WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Smart Queue Management System Real-Time Token Queue with Emergency Priority and Phone Notifications Project Journal

Pragadees A.M, Naveen K.C, Jeevanesh .T, Ms. Nishmitha K.R

Department of Information Technology. Sri Shakthi Institute of Engineering and Technology, Coimbatore-641062, Tamil Nadu, India.
Email ID: pragadeesmayil@gmail.com

ABSTRACT

Waiting lines in hospitals, banks, billing counters and government offices are often managed with paper slips or by customers standing in a physical line, so a person cannot tell how many people are ahead or when the turn will come. This project presents the Smart Queue Management System, a web application in which a customer obtains a digital token, follows the queue live on a phone or computer, and can register the phone for a notification about the turn. The interface is written in React with TypeScript and built with Vite. Google Firebase provides the back end: Cloud Firestore stores the tokens and counters and streams every change to the open pages in real time, Firebase Authentication protects the admin panel, Firebase Cloud Messaging is used to register devices for push notifications, and Firebase Hosting is configured to serve the built application. Token numbers are generated inside a Firestore transaction so that two customers cannot receive the same number. Approved administrators sign in with an email and password, call the next token, skip a token, create emergency tokens that are served before ordinary tokens, and reset the queue. The customer page shows the current token, the number of people ahead, a wait estimate of five minutes for each person ahead, and alerts when the turn is near and when it arrives. This journal describes the design, data model, workflow and source-level behaviour of the supplied version, lists twelve manual test scenarios, and records its limitations. These include admin checks that run in the browser, a fixed wait estimate, and the absence of the server-side notification sender and of automated tests from the supplied archive. The project demonstrates a small serverless, real-time queue application.

Keywords: Queue Management, Token System, React, TypeScript, Firebase, Cloud Firestore, Real-time Updates, Push Notifications, Emergency Priority.

1. INTRODUCTION

Public and commercial services in India, such as hospital outpatient counters, bank branches, ration shops and government offices, serve many people through a small number of counters. Customers usually stand in a line or hold a paper slip and keep watching the counter to learn whether their turn has come. This wastes time, crowds waiting areas, and creates disputes about who arrived first. Urgent cases such as a patient who needs immediate care are handled by staff shouting or by informal arrangements. These problems motivate a simple digital queue that gives every customer a number, shows the live position, and lets staff manage the line from one screen.

Web technologies make such a tool easy to deliver, because the system can be opened in any browser without installing an application. In this project, the Smart Queue Management System combines a React interface, a cloud database that pushes changes to every open page, administrator login, and phone notification registration in one application.

1.1 Existing challenges

With a manual queue, the customer has no view of the number of people ahead and cannot leave the waiting area without losing the place. Staff have no single record of who is waiting, who has been served, and who was skipped. A customer who arrives late may push in, and an emergency case may be delayed because it is not distinguished from ordinary cases. A useful system should therefore issue unique and ordered token numbers, show the current token and the position of each customer in real time, let staff call, skip and prioritise tokens, restrict these controls to authorised staff, and tell the customer when the turn is close.

1.2 Role of web technologies

React builds the interface from components and re-renders it when the data changes, which suits a queue board that changes every few seconds. TypeScript adds type checking to the source code, and Vite provides a fast development server and a production build. Cloud Firestore is a cloud document database whose real-time listeners deliver every change to all connected pages, so the project does not need its own server or polling code. Firebase Authentication

supplies email and password login and anonymous sign-in. Firebase Cloud Messaging and a browser service worker allow a web page to receive notifications on a phone, and Firebase Hosting serves the built files over HTTPS.

1.3 Problem statement and objectives

The project addresses the need for a convenient queue system that issues digital tokens, shows the live state of the queue to customers, gives authorised staff the controls to run the queue, and serves emergency cases first. The supplied version is a working prototype that uses a Firebase project and is organised around two services, General and Billing, plus an Emergency category created by staff.

The objectives of the Smart Queue Management System are:

i. To let a customer obtain a numbered digital token for the General or Billing service by entering a name.

ii. To generate token numbers without duplicates, even when many customers join at the same time.

iii. To show every user a live queue that updates without refreshing the page.

iv. To show the customer the current token, the people ahead, an estimated wait and the status, with alerts when the turn is near and when it arrives.

v. To let approved administrators sign in and call the next token, skip a token and reset the queue.

vi. To serve emergency tokens, which can be created only from the admin panel, before ordinary tokens.

vii. To let a customer register a phone for push notifications.

viii. To deploy the application as a static web application on Firebase Hosting.

2. MATERIALS AND METHODS

2.1 Development approach

The project was developed through requirement identification, data model design, implementation of the service functions, development of the customer interface, development of the admin interface and login, notification set-up, and deployment configuration. The application was created from the Vite React and TypeScript template and is kept in a Git repository. The source folder contains three React components (App.tsx, Admin.tsx and Login.tsx), two service modules (queueService.ts and notificationService.ts), the Firebase initialisation file (firebase.ts) and one stylesheet (App.css). The browser talks to Firebase directly through the Firebase JavaScript SDK, so the project has no server code of its own.

2.2 System architecture

The system follows a client and cloud-service architecture. The browser downloads the React application from Firebase Hosting. When the page opens, the customer interface subscribes to the tokens collection in Cloud Firestore and receives the full ordered list, and again whenever any token changes. When a customer asks for a token, the service layer runs a Firestore transaction. When an administrator presses a control, the admin component updates the status of a token document, and every open page receives the change through its subscription. Table 1 lists the layers and their responsibilities.

Layer / component	**Responsibility**
Customer interface (App.tsx)	Token request form, live queue list, current token, people ahead, wait estimate, turn alerts, notification button.
Admin interface (Admin.tsx, Login.tsx)	Admin login, call next, skip, reset, emergency token creation, queue summary and queue lists.
Service layer (queueService.ts)	Transaction-based token generation for normal and emergency tokens, latest-token query, real-time queue listener.
Notification module (notificationService.ts, firebase-messaging-sw.js)	Permission request, device registration with Firebase Cloud Messaging, saving the device token, showing received messages.
Firebase initialisation (firebase.ts)	Creates the Firebase app and exports the Firestore and Authentication instances.

Layer / component	Responsibility
Cloud Firestore	Stores the tokens, the two counters and the notification registrations; pushes changes to listeners.
Firebase Authentication	Email and password login for administrators; anonymous sign-in for customers who enable notifications.
Firebase Hosting	Serves the built application (dist folder) with all paths rewritten to index.html.

Table 1. Layers and components of the Smart Queue Management System.

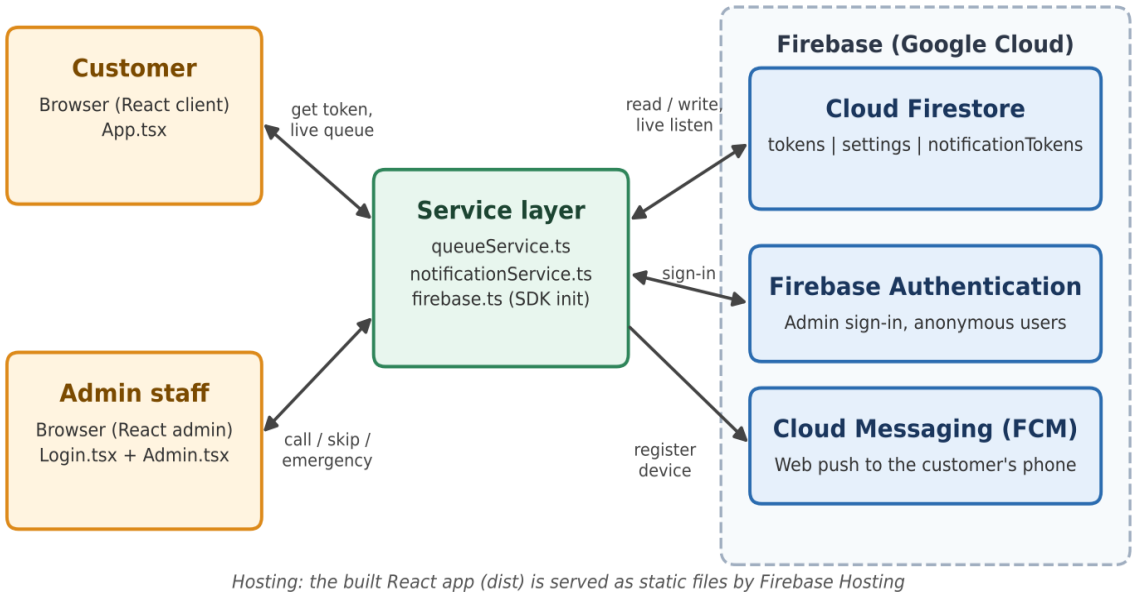


Fig. 1. System architecture of the Smart Queue Management System.

2.3 Technology stack and development environment

Technology	Use in the project
React 19	Component-based user interface and state handling.
TypeScript	Typed source code; type check runs before every build.
Vite	Development server and production build.
Firebase JavaScript SDK (^13.0.0)	Access to Firestore, Authentication and Cloud Messaging from the browser.
Cloud Firestore	Real-time document database for tokens, counters and notification registrations.
Firebase Authentication	Admin email and password login; anonymous sign-in for customers.
Firebase Cloud Messaging and service worker	Registration of a phone browser for web push notifications.
Firebase Hosting	Static hosting of the built application.

Technology	Use in the project
ESLint	Static code checks (npm run lint).
npm	Dependency installation and project scripts.

Table 2. Technologies used in the project.

The package file defines four scripts: dev starts the Vite development server, build runs the TypeScript compiler and then the Vite build, lint runs ESLint, and preview serves the built files locally. The notification feature needs a web push key, which is read from the environment variable VITE_FIREBASE_VAPID_KEY in a local .env file; the .gitignore file excludes .env files from the repository. The Firebase hosting configuration publishes the dist folder and rewrites every path to index.html, so the single-page application loads from any address. Some Firebase 13 packages declare Node.js 24.12 or newer as their engine requirement, so a recent Node.js version is needed for installation.

2.4 Database design

The application uses three Firestore collections, shown in Fig. 2. The tokens collection holds one document for every token; its identifier is generated by Firestore. The settings collection holds two documents, queue and emergencyQueue, each with a field lastTokenNumber that records the last number issued. The notificationTokens collection holds one document for every customer who enabled notifications; its document identifier is the token code, so a registration can be matched with its token. Table 3 describes the fields.

Collection.field	Type	Purpose
tokens.token	string	Display code, for example A001 or E001.
tokens.number	number	Sequence number; used to order the queue.
tokens.name	string	Customer or patient name.
tokens.service	string	General, Billing or Emergency.
tokens.status	string	Waiting, Serving, Completed or Skipped.
tokens.priority	boolean	True for emergency tokens.
tokens.createdAt	server timestamp	Time the token was created.
settings/queue.lastTokenNumber	number	Last number issued for normal tokens.
settings/emergencyQueue.lastTokenNumber	number	Last number issued for emergency tokens.
notificationTokens.fcmToken	string	Device token from Firebase Cloud Messaging.
notificationTokens.ownerUid	string	Anonymous user that registered the device.
notificationTokens.approachingSent / turnSent	boolean	Flags for the two messages; both start as false.

Table 3. Main fields of the Firestore collections.

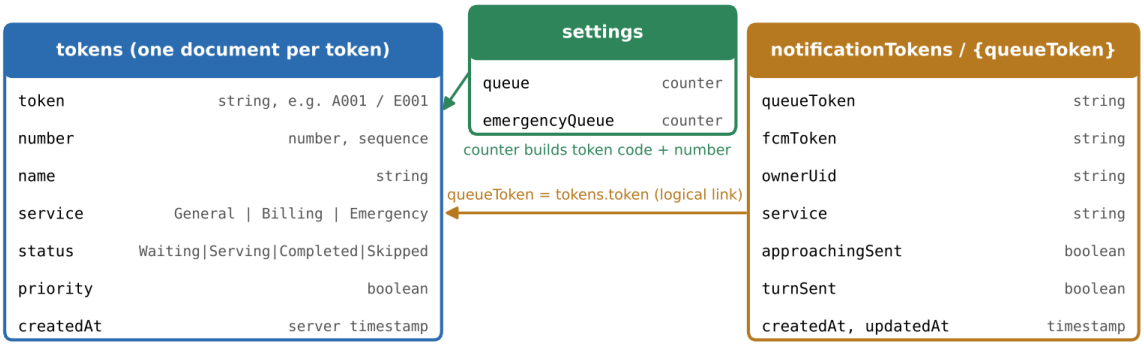


Fig. 2. Firestore data model: tokens, the two counters in settings, and notification registrations.

2.5 Token numbering, status and customer-side rules

A normal token receives the letter A and the next counter value padded to three digits, for example A001, A002 and A003. An emergency token receives the letter E from its own counter, for example E001. Each token is created inside a Firestore transaction that reads the counter, adds one, and writes the counter and the new token together. If two customers press the button at the same moment, Firestore retries the transaction for one of them, so the two tokens receive different numbers. A token starts as Waiting, becomes Serving when the administrator calls it, and then becomes Completed when the next token is called, or Skipped when the administrator presses Skip (Fig. 3).

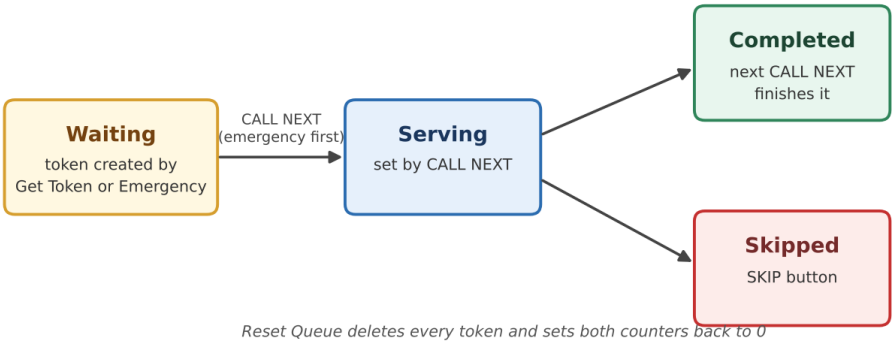


Fig. 3. Life cycle of a token.

The customer page calculates three values from the live list. People ahead is the number of tokens with status Waiting, the same service and a smaller number than the customer's token. The estimated wait is people ahead multiplied by five minutes, a fixed value in the code. The page then chooses the message in Table 4. The token code is also stored in the browser under the key myQueueToken, so the customer's view is restored when the page is reloaded.

Condition	Message shown to the customer
Status is Serving	It's Your Turn! Please proceed to the service counter.
Status is Waiting and people ahead is 2 or fewer	Your Turn Is Approaching! Only N customer(s) ahead of you.
Status is Waiting and people ahead is more than 2	Your token is waiting in the queue.

Table 4. Customer alerts and the conditions that trigger them.

2.6 Frontend design

The customer page contains the heading, an Admin Dashboard button, and a Client Dashboard card with a name field, a service selector (General or Billing) and a Get Token button. After a token is issued, a token box shows the code, an Enable Phone Notifications button, the alert banner, the current serving token, people ahead, estimated wait and status. Below it, a Live Queue list shows the code, name, service and status of every token. The admin

page has a header, a Current Token card with the buttons Call Next, Skip and Reset Queue, a Queue Summary card with four counters (Emergency, Waiting, Completed and Skipped), a form for creating an emergency token, and two lists: the Emergency Queue and the Waiting Queue. The admin screen is reached through the Admin Dashboard button and a login page; the application does not use separate web addresses for these screens. The layout and colours are defined in App.css (about 530 lines) using card layouts.

2.7 Queue workflow

1. The customer opens the application. The page subscribes to the tokens collection, ordered by number, and restores any token saved in the browser.
2. The customer enters a name, selects General or Billing and presses Get Token. An empty name is rejected with a message.
3. generateToken runs a transaction that increments the counter in settings/queue and creates a token document with status Waiting and priority false.
4. The token code is shown and saved in the browser. Every open page receives the updated list immediately.
5. The page works out the current token, people ahead, estimated wait and the alert from the live list.
6. An administrator signs in and presses Call Next.
7. callNext marks the token that is Serving as Completed, then picks the next token: the first waiting emergency token if there is one, otherwise the first waiting token in number order, and sets it to Serving.
8. All pages update. The customer whose token is Serving sees the turn alert.
9. If the customer does not come, the administrator presses Skip and the token becomes Skipped.
10. At the end of the working period, the administrator presses Reset Queue and confirms. All tokens are deleted and both counters return to zero.

2.8 Emergency priority and notification workflow

Emergency tokens can be created only from the admin panel. The administrator enters the patient's name, and generateEmergencyToken creates a token with the service Emergency, the priority flag set to true and a code such as E001 from the separate emergency counter. When Call Next runs, waiting emergency tokens are always chosen before ordinary tokens, and among emergency tokens the one with the smallest number goes first.

For phone notifications, the customer presses Enable Phone Notifications after getting a token. The function enablePhoneNotifications checks that the browser supports notifications and service workers and that Firebase Messaging is supported, confirms that the web push key is present, and asks the user for permission. It signs the user in anonymously if nobody is signed in, and refuses to continue if an administrator account is signed in. It then registers the service worker file firebase-messaging-sw.js, obtains a device token from Firebase Cloud Messaging, and writes a document in notificationTokens that holds the device token, the queue token, the anonymous user identifier, the service and two flags, approachingSent and turnSent, both false. A listener in the page shows messages received while the page is open by calling the service worker's showNotification function. The supplied archive does not contain the server-side code that decides when to send a message, so the delivery of a push message at the time of the turn could not be confirmed from the archive.

2.9 Service and component functions

Function	**Location**	**Purpose**
generateToken(name, service)	queueService.ts	Transaction that issues a normal token (A###).
generateEmergencyToken(name)	queueService.ts	Transaction that issues an emergency token (E###).
listenToQueue(callback)	queueService.ts	Real-time listener on tokens ordered by number.
getLatestToken()	queueService.ts	Reads the token with the highest number; defined but not used by the pages.
enablePhoneNotifications(token, service)	notificationService.ts	Registers the device and saves the notification record.
listenForPhoneNotifications()	notificationService.ts	Shows messages received while the page is open.

Function	Location	Purpose
handleLogin()	Login.tsx	Email and password sign-in and approved-admin check.
callNext(), skipToken(), resetQueue(), createEmergency()	Admin.tsx	Queue control actions on the tokens collection.

Table 5. Main functions of the project.

2.10 Input validation and security

The name field must contain text, and a token cannot be requested otherwise. Emergency creation also requires a name, and Reset Queue asks for confirmation before deleting. Administrators sign in with Firebase email and password authentication. After a successful sign-in, the code compares the e-mail address with a list of three approved administrator addresses in Login.tsx; any other account is signed out at once and the message Access denied is shown. A signed-in administrator account is not allowed to register customer notifications. Names are displayed through React, which escapes text by default. The web push key is kept in an environment file that is not committed.

These controls do not make the application production-secure. The admin screen is shown according to a state value inside the browser, and the approved list is part of the client code. The archive contains no Firestore security rules file, so it cannot be confirmed that the server rejects token deletion, status changes or emergency token creation from an ordinary visitor. There is no rate limiting, and the Live Queue shows the names of all customers to every visitor. The Firebase web configuration values in the source identify the project and are not secrets, but the security of the data depends on the rules configured in the Firebase console.

2.11 Testing and evaluation method

The project does not include an automated test suite; it includes an ESLint configuration and a TypeScript build step that checks types. For this journal, twelve manual test scenarios were defined from the behaviour of the source code (Table 8). They are meant to be run on a local or deployed copy with two browser windows, one for the customer and one for the administrator. The Result column is to be completed with Pass or Fail when the scenarios are executed. No performance, load or usability measurements were taken, so none are reported.

3. RESULTS AND DISCUSSION

3.1 Implemented application features

The supplied project contains a customer interface, an admin interface with login, a service layer with transaction-based token generation, a real-time queue listener, an emergency priority mechanism and a notification registration module. A customer can obtain a token, see the live queue and receive on-screen turn alerts. An administrator can call, skip and reset tokens and create emergency tokens. These are implementation features verified from the project source; the usefulness of the system for real counters would need separate evaluation.

3.2 Implementation summary

Item	Value
React components (App, Admin, Login)	3
Service modules	2 (queue, notification)
TypeScript / TSX source	About 1,170 lines
Stylesheet (App.css)	About 530 lines
Firestore collections	3 (tokens, settings, notificationTokens)
Token statuses	4 (Waiting, Serving, Completed, Skipped)
Services offered to customers	2 (General, Billing) plus Emergency from the admin panel

Item	Value
Admin actions	4 (Call Next, Skip, Reset Queue, Create Emergency Token)
Approved administrator accounts in the code	3
Wait estimate	5 minutes per person ahead
“Turn approaching” threshold	2 or fewer people ahead
Manual test scenarios defined	12
Automated tests	0

Table 6. Summary of the implementation.

3.3 Worked example of the queue logic

Table 7 applies the rules of Section 2.5 to an illustrative queue. The queue is an example computed from the logic in the source code; it is not data from a live run. A001 is Completed, A002 (Billing) is Serving, A003 to A006 (General) are Waiting, and E001 is a waiting emergency token. A waiting token's people ahead count includes only waiting tokens of its own service with a smaller number.

Token	Service	Status	People ahead	Estimated wait	Message
A003	General	Waiting	0	0 min	Approaching
A004	General	Waiting	1	5 min	Approaching
A005	General	Waiting	2	10 min	Approaching
A006	General	Waiting	3	15 min	Waiting in queue
E001	Emergency	Waiting	0	0 min	Approaching

Table 7. Example values calculated by the customer page for an illustrative queue.

When the administrator presses Call Next, A002 becomes Completed and E001 becomes Serving, because waiting emergency tokens are chosen before the lower numbers A003 to A006. On the next press, E001 becomes Completed and A003 becomes Serving. The example also shows a weakness of the estimate: the emergency token is served before A003, but it is not counted in the people ahead of A003 to A006, so their estimates are one person short while it is waiting.

3.4 Application screens

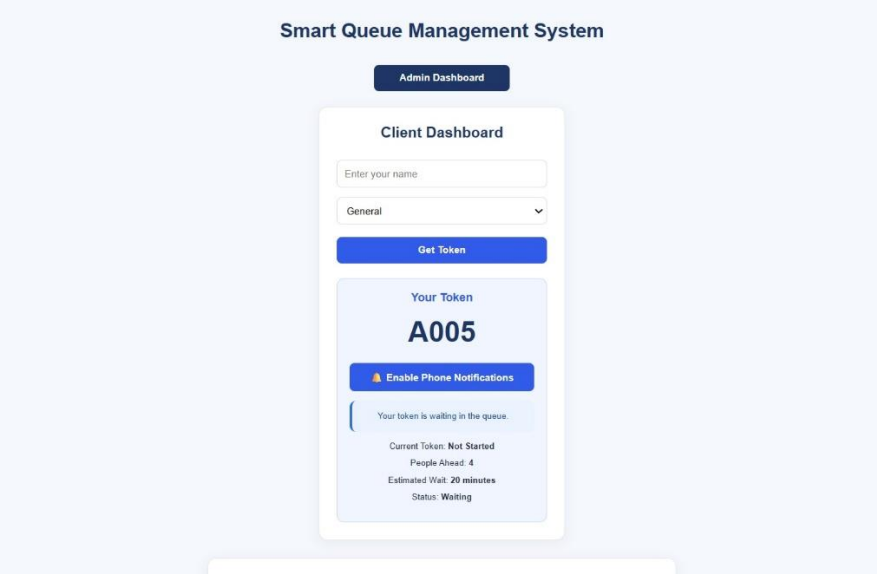


Fig. 4. Customer dashboard showing the token, people ahead, estimated wait and the Live Queue.

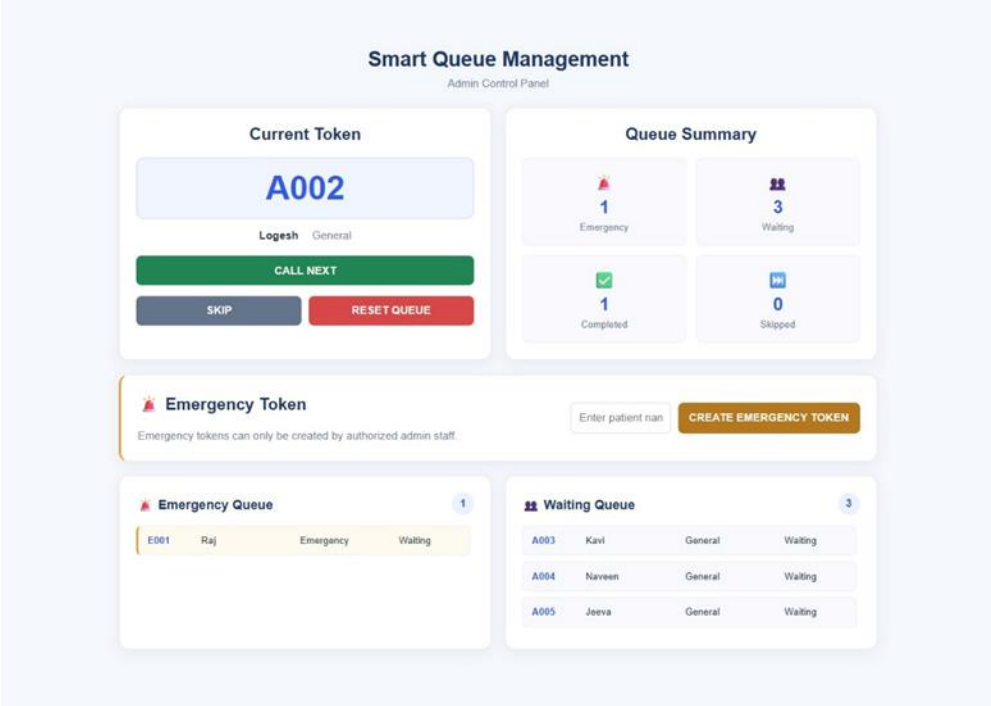


Fig. 5. Admin control panel after login.

3.5 Functional test scenarios

ID	Module	Test scenario	Expected result (from source)	Result
T1	Customer	Press Get Token with an empty name	Message asking for the name; no token created.	Pass / Fail
T2	Token generation	Enter a name, choose General, press Get Token	Token such as A001 shown, status Waiting, saved in the browser.	Pass / Fail

ID	Module	Test scenario	Expected result (from source)	Result
T3	Token generation	Two customers press Get Token together	Two different, consecutive numbers.	Pass / Fail
T4	Live queue	Open two windows and add a token in one	New token appears in the other without reload.	Pass / Fail
T5	Notification	Press Enable Phone Notifications with no token	Message asking to generate a token first.	Pass / Fail
T6	Admin login	Log in with a wrong password	Message: Invalid email or password.	Pass / Fail
T7	Admin login	Log in with a valid account that is not approved	Signed out; message: Access denied.	Pass / Fail
T8	Call Next	Press Call Next with no waiting token	Message: No waiting tokens.	Pass / Fail
T9	Emergency	Create E001, then press Call Next with A-tokens waiting	E001 becomes Serving before the A-tokens.	Pass / Fail
T10	Skip	Press Skip while no token is serving	Message: No token is currently serving.	Pass / Fail
T11	Emergency	Create an emergency token with an empty name	Message asking for the patient's name.	Pass / Fail
T12	Reset	Press Reset Queue and confirm	All tokens deleted; next token is A001 again.	Pass / Fail

Table 8. Manual functional test scenarios. Expected results are derived from the source code; the Result column is filled in when the scenarios are run.

3.6 Discussion and limitations

The system shows that a small real-time queue application can be built with a React interface and Firebase services alone, with no server code to maintain. Because Firestore pushes changes to every listener, the customer view, the admin panel and the live list stay consistent without polling, and the transaction-based counter gives every token a unique number. The interface is separate from the stored data, so the screens can be redesigned without changing the database.

Several limitations of the supplied version should be stated.

- The wait estimate is a fixed five minutes per person and is not based on measured service time. It also leaves out emergency tokens that will be served first.
- There is a single serving position. Call Next picks the lowest waiting number across General and Billing together, so the two services do not have separate counters or separate queues.
- Call Next performs two separate updates (complete the current token, serve the next one) that are not in a transaction, and Reset Queue deletes tokens one at a time. Two administrators acting together could leave an inconsistent queue, and a failure during reset could leave part of the data behind. Reset also does not clear notificationTokens.
- Admin access is checked in the browser, and no Firestore rules file is included, so server-side protection could not be confirmed.
- The code that sends the approaching and turn messages is not part of the archive, so end-to-end push delivery could not be confirmed.
- A skipped token cannot return to the queue, and reset deletes all history, so there is no record for reports or for calculating real service times.
- Normal and emergency counters both start at 1 and are sorted on the same field; ordering is correct only because of the priority flag.
- The customer's token is linked to the browser through local storage, so clearing the storage loses the link. The Live Queue shows every customer's name to all visitors, which may not suit a hospital.

- The interface is in English only, the component state uses loosely typed arrays, getLatestToken is not used, and the project has no automated tests.

4. CONCLUSION

The Smart Queue Management System is a web application for issuing digital queue tokens and managing the queue in real time. It uses React, TypeScript and Vite for the interface and Firebase services for the database, login, notification registration and hosting. Customers can obtain a token, follow the queue live, and see alerts when their turn is near or has come. Approved administrators can call, skip and reset tokens and create emergency tokens that are served first. Token numbers are generated in transactions so that they are unique.

The supplied version is a prototype. The wait time is a fixed estimate, the admin check runs in the browser, and the notification sender and the automated tests are not part of the archive. Future work includes writing Firestore security rules and moving admin actions to trusted server functions, adding the server code that sends the approaching and turn notifications, using measured service times for the estimate, adding separate counters for each service, using transactions for Call Next and Reset, keeping a history for reports, hiding customer names from the public list, supporting Tamil and Hindi, and adding automated tests. With these improvements the system could support real service counters.

5. ACKNOWLEDGMENTS

The authors express their sincere gratitude to Sri Shakthi Institute of Engineering and Technology, Coimbatore-641062, Tamil Nadu, India for providing the academic environment and facilities required for this project. The authors thank the project guide, Ms. Nishmitha K R , for guidance, encouragement and support throughout the development of the Smart Queue Management System. The authors also acknowledge the faculty members and contributors who supported the learning and implementation process.

6. REFERENCES

1. Meta Open Source, “React Documentation.” Available: https://react.dev/
2. Vite, “Vite Documentation.” Available: https://vite.dev/
3. Microsoft, “TypeScript Documentation.” Available: https://www.typescriptlang.org/docs/
4. Google, “Cloud Firestore Documentation.” Available: https://firebase.google.com/docs/firestore
5. Google, “Transactions and batched writes,” Cloud Firestore Documentation. Available: https://firebase.google.com/docs/firestore/manage-data/transactions
6. Google, “Firebase Authentication Documentation.” Available: https://firebase.google.com/docs/auth
7. Google, “Set up a Firebase Cloud Messaging client app on the web.” Available: https://firebase.google.com/docs/cloud-messaging/js/client
8. Google, “Firebase Hosting Documentation.” Available: https://firebase.google.com/docs/hosting
9. MDN Web Docs, “Service Worker API.” Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
10. J. D. C. Little, “A Proof for the Queuing Formula: L = λW,” Operations Research, vol. 9, no. 3, pp. 383–387, 1961.
11. Project source code, “Smart Queue Management System,” supplied project archive, Queue-management.zip. Includes src/App.tsx, src/Admin.tsx, src/Login.tsx, src/queueService.ts, src/notificationService.ts, src/firebase.ts, src/App.css, public/firebase-messaging-sw.js, firebase.json and package.json.

APPENDIX: PROJECT ARTIFACTS USED

The journal was prepared from the uploaded project archive. Files used to ground the implementation description include: src/App.tsx, src/Admin.tsx, src/Login.tsx, src/queueService.ts, src/notificationService.ts, src/firebase.ts, src/App.css, src/main.tsx, public/firebase-messaging-sw.js, firebase.json, .firebaserc, .gitignore, package.json and the Vite and TypeScript configuration files. Figs. 1, 2 and 3 were drawn for this journal from the project structure. Figs. 4 and 5 are to be captured from the running application. The archive did not contain Firestore security rules, server-side notification code, automated tests, or any recorded test results.