NFR

The top 3 NFRs that we prioritized this sprint were:

1. Useability
    a. We wanted our website to be as intuitive as possible. We made sure this happened by creating simple pages that users can navigate around in. With many quality of life improvements, it helps give the website a more professional feel, ensuring users enjoy their time on our website. This was chosen to be a priority because we believed that having a professional looking and feeling website would enhance the UX and UI of our website. We also believed that the functionalities we created for the website should be our top priority, ensuring the core behaviours of the website would be the main focal point and would draw the attention of potential consumers.
2. Scalability
    a. We validated endpoints to ensure the API calls were receiving the correct parameters which it needed to complete the API call. We also validated whether users were allowed to visit certain pages or access certain buttons/behaviours. We did this using Firebase Authentication and validating user tokens to ensure certain features/behaviours remained hidden from users that were not supposed to use it. We prioritized this because a fully-functioning website needs to have this sort of implementation. Regular users of the website have limited permissions, while the admins do not, allowing admins to oversee the actions that anyone else can do. This also ensures no unexpected behaviour happens, and the website's functionality and behaviours are maintained consistently and constantly.
3. Maintainability
    a. We made sure to create as many reusable components as we possibly could to maximize reusability and maintainability. This helps ensure maintainability because if a component which is used in multiple places on the website has a bug, we can fix said component and fix the functionalities on the affected areas of the website. This can also help us grow our website because the different sections of our website are not intertwined, they can be isolated for unit testing and confirmed they work as expected. Creating automated test cases can help us test the core functionalities as our website grows helps ensure maintainability as well.

2 trade-offs we had to make were:

1. Personal Time
    a. Since we wanted to be ambitious and planned to create more features than this project required (all while maintaining good code etiquette and making the website look as professional as possible), we had to sacrifice our personal time. Every member of the team sacrificed time that they could have used to prepare

better for other courses, to catch up on sleep, or to build relationships with their family and friends to ensure that the project's development stayed consistent with the roadmap we had initially created. It was our first time working as a group on a large scale and very professional website, and we wanted to make every sacrifice possible to ensure our goals would come to fruition. This resulted in a great turnout since our website has many interesting and unique features, as well as keeping a great UI for potential users. Although this seems like more of an abstract idea, we believe that this is a pivotal point in what made our website as great as it is.

2. Newer and Better Features vs Website Maintainability + UX
    a. In the development of this project, there were many times where we were working on features, but decided to delay working on the feature to instead work on some chore that would help website maintainability and customer experience. We believed that perfecting the features and aspects of the website before moving onto developing new features would help keep the website afloat, and would lighten the load and risk of potential bugs in the future. Although sometimes we may have been focused on getting a certain feature done, the group decided as a whole that ensuring UX and website maintainability was a requirement and not something that we could ignore (or delay for later).

Performance Testing Report

Using a JMeter testing suite, this was the report.

LABEL	# of Samples	Avg (ms)	Min (ms)	Max (ms)	Std. Dev.	Err %	Throughput	Received KB/sec	Sent Kb/sec	Avg. Bytes
Clubs	3000	4875	110	10614	1382.69	0	19.2	2215.63	2.79	342731.2
Club Page	1000	77	61	120	6.46	0	26.4	751.34	0.94	119306

Part 1. GET Clubs

This test focused on getting all the clubs within our database.

Part 2. GET Club Page

This test focused on getting information from 1 club for the club page

System Behavior

- The performance difference between the two parts were quite significant, with GET Club Page being a lot faster
- The club page met our expectations of being very quick to retrieve so we are able to display the club information to the user
- No errors were detected during the 4000 tests which indicates we have a stable backend
- Retrieving all clubs took a bit longer which means there is room for optimization if needed

Updated Performance Testing Report

Using a JMeter testing suite, this was the report.

LABEL	# of Samples	Avg (ms)	Min (ms)	Max (ms)	Std. Dev.	Err %	Throughput	Received KB/sec	Sent Kb/sec	Avg. Bytes
Clubs	2500	153	81	358	48.3	0	12	626.36	1.44	53492
Club Page	1000	80	64	113	6.46	0	92.6	72.28	13.30	799

Part 1. GET Clubs

This test focused on getting all the clubs within our database.

Part 2. GET Club Page

This test focused on getting information from 1 club for the club page

Updated System Behavior

- Since the last performance testing, Part 1: GET Clubs has received a massive increase in efficiency going from an average of 4875ms to just 153ms. This performance boost is largely due to the new method of image storage in our database.
- Again, no errors were detected during the thousands of requests which indicates that the backend is still stable following these changes
- While the average time to retrieve a single club has stayed relatively stable, the average data size has dropped significantly from 119306 bytes to just 799 bytes, again thanks to the optimized image storage.

Security Measures and Testing

Authentication:

- We authenticated users by using Firebase/Firestore functions. It takes in a user's email and password and creates their account (if signing in for the first time) or checks their credentials (if already a user).

Authorization:

- We have a Middleware and this serves as a layer between the frontend and the backend APIs. This is where we check what permissions a user has. We do this by grabbing the token of the user, using the token to fetch the userID, then using this ID to find out if the user is a regular member, executive member, or an admin. This allows us to ensure that a regular user is not able to access a certain page on the website that is meant only for executives or admins.

Data Protection Mechanisms:

- As said before, we use our Middleware to help authorize users and ensure that they have the correct permissions when using our website (and nothing more). This ensures that all sensitive data can only be seen by either executives or admins (depending on what the data is).
- We also hash passwords to ensure it is harder to have data breaches or account compromises, ensuring that once an account has been created and saved into our database, only the admins of the website are able to access it. This helps make sure the information related to the account cannot be compromised either.

Security Best Practices:

- We hash the passwords of users when they create an account by calling upon Firebase functions. This ensures maximum security as Firebase collects the password, hashes it, and saves it into its database without having to rely on external factors or technologies.
- We have implemented input validation by ensuring that when an API call is made, we check if it has the necessary parameters needed to complete the API call. If it does not, then we return an error message.

XSS Prevention:

- We ensure this doesn't happen by our Middleware functions (which regulate the permissions that any user can have). We also prevent malicious attacks from users by having API input validation (where we only take in parameters that we need). This can stop all malicious attackers that want to inject our website with infectious scripts because everything (ranging from the information the users try to send to the permissions that the user has) will be verified multiple times. All of this will stop any harm from occurring to our website.

SQL Injective:

- We don’t use SQL. We use Firebase and Firestore which verifies all information going in and out of it. This is a very secure and reliable method, ensuring nothing parallel to SQL Injection can happen to our database.

Scalability & Availability Considerations

How the system handles increased users or data

- To manage growth in users, our system will be designed for horizontal scalability so that we can handle an increased load by adding more servers
- To handle data, we plan to separate the database into a read replicas and maintaining a write-optimized primary database to make those operations quicker

Load balancing/Caching strategies

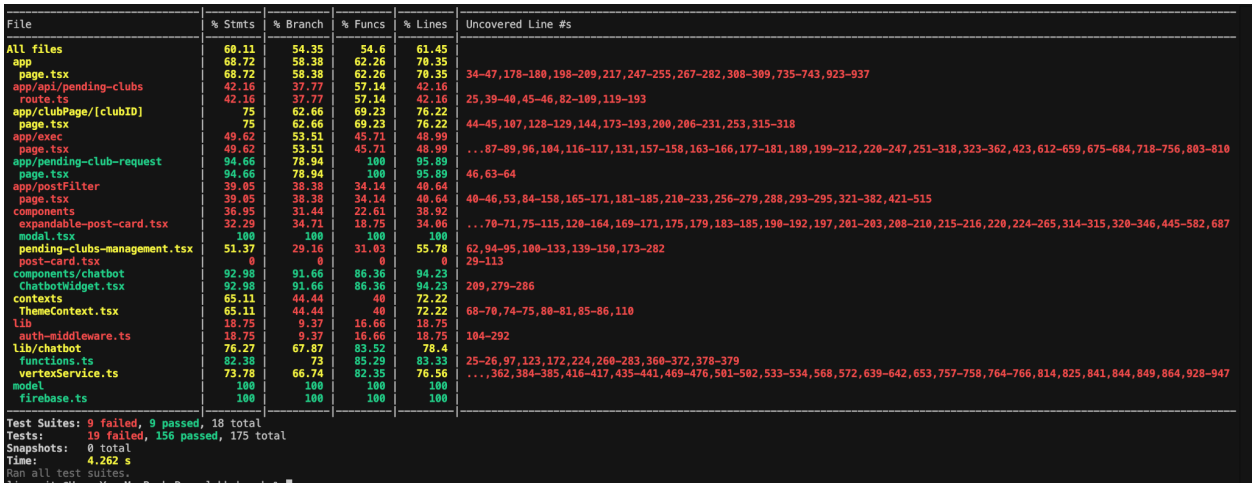
- To balance the load, we may use a load balancer to distribute the loads to the different servers to prevent any server from becoming a bottleneck

Deployment strategies and uptime considerations

- We plan to use a blue-green deployment strategy where we're running 2 identical production environments.
- Once the newer environment is fully tested and functional, we route traffic to the newer environment to ensure 0 downtime

NOTE:

There are a few test cases that fail in our testing file. It is meant to fail because it throws an error and we couldn't handle it. All new features were tested in the testing file. Some files are from previous sprints but have new additions to them, so those new additions were tested but not the entire file but most do have at least 70% line coverage. These features include calendar exports, autoscrolling feature for the home page. .



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	60.11	54.35	54.6	61.45	
app	68.72	58.38	62.26	70.35	
page.tsx	68.72	58.38	62.26	70.35	34-47,178-180,198-209,217,247-255,267-282,308-309,735-743,923-937
app/api/pending-clubs	42.16	37.77	57.14	42.16	
route.ts	42.16	37.77	57.14	42.16	25,39-40,45-46,82-109,119-193
app/clubPage/[clubID]	75	62.66	69.23	76.22	
page.tsx	75	62.66	69.23	76.22	44-45,107,128-129,144,173-193,200,206-231,253,315-318
app/exec	49.62	53.51	45.71	48.99	
page.tsx	49.62	53.51	45.71	48.99	...87-89,96,104,116-117,131,157-158,163-166,177-181,189,199-212,220-247,251-318,323-362,423,612-659,675-684,718-756,803-810
app/pending-club-request	94.66	78.94	100	95.89	
page.tsx	94.66	78.94	100	95.89	46,63-64
app/postFilter	39.05	38.38	34.14	40.64	
page.tsx	39.05	38.38	34.14	40.64	40-46,53,84-158,165-171,181-185,210-233,256-279,288,293-295,321-382,421-515
components	36.95	31.44	22.61	38.92	
expandable-post-card.tsx	32.29	34.71	18.75	34.06	...70-71,75-115,120-164,169-171,175,179,183-185,190-192,197,201-203,208-210,215-216,220,224-265,314-315,320-346,445-582,687
modal.tsx	100	100	100	100	
pending-clubs-management.tsx	51.37	29.16	31.03	55.78	62,94-95,100-133,139-150,173-282
post-card.tsx	0	0	0	0	29-113
components/chatbot	92.98	91.66	86.36	94.23	
ChatbotWidget.tsx	92.98	91.66	86.36	94.23	209,279-286
contexts	65.11	44.44	40	72.22	
ThemeContext.tsx	65.11	44.44	40	72.22	68-70,74-75,80-81,85-86,110
lib	18.75	9.37	16.66	18.75	
auth-middleware.ts	18.75	9.37	16.66	18.75	104-292
lib/chatbot	76.27	67.87	83.52	78.4	
functions.ts	82.38	73	85.29	83.33	25-26,97,123,172,224,260-283,360-372,378-379
vertexService.ts	73.78	66.74	82.35	76.56	...,362,384-385,416-417,435-441,469-476,501-502,533-534,568,572,639-642,653,757-758,764-766,814,825,841,844,849,864,928-947
model	100	100	100	100	
firebase.ts	100	100	100	100	

Test Suites: 9 failed, 9 passed, 18 total
Tests: 19 failed, 156 passed, 175 total
Snapshots: 0 total
Time: 4.262 s
Ran all test suites.