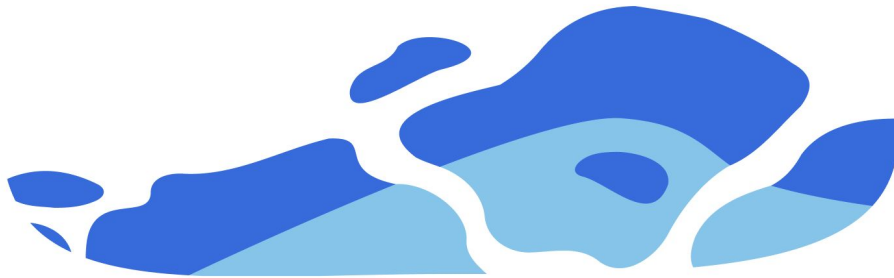


CS3216 Assignment 3



queuing made serene.



Group 8: *whoosh*

Ho Hol Yin
Liu Zechu
Ong Yan Chun



Contents

Accessing the web app	3
Milestone 0	4
Milestone 1	4
Milestone 2	5
Milestone 3	7
Milestone 4	8
Milestone 5	8
Milestone 6	9
Milestone 7	10
Milestone 8	11
Milestone 9	12
Milestone 10	13
Milestone 11	13
Milestone 12	14
Milestone 13	16
Milestone 14	20
Milestone 15	22
Appendix	23



Accessing the web app

As a restaurant manager

You may access the restaurant-side website via <https://hoholyin.github.io/whoosh/#/> and then click “get started”. Afterwards, you may register a new account or log in with an existing account. We have made an account populated with several queue groups for testing purposes.

Test account email: dintaifung@mail.com

Test account password: dintaifung

As a diner

You may scan the QR code below using your phone camera, or access the website to join the queue via https://hoholyin.github.io/whoosh/#/joinQueue?restaurant_id=5



The appendix contains some things of note about the implementation.



Milestone 0

Currently, there are three main types of queuing systems: 1) Physical queues, 2) Restaurant staff record phone numbers and manually call diners, 3) Ticketing system where diners input their information and wait until the ticket number is called.

For diners, physical queuing is tiring and a waste of time. The phone number waiting list provides no feedback as to their waiting time. The ticketing system requires them to wait at the restaurant or frequently check the machine so as to not miss their entry. For restaurants, it is troublesome to manage a physical queue due to designating extra physical space and utilising extra manpower. Due to the COVID-19 pandemic, restaurants need to enforce social distancing, causing the queues to take up a lot more space. The phone number waiting list is time-consuming and inefficient due to the process of calling diners one by one. It is possible that they make a mistake in recording information as well. The ticketing system requires a huge investment into a physical machine for each outlet.



Milestone 1

Our application takes a diner or a group of diners into a queue upon their arrival, and maintains a digital representation of the queue. There are two interfaces, one for diners to visualise their position in the queue and estimated waiting time, and one for restaurant staff to manage the queue.

The quality of availability of mobile cloud applications is ideal for our user needs. The act of queuing is something one-off and spontaneous. No one wants to download an application just so they can queue at a restaurant. Because the diners can access the app at any time without having to make any commitments to downloading, having it on the mobile web is ideal.

Our application is dependent on the productivity that mobile cloud applications bring—we exactly want the queuing to become something that can be maintained on the go, and for diners to be notified when they can enter the restaurant. The peace of knowing that you can check your position any time while you are, for example, looking at other stores in the same shopping mall makes the queuing process much more pleasant.

Our application must work on all types of mobile devices that people use. We cannot deny a diner from entering a queue just because they don't own an iPhone. It is necessary that our application is device independent, which makes a mobile web application the perfect fit.

Furthermore, because there are very low hardware requirements, any device that can access the internet will likely have no issues.

Regardless of its size and location, any restaurant can use our application as all data will be handled by our cloud database. This feature creates a unified platform for all types of restaurants to manage queues conveniently without any additional investments.



Milestone 2

Our target users are walk-in restaurant diners and restaurant managers.

Restaurant diners will be expected to use the application as long as a restaurant that they wish to eat at is using our application as its waitlisting system. We plan to use the scanning of a QR code at the entrance of the restaurant to allow diners to access our application and join a queue, which is something that many Singaporeans are accustomed to by now due to measures such as SafeEntry.

Our business model will involve a free subscription and a premium subscription. The free subscription will have a limited access to the core functionalities of our application, for example the number of diners per hour may be limited. The premium membership will have possible extensions with additional features such as customisable messages to diners, data analytics and displaying promotions to diners. If we can persuade investors of the commercial potential of our product, we will be able to obtain funding and subsequently direct the capital towards marketing efforts and maintaining our servers.

To promote our application, we will target restaurant managers and convince them of how they can benefit tremendously from adopting our product. *whoosh* solves all the problems of current queuing methods. There is no physical queue. Diners get exact feedback on their waiting time because *whoosh* visualises the actual queue and provides time estimates. Diners can shop or walk around without stress because they will be prompted by the app and by SMS that they are to enter the restaurant. The waitlist management is easy for restaurants because diners enter their own data and restaurants just need to alert the diners to come. *whoosh* also does not require monetary investment into purchasing any machines or devices.

In particular, we will emphasise the fact that *whoosh* will reduce the workload of the restaurant staff and keep diners informed of the status of the queue. In addition, we will also share that *whoosh* can potentially keep the queuing diners from becoming dissatisfied and therefore leaving the queue. With all these benefits to gain and very little cost of adoption, we believe that a large number of restaurants will be willing to use *whoosh*.

Here are some of the marketing materials we have created:

A white flyer with the word "whoosh" in blue lowercase letters, accompanied by three small yellow stars. Below it, the tagline "queuing made serene." is written in blue. A blue QR code with a black 'W' in the center is positioned on the right. The bottom of the flyer features a stylized blue and white wave pattern.

Managing a physical queue is difficult. Why not make it digital?

No more wait.

whoosh makes queuing a breeze, both for you and your customers. You never have to manage the queue again. Your customers will have a great time and be automatically notified to return.

Get Data

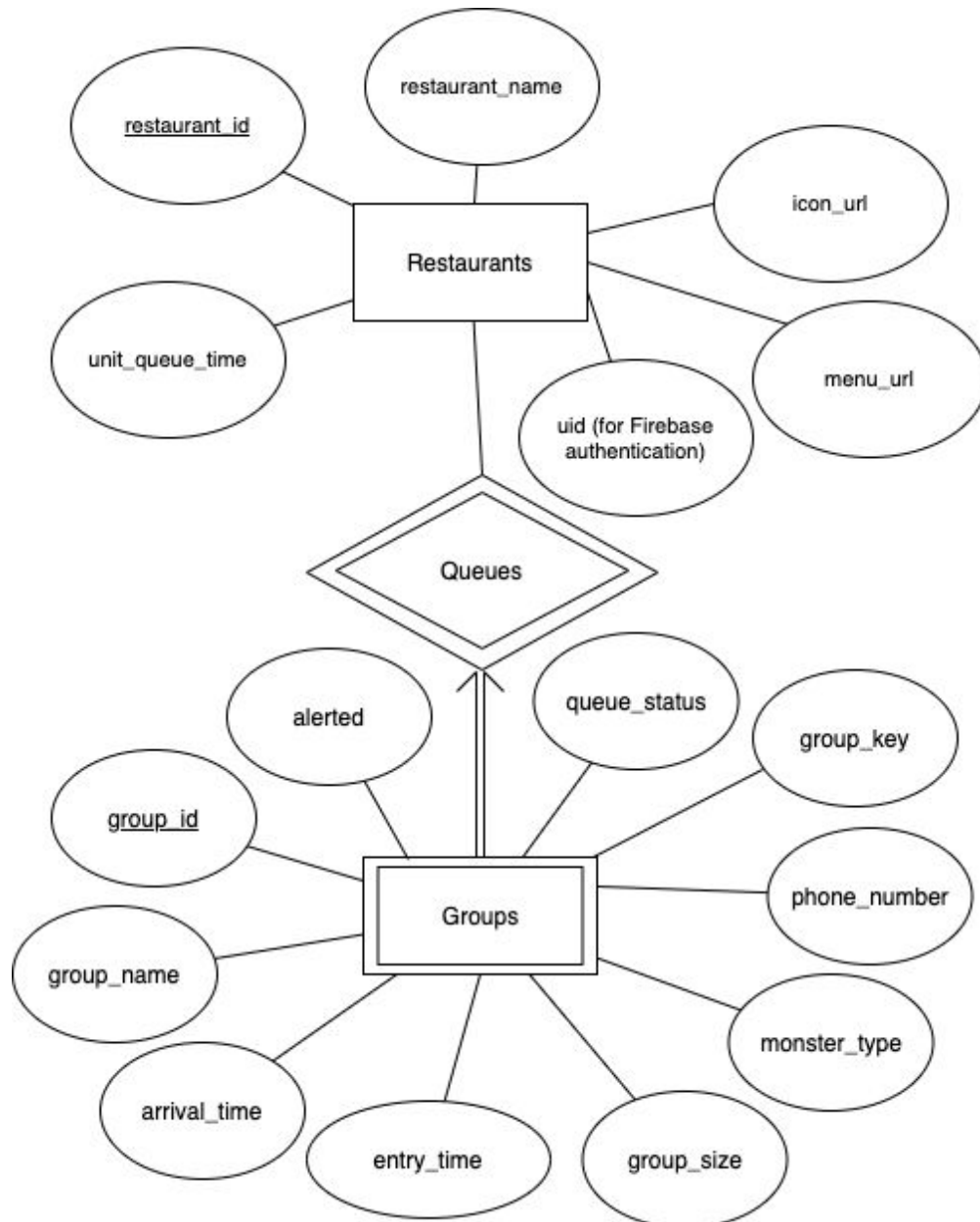
With whoosh, you'll have real data about your waiting times and customer footfall. And data is power. You'll know exactly what's working well and what's not working out.

visit gg.gg/whooshapp for more



Milestone 3

Please refer to the following image for our ER diagram.





Milestone 4

The alternative that we considered is GraphQL, which is an increasingly popular approach to design client-server APIs. The most significant difference between GraphQL and REST is that instead of being limited to a fixed set of API endpoints defined by a REST API server, a client can query for just the right amount of information needed. A query in GraphQL is written in exactly the same format as the entities and attributes that the client would like to receive. In contrast, a REST API client has to use one of the HTTP methods to query a pre-defined endpoint.

GraphQL has several advantages over REST. Firstly, the client does not have to be limited by what the server provides. In REST, sometimes the client either has to fetch a lot of redundant information in order to obtain a subset of it, or the client has to request multiple times to obtain several resources (e.g. a group of users). GraphQL solves this problem by only returning just the required amount. Secondly, schemas in GraphQL represent relationships between real world objects. This characteristic makes it easier to manage complicated entity relationships, as compared to a linear URL in REST APIs.

REST trumps GraphQL in some aspects. Firstly, REST APIs are very widely used and easy to implement. Many development frameworks support REST APIs well. Secondly, the server has strict control over what resources are exposed. It is easier for developers to control what and how much third party clients can request from the server. Thirdly, REST builds on HTTP specifications which facilitate caching. GraphQL does not follow these specifications.

We eventually decided to adopt REST to design the interactions between our client and server due to the following reasons. First, our entity relationships are relatively straightforward, involving just restaurants and queue groups. GraphQL's advantage in managing complex entity relationships does not apply to our application. Second, our REST APIs can be designed in such a way that only the exact amount of information needed is returned to the client. GraphQL's advantage in returning "nothing more nothing less" can also be achieved by REST in our application. Third, REST APIs are easy to implement and test. Using GraphQL would be overkill here.



Milestone 5

Please refer to this link for our REST API documentation:

<https://whooshserver.docs.apiary.io>

Our API design conforms to the REST principles in the following ways.

- First, all the exchanges between the server and the client follow the JSON format.
- Second, all the endpoints are nouns, where a plural noun signifies a collection of resources.
- Third, the URLs are arranged in hierarchical order that reflects how actual resources are related. For example, the endpoint to GET a queue group's information is `/restaurants/:restaurant_id/groups/:group_id`.
- Fourth, proper HTTP status codes are returned to inform the client. For example, a 201 code is returned together with the resource when a new resource is created. 404 is returned when the resource in the HTTP query does not exist.
- Fifth, we implemented filtering via the query parameters so that the client only receives relevant information. For example, the endpoint `/restaurants/:restaurant_id/groups?status=:queue_status` allows the client to only get queue groups of a certain status (e.g. currently in the queue).



Milestone 6

Query 1:

```
CREATE TABLE restaurant${restaurant_id} (  
    group_id SERIAL PRIMARY KEY,  
    group_key CHAR(8) NOT NULL,  
    group_name VARCHAR NOT NULL,  
    arrival_time TIMESTAMP WITH TIME ZONE NOT NULL,  
    entry_time TIMESTAMP WITH TIME ZONE,  
    group_size INTEGER NOT NULL,  
    monster_type VARCHAR NOT NULL,  
    queue_status INTEGER NOT NULL,  
    phone_number CHAR(8),  
    alerted INTEGER NOT NULL,  
    CONSTRAINT check_phone CHECK (phone_number NOT LIKE  
'%[^0-9]%') );
```

Every time a new restaurant is registered in our system, a new table containing information about this restaurant's queue will be created. This table has the above schema. `group_id` is the primary key to identity a group. `group_key` is a random string of 8 characters associated with a particular group, to prevent diners from another group from accessing the queue information of this group by simply changing the `group_id` parameter in the URL. `group_name` is a randomly assigned name related to food items. `arrival_time` and `entry_time` refer to a group's

time when they start queuing and when they enter the restaurant respectively. `monster_type` is a string that encodes information about a group's avatars. `queue_status` is an integer that indicates whether a group is currently in the queue (0), has entered the restaurant (1) or has left the queue (2). `alerted` is an integer that indicates whether a group has been alerted (1) to come to the restaurant when it is their turn. Its default value is 0 (not alerted).

Query 2:

```
SELECT * FROM restaurant${restaurant_id} WHERE queue_status = ${queue_status}
```

This query is used to obtain all the queue groups of a certain queue status in a particular restaurant. When the status code is 0, this query will return all the groups in the current active queue. This is useful when generating the virtual queue visualisation for diners and restaurant managers.

Query 3:

```
INSERT INTO restaurant${restaurant_id}
VALUES (DEFAULT, '${group_key}', '${group_name}', NOW(), NULL,
${group_size}, '${monster_type}', ${queue_status}, '${phone_number}',
0) RETURNING group_id;
```

This query is used to add a new customer group into a restaurant's queue. Since we use `SERIAL` as the data type for `group_id`, we insert `DEFAULT` into the first column so that the database will generate an incrementing ID automatically. This ID is then returned through the `RETURNING` statement. `NOW()` generates the current time, which indicates the arrival time of this customer group.

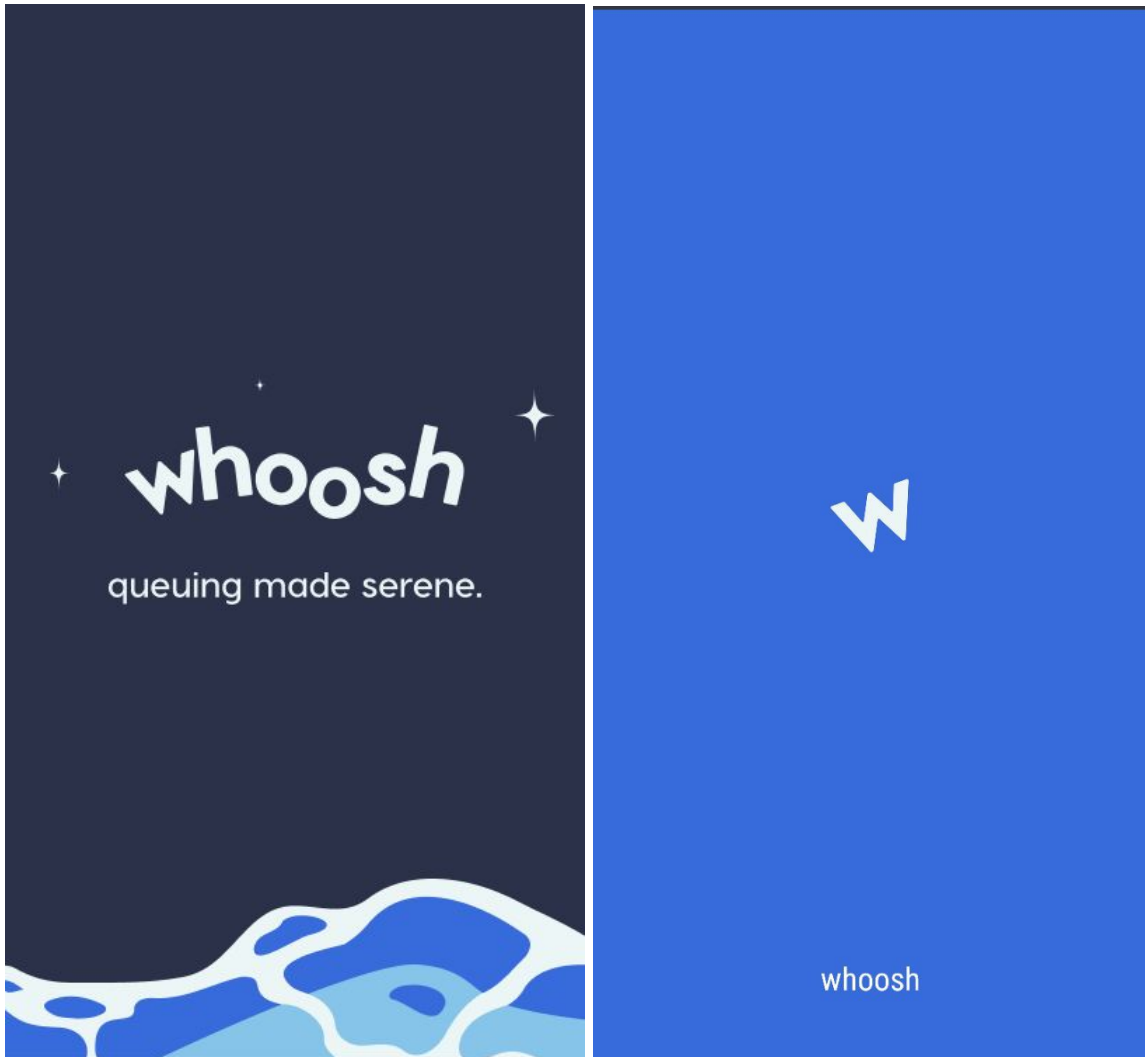


Milestone 7

Icon



Splash Screen



Milestone 8

Although styling in Flutter does not use explicit CSS, we have nonetheless adapted from the CSS methodologies to allow them to become applicable in the context of Flutter. For instance, where components share the same color, we abstract the common color into a Commons file so that all widgets (that's what they call the UI components in Flutter) will share this same property. If there are any changes to the color, the changes will be uniformly applied to all related widgets. By abstracting all the style-related variables into a Commons file, we will have a higher level understanding of the style of the widget.

For example, compare the difference between a widget with the color property of `Color(0xFF9A0000)` versus `Commons.whooshErrorRed`.

Consider also, `Image.asset('images/static/restaurant_menu_button.png')` versus `Commons.restaurantMenuButton`.

Abstraction allows us to give a higher level meaning to the variables that we are assigning, which potentially makes our code much more maintainable.

We also made use of themes in order to allow a group of screens to be styled in a similar way with shared background color, icons, and so on. In particular, we have two themes, namely the `restaurantTheme` and the `queueingTheme`, which are for the screens responsible for the restaurant managers to access and the screens the diners will see respectively.

Let's compare this to classes in CSS. Classes are a form of abstraction that tends to hide the actual details of the component's style in the `.css` file. By using themes, we abstract the actual details of the styling to make the codebase that is responsible for styling easy to maintain and reuse.



Milestone 9

We identified three main communication channels in our application:

1. Users (including restaurants and diners) → *whoosh* frontend website hosted on GitHub Pages
2. *whoosh* frontend website hosted on GitHub Pages → *whoosh* backend server hosted on Heroku
3. Users (restaurants) → Firebase Authentication (through *whoosh* frontend website).

To ensure all communications are encrypted, all of these three channels need to be SSL-secured. For (1), our website is hosted on a `*.github.io` domain, for which a free SSL certificate is provided by GitHub servers. For (2), although Heroku does not provide a free SSL certificate for a **custom domain** server hosted on a Free-tier dyno, our backend server uses Heroku's `*.herokuapp.com` domain which is automatically [SSL-certified](#). For (3), Firebase servers are SSL-certified as well. Therefore, all communications within our application are protected via HTTPS.

If a user tries to access our frontend website using HTTP, he/she will be redirected to the HTTPS version as the “Enforce HTTPS” option has been selected on our GitHub Pages. Also, we used “express-sslify” in our Node.js server to force any attempts at HTTP requests to be communicated over HTTPS instead.

Three best practices for adopting HTTPS:

1. We use robust security certificates from reliable Certificate Authorities through our hosting platforms (GitHub Pages and Heroku), which also provide reliable technical support. For example, the certificate of our frontend website is DigiCert SHA2 High Assurance Server CA, which is from one of the top SSL certificate providers.
2. We do not have mixed content served via HTTP and HTTPS. All our resources are served over HTTPS.
3. Our sites support HSTS (HTTP Strict Transport Security), which informs the browser to only request HTTPS pages, even if the user attempts to connect to the HTTP version. We achieved this by adding the response header `Strict-Transport-Security: max-age=604800` to all our HTTP responses.



Milestone 10

Our app stores a cached version of the queue, waiting time, as well as information about the restaurant they are queuing at. This means that even when the user disconnects from the Internet temporarily, for instance, switching from Wi-Fi to 4G, they will still be able to view information about the queue, the estimated waiting time and the restaurant. The app then seamlessly tries to connect back to the backend to retrieve the latest information once the connection is restored before updating the user's device with the newest information.

In addition, we have an SMS notification function to notify diners when they are due to enter the restaurant. As long as the diner has successfully scanned to enter the queue, their phone number will be recorded on the restaurant manager's waitlist. Even if they lose connectivity and will not know if they are next to enter, *Whoosh* will automatically send an SMS to notify them.



Milestone 11

Compared to session-based authentication, token-based authentication has several advantages. Firstly, token-based authentication is stateless on the server side. The user state is stored on the client application, and the client authenticates itself in each HTTP request by including the token in its request header. This removes the need for the server to store any user information and simplifies server design. Secondly, JSON Web Tokens (JWTs) are lightweight and can be easily implemented on different platforms. However, JWT has a slightly bulkier payload than a session ID. In contrast, session-based authentication stores the user state on the server's memory and the session ID is stored in a cookie on the user browser. The user then sends the session ID on subsequent requests to the server which then compares it against the stored records. This makes authentication stateful on the server side.

We decided to adopt token-based authentication in our client-server communications using JWT. We felt that the stateless nature of token-based authentication aligns with the principle of REST architecture. In REST architecture, the server does not store any state about the client session. The client should send all necessary information to the server without relying on stored data. In other words, each request from the client to the server should be self-contained. This is exactly what JWT does, as the only information required for authentication is contained in the client's request header. Moreover, using token-based authentication simplifies our server design. We also will not have to worry about server memory issues when a large number of users use our application.

Please note that JWT is used for communications between the frontend website and the backend server. From the users' perspective, restaurant managers' authentication system is managed by Firebase Authentication instead, whereas diners do not require authentication to join a queue.



Milestone 12

We have chosen Flutter and its accompanying libraries in developing *whoosh*.

We knew that we wanted to have animations for the monsters. We wanted to make the queuing experience more interesting and interactive. Animations bring the monsters to life, adding more dynamism and whimsy. Flutter allows us to produce high-quality animations with a near-native performance. Animations would be difficult to implement with another framework such as React.

Flutter allows us to write a single code base which may potentially be used on other mobile platforms such as Android and iOS. Although we intended our application to be used primarily on the web, we may consider implementing a mobile version for restaurant managers to manage the waitlist more conveniently. Using Flutter speeds up this transition since we won't have to rewrite our code when developing the mobile app in the future. This feature of Flutter is also especially important because our team is lean (we have only 3 members).

Flutter boasts a really useful feature for fast development called Hot Reload. When running a Flutter app locally, we can simply press "r" to apply changes to our code instantly to the running instance, enabling us to view our UI changes and fix bugs in real time. This feature speeds up our development tremendously especially when wanting to experiment with different UI styles and using a lot of trial and error.

Another important consideration for choosing Flutter over other tools is that Flutter was initially designed for mobile apps. As a result, its UI elements have a native look and feel. This

characteristic makes Flutter an appropriate option since we wanted our application to feel like a native mobile app while exploiting the convenient nature of a web app.

However, Flutter does come with some disadvantages, the most significant of which is that Flutter Web is still in its nascent stage. It was only launched one year ago. Some features may not be stable yet and most of the online support tends to focus on the mobile version. In comparison, popular web development tools such as React have huge community support and access to a wide range of plugins and libraries to simplify the work. Nevertheless, we decided to go ahead with Flutter since most of the user interactions only require basic UI elements and transitions. It can also integrate well with Firebase Authentication using FlutterFire plugins.

These are some of the principles that we have adhered to:

Keep calls to action front and center

The main call-to-action on the main page is “get started”. It’s top on the page, right after the *whoosh* logo. In general, all buttons are designed in a mobile-friendly way. They are large, obvious, and clearly indicate the function of the button. Look at Milestone 13 for a detailed breakdown of the UI design of buttons. This principle can be seen on all screens.

Keep menus short and sweet

Menus are short and useful. They are composed of only the necessary components. The navigation menu for restaurant management has as few tabs as necessary. Another example is the waitlist screen which has a short menu for the manager to interact with the specific queue groups. There are only three actions which are all necessary: alert, confirm arrival, and kick out. This principle can be seen in particular in the menus on these two screens.

Optimize your entire site for mobile

The site is optimised for mobile. Elements on screen scale based on the device. The site is not optimised for desktops at all. We only offer a mobile version on our website. This restriction was intended because it is unlikely that users would bring a laptop in order to scan a QR code at a restaurant. The site is clean and with large easy-to-navigate buttons. In addition, we only have vertical scrolling to view the queue, which is a natural action for mobile users. This principle can be seen on all screens.

Don't make users pinch-to-zoom

There is no need to pinch zoom on any screen. There is no horizontal scrolling on any of the screens. All the information a diner or restaurant owner might see is clearly displayed at appropriate sizes. For instance, the estimated waiting time which is the most important to a queuing customer, is always the largest in the center of the screen. This principle can be seen on all screens. In particular, the queueing screen shows a focus on vertical scrolling over horizontal scrolling.

Keep your user in a single browser window

Our users are kept on the same browser window throughout the app. This principle can be clearly seen in all restaurant management screens. There is only one deviation from this principle: the viewing of the restaurant menu. This feature was a necessity because we wanted them to visit the restaurant's menu for the purposes of choosing their meal and promotion on the restaurant's end. We intended for diners to venture out of our site in this case such that they might find more information about the restaurant in question.

Avoid "full site" labeling

We have no such labelling of sites or switching of modes because as mentioned, we only offer a mobile-optimised version on our website. As such, users will never be worried that they are missing out on some content or features.



Milestone 13

Diner: Entering a Queue


The most important workflow is the act of entering a queue—the highlight of the web app and the first screen diners see. We wanted to ensure this user flow was efficient, clear, and still entertaining. Once diners scan the QR code at the restaurant, they are immediately brought to the queuing information page. No unnecessary information or delay. The diner only has to key in 2 crucial pieces of information: their phone number, and the number of people in their group. They will then immediately be brought to the waiting in queue page.


We focussed on clarity. The most important detail is the identity of the restaurant they are queueing for. Hence, this is prioritised in the information architecture. The top of the page proudly displays the icon and title of the restaurant, so the user can be sure that they are queuing for the right restaurant. Once the user keys in their phone number, validation is done immediately to ensure that the right phone number was entered. The slider has a clear indication of group size: a big number, and the number of monsters in the illustration changes as well.

We wanted this page as a fun introduction to the application. Hence, one of the highlights of the page is the monster group illustration that changes based on the size of the group. Furthermore, each monster is randomised, so each time a diner

whoosh

you're queueing for

 **Din Tai Fung**



phone number

group size

1

enter the queue

enters a queue they have a different group of monsters. The monsters are also dynamically animated, so it brings the page to life in the short moments that they take to key in their information.

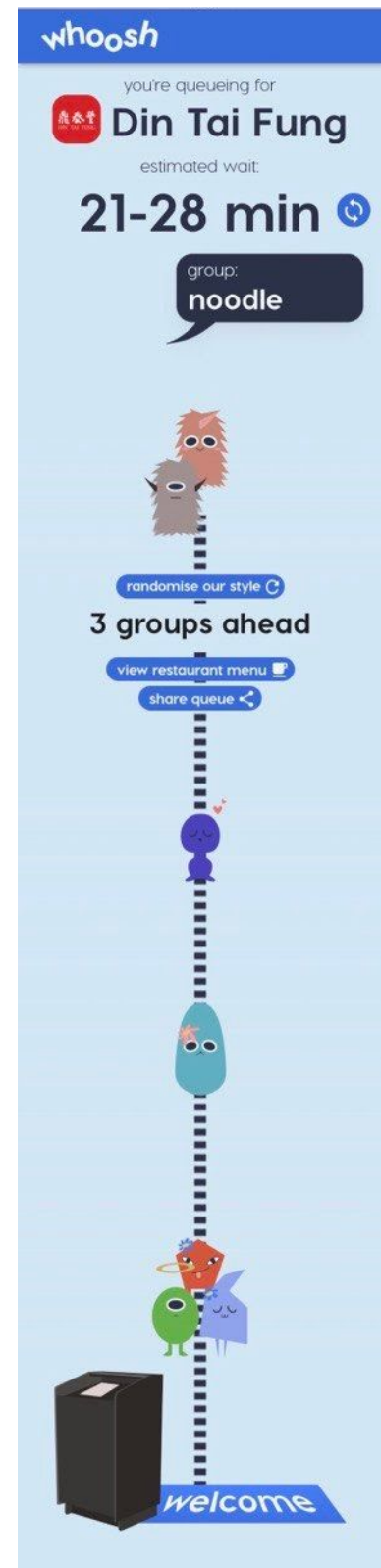
Diner: Waiting in Line

The other half of the workflow for diners is waiting in line. The main things we wanted to ensure for this user flow was clear information on the queue waiting time, offering features not offered by competitors, and being entertaining.

To provide information for the queue waiting time, we again placed the most important information at the top of the screen. First, the restaurant name and icon to reassure the user that they are queuing for the right restaurant. Second, the estimated wait time. Third, the group code to enter the restaurant. Considering Nielsen's second usability heuristic (Match between system and the real world), we decided to have a line of characters in a queue to match the physical queue that they cannot see. Not only does this match the mental model of a queue, but it also provides users with a better understanding of the wait time that they cannot get otherwise. For example, if the user is a single member group, they may expect to be seated earlier than a 5-member group.

We also intended to cover the full suite of options that a diner may want while waiting in line. For one, they are allowed to view the restaurant menu, so that they can decide on their order before even entering the restaurant. This feature has an added benefit for restaurants to advertise their promotions. Another important feature is the sharing of the queue, which allows them to send a link to their friends to also look at the queue status. Finally, there is a sync button to refresh the queue status for peace of mind. Although the queue status is automatically refreshed, it is reassuring for a user to feel in control.

The page was also intentionally made to be entertaining for users. We wanted to make the queueing experience feel as quick as possible. Users can randomise the style of their group. An animation of a cloud expanding provides visual interest when the style is changed. The randomness of the styles keeps users clicking until they find something that represents them. When users look at the queue ahead of them, besides getting information about the waiting time, they get to see how other groups choose to express

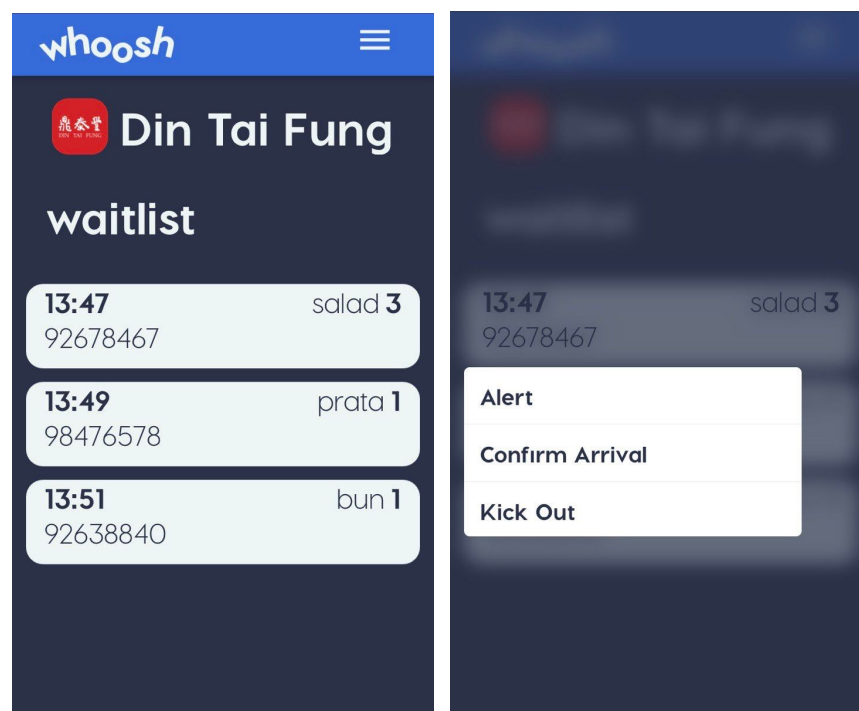


themselves and see the variety of possible monsters. Furthermore, users can interact with their own group of monsters by clicking on them, producing random animations like confetti, rising hearts, and so on. The interest of the page can help to pass the time and enhance the queuing experience.

Finally, this page also facilitates the entering of the restaurant. Once the diner is the next in line to enter the queue, the app will display a message saying 'you're next', which prompts the diner to bring his/her group to the entrance of the restaurant and wait to be seated. In order to gain entry to the restaurant, the diner will either have to show the SMS that is sent automatically by *whoosh*, or show the restaurant staff their group name through the *whoosh* app in order to verify their identity.

Restaurant: Waitlist Management

On the side of the restaurant, the main workflow is handling the waitlist/queue. The key qualities we wanted this workflow to have was first, control, and second, ease of reading.



Control is important because the main point of the page is for waitlist managers to manipulate the queue. Each diner group is represented on a card. On long press, each card has a small menu pop up. Long press was chosen so that no mishaps happen by accidentally dismissing a diner group. The menu has 3 buttons for interaction: Alert, Confirm Arrival, Kick Out. Alert is at the top as it is usually the first interaction the manager will use. Alert sends an SMS to the phone number of the alerted group. This feature is important as diners who are not at the front of the queue could enter first, for example when a table for one opens so a single diner is prioritised. Confirm

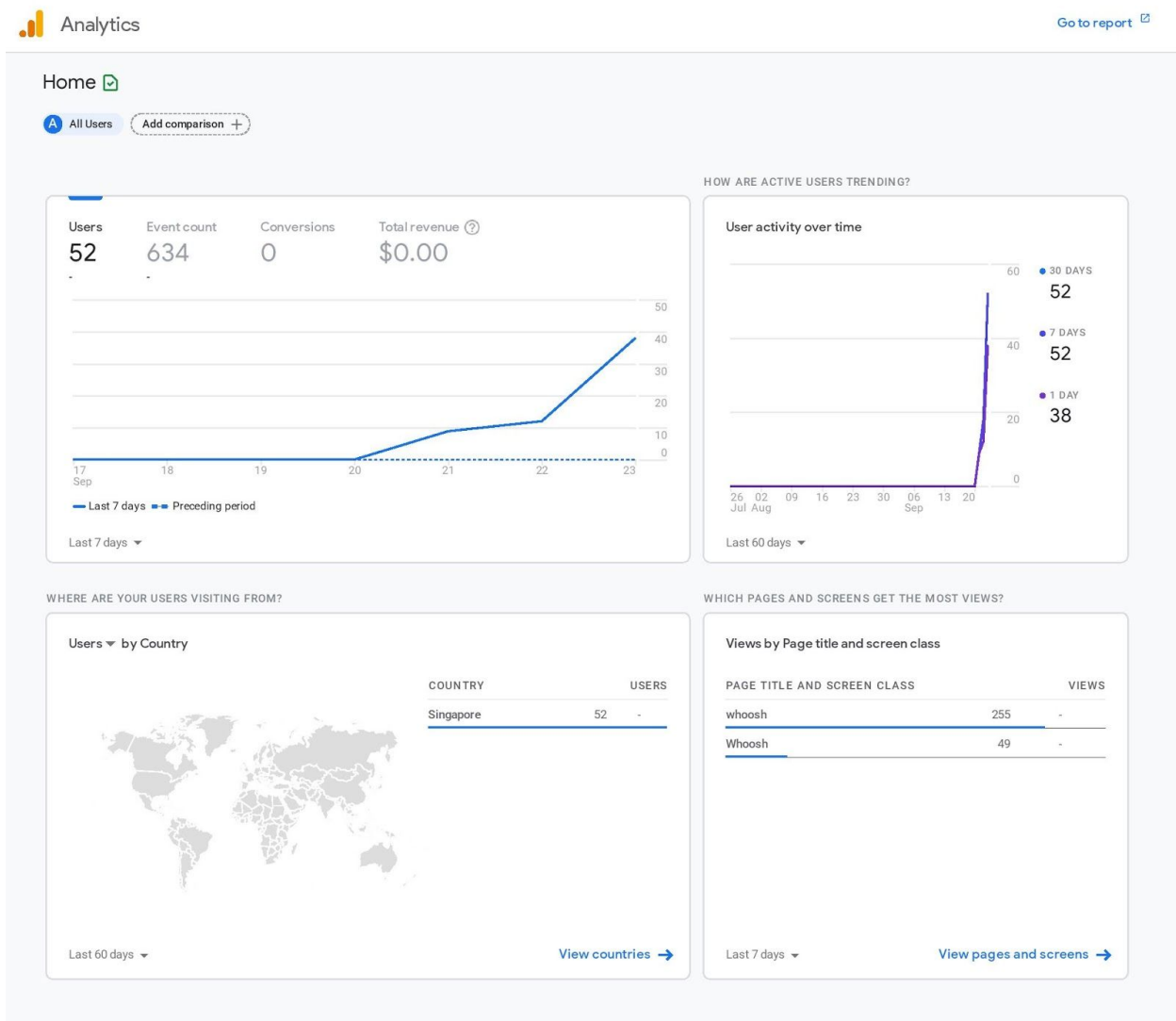
Arrival is used when the group has already entered the restaurant. Finally, Kick Out allows the managers to remove a group that is not turning up. Both Confirm Arrival and Kick Out remove the card from the manager's view as it is no longer relevant to them. A toast is brought up confirming the action that the manager has taken for feedback from the system.

Ease of reading is important because the waitlist managers need to accomplish their task quickly, and many times more diners in the queue correlate with more stressful situations. In order to achieve ease of reading, the cards for each customer in the queue have a clear information hierarchy. The most important information is their time of entering the queue, so that they can manage diner expectations. That time is also what determines the order of cards. This information is bold and presented at the top left. The next most important information is the group size, for table planning. The group size is bold and presented at the top right. Next is the group name, for verification of the group, which is placed beside the group size due to the law of proximity; we want users to associate the group name and size. It is important to note here that we chose group names instead of group numbers. We found that group names tend to be easier to recall than numbers, making it more easily managed by the waitlist managers. Additionally, the group names add a level of whimsy and interest for the diners. Finally, on its own row below the other information is the phone number of the group, which should only be necessary if the group does not seem to be responding. It is deprioritised below, but easy to read due to its isolation when in a frustrating situation that the customer is not replying.



Milestone 14

Please refer to the following screenshots from our Google Analytics report which were generated at 15:30 on 25 September.



WHAT ARE YOUR TOP EVENTS?

Event count by Event name

EVENT NAME	EVENT COUNT
page_view	304
user_engagement	185
session_start	93
first_visit	52

Last 7 days ▾

[View events →](#)

WHAT ARE YOUR TOP CONVERSIONS?

Conversions by Event name

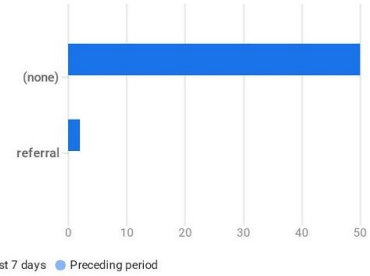
No data available

Last 7 days ▾

[View conversions →](#)

WHERE DO YOUR NEW USERS COME FROM?

New users by User medium ▾



Last 7 days ▾

[View new users →](#)

HOW WELL DO YOU RETAIN YOUR USERS?

User activity by cohort

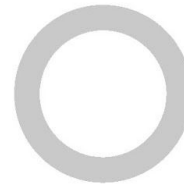
Based on device data only

	Week 0	Week 1	Week 2	Week 3	Week 4	Week 5
All Users	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Aug 9 - Aug 15						
Aug 16 - Aug 22						
Aug 23 - Aug 29						
Aug 30 - Sep 5						
Sep 6 - Sep 12						
Sep 13 - Sep 19						

6 weeks ending Sep 19

HOW DOES ACTIVITY ON YOUR PLATFORMS COMPARE?

Conversions ▾ by Platform

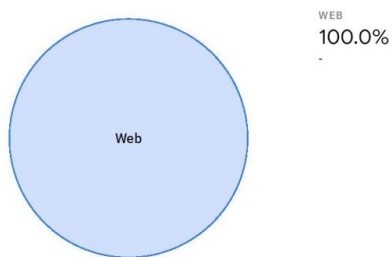


Last 28 days ▾

[View cross-platform →](#)

WHAT'S THE GROUPING OF USERS BY PLATFORM?

Users by Platform



Last 28 days ▾

[View cross-platform →](#)



Milestone 15

Please refer to our GitHub repository for our Google Lighthouse HTML report.

We attained a score of 11/14 on the Google Lighthouse report.

The missing 3 points are related to service workers. Due to Flutter Web being relatively new, there is a lack of community support with specific issues regarding registration of service workers. There was little support for service workers with Dart. Other developers also faced issues regarding service workers when hosting Flutter Web apps on GitHub Pages. For instance, see: [Flutter Service Worker not working with Github pages · Issue #52675 · flutter/flutter](#). Even when following the suggested steps, the issue was not resolved. We tried very hard in a variety of ways, but ultimately it was not solvable.



Appendix

This appendix contains some things of note about the implementation.

SMS

While the SMS alert feature has been implemented in our code, it is currently commented out. This is because the free version of the SMS APIs service by Vonage only allows sending SMS messages to a pre-registered phone number. The following is a screenshot of the message we received on our registered phone, after pressing the “Alert” button on the restaurant Waitlist screen.



Heroku

As we are using a Free-tier account to host our backend server on Heroku, its dyno sleeps after 30 minutes of inactivity. When a new request is sent to the server during this inactivity period, the server may take a while to “wake up” and respond. Subsequent requests should be significantly faster after the server has woken up.

Future Extensions

We plan to implement data analytics features for restaurants who don’t mind paying a small price for gathering insights into the queueing data we collect. Since we have access to the time and group size of all the customers who have queued at a restaurant, we can present the customer traffic and rate of serving at different times of the day, so that restaurants can better allocate manpower and optimise their workflow. This also enables us to cater to different market segments when monetising our product.

Another feature we would like to implement is that restaurants may allow diners to specify their seating preferences (e.g. counter seats, baby seats etc.). This feature is useful in restaurants with multiple types of seating arrangements, such as a sushi restaurant.



the end