Final Report

CS261 - Software Engineering

Group 25
Finn Butler, Ion Orins, Petru Biriloiu, Shivakumar Mahesh, Zerui Shan Chen

1 Introduction

1.1 Motivation

During a single year, Deutsche Bank runs numerous events, from workshops, demos, presentations and lectures to reviews of the department. One of the primary aims of all of these events is to ensure that the audience is engaged and actively participating. While the participants' feedback is precious and can be used for the improvement of future events, the current audience will, unfortunately, not benefit from the feedback provided. Thus, it is essential to develop a system that allows the attendees to give feedback on the spot. This will allow the host to make the appropriate changes during the specific event and offer a smoother running for the audience, satisfying their needs and ensuring a better engagement.

1.2 Project Aims and Requirements

The aim of the project was to produce a prototype system that allows the participants of an event to provide live feedback to the host. Moreover, it will give an estimate of the overall "mood" of the whole group during the event or some particular sessions. Our team described the plan of the implemented system in the Requirements Analysis and Planning and Design reports; these represented the project's starting point. The Final Report presents more in-depth and more detailed the variety of functions the system has, the accomplishment of the requirements, the step-by-step implementation, the progress the team made with the development and the limitations, as well as the possible future extension of the work.

2 Market Research

Prior to the development cycle we conducted research into similar existing applications and those with a UI design we thought was effective. The aim of our research was to examine how these pre-existing applications managed and executed aspects such as UI, data representation, event structure and system architecture. We focused mainly on three applications: Slido [1], Vevox [2] and Kahoot! [3] We chose these applications as they are what we deemed the most popular applications resembling somewhat our design plan.

Slido most closely resembles the design we created in our Planning and Design document, so we decided to choose this as a primary research source. Slido is a Q&A and polling app designed to be used by event hosts during conferences, talks, etc. with the aim of engaging listeners and providing live feedback to the host. We were able to gain many valuable insights by investigating Slido and running a demo, such as an idea of how to represent feedback data using line charts, how to streamline the event joining process using QR codes and how to cater the UI to all levels of technological knowledge by making it forgiving and minimal. Vevox and Kahoot are similar applications however they are more focused towards knowledge-based quizzes as opposed to event feedback. However, they still proved useful as reference for shared features such as event code format, mobile-friendly UI and poll/quiz response UI.

Examples of components in BallotBox that are influenced by the research include:

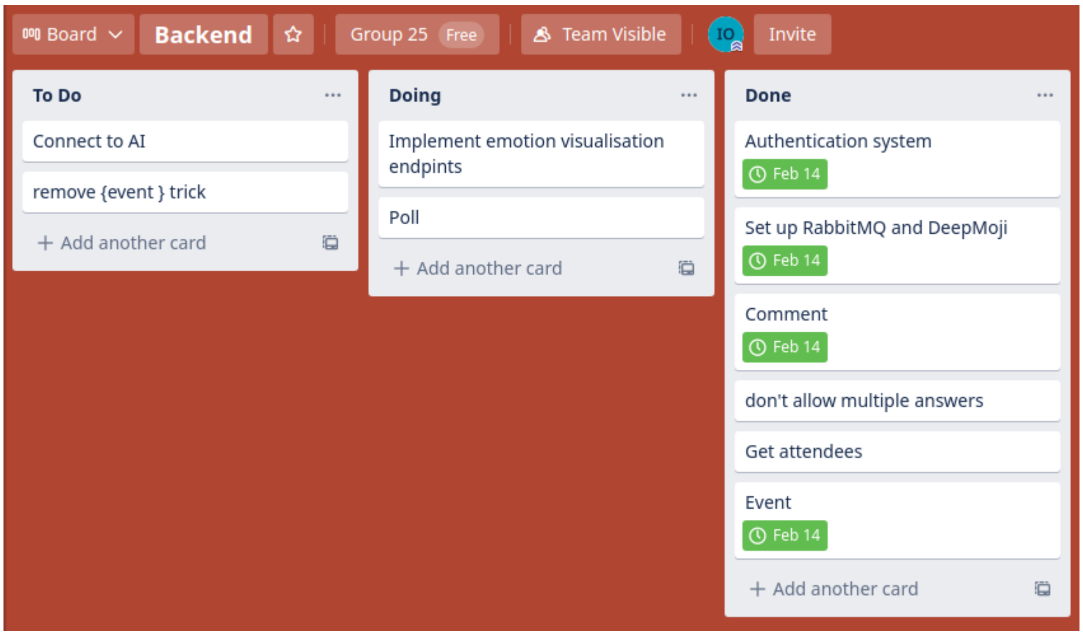
- Event code format. All 3 applications use a string of numbers of length 5-9, however we felt this could too easily be brute forced and scaled poorly. For example with codes of length 5 there can be at most 100,000 different event codes. In comparison, our implementation of the event code has almost 1 quadrillion possibilities. A greater possibility of event codes provides a greater level of security against brute force attacks and unauthorised access to events. Moreover, it means there can be a greater number of total active events.
- Mobile UI. All 3 applications put a lot of emphasis on mobile responsiveness, specifically by designing slightly different UIs for different devices without creating standalone apps. One key factor we noticed was that the mobile versions tend to be less detail oriented than desktop versions due to lack of presentation space. We used this principle for our frontend development, combined with the React library, in place of creating a separate mobile application.

3 Management

3.1 Procedures

The management process began with writing a procedures document, which contained the following:

- best coding practices (git with development and feature branches, use of code formatters, use of comments, emphasis on testing, etc.);
- channels of communication (when to use Discord, when to use email);
- overall agenda of meetings (review progress, update requirements and design, assign new tasks);
- introduction to the Kanban board [4] (figure 1).



To Do
- Connect to AI
- remove {event} trick

Doing
- Implement emotion visualisation endpints
- Poll

Done
- Authentication system — Feb 14
- Set up RabbitMQ and DeepMoji — Feb 14
- Comment — Feb 14
- don't allow multiple answers
- Get attendees
- Event — Feb 14

Figure 1: A snapshot of the backend Kanban board during development

In addition, a set of principles have been outlined:

1. The code must be easy to test.
2. The assigned roles are fluid, i.e. tasks can be transferred and shared.
3. We want to have a product identity.
4. Requirements should be easy to expand.
5. The requirements should be minimal for the current cycle.

The last two principles were put into place in order to be prepared for potential requirements changes demanded by the client.

3.2 Sprint Cycles

The methodology, mainly agile, was heavily inspired by the project manager's experience in the industry. As the client did not require to see any demos throughout the development process, the emphasis was placed on sheer progress, rather than having a deliverable at the end of each sprint cycle. This allowed us to design a system with fully independent components and assign each developer only one component, which gave the programmers the possibility to choose to work on any task they wanted. This increased motivation, practically eliminated the need for developer X to finish working on A, in order for developer Y to be able to start working on B, and overall speed up development.

The sprint cycles had the length of one week and started with a meeting, in which every member demoed their work and presented the obstacles they faced. Next, we found solutions to the issues or remade the design of the system in certain cases in order to avoid the obstacles. The last item on every meeting's agenda was setting goals for the next sprint cycle, followed by a week of mostly asynchronous development. In addition, a week before deadlines, we would start having nightly documentation writing sessions.

We hosted the entire codebase in a GitHub repository, with each aspect of the development process having a different branch. After each weekly sprint cycle these branches were pulled to main, assuming they were functional, such that the entire application could be regularly tested. Over the entire development cycle of roughly 50 days, a total of 220 commits were pushed by the developers of each component.

The responsibilities for each member were the following:

- frontend developer: create an interface for the user to perform all the actions outlined in the requirements document;
- backend developer: implement the business logic;
- machine learning developer: create an algorithm, which given a sentence, returns emotion data on it;
- tester: make sure all functionality is properly implemented;
- business analyst: write the bulk of the documentation, plan out the demo video.

The team members generally worked on their own responsibilities, with two exceptions:

- because during the middle of the term, there was less documentation to be written, the business analyst has helped the machine learning developer sanitise and preprocess data;
- because frontend was the most complex task and was running a bit late on the schedule, the backend developer has also contributed to the frontend.

3.3 Risks

A few risks outlined in the risks matrix from the Planning and Design document were actually encountered. One of them was the "software incompatibilities" risk, which will be described in the "Backend" section of the "Development" chapter, and another one was the "Dataset-label unavailability" risk, described in the "Emotion Detection" section of the same chapter. Both of them were solved using the contingency plan we outlined at the time of creating the document.

One other obstacle we faced was the fact that the members of our team live in different time zones, which made scheduling and communicating time rather difficult. This issue was partially solved by resorting to using the GMT time zone when scheduling any meetings.

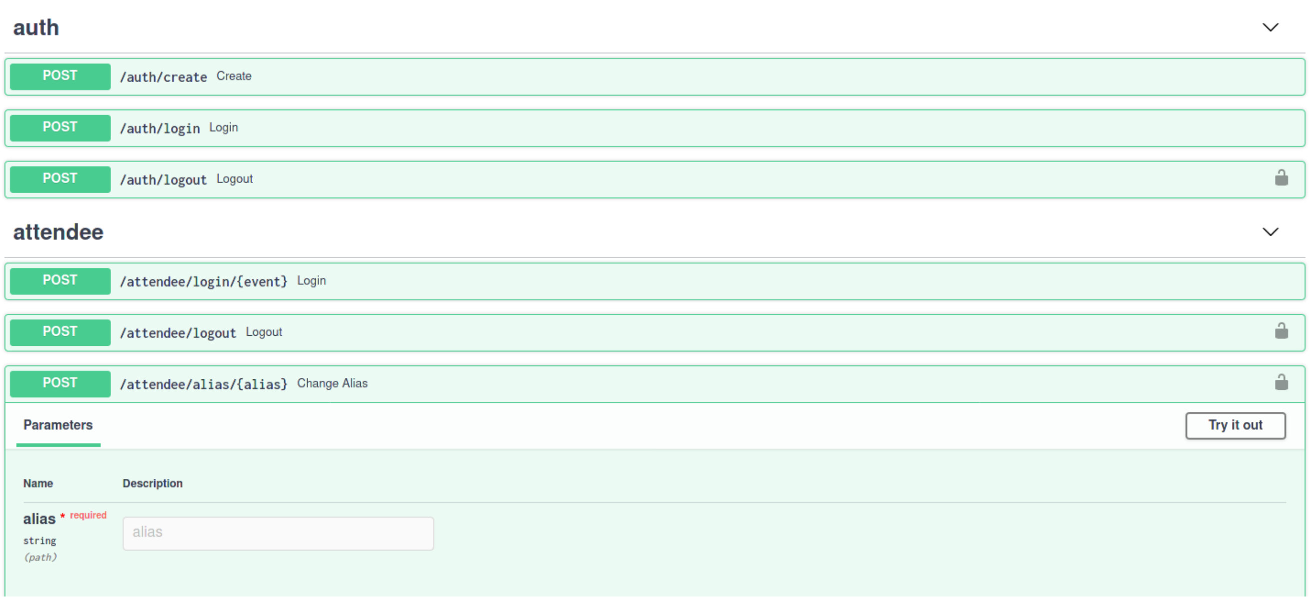
Unexpectedly, a risk that we hadn't foreseen arose from one of our contingency plans, namely using Docker [5]. Even if it alleviated the need to configure each dependency manually, getting Docker to run turned out to be a challenge in itself. As every developer uses a different operating system, our experiences varied widely, as follows:

- Manjaro - ran smoothly with no issues;
- Windows - issues with speed and used resources, a bit of tweaking however solved most of the problems;
- MacOS - initially was not able to build the containers because of memory limitations, changing the settings and a restart have solved the issue;
- Ubuntu - critical issues when configuring Docker Compose [6], however as Docker was properly installed, we have manually written a bash script that orchestrated the relevant containers for the affected developer.

4 Development

4.1 Backend

The development of the backend component was the fastest, mainly thanks to the modern technologies that have been used, the bulk of the work being done well ahead of the schedule (about 2 days for 90% of the functionality). FastAPI [7] allowed for quick iterations, with its auto-reload function, triggered each time changes in the code base are detected, and the automatic interactive API documentation generator (figure 2), which allowed the developer to validate the implemented functionality rapidly.


auth

POST /auth/create Create

POST /auth/login Login

POST /auth/logout Logout

attendee

POST /attendee/login/{event} Login

POST /attendee/logout Logout

POST /attendee/alias/{alias} Change Alias

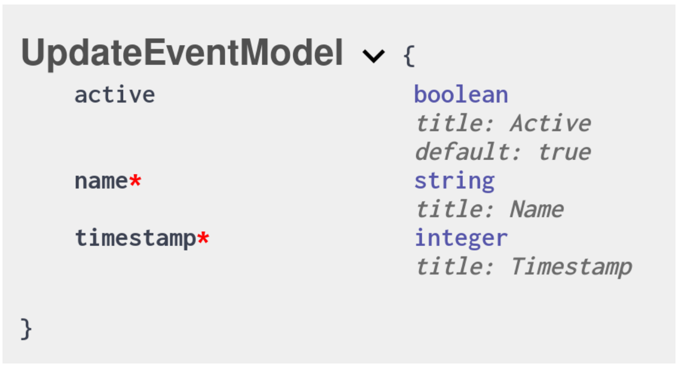
Parameters

Name	Description
alias * required string (path)	alias

Figure 2: SwaggerUI [8]: A tool that made backend development and testing significantly easier

Python [9] allowed for flexibility in the code, rewriting functionality and fixing bugs being quick and easy. In combination with MongoDB [10], this created an interesting synergy, as any structure could be defined on the fly and saved in the database. For example, saving likes for the comments became less cumbersome compared to a relational database, as we saved them as a list of the IDs of the people that have liked the comment in the comment model. Thus liking, means adding the ID of the attendee to the list, unliking, means removing the ID, and counting the number of likes a comment has means calculating the length of the list.

Even if this flexibility is allowed, it should be used wisely, as it can make the programmer more prone to errors. Thus, we used Pydantic [11], a data validation library, to parse and validate data received from the user and make sure that the data saved in the database is consistent (figure 3).


```
UpdateEventModel ⌄ {
    active          boolean
                    title: Active
                    default: true

    name*           string
                    title: Name

    timestamp*      integer
                    title: Timestamp

}
```

Figure 3: An example of a schema defined using Pydantic

The swiftness of the backend implementation prompted a unique interaction between the frontend developer and the backend developer: the frontend developer could request changes to the backend in real-time, in order to better suit the logic of the component and the needs of the used libraries. As an example, the plotting library required a certain format, which was easy to implement in the backend, alleviating the need for data preprocessing to be done on the devices of the users.

During the development of this component, an issue related to the integration of the emotion detection algorithm arose. It was written in Python 3, but it had to use a deep learning model written in Python 2. The problem was solved by placing the algorithm and the model in different containers (which fixed the dependency issues) and letting them communicate between each other and with the backend through a message-broker software (RabbitMQ [12]). In addition, we have added a functionality that restarts the executables in the two containers in case of a crash, in order to increase robustness.

Overall, the backend consisted of three FastAPI apps: auth, host and attendee. The auth app implemented the host authentication functionality, using the OAuth [13] standard, and included the endpoints below (table 1).

Endpoint	**Description**
POST /auth/create	Receives email and password as input, creates user in database, returns session token.
POST /auth/login	Receives email and password as input, returns session token.
POST /auth/logout	Receives session token as input, deletes session from database.

Table 1: The auth endpoints

The host app contains the rest of the endpoints relevant to the host. All of them require the session token to function (table 2).

Endpoint	**Description**
POST /host/event	Creates event, returns code for the event.
GET /host/events	Returns events owned by host.
GET /host/event/{code}	Returns details for specified event.
PUT /host/event/{code}	Modifies the details for specified event.
GET /host/event/{code}/comment	Returns all comments for event.
POST /host/event/{code}/comment	Allows host to comment.
POST /host/event/{code}/comment/like/{id}	Allows host to like/unlike comment.
POST /host/event/{code}/poll	Allows user to post a poll.
GET /host/event/{code}/polls	Returns posted polls with answers for specified event.
PUT /host/event/{code}/poll/{id}	Allows user to modify specified poll.
GET /host/event/{code}/attendees	Returns the IDs of the attendees for specified event.
GET /host/event/{event}/mood/polarity	Returns the sentiment polarity graph for specified event. Takes interval as optional parameter.
GET /host/event/{event}/mood/{emotion}	Returns the mood graph for specified emotion.
GET /host/event/{code}/currentmood	Returns the radius for the emoji shown in the emotion dashboard as calculated from the average mood for the last x minutes.

Table 2: The host endpoints

The endpoints implementing the attendee functionality are contained within the attendee app and are as follows (table 3):

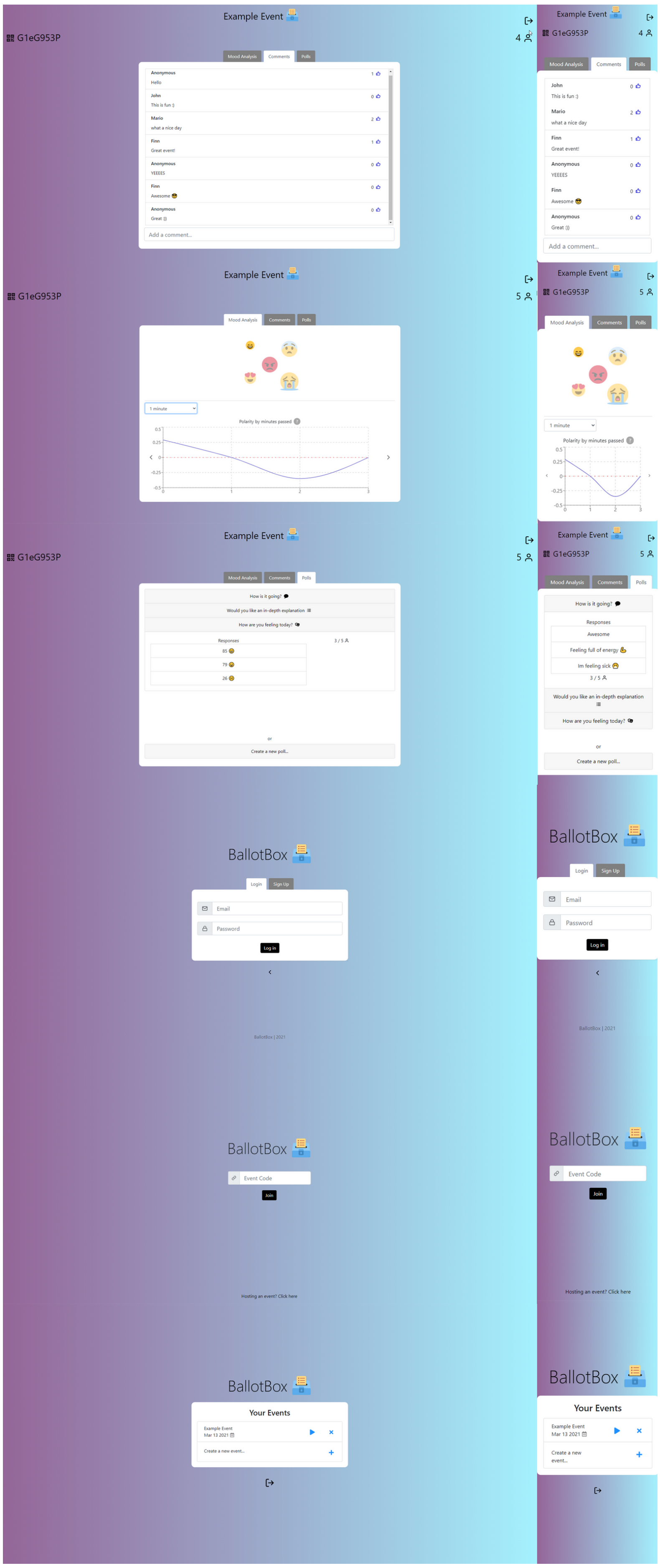
Endpoint	**Description**
POST /attendee/login/{event}	Returns session token for specified event.
POST /attendee/logout	Deletes session token from database.
POST /attendee/alias/{alias}	Changes the attendee's alias.
POST /attendee/comment	Allows attendee to comment.
GET /attendee/comments	Returns all comments from the event the user is attending.
POST /attendee/comment/like/{id}	Allows attendee to like/unlike comment.
GET /attendee/polls	Returns all polls posted by the event's host.
POST /attendee/poll/{id}/answer	Allows user to answer to specified poll.

Table 3: The attendee endpoints

All of the endpoints above, except the login endpoint, require a session token in order to be called.

4.2 Frontend

The development of the frontend component was the lengthiest aspect of the entire project. UI design and implementation iterations began immediately after the group's initial meetings and continued until the very end of the development cycle. As discussed in our early documents, the overall development of the frontend component strongly followed the incremental build model: initially the bare pages for each view model were created, with required features being added incrementally during weekly sprints until all requirements were fulfilled. From there, feasible stretch goals were attempted and the system testing process began.



Anonymous
Hello

John
This is fun :)

Mario
what a nice day

Finn
Great event!

Anonymous
YEEEES

Finn
Awesome 😎

Anonymous
Great :))

How is it going? 💬
Would you like an in-depth explanation
How are you feeling today?
Responses
85 😀
79 😀
26 😀
3 / 5

Awesome
Feeling full of energy 💪
Im feeling sick 😷

Figure 4: Screenshots from the application, desktop on the left and mobile on the right

The UI was built using React Bootstrap [14], a library implementing the popular CSS framework Bootstrap [15] into React-specific components. The library provides pre-built and styled low-level components such as containers, form groups, buttons, tooltips and much more [16]. Bootstrap's own CSS also provides prewritten classes for typography, margins, padding etc. We wrote our own CSS classes for specific components and overwrote some of bootstrap's pre-built CSS to ensure continuity between components and a custom colour scheming across the entire app.

For displaying the mood charts, we used the Recharts library [17]. Recharts is a React wrapper for D3.js (Data-Driven Documents), a pure JS library for data visualisation and graphics [18]. Recharts provides an excellent set of chart and graph components for interactive and responsive data representation while still being easy to use. We used Recharts' Line chart component, customised to be responsive in our application and with customised axes, labels and colours. From there, we connected the mood data API endpoint to the chart, resulting in a clean, animated and interactive line chart.

During development these smaller individual components, either bespoke or from libraries, were combined to create larger scale macro components such as the comment wall and the login/signup forms. As these macro components were incrementally created they were positioned and styled to fit on the base pages of the different views (e.g. Host and Event). This created a tree-like hierarchy of components built out of subcomponents, as seen in figure 5, with arrows representing dependency on another component.

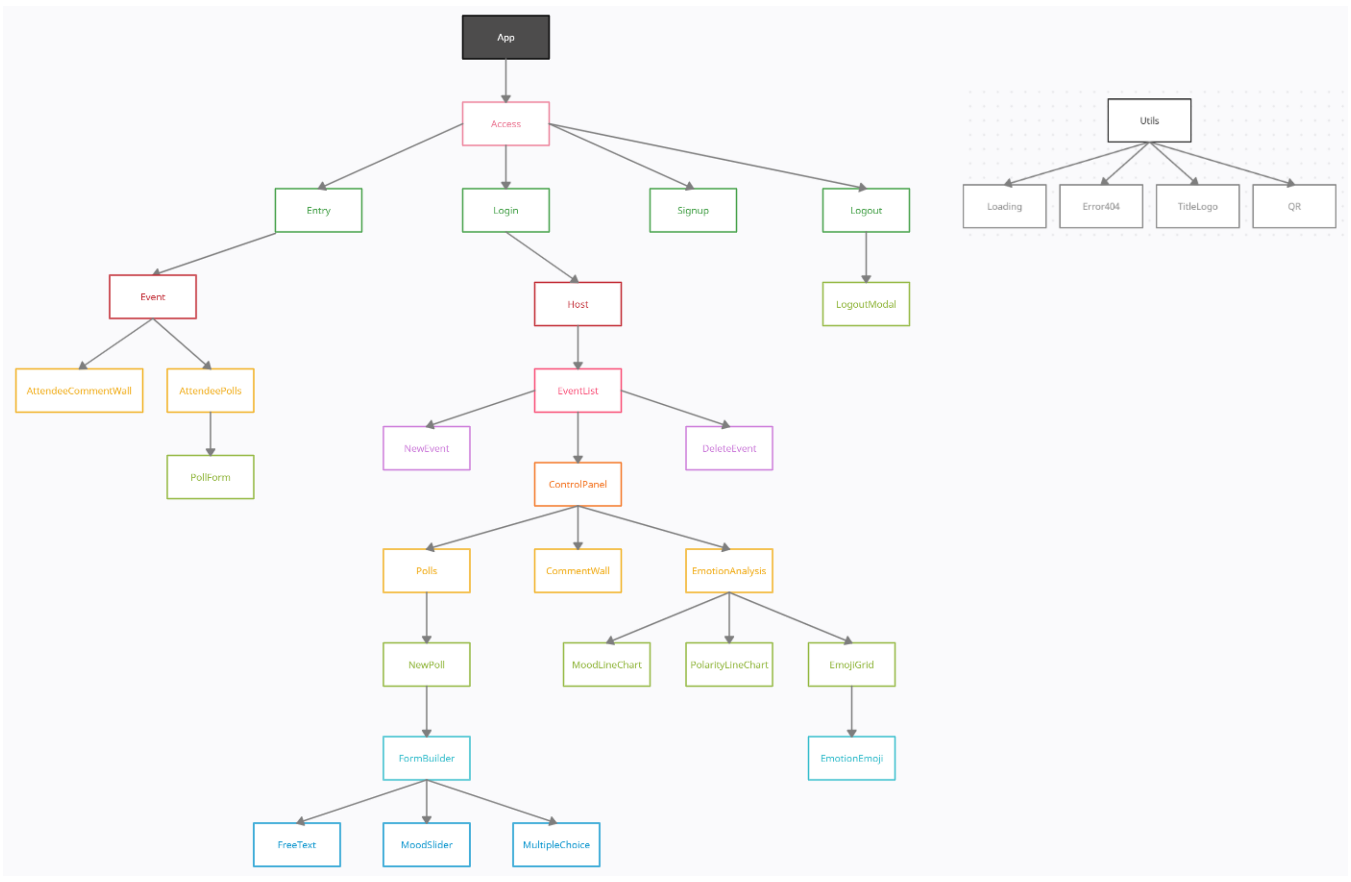


Figure 5: Hierarchy of React components

The branding and identity of the project remained relatively constant from the start as laid out in our design document, with the product name and colour scheme being the same. A minor change was carried out by changing the logo emoji from Apple's version to Twitter's as it is open source and free to use for any commercial or non-commercial product [19].

In terms of user experience, the design of our interface attempted to follow some common laws of UX [20]. Throughout the application, we have crafted the styling according to a number of these rules, including Hick's Law, where the time taken to make a decision increases with complexity and number of choices [21]. As well as this, we focused on Jakob's Law which states that users would prefer the website to work the same as other well known sites [22], such as those we picked for research early on in the project. From these two laws we decided that simplicity would be essential especially as all users would have different levels of technological knowledge and computer literacy. Moreover, we decided to implement common features in ways similar to other applications related and non-related. For example, the use of icons and terminology such as "login", "signup", "homepage", "like" etc. are extremely similar to those used by many other websites, so will provide our users with a familiar interface to interact with. Achieving the correct balance between simplicity and detail for hosts was rather difficult, but we aimed to solve this by creating a novel concept of variably sized emoji icons to display mood, that combines the simple concept of emoji with the more complex concept of sentiment analysis. A secondary key aim of our UI design was to be forgiving. Having assessed the possible utility of our application, we recognised that it should cater to all levels of digital literacy and technical capabilities. If our application is forgiving, users will be less likely to make significant mistakes and will have a better experience overall. To achieve this, we implemented various common practices in forgiving UI, such as confirmation dialogue boxes on event deletion/logout and back buttons. Some of these features came about and were validated as a result of the acceptance testing we performed (see section), from candidates who we assessed to have low digital literacy.

Weekly check-ins were held with the project manager who would examine and test that specific week's iteration of the build. Any urgent issues or changes that needed to be made, as well as suggestions for the next cycle's development were reported to the frontend developer during these meetings. This meant that the end product of the overall development cycle was satisfactory for both the frontend developer and the project manager, and the only further changes needed would be as a result of testing. The testing head then began system and integration testing, submitting any bugs found to the frontend developer right away. Various bugs were found initially, ranging from styling issues to security vulnerabilities. Having fixed these bugs, the acceptance testing period began as discussed in the testing section. The final changes made to the overall product were made as a result of comments made during these tests, mostly involving accessibility and clarity issues. For example, many users found it challenging to understand the graphical representations of mood, so as a result, a help tooltip was added to explain the meaning of the charts. As well as this, some extra icons were added to improve clarity of where hosts could find the code for their event.

Overall, we attempted to follow and fulfil all the standards of creating a progressive web application [23], including cross compatibility, responsivity, loading speed, security and similarity to installable applications. Progressive web apps are hugely beneficial to user experience, generally providing a clear and easy to use interface that can be accessed by anyone anywhere with an internet connection. Further, they are secure and streamlined, ensuring all data is encrypted and no installation of any sort is required to utilise the full capabilities of the application. To ensure we were following the necessary standards for a progressive web app, we used Google's Lighthouse tool [24]. Lighthouse is an automated auditing tool that can be run in-browser to measure the quality of any web app. Aspects of the application that are measured include performance, accessibility and SEO, as well as all the progressive web app standards [25]. Some aspects of the audit were unattainable or not worth focusing on for this prototype, specifically SEO related quantifiers. We mainly focused on the performance and speed indicators to make sure our app loaded in acceptable times as efficiently as possible. One key change to the frontend that came as a result of using Lighthouse was to incorporate the use of React's lazy loading, [26] where only parts of the generated JS would be sent at a time. As well as this, a fallback page was added such that instead of going to a blank white screen when loading, we could use a custom made loading page so as to further achieve the sense of being a "one-page" application. All in all, these changes should make the app more efficient at loading on slower networks or devices, something that greatly increases the quality of life for users.

4.3 Emotion Detection

Sentiment analysis is the flagship feature of our app and is performed using an algorithm designed by our team, which, by being able to use both text and emojis, pushes the limits of what is currently possible. (figure 6)

The first approach we considered was training a neural network to classify raw emotion based on text and emoji data.

We wanted to differentiate explicitly between the words in the text and emoji data because the emoji ☺ is fundamentally different from the word "they" (for example) in its ability to predict joy. It is clear that emojis are superior in their prediction power to most words, and so the text with emoji data would be parsed, with the emojis being treated differently to standard words in the English vocabulary. (If there was no emoji present we would use text input alone and vice versa.)

We found the deep learning model DeepMoji [27] that could take in as input text data and output a probability distribution over emoji. If the output of this model had been some probability distribution over emotions, taking the maximum likelihood estimate of emotion from that distribution would solve the mood prediction problem, but this was not the case. However, if we could find out the raw emotion expressed by each emoji as $\mathbb{P}(emotion|emoji)$ then we could take a matrix vector product with the matrix containing as its elements conditional probabilities $\mathbb{P}(emotion|emoji)$ over all considered emotions (as rows) and emojis (as columns), and with the vector being a probability distribution over emojis. The matrix vector product, i.e. the linear transformation, is rigorously justified due to the law of total probability.

Further, given emoji data, one can convert this into a probability distribution over emojis and apply the same linear transformation to get a probability distribution over emotions. This was termed the *emoji learner*.

The task therefore was to estimate the conditional probabilities $\mathbb{P}(emotion|emoji)$. The idea was that DeepMoji had already learnt the statistically significant properties of emotion from text using a neural network and over a million tweets. We just needed to leverage DeepMoji's prediction power by estimating the above conditional probabilities.

The main objective was to discover and apply rigorous methodology to estimate the conditional probabilities $\mathbb{P}(emotion|emoji)$ over all considered emotions and emojis, in order to convert from a probability distribution over emojis to a probability distribution over emotions.

Using Bayes Theorem we could find

$$\mathbb{P}(emotion = e|emoji = j) = \frac{\mathbb{P}(emoji = j|emotion = e)\mathbb{P}(emotion = e)}{\mathbb{P}(emoji = j)}.$$

$\mathbb{P}(emotion = e)$ was available from the data, as $\frac{|\{i : y(i) = e\}|}{N}$ where y is the emotion labels, and $\mathbb{P}(emoji = j)$ can be derived using law of total probability once the likelihood $\mathbb{P}(emoji|emotion)$ was derived. For simple models, an analytical formula for the likelihood function can typically be derived. However, for more complex models, such as ours, an analytical formula was elusive.

We used Approximate Bayesian computation (ABC) methods to bypass the evaluation of the likelihood function analytically. We instead approximated a likelihood function using the transformed dataset by conditioning on the sample space where the emotion labels were a particular emotion (for example "anger") and then taking the average of all of the emoji distributions whose label was "anger". Hence the ABC method applied on the transformed dataset X whose rows are probability distributions over emojis with N rows and p columns and emotion labels y, yielded the formula:

$$\begin{aligned}
\mathbb{P}(emotion = e|emoji = j) &= \frac{\mathbb{P}(emoji = j|emotion = e)\mathbb{P}(emotion = e)}{\mathbb{P}(emoji = j)}. \\
&= \frac{\mathbb{P}(emoji = j|emotion = e)\mathbb{P}(emotion = e)}{\sum_{e'} \mathbb{P}(emoji = j|emotion = e')\mathbb{P}(emotion = e')} \\
&\approx \frac{\frac{\sum_{\{i:y(i)=e\}} X(i,j)}{|\{i : y(i) = e\}|} \frac{|\{i : y(i) = e\}|}{N}}{\sum_{e'} \mathbb{P}(emoji = j|emotion = e')\mathbb{P}(emotion = e')} \\
&= \frac{\frac{\sum_{\{i:y(i)=e\}} X(i,j)}{N}}{\sum_{e'} \frac{\sum_{\{i:y(i)=e'\}} X(i,j)}{N}} \\
&= \frac{\sum_{\{i:y(i)=e\}} X(i,j)}{\sum_{1\le i\le N} X(i,j)}
\end{aligned}$$

However since Bayesian paradigms assume prior distribution, and our prior distribution over raw emotions was estimated from the dataset, any dataset which had unrealistic frequency of a particular emotion would severely impact the prediction accuracy of the model.

The solution was to assume that all emotions were felt with realistic frequency and delete particular rows to modify the dataset to reflect that assumption. However, unintuitively, it led to bad estimates on the complex emotion of emojis because we had made the assumption that DeepMoji would return a theoretically perfect probability distribution over emojis which turned out not to be the case. So the best solution was to assume equal frequencies of emotion labels in the dataset, and the dataset was modified to reflect this assumption. This corrected any bias in the training data and led to much better estimation of the conditional probabilities.

We were then able to separately learn a probability distribution over raw emotions from text data and learn a probability distribution over raw emotions from emoji data, with the next step being to combine these two learners.

The initial idea was to use Bayesian statistics. We considered two distributions on the probability that an attendee was feeling a particular emotion. The prior was to consider their emotional state based purely on text and the posterior would then condition on the "evidence" which would be the emoji present in the text. As for the likelihood, "the probability that one uses a particular emoji given that they feel a particular emotion" was also tractable.

But there was a huge problem with this approach. The issue was that the likelihood was only tractable if the user inputted exactly one emoji (or none) in their text. The question "what is the probability that one uses these three emojis given that they feel angry" was not addressable without making additional simplifying assumptions about emoji distributions that may not hold in real time.

As a consequence, we then needed to find a different procedure. Thus, we considered a simpler approach to combining two probability distributions. Since both the emoji learner and text learner were strong predictors all we really needed to do to combine their power was to take a simple weighted average of the distributions. This method is called Simple Weighted Ensemble. However this begs the question "Which is the best weighted average between these two distributions that best explains emotion?" This could be answered by estimating the ensemble parameter, which is a measure of how much to weight each of the distributions. All we then had to do was perform tests on validation data to estimate the best parameter.

However, we then faced another problem: obtaining the validation data. We had assumed in our planning that raw text data with emojis would be readily available on the internet. It was an exercise in futility to search "text dataset containing emojis". Unfortunately, we were not able to find such a dataset in a reasonable time frame. We understood that the best option was to create our own dataset that contained the WhatsApp/Snapchat/Facebook messages we had sent to or received from our own families and friends (with their consent). The data was then hand-labelled into raw emotion and polarity categories.

Now that we had our validation dataset, we estimated the ensemble parameter which had best performance on this data. And the best estimate was to use 60% of prediction power from text and 40% from emojis.

The test accuracy of the model on raw emotion (5-class classification) was 71.1%. This was mostly due to the confusion within positive emotions (love and joy) and confusion within negative emotions (anger, fear, disgust).

The same algorithm applied to binary classification of text and emoji data to "positive emotion" or "negative emotion" was close to 100%. The algorithm used on polarity (3-class classification which included positive, neutral, negative) was 90% accurate.

The main features of the emotion detection algorithm is therefore a Neural Network (DeepMoji) (transformation of text to emoji distribution), Approximate Bayesian computation (transformation of emoji distribution to complex emotion) and Ensemble Learning (integrating both the text learner and emoji learner).

The last step was displaying the results to the end users. We used the following algorithm to display the mood evolution graphs of the audience:

1. Each time a comment is submitted, analyse the comment and compute it's polarity and complex emotion (probability distribution over emotions);

2. For each evenly-spaced time interval of length m minutes starting from the first comment of the event to the present, compute the weighted average of the polarity and the weighted average complex emotion for each comment posted in the respective window, weighted proportional to $1 + \#likes$.

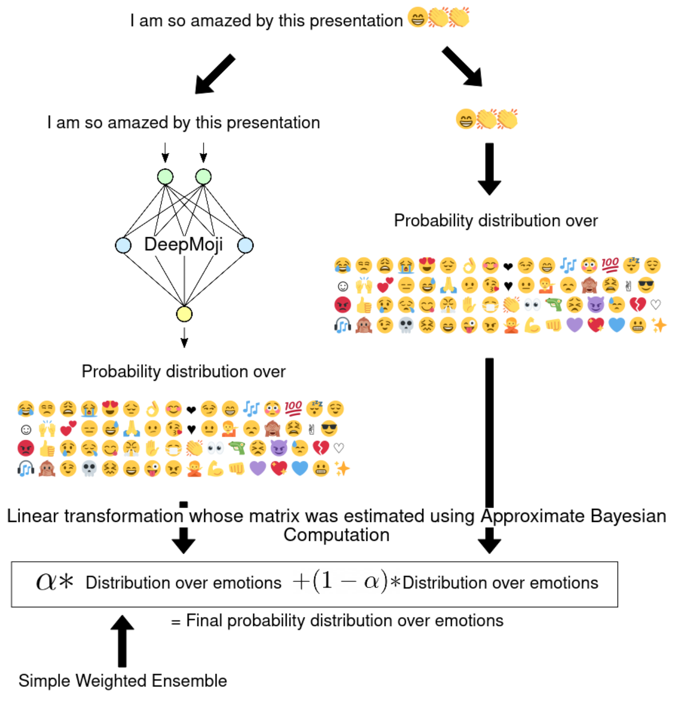

I am so amazed by this presentation 😁👏👏
I am so amazed by this presentation
😁👏👏
DeepMoji
Probability distribution over
Probability distribution over
Linear transformation whose matrix was estimated using Approximate Bayesian Computation
$\alpha*$ Distribution over emotions $+(1-\alpha)*$Distribution over emotions
= Final probability distribution over emotions
Simple Weighted Ensemble

Figure 6: The pipeline of our emotion analysis algorithm

In addition, for the emoji sizes described in the frontend section, we compute the radius of each of the 5 emojis indicating "joy", "anger", "fear", "sadness" and "love" by averaging the complex emotion of all comments submitted in the last m minutes, and square rooting the results, so that the relevant information is reflected in the emoji areas. That is, the area (and not the radius) of each of those emoji must be proportional to the it's respective probability.

5 Testing

5.1 Unit Testing

5.1.1 Backend

The backend unit testing was done using the Pytest library [28]. Unit tests were written for each API endpoint, and using the extension pytest-cov [29] for the library that shows the coverage, we made sure that the tests cover the entire body of the tested methods.

5.1.2 Frontend

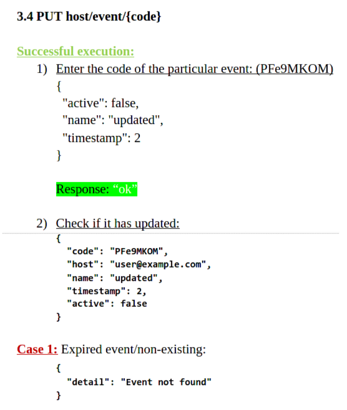
As the frontend was created using Create-React-App [30], it came built in with a testing environment using Jest [31] and react-testing-library [32]. Jest is a minimalist JavaScript testing framework that, when working in conjunction with react-testing-library, provides a rigorous set of tools for testing individual React components. We utilised this testing environment to make sure all individual lower-level components in the frontend codebase would render correctly without crashing.

5.2 RESTful API Testing

In order to check the functionality and performance of our programming interfaces, we carried out some API testing. Testing was carried out by sending requests to each backend endpoint and verifying the system's response to those calls. The objective was to watch the performance of the data exchange between the APIs and which requests and/or data formats can be used. Different endpoints returned different types of results:

- returns value based on the input condition (verify expected and obtained outputs)
- no return value: check the behaviour of the API or status (failed/passed)
- API intended to trigger some other API (track those APIs)
- update data (validate outcome)

We performed different methods for API testing which included discovery testing; manually executing calls and verifying which resources can be created/deleted/updated by each interface or usability testing; making sure the APIs are functional and developer friendly.



3.4 PUT host/event/{code}

Successful execution:

1) Enter the code of the particular event: (PFe9MKOM)

```
{
 "active": false,
 "name": "updated",
 "timestamp": 2
}
```

Response: "ok"

2) Check if it has updated:

```
{
  "code": "PFe9MKOM",
  "host": "user@example.com",
  "name": "updated",
  "timestamp": 2,
  "active": false
}
```

Case 1: Expired event/non-existing:

```
{
  "detail": "Event not found"
}
```

Figure 7: Excerpt from the API testing document

Each endpoint was individually tested with different scenarios to examine how they performed and display any potential exceptions and/or limitations. Different scenarios included sending unusual inputs in our calls to see if the interfaces threw errors which were then documented (figure 7). In successful scenarios we verified the output and documented the results, as most of the API's outputs are related to each other. Some of the minor errors we found and solved where the following:

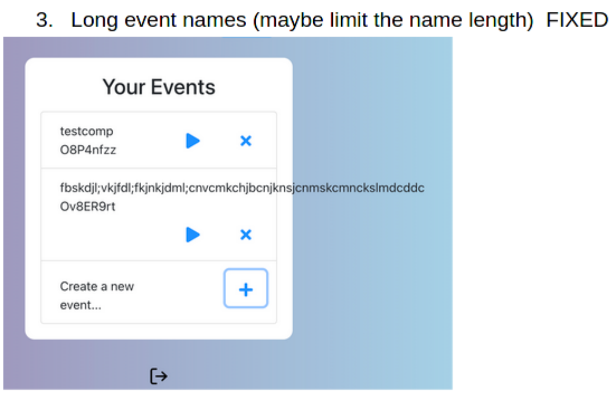
- internal server errors on certain endpoints when no comments were submitted yet (caused by division by 0)
- data being correctly outputted but not how we intended it to work, as inputting no name events in "POST /host/event"

5.3 System Testing

After the integration of every component, we carried out verifications to check that the system met the functional and nonfunctional requirements specified.

Firstly we wanted to make sure that our product could be run on different devices, as we intended the host to use larger devices such as large tablets/desktops and mobile phones for attendees. We developed our application in a responsive way and took advantage of Google Chrome's "toggle device toolbar" to make sure that the program showed properly in mobile devices. However, when actually trying it on physical devices, some of the HTML containers would be differently placed, so we had to make some adjustments on the containers placements manually.

The main tests were carried out by running a test event on the application and checking that the interaction between the frontend and backend did not cause any unusual errors and that they outputted the expected results. Moreover, any issues found were added to a document with fixes that could be implemented, some with higher or lower priority, as we thought that time management could be an issue if every fix was taken into account (figure 8).


3. Long event names (maybe limit the name length) FIXED

Your Events
testcomp
O8P4nfzz
fbskdjl;vkjfdl;fkjnkjdml;cnvcmkchjbcnjknsjcnmskcmncksImdcddc
Ov8ER9rt
Create a new event...
+

Figure 8: Excerpt from the system testing document

Some of the major changes included long event names that would display outside of the container, which was fixed by limiting the length of the event names; entering links to events that do not exist would also break the website, so we implemented a "404 error" page that would show up if this was the case.

Other minor fixes were such as wrong email formats outputting "wrong password alerts" or being able to set a 6 digit year on events, which was fixed by limiting the event creation date to 50 years in the future.

Overall, we made sure that the system was functioning correctly and it met the specified requirement; fixed and implemented some other features that we found during the process of testing.

5.4 Stress Testing

Stress testing was carried out in order to make sure that our system was robust. It was automatically performed by using scripts which pushed the system to its limits. This process was performed both to individual components and the system as a whole.

For instance, our main concerns were about checking the maximum number of possible attendees per event. For this case we ran a script which would call the attendee login endpoint to a certain event for a long period of time. Results showed that our system was able to support large numbers (¿200) and still perform without any problems.

Another concern was spamming comments. Overall, most of the test cases passed successfully. First, we tried spamming manually ourselves, but the system showed no signs of slowing down. The system was also shown to be able to withstand certain text inputs known for their buggy behaviour, such as the longest Unicode character and injectable scripts. However, we did find an unsuccessful test scenario: when using a macro spammer at 1ms intervals, the system would respond with a great delay as it would be processing every comment being spammed by the macro.

Two more automatic stress tests were performed: a slow loris attack [33] and an HTTP flood attack [34], in order to see if the app could handle a load a few orders of magnitude higher than a trivial or expected amount. However, the system withstood them without any issues.

5.5 Acceptance Testing

In order to perform our acceptance tests, we presented our app to potential users: friends and members of our families. The intention was to get live commentary from them while we also monitored their progress and the errors they may make. They were provided with a series of tasks specifically selected to help us determine if the system was intuitive and easy to use from an end-user's perspective.

These tasks were mainly focused to check if our product met the functional and non-functional requirements specified in the requirements analysis report. Tasks were divided in 3 categories, host tasks, attendee tasks, and common tasks. (These are displayed in table 4, Blue for host; Green for attendee; and Orange for common.)

Tests were carried out in 2 parts. First the user would be provided with host tasks, when we, the developers, would act as attendees, then vice versa. These tests gave us important feedback as our main intention was that the user should not require much technical knowledge in order to use our product. Most of them found the application quite intuitive overall but some found it hard to understand some concepts such as the emotion analysis charts or they just pressed incorrect buttons when asked to perform certain task. On average, the tasks were performed as we expected time wise, 1-5 seconds in minor tasks (e.g. liking a comment) and 30 seconds to 1 minute in longer tasks, such as creating an account.

Some of the feedback provided was related to minor fixes we could implement (e.g. changing the colours of the "Yes" and "No" buttons to green and red respectively on the delete button's message box).

Requirement to be tested	**Task given to the user**	**Issues discovered**	**Our response**
Access the system through a browser	Go to the site: http://ballotbox.ml	N/A	N/A
Website should be easy to navigate from mobile devices	Access the site http://ballotbox.ml through a mobile browser	N/A	N/A
Create an account and log in	Attempt to create an account and to log in.	The password alert does not show that the password should contain letters	Modified the password alert to say "Password requires 8 characters including at least 1 number and at least 1 letter".
Create events	Attempt to create an event with the name "Test event" on the date 10/03/2021	N/A	N/A
Share events to attendees or Display a QR code	Attempt to retrieve the QR code from the event or copy the event code	N/A	N/A
Read user feedback	Access the comment wall	N/A	N/A
Observe the evolution of mood and emotions	Attempt to view and interact with the emotion analysis dashboard	Some users did not find the graphs very intuitive	Added a help tooltip that explains polarity and emotion charts
Create personalized polls and see results	Attempt to create a poll of type Free text with prompt "Test Question" and view poll responses	Some hosts found it hard to find the polls' answers	Adjusted the pointer to display a hand icon when pointing to a created poll
Be anonymous	Attempt to change alias	The alias box does not close automatically	Fixed the "change alias" box by closing it once the user enters their new alias
Join an event	Given an event code, attempt to join the event	Links to events that do not exist breaks the website	Added a "404 error page" that shows up if the entered event does not exist
Submit direct feedback	Attempt to write a comment on the comment wall	Some users found it hard to send comments on mobile devices	Adjusted the default mobile keyboard layout used by the website to contain a return button
Use emojis as "mood" feedback	Attempt to write a comment on the comment wall using emojis	N/A	N/A
Respond to polls	Attempt to respond to the poll named "Test Poll"	N/A	N/A

Table 4: A curated list of performed acceptance testing

6 Evaluation

6.1 Success

The best way to evaluate the success of a product is to compare the final prototype with the given requirements. Also, it is advisable to compare it with the other similar products. The prototype system complies with all relevant requirements in a reliable and responsive manner. Moreover, the prototype is user-friendly and suitable for non-technical users, as it does not require prior knowledge or training. (tables 5 and 6)

Project Specification	**Proof**
Allow a user to set up an event for a particular session, series of workshops, project etc.	The event creation page.
This event should be viewable by attendees or team members.	Anyone who has received the code can join.
Attendees or team members can provide live feedback, possibly using templates.	Attendees can write comments and respond to polls.
Attendees or team members can choose to be anonymous when submitting feedback.	Attendees can choose not to have an alias.
Attendees or team members can also provide a "mood" and context for their feedback.	Attendees can use emoji to express their mood.
Hosts can see feedback live as it is submitted.	The host can see the submitted comments in real life.
Provide hosts with analytics on their feedback such as general sentiment, sentiment over time, repeated requests etc.	The emotion detection dashboard.
Hosts can edit templates to be specific to their event.	Hosts can create and customise polls on the fly.

Table 5: Evaluation of the functional requirements

Non-Functional Requirement	**Proof**
Your system should be suitable for non-technical users	Validated through acceptance testing.
Your system must remain responsive.	The system remained responsive in all tested scenarios.
The system should be intuitive and require little to no training.	Validated through acceptance testing.
The system should remain up-to-date.	Validated through system testing.
The performance of your system must be validated using appropriate techniques.	We have performed unit testing, API testing, system testing and extensive stress testing.

Table 6: Evaluation of the non-functional requirements

Two of the original requirements listed in the analysis document were updated such that they were either removed or designated to be a stretch goal. Firstly, we originally planned to allow sorting of comments by recency OR number of likes, however in the end we decided it would be best to sort only by recency instead of likes. We justified this decision by discussing the concept of "live" feedback. Certain comments that were posted at the beginning of an event and garnered a lot of likes may no longer be relevant at the end of the event. In comparison, the most recent comments directly reflect on the current state of the event, hence they will be more influential to the host.

We also decided to remove the requirement stating that there should exist an emoji suggester/recommender such that attendees were prompted to use emoji and were shown predictions of emoji they may want to use. Having analysed the impact of implementing this feature and the actual utility of the feature itself, we decided it was no longer necessary for our application. We decided instead to focus on improving the capabilities of the text learner aspect of the machine learning model, instead of solely relying on an emoji learner. As well as this, we realised that suggesting emoji based on an attendee's previous comments or currently typed text provides no benefit to actually obtaining feedback. In fact, this feature is already implemented by certain OS keyboards such as Apple's [35]. Analysing each previously typed comment for every attendee would greatly increase the work of the machine learning model without providing much utility, especially considering how mood can be extremely volatile over the course of an event.

The overall robustness of the application has been validated through the harsh stress testing scenarios we have gone through, while the reliability of the system is guaranteed by the virtualisation software we are using. Nonetheless, we did not take it for granted: we deployed the system on a cloud virtual machine and used it for testing and evaluation purposes for two weeks. Excluding the periods of maintenance, the total downtime of the website was zero, proving the reliability of the system.

In addition, the application is very easy to scale:

- functionality-wise: we use modern frameworks, based on dynamic programming languages, which allow for quick iterations and speedy implementation of new features
- availability-wise: in case the number of users grows quickly, the system can keep up using the following techniques - allocate more resources to the VMs containing the most stressed components (vertical scaling), spin up more VMs with the most stressed components (horizontal scaling), use MongoDB's sharding feature [36] (horizontal scaling).

Regarding the teamwork and management, all members of the team had well-established roles. However, if one member of the team was unable to complete his tasks, the other members immediately took that work until the specific member was able to return, which helped the project to progress much faster. In retrospect, communication is the key to success, an attribute which our team did not lack.

6.2 Limitations

6.2.1 Backend

The system might become unstable when huge amounts of data are being fetched through some GET endpoints (for example if there are gigabytes of comments). This issue could be fixed by implementing a pagination system, however, as a scenario that would break the app through this means is highly unlikely, the development resources have been used to implement new features and fix higher priority bugs.

Another limitation has appeared as a result of a poor design choice: the primary key of the attendees has been chosen to be the access token, instead of an independent ID. At first, it seemed to be a good choice, because it reduced the number of fields in the model and any information about the user could be retrieved using a single query (as the access token is always in the HTTP request, a database interrogation to find out the ID associated to the user was not needed). However, because of this, endpoints that return other users' comments, likes and responses required extra attention in order not to leak any access tokens.

The last limitation discussed in this section relates to an issue found during the system testing phase: a very large amount of comments per unit of time might significantly throttle the app. This is because the comment sentiment analysis routine, which is resource heavy, is performed synchronously together with all the other comment processing logic. The risk can be easily reduced by vertically/horizontally scaling the machines on which the emotion detection container runs on. Nonetheless, a full fix would require a rewrite of the entire business logic as follows: the emotion detection component should independently look into the database for comments that haven't been processed yet and run the algorithm on them; the backend component should disregard the comments that haven't been analysed yet when computing the data for the emotion dashboard. However, this would increase the complexity of the system dramatically.

6.2.2 Frontend

Limitations for frontend mainly involved compatibility issues. During the development process, we found it difficult to maintain and test our product on different browsers and viewports. In order to be fully compatible, the app must function and look good on phones, tablets and desktop computers of all different sizes. As well as this, all components and styling must be compatible on a range of different browsers, such as Chrome, Safari, Firefox and Edge. To assess this we used LambdaTest [37], a website that runs VMs for different browsers. However, we were unable to reliably test the website on every browser without incurring costs. Furthermore, installing and running every browser on every display size would be very time-consuming.

Overall we did manage to verify the tests on the 3 most used browsers: Chrome, Safari and Firefox [38] and on some common viewport sizes (average desktop, tablet, mobile) using Chrome's inbuilt developer tools.

Ideally, we would like to have used a common cross-browser testing tool such as BrowserStack [39] and verify the application on all browsers and their versions to ensure full cross-compatibility.

As the frontend is heavily dependent on external libraries, our app's integrity relies on the upkeep and maintenance of said libraries. If one of these libraries failed, we would potentially have to rebuild most of the application using a different library or bespoke components. If given a greater timescale it could have been feasible to create all components from scratch however this would add unmatched levels of complexity and would not be worth carrying out especially for a prototype. The likelihood of a library such as React Bootstrap becoming abandoned is very low, and further the potential switch to a similar frontend library such as Ant Design or Material UI [40][41] would not be too difficult to execute.

6.2.3 Emotion Detection

We have only used the 5 of the most frequently felt emotions. It is certainly possible to include more emotion categories such as “surprise”, “faith”, etc. However because of the lack of data on the other less frequently felt emotions, using those data points would have led to a worse prediction accuracy. Moreover, it would contradict the frontend’s design principles to include a greater range of emotions (Hick’s Law) hence in some ways only choosing 5 emotions was beneficial. In the future we could scavenge for better datasets available with a range of emotions which would be used with the same machine learning model, as the infrastructure is highly expendable.

The best way to improve the model accuracy further would be to reobtain the entries of the linear transformation matrix using a massive dataset. If DeepMoji is also updated to increase in its ability to generate more and more realistic probability distributions then we may assume realistic frequency over emotions in the dataset and perhaps render our model state of the art.

6.3 Further Work

During the development process we had a brainstorming board, where we wrote any potential features that we could integrate into the app. Some of them didn’t make it into the final product, because of time constraints, technology constraints, or plainly because they were not suitable. Below we have enumerated a curated list of these ideas.

- make a fully fledged mobile app: we have not done this because the added value is close to nil. Nonetheless, this would be easily done, as we used React for frontend, which can be ported to React Native [42] in order to create executables that can run on most modern operating systems.
- assign anonymous users a colour: we thought that it would be useful to know if two anonymous comments come from the same person. We could solve this issue by displaying the comments from each user using a different unique colour, however this raises privacy concerns.
- delete/edit comments: useful features, but massively increases implementation complexity and makes the UX less intuitive.
- edit event details: useful feature, but can be emulated to a certain extent using the delete functionality. Skipped because other features had higher priority.
- analyse the subjectivity of the attendees: add another graph, which shows how subjective are the comments entered by the attendees, which would be easy to implement because extensive research has been done in this area [43]. Skipped because it was not clear how useful this feature would be.
- end of event analytics: show a summary containing useful stats at the end of an event. Skipped because other features had higher priority and line chart covers whole event.
- dark mode: let the user choose a different theme with darker colours. Skipped because it does not add much value to the product.
- accessibility features: such as colourblind mode, image captioning, ARIA attributes [44]. They are essential in every website, though were skipped as they would require a lot of time researching and it is unclear how we would be able to test them.
- support for different languages: we were limited by the ability to perform emotion analysis on languages other than English.
- language filter: useful, but very tricky to implement, skipped because of time constraints.
- spam protection: same as the above.
- monetisation: come up and implement a business model that would generate revenue through the app, skipped because it does not follow the spirit of the assignment.

7 Conclusion

7.1 Summary

All things considered, we deem this project a success. We have created a full-stack application that efficiently allows event attendees to provide live feedback to a host, fulfilling all necessary requirements set out at the beginning of the project. We worked extremely well as a team, with a clear and concise plan from the start and frequent detailed communication between team members. Overall, the project remained sufficiently on-schedule all throughout the project and was never rushed, showing that our planning was thorough and effective. Hence, the finished product displays and embodies our combined technical abilities as software engineers.

7.2 Our Thoughts

This experience has allowed us to greatly develop our team working, analytical and creative abilities. We have been put face-to-face with challenges and complex situations that could have hindered us, but we were always able to overcome these situations. This project was an excellent learning experience for many reasons, such as providing an example of what it is like to work as a part of a team in a project that simulates a real-life scenario in software engineering, something that we will likely face in our future careers.

References

[1] Slido — Audience Interaction Made Easy. `sli.do`. Accessed: 2021-03-12.

[2] Vevox — Making virtual meetings unmissable! `vevox.com`. Accessed: 2021-03-12.

[3] Kahoot — Make learning awesome! `kahoot.com`. Accessed: 2021-03-12.

[4] Max Rehkopf. What is a kanban board? `atlassian.com/agile/kanban/boards`. Accessed: 2021-03-12.

[5] Empowering App Development for Developers — Docker. `docker.com`. Accessed: 2021-03-12.

[6] Overview of Docker Compose. `docs.docker.com/compose`. Accessed: 2021-03-12.

[7] FastAPI. `fastapi.tiangolo.com`. Accessed: 2021-03-12.

[8] Swagger UI — REST API Documentation Tool. `swagger.io/tools/swagger-ui`. Accessed: 2021-03-12.

[9] Python.org. `python.org`. Accessed: 2021-03-12.

[10] MongoDB — The most popular database for modern apps. `mongodb.com`. Accessed: 2021-03-12.

[11] pydantic. `pydantic-docs.helpmanual.io`. Accessed: 2021-03-12.

[12] RabbitMQ — Messaging that just works. `rabbitmq.com`. Accessed: 2021-03-12.

[13] OAuth. `oauth.net`. Accessed: 2021-03-12.

[14] React Bootstrap. `react-bootstrap.github.io`. Accessed: 2021-03-12.

[15] Bootstrap — The most popular HTML, CSS, and JS library in the world. `getbootstrap.com`. Accessed: 2021-03-12.

[16] React Bootstrap Documentation — Components. `react-bootstrap.github.io/components/alerts`. Accessed: 2021-03-12.

[17] Recharts — A composable charting library built on React components. `recharts.org`. Accessed: 2021-03-12.

[18] D3.js — Data-Driven Documents. `d3js.org`. Accessed: 2021-03-12.

[19] Twemoji. `twemoji.twitter.com`. Accessed: 2021-03-12.

[20] Laws of UX. `lawsofux.com`. Accessed: 2021-03-12.

[21] William E Hick. On the rate of gain of information. *Quarterly Journal of experimental psychology*, 4(1):11–26, 1952.

[22] Jakob Nielsen. End of Web Design. `nngroup.com/articles/end-of-web-design`, Jun 2000. Accessed: 2021-03-12.

[23] Progressive Web Apps. `web.dev/progressive-web-apps`. Accessed: 2021-03-12.

[24] Lighthouse — Tools for Web Developers — Google Developers. `developers.google.com/web/tools/lighthouse`. Accessed: 2021-03-12.

[25] Progressive Web App Checklist. `web.dev/pwa-checklist`. Accessed: 2021-03-12.

[26] React — Code-Splitting. `reactjs.org/docs/code-splitting.html`. Accessed: 2021-03-12.

[27] Bjarke Felbo, Alan Mislove, Anders Søgaard, Iyad Rahwan, and Sune Lehmann. Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[28] pytest: helps you write better programs — pytest documentation. `docs.pytest.org/en/stable`. Accessed: 2021-03-12.

[29] Pytest plugin for measuring coverage. `pypi.org/project/pytest-cov`. Accessed: 2021-03-12.

[30] Create React App. `create-react-app.dev`. Accessed: 2021-03-12.

[31] Jest — Delightful JavaScript Testing. `jestjs.io`. Accessed: 2021-03-12.

[32] React Testing Library. `testing-library.com`. Accessed: 2021-03-12.

[33] Slowloris DDoS Attack. `cloudflare.com/learning/ddos/ddos-attack-tools/slowloris`. Accessed: 2021-03-12.

[34] HTTP Flood Attack. `cloudflare.com/learning/ddos/http-flood-ddos-attack`. Accessed: 2021-03-12.

[35] Use emoji on your iPhone, iPad and iPod Touch — Apple Support. `support.apple.com/en-gb/HT202332`. Accessed: 2021-03-12.

[36] MongoDB Manual — Sharding. `docs.mongodb.com/manual/sharding`. Accessed: 2021-03-12.

[37] LambdaTest — Cross Browser Testing Cloud. `lambdatest.com`. Accessed: 2021-03-12.

[38] StatCounter — Browser Market Share. `gs.statcounter.com/browser-market-share`. Accessed: 2021-03-12.

[39] BrowserStack — App & Cross Browser Testing Platform. `browserstack.com`. Accessed: 2021-03-12.

[40] Ant Design — The world's second most popular React UI framework. `ant.design`. Accessed: 2021-03-12.

[41] Material-UI: A popular React framework. `material-ui.com`. Accessed: 2021-03-12.

[42] React Native — A framework for building native apps using React. `reactnative.dev`. Accessed: 2021-03-12.

[43] Bing Liu et al. Sentiment analysis and subjectivity. *Handbook of natural language processing*, 2(2010):627–666, 2010.

[44] ARIA — Accessibility. `developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA`. Accessed: 2021-03-12.