Technical Spec Document

Here is the Technical Specification Document for the 28HSE Assignment Platform.

This document is written for developers. It translates the business requirements into concrete code structures, database schemas (Prisma), and architectural decisions based on the Next.js + Express + PostgreSQL + AWS stack.

🛠️ Technical Specification Document

Project: 28HSE Assignment Platform

Version: 1.0

Date: November 24, 2025

Tech Stack: MERN-SQL Hybrid (Next.js, Express, PostgreSQL, Prisma, AWS)

1. System Architecture

1.1 High-Level Overview

The system follows a Decoupled Client-Server Architecture.

- Frontend (Client): Next.js handles the UI, Server-Side Rendering (SSR) for SEO-critical public pages, and static generation for dashboards.
- Backend (API): A standalone Express.js application acts as the REST API gateway, business logic layer, and background worker host.
- Database: PostgreSQL managed via Prisma ORM.
- File Storage: AWS S3 for secure, encrypted document storage.

1.2 Infrastructure Diagram (AWS Canada Central)

Code snippet

```
graph TD
  Client[Next.js Frontend] -->|HTTPS/JSON| API[Express API Server (AWS App Runner/EC2)]
  API -->|ORM| Prisma[Prisma Client]
  Prisma -->|SQL| DB[(AWS RDS PostgreSQL)]
  API -->|Uploads| S3[AWS S3 Private Bucket]
  API -->|Queues/Cache| Redis[(AWS ElastiCache)]
  API -->|Email| SendGrid[SendGrid API]
  API -->|SMS| Twilio[Twilio API]
```

2. Database Schema (Prisma)

We will use Prisma Schema (schema.prisma) to define the data models. This replaces Mongoose schemas but provides similar developer ergonomics with better type safety.

2.1 Enums

Code snippet

```
enum Role {
  HOMEOWNER
  REALTOR
  ADMIN
}

enum SubscriptionTier {
  FREE
  BASIC_AGENT
  PRO_AGENT
}

enum VerificationStatus {
  UNVERIFIED
  PENDING
  VERIFIED_OWNER
  REJECTED
}

enum LeadPoolStatus {
  NOT_IN_POOL
  ACTIVE_TIER_1  // Visible to Pro Agents
  ACTIVE_TIER_2  // Visible to Basic Agents
  CLAIMED
}
```

2.2 Models

User Model

Code snippet

```
model User {
  id             String   @id @default(uuid())
  email          String   @unique
  passwordHash     String
  firstName       String
  lastName        String
  phone          String   @unique // For MFA
  role           Role     @default(HOMEOWNER)
  subscriptionTier  SubscriptionTier @default(FREE)
  creditsBalance   Int      @default(0)

  // Relations
  listings        Listing[]
  claimedLeads     Listing[] @relation("ClaimedLeads")
  transactions     Transaction[]

  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt
}
```

Listing Model

Code snippet

```
model Listing {
  id             String   @id @default(uuid())
  ownerId         String
  owner          User     @relation(fields: [ownerId], references: [id])

```

```
// Public "Teaser" Data
neighborhood    String
bedroomCount    Int
bathroomCount   Int
approxSqFt      Int
completionDate  DateTime
developerInitials String

// Private "Member" Data
projectName     String
unitNumber      String
originalPrice   Decimal  @db.Decimal(12, 2)
askingPrice     Decimal  @db.Decimal(12, 2)
depositPaid     Decimal  @db.Decimal(12, 2)
assignmentFeePercent Decimal @default(1.0)

// Compliance & Verification
verificationStatus VerificationStatus @default(UNVERIFIED)
contractDocS3Key  String?  // Path to redacted PDF in S3

// Lead Pool Logic
leadPoolStatus   LeadPoolStatus @default(NOT_IN_POOL)
poolEntryAt      DateTime?
assignedAgentId  String?
assignedAgent    User?   @relation("ClaimedLeads", fields: [assignedAgentId], references: [id])

inquiryCount    Int    @default(0)
createdAt       DateTime @default(now())
```

```
}
```

Transaction Model

Code snippet

```
model Transaction {
  id          String   @id @default(uuid())
  userId      String
  user        User     @relation(fields: [userId], references: [id])
  amount      Int      // Positive for buy, negative for spend
  description String
  referenceId String?  // ID of Listing or Stripe Charge
  createdAt   DateTime @default(now())
}
```

3. API Design (Express.js)

Since you are using Express, structure the application using the Controller-Service-Repository pattern to keep logic clean.

3.1 Authentication

- Library: jsonwebtoken + bcryptjs.
- Middleware: authMiddleware.js extracts Bearer token, verifies signature, and attaches req.user.

3.2 Lead Pool Logic (The "Stagnation" Cron)

This service runs inside the Express app using node-cron.

JavaScript

```javascript
// services/leadPoolService.js


const promoteStagnantListings = async () => {
  // 1. Find listings older than 14 days with < 3 inquiries
  const cutoffDate = new Date();
  cutoffDate.setDate(cutoffDate.getDate() - 14);


  const stagnantListings = await prisma.listing.findMany({
    where: {
      createdAt: { lt: cutoffDate },
      inquiryCount: { lt: 3 },
      leadPoolStatus: 'NOT_IN_POOL'
    }
  });

  // 2. Transactional Update
  await prisma.$transaction(
    stagnantListings.map(listing =>
      prisma.listing.update({
        where: { id: listing.id },
        data: {
          leadPoolStatus: 'ACTIVE_TIER_1',
          poolEntryAt: new Date()
        }
      })
    )
  );

  // 3. Trigger Notifications (Queue)
  // ... sendPushNotificationToProAgents()
};
```

3.3 The "Claim Lead" Endpoint (Optimistic Locking)

This is critical for preventing double-claims.

JavaScript

```
// controllers/leadController.js

const claimLead = async (req, res) => {
```

```
const { listingId } = req.body;
const agentId = req.user.id;
const COST = 50;

try {
  const result = await prisma.$transaction(async (tx) => {
    // 1. Check Agent Balance
    const agent = await tx.user.findUnique({ where: { id: agentId } });
    if (agent.creditsBalance < COST) throw new Error("Insufficient Credits");

    // 2. Attempt to Lock & Claim Listing
    // The 'update' will fail if the status has already changed
    const updatedListing = await tx.listing.update({
      where: {
        id: listingId,
        leadPoolStatus: { in: ['ACTIVE_TIER_1', 'ACTIVE_TIER_2'] } // Guard
      },
      data: {
        leadPoolStatus: 'CLAIMED',
        assignedAgentId: agentId
      }
    });

    // 3. Deduct Credits
    await tx.user.update({
      where: { id: agentId },
      data: { creditsBalance: { decrement: COST } }
    });
    // 4. Log Transaction
    await tx.transaction.create({
      data: {
        userId: agentId,
        amount: -COST,
        description: `Claimed Lead: ${listingId}`
      }
    });

    return updatedListing;
  });

  res.json({ success: true, listing: result });

 } catch (error) {
  if (error.code === 'P2025') { // Prisma error for record not found (failed guard)
    return res.status(409).json({ error: "Lead already claimed by another agent." });
  }
  res.status(500).json({ error: error.message });
 }
};
```

4. Frontend Specifications (Next.js)

4.1 SEO Strategy (Public Teaser Views)

- Page: /assignments/[neighborhood]/[id]
- Rendering: SSR (getServerSideProps) or ISR.
- Logic: Fetch data from Express API. If User is NOT logged in, the API returns the "Teaser DTO". If logged in, API returns full data.
- Meta Tags: Dynamically generate Title/Description based on "Neighborhood + Bedroom Count" (e.g., "1 Bed Assignment in Brentwood - $600k Range").

4.2 Document Upload (Client-Side Direct to S3)

Do not pipe large PDFs through the Express server.

1. **Client: Request pre-signed URL from API.**
   - **GET /api/upload/sign?fileName=contract.pdf**
2. **API: Generate AWS S3 Presigned URL (PUT method, 60s expiry).**
3. **Client: PUT the file directly to S3.**
4. **Client: Send the S3 Key to the API to save the record.**

5. Security & Compliance

5.1 Data Masking Middleware

To ensure REDMA compliance, create a utility function that sanitizes listing objects before sending them to the frontend.

JavaScript

```javascript
// utils/sanitizer.js
const sanitizeListing = (listing, user) => {
  if (!user) {
    // Public Teaser View
    return {
      id: listing.id,
      neighborhood: listing.neighborhood,
      bedrooms: listing.bedroomCount,
      completion: listing.completionDate,
      priceRange: estimateRange(listing.askingPrice), // "500k-600k"
      developerInitials: listing.developerInitials,
      // STRIP EVERYTHING ELSE
    };
  }
  return listing; // Full View for Members
};
```

6. Implementation Checklist

Phase 1: Foundation

- **[ ] Set up AWS RDS (Postgres) and connect Prisma.**
- **[ ] Implement Express Auth (JWT + Password Hashing).**
- **[ ] Build Next.js "Landing Page" and "Dashboard Skeleton".**

Phase 2: Core Features

- **[ ] Listing CRUD: Implement Teaser vs. Full logic.**
- **[ ] Verification: AWS S3 Bucket setup + Presigned URL endpoint.**
- **[ ] Tax Calc: Implement BCHomeFlippingTax logic in frontend utility.**

Phase 3: The Revenue Engine

- **[ ] Stripe: Create Payment Intents for Credit Packs.**
- **[ ] Lead Pool: Implement the node-cron job and Transactional Claim logic.**
- **[ ] Deployment: Dockerize Express app; Deploy Next.js to Vercel or AWS Amplify.**

Why this works for you:

1. **Express Logic: You can write standard Node.js logic for the backend.**
2. **Prisma: Saves you from writing raw SQL but gives you the safety of SQL.**
3. **Next.js: Handles the hard part of making the site "Public" for Google while keeping the "Private" parts secure.**