DOCUMENTATION:

PERSONAL PORTFOLIO WEBSITE

Prepared by
Tarus Mercy
2026

mercytarus644@gmail.com

Table Of Contents

PROJECT OVERVIEW

To practice my coding skills, I decided to build a full stack portfolio website that is narrative driven and showcases my skills designed to differentiate it from conventional template-based personal websites present.

The objective was to demonstrate.

- System design thinking
- Frontend architecture discipline
- Backend API Integration
- Database Modelling
- Brand Identity

The project follows a clear structural hierarchy that ensured usability and cognitive clarity.

This project evolved into a complete full-stack application deployed in production

THE PROBLEM

As a multidisciplinary techie who likes to explore (Dev + UI/UX + Design) I wanted a portfolio that reflected my creative identity, showcased my technical depth, demonstrated architectural thinking, included real backend Infrastructure and above all kept a log of my history and achievement under one roof. I reviewed different developer portfolios to come up with what I wanted and found that most of their portfolios suffer predictable patterns, they have identical UI layouts, no documented reasoning and limited backend integration.

What was the challenge?

How might I design a portfolio that has good UI, seamless UX, satisfies my artistic side, story tells who I am while being architecturally and technically complete?

DESIGN

a. Discovery

Auditing to modern developer portfolios noting the general structure and Identifying the gaps like the absence of narrative which takes away their personal tough as developers, however it is good to note that there are some awesomely done websites with a lot of personality and structure and that’s what the aim was.

b. Defining

Creating a system that merges storytelling, engineering structure and visual memory.

c. Development

It is a Newspaper-Inspired UI system with a flipping feature, utilizing a modular component structure with an API-backend contact system and MongoDB modelling.

d. Delivery

It was deployed on Netlify for the front end and Render for the backend, has MongoDB integration, cross-device responsiveness and a stable form submission lifecycle.

TECHNICAL ARCHITECTIRE

I. Frontend

React (Vite)

Tailwind CSS

Custom CSS

React Router

Modular component structure

II. Backend

Node.js

Express.js

RESTful API endpoint

Input validation & error handling

III. Database

MongoDB - Cloud Hosting

Mongoose ODM

Timestamped document schema

IV. Deployment

Frontend – MongoDB

Database – MongoDB Atlas

SYSTEM ARCHITECTURE

The high-level architecture shows the separation of concerns between the presentation layer, application logic layer, data persistence layer and cloud infrastructure. This is important as it ensured scalability, maintainability, clear responsibility boundaries and easier debugging.

Because the front end and backend are deployed on separate domains, we required CORS configuration, environment variable management for API URL and Devs VS Production handling. Instead of a decoupled frontend backend API +database instead of a purely serverless approach because I needed a clear demonstration of routing & controllers, it will be easier for expansion into other pages like admin and avoid cold start and function timeout.

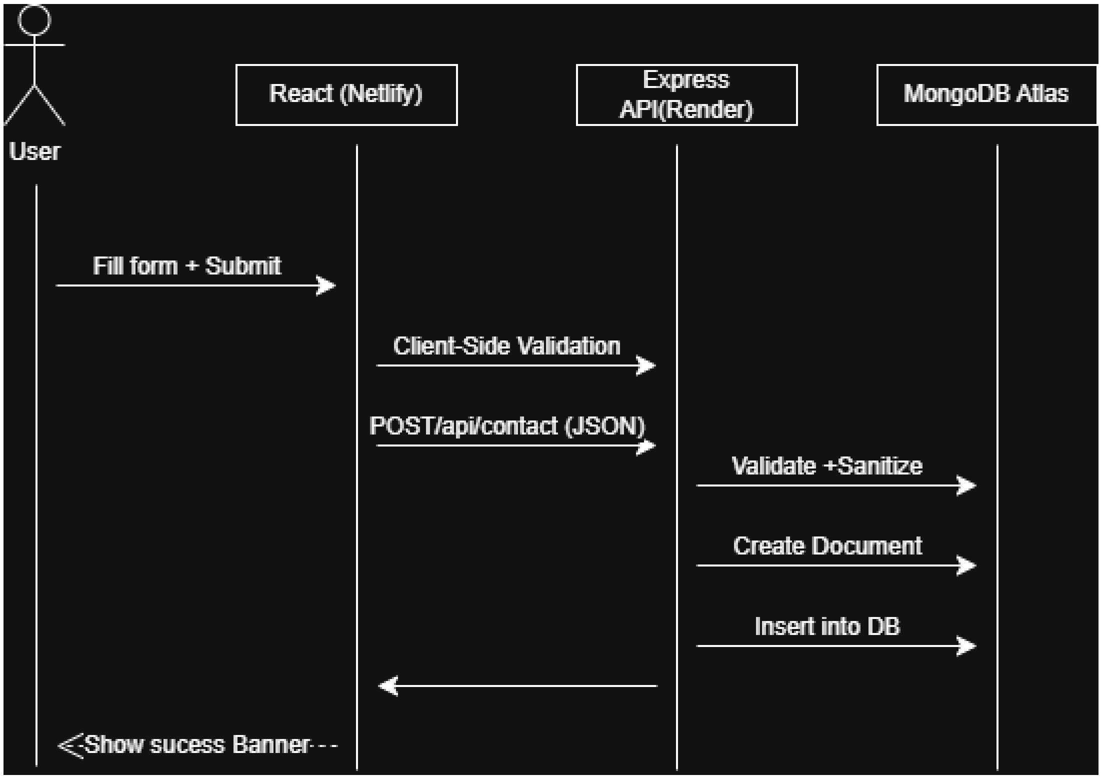


Fig 1.0 Sequence Diagram

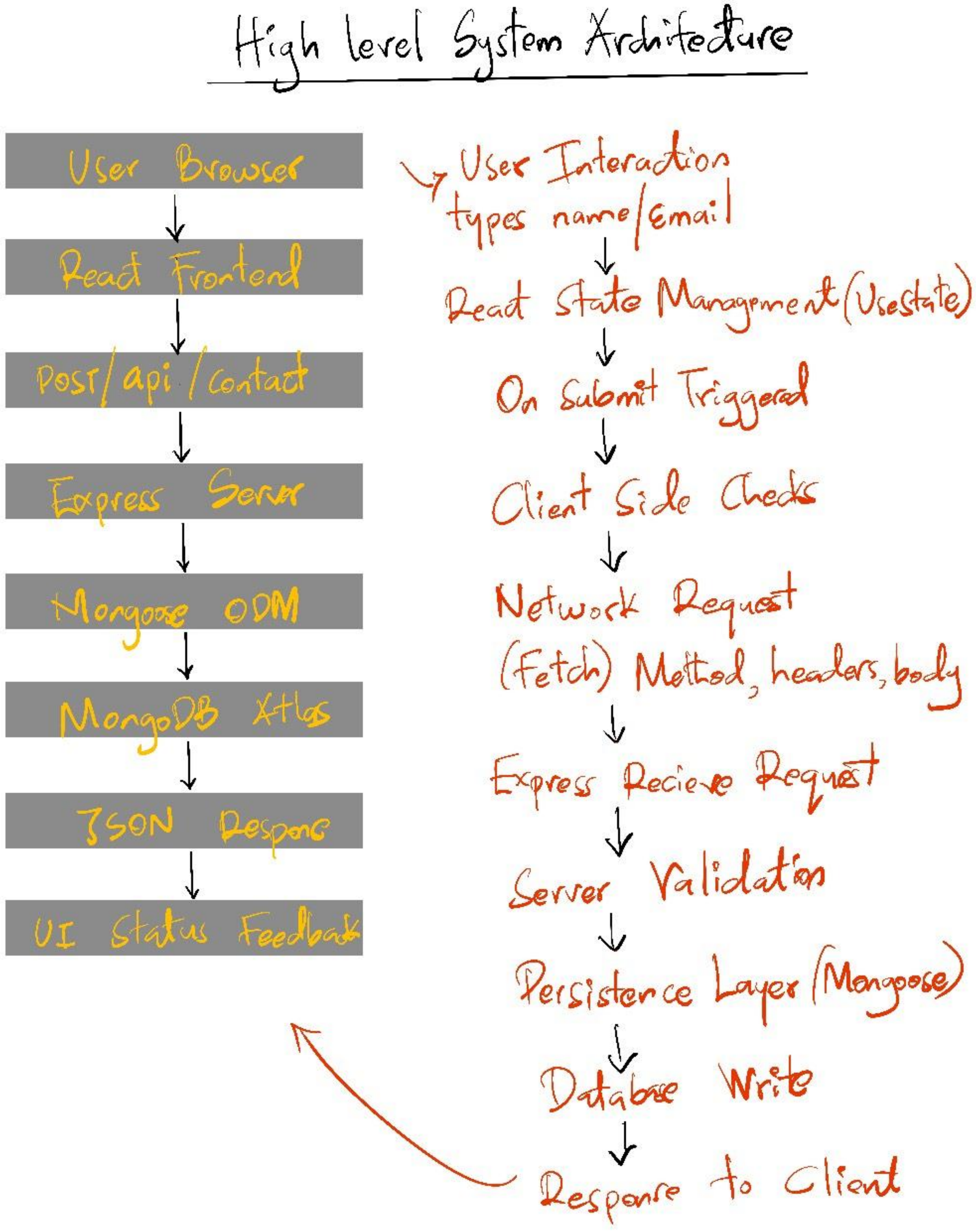


Fig 1.1 System Architecture & Request Lifecycle

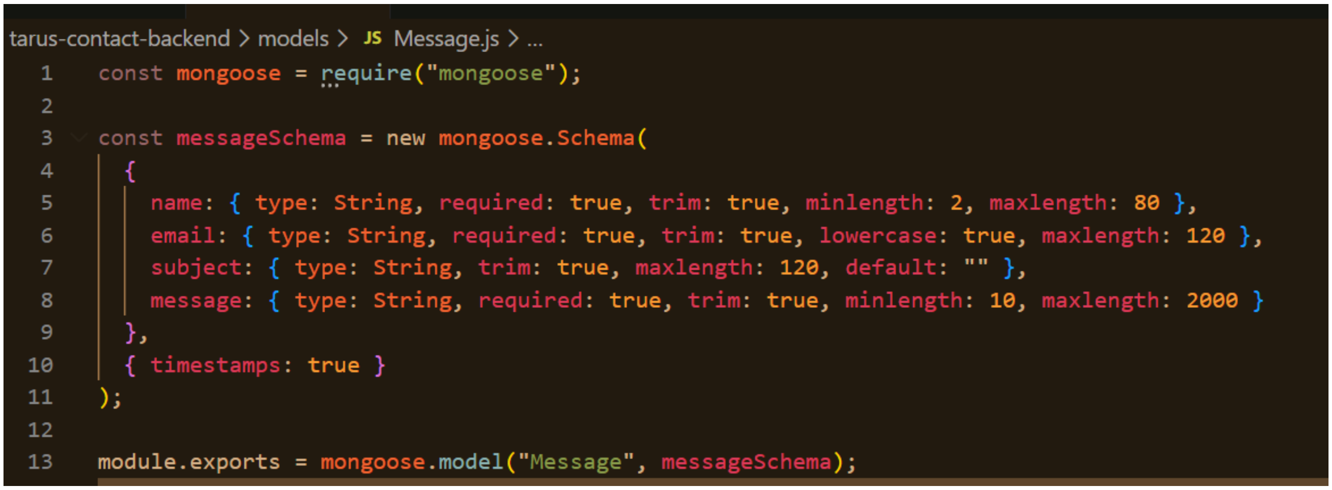
DATABASE DESIGN

I decided to use MongoDB because of its flexible document structure, Rapid iteration capability, seamless integration with Node/Express and the scalability potential it sets us on.

Field	Type	Required	Purpose
Name	String	Yes	Sender Identity
Email	String	Yes	Contact Reply Channel
Message	String	Yes	Message Content
Created at	Date	Auto	Submission Timestamp
Updates at	Date	Auto	Update timestamp

Fig 1.2 Mongoose Schema

Mongoose Schema



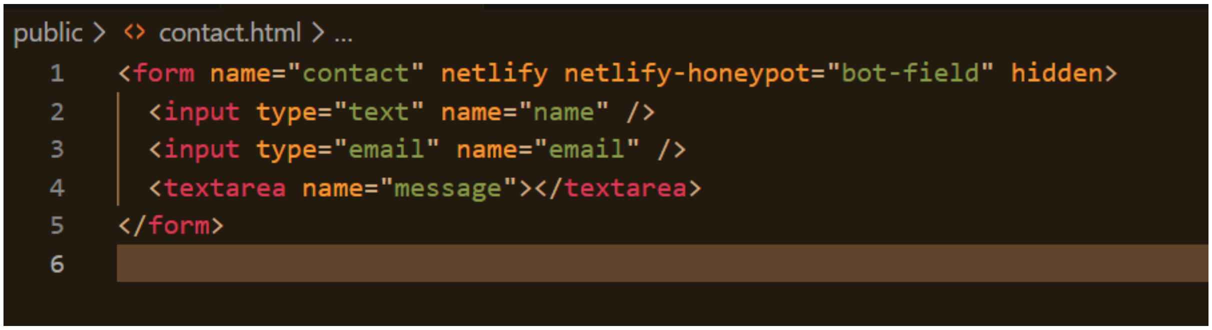
```
const mongoose = require("mongoose");

const messageSchema = new mongoose.Schema(
  {
    name: { type: String, required: true, trim: true, minlength: 2, maxlength: 80 },
    email: { type: String, required: true, trim: true, lowercase: true, maxlength: 120 },
    subject: { type: String, trim: true, maxlength: 120, default: "" },
    message: { type: String, required: true, trim: true, minlength: 10, maxlength: 2000 }
  },
  { timestamps: true }
);

module.exports = mongoose.model("Message", messageSchema);
```

Fig 1.3 Mongoose Schema

API DESIGN

This showcases the server responsibilities like valid input, sanitization of the payload, persist to MongoDB, Return structured JSON response and handling of errors gracefully

POST/api/contact



```html
<form name="contact" netlify netlify-honeypot="bot-field" hidden>
  <input type="text" name="name" />
  <input type="email" name="email" />
  <textarea name="message"></textarea>
</form>

```

Fig 1.4 Mongoose Schema

ENGINEERING DECISIONS

1. Instead of a Static Form Service from third parties, I decided to implement a custom backend so that I can demonstrate REST API design capability, Database Integration, deployment knowledge and Environment Variable Management.
2. Although it's possible to deploy all the layers combined or deploying them on GitHub pages, the frontend and backend were deployed separately to mimic real-world production architecture.
3. Used a token system so that all CSS variables were centralized to create a consistent theming, maintain design scaling and reduce the occurrence of duplication.

SECURITY

The security measures added were CORS configuration, Structured error responses e.g. server error, Production deployment separation and server-side validation.

CHALLENGES ENCOUNTERED AND SOLUTIONS

1. Managing cross-origin communication between Netlify and Render which required CORS, Environment variable, and API URL management.
2. I had to ensure that the submission had reset logic, success messaging, loading states and error handling.
3. Maintaining the general newspaper aesthetic and theme which required me to hold my artistic side not to over animate.
4. Cursor not rendering, I used a customized cursor and at some point, it was having conflicts especially when the website was moving to different screens.
5. Flip page interaction initially used the stPageFlip () Library aiming to create a realistic page flip feeling however it was hard to maintain the structure.
6. Netlify was not detecting the contact form, hence using the email automation was failing, added a contact.html so that no matter what happens Netlify could see the form.

WHAT CAN WE DO IN THE FUTURE?

1. I would make the admin dashboard to view people's submission in a continuous manner.
2. Incorporate Rate limiting and Spam filtering.
3. Analytics Integration so I can view the activities.