Piqued

Adam Temesvary[1], Braedon Wooding[2], Jimmy Chen[3], Nicholas Thumiger[4], Matthew Lim[5]

Abstract
A messaging application focused on encouraging academic conversation.

[1] *a.temesvary@student.unsw.edu.au (z5075774)*
[2] *b.wooding@student.unsw.edu.au (z5204996)*
[3] *jimmy.chen1@student.unsw.edu.au (z5258222)*
[4] *n.thumiger@student.unsw.edu.au (z5206390)*
[5] *matthew.lim2@student.unsw.edu.au (z5112968)*

Contents

Introduction

Mission Statement Foster and create your own reverberation chamber founded on common interests.

Piqued aims to be an academic platform for communication unlike anything UNSW or the world has seen, we aim to accomplish this through the use of modern technologies that empower our ability to push out maintainable and fast code quickly.

In this report we'll detail very explicitly the decisions and choices that lead us to the finalisation of Piqued as well as a manual for how to setup/run it

1. Architecture

1.1 Overall Design

This is a rough overview of the entire architecture, a more detailed explanation follows this

1.1.1 Model View Controller

Piqued's architecture is straightforward, Next.js which is built upon ReactJS, a Javascript framework, acts as our colloquial 'front-end' and interacts with a Django (Python) 'back-end'.

There are two types of communication between them, firstly HTTP requests are used for simple changes such as registration of users, updating user information, updating profile pictures, and so on. These are structured in the standard way where the following 'verbs' translate to requested modifications to the underlying model.

- **POST**: Submit an updated state to an entity
- **GET**: Request an entity
- **PATCH**: Partial modifications to an entity
- **DELETE**: Removal of an entity
- and others (we only use the ones listed above however)

This aligns to the industry standard of developing scaleable and maintainable applications through Model View Controller (MVC).

The overarching design principle utilised in the construction of 'Piqued' has been the 'model-view-controller' paradigm. The goal of the principle is to separate the components of the software into blocks that have discrete objectives. A diagrammatic overview is presented in Figure 1

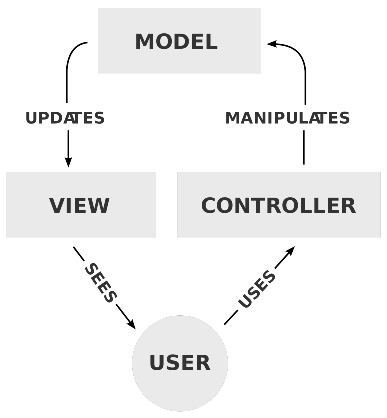


Figure 1. Interactions Between MVC Components [1]

The software that handles the user interaction, and displays the interface is known as the 'view'. In Piqued, the Next.js framework was selected for this purpose.

The software component that handles the functionality of the pages and handles user requests is known as the 'controller'. For instance, Piqued's ability to extract interests from a user's Facebook page and send messages is handled by the controller.

Finally, the software in charge of manipulating data and interacting with the database is known as the 'model', and is completely independent of the user. In Piqued, both the model and the view components are handled by the Django Framework: further discussed in Section 1.4.

Such a framework is common place across the software industry, and allows a degree of separation between the varying components. This has the advantage of reducing complexity, improving the ability to scale and reducing data access/manipulation vulnerabilities. Additionally, changes and performance improvements can often be made to each component independent of the others. This enables a rapid development process: a core requirement of the Piqued team due to the short turnaround time and development schedule.

Since Piqued's model and controller components were enforced by the Django framework, the inter-component communication was abstracted away. However, the utilisation of an independent technology (Next.js) for the view component required a manner of communication to the remainder of the system. A Representational State Transfer (REST) design framework was used for this purpose. This is a common approach, and underpins all of the advantages discussed above. The view makes regular hypertext transfer protocol (HTTPS) requests to the Django framework in order to engage the controller and produce results for the user. The only exception to this design principle are the chat messages themselves, which utilise a web-socket protocol and is discussed further in the next section.

This approach increases the degree of separation between the server and client resulting in simplified development and a reduced chance of bug riddled inter-dependencies. This also allows Piqued to deploy the server and client onto separate infrastructure, allowing for an improved ability to load balance for varying situations. The REST framework also enables the easy future implementation of a data cache, which will improve performance as Piqued scales up.

1.1.2 Publish Subscribe Model

The second method of communication is via websockets, these are implemented through Django's Channels as a publish subscribe model. This allows bi-directional communication which allows us to cheaply push message updates and histories without requiring a more archaic method like long polling. The downside of long polling is that it effectively is a 'pull' model where you constantly have to poll the webserver for more updates, this means you have a higher load on the server (slowing down requests), you also use more bandwidth on the client side, and updates typically don't show up as quickly (depending on polling interval).

Instead what we do is 'push' any updates from the server, there are 3 kinds of pushes that occur here.

Firstly, we have just standard message updates when the user has the app open, these are sent through websockets and represent all permutations of message updates (new messages, users joining/leaving groups, user status updates, message modifications/deletions, and so on). User's websockets on the server side subscribe to a 'channel' for each group they are part of allowing us to publish a change to an entire group.

Secondarily, we have web notifications, these are done through firebase which allows us to send push notifications that direct the user to the website **even** if they don't have it open! This allows us to increase user engagement and make sure users don't miss any important emails. To help with busy groups, users can choose to mute groups for a set period of time or forever if they so desire.

Lastly, we have email notifications, these only occur due to 2 events; password reset, and welcome email which is sent after the user registers into our system. This could be extended in the future as need arises to provide a summary of a channels discussion during the day, though this alone would be a massive scope and probably an entire project within itself. We currently use SendGrid [2] as a third party service provider to send these emails.

1.2 Models

Messages are stored in a wide column DB (Azure Table Storage) which allows us to lookup / modify specific messages in $O(1)$ time (in a very scalable fashion), and the use of a partition key allows us to efficiently segment our messages into groups allowing for more efficient mass history downloads for a specific group. This methodology was implemented by Discord (another popular messaging application) in its early days of existence, and worked well for them up to more than 100 million messages. As such, this is a very appropriate architecture for the application [3].

This sort of model allows for E2E (end to end) encrypted messages to be implemented since you can share a SAS token to a specific table which is encrypted using a shared group secret (that is unknown to us but stored on their devices) of course this was way out of scope for our proposal and would be a massive project by itself.

Attachments and profile images are stored using Azure Blob Storage, which behaves similar to AWS S3 Buckets, we use the 'hot storage' equivalent (cheaper downloads/streaming, more costly storage) since we typically have pretty small files and because it's faster to download files. This will increase our costs over time, however the typical industry solution to this is straightforward and only applicable when you reach mass scale. Effectively, you 'archive' old files to a cold storage table, this means files you sent more than a month ago are slower to download but cheaper to store (which is fine since old files are less likely to be downloaded / viewed).

We allow groups to 'subscribe' to news, weather alerts, blogs, and other updates via the use of a 'rss' feed ([4]), rss feeds are updated by a job that runs independently on a virtual machine but could be run as a web service job or just simply run as a scheduler.

This update script just queries all feeds, then goes and downloads the rss data for that feed using feedly [5] ref and then pushes it to the web server for it to distribute out immediately to people (storing it also in that channels history). In this way RSS feeds act like 'bots' and this idea could be extended to actually implement proper bots in a discord fashion [6] that would let us to let users play games such as chess and so on throughout the application, or add custom bots to do extra actions. For example maybe a UNSW Physics channel for a specific research group adds a bot that pushes updates whenever a long running computational experiment finishes a 'frame' (maybe attaching a density plot or something similar). This way they can very easily share updates without email or other more manual procedures.

We run the feed updater script on a lock via the following Linux command

```
* * * * flock -n
    /var/lock/piqued_feed_updater.lock
    python3 /home/feed_updater.py -u
    https://admin.piqued.club >> /home/log.out
```

(note above is just a single line) this enables us to run it consistently (every minute) while making sure we aren't having multiple updates run with conflicting views of what the 'feed' state is.

Pulling updates from the beginning of time every single time is inefficient, so instead we track the 'latest' update time and only download feeds from that point (tracking in SQL). This gives us another benefit too, that when you add a RSS feed to a channel it'll download all messages previously tracked (without having to fetch it from the API) and this is quite fast.

This scales pretty well, since the script is external and effectively is 'pushing' and we only read 20 feed updates at a time. Since we run this every minute, unless a SINGLE feed is pushing more than 20 updates per minute we'll be able to handle it. This means that you won't face slowdown as this scales. Of course sending messages have to be sent to each group individually which will slow as more groups exist but only linearly and messages are batched.

We also have a straightforward change we can make to enable it to scale even further, effectively we would stop pushing it to groups and just have group history updates 'pull' from the rss feed tables as well. Then to handle the issue with having to individually push to each group we could subscribe each user to the RSS feed as well for their groups, this way it becomes just a single multi-cast channel send rather than multiple single cast channel sends (it's more efficient). This would allow us to scale very easily since we can always scale more instances, this is talked more deeply in the deployment section of system design.

1.3 Overall System Design

1.3.1 Recommender Algorithm Design

The realization of Piqued's vision required the creation of a recommendation system. Piqued recommends groups to users based on their interests to ensure that people find like-minded individuals and can discuss their passions. The algorithm utilised is known as a 'content filter recommender system' [7].

Piqued was designed such that each group *and user* is tagged with 'interests'. The list of possible interests was predefined by us, and is more than 1000 elements in length. It was scraped from the internet and ranged from things like "cats" to "ping-pong". These elements formed our 'features' for the content filter, and groups were recommended to users based on how closely their features matched. In a typical system, each feature usually has a degree of relevance, but in Piqued it was binary: either the user/group had the interest, or didn't have the interest. As such, to calculate the embeddings or similarity between groups and users, a dot product was not necessary and a simple 'overlap' determination sufficed.

This methodology ended up working very well, and required minimal computation to find groups that may appeal to users. This component of the recommender simply looks at existing groups and recommends ones that match user interest. The second part, described below, physically creates groups.

1.3.2 Automatic Group Creation System

The more complicated component of the recommender system is its ability to determine popular sets of interests and automatically create groups to encompass them. The algorithm used to achieve this is discussed below.

Every time a user edits the interests in their profile, all possible combinations of their interests are determined, up to a maximum of 3 elements per combination. In other words, assuming the number of user interests to be 'n', the following set 'S' was evaluated:

$$S = \binom{n}{1} \cup \binom{n}{2} \cup \binom{n}{3} \tag{1}$$

The database is consequently queried to determine two facts about each expression in the set 'S'. Firstly, how many other users share the same combination within their list of interests. Secondly if a group exists that has *all* the elements in the expression: no more and no less. If there are enough users who share the interests, and no groups exist to cater to the combination of interests, the group is automatically created.

An example is provided here for clarity. The interests for three users are listed below (where 'interests' in this example are letters).
$User1 = \{A, B, C\}$
$User2 = \{A, B, D\}$
$User3 = \{A, B, E\}$

From these three users, it is clear that the following combinations of interests are popular:
$Popular = \{A\}, \{B\}, \{A, B\}$

If a group doesn't exist for each of the three combinations of interests above, then one is created and recommended to the relevant users as per the previous algorithm.

The group names were created using the NLTK word generator to produce names for the groups. For instance, a group for 'cats' and 'dogs' would have a name generated like: 'fantastic dogs and amazingly cute cats'. The adjectives were automatically generated by the semantic word generator. This aspect of the system produces hilarious and memorable names that will help start a conversation in these automatically generated groups!

1.3.3 Facebook Interest Extraction

It is often difficult to select all of your interests in systems that require it such as Piqued. The large selection Piqued provides and possible lack of self awareness as to what you want to discuss in the application means you typically chose only a small subset of your true interests. Piqued resolves this problem by allowing users to integrate their Facebook accounts.

Through the base-level of permissions required by the Facebook API, Piqued extracts a list of all the pages that the user has liked on Facebook. A proprietary algorithm takes these liked pages and links them up to specific interest categories in Piqued. For example, if a user likes a group called "Cat Loverz", Piqued must assign the user an interest called 'cats'. such a categorization problem is incredibly complex, and Piqued takes a minimalist and computationally simple approach.

A natural language processing algorithm was required to achieve this. A simple Levenstein similarity algorithm was implemented to determine how similar a given interest is with a given group that the user likes [8]. This worked incredibly well, and could easily extract the core user interests simply from the Facebook pages that the user liked. A version of this algorithm was also implemented that used Natural Language Toolkit (NLTK) semantic word trees to actually extract the 'meaning' of words and assign relevant interests [9]. However this became unwieldy and inaccurate. So the simpler (and quicker) levenstein approach was used.

1.3.4 Discover Interests and Groups

As Piqued leverages user interests to help facilitate connections with between different users, an avenue was created for a user to tweak their interests. Whilst users often know what they're looking for, others can branch out using Piqued's "Discover" page. Upon loading the discover page, several GET requests are made, requesting the most popular groups and interests on the platform, amongst all other interests. These endpoints are created in the backend through database queries. Firstly, groups and interests are filtered based on a predicate of having greater than zero users joined. Then interests and groups are returned in a response in descending order of participants. The Discover page, then represents the current user's interests, as well as popular interests as a functional component. Furthermore, the site-wide popular groups are stored in a separate functional component. The user can then toggle between the display of either to manifest and elegant user experience, as users seek to expand their network.

1.3.5 Transcript Upload

Increasingly, automation has become a key factor in ensuring smooth, pain-free user experiences throughout modern digital platforms. Piqued is no different here through the implementation of user transcript upload functionality. When a user registers to Piqued, they have the option to upload their academic transcript in the front-end as a .pdf file. Upon doing so, the file is posted to a back-end route that directs the request and the payload to the transcript viewset. From here the file is opened using the PyPDF2 PDF reader. Regular expression queries were implemented to match specific patterns within the transcript and to capture strings subsequently. These capture groups scrape the the user's enrolled courses and degree program.

After this information is captured from the PDF, the file is closed and the scraped details are returned to the viewset. From here, interests and groups are subsequently queried, and should either not exist, created. Furthermore, course and program objects are also queried for eventual return to the front-end, and likewise, if non-existant, they are swiftly created. After courses, a program, and relevant groups are collated in the back-end, they are serialized into a response, to surface in the front end. These details are then placed into local storage, and retrieved when the user progresses through the registration process.

1.3.6 Notifications

As it becoming increasingly expected by modern users, the messaging functionality of Piqued necessitates an accompanying notification feature. This allows Piqued to become fully integrated into a user's workflow, enabling them to be alerted of new messages promptly respond.

Piqued's notification system is built using Firebase Cloud Messaging (FCM) [10]. FCM is a cross-platform messaging solution that allows messages to be reliably sent from a secure sender to a number of disparate locations; in our use case, from the Piqued server to all of the relevant client devices.

When a user logs in to Piqued, the client application calls a `getToken()` function from the Firebase Javascript API. This accepts a key as a parameter, associating the token with the particular Firebase project set up by the Piqued team. Once the token is generated, uniquely associating the device, a request is made to the backend to store this token in the database, and associate it with the current user. In this way, all logged in users have unique Firebase identification held in the system. For example, if User A has signed in on both their desktop and their mobile device, Piqued would hold a token for each device, both associated with User A.

When a message is sent to a group, the server iterates over each member and multicasts a push notification to each of their associated tokens. In this way, all signed in devices receive an alert informing them of a new message in one of the groups they are members in.

As with any system with notifications, muting is a desired option. Piqued handles this simply by storing a dictionary of muted users and the corresponding duration for each group in the database. When handling notification-sending logic, the server can simply check this dictionary and abstain from notifying any users presently muted.

1.3.7 Password Reset Emails

To handle improving our user experience when they forget their password, Piqued has a page where users can type in their email. This will only send a password reset email provided that the email exists and the email will contain a clickable button that sends the user to a page in Piqued to change their password. A JWT token that expires in 15 minutes will also be passed as a query parameter to that page which allows us to verify that the request to change their password is valid.

For the third party service we decided to use to aid us in sending out our emails, we decided on using SendGrid. SendGrid is a reliable email service provider that has an easy to use API for python and also has a free daily limit of 100 emails which most other providers do not have. It is also very simple to add our own custom dynamic email templates to use and does not complicate our backend system.

1.3.8 Text Shortcuts

To further aid visual communication, Piqued implemented text shortcuts, wherein particular string could be replaced with particular images when sent. This includes both built-in and user-configured shortcuts.

Built-in shortcuts are implemented simply in the front-end. When a message is sent, the string is compared to a number of cases, each associated with an image asset stored in the client application. If there is a match, the message string is removed and the associated image URL is appended to the message object and sent to the back-end.

User-defined shortcuts are slightly more complex. On their profile page, users can attach an image and enter an associated string. When the form it submitted (ie. profile changes are saved), the image is uploaded to Azure Blob Storage and a resource URL is returned. The submitted string is then associated with this URL and stored as a serialised dictionary in the User model. Whenever the user subsequently sends a message, the sent string is compared against all of the user's configured shortcuts. If there is a match, the string is removed and the resource URL is appended to the message object.

1.4 General overview of technologies used

1.4.1 Next.js / React

We decided to use Next.js framework [11] that is built upon React as it was relatively simple to setup a prototype website on our local desktops and quickly start developing the many features stated in our objectives. Next.js provides its own inbuilt features for routing pages on our website which we can specify with our folder structure. This not only makes us conform to a consistent layout of our project for all team members, but also allows everyone to navigate to the desired sections easily.

Deployment: Next.js has multiple ways to be deployed / 'built', the first way is SSR (server side rendered) which is a bit expensive to host and doesn't offer many benefits of a client side approach in our case since our rendering is pretty cheap and is more just memory intensive then CPU anyway.

The second is a hybrid; hybrid could actually be beneficial here for the sake of loading large documents which could be loaded server side (where the data is very close) and then streamed across, however this would slow down the overall 'initial' load of the page which we felt was worse then having all messages load quickly and having assets such as PDFs/images load over time (asynchronously).

If security around these assets becomes an issue, you can always use 'SAS' (shared access signature) tokens to allow limited time access to a private blob container, in our case however since all groups are public this wasn't an issue and we could just let our blobs be public simplifying the design.

Finally, the method we chose was entirely client side (statically served), this has downsides such as not being able to use dynamic routes, however we had no need for dynamic routes in our solution so it was straightforward to convert our routes to not be dynamic and instead use query params in the rare case we needed to pass some actual state.

1.4.2 Authentication Flow

When a user registers or logs in to our website, we provide them with a long-lived (1 day) refresh token and a short-lived (2 hour) access token which are JSON web tokens (JWTs) stored in the local storage of the user's browser. If the user receives an unauthorised or forbidden error from a request, we will attempt to find the refresh token and request for another access token. If that fails, then we will redirect said user to the login page for them to login again.

Every time the user refreshes the access token, we provide them with a new set of refresh and access tokens. Therefore, as long as the user does not cease all activities for 1 day, they will never be logged out.

We decided on using JWTs for our authentication flow because it is stateless, and therefore allows us to scale the application to many users very easily without maintaining information about each user that is logged in.

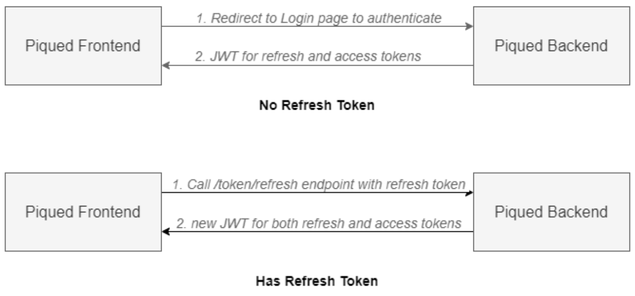

Piqued Frontend
1. Redirect to Login page to authenticate
2. JWT for refresh and access tokens
Piqued Backend
No Refresh Token
Piqued Frontend
1. Call /token/refresh endpoint with refresh token
2. new JWT for both refresh and access tokens
Piqued Backend
Has Refresh Token

Figure 2. Authentication flow with no refresh token vs having refresh token

1.4.3 Responsive Layout of Chat

The responsive layout of our chat system is handled by the inbuilt media queries that Material UI provides us out of the box when we use their components. This allows us to easily specify the sizes of our components according to the screen width of the user's window, making it possible to adjust side panel layouts and the like when their screen sizes get smaller. This is done through the use of 'flex-box' [12] which changes in its width and height to best fill up the available space.

If we wanted to support mobile devices as well we would most likely implement what most applications do, a completely different view for mobile devices which relies on more screens to show information such as group members or groups, for example the mobile design for messenger compared to it's desktop design is shown below.

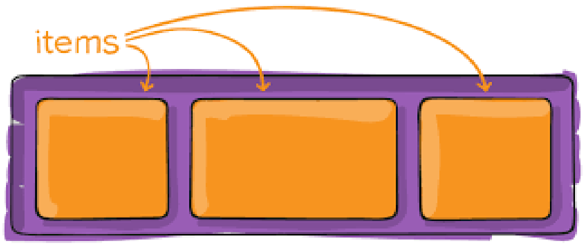


Figure 3. flex box with items in it

1.4.4 Low Latency Chat Messaging

The most important aspect of our system to be fast is messaging, for this reason we do a few unique optimisations.

Unification of websocket: We unify all groups for a user under a single websocket, this simplifies switching groups and notifications of messages in other groups (the red dot), the user then subscribes to each group it's part of as part of the websocket initialisation and a multi-cast channel send is done.

Batching group history on connection: We batch all the group's historic messages into a single websocket send rather than splitting it up into a bunch of individual sends, this enables us to reduce the latency cost that accumulates. The downside of this is that you can't scale this as easily for thousands of thousands of messages **per** group, however changing this is simple and can be done at any point when we actually need to do it rather than just prematurely which is discussed below.

Storing messages based on send time: Messages are stored using a composite key which represents 1) the group they were sent to and 2) the time of the send in nanosecond precision, the fact we have such high precision here means that it's practically impossible that 2 individuals from the **same** group would ever send a message at the **same** time, however if this became an issue the fix would be to simply use a consistency tag (e-tag) and fail if it existed, and just fallback and retry again. By using that time +1 ns, using an extremely small timestep like this is likely to succeed (but we could use a quadratic fallback of course too). A method like this has been successful in enabling scale for various other systems such as Discord [3] and theirs only broke down at 100 million messages.

However, we are a bit unique and the use of table storage will scale MUCH much better, we'll be looking at 1 billion messages at minimum based on my own experience with table storage in handling large quantities of sends (at work we send out very large quantities of surveys tracking through entirely through table storage). In the way we've designed it we ended up being a mixture of how they used to do it and how they currently do it, the use of a wide column store is a novel solution to such a problem, that aligns with how Microsoft

Teams implemented chat [13], they are moving to CosmosDB (1 on 1 chats are already using it), however this isn't due to scaling; rather instead it makes integration with the rest of their systems (particularly exchange) much simpler, indicating we could probably stay on this for a significantly longer period of time than 100 million messages.

The benefit of using the group as the partition key is that it makes it cheap to query all messages for a specific group rather than what would probably be an awfully slow table scan, and if they have both the group and the timestamp (which is effectively the 'message id') they can instantly lookup that message, this gives us a massive benefit which allows us to say give us every message from this group with a row key (timestamp) greater than X date meaning we can for example just query all messages from the first week and as the user scrolls up look further and further into the history, giving us the ability to scale even as groups get very large.

Asynchronous File Uploading: Files are uploaded separately to the message send and once uploaded the message will go through. The benefit of this is that we can upload multiple messages at the same time and it means each message has a significantly smaller amount of data that needs to be stored, we just need to store the message contents and a JSON array of all files (url + type) that are attached to this image, the URL points to our blob storage container but in the future we could even move to a dynamic CDN for certain groups (under some sort of payment model).

2. Implementation Challenges

As with any software architecture there are always shortcomings / flaws in the design, we aim to indicate what said shortcomings are and how they were alleviated.

2.1 Consistent Python Builds

The downside of using Python as a production language is that it's package manager often has to build it's own binaries or requires external dependencies, in our case our use of Microsoft SQL (MSSQL) requires a driver. There are two main options; 1) FreeTDS (Source: [14]), and 2) Microsoft's own provided ODBC driver.

Both of these require installing external packages in the OS, as well as requiring configuration (for FreeTDS) that has to be registered in privileged locations. This is a major downside when attempting to deploy Python conventionally on platforms like Azure Web Services (Source [15]).

However, a nice alternative to this structure is to utilise Docker (Source [16]), which is a nice way to 'containerise' your deployments/builds, effectively every build is it's own complete 'flash' of an OS including all the extra packages and configuration required, we used Linux Ubuntu 18.04. This solves the issues stated prior and guarantees consistent, easy to test and verify builds.

We can use private docker builds to prevent any unauthorized parties from accessing the builds (important since access to any sort of source code would allow reverse engineering of potential security flaws).

A full build from scratch takes around 2000s, however this includes **every** single python package required, as well as the full OS. A build where just code changes are required takes around 100s[1].

2.2 Django Serialization Performance

A series of requests are significantly slower in Django out of the box such as just getting basic user information or querying all groups with most of the extra cost coming down to the cost of Django Serializers which are absurdly expensive in terms of CPU. This would limit our ability to scale.

In reality Python was chosen for it's easy of development not it's performance, and it's likely that as the application scales a more suitable backend implementation such as *C#* may be better.

However, in the meantime we were able to get much faster requests by using a decently powerful server (S2 Web Service [17]) combined with reducing the amount of data our service had to serialize (using 'Simplified' versions of models) this reduced for example the cost of a `self` query (where you get information about the current user) from 2-3s locally to around 200ms on the server which is reasonable.

Due to the fact that we have dockerised the deployment we could very easily horizontally scale here too, and use a load balancer on top to distribute you across the servers, to allow servers to communicate with each other we can use Redis [18] as a channel layer backend.

By separating the frontend from the backend and by simplifying our API we would allow for a rewrite to occur in the future alongside continued development of the python backend without significant changes to the api, given the event we need to use a more suitable backend.

2.3 Notification Consistency

As the notification system lives somewhat adjacent to the main application, extra effort must be made to ensure a certain consistency such that users are not receiving notifications for previously logged in users. This requires correctly maintaining the tokens that live in the database, both in terms of presence and validity. When a user logs out explicitly by pressing the logout button, handling this challenge is trivial. We simply call the aforementioned `getToken()` function, and make a request that searches for this FCM token in the database to have it removed.

A more challenging case is when a user "logs out" from inactivity (expiration of access and refresh tokens). Since the access token is expired (and cannot be refreshed without re-authenticating with username and password) requests to the back-end can no longer be successfully made. As such, FCM tokens cannot be removed from the database in the same way as when the user explicitly logs out. If left unresolved, users

[1]Measured on a Mac Mini M1 using Rosetta 2 for compilation on a x64 environment.

who subsequently log in would receive notifications intended for the previous user. Since requests to the back-end cannot be made, this implies that the handling must take place in the front-end. To achieve this, logic is appended to the axios interceptors [19] that handle all requests and responses. Typically, the response interceptor simply checks whether a 401 or 403 error occurs (indicating the need for re-authentication) and redirects the user to login page. Instead, we additionally include logic that calls `messaging.deleteToken()` from the Firebase Javascript API. This invalidates the associated token from Firebase itself, such that if a message is sent to it, it fails.

In short, when a request is made with an expired token (indicating an implied "logout"), the axios interceptor catches this and removes the FCM token associated with that user and that device. In this way, Piqued ensures notifications only appear for currently logged in users.

2.4 Recommender System Performance

The details of the actual algorithm concept that was designed is presented in section 1.3.1. The original implementation involved a naive brute force approach, wherein each of the combinations is computed and the required parameters is determined for each one (number of users, and number of groups). It is clear however, that this would be an O(n^3) algorithm, where 'n' is the number of interests for the user in question.

Upon implementation and testing, for a user with 10 interests, the algorithm took approximately 30 seconds. This included the communication and response time. A large factor that caused the significant length of time was the number of database queries that were made. A query was made for each combination. Thus if $n = 10$, there are nearly 300 possible combinations, and each query of 0.07 seconds eventually adds up. It was therefore clear that the approach was not viable.

The next step was to determine a way that could allow the same algorithm to be used, but require only a single query (instead of one for each interest combination). Such a query became unwieldy, complex and difficult to understand. As such it was not pursued further, as it would be a bad embodiment of software design practices. An alternative approach was required.

To achieve such a complex request, and entire new database table was devised to enable it. The table simply had five fields: three for each interest that can fit in the combination, one for a group that is tagged by those interests, and one for a user that has those interests. The latter two columns are foreign keys to groups and users respectively. This structure enabled a one-to-many relationship between each possible combination of interests with users and groups. Assuming this table is continuously updated and kept up-to-date, it is possible to quickly and easily determine which interests are popular among many users with a single query.

The user enters a list of interests. All of the rows of the database can be selected wherein the interests in the three columns all exist within the user's selection. All like-for-like interest rows are combined: counting all of the groups that have those interests and all the users that have those interests. If the number of groups is 0 and the number of users is greater than a threshold, then the group is created. This is a single query that speeds up the entire request to just 5 seconds: a nearly 600% performance increase.

The only drawback to this approach is the requirement for keeping the table continuously up-to-date. However, these update queries are not expensive at all. When a group is created, the group must be linked to the row with the relevant interest combinations. This is a cheap query due to the optimisation of the SQL database. The only other update is made when users add or remove interests. This is a fairly expensive query, because all the rows related to the user must first be deleted and then reinserted. However this can happen asynchronously without any performance impact for the user. It can also be achieved in just 3 queries, as opposed to a query for each interest combination. Overall, this innovative approach significantly sped up the process and is a much more effective method of implementing the algorithm.

2.5 Transcript Scraper Front-End Integration

Whilst the back-end development required to enable the users to upload their academic transcript didn't have any point of particular complexity, front-end integration was of particular difficulty due to a variety of circumstances. The transcript upload functionality was to be integrated with pre-existing user registration components in the front-end. This pre-existing functionality contained several React [20] `Autocomplete` components that were used to initialise user details in the back-end. This was a sound decision but as not all members had a solid understanding of the pre-existing functionality, integration issues soon arose. Semantically, the back-end returned a list of courses, a program and a set of groups the user could be added to, at their discretion. To achieve the functionality, these items, returned from the back-end, needed to be pre-populated within the `Autocomplete` components. Not only did these details have to surface in the `Autocomplete` components, but they had to be just as easily removed by the user as to not compromise their experience with a lack of autonomy.

Whilst the solution was simple enough eventually, it was not initially clear to those tasked with its implementation. Finally, through effective teamwork and communication, a solution was devised. The relevant data was used to initialise arrays and objects instantiated with a React `useState()` hook. This yielded new objects and methods to update their states. Thus these newly instantiated objects that contained the data from the back-end were passed through to the `Autocomplete` component, via the `value` property. This aspect of the solution enabled functional pre-population of the relevant data fields. Furthermore, by passing in the previously obtained `setState()` functions, the selected values were then able to be modified, then captured for a subsequent back-end call, where the user's details would be accurate, and to their liking.

3. Functionalities and Project Objectives

This section will outline Piqued's core features, categorising them under the core objectives that were described in the original proposal document.

3.1 Objective 1 - Login and User Management System

Objective 1 was implemented through the following user stories:

- FHP-18
- FHP-50
- FHP-51
- FHP-56
- FHP-62
- FHP-64
- FHP-143
- FHP-144
- FHP-145

A robust authentication and user management system is core to ensuring the delivery of a secure, safe, and enhanced user experience for Piqued's patrons. Wide ranging development across the platform contributed to the actualisation of this objective, providing features such as guarantees of user affiliation to UNSW, an administrative interface, email verification and password management capabilities.

Firstly, a registration system was created to allow new users to engage with the platform, in a meaningful way. As Piqued is a UNSW exclusive product, care was taken to verify that attempted registrations indeed came from UNSW affiliated individuals. This was done through the use of a custom verification tool that mandated a user's registration email, was of a UNSW domain. This feature, prevents the users of Piqued from being exposed to inauthentic users and thus, renders a safer, more positive user experience.

Furthermore, an authentication system with the capability to reset passwords was developed. To access any communication functionality within Piqued, the user must either register, as above, authenicate themselves as an already registered user. This is done through the implementation of a login form, allowing users to submit their username and password to be authenticated. To provide a more flexible experience Piqued also offers infrastructure for users to change their password. Thus, through the creation of the appropriate front-end interfaces, and server-side API endpoints, the users of Piqued can change their password as they please. This is a critical feature for both security and user experience, as in today's ever-changing security landscape, flexibility in changing passwords is key to maintaining a secure user account.

Piqued provides an avenue for users to express themselves to an extent at which they are comfortable. As Piqued a key user experience of piqued is for users to expand their social, academic and professional networks, infrastructure was implemented to allow users to share details about themselves, for others to see. Key aspects of this include user profile pages and user detail submission forms. The user profile page was created through the development of front-end components that enabled functionality such as setting a new profile picture, including both image scale and relative placement, resetting name details, and adjusting date of birth. This serves to create an immersive user experience as users connect with each other, throughout Piqued.

Furthermore, upon successful registration, a user would be sent a greeting email. This functionality was developed through the use of SendGrid. This welcome email served to both, establish an inclusive environment for the user, but also provides a precedent as to how Piqued will communicate with users, as outlined above, for password resetting.

Piqued aims to provide administrators with the capability to maintain an inclusive and safe platform. The development team implemented tools that enable administrators to modify groups, interests and the user-base itself, within Piqued. These tools were implemented using the Django administrator interface, which allowed reading of database model metadata in an efficient fashion, enabling querying and modification data. Naturally, users must have administrator privileges in order access this interface to prevent erroneous or malicious modification of data.

3.2 Objective 2 - Group Formation

Objective 2 was implemented through the following user stories:

- FHP-48
- FHP-99
- FHP-100
- FHP-102
- FHP-103
- FHP-104
- FHP-105

The key principle underscoring Piqued's design is the ability for users to interact with people that share interests and circumstances. To achieve this, Piqued functions with a group-based system whereby users are able to join groups that reflect a combination of their interests. Unlike other platforms, users are not permitted to interact with others outside of these forums through mediums such as one-to-one direct messages. This mandate creates a platform that eases difficulties associated with moderation, by reducing avenues of communication.

However, it also places further emphasis on Piqued's foundational goal of augmenting group communication by encouraging interaction between individuals, whom would seldom cross paths otherwise. These groups can be created to provide users an avenue to discuss passions, hobbies, pursuits and interests, but also to facilitate academic discussions surrounding enrolled courses of users. Through this inclusion, Piqued aims to enable discourse for all course participants, and provide peer-to-peer support through constructive discussion of coursework.

Groups intended for courses are automatically created, and users can elect to be included through either uploading their academic transcript, or manually electing to join a given course. This serves to break down barriers to entry for all students; democratising academia outside of the classroom. Furthermore, these groups will expire when the term offering of that subject elapses. Whilst remaining archived, these groups will not be surfaced to the user's front-end, thus, automating removal of clutter from the user's home page, ensuring a seamless and intuitive user experience. To complement this feature further, Piqued includes limits on creation of groups by users. Through relevant back-end queries and dynamic front-end features, users are prevented from creating more than 3 groups, each. This serves to preserve the lean, sleek nature of Piqued, by stemming avenues for spam and clutter, within both the front-end and back-end.

As previously mentioned, accessibility to Groups is pivotal to fulfilling the vision with which Piqued was created. To aid this goal, Piqued includes a discovery feature where amongst other things, users can see the most popular groups across the platform, based upon membership. This is done through the display of popular items in descending order, by the front-end. Users can then opt to join a popular group as desired. By providing this popular group interface, users can broaden their perspectives on different communities within platform in a holistic manner.

3.3 Objective 3 - User Interest Submission

Objective 3 was implemented through the following user stories:

- FHP-96
- FHP-97
- FHP-98

User interests are an integral part of Piqued's unique platform and serve to aid in personalising both profiles and groups to users. The developers of Piqued curated a universal static list of interests, incorporating a multitude of hobbies, games, sports and media, among topics. Users are encouraged to select interests that represent their hobbies and passions, such that Piqued can deliver a considered, personalised user experience. This is achieved through a variety of avenues within the platform. Initially, upon sign-up, users have the option to log into Piqued using Facebook. This functionality pre-populates profile details such as name and date of birth, but also extracts the users "liked" Facebook pages. These extracted "likes" are parsed by a natural language processing algorithm, to determine any relation to previously created interests on Piqued. Upon progressing through the registration, the user can then choose to confirm these relevant interests as accurate or, if not, remove them. This feature serves to streamline the personalisation process of a profile, when signing up to the platform. This ultimately aids to address Piqued's aims to distinguish the platform from other social networks by the removal of such pain-points and inefficiencies.

Interests can be further configured after sign up through the Piqued "Discover" interface. This interface aims to help users explore the platform further by offering popular interest and personal interest modification interfaces. The user is displayed a list of popular interests based on the amount of users that share that interest, in a descending order. From here users can then add interests to their profile and remove them as they please. This allows Piqued to deliver a dynamic user experience that shifts with the personalities, passions and hobbies of users. Hence, this functionality addresses the goal of providing an immersive interest based system, to facilitate group interaction.

3.4 Objective 4 - Transcript Upload

Objective 4 was implemented through the following user stories:

- FHP-45
- FHP-47

As stated above, many social media networks lack a focus on academia, a cornerstone of Piqued's platform. Furthermore, when attempting to facilitate personalisation of profiles by users, other products offer little avenues for automation, leading to a cumbersome user experience. Piqued addresses the above pain-points by implementing an avenue for users to upload their academic transcript. Upon registration, this functionality allows users to select a transcript from their computer, in the front-end, before uploading it to the Piqued back-end.

Once posted, the users relevant academic details are scraped from the transcript. The scraper implemented within Piqued focuses upon the current trimester, and extracts the student's enrolled courses and degree program. These are returned back to the front end where the user can opt in to the addition of these courses and program to their profile. Furthermore, the user will also be added to the groups relating to these courses and programs automatically. This serves to greatly reduce the time-investment users will make when joining Piqued, and promote further academic inclusion.

3.5 Objective 5 - Interest & Group Oriented

Functionality surrounding Objective 5 was developed through the following user stories:

- FHP-23
- FHP-52

As discussed, Piqued uses a list of predefined interests to categorize groups and users. This process helps users identify like-minded individuals, as well as find groups that matches their needs. Piqued further assists this process by recommending groups to users based on a variety of factors. When users enter the 'group discovery' page of the application, they are presented with a recommendation list full of groups they are likely to enjoy. The heuristics used to generate this list are presented below.

Firstly, the top groups in Piqued are presented to the user. These are simply the groups with the most number of participants and typically encompass the most generic and common interests: such as 'cats' and 'dogs'. The goal of this aspect of the recommender is to encourage users to see what groups exist on the platform, and join the wide and lively Piqued community.

Secondly, the Piqued system actually creates groups that it determines are of interest to a sufficient number of people. The way this works is described in detail in Section 1.3.1, and briefly here for clarity. When a user registers their interests, they are processed into all possible combinations (with a maximum of 3 simultaneous interests). Each of these combinations is compared against all other users in the system to determine how popular the combination is. If the combination is shared by a significant number of other users, a group is automatically created and tagged with the specific collection of interests. Such a group is then considered a 'perfect match' for the relevant users and shoots to the top of their recommended list. The advantage of this system is that when a collection of interests becomes popular enough, users that share them can join a group and meet each other: sparking off a discussion/friendship based upon a well defined common ground.

3.6 Objective 6 - Messaging Others in a Group

This functionality was developed in tandem with these user stories:

- FHP-16
- FHP-20
- FHP-53
- FHP-101
- FHP-106
- FHP-107

Messaging is perhaps the core functionality on which Piqued is built. In driving toward Piqued's mission - fostering reverberation chambers founded on common interests - the ability to communicate to others via instant messaging is the cornerstone for enabling and facilitating lively and fruitful interactions.

Fundamentally, Piqued implements a reasonably standard but elegant and robust underlying message service. Utilising Django Channels, users are connected to websockets that are associated to each group they are in. When a message is sent to a particular group, the server handles delivering this message along the websockets associated with each member. This general architecture forms the basis of the messaging service, allowing users to send and asynchronously receive messages they are interested in.

At this point, a bare-bones messaging service would be complete. However, modern users have rightly become accustomed to messaging "adornments" such as whether a message has been "Seen", whether the users they are talking to are "Online", as well as the ability to update and delete sent messages. Piqued implements each of these adorning features by building on top of the same underlying architecture used for the primary messaging functionality. When a message is received by a client, a "Seen" message is sent back along the websocket to the server. This is once again propagated to the members of the relevant group. Once received by other users, the respective front-ends are able to update their UI to clearly display which members have seen the message. Similarly, when a user logs in, a status message can be sent to the websocket and forwarded to all interested users allowing them to see which users are "Online" and which are not. Finally, update and deletion of an existing message is handled in much the same fashion, with a dedicated "Update" message being propagated to members of a group, informing them of the details of a modified message. The front-ends can handle this information accordingly and update or delete the message displayed to the user.

3.7 Objective 7 - Uploading and Viewing Media

The following user stories were used in addressing this objective's functionality:

- FHP-36
- FHP-37
- FHP-38
- FHP-41
- FHP-42
- FHP-43
- FHP-44
- FHP-55
- FHP-137
- FHP-138

Visual media is becoming an increasingly important form of communication as students seek to share images, videos, pdfs and other content that carries information that cannot be conveyed solely through text. This may include assignment specifications, useful learning resources, or even helpful code snippets.

While file sending has become something of a staple in many messaging applications, a common pain point is the inability to preview received material. Users are subsequently required to downloaded files in order to inspect them, a tedious inconvenience which compounds as more files are exchanged. Not only does this incur a time cost to the user, but it also consumes space on the user's device with many files unnecessarily downloaded simply to inspect the received contents. Furthermore, the inability to preview content lends to greater difficulty in finding previously sent/received files, the absence of a visual aspect meaning that recall is based solely on a potentially obscure filename.

As such, Piqued has placed emphasis on implementing file-sending with accompanying previews, while retaining the usual download capability. This functionality is largely facilitated by Azure Blob Storage. When files are sent to a group, they are not sent byte-by-byte along the websockets to each user. Rather, when a message contains a file, the client first sends an HTTP request to the server with the file object attached. From this endpoint, the file is uploaded to Azure Blob Storage and a URL to the asset is returned to the client. As part of the message object, the URLs of attached files are appended and the message object is sent and propagated as normal. When a client receives a sent message, the files field containing the asset URLs are inspected. Here, the front-end renders the files according to its type, seamlessly displaying it to the user as part of the chat. Piqued accounts for numerous file-types including images, videos, pdfs, word documents and code snippets.

3.8 Objective 8 - RSS Feed Integration

Objective 8 was implemented through:

- FHP-34

RSS feeds are more than just an archaic standard, they are still used to this day by millions. In our case they function as a way of people to link up news and blog sites, as well as other things such as weather alerts and anything that is an RSS feed.

The way they are implemented is two fold, firstly we have a query system to allow us to grab relevant feeds based on a search parameter, we then utilise tags to give users an idea of what they are looking at; is it a funny blog about cats? what about a very serious news channel? or maybe just a parody comic of modern day politics.

They'll get notifications when new alerts come in of course as well, and the system runs at a regular interval meaning we very quickly send out alerts after a post is created.

Users can add/remove feeds at any time, this may be somewhat subjective to abuse, and in the long term a way to prevent this (maybe a voting system or other sort of mechanism) could exist.

But the concept isn't necessarily just to stop at RSS feeds but as discussed prior, we could extend it towards other bots. It's a novel idea that is complete in it's goals, to provide an automated message endpoint. But it could develop into a much more critical system, this could include bots to play games, receive experiment updates, push assignments/tutorial materials and so much more.

3.9 Other Features

The functionality outlined in 3.1 to 3.8 satisfies the core objectives set out at the proposal of this project. However, numerous features were added atop this primary functionality, to streamline, enhance and extend the user's experience, all focused on better serving Piqued's mission - to facilitate reverberation chambers built on common interests among students.

In further striving to create an elegant and robust messaging service that underpins the system, Piqued offers both notifications and muting capabilities. Through Firebase Cloud Messaging (FCM), user devices are associated with an FCM token when they sign in. This token is associated with the user and stored in the database. When a message a sent to a group that the user belongs to, the server iterates over all associated FCM tokens and sends a push notification. Thus, if the Piqued application is in the background, users will be alerted to a new message and can respond quickly if desired. Notification functionality necessitates the option of muting, lest the continuous alerts become an irritation or distraction to users. Thus, Piqued gives users the ability to mute particular channels for either an hour or indefinitely. This information is once again held in the database, and the server will scan for these muted users and abstain from sending them notifications until the specified time period has expired or the user has explicitly unmuted.

Continuing to develop the intuitive way Piqued handles visual media, text shortcuts have also been introduced. That is, certain text phrases are associated with a visual element (typically an image or gif) thus allowing them to be sent effortlessly and without the need to search. Piqued comes with come built-in shortcuts including the names of our founders ('/name') which are automatically translated to their respective image. Similarly, sending the message '/piqued' will send an image of the Piqued logo. However, pre-built shortcuts are limited in their utility. As such, Piqued offers users the ability to define their own shortcuts on their profile page. Here they can specify any text-string and an associated image and have this personalised shortcut available for use.

4. User Manual

4.1 Running Locally for Development

Running this locally has a few steps, and depends massively on how you want to run it, i.e. do you want to use a MSSQL server? Do you want to use another kind of SQL server (i.e. MySQL, Oracle, and so on). Do you want to simulate it via sqlite and so on.

Let's begin with the easiest steps first, we are going to set it up so that we have a locally running front end which is connected to our back end. This way we ensure that your front-end is all working fine before we run a local back-end.

1. Git Clone the repository or download the zip

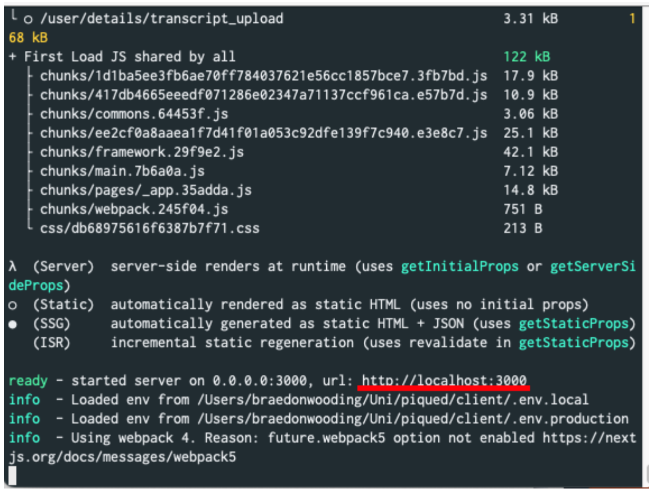
2. Make sure you have `npm` installed, you can install it by following the given instructions here `https://www.npmjs.com/get-npm`

3. `cd` into the client directory and run the following command (this will build the client and run it against a production back-end).

Unix: `npm install && npm run prod`

Windows: `npm install` followed by `npm run prod`

4. Verify the app is working by going to the address `http://localhost:3000`, note it'll state this address as one of the final lines of output as shown in the image below [Figure 4.] so if you have other stuff running it may be running on a different port.



```
└ ○ /user/details/transcript_upload                       3.31 kB        1
68 kB
+ First Load JS shared by all                             122 kB
  ├ chunks/1d1ba5ee3fb6ae70ff784037621e56cc1857bce7.3fb7bd.js  17.9 kB
  ├ chunks/417db4665eeedf071286e02347a71137ccf961ca.e57b7d.js  10.9 kB
  ├ chunks/commons.64453f.js                              3.06 kB
  ├ chunks/ee2cf0a8aaea1f7d41f01a053c92dfe139f7c940.e3e8c7.js  25.1 kB
  ├ chunks/framework.29f9e2.js                            42.1 kB
  ├ chunks/main.7b6a0a.js                                 7.12 kB
  ├ chunks/pages/_app.35adda.js                           14.8 kB
  ├ chunks/webpack.245f04.js                              751 B
  └ css/db68975616f6387b7f71.css                          213 B

λ  (Server)  server-side renders at runtime (uses getInitialProps or getServerSi
deProps)
○  (Static)  automatically rendered as static HTML (uses no initial props)
●  (SSG)     automatically generated as static HTML + JSON (uses getStaticProps)
   (ISR)     incremental static regeneration (uses revalidate in getStaticProps)

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
info  - Loaded env from /Users/braedonwooding/Uni/piqued/client/.env.local
info  - Loaded env from /Users/braedonwooding/Uni/piqued/client/.env.production
info  - Using webpack 4. Reason: future.webpack5 option not enabled https://next
js.org/docs/messages/webpack5
```

Figure 4. npm install && npm run prod

Pat yourself on the back this is half the setup to get it fully running locally. Feel free to play around with the application it should work fully do note however that you are using a production back-end. Normally this wouldn't be recommended and you would add CORS restrictions to prevent this from happening (or require a cert or some app key) however, for an application intended for a uni project it simplifies installation.

Next step will be getting a back-end running!

1. Open up a new terminal and go to the server directory (not the client one). I highly recommend a new terminal here.

2. Make sure you have `python3` installed, while we technically only support python3.8 and above lower versions **should** work (you need ATLEAST python3.4) but if you are having issues I recommend installing python 3.8 or above as per these instructions `https://www.python.org/downloads/`.

3. Setup a python virtual environment via the commands `python3 -m venv .venv` (make sure you are in the server folder)

4. Activate the virtual environment

Windows: `.venv\Scripts\activate.bat`

UNIX: `source .venv/bin/activate`

5. Install all requirements for running the backend this is as simple as running (this can take quite a while, especially on rarer platforms)

```
python3 -m pip install -r requirements.txt
```

6. Run the back-end via the command

UNIX:

```
    ENV_USE_SQLITE=1
      python3 manage.py runserver
```

WIN:

```
    $env:ENV_USE_SQLITE=1;
      python3 manage.py runserver
```

note ANYTIME you want to run this you have to make sure you've activated the virtual environment. We'll begin by running it using sqlite and a DB we've already prepped with some nice beginning data.

7. Once that's succeeded you just need to run the front-end in local mode which is as simple as running

```
npm run dev
```

note: you need to run this from the client app and should use the other terminal (both have to be running).

8. Verify the app is working by going to the address `http://localhost:3000`, note it'll state this address as one of the final lines of output as shown in the image below [Figure 4.] so if you have other stuff running it may be running on a different port. The admin page will be at `http://localhost:8000` (it'll state it after running python run server).

This however, won't cause rss feeds to update. To have this happen you'll have to run the rss feed updater manually.

```
python3 feed_updater.py -u http://localhost:8000
```

We typically have a job setup on a VM to run this every minute (as discussed in detail in the overview).

Running the non sqlite isn't recommended due to requiring installation of ODBC driver, `https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-serv view=sql-server-ver15`, however once installed just run the same command to start backend but use `ENV_USE_SQLITE=0` rather than 1.

However, note that you can ONLY run this if using the fully isolated sqlite setup, if you want to use SQL Server you'll have to disable the job that is setup on an Azure VM, the certificate is supplied in the repository (under server/PiquedVM_key.cer) and connecting to the VM is as simple as running

```
ssh -i <path to cert> superadmin@20.188.209.62
```

Then you'll have to just disable the cronjob by running

```
sudo crontab -e
```

And then by adding a infront of the first line (the only cron job)

Note: there are many other ways to run/deploy this, such as the docker file in the main repository (under server). This can just be run as is and will act as a back-end (you'll have to build the image of course but is fully self enclosed).

4.2 How to use Piqued

The below sections will outline how to use Piqued, and all of the features that are available to its users.

4.2.1 Login

When first accessing Piqued, the user will be presented with a login page, as shown in Figure 5.

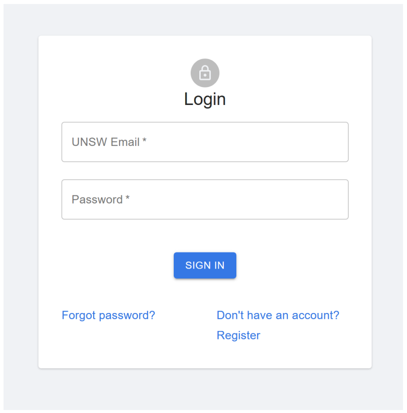


Figure 5. Login Page

If the user has previously registered, or has an existing account, they are able to enter the email and password here. Clicking 'Sign In' will then redirect the user to the main chat window.

If a user forgets their password, the 'Forgot Password?' button can be pressed. As depicted in figure 6, the user will be prompted with an email box. If an account is linked with the email, a password reset link will be sent to them. This will enable them to create a new password.

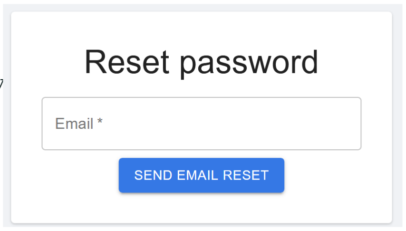


Figure 6. Forgot Password Page

4.2.2 Register

On the login page, if the user would like to register, they are able to select the 'don't have an account' button. This takes them to the register page depicted in Figure 7.

The main novel feature of this page is highlighted by label number 1 in Figure 7. This is an integration between the Piqued application and the Facebook authentication API. Clicking on this button allows the user to log into their facebook account through piqued. Piqued will subsequently extract the core user details (like name, age and email) and autofill the register page - simplifying the user experience.

Furthermore, Piqued will automatically extract the user's interests from their facebook profile. This information will be passed through the algorithm discussed in Section 1.3.1 to create the unique functionality and experience the Piqued prides itself on. As a result, registering with Facebook is highly recommended.

Once all the user information is entered, the user can press the button annotated by label number 2 in Figure 7 to sign up. This will then redirect the user to 'upload transcript' page.

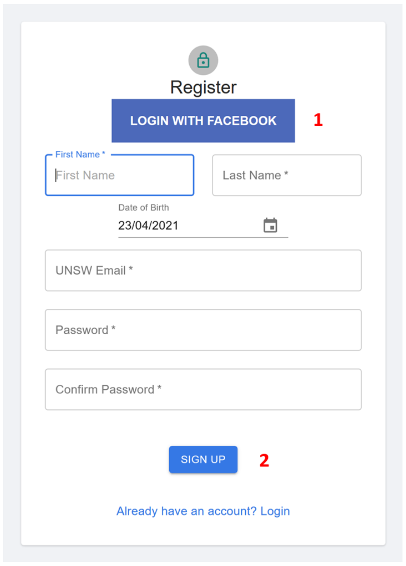


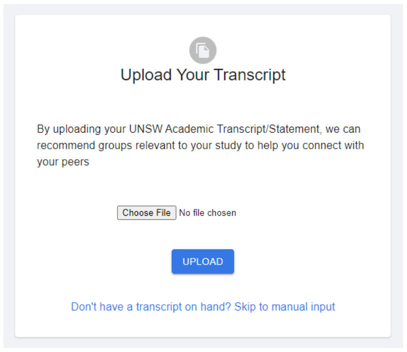
Figure 7. Register Page

4.2.3 Upload Transcript

After entering the basic registration details, the user has the option to upload their transcript into piqued. Piqued categorises groups by both interests and UNSW courses in an attempt to get like-minded people into groups to chat with one another. The upload transcript feature allows the user to side-step the hassle of remembering and typing in all of their courses. Instead, Piqued can automatically extract pertinent information directly from the transcript. Pressing the 'upload' button in Figure 8 will open the computer's file system and allow the user to select the transcript.

Note that the only data collected from the transcript is the course name and program name. Information like course grades, WAM and academic standing are not extracted for privacy reasons. Despite this assurance, Piqued is aware that some customers will be uncomfortable with such a functionality. As such, there is a button underneath the ’UPLOAD’ button that allows users to directly skip to manual input without uploading a transcript.

Next, the user is redirected to the interest and course selection screen.



Upload Your Transcript

By uploading your UNSW Academic Transcript/Statement, we can recommend groups relevant to your study to help you connect with your peers

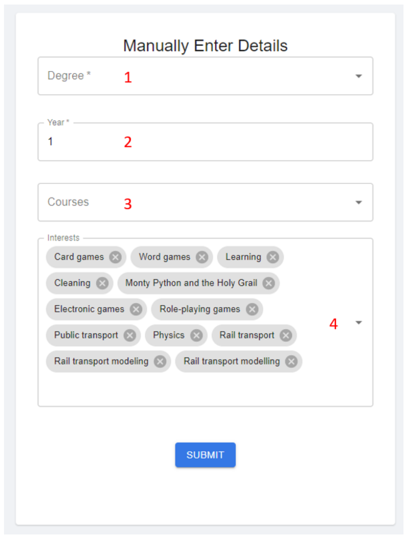
Figure 8. Transcript Upload Page

4.2.4 Interest and Course Selection

Figure 9 outlines this core screen. Each area of the page is labelled from 1 to 4, and described below:

1. Degree: This section is a drop down list of all Degrees at UNSW. They are automatically scraped from the UNSW websites, and the user simply has to choose the degree that matches their program.
2. Year: The user is required to enter what year of university they are in.
3. Courses: The user is encouraged to select all of the courses they are currently attending. This will allow Piqued to recommend groups that match the user’s needs and interests. Note that this field, and the previous two, are automatically completed for the user if they uploaded their transcript on the previous screen. Even if they are prefilled, the user is able to edit them in any way that is wished
4. Interests: The user is encouraged to select all of their interests from our list of 1000 carefully created categories. Ranging from interests like ”Football” and ”Physics”, to movies and videogames like ”Minecraft” and ”Lord of the Rings”, our interest categories encompass a large variety of hobbies and passions. The more the user selects, the better recommendations and user experience the user will obtain. Note that in the example presented in Figure 9, the interests have been pre-filled by the Facebook login feature described in a previous section. For a visual depiction of how the interests are selected, view Figure 11.

Once all of the information has been entered, the user is redirected to the main ’chat’ application window.



Manually Enter Details

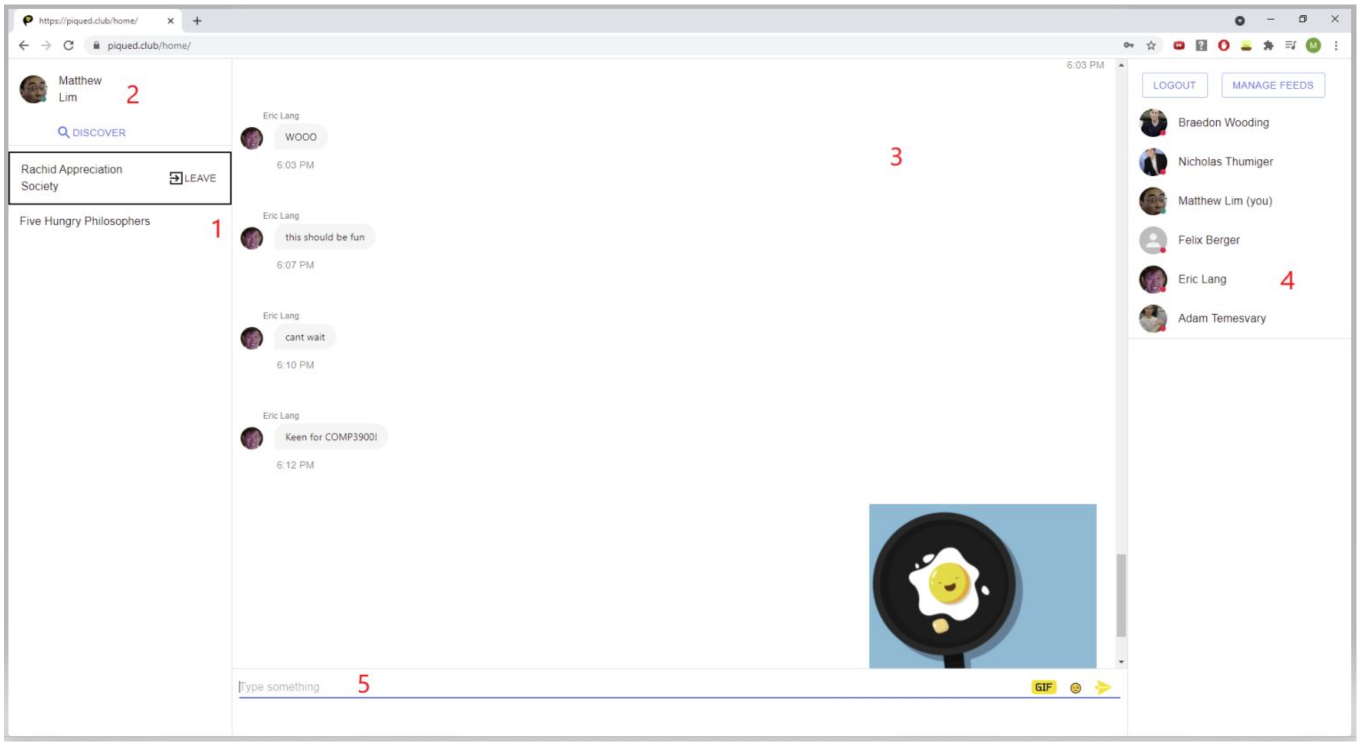
Degree *

Year *: 1

Courses

Interests: Card games, Word games, Learning, Cleaning, Monty Python and the Holy Grail, Electronic games, Role-playing games, Public transport, Physics, Rail transport, Rail transport modeling, Rail transport modelling

Figure 9. Interest and Course Selection Page



Eric Lang
WOOO
6:03 PM

Eric Lang
this should be fun
6:07 PM

Eric Lang
cant wait
6:10 PM

Eric Lang
Keen for COMP3900!
6:12 PM

Figure 10. Main chat screen with annotated sections

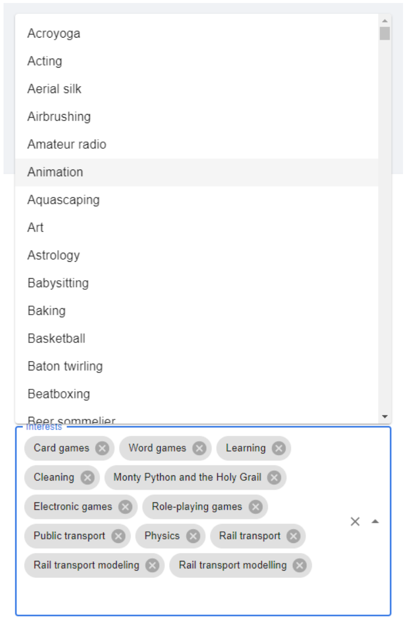


Figure 11. Selecting Interests

4.2.5 The Main Application Screen

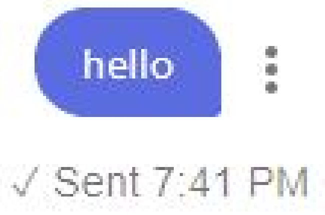
Figure 10 displays the main user interface for a user of Piqued. The red numbers from 1-5 denote 5 major areas of the user interface. The categorisation of these ”areas” are for pedagogical purposes, aiming to clarify navigating the interface for a reader of this guide. These sections are explained below.

1. This is the list of groups that the user is currently in. In Figure 10, it can seen that the user is a member of ”Rachid Appreciation Society” and ”Five Hungry Philosophers”. These tabs can be clicked to navigate between groups. Above the group list is the ”Discover” button. Clicking this button will navigate the user to a new page wherein they can search and discover new groups to join or create.
2. This section can be thought of as the user section. It consists of a single clickable element which displays the current user’s name and profile picture. In Figure 10, the user’s name is ”Matthew Lim”. Clicking on this user will navigate to the main user page which will be discussed later.
3. Section 3 represents that main chat area where messages appear. Messages sent from other users appear on the left while messages sent by the current user appear on the right. New messages appear at the bottom of the scrollable element as is expected of typical messaging systems. Figure 10 shows a number of messages sent by members of the group, and a cute egg image sent by the user.
4. This section lists the current members of the selected group. Each member is represented by their name and their profile picture. A small coloured circle at the bottom right of the profile picture indicates each user’s status. Red indicates that the user is offline while green indicates online. At the top of this section are two buttons, ”Logout” and ”Manage RSS”. ”Logout” simply logs out the current user. ”Manage RSS” takes the user to a new page to manage their RSS feeds, a topic that will be further discussed later.
5. Section 5 represents the chat box where messages can be entered.

Figure 10 and the corresponding numbered areas will be referred to for the remainder of the document.

4.2.6 Sending Messages

Area 5 in Figure 10 represents the user interface for sending messages. The user can type in this section and press enter to send the message. Alternatively, the yellow arrow icon of the right of the chat box can be clicked to send the message. A chat message is depicted in Figure 12 below.


hello

✓ Sent 7:41 PM

Figure 12. Blue message indicating it was sent by the current user

When a chat message is sent, it is immediately sent to all other people in the group. The user can tell that they sent the message due to the colour of the bubble: a dark blue. The user can also see the status of the message by looking at the description directly underneath it. If a single tick mark is seen, as is the case in Figure 12, it indicates that the message has been sent, but has not yet been seen by *everyone* in the group. The description next to the tick will also indicate who has seen the message.

In the chat box, in Area 5 of figure 10, next to this send button are two icons, one for sending emojis and one for sending gifs. Clicking the GIF icon opens a selector where Gifs can be clicked on an immediately sent to the chat. As shown in Figure 14, the user can search for specific gifs. The icon adjacent to this is the emoji selector which works in a similar fashion. When an emoji is selected, it is added to the text inside the chat box.

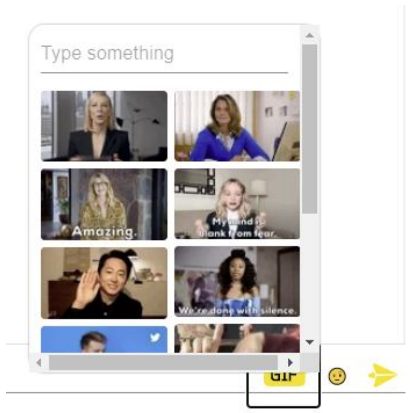


Figure 13. Clicking on the GIF icon opens the Gif selector

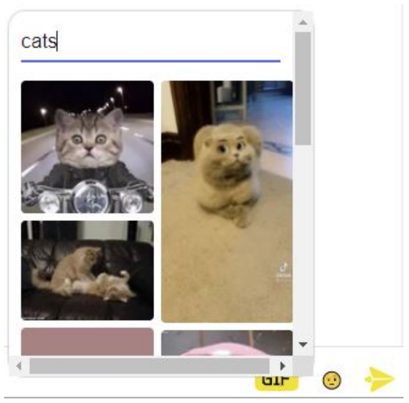


Figure 14. User can search for specific kinds of Gifs

4.2.7 Uploading Files

As well as sending messages, emojis and gifs, Piqued allows users to send files of any type, with many of them having supported renderers such that they can be viewed in the browser. The simplest way to send a file is to attach it to a message by dragging the file into the chat box. After dragging one or multiple files, their names will appear below the text box as shown in Figure 16

When these are sent, Piqued is able to seamless render them inside the chat, allowing users to interact with them in file-specific

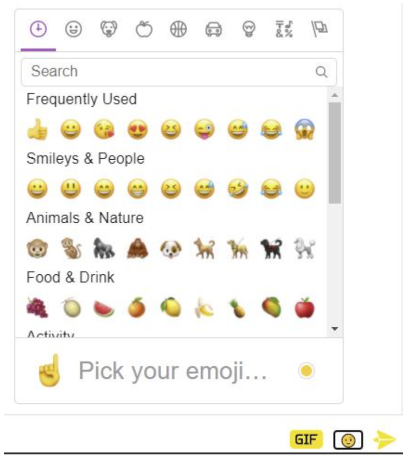


Figure 15. Clicking on an emoji adds it to the text box

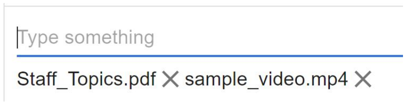


Figure 16. Files can be dragged and dropped into the chat box and sent alongside messages

ways (viewing different pages of the PDF, playing and pausing the videos and so forth).

4.2.8 Editing and Deleting messages

Piqued gives users the ability to edit and delete messages they sent. Clicking on the 3-dot icon to the left of a message allows users to either "Edit" or "Delete" the message. Clicking "Edit" will display the modal displayed in Figure 18.

4.2.9 User Profile

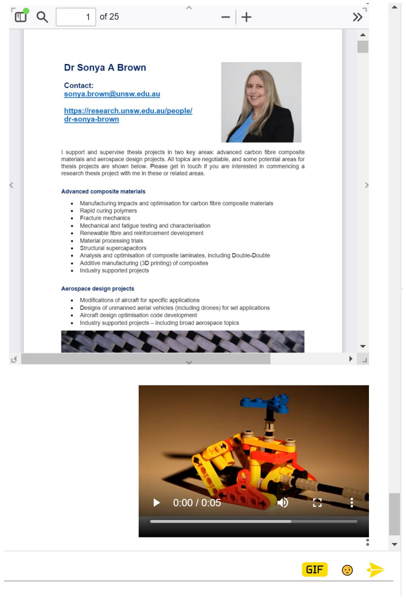
Clicking on the user element (area 2 from Figure 10) will take the user to a page where they can modify their profile. This is displayed in Figure 19.

On this page, the user can modify their first name, last name, and date of birth. They can also configure text shortcuts they wish to use. Figure 19 shows the user having the text shortcut "/pizza". This means that if a user types "/pizza" in the chat and sends it, the corresponding image of a pizza will be sent instead. Below this is an empty field with no text and no image. Clicking the "Add shortcut" button will append another blank field to the list. These empty fields can be used to configure new text shortcuts, while the "Delete" button on the right will remove them. Clicking the "Save" button will save all changes made to the profile.

4.2.10 Text Shortcuts

The above section has outlined how to configure text shortcuts, but built-in shortcuts exists as well. They are:

- /adam - sends an image of founder Adam Temesvary
- /braedon - sends an image of founder Braedon Wooding
- /jimmy - sends an image of founder Jimmy Chen
- /matthew - sends an image of founder Matthew Lim
- /nicholas - sends an image of founder Nicholas Thumiger
- /piqued - sends an image of the Piqued logo



Dr Sonya A Brown

Contact:
sonya.brown@unsw.edu.au

https://research.unsw.edu.au/people/dr-sonya-brown

I support and supervise thesis projects in two key areas: advanced carbon fibre composite materials and aerospace design projects. All topics are negotiable, and some potential areas for thesis projects are shown below. Please get in touch if you are interested in commencing a research thesis project with me in these or related areas.

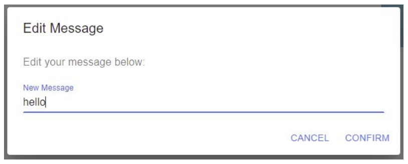
Advanced composite materials

- Manufacturing impacts and optimisation for carbon fibre composite materials
- Rapid curing polymers
- Fracture mechanics
- Mechanical and fatigue testing and characterisation
- Renewable fibre and reinforcement development
- Material processing trials
- Structural supercapacitors
- Analysis and optimisation of composite laminates, including Double-Double
- Additive manufacturing (3D printing) of composites
- Industry supported projects

Aerospace design projects

- Modifications of aircraft for specific applications
- Designs of unmanned aerial vehicles (including drones) for set applications
- Aircraft design optimisation code development
- Industry supported projects – including broad aerospace topics

Figure 17. Piqued can render PDFs, videos, code and more in-browser



Edit Message

Edit your message below:

New Message
hello

CANCEL CONFIRM

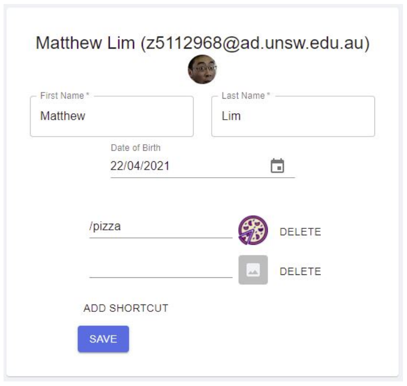
Figure 18. Replace text to edit the message

Additionally, sending the string "/gif-*" (where * can be replaced with any string), will send the top rated Gif matching that search criteria.

4.2.11 Leaving and Muting Groups

For the currently selected group, the option to leave is displayed by the "Leave" button. Clicking this will remove the user from that group and it will no longer be listed on the left of the page. Note that the button to leave only appears for the currently selected group, so in order to leave a group, the user must first select the group and check whether there are any final important messages.

When hovering over any group (whether selected or not), the option to mute becomes available. Clicking the mute button shown in Figure 20 brings up the modal shown in Figure 21. The user can select to mute the group for an hour, indefinitely, or unmute it.


Matthew Lim (z5112968@ad.unsw.edu.au)

First Name *	Matthew
Last Name *	Lim
Date of Birth	22/04/2021

/pizza DELETE

DELETE

ADD SHORTCUT

SAVE

Figure 19. Profile Page

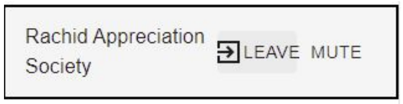


Figure 20. Hovering over a group

4.2.12 Discover Groups and Interest

At the bottom of area 2 in Figure 10, the Discover button can be found where user's can navigate, to join groups and further add interests to their profile. By clicking this link the user will be taken to the discover page, shown in Figure 22 where, popular and recommended groups will be displayed. Users can then choose to join popular or recommended groups. Scrolling down further with groups active, the user can then either search for a group or navigate to the group creation page.

The user can then toggle to the interests page by clicking on the interest button as shown in Figure 22 . By toggling this button, the user will be presented with site-wide popular interests and the user's preselected interests. Interests can then be submitted and removed from this screen. Clicking the "X" in the top right corner of the discover page, will navigate the user back to the Main chat screen, as can be observed in 23.

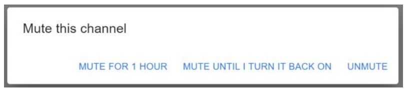
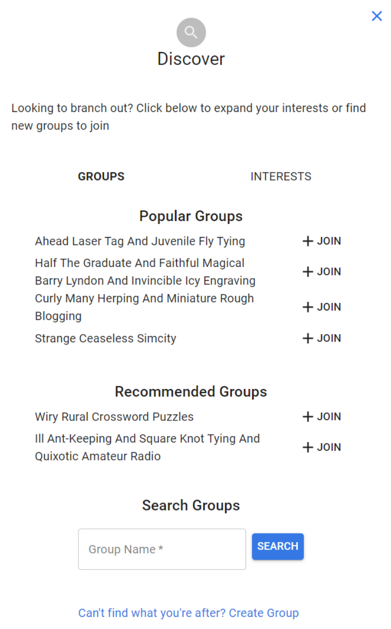


Figure 21. Mute Options


Discover

Looking to branch out? Click below to expand your interests or find new groups to join

GROUPS INTERESTS

Popular Groups

Ahead Laser Tag And Juvenile Fly Tying	+ JOIN
Half The Graduate And Faithful Magical Barry Lyndon And Invincible Icy Engraving	+ JOIN
Curly Many Herping And Miniature Rough Blogging	+ JOIN
Strange Ceaseless Simcity	+ JOIN

Recommended Groups

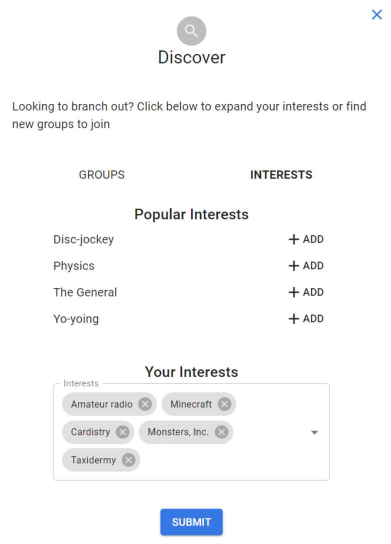
Wiry Rural Crossword Puzzles	+ JOIN
Ill Ant-Keeping And Square Knot Tying And Quixotic Amateur Radio	+ JOIN

Search Groups

Group Name * SEARCH

Can't find what you're after? Create Group

Figure 22. Discover Groups page


Discover

Looking to branch out? Click below to expand your interests or find new groups to join

GROUPS INTERESTS

Popular Interests

Disc-jockey	+ ADD
Physics	+ ADD
The General	+ ADD
Yo-yoing	+ ADD

Your Interests

Interests: Amateur radio, Minecraft, Cardistry, Monsters, Inc., Taxidermy

SUBMIT

Figure 23. Discover Interests page

4.2.13 Managing RSS feeds

At the top of area 4 in Figure 10, clicking "Manage feeds" takes the user to the page displayed in Figure 24.

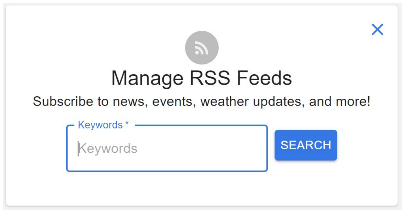
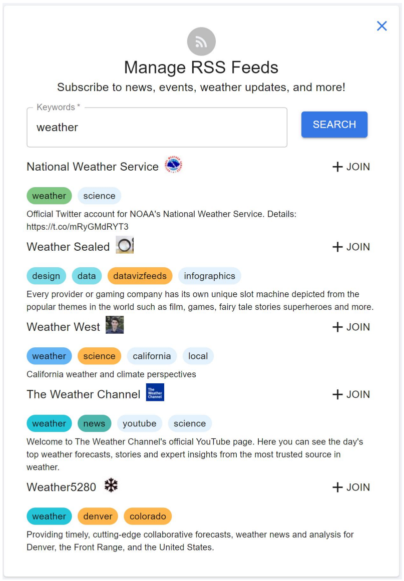


Figure 24. Manage RSS feeds



National Weather Service + JOIN
weather science
Official Twitter account for NOAA's National Weather Service. Details: https://t.co/mRyGMdRYT3

Weather Sealed + JOIN
design data datavizfeeds infographics
Every provider or gaming company has its own unique slot machine depicted from the popular themes in the world such as film, games, fairy tale stories superheroes and more.

Weather West + JOIN
weather science california local
California weather and climate perspectives

The Weather Channel + JOIN
weather news youtube science
Welcome to The Weather Channel's official YouTube page. Here you can see the day's top weather forecasts, stories and expert insights from the most trusted source in weather.

Weather5280 + JOIN
weather denver colorado
Providing timely, cutting-edge collaborative forecasts, weather news and analysis for Denver, the Front Range, and the United States.

Figure 25. Search for "weather" RSS feeds

Here, the user can search for relevant RSS feeds and click the "Join" button if they wish for it to be added. Figure 25 shows the search for RSS feeds related to weather and Figure 26 and Figure 27 show the selected RSS feed listed on the right of the page and also integrated into the messaging stream. Clicking the "Remove" button to the right of the listed RSS feed will remove it from the chat. Existing "messages" from the feed will remain but no new RSS updates will come through.

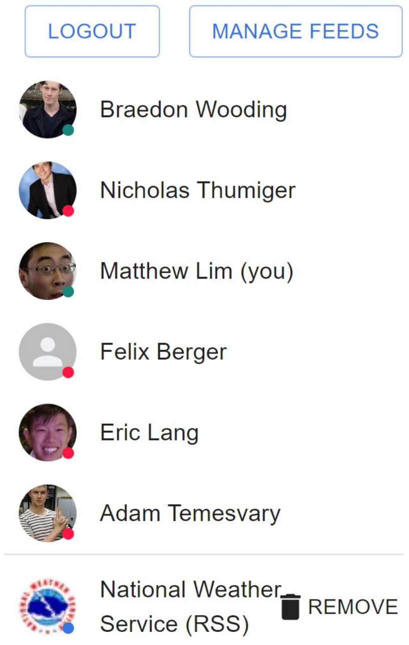
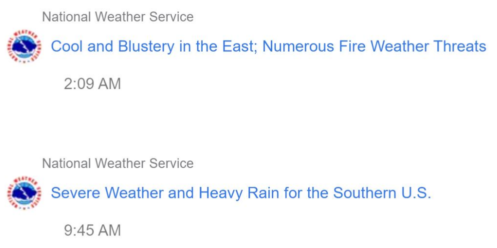


Figure 26. List of added RSS feeds


National Weather Service
Cool and Blustery in the East; Numerous Fire Weather Threats
2:09 AM

National Weather Service
Severe Weather and Heavy Rain for the Southern U.S.
9:45 AM

Figure 27. "Messages" from selected RSS feeds

Acknowledgments

We would like to thank and acknowledge the support of our friends and family over the past semester and their continued patience towards the many demonstrations that were part of the build up and creation of this report.

We would also like to give special thanks towards Dr. Rachid Hamadi for his ever-flowing mentorship and guidance over this semester.

References

[1] Reenskaug Trygve. The dci architecture: A new vision of object orientated programming, 2009. Last accessed 18 April 2019.
[2] SendGrid, 2009. Last accessed 23 April 2021.
[3] Stanislav Vishnevskiy. How discord stores billions of messages, 2017. Last accessed 18 April 2021.
[4] Teske Coletta. What is an rss feed? (and where to get it), 2020.
[5] Feedly. Feedly, 2021.
[6] Discord Bot List. Discord bot list, 2021.
[7] Google. Content-based filtering, 2017. Last accessed 18 April 2021.
[8] Ethan Nam. Understanding the levenshtein distance equation, 2019. Last accessed 18 April 2021.
[9] Edward Loper Steven Bird. Natural language toolkit reference, 2001. Last accessed 18 April 2021.
[10] Firebase, 2021. Last accessed 23 April 2021.
[11] NextJS, 2016. Last accessed 23 April 2021.
[12] Chris Coyier. A complete guide to flexbox, 2013. Last accessed 23 April 2021.
[13] Tom. Where is everything in microsoft teams stored at rest?, 2013. Last accessed 23 April 2021.

[14] FreeTDS, 2021. Last accessed 23 April 2021.

[15] Amazon Web Services, 2021. Last accessed 23 April 2021.

[16] Docker, 2021. Last accessed 23 April 2021.

[17] Azure Web Server S2. Azure web server s2, 2021.

[18] Redis. Redis, 2021.

[19] axios, 2021. Last accessed 23 April 2021.

[20] React, 2021. Last accessed 23 April 2021.