Canvassing Web Application High Level Design

By Jessica Tran

Abstract

This document outlines my strategy for designing and implementing a canvassing web application. Due to time constraints, in this document, I outline a very simple demo application solution while proposing what the longer term implementation would look like if we want to account for long term scalability and security.

Problem Statement

Empower Project is an organization that runs programs to help support Relational Programs at scale. As part of the interview process for the Software Engineer position at Empower, I am designing and implementing a canvassing application.

The goal of this design is to discuss my strategy for implementing a canvassing web application.

Requirements

P0 requirements:

1. You should have one page that lets a canvasser write down a name and some free-form notes about a person they just talked to.
2. You should have another page that lets the canvasser view all of their canvassing notes.
3. It's okay if you don't have a user login / authentication system implemented.
4. It's okay if your pages are simple and/or ugly
5. Your backend should have some sort of a JSON API.
6. Please provide any documentation necessary to run your code and try it out.
7. Make a git repo with an initial commit when you start working, and include the repo in your submission
8. We understand that a lot of folks are experimenting with AI coding tools, particularly in interviews. If you want to, that's fine! Just say that you're using AI in your readme. Since a greenfield app like this is something AI should be good at, if you're using AI, we'll also expect you to get more of the "extra credit" done.
9. At Empower, we use React (with hooks) for the frontend, Node for the backend, MySQL for the database, and Typescript. Using these technologies would be ideal, but if you need to use the technologies of your choice, that's okay too.

P1 requirements:

Some ideas for enhancements to make to your project if you finish early:

1. Implement users and authentication
2. Allow editing the canvassing notes
3. Add an email field, and add some validation on that email
4. Make it possible to search across canvassing notes
5. Make it possible to export the canvassing results as CSV
6. Make the pages look nice
7. Make the pages look nice on mobile
8. Actually deploy it on a server somewhere

Proposed Solution

Technology Choices

To get started, I created a standard React with Next.js application using the following command:

```
npx create-next-app@latest . \
 --typescript \
 --tailwind \
 --eslint \
 --app \
 --src-dir \
 --import-alias "@/*"
```

In the table below, I outline the tech stack for various components of the application as well as offer a long-term solution I would implement if this project was not simply a demo.

Component	Technology	Notes
Frontend	React with Next.js	
Backend	Next.js API Routes	Longer term, Java package for business logic, Smithy model for API design
Database	SQLite (for demo) with Prisma.	Longer term, AWS DynamoDB

Deployment	Local (for demo). Vercel for longer term	Longer term, pipeline deployment via AWS CodeDeploy
Version Control	Git	

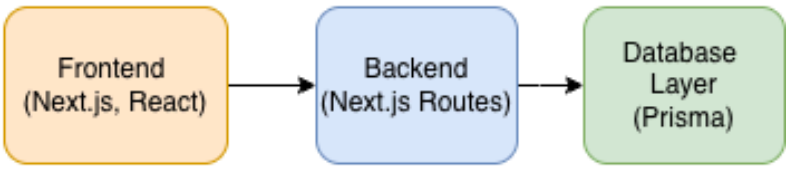
For simplicity sake, the application will be built as a React app using Next.js and will use Prisma for data persistence. Ideally, if this was a long-lived application that needed to scale, I would lean towards using AWS to stand up the infrastructure with the persistent storage solution using DynamoDb. For the purposes of this demo application, I am moving forward with Prisma.

System Overview

As stated above, the system is built to allow a canvasser to record a new interaction by submitting a form on the “New Note” page. The frontend calls a /api/notes endpoint, which stores the note in the SQLite database through Prisma. The “Notes” page fetches all existing notes from the same API.

The components of this application are demonstrated int he diagram below.

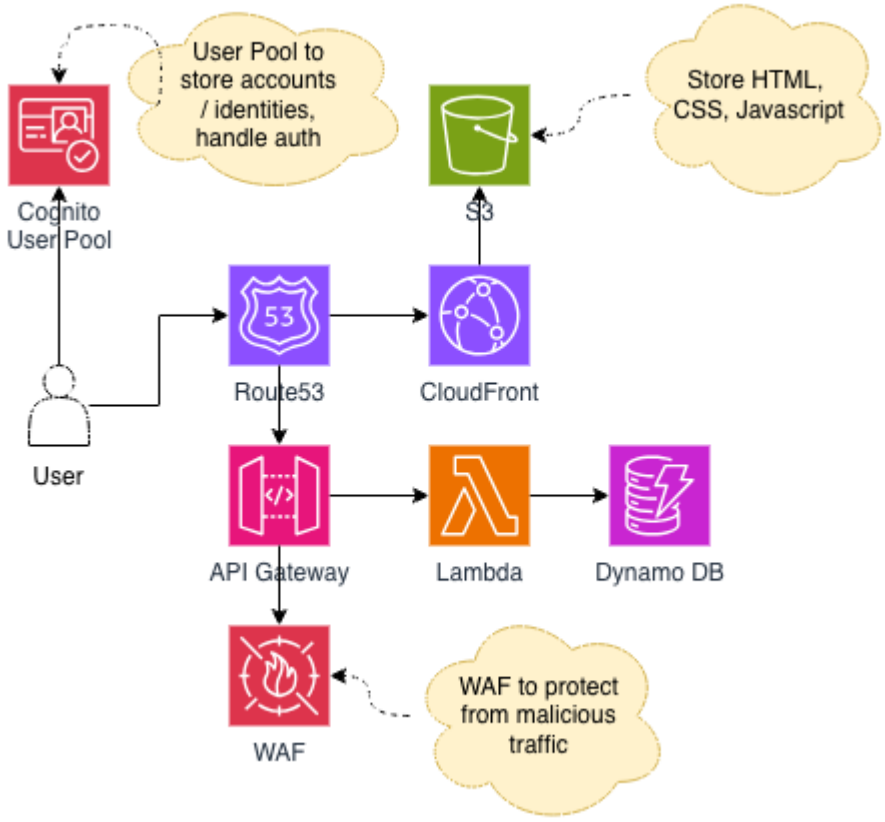
[Short term] High Level Architecture Diagram



[Longer term] High Level Architecture Diagram

A long term architecture could involve a system in which the API layer is abstracted behind AWS API Gateway, Lambda would handle business logic, and DynamoDB would store records. In a real-world system where there are multiple users logging into the application, we would utilize

AWS Cognito with User Pools to store account information and enable OAuth. I demonstrate what this could look like in a draw.io diagram I made below -



Draw.io Diagram

Data Model

The following table outlines the data model / schema for our application.
For our demo application, this Note entity is quite bare bones. For future enterprise web application considerations, I would propose we store additional data, such as location data, canvasser metadata, etc.

Entity	Field	Type	Description
Note	id	integer	Unique identifier
	name	string	Name of participant

	email	string	Email address
	notes	string	Free-form notes
	createdAt	datetime	Date created
	updatedAt	datetime	Date last updated

API Design

The application exposes a simple JSON API that allows the frontend to create, read, update, canvassing notes.

In the demo implementation, these endpoints are implemented using Next.js API routes under /app/api/notes.

In the longer-term implementation, these endpoints would be modeled in Smithy, with business logic implemented in AWS Lambda, fronted by API Gateway. Additionally, the longer-term implementation would version the API and route the API routes under something like /v1/notes.

For the demo, the endpoints exist under:

http://localhost:3000/api/notes

Longer term, we would want to set up a custom endpoint name such as:

https://api.empowerproject.org/canvassing

Endpoint	Method	Description
/api/notes	GET	Fetch all canvassing notes
/api/notes	POST	Create a new canvassing notes
/api/notes/:id	PUT	Update an existing note
/api/notes/:id	PATCH	Update an existing note

For the sake of simplicity, I will only outline the request and response objects for the POST method.

POST Request Body

```
{
  "name": "Jane Doe",
  "notes": "Interested in volunteering next week."
}
```

POST Request Response

```
{
  "id": 1,
  "name": "Jane Doe",
  "notes": "Interested in volunteering next week.",
  "createdAt": "2025-11-09T22:12:00Z"
}
```

Error Responses

- 400 Bad Request – missing or invalid input (e.g., empty name or invalid email)
- 500 Internal Server Error – database or server-side failure

Testing

We will want to implement testing for this application in various levels -

1. Firstly, we will have unit tests for our frontend code using Jest, where we mock Prisma for the backend.
2. For a more robust testing strategy, we can implement integration tests that exercise the /api/notes endpoints end-to-end. These tests can verify not only that requests and responses behave as expected, but also that data is correctly persisted and updated. Additionally, the integration test suite can serve as a lightweight canary for our backend, providing early detection if the system becomes unavailable or unresponsive.

Security Considerations

To make sure the application is secure, the long term approach would be to implement the following features:

1. Input sanitization
2. API rate limiting / throttling on the API Gateway
3. HTTPS enforcement
4. Implement the principle of least privilege by using IAM roles for DynamoDB and Lambda

Glossary of Terms

Term	Definition
Canvasser	A user who records interactions with people in the community.
Canvassing note	The name and notes recorded by a canvasser after a conversation.
ORM	Object-Relational Mapping – a way to interact with a database using code rather than SQL.
Prisma	A modern ORM for Node.js and TypeScript used to connect the backend to a database.
Next.js	A React-based framework for building full-stack web applications.