I want you to build a web app for my pet waste removal business called "USA Scoops."

High level

- Frontend: React with React Router and MUI (Material UI or other framework) for layout and styling.

- Backend: Firebase for Authentication, Firestore, and later Storage if needed.

- Hosting: Netlify for the React app. Do not use Replit hosting in production.

- Database: Firestore as the main data store.

- Authentication: Firebase Auth with email and password.

- Roles: customer, technician, admin, managed via Firebase Auth and Firestore.

Important constraints

- The app should be designed so that it can be deployed outside Replit, for example on Netlify.

- All backend logic should go through Firebase. Do not add a custom Node or Express server.

- Use environment variables for Firebase config (for example VITE_ or REACT_APP_ style vars).

- Keep the code organized and easy to extend.

Business description

The app is for a local pet waste removal service called USA Scoops. Customers can:

- Check if their zip code is in the service area.

- Sign up with their contact information, address, and number of dogs.

- Receive a quote based on number of dogs

- Book a service time window on available days.

Admins can:

- Control which zip codes are allowed.

- Control which days are available for service.

- Define and edit time windows for those days, for example 9:00 to 11:00, 11:00 to 13:00.

- Set capacity for each time window (how many customers can be booked in that slot).

- View all bookings and optionally assign technicians.

- Set pricing

*Some Admins could be Technicians

Technicians can:

- Log in and see the list of visits assigned to them.

- Mark visits as completed.

*Some Technicians could be Admins

I want the app to support any day of the week. The admin will turn

days on or off through the admin UI as part of the availability.

Routing structure

Use React Router. Set up these main routes:

Public routes

- "/" Landing page
- "/signup" Signup and booking flow for customers
- "/login" Login page for customers, technicians, and admins

Protected routes

- "/portal" Customer portal (role customer)
- "/tech" Technician portal (role technician)
- "/admin" Admin dashboard (role admin)

Handle roles in a simple way at first

- Store role information in Firestore and mirror it in a small context or hook.
- You can assume I will manually set roles in Firestore during MVP testing, or you can create a very simple admin user seeding script.

UI and styling

- Use MUI components for layout and styling.

- Use a simple, clean look.

- I already have a logo. Add a placeholder area for the logo at the top, I will swap in the actual image later.

- Make the layout responsive and mobile friendly.

Data model in Firestore

Define these collections and fields. Use these as a guideline.

1) customers

- uid string should match Firebase Auth uid

- name string

- email string

- phone string

- address object:

  - street string

  - city string

  - state string

  - zip string

  - gate_code string optional

  - notes string optional

- dog_count number

- status string example "active", "paused", "prospect"

- created_at timestamp
- updated_at timestamp

2) service_zips

This collection defines which zip codes are allowed to sign up.

- id auto id
- zip string
- active boolean

3) slots

A slot represents a bookable time window on a specific date.
Admins create slots. Customers book them.

- id auto id
- date string, format "YYYY-MM-DD"
- window_start string, format "HH:mm" in local time
- window_end string, format "HH:mm"
- status string, for example "open", "held", "booked", "blocked"
- capacity number, how many customers can book this slot
- booked_count number, how many bookings exist for this slot
- created_at timestamp

- updated_at timestamp

4) visits

A visit represents a scheduled service for a customer in a specific slot.

- id auto id

- customer_uid string

- slot_id string

- scheduled_for timestamp, derived from date plus window start

- status string, for example "scheduled", "completed", "skipped", "canceled"

- technician_uid string optional

- notes string optional

- created_at timestamp

- updated_at timestamp

5) technicians

- uid string matches Firebase Auth uid

- name string

- email string

- phone string

- service_zips array of strings (zip codes they cover)

- active boolean

6) messages

For customer contact or support messages.

- id auto id
- customer_uid string
- subject string
- body string
- status string example "open", "closed"
- created_at timestamp

7) waitlist

For people outside the service area.

- id auto id
- name string
- email string
- zip string
- created_at timestamp

We do not need a payments collection in the MVP. Treat bookings as free for now, but create a form that simulates

payment. Later I will add Stripe.

Auth and roles

- Use Firebase Auth email and password.

- On signup, create a customer user with role "customer" by default.

- For technicians and admins, assume I will create their account, and set their role in Firestore manually. I should be able to mark a tech or admin as such in Firestore. Some admins will be techs and some techs will be admins.

- Create a small "useAuth" hook or context that:

  - Listens to Firebase Auth state.

  - Fetches the role and basic profile from Firestore for the current user.

  - Exposes user, role, loading, and error state.

Route protection

- Customer portal "/portal" should only be accessible to users with role "customer".

- Technician portal "/tech" should only be accessible to users with role "technician"

- Admin dashboard "/admin" should only be accessible to users with role "admin".

- The login route should redirect appropriately after login based on role. If a technician is an admin make it easy for them to switch to the admin page. Likewise, if an admin is a technician make it easy

for them to switch to the technician page. If an admin flag is present, the admin page should be displayed by default to those users.

Page details

1) Landing page ("/")

- Show logo and simple hero section. I have a custom logo to use

- Headline such as "America's Cleanest Yards, One Scoop at a Time".

- Short paragraph about the service.

- Simple "How it works" section with three steps and icons:

  1. Check your zip

  2. Pick a time window

  3. We scoop your yard

- A prominent "Get started" button that routes to "/signup".

2) Signup flow ("/signup")

Implement signup with clear steps in a single page or as multi step components.

Step 1 basic info and zip check

- Ask for:

  - name

  - email

  - phone

  - zip code

- Validate zip code by checking against the "service_zips" collection where active is true.

- If zip is allowed:

  - Let the user proceed to the next step.

- If zip is not allowed:

  - Show a friendly message that we are not in their area yet.

  - Offer a short form (name, email, zip) and save it into "waitlist".

  - Do not allow booking if the zip is not in the allowed list.

Step 2 address and dog details

- Ask for:

  - street

  - city

  - state

  - zip (pre filled from step 1)

  - optional gate code or notes about the yard

  - number of dogs

- If the user is not authenticated yet, create a Firebase Auth user with email and password or use a basic signup pattern.

- Create or update a "customers" document linked to the Firebase

Auth uid.

Send user a quote for service. Provide a button to proceed to step 3.

Step 3 pick a time window (slot)

- Query the "slots" collection for upcoming dates where:

  - status is "open"

  - booked_count is less than capacity

- Display the available slots grouped by date.

- Show each slot as something like "Saturday, May 10 9:00 to 11:00 (2 of 4 spots left)".

- When the user selects a slot:

  - Create a new "visits" document with:

    - customer_uid set to the signed in user uid

    - slot_id set to the selected slot

    - scheduled_for derived from the slot date and start time

    - status set to "scheduled"

  - Increment slot.booked_count in a safe way. Use a transaction or a Cloud Function if needed to avoid race conditions.

For MVP, you can implement the slot booking update directly from the client, but design it so that it will be easy to move to a Cloud Function later for stricter control.

Confirmation

- After booking, show a confirmation screen with:
  - "Thanks, you are booked"
  - The date and time window
  - A link or button "Go to my portal" that routes to "/portal".

3) Login page ("/login")

- Simple login form: email and password.
- After successful login:
  - If role is "customer" route to "/portal".
  - If role is "technician" route to "/tech".
  - If role is "admin" route to "/admin".
- Show helpful errors for invalid credentials.

4) Customer portal ("/portal")

This is for authenticated customers.

- Show a card with the next scheduled visit:
  - Date
  - Time window
  - Address
  - Visit status
- Show a table or list with past visits and their status.

- Add a simple "Contact us" form that writes a message to the "messages" collection.

5) Technician portal ("/tech")

This is for authenticated technicians.

- Show a date picker or default to today.

- Query "visits" where:

  - technician_uid equals the current technician uid or is null if you want to show unassigned visits.

  - scheduled_for is on the selected date.

- For each visit show:

  - Customer name

  - Address

  - Time window

  - Dog count

  - Notes

  - Status

- Each visit row should have a "Mark completed" button that updates the visit status to "completed" and sets an updated_at timestamp.

6) Admin dashboard ("/admin")

This is for admins.

Organize the admin dashboard into tabs or sections using MUI components.

a) Zip code manager

- Read from "service_zips".
- Show a table with zip and active.
- Provide a small form to add a new zip code.
- Provide a toggle or button to deactivate or delete a zip code.

b) Slots and service day manager

This controls what days and windows customers can book.

- Form to create slots:
  - Pick one or more dates using a date picker.
  - Define one or more windows per day, for example:
    - 09:00 to 11:00
    - 11:00 to 13:00
  - Set capacity for each window.
- When the admin clicks "Generate slots":
  - Create "slots" documents in Firestore for each date and window with:
    - status set to "open"

  - capacity set to the chosen capacity

  - booked_count set to zero

- Also show a table of upcoming slots with:

  - date, window_start, window_end

  - capacity

  - booked_count

  - status

- Allow admin to:

  - Change capacity

  - Change status to "blocked" if they want to turn off a specific slot

  - Delete a slot entirely if there are no visits booked yet

c) Visits overview

- Table of visits, filterable by date and status.

- Include:

  - customer name and zip

  - date and window

  - status

  - technician assignment

- Add a dropdown on each row to assign or change a technician by technician uid.

- When a technician is set, write technician_uid on the visit

document.

Code organization

Use a clear folder structure for the React app.

For example:

- src/
  - main entry file
  - App component and routing
  - pages/
    - LandingPage.jsx
    - SignupPage.jsx
    - LoginPage.jsx
    - CustomerPortal.jsx
    - TechnicianPortal.jsx
    - AdminDashboard.jsx
  - components/
    - shared components like layout, navigation bar, forms
  - firebase/
    - firebaseConfig.js or similar
    - helper functions for Firestore queries
  - hooks/

  - useAuth.js or similar hook for auth and roles
- context/
  - AuthContext if needed

General expectations

- Add basic loading and error states for async operations.
- Add simple form validation for required fields.
- Comment the code where the logic is non trivial, such as slot booking, role checks, and admin actions.
- Keep everything in a single Replit project but always assume the app will be deployed to Netlify, not to Replit hosting.

If anything is unclear, make reasonable assumptions, or ask for clarification, and keep the code simple and well structured. Do not use placeholder pages, create a fully functional app as describe from the start.