Author: TS
Date: 11-Aug-25
DRAFT

HAILIE Insights Engine DRAFT

Architectural Overview

A modern, scalable web application built using a **three-tier architecture** consisting of a frontend (client-side), a backend (server-side), and a database. For this application, a **cloud-native approach** is used; leveraging managed services to enhance scalability, reliability, and security while reducing operational overheads.

- **Cloud Provider: Amazon Web Services (AWS)** or **Microsoft Azure**. Both have data centers in the UK, which is crucial for data residency and GDPR compliance. For this blueprint, I will use AWS services as examples.

- **Core Philosophy:** Build a **monolithic application to start**, as it's simpler to develop and deploy initially. As the application grows in complexity, you can strategically break off services into a **microservices architecture**.

Functional Requirements: Technical Decisions

1. User Authentication & Authorization

- **Service: AWS Cognito** or **Auth0**.
- **Reasoning:** These are managed "Identity as a Service" platforms that handle the complexities of user registration, login, password management (including resets), multi-factor authentication (MFA), and social logins out of the box. They are highly secure, scalable, and provide role-based access control (RBAC). Using a managed service saves significant development time and reduces security risks compared to building a custom solution.
- **Implementation:**
  - Use AWS Cognito User Pools to manage your user directory.
  - Define user groups within Cognito (e.g., Admin, HousingProvider_A, HousingProvider_B).
  - When a user logs in, Cognito will return a JSON Web Token (JWT). This token

will be sent with every API request from the frontend to the backend.
    - The backend will validate the JWT and use the information within it (like the user's role/group) to authorize access to specific data.

2. Data Upload, Import & Processing

This is a critical, multi-step process.

- **Frontend (Upload):**
    - **Technology:** A JavaScript framework like **React.js** or **Vue.js**.
    - **Library:** Use a library like react-dropzone for a user-friendly drag-and-drop upload interface.
    - **Process:** The frontend will send the Excel/CSV file directly to a secure, pre-signed URL provided by the backend. This prevents large files from bogging down your main application server.
- **Backend (Processing):**
    - **Storage:** The uploaded file lands in an **AWS S3 bucket**.
    - **Trigger:** The file upload to S3 triggers an **AWS Lambda function**. This is a serverless approach, meaning you only pay for the compute time when a file is being processed.
    - **Parsing:** Inside the Lambda function, use a robust library to parse the data. For Node.js, SheetJS is excellent for Excel files and csv-parser for CSVs.
    - **Validation:** The Lambda function will perform rigorous validation against a predefined schema (e.g., checking for property_id, postcode, rent_amount columns and correct data types). If validation fails, update the database with a "failed" status and notify the user.
    - **Asynchronous Task:** For large files, the initial Lambda function's job is just to validate and then place a message into an **AWS Simple Queue Service (SQS) queue**. A separate, more powerful compute service (like **AWS Fargate** or another Lambda function with a longer timeout) will pull messages from the queue and perform the heavy lifting of importing the data into the database. This ensures the initial upload request is fast and reliable.

3. Database

- **Technology: PostgreSQL** (managed via **AWS RDS - Relational Database Service**).
- **Reasoning:**
    - **Relational:** Your data is structured, making a relational database a natural fit.
    - **Powerful:** PostgreSQL has excellent support for complex queries, indexing, and data integrity. It also has powerful extensions like PostGIS if you ever need to perform geospatial analysis on property postcodes.

    - **Managed:** AWS RDS handles backups, patching, and scaling, which is a huge operational win.
- **Schema Design:**
    - Create tables for Users, HousingProviders, UploadedFiles (with status like processing, completed, failed), and the core data tables (e.g., Properties, Tenancies).
    - Crucially, every table containing provider-specific data must have a housing_provider_id column. All queries must filter by this ID to ensure data segregation.

4. Data Analysis, Querying & Visualization

- **Backend (API):**
    - **Language/Framework: Node.js with Express.js** or **Python with Django/Flask**. Node.js is a good choice for its non-blocking I/O, which works well for data-intensive applications.
    - **ORM:** Use an Object-Relational Mapper like **Sequelize** (for Node.js) or **SQLAlchemy** (for Python) to safely construct SQL queries and prevent SQL injection.
    - **API Design:** Create a RESTful API with endpoints like GET /api/insights/rent-average or a more flexible **GraphQL API** which would allow the frontend to request the exact data it needs for a specific visualization.
- **Frontend (Visualization):**
    - **Libraries:** Use a powerful charting library like **Chart.js**, **D3.js** (for highly custom visualizations), or a commercial library like **Highcharts**. These libraries can take JSON data from your backend and render interactive charts and graphs.

Non-Functional Requirements: Technical Decisions

- **Scalability:**
    - **Compute:** Use **AWS Elastic Beanstalk** or **AWS Fargate** for your backend API. These services automatically scale the number of running instances based on traffic, ensuring performance during peak loads.
    - **Database:** AWS RDS allows you to scale your database instance size with a few clicks. For read-heavy workloads, you can add read replicas to distribute the load.
    - **Serverless:** Using Lambda for data processing is inherently scalable.
- **Accessibility:**

    - **Standard:** Adhere to **WCAG 2.1 AA** guidelines.
    - **Frontend:** Use semantic HTML5 (e.g., <nav>, <main>, <button>) from the start.
    - **Tools:** Integrate automated accessibility checking tools like axe-core into your development pipeline. Regularly perform manual audits using screen readers (e.g., NVDA, VoiceOver).
- **Security:**
    - **Authentication:** Handled by AWS Cognito (as above).
    - **Encryption:**
        - **In Transit:** Enforce HTTPS on all connections using an **AWS Certificate Manager (ACM)** certificate.
        - **At Rest:** Enable encryption on your AWS RDS database and your S3 buckets. This is a simple checkbox setting.
    - **Compliance:** Store all data within AWS regions located in the UK (eu-west-2 in London) to comply with data residency requirements.
    - **Vulnerability Scanning:** Use **AWS Inspector** to automatically scan your infrastructure for vulnerabilities.
- **Usability (UX/UI):**
    - **Frameworks:** Use a component library like **Material-UI (MUI)** for React or **Vuetify** for Vue.js. These provide pre-built, accessible, and professional-looking UI components, which drastically speeds up development and ensures a consistent user experience.
- **Reliability & Availability:**
    - **Infrastructure:** Deploy your application across multiple **Availability Zones (AZs)** within the AWS London region. An AZ is a distinct data center. If one fails, your application will continue to run in another. AWS Elastic Beanstalk and RDS can be configured to do this automatically.
    - **Backups:** Configure automated daily snapshots of your RDS database.
- **Performance:**
    - **Content Delivery:** Use **AWS CloudFront**, a Content Delivery Network (CDN), to cache your frontend assets (JavaScript, CSS, images) closer to your users, reducing latency.
    - **Caching:** Implement a caching layer using **Redis** (managed via **AWS ElastiCache**) to store the results of frequent or expensive database queries.
    - **Database:** Add indexes to your database tables on columns that are frequently used in WHERE clauses (e.g., housing_provider_id, postcode).
- **Maintainability:**
    - **Version Control:** Use **Git** and host your repository on **GitHub** or **AWS CodeCommit**.
    - **CI/CD:** Implement a Continuous Integration/Continuous Deployment pipeline

using **GitHub Actions** or **AWS CodePipeline**. This will automatically test and deploy your code, ensuring a reliable and repeatable deployment process.

- **Infrastructure as Code (IaC):** Use a tool like **AWS CloudFormation** or **Terraform** to define your entire cloud infrastructure in code. This makes it easy to replicate, modify, and version control your infrastructure.