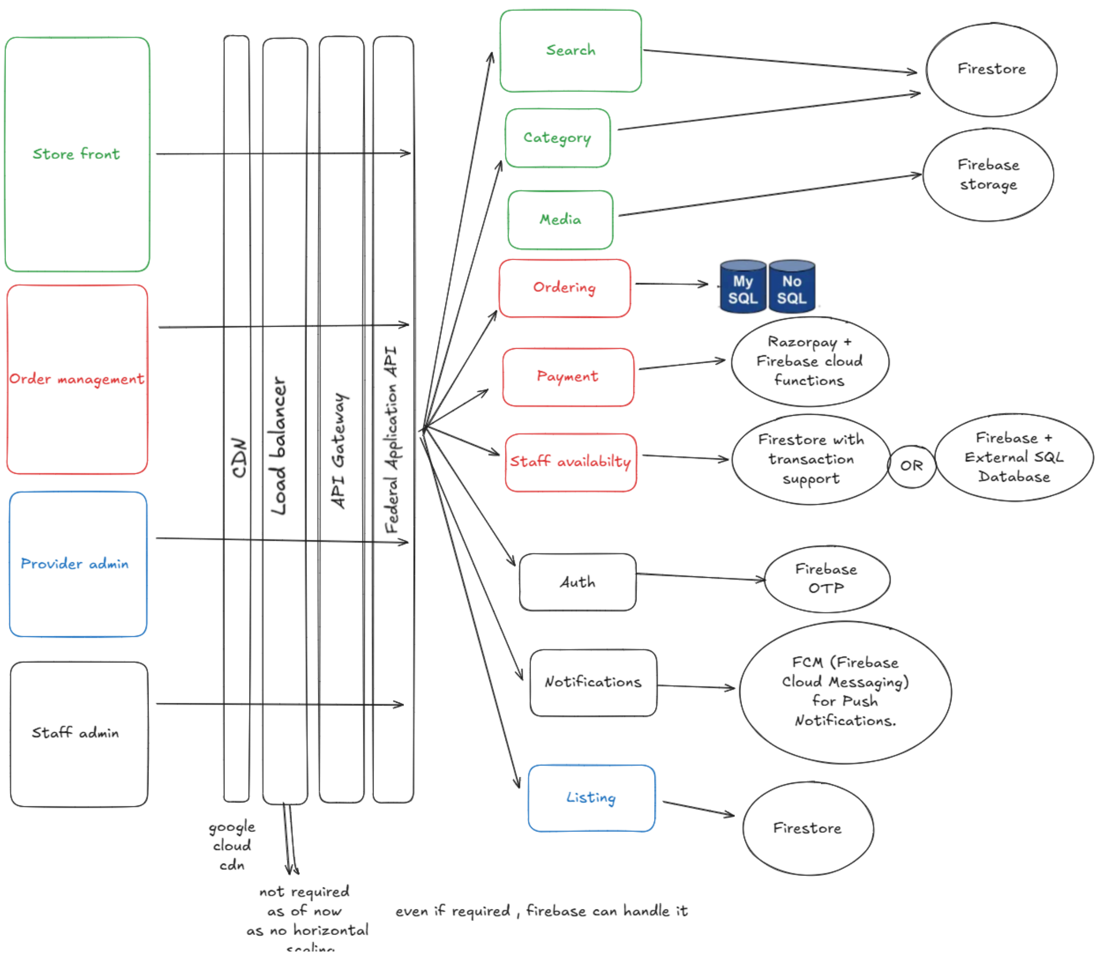
Nurch - system design

TLDR



Requirements

Providers list their services , users search, book, pay for the service

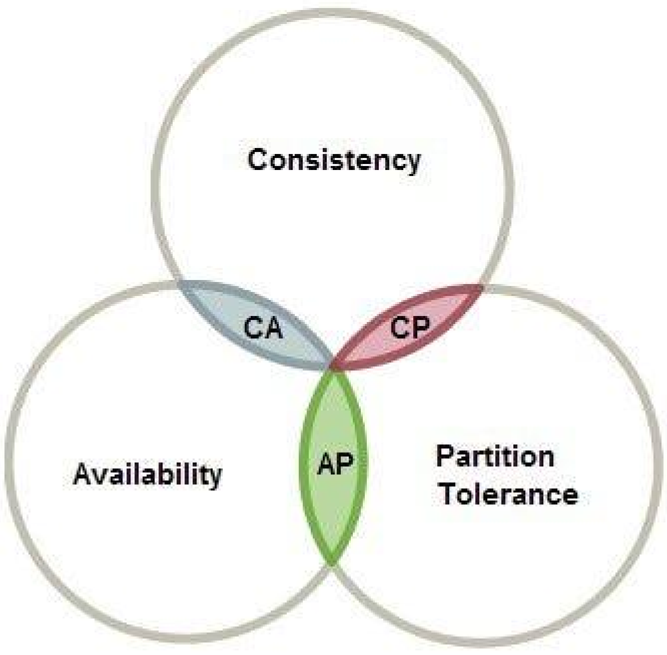
Scale

Not big of an issue as of now since developing an MVP , even after , users can be very few compared to ecom sites

Core Building Blocks of Data

1. Listing : title, description,staff. One service can be sold by multiple Providers and can be located in different regions.
2. Merchandisable Item: Listing, Provider, Cost, Price, staff availability Info
3. Order: listing, Address, Customer, Order Status, etc
4. Address: Street, City, State, Zip Code, Country, Coordinates, Phone
5. User: phone number , Name, Can be a Customer/provider/staff
6. Customer: User, Addresses, Payment Instruments
7. Provider : User, Staff info, listing info
8. Staff : User, provider info , availability status (self marking)

Data Store Choices: CAP theorem tradeoffs



Listing Storage

Needs to be horizontally scalable and *highly available*; AP out of CAP.
High read/write ratio. Attributes can diverge by service type/worker class/additional requirements of the user, requiring a flexible structure.

Based on the above, Listing metadata such as name, description, localizations, provider info , staff info , etc (excluding product media images/videos storage) can go into a document store that supports flexible query forms, and to support text-based search. **ElasticSearch** can be considered.

But , we use - Firestore by firebase as :

- Horizontally Scalable
- Highly Available
- Flexible Schema
- Optimized for High Read/Write speeds

Multimedia Storage

Images, documents and videos are relatively larger and would require a cost-efficient, highly-available store to persist those files.

Firebase storage !!

Future scope : Cloud blob storage comes into play here, like **AWS S3**, **Google Cloud Storage**, or **Azure Blob Storage**. These services provide cheap, replicated and hence HA storage for object storage.

Content Delivery Network:

Firebase Hosting (Built-in CDN - Google Cloud CDN)

- Globally Distributed: Uses Google Cloud CDN, ensuring fast static file delivery.
- Auto-Optimized Routing: Serves images, scripts, and videos from nearest edge locations for low-latency access.
- Custom Caching Rules: Set TTL (Time-to-Live) values for optimal cache expiration.

Staff availability Storage

The need is clear for *consistency* in how data is managed; CP out of CAP.
This data enjoys a well-defined, generally stable structure and requires ACID guarantees, which can be offered by an RDBMS like **Postgres**, **MySQL**, **Oracle**, or **MS SQL Server**.

2 options :
1. Firestore with transaction support

2. Firebase + External SQL Database

Order Storage

A submitted Order is an entity composed of different elements of data and references, i.e. relations. Orders carry state info, data which would be required to handle, fulfill, or cancel the order.
Consistency is required, at least until the order is completed, reaching an end state (e.g. delivered or canceled); and hence, CP. Hence, an RDBMS would serve us very well for storing and managing orders.

Potential Optimization

For orders that get completed, one optimization that can be considered is to move those to a more readily scalable, and highly available system as strict consistency becomes less of a concern making eventual consistency acceptable. **Wide-column** NoSQL solutions can be considered here, like **Cassandra** or **DynamoDB**.

Identity Storage

Auth - Firestore OTP

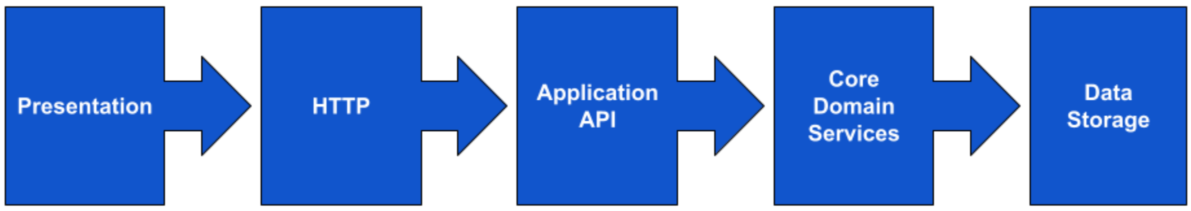
Personal info - firestore

Architecture

Key architectural decisions for this system:

Layered Architecture

Components are grouped into layers, where each layer performs a specific role within the overall system.
Each layer is isolated and independent of the other layers, having no knowledge of the inner workings of other layers in the system. Well-defined APIs are exposed to be used by, exclusively, the higher layers.



Layered Architecture with isolated layers communicating from left to right.

The layers in our architecture are:

1. **Presentation**: User Interfaces
2. **HTTP**: Load balancing, Rate Limiting, API Gateway, Authentication, etc
3. **Application API**: Federated API to enable different product use cases by communicating with core domain services
4. **Core domain services**

💡 Key Components in the Architecture

1. User (Client) App

- Allows users to search for elderly care services.
- Filters results based on location, date, service type, and additional needs.
- Books services and makes payments.
- Tracks assigned staff in real time (via Firestore).
- Leaves ratings & reviews.

2. Staff Dashboard

- OTP-based login via Firebase Authentication.
- Uploads personal details, ID documents (Aadhar, certificates).
- Sets service preferences (medicine assistance, household help, etc.).
- Clocks in/out and shares live location on duty.
- Accepts or rejects job assignments.
- Attendance is tracked using Firestore and Cloud Functions.

3. Provider Dashboard

- Manages staff profiles and verifies documents.
- Assigns jobs to staff and tracks their location and attendance.
- Reviews staff ratings & feedback.
- Manages payroll (integrated with payment gateway via Cloud Functions).

🔥 High-Level Firebase-Based System Architecture

Frontend (React.js / Next.js for all 3 Apps)

- Calls Firebase Authentication for login (via OTP).
- Uses Firestore for reading/writing data.
- Uses Firebase Storage for document uploads.
- Uses Google Maps API for location-based services.
- Uses Firestore subscriptions for real-time updates.
- Uses **Service Workers** for offline caching of API responses.
- Implements **IndexedDB** to cache previous search results.

- Loads static assets via **Firebase Hosting CDN**.

Backend (Firebase Services)

1. **Authentication**
    - Firebase Auth for OTP-based login.
    - Role-based access (User, Staff, Provider).
2. **Database (Firestore + Cached Reads)**
    - Users, Staff, and Providers stored as separate collections.
    - Real-time job assignments and status updates.
    - Attendance logs and check-in/check-out tracking.
    - Frequently accessed collections (Users, Providers, Services) are cached in **Redis**.
    - **Firestore cache rules** to allow client-side reads without hitting the backend frequently.
3. **Storage (Firebase Storage)**
    - Stores images, documents, and certificates.
    - Uses **Cloudflare CDN** for optimized delivery.

4. **Cloud Functions(Job Assignment, Notifications, Payments)**
    - Assigns jobs automatically based on availability.
    - Sends notifications (FCM) for job updates.
    - Handles payment processing with Razorpay/Stripe.
    - Manages scheduled tasks (e.g., auto-checkout reminders).
    - Caches job assignment data to reduce re-computation.
    - Uses **Redis Pub/Sub** to store pending job assignments.

5. **Real-time Features**
    - Firestore real-time updates for job status tracking.
    - Location tracking updates optimized with **rate-limiting cache**.

Caching Strategies:

1. **Product Data Caching**
    - Cache frequently accessed service provider data (e.g., staff profiles, provider details).
    - Use **Firebase Remote Config** for storing less frequently updated settings.
    - Use **Redis** (via Upstash or Firebase Extensions) for faster reads.
2. **Static Content Caching**
    - Cache frontend static assets (images, JS, CSS) using **Firebase Hosting CDN**.

    - Use **Cloudflare CDN** or **Google Cloud CDN** for additional optimization.
3. **Frequently Accessed Search Queries Caching**
    - Cache popular search queries (e.g., caregivers in a city, nurses available this week).
    - Use **Redis** (hosted on Upstash or Google Cloud Memorystore).
    - Store precomputed results in **Firestore Cache Layer** (using Firebase Firestore Rules).
    - Implement **IndexedDB** caching in the frontend for faster offline support.