Front End Overview:

- The front end uses VueJS 3 to render and the Quasar framework to render in front end reactive components.
- The front uses Pinia (a fluid framework) for state management on the front end as well as Axios to perform api calls.
- It uses a keycloak sdk to handle all interactions with the keycloak instance that is running and uses html2pdf to convert the bibliographies to pdfs.
- There are 2 scripts for startup that are worth noting. The first is `quasar dev` which sets up a development instance of the backend node js server which Vue and Quasar sit on top of. The second is `quasar build` which minifies the typescript and removes the development web server so that the content can be served using something like NGINX or APACHE.
- For the folder structure there are a few things to note in the front end
    - The api directory contains models that are used by the middle tier to deserialize JSON into a typed object
    - The auth directory contains all services to handle passing auth with the OAuth2.0 workflow
    - The boot folder consists of setup functions and injections into the Quasar framework
    - The pages directory contains the majority of the Vue component code.
    - The router directory contains the logic for switching between different URL routes
    - The stores directory contains all the app state management using Pinia store files.

Potential Front End Additions / Suggestions

- Complete the feedback email hookup on the front page
- Allow user management from the application for admin accounts so that the admin doesn't have to go through keycloak to do it.
- Generate APA style bibliographies and additional citation types
- Generate different styles for the generated PDF file.
- Be able to fully deploy the application which was not able to be conducted for this iteration of the application unfortunately.

Front End Resources

- https://quasar.dev/docs/
- https://vuejs.org/guide/introduction.html
- https://axios-http.com/docs/intro

Middle Tier Overview:

- The middle tier is built upon Python's FastAPI in addition to Keycloak which we use for our user verification. We are using FastAPI to create endpoints for our "User History", "User Bookmark", and "User Bibliographies" features.
- We are using MongoDB to store our data collected via the "User History" and "User Bookmark" and "User Bibliographies" features. We have set up the MongoDB instance to open on port **2717**.
- We created helper methods in */middle/config/mongoUtil.py* which aid in creating, updating and deleting documents along with any collections or databases needed to run this project.
- We also created a python script (*/middle/config/dbStartupScript.py*) to initialize the database to our liking.
- There are three main directories in the middle tier.
- */middle/models* holds how data is structured our MongoDB instance
- */middle/routes* contains all our endpoints for the bookmarks, history, bibliography and keycloak functionality
- */middle/schemas* contains our code for how we specify what structure we will be returning the data from the database
- */middle/index.py* collects all the routes and builds the application
- The Keycloak API gets initialized with the */middle/index.py* file.
- The Keycloak code is located at the */middle/routes/keycloak.py* file.
- Keycloak is available as a docker container.
- API is implemented to listen for the httpx requests directed at the FastAPI endpoint when it is running.
- Additionally, the API is implemented based on functionality of Keycloak's Admin REST API.
- Parameters are passed within the body of a request toward the FastAPI endpoint.
- One of the API's function parameters is the request itself from which the bearer token of a user (JWT) gets extracted from the "Authorization" header of the request.
- The bearer token gets created with each authentication and is stored in the Keycloak's docker container for a limited amount of time.
- The bearer token is used during the authenticated session to communicate privileges (user or admin) when issuing Admin REST API requests toward the Keycloak server.
- Additionally, the Keycloak API supports extraction of a user id from the bearer token which can be done by decoding the bearer token and accessing the "sub" field. For example: userID = decoded_bearer_token["sub"].

Middle Tier Process:

- Front end sends HTTP method requests to the FastAPI backend. Depending on the type of requests, we check whether the user is signed in or not. If they are signed in, features like bookmark and history are now available.
- If the information in the requests is in order with our specification, the request will go through and return a status code. This status code will tell the front end if the request has successfully gone through.
- The Keycloak API found under */middle/routes/keycloak.py* is integrated with the front end and helps with authentication of the users.
- The API supports functionality to do the following: authenticate user, retrieve all users, retrieve a specified user, set a password, create a user, delete a user, promote a user to an Admin, demote an Admin to a user, enable, and disable a user.

Potential Middle Tier Future Additions/Suggestions:

- Create deletion system to clear user history after a certain time period
- Implement more options on user deletion and the deletion of a users data
- Expanding on bookmark features and history features
- Reviewing Keycloak configuration in depth and making changes based on the needs.
- Adding additional Keycloak functionality using the resources provided below as a guide (keycloak rest api). Postman can be used for testing the requests before implementation.

Middle Tier Resources

- https://fastapi.tiangolo.com/
- https://www.mongodb.com/docs/manual/
- https://www.keycloak.org/documentation
- https://docs.docker.com/reference/
- https://www.python-httpx.org/
- https://www.uvicorn.org/
- https://jwt.io/
- https://www.keycloak.org/docs-api/15.0/rest-api/index.html
- https://www.postman.com/
- https://www.keycloak.org/getting-started/getting-started-docker

Back End Overview:

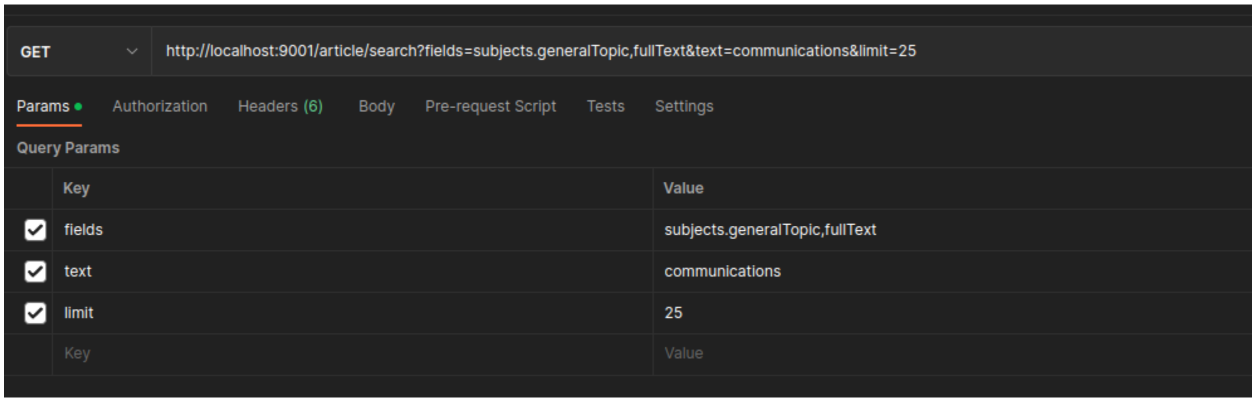
- The back end consists of Spring Boot for our REST API, Hibernate Search using Apache Lucene for database indexing and searching, Apache PDFBox and POI to parse pdf files and database excel dumps, and Lombok for annotation based class boilerplate without bloating the code.
- Configuration can be found in backend/spring/src/main/resources/application.yaml
- Libraries downloaded through maven are specified in backend/spring/pom.xml
- Application.java is where the application is run and has some configuration information present through annotations, other configurations can be found in backend/spring/src/main/java/com/capgroup/spring/config/
- All endpoints the frontend makes requests to can be found in ArticleController
- Be wary of updating existing frameworks to newer versions, as methods are frequently deprecated from version to version
- Postman can be used to verify portions of the backend without needing to launch the rest
- Generated JavaDocs can be found in documentation.zip in the backend/spring directory

Back End Process:

- Front end sends HTTP method request to back end which is handled in ArticleController
- If a valid request (required fields present, bearer token authentication if a request that requires admin privileges), forwards request parameters to the proper service.
- Service then handles requests made to the repository
- Repository builds Hibernate Search query if necessary, then returns hits for documents that match
- If process was successful, returns results and a response entity with a matching HTTP status code

If Testing Back End Through Postman:

- Default port for back end is 9001, so the root URL should be http://localhost:9001/article/
- For endpoints that do not require admin privileges like general search, select the correct request method and put query parameters and send request while backend is running

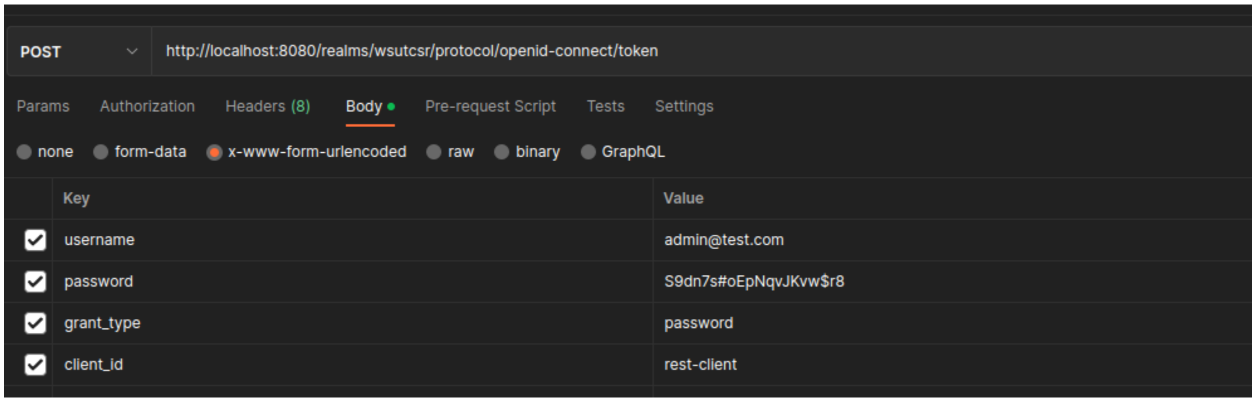

GET http://localhost:9001/article/search?fields=subjects.generalTopic,fullText&text=communications&limit=25

	Key	Value
✓	fields	subjects.generalTopic,fullText
✓	text	communications
✓	limit	25
	Key	Value

- For requests that require admin permissions, an instance of Keycloak must be running through either Docker or a daemonless container manager like Podman which can emulate Docker commands
- To launch Keycloak through Docker/Podman, enter these commands:

```
#load image of existing docker instance
sudo docker load < wsutcsr.tar

#get the ID of new image
sudo docker image ls -a

sudo docker tag <image ID> wsutcsr

#check again to ensure image repository has been changed
sudo docker image ls -a

sudo docker run -p 8080:8080 wsutcsr start-dev
```

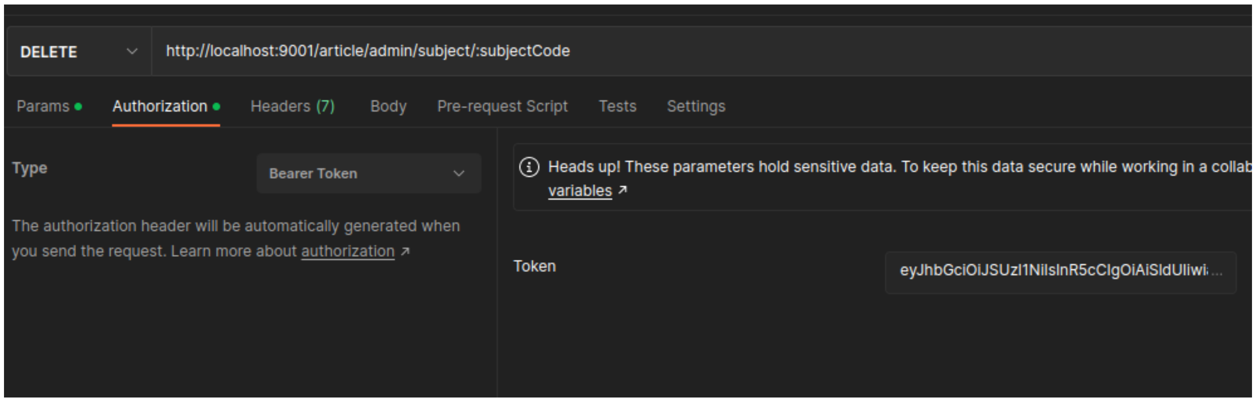
- With the Keycloak instance running, request a bearer token using an admin account with valid credentials (example in screenshot not valid and will need to be checked)



POST http://localhost:8080/realms/wsutcsr/protocol/openid-connect/token

Key	Value
✓ username	admin@test.com
✓ password	S9dn7s#oEpNqvJKvw$r8
✓ grant_type	password
✓ client_id	rest-client

- On a successful request, the bearer token will be given which can then be copied and sent to any requests that require admin privileges in the authorization tab for as long as the token is valid



DELETE http://localhost:9001/article/admin/subject/:subjectCode

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collab
variables ↗

Token: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwi...

Back End Testing

- To test functionality of the backend, we performed integration testing via Podman and Postman
- Postman was used to test endpoints and methods
- All tests led to positive results or revealed bugs that have since been fixed
- Full external testing requested by the sponsor could not be completed due to time constraints and delayed deployment

Potential Back End Future Additions/Suggestions:

- Batch insertion/deletion of articles and subjects
- An endpoint that rebuilds the database on request (may require excel files and all pdfs to be stored on server rather than just the indexed database)
- Normalize sources in articles to be an IndexEmbedded entity with a OneToMany relationship and create a SourceRepository with corresponding endpoints to add/update/delete sources like subjects
- Cleanup repositories so that SubjectRepository does not inherit unused methods
- Reorganize configuration structure so that all configuration settings to to /config/ rather than having some be present in Application.java
- Add numerical support for searching years in general search (currently only works with strings), will require handling different cases for whether or not a token can be converted to a numeric type or not, potentially move away from lambdas in where clause to support this (referring to SearchRepositoryImpl methods)
- Change Subject's ID to a numeric type once better numeric support is implemented
- Implement weights for different fields in general search
- More exception handling on various repository actions (was left frozen as to not cause any regressions during integration)
- Unit testing suite that does not overwrite the existing database which ensures Controller methods are all giving expected outputs
- Cleanup potentially redundant annotations, write explicit getters/setters to remove annotation bloat
- Normalize author source data for better citation generation
- Save logging information to a text file

Back End Resources:

- https://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/
- https://gauthier-cassany.com/posts/spring-boot-hibernate-search
- https://github.com/thomasdarimont/keycloak-project-example
- https://spring.io/guides
- https://docs.spring.io/spring-data/data-commons/docs/3.0.0/reference/html/
- https://www.postman.com/
- https://podman.io/