(To see the latest version of this document, follow this link;
https://docs.google.com/document/d/1uiEfdH5sfOISFvWGlkH-m3wrOLfjJfTHfc_Ga5Cqz_4/edit?usp=sharing)

Pass

Pass (Protocol Aggregation and Sharing System) is a web application intended to help researchers coordinate the resupply of materials only available for purchase in quantities exceeding the needs of a single lab. It was built primarily for UdeM (the University of Montreal), but has been designed such that it could easily be wired for a group of labs from a different university, or even turned into an open website.

In short, Pass is a platform with lab-level protocol creation and editing, but with cross-lab protocol visibility.

History;

The initial development of Pass was quite sporadic, caused by my progressing understanding and knowledge on fullstack web applications. It started off as a LAMP stack application, before being remade using the MERN stack for better handling of routes and states. It might eventually step away from the 'stack' ideology, to settle on disconnected but more appropriate technologies, but this has yet to be decided.

AN: I will mention that I've been thinking of implementing a typescript based system, alongside postgres and a more opinionated class-based frontend system like angular, but ended up deciding to stick to the mern stack which I was more familiar with, and had better hosts for. (there weren't any good free tier with the postgres hosts I had found)

Host

The Pass application is currently hosted on render as a web application, using their free tier. They provide both an https wrapper around the entire server as well as an address;

https://pass-udem.onRender.com

Technologies

MERN stack application

Pass was built as a MERN stack application, wherein the stack mentality is that the technologies of the stack work well together. In this case you have;
MongoDB: core data
Express: backend route management tool
React: core frontend (served statically by the frontend)
Node: core backend

Dependencies

Both the server and client use various modules to help fill in some functions not offered by the core technologies. These lists include all dependencies used in the code, including some that are a part of core technologies. Those that are a part of core technologies are marked with *.

Server dependencies;

- bcrypt: password encryption
- compression: file size reduction
- cors: cors header preset
- dotenv: .env setting file scanner
- *express: general server creation tool
- express-rate-limit: request limiter
- helmet: general header preset
- imagekit: credential generator for client-side file uploads
- joi: input validation
- jsonwebtoken: signed credential token
- *mongoose: mongoDB interface tool
- mongoose-unique-validator: mongoose module for unique variables
- *fetch: http request tool (comes with node)
- *URL: url composition tool (comes with node)
- *crypto: non-readable data encryption tool (comes with node)
- *path: folder path composition tool (comes with node)

Client dependencies;

- *react: dynamic UI building library

- *react-dom: react rendering engine for web browsers.
- bootstrap: style library
- bootstrap-icons: icon library

API services;

The server and client use services offered by various apis, as listed here;

- Zeruh: email verification service	[used in backend]
- MongoDB: structural database service	[used in backend]
- Imagekit: data database service	[used in frontend]
- Google: used for OAuth authentication	[used in backend]

Database structure

Pass operates from a single server, serving both client and api (server) content, with the '/api' routes for the latter. This api layer interfaces with the mainly structural mongodb database (the ImageKit database is used for resource-heavy files like images, pdfs, or zipped folders). It hosts five types of documents, as described here, each in their own collection.

```
collection: formats
format object;
        {
                _id ObjectID,
                name String,
                description String,
                componentsModel [
                        {
                                name String,
                                type String
                        }...
                ]
        }
```

```
collection: laboratories
laboratory object;
        {
                _id ObjectID,
                name String,
                description String
        }


collection: users
user object;
        {
                _id ObjectID,
                handle String,
                email String,
                isAdmin Boolean,
                isEnabled Boolean,
                credentials {
                        local String,
                        google String,
                        microsoft String
                }
                name String,
                laboratoryID Mixed
        }


collection: resources
resource object;
        {
                _id ObjectID,
                isImage Boolean,
                placeholder String,
                name String,
                description string
        }


collection: protocols
protocol object;
        {
                _id ObjectID,
                description String,
                resourceIDs [
                        ObjectID…
                ]
                laboratoryID ObjectID,
                contributorIDs [
                        ObjectID…
                ],
                lastModificationTime Date,
                protocol String,
                formatID ObjectID,
                components [
                        {
                                name String,
                                value String
                        }...
                ]
        }


collection: groups
group object;
        {
                _id ObjectID,
                name String,
                description String,
                protocolIDs:[
                        ObjectID…
                ],
                adminOnly Boolean,
                laboratory ObjectID,
                contributorIDs [
                        ObjectID…
                ],
                lastModificationTime
        }
```

API communication

To communicate, pass generally uses json files structured in packets. A packet is a json object with two fields; type to describe the data's structure, and content to locate the data structure. Packets are mainly used in replies, but some routes require data input of specific types of packets. Additionally, in some cases, cookies are used to transmit data, although this data itself is again in the form of packets if the data is being given to the client.

The server internally uses errors to exit promise chains, but associates errors with http code, which are then used for the response. In short, any and all responses from the server will be packet-based, even with error codes like 500 or 404.

Packet types

error type: when an error occurs, this type is used to communicate the issue, with the target specifying the cause or source of the issue.
usage: output
data format;
```
{
    type String,
    target String,
    message String
}
```

confirmation type: used to communicate a message to show to the user.
usage: output
data format;
```
{
    message String,
    isMinor Boolean
}
```

reference type: used to relay the object id of the new object.
usage: output
data format;
```
ObjectID
```

multiple type: used to pass back multiple packets at once.
usage: input-output
data format;
```
[
    Packet…
]
```

authentication type: used to pass login information.
usage: input
data format;
```
{
    principal string,
    credential string
```

```
	}
```

signup type: used to pass partial signup information from OAuth.
usage: output
data format;

```
	{
		credential {
			local Null,
			google String | Null,
			microsoft String | Null
		},
		email String
	}
```

status type: used to pass requesting users' permissions. When sent by the server, usually accompanied by a credentials cookie (status* means just the status, no cookie).
usage: input-output
data format;

```
	{
		userID ObjectID,
		isAdmin Boolean,
		isEnabled Boolean,
		laboratoryID ObjectID | Boolean
	}
```

search type: used to return the search suggestions
usage: output
data format;

```
	{
		name String,
		searchType String,
		searchID ObjectID
	}
```

format type: used to pass protocol application information.
usage: input-output
data format

```
	{
		formatID ObjectID,
		formatName String,
        description String,
        componentsModel [
            {
                name String,
                type String
            }...
        ]
    }
```

laboratory type: used to pass laboratory information.
usage: input-output
data format;

```
    {
        laboratoryID ObjectID,
        laboratoryName String,
        description String
    }
```

user type: used to pass user information.
usage: input-output
data format;

```
    {
        userID ObjectID,
        userHandle String,
        email String,
        isAdmin Boolean,
        isEnabled Boolean,
        credentials {
            local String | Null,
            google String | Null,
            microsoft String | Null
        },
        name String,
        laboratoryID ObjectID | Boolean,
        laboratoryName ?String
    }
```

*the credentials element is always present, but is set to null for security when sent by the server

resource type: used to pass resource information.
usage: input-output
data format;

```
    {
        resourceID ObjectID,
        placeholder String,
        resourceName String,
        isImage Boolean,
        description String
    }
```

protocol type: used to pass protocol information.
usage: input-output
data format;

```
    {
        protocolID ObjectID,
        description String,
         protocolFiles [
            {
                resourceID ObjectID,
                resourceName String,
                link String,
                description String
            }...
        ],
        imageFiles [
            {
                resourceID ObjectID,
                resourceName String
            }...
        ],
        laboratoryID ObjectID,
        laboratoryName String,
        contributors [
            {
                contributerID ObjectID,
                userHandle String
            }...
        ],
        lastModificationTime Date,
        protocol String,
        formatID ObjectID,
        formatName String,
        components [
            {
                name String,
                        type String,
                        value String
                    }...
                ]
            }
```

group type: used to pass group information.
usage: input-output
data format;

```
    {
        groupID ObjectID,
        groupName String,
        description String,
        protocols:[
            {
                protocolID ObjectID,
                laboratoryName String,
                formatName String
            }...
        ],
        isAdminOnly Boolean,
        laboratoryID ObjectID,
        laboratoryName String,
        contributors [
            {
                contributorID ObjectID,
                userHandle String
            }...
        ],
        lastModificationTime Date
    }
```

Error targets

When the api encounters an error in its processes, it returns that error's message to the client, alongside a target having caused or been involved in the error. The majority of errors will be due to input requirements not being respected, in which cases a generic format type error will be thrown. For format error, the target is a folder style path from the type down to the specific component which has an issue. Some format errors are hidden under wrap error, either for safety or simplicity. Lastly, errors pertaining to system malfunction and api issues are returned as server errors.

Here is a list of the non generic error types;

type	cause	issue
wrap	login	An issue during login
server	api	An issue with an API
server	broken	Misuse of server elements
server	unknown	An unanticipated or unhandled error
wrap	server	An issue with the server

Here is are some examples of format errors;

type	cause	issue
format	authentication/principal	A format issue
format	user/credentials/local	A format issue

Cookies

There are three cookies that are used by the application. The authorization cookie, the state cookie and the data cookie.

The auth (authorization) cookie carries the proof of session which allows users to stay connected from one page to another and make requests to the server. It contains the user id as well as a jwt token which is encrypted on the server side with the jwt secret. This cookie does not use the packet format, but is also only used internally by the server, the client doesn't ever need it or read it. It's set to last as long as the authentication token it carries.

The state cookie is used to ensure that the client receives the correct callback response during the OAuth process. It contains the randomly generated state and nonce strings, and similarly to the auth cookie, it does not use the packet format. It is short-lived, killed by the server on the callback phase of the OAuth process.

The data cookie is used to transmit information back to the client. It is used solely for information that is not confidential, such as a signup or status packet. It is short-lived, killed by the client once the information is retrieved.

OAuth and OpenID

OAuth, or open authentication, is a protocol which allows various systems to act as authenticators for applications, allowing applications to use apis on behalf of the users. The OpenID protocol (open identification), a sub-protocol of OAuth, allows users to log in with their already existing account, such as google or github. (Note that OAuth typically refers both to OAuth and OpenID in this document; OAuth is only ever used with OpenID.) This process takes place in the user's browser rather than on the server, and allows the authenticator to provide certain information to the application with user agreement. The basic flow is client redirect, server setup, outer agreement, server callback and back to client.

To make this flow possible (specifically the callback segment), the application typically needs to be signed up with the authenticator, which can vary in complexity. Signing up provides you with a client id and secret, with the secret used to redeem tokens given back in the server callback, and client id used to make the redirect request in server setup

In more details, the OAuth flows is
1: the client redirects to the server OAuth route, something like /api/auth/"outer".
2: the server randomly generates a state and a nonce variable, which are stored in the state cookie. It then redirects the browser to the authenticator's OAuth site, alongside the request components;

response_type,	(the type of data requested: token)
client_id,	(the client id given during signup)
redirect_uri,	(the callback uri)
scope,	(the variables/elements needed; openid, email and profile)
state and nonce.	(the security/protection variables)

3: the authenticator, after consent is given, redirects the browser to the server's callback uri, alongside two components, a code and a state.
4: the server retrieves the state and nonce stored in the cookie, consuming it. It then verifies that the state in the request is the same as the one that was stored in the cookie. After that, it sends a post request to the authenticator's OAuth token site, with the following components form-encoded;

client_id,	(the client id given during signup)
client_secret,	(the client secret given during signup)
code,	(the authorization code in the request)
redirect_uri,	(the callback uri)
grant_type	(the type of token requested: authorization_code)

The response will then include 5 elements; access_token, expires_in, id_token, scope and token_type. The server then validates the id_token with;

the audience (the client_id, same as the other times),
the nonce (previously stored in the cookie),
the expiry (the date of expiration being in the future),
the signature (the authenticator's public key) and
the issuer (one of the authenticator's listed endpoints)

Finally, the server attempts to find an account using the email and account id provided by the validated id_token, and then either sets the authorization cookie or sets a data cookie with a signup packet.

API routes

Both the server and client use routes, with most present for both, except for a few which are server only (server routes generally are the same as client routes, but with /api at the start). Some routes have inputs, others don't, but they all have outputs of at least an error. Some routes are only accessible to certain members, based on their access levels.

For reference, the access levels are;
public: there are no user restriction
non-member: the user is not a member
member: the user is a member
admin: the user is an admin member
member-connected: the member id is present in relevant document
lab-connected: the user's lab id is present in relevant document

AN: for clarity, a route noted as client essentially means the client has a display associated with the action of the route (directly or by page mode), that the route isn't used purely internally.

Search routes

/search/:searchterm
- read;
    - client and server route
    - access level: public
    - inputs: :searchterm
    - outputs: error | multiple[search…]

/search/:searchtype/:searchterm
- read;
    - client and server route
    - access level: public
    - input: :searchtype :searchterm
    - output: error | multiple[search…]

Laboratory routes

```
/laboratory
        read;
                client and server route
                access level: public
                input:
                output: error | multiple[laboratory…]
        write;
                client and server route
                access level: admin lab-connected
                input: laboratory
                output: error | multiple[confirmation, reference]

/laboratory/:identifier
        read;
                client and server route
                access level: public
                input: :identifier
                output: error | laboratory
        update;
                client and server route
                access level: admin lab-connected
                input: :identifier | laboratory
                output: error | confirmation
        delete;
                client and server route
                access level: admin
                input: :identifier
                output: error | confirmation | multiple[confirmation, status]

/laboratory/:identifier/members
        read;
                client and server route
                access level: public
                input: :identifier
                output: error | multiple[user…]
```

Format routes

```
/format
        read;
        client and server route
        access level: public
        input:
        output: error | multiple[format…]
    write;
        client and server route
        access level: admin
        input: format
        output: error | multiple[confirmation, reference]


/format/:identifier
    read;
        client and server route
        access level: public
        input: :identifier
        output: error | format
    update;
        client and server route
        access level: admin
        input: :identifier format
        output: error | confirmation
    delete;
        client and server route
        access level: admin
        input: :identifier
        output: error | confirmation
```

User routes

```
/user
    read;
        client and server route
        access level: public
        input:
        output: error | multiple[user…]


/user/:identifier
    read;
        client and server route
        access level: public
        input: :identifier
        output: error | user
    update;
        client and server route
        access level: member admin member-connected
        input: :identifier
        output: error | multiple[confirmation, status]
    delete;
        client and server route
        access level: member admin member-connected | admin lab-connected
        input: :identifier
        output: error | multiple[confirmation, status] | confirmation


/requests
    read;
        client and server route
        access level: admin
        input:
        output: error | multiple[status*…]
    update;
        server route
        access level: admin lab-connected | admin*
        input: status
        output: error | confirmation | multiple[confirmation, status]
```

*special case for users requesting new labs, any admin can authorise them.

Protocol routes

```
/protocol
    read;
        client and server route
        access level: public
        input:
        output: error | multiple[protocol..]
    write;
        client and server route
        access level: member admin
        input: protocol
        output: error | multiple[confirmation, reference]
```

```
/protocol/:identifier
        read;
                client and server route
                access level: public
                input: :identifier
                output: error | protocol
        update;
                client and server route
                access level: member member-connected | admin lab-connected
                input: :identifier protocol | :identifier multiple[protocol, resource…]
                output: error | confirmation
        delete;
                client and server route
                access level: member member-connected | admin lab-connected
                input: :identifier
                output: error | confirmation


/protocol/:identifier/examples
        read;
                client and server route
                access level: public
                input: :identifier
                output: error | multiple[resource…]
        update;
                client and server route
                access level: member member-connected | admin lab-connected
                input: :identifier resource
                output: error | confirmation


/protocol/:identifier/examples/:resource
        update;
                client and server route
                access level: member member-connected | admin lab-connected
                input: :identifier :resource resource
                output: error | confirmation
        delete;
                client and server route
                access level: member member-connected | admin lab-connected
                input: :identifier :resource resource
                output: error | confirmation
```

Group routes

```
/group
    read;
        client and server route
        access level: public
        input:
        output: error | multiple[group]
    write;
        client and server route
        access level: member
        input: group
        output: error | multiple[confirmation, reference]


/group/:identifier
    read;
        client and server route
        access level: public
        input: :identifier
        output: error | group
    update;
        client and server route
        access level: member member-connected | admin lab-connected
        input: :identifier group
        output: error | confirmation
    delete;
        client and server route
        access level: member member-connected | admin lab-connected
        input: :identifier
        output: error | confirmation
```

Authorization routes

The implementation of OAuth requires some additional routes for the redirection process which gives it its security by keeping the processes within the browser. These routes are not reflected between the client and the server, each having different routes for the process. All auth routes, including OAuth, are listed here;

```
/signup
/api/auth/local/signup
```

```
        write;
                client and server route
                access level: non-member
                input:
                output: error | multiple[confirmation, status]


/login
/api/auth/local/login
        read;
                client and server route
                access level: public
                input: authentication
                output: error | status | confirmation


/self
        read;
                server route
                access level: member
                input:
                output: status*


front redirect to;
/api/auth/google
        link;
                server route
                access level: public
                redirect: https://accounts.google.com/o/oauth2/auth
                sets a temporary cookie


/api/auth/google/callback
        link;
                server route
                access level: public
                redirect: https://pass-udem.onRender.com/user/:identifier
                removes the temporary cookie
```

Settings

Headers

cors;
    origin: 'https://pass.onRender.com'
    credentials: true

This lock out http requests coming from other addresses than itself, but not direct loads (such as opening the site from a link)

helmet;
    content security policy;
        default-src: 'self'
        img-src: 'self' https://ik.imagekit.io

This only allows data to be fetched from the backend (itself) or from imagekit which is the data database.

Cookies

```
auth cookie;
    name: userAuth
    value:{
        token: jwt token          (the token is made from the userID)
    }
    HttpOnly: true                (The cookie cannot be read using javascript)
    secure: true                  (The cookie is only sent on https requests)
    maxAge:3600000                (The cookie will last 1 hour, just like the jwt token)
    sameSite:Strict     (The cookie is only sent when pass makes http requests to pass)
    path:/api                (The cookie is only sent with request to pass' server)

state cookie;
    name: userAuth
    value:{
        state: String             (A verification string for the request and callback)
        nonce: String             (A verification string for the response's data)
    }
    HttpOnly: true                (The cookie cannot be read using javascript)
    secure: true                  (The cookie is only sent on https requests)
```

```
        maxAge:300                      (The cookie will last 300 milliseconds)
        sameSite:Lax                    (The cookie is sent with all http requests to pass)
        path:/api/auth          (The cookie is only sent with request to pass' auth routes)

data cookie;
        name: userAuth
        value:{
                type: signup            (The value is a signup packet)
                content: ...    (The type and content are the same as any other packet )
        }
        HttpOnly: true                  (The cookie cannot be read using javascript)
        secure: true                    (The cookie is only sent on https requests)
        maxAge:300                      (The cookie will last 300 milliseconds)
        sameSite:Strict     (The cookie is only sent when pass makes http requests to pass)
        path:               (The cookie is only sent with request to pass' server)
```

API

mongoDB connection string:

```
mongodb+srv://passUser:${DB_Password}@pass.oz7asfg.mongodb.net/?retryWrites=tr
ue&retryReads=true&w=majority
```

This string is used to connect with the mongodb cloud database service.
It's composed of the address;

```
mongodb+srv://passUser:${DB_Password}@pass.oz7asfg.mongodb.net/
```

And four variables; retryWrites, retryReads, w and appName.

```
        retryWrites = true          (if a write fails due to bad connection, it'll retry)
        retryReads = true           (if a write fails due to bad connection, it'll retry)
        w=majority          (a majority of shards must accept the write for it to be valid)
```

Zeruh api string:

```
https://api.zeruh.com/v1/verify?api_key=${ZERUH_KEY}&email_address=${email}
```

This string is used to send email verification requests to the Zeruh api.
It's composed of the address;

```
https://api.zeruh.com/v1/verify
```

And two variables; api_key, email_address

```
        api_key                     (the ZERUH_KEY in the secrets)
        email_address               (the email address to verify)
```

Secrets

The secrets are stored in this document;
https://docs.google.com/document/d/1kxTrQPYJjSYdG8XHf5A69gmYfbuas0eq9pO--GVLUJc/edit?usp=sharing

installation

In general, the installation is pretty simple, outside of the specifics of the host. Generalizing to the node requirements, you'll only really need 2-3 things; git repo, the start command and possibly the build command. The commands are "npm start" and "npm run build", and should be run in the source folder containing the server and client folder and the package.json file.

For testing, it is recommended to use separate databases/api keys to avoid interference with the stable version(s).

When making a major transition (like say going from v1 to v2), the previous version should be split off onto its own branch and frozen for reference.