Project Report – Almaari Organizer

Manveer Sohal

manveersohalwork@gmail.com | 647-830-5602 |

Project Overview

Almaari Organizer is a cloud-hosted wardrobe management platform designed to help users digitally catalog and organize their clothing collections. The system enables users to upload, categorize, and search for items with ease. It integrates a cloud-based image upload pipeline, secure user authentication, and dynamic filtering capabilities to deliver a fast, responsive, and intuitive wardrobe management experience.

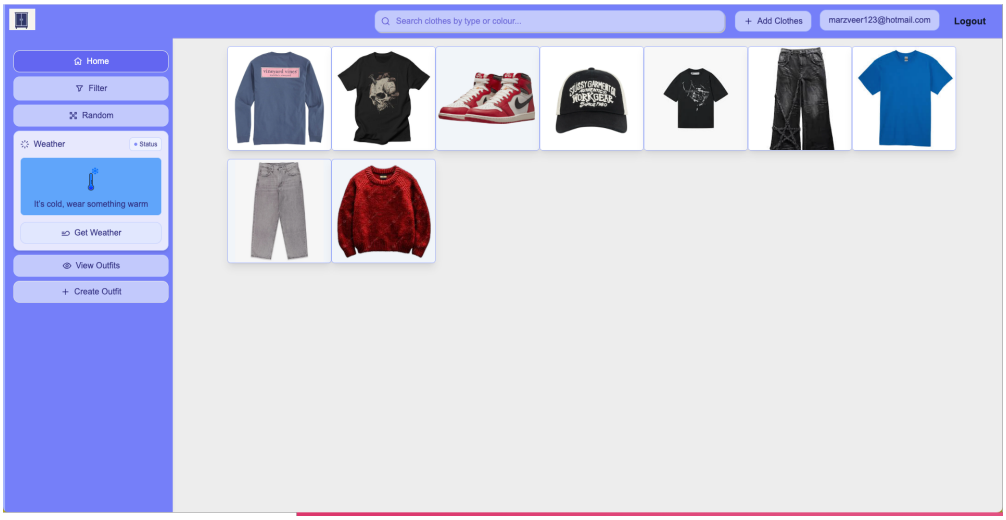

Figure 1: Main dashboard interface of *Almaari Organizer*, showcasing the clothing grid and category filters.

Technology Stack

TypeScript, React, Next.js, Node.js (Express), MongoDB, Auth0, Tailwind CSS, Docker, Google Cloud Run, AWS S3.

Key Features and Contributions

- Developed a **full-stack, Dockerized** wardrobe organizer enabling users to store, tag, and filter clothing items in real time.
- Deployed on **Google Cloud Run** for automated scaling and zero-downtime updates; integrated **AWS S3** for image storage.

- Optimized MongoDB queries and caching with **Redis**, reducing search latency by **50%**.
- Designed a responsive and minimalistic UI using **React** and **Tailwind CSS**.
- Implemented secure authentication and authorization with **Auth0**.
- Built **RESTful APIs** and configured **CI/CD pipelines** for continuous deployment.

Technical Highlight: AWS S3 Image Storage Pipeline

- Implemented a secure, scalable image upload workflow using AWS S3 presigned URLs to avoid routing large files through the backend.
- Frontend requests a **presigned URL** from the backend (generated using the AWS SDK) and uploads the image directly to S3.
- Backend stores an image reference in MongoDB for easy retrieval and categorization.

```
{
  "user_id": "123",
  "s3_key": "user_123/shirt_blue.png",
  "category": null,
  "cropped": false
}
```

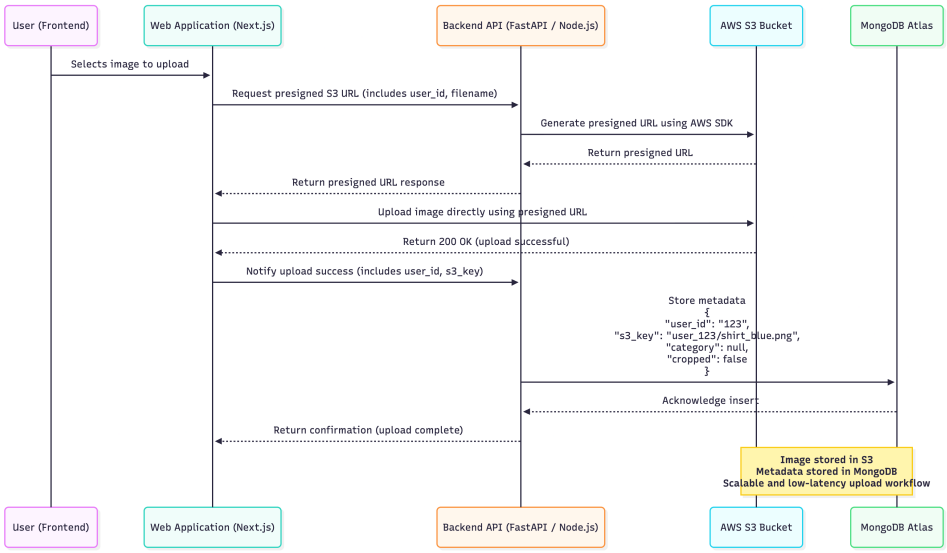

Figure 2: Sequence flowchart of AWS S3 image upload pipeline

This design improved reliability and throughput for file uploads, enabling high-volume concurrent operations with minimal backend load.

Data Architecture and Design Decisions

The database design uses **MongoDB** with three primary collections: `User`, `Clothes`, and `Outfits`. Each clothing item and outfit is stored as an independent document linked to a user ID. This normalized structure increases modularity and scalability as the system grows.

Schema Implementation (Mongoose)

```
// Clothes Schema
const ClothesSchema = new mongoose.Schema({
  uniqueId: { type: String, required: true, unique: true },
  type: { type: String, required: true, index: true },
  imageSrc: { type: String, required: true },
  favourite: { type: Boolean, default: false },
  createdAt: { type: Date, default: Date.now },
  colour: { type: [String], required: true },
  season: { type: [String], default: [] },
  waterproof: { type: Boolean, default: false },
  slot: { type: String, required: true },
});

// Outfits Schema
const OutfitsSchema = new mongoose.Schema({
  uniqueId: { type: String, required: true, unique: true },
  name: { type: String, default: "" },
  favourite: { type: Boolean, default: false },
  createdAt: { type: Date, default: Date.now },
  colour: { type: [String], required: true },
  season: { type: [String], default: [] },
  waterproof: { type: Boolean, default: false },
  outfit_items: [{ type: mongoose.Schema.Types.ObjectId, ref: "Clothes" }],
});

// User Schema
const usersSchema = new mongoose.Schema({
  auth0Id: { type: String, required: true, unique: true, index: true },
  email: { type: String, required: true },
  clothes: [{ type: mongoose.Schema.Types.ObjectId, ref: "Clothes" }],
  outfits: [{ type: mongoose.Schema.Types.ObjectId, ref: "Outfits" }],
});
```

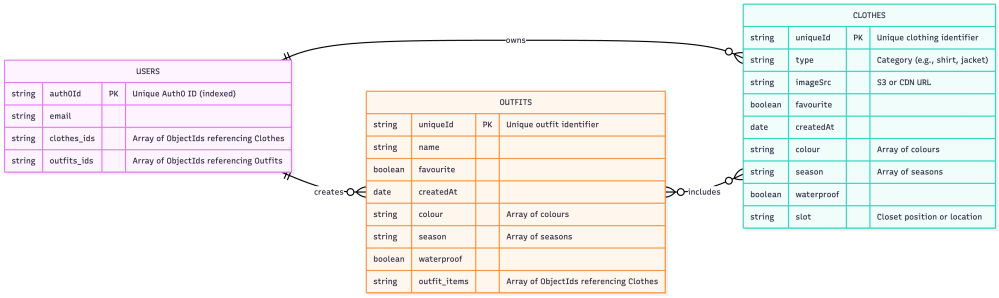

Figure 3: Database relationship diagram showing User–Clothes–Outfits references

Schema Breakdown and Efficiency Rationale

- Each clothing item is stored as a separate document rather than nested under a user, allowing **parallel indexing** and faster retrieval.
- Logical decoupling improves scalability and allows global queries (e.g., "all blue shirts across users").
- Arrays such as `colour` and `season` support flexible metadata tagging.
- MongoDB's dynamic structure allows easy addition of fields like `material` or `style` without migration.

Drawbacks and Trade-offs

While the current focus of *Almaari Organizer* is to provide a fast and reliable platform for users to store and manage their wardrobe, advanced data analytics is not a primary design requirement at this stage. The system's architecture is optimized for real-time interactions—quick uploads, responsive filtering, and seamless retrieval—rather than heavy read operations or large-scale analytical queries. Consequently, MongoDB was chosen for its flexibility, ease of scaling, and ability to handle frequent write operations efficiently. However, if deeper analytics (e.g., identifying most worn colors or seasonal trends) were to become core features, the current data model would face performance challenges due to MongoDB's limited aggregation efficiency.

- MongoDB's unstructured format is not ideal for analytical queries; aggregations like *"most worn color by month"* require memory-heavy pipelines.
- **PostgreSQL** would perform better for analytical workloads due to structured joins and efficient SQL aggregations.
- A hybrid solution—MongoDB for real-time operations and PostgreSQL for analytics—would balance performance and flexibility.

Impact and Outcomes

- Reduced API response times by **25%** via query optimization and indexing.
- Implemented **Redis caching** to accelerate data reloads and improve application responsiveness after page refreshes.
- Designed a scalable data architecture supporting seamless cloud deployment and user growth.