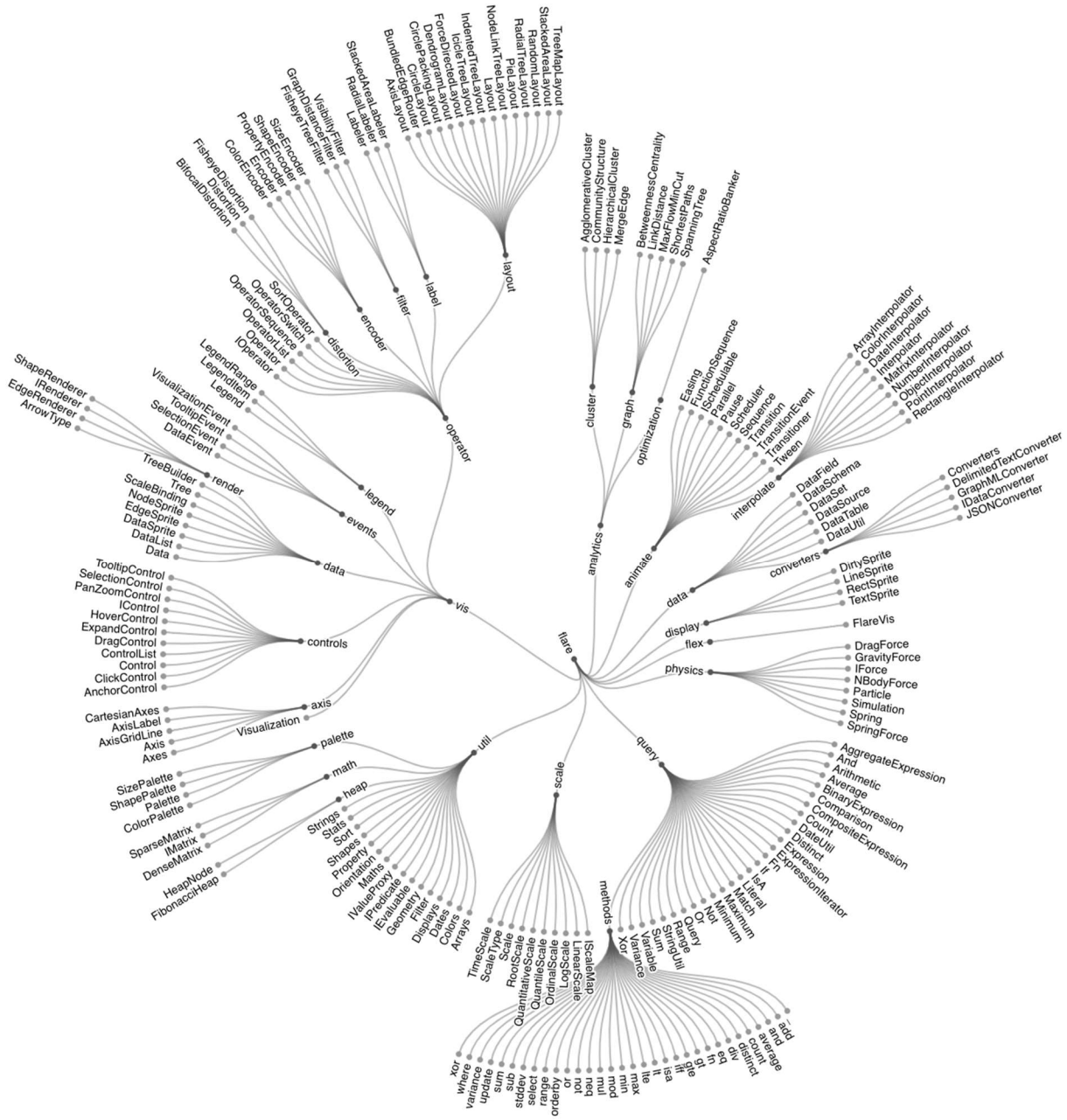
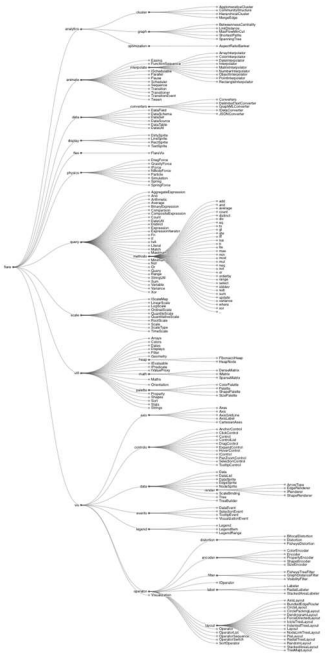


Figure 73. Radial Tidy Tree

The image above displays the example Radial Tidy Tree provided by the Observable project. In the implementation of this for Entangled Philosophies, each tag would be displayed and easily visible on the outside of the radial tree, making it easy to follow which categories these tags are related to on the inside of the tree.



For comparison, the miniaturized image to the left is an example of the same tree if it were to be displayed as a list. The tree would be around double the size of the radial tree despite them both all having the same nodes. The radial tree would greatly enhance the user experience although the radial tree also has its own drawbacks. For one, the readability on the radial graph suffers because of the nature of having to display the text in a circle. The text displayed close to the y-axis of the circle (under the assumption that the center of the circle is at (0,0) in an xy-plane) is completely tilted sideways due the nature of it being on a vertical.

The way we can populate the tree in D3 is by storing the data in a JSON file to be interpreted by D3.

Figure 74. Listed Tidy Tree

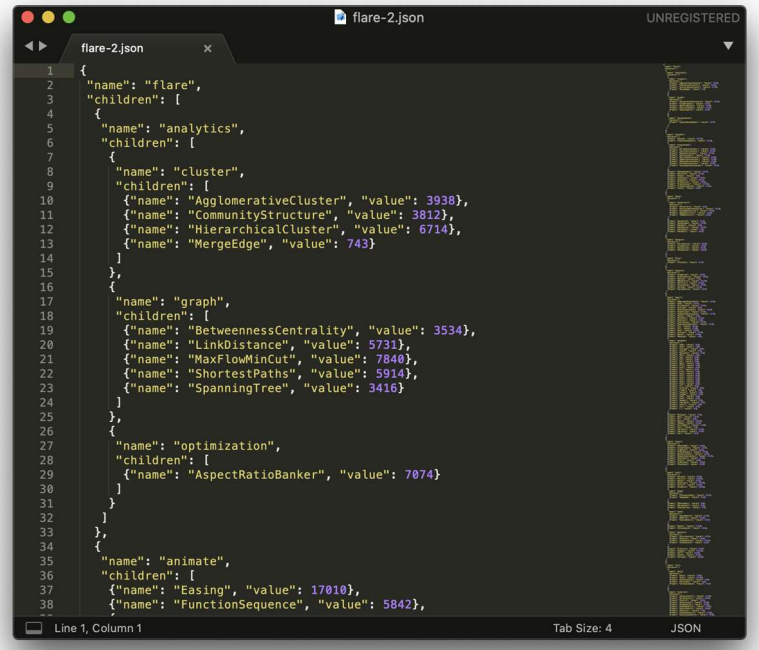


Figure 75. Tree JSON Data

The image above is a snippet of the JSON file that is being fed into D3 to produce the data hierarchy. For the project, there should be a JSON file such as the one shown that

contains all of the tags specified for the data visualization. It would only need to list all of the nodes to be displayed.

Figure 76. Parsed JSON Tree Data

The image above shows the data that is being taken by D3 in order to display the Radial Tidy Tree. There is a variable “data” that creates a hierarchy from the given JSON file and sorts this hierarchical data in alphabetical order. The root of the tree from the example, “flare”, is stored as an object which holds the information about the tree such as the children “flare” has as well as every child’s children and so on.

Figure 77. Nested Nodes

The image above details how all of the data is stored as nested objects. There is also a height field as well as a depth field for each of the nodes in the tree. The height field dictates the height of the tree at that given node, for example the node “vis” has a height of 3 while the node “flare” has a height of 4, meaning that vis is a layer below flare on the tree. Since “flare” is the only node at a height of 4, we can assume that “vis” is a child of “flare.”

The depth field indicates which layer of the tree the node is located, take for instance the node “flare”, this node has a depth of 0 which would make it the root node. The depth of the node “vis” has a depth of 1, which would indicate that this node is a direct child from the root. Aside from this type of categorizing being needed to maintain the structure of the tree, it could also become useful when we implement any sort of algorithm into the visualization, whether it is removing or adding elements into the tree to further aid researchers trying to see the relations of specific hierarchies, or any other type of process that requires an algorithm. Note how these fields were not provided in the JSON files, D3 provides all of this information after creating the hierarchy.

5.1.7 Relational Data Visualization

A second type of D3 data visualization was suggested by the sponsor to use in Entangled Philosophies. This type of data visualization examines how different things relate to each other and displays them in a diagram that is easy to follow. The example comes from an Observable project titled “Mobile Patent Suits” by Mike Bostock. [31] This example project

shows us a “view of patent-related lawsuits in the mobile communications industry.” [33] This visualization has every mobile communications company that has been involved in a patent-related lawsuit representing a node in the diagram. The way one can tell how companies are related, meaning which companies have created a lawsuit against another company, is by having an arrow point from one node to another. These arrows also have different colors attached to them which are prescribed by a legend.

Figure 78. Mobile Patent Suits

From the diagram and the legend, one can deduce that the node from Oracle having an orange arrow pointing at Google means that Oracle served Google a lawsuit, further research suggests this is known as the Google LLC v. Oracle America, Inc. lawsuit. As one could see, this type of data visualization is really useful to get to know how different nodes in the diagram interact with each other, which could be very useful when we want to see how different tags in the Entangled Philosophies database also relate to each other.

```
nodes: ▼Array(20) [
  0: ▶Object {id: "Microsoft"}
  1: ▶Object {id: "Amazon"}
  2: ▶Object {id: "HTC"}
  3: ▶Object {id: "Samsung"}
  4: ▶Object {id: "Apple"}
  5: ▶Object {id: "Motorola"}
  6: ▶Object {id: "Nokia"}
  7: ▶Object {id: "Kodak"}
  8: ▶Object {id: "Barnes & Noble"}
  9: ▶Object {id: "Foxconn"}
  10: ▶Object {id: "Oracle"}
  11: ▶Object {id: "Google"}
  12: ▶Object {id: "Inventec"}
  13: ▶Object {id: "LG"}
  14: ▶Object {id: "RIM"}
  15: ▶Object {id: "Sony"}
  16: ▶Object {id: "Qualcomm"}
  17: ▶Object {id: "Huawei"}
  18: ▶Object {id: "ZTE"}
  19: ▶Object {id: "Ericsson"}
]
links: ▼Array(28) [
  0: ▶Object {source: "Microsoft", target: "Amazon", type: "licensing"}
  1: ▶Object {source: "Microsoft", target: "HTC", type: "licensing"}
  2: ▶Object {source: "Samsung", target: "Apple", type: "suit"}
  3: ▶Object {source: "Motorola", target: "Apple", type: "suit"}
  4: ▶Object {source: "Nokia", target: "Apple", type: "resolved"}
  5: ▶Object {source: "HTC", target: "Apple", type: "suit"}
  6: ▶Object {source: "Kodak", target: "Apple", type: "suit"}
  7: ▶Object {source: "Microsoft", target: "Barnes & Noble", type: "suit"}
```


Figure 79. Mobile Patent Suits Data

As for the data in this type of data visualization, one would only need to know which nodes are related to each other as well as how they're related to each other.

5.1.8 Bubble Charts

The last type of D3 data visualization that was recommended for use on this project according to the sponsor is Bubble Charts. The example provided comes from an Observable project titled "Bubbly Chart" by Mike Bostock. [34]

Figure 80. Bubble Chart

Each circle in the example bubble chart represents a file, the size of the file represents how large the file is. This value that dictates the size of the circle can be changed into something that could suit the needs of the Entangled Philosophies project, such as frequency of a tag, which would make it similar to a heat map without the map. With this type of data visualization we could see how frequent a certain tag is used as well as how this tag relates to other tags, which could be grouped by a color.

Figure 81. Bubble Chart Data Snippet

In this code snippet one can see that all of the circles that represent these data points in the bubble chart have the same color, blue.

All of the examples given above were created using D3JS. Implementing the project with D3JS will save us much time as members of our group are familiar with JavaScript and can get used to working with D3JS in a timely manner. D3JS also gives us flexibility when working with different types of visualizations, from word clouds to heat maps. The build in tree layout for D3JS would also help greatly with meeting the requirements from the sponsor requirements.

5.2 Back End

The back end is designed to connect the information in the database to the front end user. A user can interact with the webapp without seeing behind the scenes. The back end uses an Application Programming Interface (API) to transfer the user's input data to a database query or update. Then it takes the return value of that query or update and again transfers it back to the user who then sees the change or result on the webapp.

In this case, the database is MySQL and the webapp uses React, written in JavaScript. This means the API will convert JavaScript input, which is in JSON format, to a SQL query, and then convert the return value back to JSON to be used by the HTML back at the React app.

The language used for this API is PHP, fitting with the LAMP stack standard used. Although React is not typically included with LAMP, it is HTML running on an Apache server just the same. In this way, PHP is able to be used for connecting the two in the same way. The API requests are made with an XMLHttpRequest (XHR), which posts a header to a URL where the PHP files are stored on the server.

5.2.1 Password Hashing

Knowing that the language being used for the API will be PHP, there will have to be a way to implement a password hashing system that is suitable for PHP. In Section 4.5.2, the type of password hashing that was found to best suit these needs for Entangled Philosophies resulted in Bcrypt.

5.2.2 Login and logout

A user needs the ability to have an account in order to store personal data such as saved queries, preferred language, and custom tags. For this we need multiple API endpoints to handle logging in, logging out, registering for a new account, and deleting the account.

```php
<?php
// Usage 1:
echo password_hash('rasmuslerdorf', PASSWORD_DEFAULT)."\n";
// $2y$10$xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// For example:
// $2y$10$.vGA1O9wmRjrwAVXD98HNOgsNpDczlqm3Jq7KnEd1rVAGv3Fykk1a

// Usage 2:
$options = [
  'cost' => 11
];
echo password_hash('rasmuslerdorf', PASSWORD_BCRYPT, $options)."\n";
// $2y$11$6DP.V0nO7YI3iSki4qog6OQI5eiO6Jnjsqg7vdnb.JgGIsxniOn4C
```

Figure 82. Example of PHP password hashing with bcrypt [35]

Once the user inputs their username and password to login with, an encryption function is first called to hash the password for a more secure login. For this a bcrypt hash is used (Figure 82), which is included directly in PHP through its password_hash function [36]. The encrypt endpoint only takes the password and returns the hashed version through bcrypt. This is the password that would be saved in the database, and will then be used to query it.

The login function is just a simple query of the database which checks if a row exists in the user table with the given username and password. If successful, it returns a success, and on failure it returns an error. A failure can also be given if the login is found but not yet verified. Based on this boolean response, the front end can display the correct result. On success, the user will be brought to the landing search page with the login stored in a cookie. On failure, the user will stay on the same page and an error for invalid login will be displayed (Figure 83).

When the user would no longer like their login stored in the cookie or would like to change accounts they would then logout. Pressing a logout button call a logout function, which removes the login information from the cookie or current session then sends the user back the landing search page. No checking of the database is needed and this can all be done front end only.

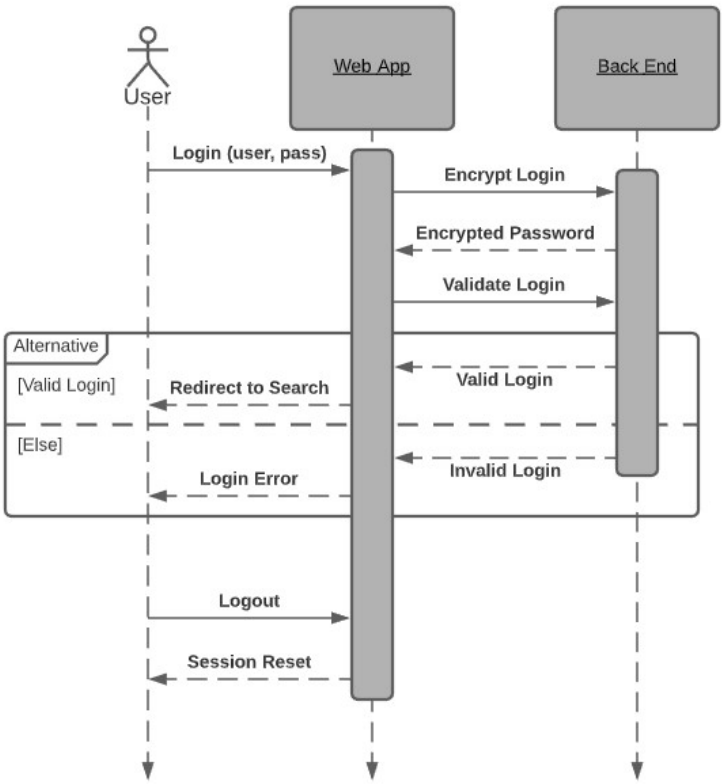


Figure 83. Login and logout

5.2.3 Register

If the user does not yet have an account to login with, then they must create one. This will be done with the register endpoint which is then followed by the email verification process (Figure 84). First the username, password, email, and preferred language are entered and submitted. Then the info goes through the API. First, the hashing endpoint is called just like in the login function. This creates the encrypted password that would be stored in the database. Now it attempts to register the account by posting a new row to the database table for users. This is the next check where it verifies that the username as well as the email do not yet exist. If either of them exist, it will return an error and the new row is not added. If neither exist, the account is successfully added to the database, with its is_verified boolean set to false as a default, and their verification code set to a randomized value.

That is all the register endpoint does on its own, but in order for the user to fully use the account, it must then be verified. The verify email is sent out to the email given after the new row is added to the table with a link to verify the account. Once the user clicks the link, it calls another API endpoint to check if the code in the link matches their verification

code in their table row. Next it changes a user's is_verified value to true. Then it sends the user back to the login page with a notification that their account has been verified. Now they can login successfully as shown in the previous section.

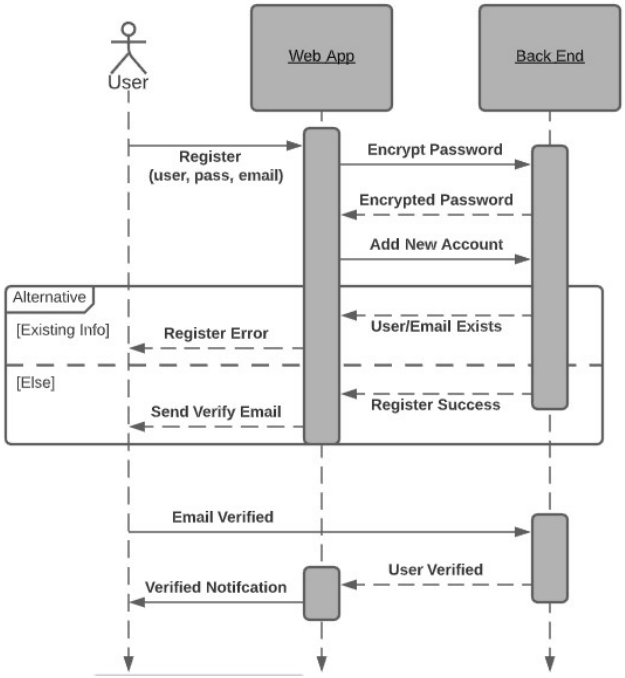


Figure 84. Register

5.2.4 Account Settings

Now that the user is able to register and login, they have access to user specific features that are saved in association with that account. The first of which are some user settings such as preferred language and opting out of cookies. The current session will store the logged in user, so when the settings are saved it will update the user table row for that user with all the new information on the settings page.

The user's settings are later loaded to read their preferences. This is used to know what to display, such as checking their preferred language to pick the display language. For this is another basic fetch of the user's associated settings stored in their table row (Figure 85). All settings are returned in JSON format to be read from and used to display the current session. Updating password happens with its own endpoint, sending the user id and new password to be hashed and updated. This is so that we can update just the password without needing to save the settings.

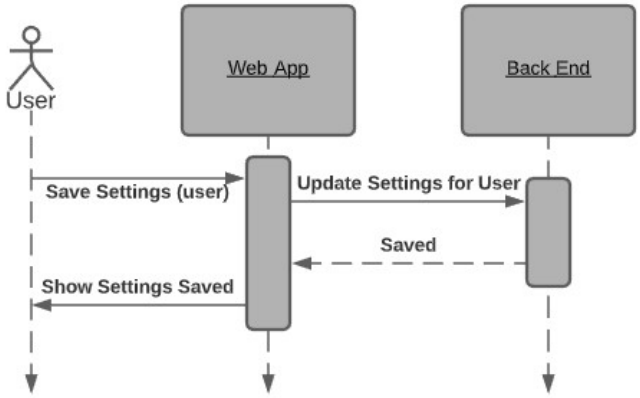


Figure 85. Save account settings

There is one other account related API endpoint that is also very simple. There is an option in the account settings that allows you to delete your account. When the user presses the confirmation button, it sends their user id as stored in the session, similar to the saved settings, to the delete account endpoint (Figure 86). It then accesses the database and simply removes the row for the given user.

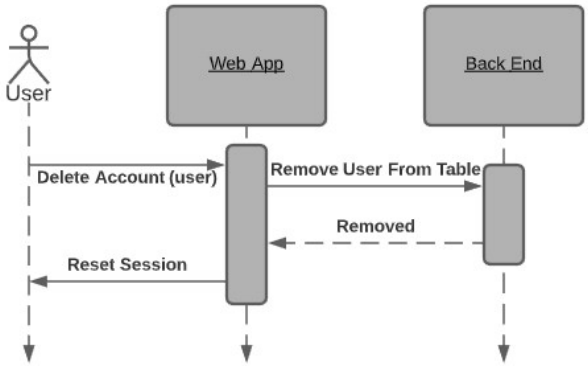


Figure 86. Delete account

5.2.5 Email API

As touched on before, email API functionality is needed for some features of the website related to the user account. Email verification was covered previously, however this functionality is also needed for password recovery. This works by calling an endpoint that will randomly generate a password to hash and update in the database, and then send an email to a given user with the new password so they can set a new one in Settings (Figure 87).

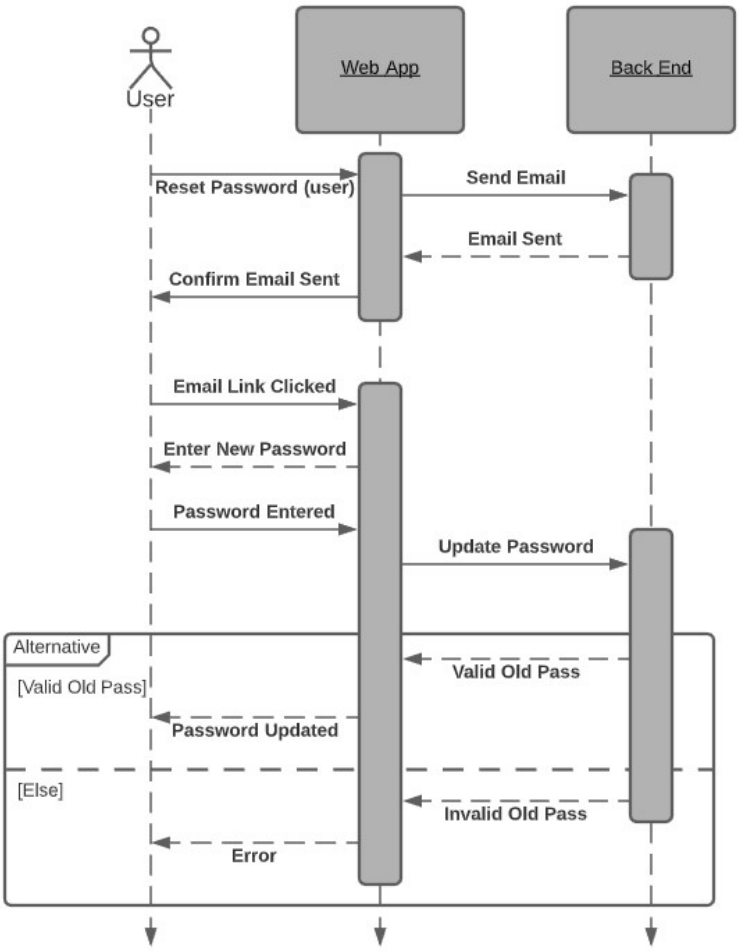


Figure 87. Reset password

This email functionality will be accomplished through the use of PHP's mail function. In the reset password and verify email endpoints, the PHP code will call this mail function to send out the email to the user's email.

The link sent for email verification directs to its respective API endpoint with reference to the user in a header, which then does the proper updating for that user after checking if the link is valid with a matching verification code for that user. Originally, as shown in the image above, reset password was going to also send a link but ended up simply sending the randomly generated new password. JWT was also planned to be used but didn't end up being implemented.

5.2.6 Search Query

The searching and filter system makes use of complex filters which are represented by SQL queries. This search is used to sort through the papers in the database. Since all of this complexity and conversion from user input to SQL query is done on the front end through JavaScript, the resulting API endpoint is actually more straightforward. All it does is query the database with a select command given the input string. The returned array of results is used to show the user their search results (Figure 88). By default, if no search query is given, it selects all papers to be shown.

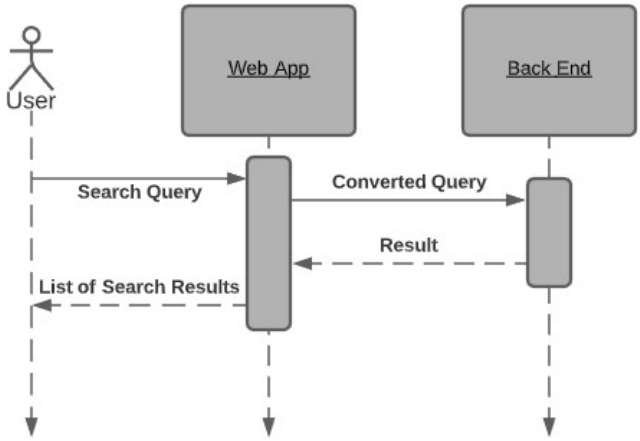


Figure 88. Search and filters

For the basic search, you can type a string into the search box to search through any of your selected columns such as name of paper and language. The selected columns are the tables to select from, and the input string is what to select. This conversion is done in JavaScript before sending the final SQL query to the API which just runs that query.

For the filters, the search becomes more complex. This can make the SQL conversion actually more obvious. The tags selected in the filters are combined with the logic given such as AND and OR, and then the same tables from before are searched, this time checking each paper of the result of the original search. It will check the associated tags of each paper when running the query. All this conversion is still done in the front end, and once again the endpoint remains the same as only a select query.

Finally the advanced search is the closest form the query actually being done, but still needs some conversion to protect against injections and invalid tag names.

5.2.7 Tagging Papers

One of the major features of the papers that allows them to be filtered and searched so effectively are the tags that are put onto them. This includes their meta data as well as the custom tags created by the site admins and added to each paper. The custom tags are also put into categories of tags such as country of origin, religion, and so on. These can only be created by admins and are assigned to the papers as they are added and can be edited or removed afterwards. Users can create their own separate private tags and categories as well.

The admin tags and meta data are treated the same by the back end, and just displayed differently to the user. Each paper has a list of tags associated with it, and each of those tags has its category stored in its entry in the tags table of the database. If it is an admin created tag, it will be visible to all users and not able to be edited by non-admins. If it is a user created tag, it will be only changeable and visible to the user that created it.

When one of these tags is added to a paper, it must be an existing tag. To check this, the tag name typed by the user is put into an endpoint to verify the tag's existence along with their username. First it needs to check if the user is an admin by checking the permission level of the row containing that username. If they are, the tag's owner uses id 0. 0 is used for all admin tags to display them together with tags added by other admins. If they are not an admin, the owner is just the user id. The owner is needed because it is possible different users have tags of the same name. Then it checks the tags table for a tag of that name and owned by that person, and if it's there then the tag exists and the id of the tag is returned (Figure 89).

Once the tag is confirmed to exist, it can be added. The ids of the tag and paper are added to a relational table. The tags associated with a paper are stored in this separate relational table where each row is a paper id and the tag id that the paper is tagged with. This means there is a row with the same paper id for every tag it's tagged with, and a row with the same tag id for every paper it has been put onto. The endpoint to do this is an update function that takes the tag id returned from the verify function and the name of the current paper, and then adds that tag id to the id of the paper name given. Removing a tag takes the same inputs, the paper name and a tag id, and will just remove the row in the relational table that matches that papers id and the tag id.

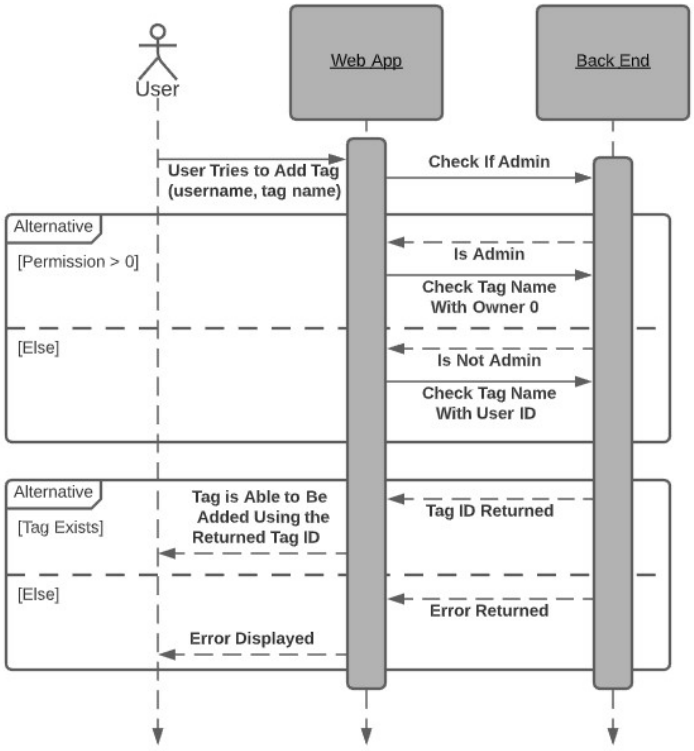


Figure 89. Verifying a tag's existence

5.2.8 Adding and Removing Tags and Categories

The tags able to be added or removed from a paper as described in the previous section are selected from a set previously created by admin or user accounts depending on the case. They can be added, removed, or edited. This only requires the use of two endpoints however, because add and edit can be combined. When the function verifies there is no duplicate being added, this is checking if it already exists and therefore is located to be edited. In this case it can be updated instead of added to.

If the user adds a new tag, a new tag id is generated and stored in a new row in the tags table along with the user's id as the owner of the tag. This endpoint will return the newly generated tag id, and then a separate endpoint is called to add a translation. A normal user only needs to supply the tag name in one language, so the name they typed will be added to the tag translation table with the returned id, the text entered, and a placeholder Default language indicating this is not an actual translation. Finally, the existing category which the tag is in should also be given, and its category id will be stored with the tag. The category name must also be verified just like the tag were, using a separate but identical process to the one in Figure 89. This returns the confirmed category id to be used in the adding function (Figure 90).

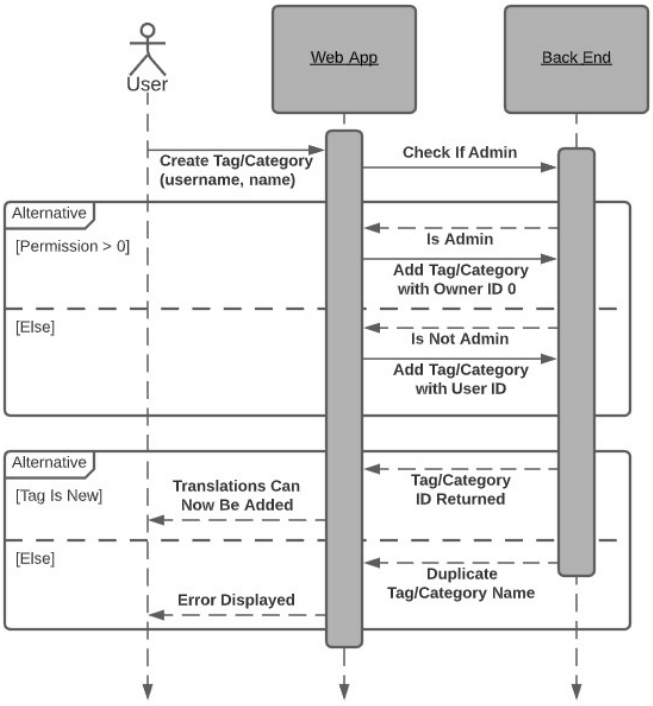


Figure 90. Creating a new tag or category

When an admin adds a tag, it requires the tag's category as well as the tag's translation into all supported languages to be input. This makes use of repeated calls to the add translation endpoint (Figure 91). Instead of Default, the specified language along with its text is added to the tag translation table associated with the tag id it's translating. The owner id of 0 will be used for admins in place of their user id.

If the existing name is already found for the given input in the add endpoint, then it will be edited instead. The row found containing the given tag or category name matching with the language it should be translated to, which also includes Default, will have its information updated with the given new information. The remove endpoint will first delete all associated translations with that id, and then deleting the row associated with the selected tag name or category name.

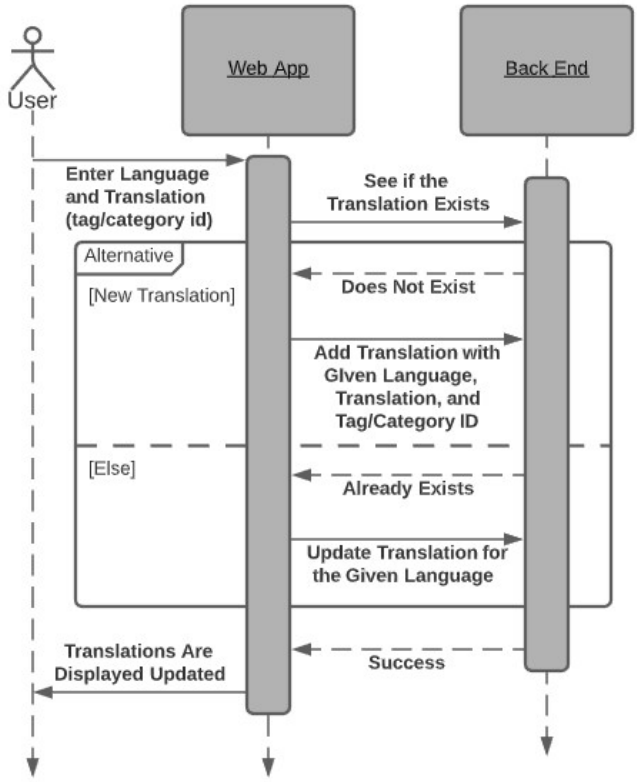


Figure 91. Adding a translation to a tag or category

5.2.9 Displaying Tags

Once the tags have been created and applied to their respective papers, they must be displayed correctly. The admin tags must not be editable while the user tags will have buttons to add or remove them. There must also be a distinction in displaying the metadata versus all the other tags. The tags all fall under certain categories, so upon retrieving the appropriate categories, all tags under them can then be displayed. These three types of tag categories each have an endpoint to retrieve them.

The get categories endpoint will check the tags table for any tag with owner id 0, indicating it was created by an admin, and only takes distinct category id values. This will find all categories of the tags owned by id 0. Later it can be checked if those categories returned in an array have an id of -1, which means it is metadata. This gets the groups that can't be edited by normal users, but are displayed as normal tags and not metadata. The result is separating all the non-metadata and metadata tag categories created by admins (Figure 92).

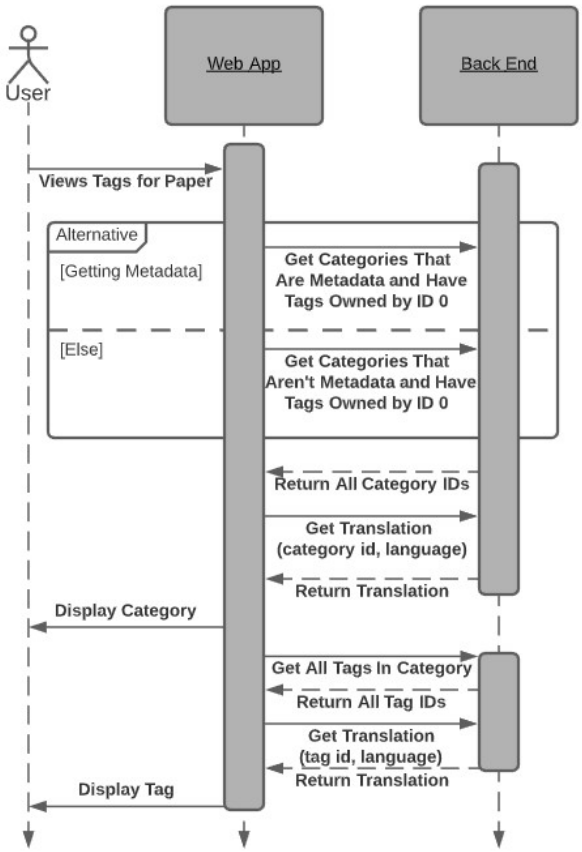


Figure 92. Use of endpoints to view all metadata and Non-metadata admin tags on a paper

The main change from the system seen in the above image was the combination of the endpoints to get the user or admin categories. By passing in the user id 0, the endpoint will know its an admin and only get admin categories. Otherwise, it gets them for the given user. The original get user categories endpoint can still be seen below (Figure 93).

Once all categories are loaded in, their names can be displayed in the correct location, and then another endpoint can be called to fetch all tags associated with that category. The categories are already sorted, so this endpoint will work the same for all 3 groups. It takes in the category id as given from the previous functions, checks the tags table for all that have that id, and then returns the list of them. This functionality can be combined with the get category code in order to only need one API call. This is the change that ended up being used, with one the get tags endpoint getting the categories first as well.

The last step in displaying these tags in the correct way is to make sure to use the correct language. There is an endpoint that will take the id of a tag or category along with a

language and get the translated name back. The language can be gotten from the user's preferred language, which is stored in the user table and can be retrieved with the get settings endpoint from a previous section. This function can be called to translate and display the correct text for each of the admin created tags in the user's selected preferred language. It does not need to be used on user created tags because they do not have associated translations.

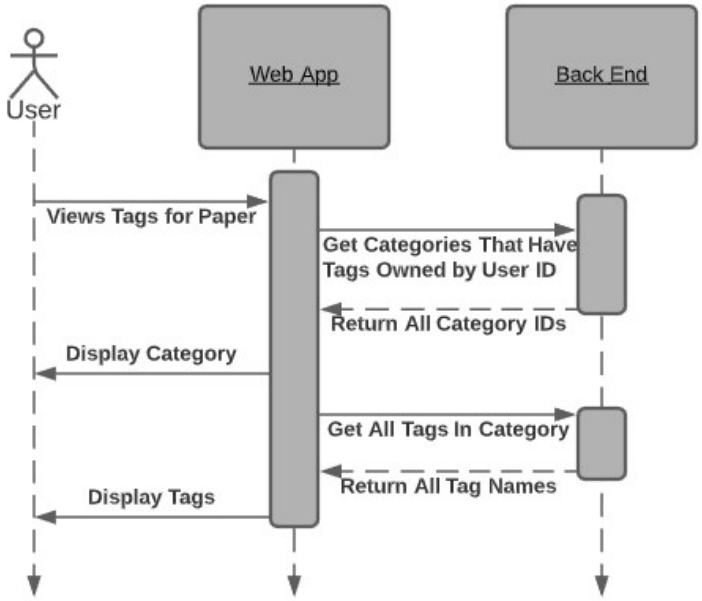


Figure 93. Viewing all of the user's private tags on a paper

5.2.10 Adding and Removing Papers

The tags that are added to a paper by an admin are initially done upon the addition of a new paper. This is when they would upload the associated file, name the paper, and then define all metadata and tags. The admin user types in all the information, which would make use of the verify tag and verify category endpoints from earlier, before submitting the creation of the new paper.

Once this button is pressed, the add paper endpoint gets called (Figure 94). It will take in the name of the paper, and the name of the file uploaded if any, and add this information to a new table row in the papers table with a newly generated paper id. The new id is then returned by a result of this function. The add tag to paper endpoint from the previous sections took in the name of the paper to add to, which is what is called immediately next. For each tag entered by the admin user, it must go through the verify function to get the

tag’s id, and then the add tag endpoint is called, taking in that tag id along with ther paper name.

To remove a paper, the paper id is given to the remove paper endpoint, which will first remove the file upload if any from the given paper’s URL, then it removes the row from the table (Figure 95).

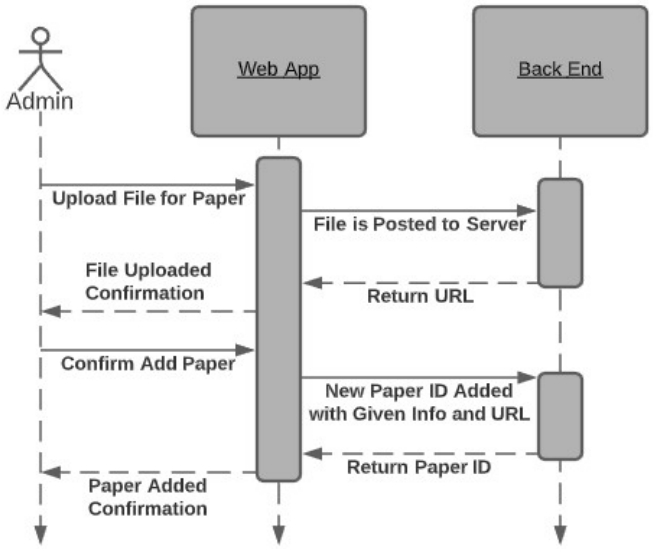


Figure 94. Endpoint to add a new paper along with the file upload

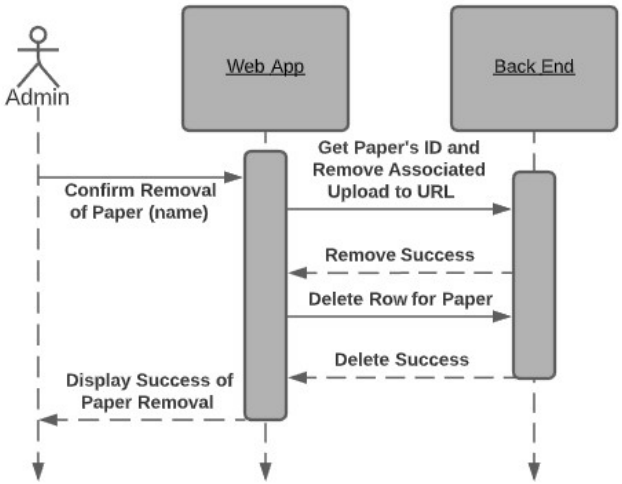


Figure 95. Removing a paper after an admin confirms to do so, removing the file upload with it

Editing the paper is also possible, which basically just sends all the information given for a paper to be updated again. The title and file upload name are set to the ones given, and any newly added tags are added with the add tag function just like before.

5.2.11 Saving Queries

A user can save any query performed on the search page to be reused later for convenience. These queries are named, and the name can be edited. Each query can also be deleted. This gives 3 endpoints once again for add, remove, and edit and all work as expected. A query is the same as the custom search format, showing the exact logic rather than listing tags.

When adding a query, that query text is sent to a save query endpoint, just the same as the search endpoint mentioned, except it also sends the current user and the given name for that query (Figure 96). If not a custom query, it will have to be converted to the same format to be displayed, which is done on the front end. There can't be queries with duplicate names, the function first checks if the given name matches any query with the same user id. If it does, it gives an error. If not, that text is added to a query table relating the user id with the given query and name.

The delete query endpoint takes the user id and the query name to be deleted, and removes the row that matches the criteria. It is possible to delete multiple queries at once using check boxes. This however doesn't use a separate function and instead calls the delete query endpoint for each query selected.

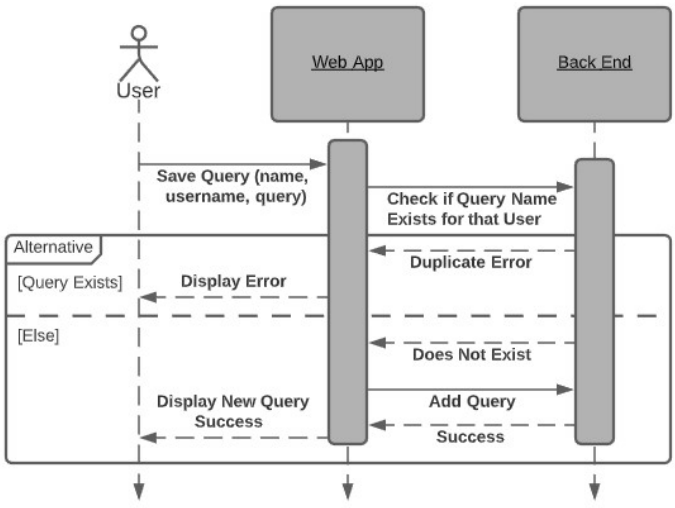


Figure 96. Saving a new query for a user

The edit query endpoint only will update the name of the query, as the query itself can't be changed. It will take the current name and user id, along with a new name. Again the row matching the name and user id is found, and then that name is updated to the new name.

Finally, a get queries endpoint takes a user id and returns an array of JSONs (Figure 97). The array includes all of the queries associated with that user id, and all other information storied for each query such as the name. As seen below, the original plan was to return two separate lists, but sending one JSON back with all the information was much easier and more efficient.

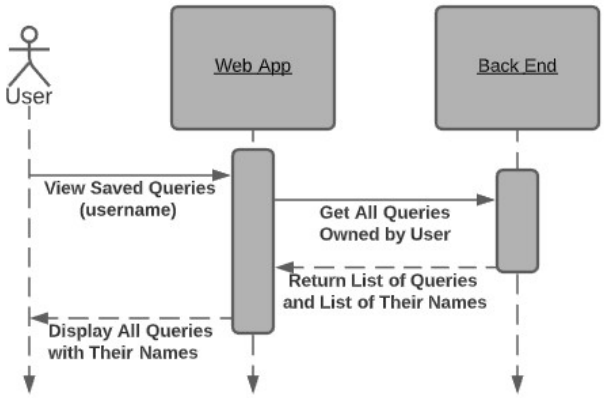


Figure 97. Loading all the user's saved queries to be displayed

5.2.12 Adding and Removing Admins

Above admins in permission levels is the super admin, of which there is only one. This super admin can add new admins, but additionally can remove them as well. The other normal admins can only add admins, but not remove them. All admins also have a list of other admin usernames for reference. A get admins endpoint gets the list of all admin usernames by searching the user table where their permission level indicates they are an admin or super admin (Figure 98). A name can be clicked in this list to autofill it into the remove admin text box.

The add admin endpoint takes a username entered into a textbox and attempts to make that user an admin (Figure 99). It first checks if the user exists by searching the table for any user with that name, which cannot be duplicate. If they do, it then sets them to an admin permission level. Afterwards the admin table can be refreshed by calling the previous function.

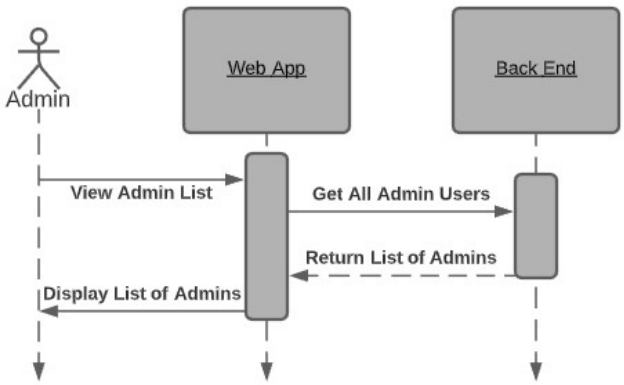


Figure 98. Getting a list of all admins to be displayed

The remove admin endpoint also takes a username from the textbox. This functionality is only accessible for a super admin. The permission level of a user found in the user table with the given name is set back to the normal level.

Because of the differing features available for different permission levels, such as the remove function for super admins, or the add paper page for admins, there needs to be a way to determine the permission level of the user on its own. A get permissions endpoint takes the currently signed in user and finds them in the user table. The permission level associated with that user is returned and can then be used for display purposes, to lock users out of functionality they are not able to use.

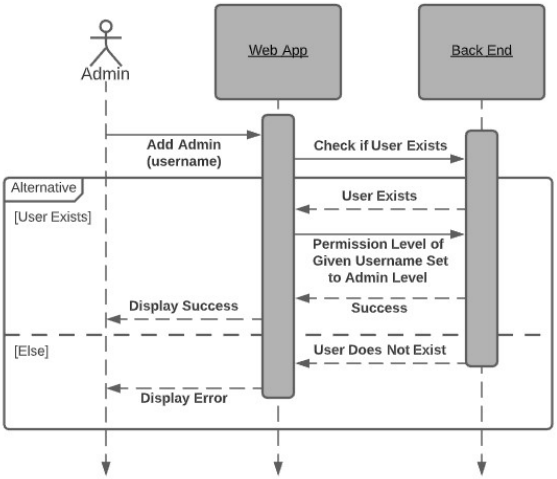


Figure 99. An admin adding another new admin from a username

5.3 Database

5.3.1 Database Management System

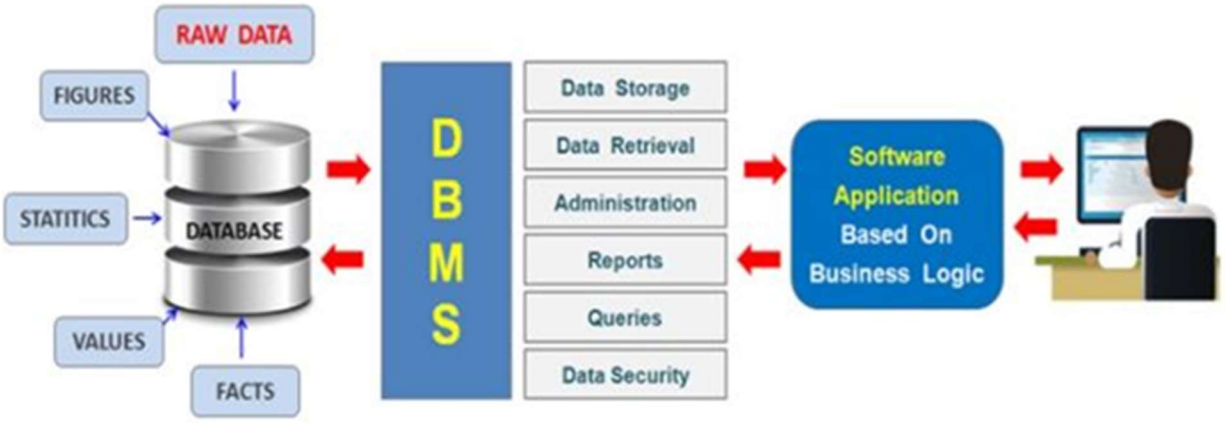


Figure 100. Database Management System Overview Example

Entangled Philosophies will use a MySQL Database Management System. After completing some in-depth research, we have found many pros of doing so.

Pros of going with a MySQL Database Management System:

- Free, you can download MySQL community edition, or the open source MariaDB, it is basically just an open-source version of MySQL

- Easy to install, in an operating system that has a package manager, with just a single install of the command line, your mysql will be ready. In Windows, the installer is straight forward.

- Not so much heavy, meaning it will not cost your computer memory or CPU usage, MySQL do not need that much of your computer resources

- Big community, so many technical, non-technical documentation online

- It keeps growing. MySQL is not going to be obsolete for a very long time, in fact its users keep growing and some additional features being added, performance and security keeps getting better.

- So many tools / database management tools for creating, modeling. SQL development, importing, exporting, etc.

MySQL will work well for our tagging system, since we will have to use a relational database, rather than a non-relational one. [37] We need a relational database, so we can join tables together to retrieve the data needed from a user's complex queries, which could include a variety of specific tags, as well as logical operators. Users will be searching for specific tags and categories within papers, as well as saving their own queries. MySQL is fast, lightweight, reliable, secure and has continuous support by their developer Oracle.

5.3.2 Database Diagram

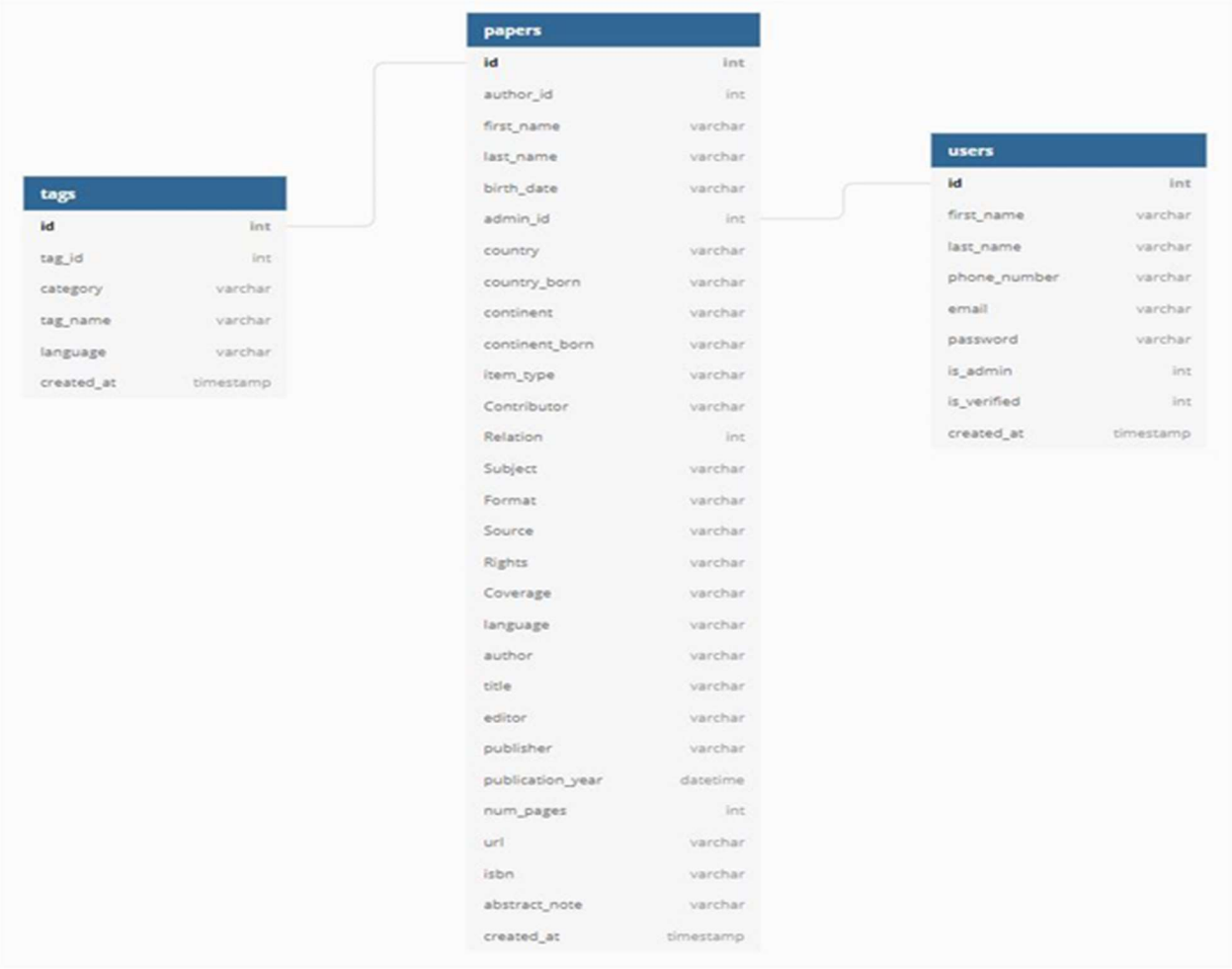


Figure 101. Entangled Philosophies Database Diagram

The Entangled Philosophies database will use 3 primary tables to perform its data storage/retrieval (tags, papers, and users) tables [38]. The papers table will store all the metadata from the paper after it is uploaded onto the Entangled Philosophies website. The metadata stored in the papers table will include the necessary information to comply with the Dublin Core metadata standard Contributor (varchar), Relation (int), Subject (varchar), Format (varchar), Source (varchar), Rights (varchar), Coverage (varchar). We chose the Dublin Core metadata standard since it is the most common and easy to understand metadata standard today.

The papers table will also include other information such as *created_at (timestamp)* which will be the time the paper was uploaded onto the Entangled Philosophies website. This is useful to look up when a user uploaded a specific paper. *num_pages (int)* will be used to let users be able to see how many pages the paper uploaded is. The other variables in the papers table include your typical information that you would expect to see from a paper like (Title of the paper, Authors first name, Author's last name, Author's date of birth, publication year, country the paper was written, country the Author is from, format of the paper, etc.). The paper's table is also connected to the tags table, by the paper's *id (int).* This connection will be made from the tags that are associated with each paper. The paper's table also has a connection with the user's table by the *admin_id (int).* The *admin_id* will let user's know which user uploaded the paper onto the Entangled Philosophies database.

The tags table will hold the tags from papers uploaded onto the Entangled Philosophies website. Once the tags from the Paper Upload Page are input and the paper is uploaded those tag's will be associated with that paper. [39] *id (int)* is a unique id which will connect the paper to the tags within that paper. *tag_id (int)* will be an id that is for each specific tag. For example, the tag Asia could have a *tag_id* of 192. Other countries like Africa, might have a *tag_id* of 193. [40] *category (varchar)* will be overlooking multiple tags such as Asia or Africa, which would be stored under the *category* continents. *language (varchar)* will let us know which language the tag is in. Since Entangled Philosophies will support multiple languages, this variable will help us do just that, by letting the tags be in multiple languages. *created_at (timestamp)* will be a timestamp of when the tag was added to the paper. Users will have the option to upload tags with papers, as well as add tags to papers later, so it is important to have a timestamp of when tags were added to each paper.

The users table will store information about the user after they register their account with Entangled Philosophies. Information like (first name, last name, email, and password) will be stored after completing the account registration. Once you complete and you verify your account through the email given *is_verified (int)* will store a 1 and the account registered will have a timestamp *created_at (timestamp)* which will let us know when the user created their account. If the user is given admin privileges, then they will have a 1 for *is_admin (int)* and a 0 otherwise. Lastly *id (int)* will be a unique integer which will specify each user's account, so that we can track which users have uploaded which tags/papers.

5.4 Tagging

5.4.1 Tags Summary

Tags can be described as specific keywords that the administrators of Entangled Philosophies can use to describe their uploaded papers. Example tags may be related to the time period such as "Modern Era" or "Classical Era" or certain topics of discussion like "Idealism" and "Hermeneutics." Tags exist in the database of Entangled Philosophies as tools to help categorize uploaded papers and also help facilitate an efficient querying process that allows for detailed research.

Entangled Philosophies is a multilingual service and as expected, each tag that exists in the database should have a translation for all supported languages. While there exists a translation for each tag, it's important to emphasize that a different language does not constitute a different tag. To clarify, if the tag "Modern Era" is in the database and the supported languages are English and German, then the tags "Modern Era" and "Moderne Ära" both point to the same subset of papers uploaded to the database. The multilingual aspect of the site exists to enhance the user experience and to make it more accessible to anyone interested in the study of philosophy.

5.4.2 User Created Tags

One of the most important aspects of making Entangled Philosophies a service for users to expedite their research process is the ability for any user to create their own custom tags and tag papers with them. While tags created by the administrators are public to everyone using the service, user created tags are entirely private and are only viewable/searchable by the created user.

We need to design our database in a way that allows for some tags to be considered “public” and for other tags to be considered “private” and only allow users’ queries to involve the public tags and their own private tags. Implementing this in a database is reasonably simple since all we need to do is just add an extra column that denotes the “owner” of the tag by using their user id. If we denote the public tags to have an owner of 0, then that will essentially be a catch all id value for a tag created by an admin (their individual user ids will not be included, it will just default to 0 for tag creation). But if a user creates a tag, then the owner field will be populated with their user id, uniquely tying a tag to a user.

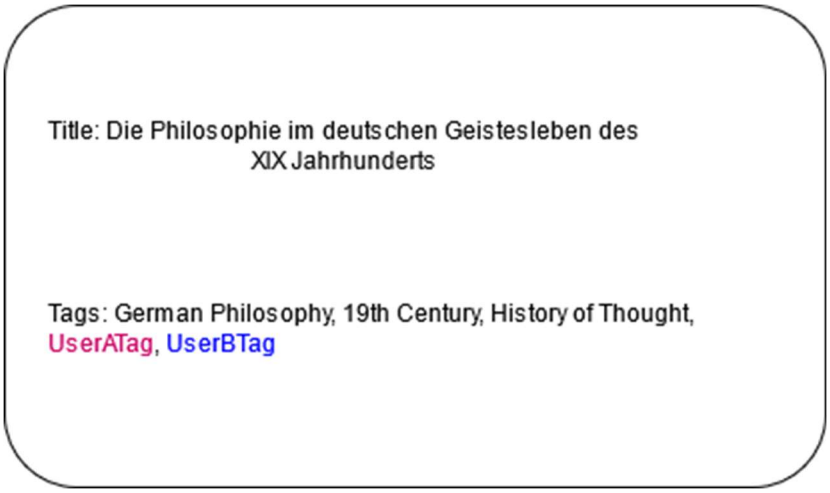


Figure 102. Example of public and private tags coexisting on paper

Now, when an SQL query is made, to ensure that we aren’t looking at an invalid table entry for a tagged paper, we can confirm that the owner of the tag we’re looking at is equivalent to the user id of whoever requested the query (the user id will be directly injected into the SQL query).

5.4.3 Tag Hierarchies

While the standard system of tagging papers is good for some reasonably simple queries, it becomes a bit more complex to handle detailed queries. For example, when adding tags to papers there may be multiple philosophical ideas involved and thus, each idea gets their own tag. But if we wanted to make a query that singles out only specific ideas being discussed (and excluding any other ideas), we would need to manually exclude each tag that we definitely do not want to see.

The tediousness of trying to handle a query like this provides motivation for us to be able to organize papers in a more direct way that can then be queried. To do that, we're considering the hierarchy system which is almost equivalent to a custom file directory that users/administrators can create at will.

When a user tags a paper with a hierarchy of tags, such as "Idealism/Modern Era," then whenever that hierarchy or any sub-hierarchy is queried, that paper will appear as a result of the query. Before, queries were only able to exclude certain tags but with this hierarchical tag structure existing in parallel with the singular tag system, queries will be able to exclude entire subclasses of papers by just excluding the parent hierarchy.

Attempting to implement both of these types of tags in parallel leads us to a few different approaches, mainly concerned with whether we need extra database tables or if both types of tags can co-exist in the same table. It turns out that there is a simple remodel we can do to the entire tags system that will allow us to easily integrate hierarchical tags: make each singular tag a hierarchy of just one tag which is essentially converting them into a single depth hierarchy. With this approach, we can implement the hierarchical tagging system in parallel with the singular tagging system by just converting everything into a hierarchy where some hierarchies are intended to be a single level and others are intended to be multi-level and expandable.

The easiest way to implement this in a database is to make use of the Nested-Set Model. What this model allows us to do is to convert each hierarchy **X** into 2 integer values that represent a range of hierarchies that **X** is a parent of. To clarify, consider this set of hierarchies:

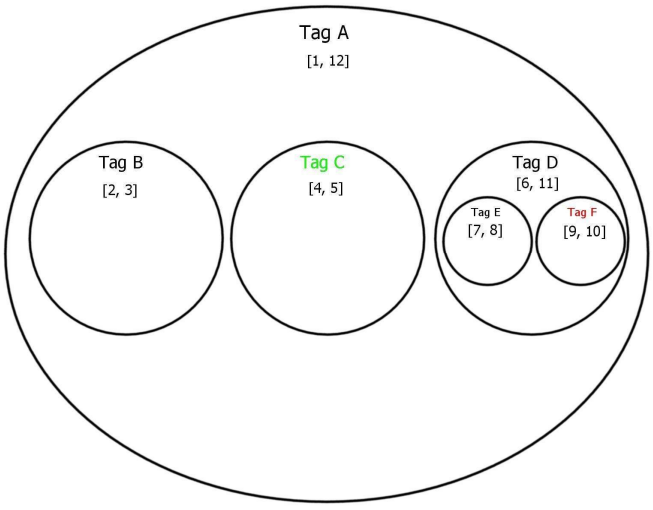


Figure 103. Nested Set Model with range values

The hierarchies represented by this picture are as follows: “Tag A”, “Tag A/Tag B”, “Tag A/Tag C”, “Tag A/Tag D”, “Tag A/Tag D/Tag E”, and “Tag A/Tag D/Tag F”. The main benefit of converting each node of the hierarchy into 2 integer values is that any node’s values are contained inside the range values of all of its parents. For example, if we analyze the values of Tag E, who is nested inside of Tag D and Tag A, we can see that it’s range values are [5, 5] which is contained inside of the range [4, 6] and inside of the range [1, 6] implying that those are Tag E’s parents. But when comparing Tag E’s values to Tag F (who is not a parent or child of Tag E), we can see that there is no overlap between their ranges [5,5] and [6,6]. This type of relationship will hold for all hierarchies represented: if a node is a parent of another, the child’s range values will be contained inside of the parent and vice versa.

While this may look reasonably complex on paper, in a database this has a simple representation: just 2 integers called “LeftValue” and “RightValue” to denote the range. In a query, when we’re attempting to include an entire hierarchy (and subsequently all its child hierarchies), we just need to validate that the current hierarchy tag we’re interested in including has its range values in between the parents range values. This can be done with a simple SQL statement:

```
1. SELECT relation.paper_id
2. FROM hierarchies AS node, hierarchies AS parent, tag_paper AS relation
3. WHERE relation.tag_id = node.id and node.LeftValue BETWEEN parent.LeftValue AND parent.RightValue
```

Figure 104. SQL to pick a sub-hierarchy

This statement guarantees that we are picking a paper-tag relationship from our relational table that involves a hierarchy tag that is a child of the parent hierarchy (whose values would be injected into the SQL query based off of user input) and if validated, returns the paper_id of that paper-tag relationship.

Adding children nodes to this hierarchy and maintaining this numeric range relationship also has an easy implementation: we can add the number 2 to each node that has LeftValue and RightValue greater than the node we’re adding to. To show this visually, let’s add a new node into the “Tag A/Tag C” hierarchy we saw earlier and see how it affects the range values for each node:

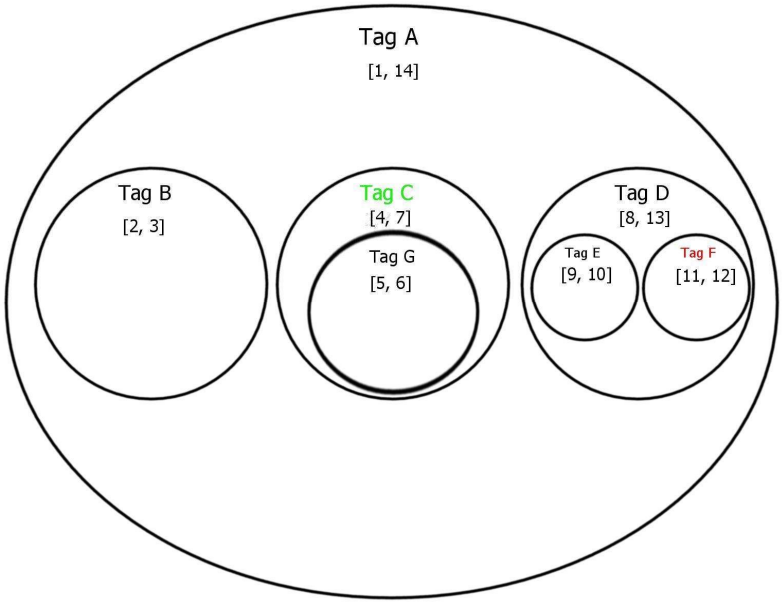


Figure 105. Updated Nested Set after adding new sub-hierarchy

Adding this new hierarchy, “Tag A/Tag C/Tag G” changes the LeftValue and RightValue in a very predictable way: it adds 2 to every pair of values located to the right of this new node (in the visual representation). Then the new node “Tag G” gets values equivalent to [parentLeftValue + 1, parentLeftValue + 2] and is inserted into the database. In SQL, this process could look like:

```
1. LOCK TABLE hierarchies WRITE;
2. SELECT myRight := rightValue FROM hierarchies
3. WHERE id = parent_id
4.
5. UPDATE hierarchies SET rightValue = rightValue + 2 WHERE rightValue > myRight;
6. UPDATE hierarchies SET leftValue = leftValue + 2 WHERE leftValue > myRight;
7.
8. UNLOCK hierarchies;
```

Figure 106. SQL statement to add new sub-hierarchy

To update the table, finding all nodes to the “right” of the one we’re trying to add is as simple as just checking who has left and right values ahead of our parent’s values and updating all of them accordingly. We would then insert the new node into the database with the proper left and right values.

We can see that this type of model makes it easy to include both singular and hierarchical tags. If we treat each singular tag as a hierarchy but disallow any sort of new children to be added to it, then we can query single tags in the exact same way we query hierarchies.

5.4.5 Failure of Tag Hierarchy

The idea of maintaining a hierarchy of tags was something we were planning to implement the entire time we were designing Entangled Philosophies. However, there were 2 main issues with this system that wasn't apparent to us until we began implementing the site:

1) The use cases of this system were too specific to warrant the extra complexity that would come from implementing it.
2) It was too complex to implement within the first 2 weeks of the Summer semester.

When we were introduced to the problem posed by the University of Hildesheim, we learned that their current model of organizing papers was similar to a file directory of papers. Hearing this inspired us to create a tagging system that incorporated their old ways of categorizing papers and merge it with a newer more flexible system. That way, they'd be able to have a system that was somewhat familiar to them but then get introduced to a method that made their tagging more efficient for searches.

However, we began to really analyze what sort of use cases we'd get out of implementing a tag hierarchy and began to notice a few flaws with how rigid the format would've been. Take, for example, a hierarchy that looked like: **China/19th Century/**. We could tag a series of papers with this hierarchy but in reality, we could've just replaced this hierarchy with a query like: **China** AND **19th Century** and we would've simulated this hierarchy exactly.

The other issue that was made apparent is that the order of the hierarchy is very important. We just discussed the **China/19th Century/** tag but based on how we were planning to implement this hierarchy, the tag **19th Century/China** would have to have been considered a different hierarchy even though they're made of the exact same tags. This was one of the core issues with this system's usability: the order of tags had to matter to properly simulate a directory system but by forcing order to matter we would essentially make this hierarchy system extremely rigid and editing these tags in the future would have been very tedious.

To summarize, the tag hierarchy system only had a very specific use case and in most scenarios, we could replace the hierarchy system with a well constructed boolean logic

query without the need of a hierarchy. Discovering these initial issues definitely made it seem like abandoning the tag hierarchy system was the way to go.

The second problem with this hierarchy system is how critical it was to finishing the rest of the site and that attempting to implement something this complex (and finish it within the first 2 weeks to keep pace with our milestones) was too much of a risk to take for the Summer semester of Senior Design. Because we were already on such a quick pace, risking setbacks this early on in the implementation process could have offset the entire schedule of the project and resulted in an incomplete prototype. It may have been different if this complex (and potentially undoable) feature we were adding was simply an add-on to the site and not necessary for it to function. For example, the custom search query system that we did end up implementing was never necessary to show the core functionality of the site: if we failed to implement custom queries, we would still have the normal filter menu that allowed for reasonably complex queries. But if we failed to implement the tags themselves, that would mess up the flow of the project for the entire semester.

The added complexity of implementing this system coupled with the fact that we couldn't really justify its usefulness prompted us to drop the idea of tag hierarchies and instead replace it with a new idea that we called "tag filtering". This idea of tag filtering was more flexible than tag hierarchies and provided a similar type of categorization to a file directory (except without permanently tagging a paper with that hierarchy).

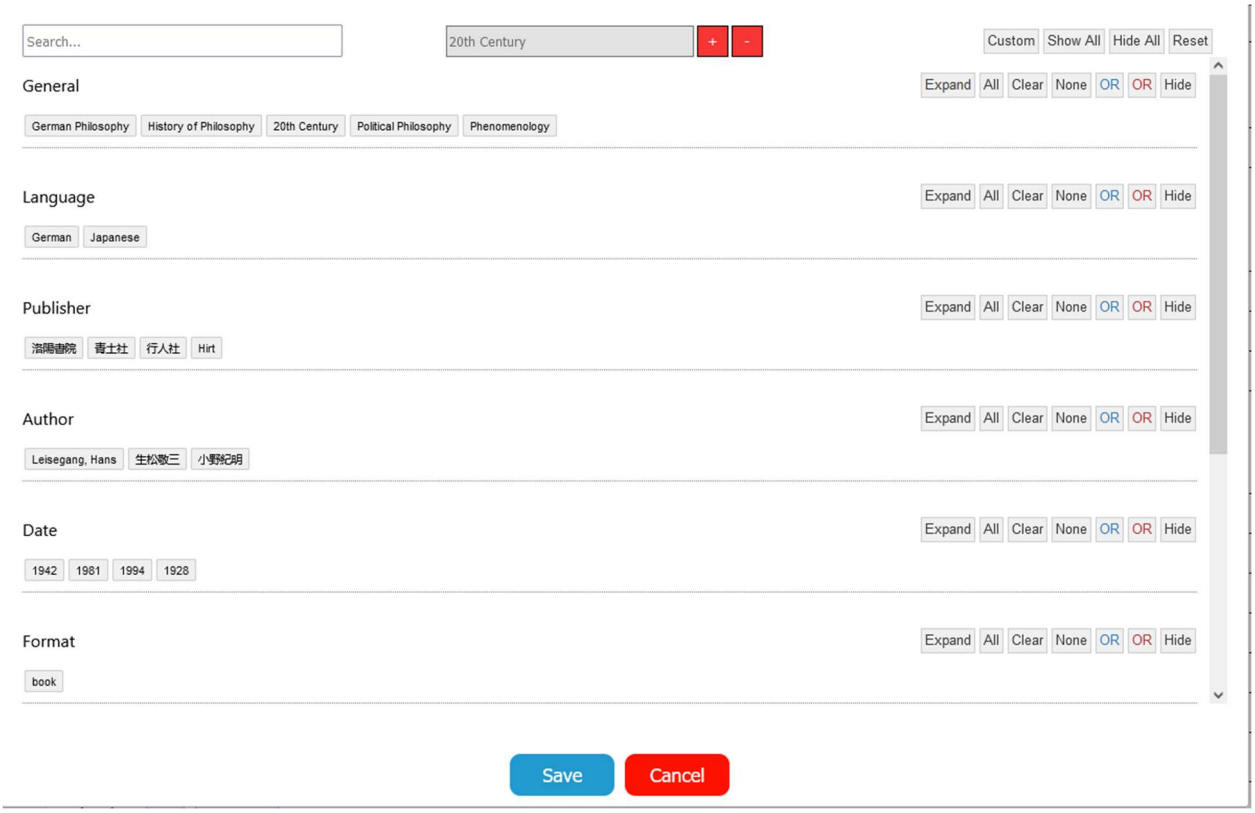
It's important in situations like this to reflect on how we could avoid something like this in the future. The biggest mistake we made that didn't allow us to catch the issue of this system was not enough group discussions about it. Even though it was one of the core features of the site (and one of the first to be implemented), we didn't have any meaningful critical design discussions until the end of the semester. It felt like we were more aiming to split up the research into isolated components and then merge them together when it came time to implement the site, but this lack of critical discussion about the practicality of tag hierarchies led to us needing to rework the tagging system's design at the very beginning of the project.

When we began to discuss the replacement of the tag hierarchies, tag filtering, we spent a significant amount of time discussing the idea among ourselves and with our sponsor and this time, came to a consensus on its usefulness and ease of implementation. This led to a more cohesive and less stressful design and should have been what we initially discussed early on in the design phase.

5.4.5 Tag Filtering

Tag filtering was the idea that we ended up using to replace the initial tag hierarchies idea. The motivating use case behind tag filtering is as follows: say you're a researcher who is interested in studying the 15th century. When entering the filter menu to perform your searches, you may be presented with a series of tags that are basically not at all related to the 15th century. In fact, depending on the contents of the database at the time, it's possible that there are very few out of hundreds of displayed tags that are related to papers that discuss the 15th century.

This issue of a user knowing a specific tag they're interested in searching for but not knowing any other related tags (and potentially being shown only irrelevant tags) motivated us to come up with a solution that shows a user more relevant tags. This is the general layout of tag filtering:



A user can click the red plus button at the top of the filter screen and then click on a tag to filter by. This will change the tags displayed to them. After a user has filtered by a tag (say 20th century), the set of tags that is displayed to them are only those that appear as a tag on a paper with 20th century as well. So, the only reason "History of Philosophy" is

visible to us on the screen is because there exists a paper in our database that has **both** the tags “20th century” and “History of Philosophy.”

The uses of a feature like this allow a user who has a general area of research they’d like to focus on to get a more indepth look at what tags are related to their interests. For example, because the tag “Idealism” was originally in the filter menu but disappeared after applying the filter, the user can conclude that they won’t find any papers relating Idealism and the 20th century and so they shouldn’t really need to see it. A user can add as many tags to the filter as they would like in order to get a proper understanding of what tags they should be looking out for.

The benefit of implementing this feature over tag hierarchies is that tag filtering is a way to simulate a tag hierarchy without the burden of restricting the order of the hierarchy. Bringing back our classic example of **China/19th Century** as a hierarchy, we can see that performing a normal query of **China** AND **19th Century** can get us the set of papers that would’ve corresponded to that hierarchy, and using tag filtering on the those 2 tags would get us the representation of that file directory (i.e. what tags are technically able to extend this current hierarchy we have). Overall, the tag filtering idea was a better planned and more useful feature than our initial tag hierarchy idea and is powerful enough to give a curious user a comprehensive glance at the database without overwhelming them with complexity.

5.4.6 Tagging System Conclusion

The tagging system is one of the core features of Entangled Philosophies and its design needs to be both simple as well as powerful when it comes to the complexity of queries it can handle. To allow users and administrators to have a large amount of freedom when it comes to creating complex tags, we added a simple single entity tagging system and gave them the ability to filter those tags to view relevance to other tags and allowed them to create custom boolean logic equations that they could use to fully query the database in any way they wished.

6 Milestones

- February 5: Initial team meeting and Bootcamp formation canvas (completed)
- February 8: Initial meeting with sponsor (completed)
- February 15: TA check-in 1 (completed)
- February 18: Deliver initial research on scope of project and framework decisions (completed)
- February 21: Make a Github and set up a Trello Board (completed)
- February 28: Research about software implementation and frameworks, research database schemas, add dummy data to database (completed)
- March 8: Attend Metadata introductory presentation (completed)
- March 10: Status presentation (completed)
- March 15: Tagging system and Framework research finished (completed)
- March 20: Begin experimenting with and understanding development environment (completed)
- April 1: TA check-in 2 (completed)
- April 5: Finalized research on language swapping and data visualization (completed)
- April 22: Finish research and begin compiling full design document (completed)
- April 28: Final design document complete and submitted (completed)
- May 17: General layout (hamburger menu) and page routing implemented on front end
- May 17: Account creation/editing on front end implemented
- May 17: API endpoints for creating accounts, resetting password, email

verification, and administrator additions implemented

- May 24: Add/Edit tags implemented
- May 24: Administrator management implemented
- May 31: Add/Edit papers implemented
- June 7: Search page (not filter) implemented
- June 14: Filter page implemented.
- June 21: Advanced Filter implemented
- June 21: Save/Load queries implemented
- June 28: Word Cloud Data Visualization implemented
- July 5: Bar Chart Data Visualization implemented
- July 5: Language Swapping implemented

7 Project Summary and Conclusion

Entangled Philosophies is an ambitious effort to help assist philosophy experts and researchers in their endeavor to categorize and analyze the trends they come across in their studies. Looking back at our progress in the last few months, we've worked hard to help design a system that allows them to store relevant information about the papers they've been tracking and add informational tags to each paper to allow them to research any trend they may want to.

We believe that Entangled Philosophies can be a powerful tool that will allow researchers to expedite their studies and share/merge their results with other interested parties. Upon completion, not only will Entangled Philosophies help the general research process, it will also make the field of philosophy significantly more accessible to aspiring students that wish to find an introduction to the field. While the scope of who may access our service isn't fully defined yet, we designed it with the general user in mind and made it a simple and accessible system for all to use.

Over the course of this semester, we've met constantly to discuss our research and findings into what may make a great search and tagging system. We often collaborated effectively to express our ideas with our sponsors and brainstorm new ideas based off of a constant stream of feedback from them. Some of the topics of debate that have been discussed have been what frameworks would be best for us (React, Bootstrap, etc.), what information should be tracked about each paper and how that should be stored in a database, how in depth and complex the tagging system should be, and many more.

Now that we have finished implementing Entangled Philosophies, we can say that most of our designs were able to be successfully implemented and any features that fell apart were quickly refactored to fit and improve the site. After implementing everything required for the project, we can safely say that Entangled Philosophies was not only an enriching experience for the whole team but was also very beneficial to our sponsor in making headway on the University of Hildesheim's problem.

.

8 References

[1] https://da-14.com/blog/its-high-time-reactjs-ten-reasons-give-it-try

[2] https://stackoverflow.com/questions/42461279/how-to-deploy-a-react-app-on-apache-web-server

[3] https://medium.com/@davisonpro/an-advanced-guide-on-setting-up-a-react-and-php-web-app-acaedb21ab3a

[4] https://www.sparkpost.com/academy/cloud-infrastructure/introduction-to-email-apis/

[5] https://jwt.io/introduction/

[6] https://auth0.com/blog/why-should-use-accesstokens-to-secure-an-api/

[7] https://nodemailer.com/about/

[8] https://www.guru99.com/php-mail.html

[9] https://developer.mozilla.org/en-US/docs/Web/API/FormData/FormData

[10] https://attacomsian.com/blog/xhr-node-file-upload

[11] Bcrypt. (n.d.). Retrieved March 15, 2021, from https://www.npmjs.com/package/bcrypt

[12] Password_hash. (n.d.). Retrieved March 15, 2021, from https://www.php.net/manual/en/function.password-hash.php

[13] Safe password hashing. (n.d.). Retrieved March 16, 2021, from https://www.php.net/manual/en/faq.passwords.php

[14] Password_verify. (n.d.). Retrieved March 15, 2021, from https://www.php.net/manual/en/function.password-verify.php

[15] Angular translate. (n.d.). Retrieved March 13, 2021, from http://angular-translate.github.io/

[16] JavaScript cookies. (n.d.). Retrieved March 13, 2021, from https://www.w3schools.com/js/js_cookies.asp

[17] Kirsten Werner. (2019, November 26). Toggling between different languages in your react website. Retrieved March 13, 2021, from https://medium.com/kirsten-werner/toggling-between-different-languages-in-your-react-website-18e2497532e1

[18] File uploading in react.js. (2021, January 29). Retrieved March 13, 2021, from https://www.geeksforgeeks.org/file-uploading-in-react-js/

[19] React-papaparse. (n.d.). Retrieved March 13, 2021, from https://react-papaparse.js.org/

[20] Todilo, TodiloTodilo 1, ZippoZippo 12.9k88 gold badges5757 silver badges5555 bronze badges, Naisheel VerdhanNaisheel Verdhan 3, & Ffxsamffxsam 20.3k2727 gold badges7070 silver badges124124 bronze badges. (1964, November 01). Render array of inputs in react. Retrieved March 13, 2021, from https://stackoverflow.com/questions/34321128/render-array-of-inputs-in-react

[21] Singh, P. (2020, October 29). Display Multi-Dimensional Array Data in React. Retrieved March 13, 2021, from https://www.pluralsight.com/guides/display-multidimensional-array-data-in-react

[22] React-cookiebot. (n.d.). Retrieved March 13, 2021, from https://www.npmjs.com/package/react-cookiebot

[23] GDPR/ePR compliant cookie banners From cookiebot. (n.d.). Retrieved March 13, 2021, from https://www.cookiebot.com/en/cookie-banner/#:~:text=the%20questions%20above.-,What%20is%20a%20cookie%20banner%3F,before%20their%20data%20is%20processed

[24] SQL database design for tagging (2008, August 21), Retrieved February 18, 2021, from

https://stackoverflow.com/questions/20856/recommended-sql-database-design-for-tags-or-tagging

[25] Keller Philipp (2005, April 24), Tags: Database Schema, Retrieved February 25, 2021, from http://howto.philippkeller.com/2005/04/24/Tags-Database-schemas/

[26] Hillyer Mike, Managing Hierarchical Data in MySQL, Retrieved February 25, 2021, from http://mikehillyer.com/articles/managing-hierarchical-data-in-mysql/

[27] Riley Jenn, Seeing Standards: A Visualization of the Metadata Universe, Retrieved April 12, 2021, from http://jennriley.com/metadatamap

[28] DCMI Metadata Terms, (2020, January 1) Retrieved April 12, 2021, from https://www.dublincore.org/specifications/dublin-core/dcmi-terms/

[29] Hearst, M. (2020, July 06). WordZones (usable alternative to WORD CLOUDS). Retrieved March 16, 2021, from https://observablehq.com/@mahviz/wordzones-usable-alternative-to-word-clouds

[30] Daly, C. (2019, May 01). USA County heat map. Retrieved March 16, 2021, from https://observablehq.com/@chrisdaly/usa-county-heat-map

[31] D3. (2020, August 26). *Radial Tidy Tree*. The magic notebook for exploring data / Observable. https://observablehq.com/@d3/radial-tidy-tree

[32] d3. (2020, August 27). d3/d3-hierarchy. GitHub. https://github.com/d3/d3-hierarchy/blob/master/README.md#tree

[33] D3. (2020, August 27). *Mobile Patent Suits*. The magic notebook for exploring data / Observable. https://observablehq.com/@d3/mobile-patent-suits

[34] D3. (2021, February 19). *Bubble Chart*. The magic notebook for exploring data / Observable. https://observablehq.com/@d3/bubble-chart

[35] https://stackoverflow.com/questions/4795385/how-do-you-use-bcrypt-for-hashing-passwords-in-php

[36] https://www.php.net/password_hash

[37] Louis Davidson (2007): Ten Common Database Design Mistakes, https://www.red-gate.com/simple-talk/sql/database-administration/ten-common-database-design-mistakes/ Retrieved April 08, 2021

[38] Quora (2020): What is the best and easy way to store tags in MySQL database?, https://www.quora.com/What-is-the-best-and-easy-way-to-store-tags-in-MySQL-database Retrieved March 28, 2021

[39] Stackoverflow (2008) : Database Design for Tagging, https://stackoverflow.com/questions/48475/database-design-for-tagging Retrieved April 20, 2021

[40] Lekker Logic (2016): MySQL Many-To-Many Tags Schema – Database Design, http://lekkerlogic.com/2016/02/site-tags-using-mysql-many-to-many-tags-schema-database-design/ Retrieved April 18, 2021

[41] Created by Entangled Philosophies team (2021): Website used to create Database Diagrams dbdiagram.io, https://dbdiagram.io/d Retrieved March 27, 2021

[42] Created by Entangles Philosophies team (2021): Website used to create UI Mockups mockflow, https://mockflow.com/ Retrieved April 3, 2021

9 Appendix

9.1 Language Swapping Instructions

Below are comprehensive instructions for the multilingual feature of Entangled Philosophies that should assist anyone who needs to edit the translations of certain phrases or add support for a new language.

Overview of Language Swapping

In the main GitHub repo, the language swapping file can be found in *entangled/src/dictionary.js*. There are 2 parts to the language swapping here:

1) A function containing a switch statement that assigns each phrase in the website to an integer value. The source code will call this function with an integer value and a language (“eng” or “ger”) and receive the properly translated version of this phrase.

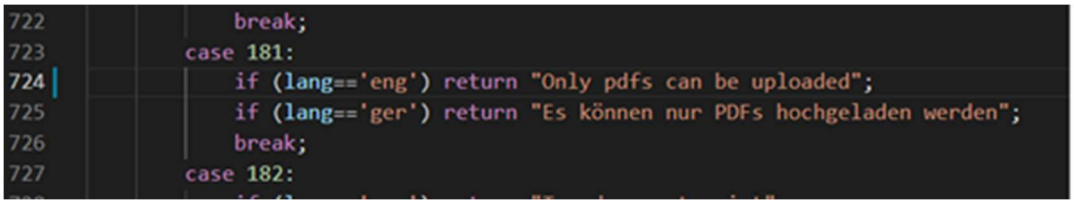
2) An array of phrases for each of the supported languages (you’ll see this near the bottom of the source code). **englishWords**[0] and **germanWords**[0] correspond to the same phrase (integer value 0) but in their respective languages. This is used when the requested phrase may not be known beforehand (i.e. a message from an API endpoint).

These are the 2 ways phrases can be translated in Entangled Philosophies

Editing Existing Terms

To edit an existing term, you simply need to find it’s integer value in the switch statement (a Ctrl-F search should be good) and edit the value in *dictionary.js*. For consistency, you’d also need to find it in the array of phrases at the bottom of *dictionary.js* and edit it there as well.

So, if there’s a typo and you want to change “pdfs” to “PDFs”, simply find its case number (181) and edit the phrase directly.



Now it becomes

```
722             break;
723         case 181:
724             if (lang=='eng') return "Only PDFs can be uploaded";
725             if (lang=='ger') return "Es können nur PDFs hochgeladen werden";
726             break;
727         case 182:
```

You would also find it in the array **englishWords** as well and edit it there:

```
gung, bevor Sie sich anmelden.', 'ist bereits ein Adminis
aper', 'Only PDFs can be uploaded', 'Tag does not exist',
```

Adding New Terms

Adding new terms is like editing terms. Just go to the bottom of the switch statement and set up a case for the next number which contains the new phrase:

```
821             if (lang == 'ger') return "Regulären";
822         case 206:
823             if (lang == 'eng') return "Success";
824             if (lang == 'ger') return "Erfolg";
825         //NEW PHRASES GO HERE
826         case 207:
827             if(lang == 'eng') return "New Phrase";
828             if(lang == 'ger') return "New Phrase in German";
829         default:
830             return "?";
831     }
```

You would also add this new phrase to the **end** of the arrays **englishWords** and **germanWords**:

```
nicht', 'Einzelne Datei hochladen',
Query', 'Success', 'New Phrase']
```

Conveniently Adding New Terms/Languages

The above methods are of course very tedious when there are lots of edits/new terms to add. To automate this process somewhat, we wrote a python script that reads directly from the Entangled Philosophy Translations spreadsheet and creates this switch statement for us.

Assuming you are on a machine that can run python, there is a script on the main repo at *entangled/src/language_swapping.py* that will print out all the case statements for us (so no one has to manually type out 200 case statements).

Simply put the Entangled Philosophy Translations Spreadsheet in the same directory as the script and run the script and it will automatically print out all the necessary case statements. It's important to note that all fields in a language's column should be filled in because the script does not do anything to especially handle empty fields.

This script makes it easy to instead just edit the Google Drive Spreadsheet that we've been maintaining and when there are large amounts of updates, simply rerun this script and paste the output in the switch statement in *dictionary.js*. Example:

language_swapping.py

```python
import pandas as pd

ex_data = pd.read_excel('Entangled_Philosophies_Translations.xlsx')

germanVals = ex_data['German'].values.tolist()
englishVals = ex_data['English'].values.tolist()

# Use the lines below to print out the arrays of terms to store at the bottom of dictionary.js
# print(germanVals)
# print(englishVals)

caseNum = 1
for i in range(len(germanVals)):
    print('case ' + str(caseNum) + ':')
    print("\tif (lang=='eng') return \"" + englishVals[i] + "\";")
    print("\tif (lang=='ger') return \"" + germanVals[i] + "\";")
    print("\tbreak;")
    caseNum += 1

```

```
PROBLEMS  OUTPUT  TERMINAL
TERMINAL
ahmad@DESKTOP-19S5LLC:/mnt/c/Users/ahmad/Documents/Programming/c++$ python3 language_swapping.py > dictionary.out
ahmad@DESKTOP-19S5LLC:/mnt/c/Users/ahmad/Documents/Programming/c++$
```

After running this script, the file *dictionary.out* now contains the entire switch statement for an easy copy-and-paste. The 2 commented lines make it easy to copy and paste the new **englishWords** and **germanWords** array as well. When implementing, I found it much easier to just make the edits in the spreadsheet and paste the output into *dictionary.js* so that there were less chances for human error mistakes.

Adding New Supported Languages

Unfortunately, we couldn't entirely automate adding a new language, but the code changes to add support for a new language (assuming all the translations are there) are somewhat minimal.

STEP 1:

The first 2 areas to edit are (assuming you are at the root of the GitHub repo):

1) *entangled/src/api.js*

2) *api/languages.php*

For *api.js*, you should see an array at the top named **supported-_languages**, that has 2 terms currently ("eng" and "ger"). Simply, add the new language (say we're adding Russian so "rus") to the end of the array:

```
2   export const cookies = new Cookies();
3   export const supported_languages = ["eng", "ger", "rus"];
4   export const urlBase = 'http://chdr.cs.ucf.edu/~entangledP
```

Do the same for *api/language.php*:

```
api > languages.php
1   <?php
2       const supported_languages = ["eng", "ger", "rus"];
3   ?>
```

This helps automate a lot of the legwork for adding new tags/categories which requires all translations of the supported languages.

STEP 2:

Navigate to *entangled/src/components/Tags.js* to add the new input fields for the new language in the Tags page. There are 2 functions of interest here that support adding/editing tags and categories:

1) *AdminEdit*

2) *AdminAdd*

We'll focus on the changes to *AdminAdd* but the changes to *AdminEdit* are essentially the same. The lines to focus on are:

```
262         <h4 class="rightBoxText" id="rightBoxTextGermanTag">{dSettings(31, userLanguage)}</h4>
263         <input type="text" className="taginputBoxes" id="gerBox" placeholder="German Tag" />
```

Line 262 is the left-adjusted header for an input field that denotes what language we're translating to. The line below is the input field for the translation. Make sure that the class/className for these HTML tags stay the same (this helps with CSS formatting). The fields to change for a new language are the *id* and the *placeholder* values.

The *id* for these fields follows the format: [language]+'Box'. So, we have 'gerBox' for the German translation and we'd have 'rusBox' for the Russian translation. Adding the fields for Russian would look like:

```
262         <h4 class="rightBoxText" id="rightBoxTextGermanTag">{dSettings(31, userLanguage)}</h4>
263         <input type="text" className="taginputBoxes" id="gerBox" placeholder="German Tag" />
264
265         <h4 class="rightBoxText" id="rightBoxTextRussianTag">{/* Case number for Russian Translation */}</h4>
266         <input type="text" className="taginputBoxes" id="rusBox" placeholder="Russian Tag" />
267
```

* The CSS should not need to be edited for this box to format itself, but in case of issues refer to *Tags.css* in the same directory.

For full language support, you'd need to add these 2 new lines to 4 locations in the code for *Tags.js*:

1) *AdminAdd* in the if statement for toggleState:

```
function AdminAdd({ toggleState }) {
    if (toggleState) {
        //we're adding a new CATEGORY, not a new TAG in this if statement
        return <div>
            <h1 id="editTagHeader2">{dSettings(18, userLanguage)}</h1>
            <h4 class="rightBoxText2" id="rightBoxTextEnglish">{dSettings(130, userLanguage)}</h4>
            <input type="text" className="taginputBoxesToggled" id="engBox2" placeholder="English Category" />
            <h4 class="rightBoxText2" id="rightBoxTextGerman">{dSettings(131, userLanguage)}</h4>
            <input type="text" className="taginputBoxesToggled" id="gerBox2" placeholder="German Category" />

            {/* New Translation boxes for CATEGORIES goes here. Note that for CATEGORIES, the ids are [language]+'Box2' */}
            <h4 class="rightBoxText2" id="rightBoxTextRussian">{/* Case number for Russian Translation */}</h4>
            <input type="text" className="taginputBoxesToggled" id="rusBox2" placeholder="Russian Category" />

            <button id="addTagButtonsAdmin" onClick={() => doAddCat(-1)}>{dSettings(27, userLanguage)}</button>
        </div>
    }
```

2) *AdminAdd* after the if statement for toggleState:

```
    //Adding a new TAG here
    return <div>
        <h1 id="editTagHeader">{dSettings(18, userLanguage)}</h1>
        <h4 class="rightBoxText" id="rightBoxTextCategory">{dSettings(29, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="tagCategoryBox" placeholder="Tag Category" />
        <h4 class="rightBoxText" id="rightBoxTextEnglishTag">{dSettings(30, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="engBox" placeholder="English Tag" />
        <h4 class="rightBoxText" id="rightBoxTextGermanTag">{dSettings(31, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="gerBox" placeholder="German Tag" />

        {/* New Translation boxes for TAGS goes here. Note that for TAGS, the ids are [language]+'Box' */}
        <h4 class="rightBoxText" id="rightBoxTextRussianTag">{/* Case number for Russian Translation */}</h4>
        <input type="text" className="taginputBoxes" id="rusBox" placeholder="Russian Tag" />

        <button id="addTagButtonsAdmin" onClick={() => doAddTag(-1)}>{dSettings(27, userLanguage)}</button>
    </div>
}
```

You essentially just need to copy and paste the 2 HTML tags for German and replace the German specific parts with the new language being used. The last 2 code changes are in the *AdminEdit* function (located above *AdminAdd*). The changes are the same except for the placeholder value which contains then old translations (since we're editing tags now):

```
    var translations = getTagTranslation(rowInfo.tag_id);

    var category = rowInfo.catText;
    var tag_eng = translations.eng;
    var tag_ger = translations.ger;

    return <div>
        <h1 id="editTagHeader">{dSettings(26, userLanguage)}</h1>
        <h4 class="rightBoxText" id="rightBoxTextCategory">{dSettings(29, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="tagCategoryBox" placeholder={category} />
        <h4 class="rightBoxText" id="rightBoxTextCategory">{dSettings(30, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="engBox" placeholder={tag_eng} />
        <h4 class="rightBoxText" id="rightBoxTextCategory">{dSettings(31, userLanguage)}</h4>
        <input type="text" className="taginputBoxes" id="gerBox" placeholder={tag_ger} />

        <h4 class="rightBoxText" id="rightBoxTextCategory">{/* Russian translation */}</h4>
        <input type="text" className="taginputBoxes" id="rusBox" placeholder={tag_rus} />

        <button class="editTagButtonsAdmin" onClick={() => doAddTag(rowInfo.tag_id)}>{dSettings(27, userLanguage)}</button>
        <button class="editTagButtonsAdmin" id="tagtDeleteButton" onClick={() => doRemoveTag(rowInfo)} >{dSettings(28, userLanguage)}</button>
    </div>
}
```

Here, we can see that for placeholder values, we pull that value from an object named *translations* that we get from an API call. We would simply define the variable “tag_rus” as *translations.rus* and use that for the placeholder as well.

Step 2 is the most involved with the code for adding a new language. It could be automated, but we ran out of time before we could get to automating it.

STEP 3:

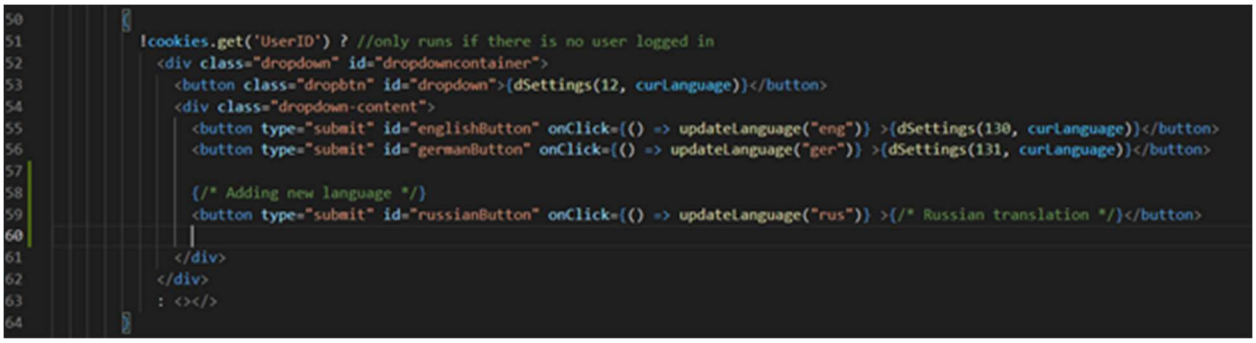
There are 3 dropdown boxes in the source code that use the supported languages:

1) *entangled/src/App.js*

2) *entangled/src/components/Register.js*

3) *entangled/src/components/Settings.js*

which allows a user to dropdown select their preferred language. To update these dropdowns, all you need to do is copy-and-paste the button styling used for the previous language and modify it to be accurate for the new language. Example for App.js dropdown:



You can find the dropdown menus on those pages by searching with Ctrl-F for the term “dropdown”.

STEP 4:

Find the file *entangled/src/components/UploadPaper.js* and in that file, the function *AddSpreadsheetPaper*. In that function, there’s a section that supports adding new tags from a spreadsheet upload that aren’t in the database:

```
94      for (const idx in tagsList) {
95          if (data.tags[tagsList[idx]] == "-1") {
96
97              //Adding a new tag from the spreadsheet that isn't in the database
98              var category = "General";
99              var eng = tagsList[idx];
100             var ger = tagsList[idx] + " - needs German translation";
101
102             //new language support:
103             var rus = tagsList[idx] + " - needs Russian translation";
104
105             tagsToAdd.push({ category: category, eng: eng, ger: ger, rus: rus });
106
107         }
108     }
```

You can find it by Ctrl-F searching for the phrase “General”. All you need to do here is add a variable exactly like “ger” but for the new language instead. This allows us to add the tag without necessarily having the translation at the time of making it.

STEP 5:

The last part is updating the script that generates the switch statement for us: *entangled/src/language_swapping.py*. All you need to do is read the column corresponding to the new language and add a print statement to the for loop that prints the cases:

language_swapping.py

```
1   import pandas as pd
2
3   ex_data = pd.read_excel('Entangled_Philosophies_Translations.xlsx')
4
5   germanVals = ex_data['German'].values.tolist()
6   englishVals = ex_data['English'].values.tolist()
7
8   # adding new languages
9   russianVals = ex_data['Russian'].values.tolist() # the name of the column that has the translations
10
11  # Use the lines below to print out the arrays of terms to store at the bottom of dictionary.js
12  # print(germanVals)
13  # print(englishVals)
14  # print(russianVals)
15
16  caseNum = 1
17  for i in range(len(germanVals)):
18      print('case ' + str(caseNum) + ':')
19      print("\tif (lang=='eng') return \"" + englishVals[i] + "\";")
20      print("\tif (lang=='ger') return \"" + germanVals[i] + "\";")
21
22      #adding new languages
23      print("\tif (lang=='rus') return \"" + russianVals[i] + "\";")
24
25      print("\tbreak;")
26      caseNum += 1
```

9.2 Video Replacement Instructions

Below are instructions on how to edit/add the existing tutorial videos created by the Entangled Philosophy team in order to better explain to future users the full capabilities of the site.

Replacing Help Videos

If you want to replace which help video is displayed for a page, navigate to the file `api.js` in the src folder of the project. Near the top of the file, you should see an exported array named `HelpVideoURLS` which is an array of YouTube embed links. There is a comment to the right of each link, explaining which video it corresponds to.

To replace these videos, all you need to do is paste in your own embed links into to the corresponding array slot and re-run the project build. Here's a screenshot of `HelpVideoURLS` in the `api.js` file:

```
4   export const urlBase = 'http://chdr.cs.ucf.edu/~entangledPhilosophy/Entangled-Philosophie
5   export const fileURLBase = 'http://chdr.cs.ucf.edu/~entangledPhilosophy/paper/';
6
7   export const HelpVideoURLS = [
8       "https://www.youtube.com/embed/fSl_mB3FY44", //Edit Paper video
9       "https://www.youtube.com/embed/OQ2sDejb-bg", //Search Page video
10      "https://www.youtube.com/embed/pTTtCoFtv_I", //Search Filter video
11      "https://www.youtube.com/embed/8xNW0mOtqVE", //Search Advanced Filter video
12      "https://www.youtube.com/embed/ZGqpHkFNVQc", //Tags Page video
13      "https://www.youtube.com/embed/XsOf1Or0KR8", //Upload Paper video
14      "https://www.youtube.com/embed/XsC7SOLw5iU", //Upload CSV video
15  ];
16
```

Editing Help Video Settings

If you wish to change more than just the link a video points to (change the settings that display the video), you can navigate to the source code files for each of the targeted pages and edit the settings there. The currently affected files are:

`src/components/EditPaper.js`

`src/components/Search.js`

`src/components/Tags.js`

`src/components/UploadPaper.js`

On each of these pages, you should be able to find an HTML tag that begins with the phrase: `<iframe`

These are the tags generated through YouTube's Share via HTML embed feature. An example one would be the tags page:

```
<Dialog open={this.state.helpVideo} onClose={this.openHelpVideo}>
    <DialogContent>
        <iframe width="560" height="315" src={HelpVideoURLS[4]} title="YouTube video player" frameborder="0" allow="accelerometer;
    </DialogContent>
    <DialogActions>
        <Button onClick={this.openHelpVideo}
            color="primary" autoFocus>
            Close
        </Button>
    </DialogActions>
</Dialog>
```

Adding New Help Videos

If you wish to add newly recorded help videos to the site, the best model to follow to replicate the same format is to look at `src/components/About.js`

While there is no help video for the About page, all the necessary code is there to insert a video. The 3 pieces of important information about displaying new videos are:

1. `helpVideo` state variable. Defined at the beginning of the class in the "state" declaration, there is a Boolean variable initialized to false. This flag determines whether we're displaying the video or not. This variable is defined in all class components that display a video.

2. The function `openHelpVideo` which is also defined in every class component that displays a video. This function simply toggles the `helpVideo` variable from True to False or False to True.

3. The last piece which is the `<Dialog>` tag seen on each of the classes that display a video. Its open condition is tied to the `helpVideo` variable, and the actual content (text or YouTube embed HTML) is displayed inside of the `<DialogContent>`

Screenshots of the 3 pieces are:

```
export default class About extends React.Component {

    state = {
        helpVideo: false
    }

```

```
31
32      openHelpVideo = () => {
33          this.setState((prevState) => ({ helpVideo: !prevState.helpVideo }));
34      }
35
```

```
71                      <Dialog open={this.state.helpVideo} onClose={this.openHelpVideo}>
72                          <DialogContent>
73                          </DialogContent>
74                          <DialogActions>
75                              <Button onClick={this.openHelpVideo}
76                                  color="primary" autoFocus>
77                                  Close
78                              </Button>
79                          </DialogActions>
80                      </Dialog>
```

There are imports for: `Dialog, DialogContent, DialogActions, Button` at the top of each file that uses a video popup. These will also need to be imported if you're adding a new video to a page that doesn't already display videos.

9.3 Changing Mailer Instructions

If you need to edit what email account is sending out the verification/password reset emails, there are 2 files that need editing (assuming you're at the root of the GitHub repo):

1) *api/sendActivation.php*

2) *api/resetPassword.php*

There are a few lines of code that would need to be edited to use a new email address:

```
29      $mail = new PHPMailer(TRUE);
30      try {
31          $mail->setFrom('regularspam34627@gmail.com', 'Entangled Philosophy');
32          $mail->addAddress($row["email"], $inData["username"]);
33          $mail->Subject = 'Password reset for ' . $inData["username"];
34          $message = 'Your new password is: ' . $new_pass;
35          $mail->Body = $message;
36
37          $mail->isSMTP();
38          $mail->Host = 'smtp.gmail.com';
39          $mail->SMTPAuth = TRUE;
40          $mail->SMTPSecure = 'tls';
41          $mail->Username = 'regularspam34627@gmail.com';
42          $mail->Password = 'TestPass123!';
43          $mail->Port = 587;
44
45          $mail->send();
46          echo '{"status":"successfully reset password"}';
47      }
```

1) Line 31, $mail->setFrom() needs to be edited to send from the new email address.

2) Line 38, $mail->Host = **smtp.[emailProvider].com** instead of gmail

3) Line 41 and 42, replace these with the credentials of the email account.

This will start sending emails from the new account. Note that it's possible that some email services will consider emailing remotely somewhat suspicious and may block the actions until you modify the email account's settings.

The lines to change in *api/sendActivation.php* are similar. Just change the setFrom() function, the Host if it isn't Gmail, and the username and password for the account.

A tutorial for how to set up and use PHPMailer can be found here:
https://alexwebdevelop.com/phpmailer-tutorial/

9.4 Project Edit Instructions

If you need to move the source code and build files, there are a few steps that need to be completed.

Moving the location of the "build" folder

To make sure our source code runs on a LAMP stack (i.e. without using node), we run a React build command which will allow us to move around the "build" folder (which is just HTML) to any location we see fit.

There's a key line of code in the beginning of the React application that sets up the page routing of the application. This line of code is in *entangled/src/App.js* and is the start of the Router tag (what manages the page routing):

```
39      <div className="container" id="outer-container">
40        <Router basename={'/~entangledPhilosophy/Entangled-Philosophies/entangled/build'}>
41          <div id="images" >
```

The "basename" parameter here is critical to the functionality of the page routing. It should be set to the exact directory the build folder will be moved to on the home server (chdr.cs.ucf.edu). If this basename parameter is set to the accurate location of the build folder, all page routing will work as intended.

Changing the location of the API endpoints

Throughout development, the API endpoints were located at the root of the GitHub repo but in case the location of the api folder changes (or the file upload location), there are some files that need to be updated:

1) *entangled/src/api.js*

2) *api/database.php*

If you are changing the location of the API endpoints, update line the variable urlBase on line 4 in *api.js* to the new URL of the API endpoints:

```
3   export const supported_languages = ["eng", "ger"];
4   export const urlBase = 'http://chdr.cs.ucf.edu/~entangledPhilosophy/Entangled-Philosophies/api';
5   export const fileURLBase = 'http://chdr.cs.ucf.edu/~entangledPhilosophy/paper/';
6
```

If you are changing the location of the file uploads for papers, change the variable fileURLBase in the screenshot above (in *api.js*) and change the variable *$paperurl* in *api/database.php*:

```php
<?php
$host = "chdr.cs.ucf.edu";
$dbname = "entangled_philosophy";
$username = "entangled_philosophy";
$password = "GjOIYa1O1OpwQR8X";
$paperurl = "/home/entangledPhilosophy/public_html/paper/";
?>

```

Note that *$paperurl* is defined as the relative location of the directory and not the exact URL like *api.js*.

Cloning the project and making edits

To clone the project and make edits to the source code, you must be on a machine that has node installed (making the HTML build requires a node command). Assuming your machine has node, if you clone the GitHub repo there may be a series of package dependencies you need to install for the code to compile.

The full list of these packages can be found in: *entangled/package.json*. In this file there's a JSON section labelled "dependencies" that list every dependency and their version number used for the project. The full screenshot of them is:

```json
"dependencies": {
  "@devexpress/dx-react-chart": "^2.7.6",
  "@devexpress/dx-react-chart-material-ui": "^2.7.6",
  "@devexpress/dx-react-core": "^2.7.6",
  "@fortawesome/fontawesome-svg-core": "^1.2.35",
  "@fortawesome/free-solid-svg-icons": "^5.15.3",
  "@fortawesome/react-fontawesome": "^0.1.14",
  "@material-ui/core": "^4.12.2",
  "@testing-library/jest-dom": "^5.12.0",
  "@testing-library/react": "^11.2.6",
  "@testing-library/user-event": "^12.8.3",
  "public-ip": "^4.0.4",
  "react": "^17.0.2",
  "react-burger-menu": "^3.0.6",
  "react-cookie-consent": "^6.2.4",
  "react-dom": "^17.0.2",
  "react-google-charts": "^3.0.15",
  "react-papaparse": "^3.16.1",
  "react-router-dom": "^5.2.0",
  "react-scripts": "^4.0.3",
  "react-table": "^7.7.0",
  "react-table-scrollbar": "^0.1.11",
  "react-wordcloud": "^1.2.7",
  "recharts": "^2.0.10",
  "universal-cookie": "^4.0.4",
  "web-vitals": "^1.1.2"
},
```

When cloning the project, depending on your system it's possible that some of the packages won't need to explicitly be installed but the terminal commands you can use to install all these dependencies are of the form: **npm install [packageName]**

So, for example, to install react-table you would run **npm install react-table** or to install the @material-ui/core package you would run **npm install @material-ui/core**.

If all packages are installed and you want to run the project in debug/localhost, run the command **npm start** in the *entangled* folder. If you're satisfied with the changes to the source code, go to the *entangled* folder and run **npm run build**. This will create a folder titled "build" in the same directory which you can then move around to the directory of your choice. This "build" folder is the HTML that allows the react project to run on servers without node installed.