Development Team Project:

Project Report

Kanan Akbarov, Matilda Nilsson, Faiz Saiyed

Abstract

This report outlines the logical database design for *ZeroTrace*, a digital platform supporting individuals with disabilities through anonymous and stigma-free assistance. Developed for DOST RIM (Digital Innovations Center), the design emphasizes user privacy, accessibility, and data integrity. Key components include a relational schema, a PostgreSQL-based DBMS proposal, and a data pipeline with integrated cleaning mechanisms.

1. Project Context and Objective

As software consultants, our team was tasked with developing a logical database for a client organization. We chose DOST RIM, a leader in inclusive services, and designed *ZeroTrace* to serve individuals with disabilities seeking anonymous help. The platform enables private, safe exchanges between users and verified helpers (e.g., volunteers or professionals), addressing stigma-related barriers in conventional support systems.

2. About ZeroTrace

Despite increasing digital access, users with disabilities often face judgment when requesting help. *ZeroTrace* tackles this by allowing anonymous questions and expert responses within a secure, inclusive environment. Key features include:

- Anonymous question submission
- Verified helper responses
- Accessibility-compliant interface
- Privacy-focused architecture

3. Database Design Overview

The database design supports *ZeroTrace*'s need for structured user data, classification of disabilities, and tracking of assistance interactions. The model reflects:

- Core user data (e.g., age, disability, preferences)
- Tiered classification of disabilities
- Support requests with urgency and status
- Helper interactions and system logs

Normalization and scalability were prioritized to ensure efficient future integration with external systems (e.g., healthcare registries).

4. Logical Design

Key entities and attributes include:

- **Users**: ID, role, demographic info
- **Disabilities**: Specific conditions tied to users
- **Disability Categories**: High-level groupings
- **Helpers**: User subset with credentials
- **Assistance Requests**: Urgency, timestamps, status
- **Responses**: Helper answers, time logs

The schema *(Visual 1)* minimizes redundancy and supports modular expansion (e.g., healthcare integration). Relationships use foreign keys for referential integrity, while ENUM types enforce consistency in categorical fields (e.g., gender, urgency).

5. Proposed Database Model and Technology

We propose implementing the model using **PostgreSQL** due to:

- Advanced data type support (e.g., UUIDs, ENUMs)
- SQL compliance and scalability
- Strong community and documentation
- Security and performance reliability

Key features include:

- **UUIDs** for primary keys: Ensure privacy and uniqueness
- **ENUMs**: Used for gender, role, urgency, and request status
- **DATE**, **VARCHAR**, **TIMESTAMP**: For accurate time and textual data
- **Foreign Keys**: Maintain relationships and integrity across entities

As seen in *Visual 2,* the structure provides maintainability, secure access, and flexibility for long-term scaling.

6. Data Flow and Cleaning Process

Data is collected via digital forms filled by users or volunteers. To ensure validity at the point of entry, fields (e.g., email, phone) use real-time format validation. In visuals 3 and 4, it is highlighted that the system follows a structured data flow represented across three DFD levels:

- **Level 0**: High-level user-system interaction
- **Level 1**: Submission, matching logic
- **Level 2**: Detail on filtering, matching, and data handling

Data cleaning includes:

- **Deduplication**: Eliminating redundant records
- **Standardization**: Unified formatting (e.g., date, name capitalization)
- **Validation**: Real-time and post-entry checks
- **Defaults**: Assigned to non-critical missing values

Cleaning is performed using SQL scripts directly on tables, preserving schema integrity and ensuring dataset consistency. *(Tables 3 and 4. Visual 6)*

7. Critical Evaluation

The ZeroTrace database fulfills the application's immediate data needs through structured entities, logical relationships, and PostgreSQL's robust capabilities. Using normalization and constraint enforcement ensures reliable, non-redundant data storage.

PostgreSQL enhances performance and supports advanced querying, while SQL-based cleaning processes improve data quality. ENUM use standardizes categorical values across the system.

8. Future Scaling

As our system evolves and the number of users and data volume continues to grow, ensuring the scalability of our database becomes essential. To prepare for future demand, we plan to implement horizontal scaling through techniques such as sharding and replication. This will allow the database to handle increased loads without sacrificing speed or reliability. From a business standpoint, this approach gives us the flexibility to expand our services and onboard more users without requiring a complete system redesign. We will also continue optimizing our indexing strategies and query structures to maintain performance. The scaling plan is aimed at keeping the system efficient, cost-effective, and ready to support long-term growth.

9. Recovery Plan

A proposed recovery plan for ZeroTrace would create a regular backup schedule, with daily or weekly full backups of the PostgreSQL dataset. Utilizing a secure cloud or offsite storage with encryption and version control for data security. In case of disaster recovery measures, we clearly define Recovery Time Objective (RTO) and the Recovery Point Objective (RPO) for critical services, using failover tools like PgBouncer for minimal downtime.

Monitoring tools such as Prometheus would be used to track database health and performance, while automated alerts configured for failures, replication lag, or unusual activity to create logs of issues. We would then run data cleaning and data verification scripts to enforce constraints and ensure schema compliance.

10. Areas for improvement:

- Introducing indexing strategies for performance optimization
- Plan for third-party system integrations (e.g., health or case management)
- Consider NoSQL extension for unstructured support data (e.g., feedback forms)

11. Team Collaboration

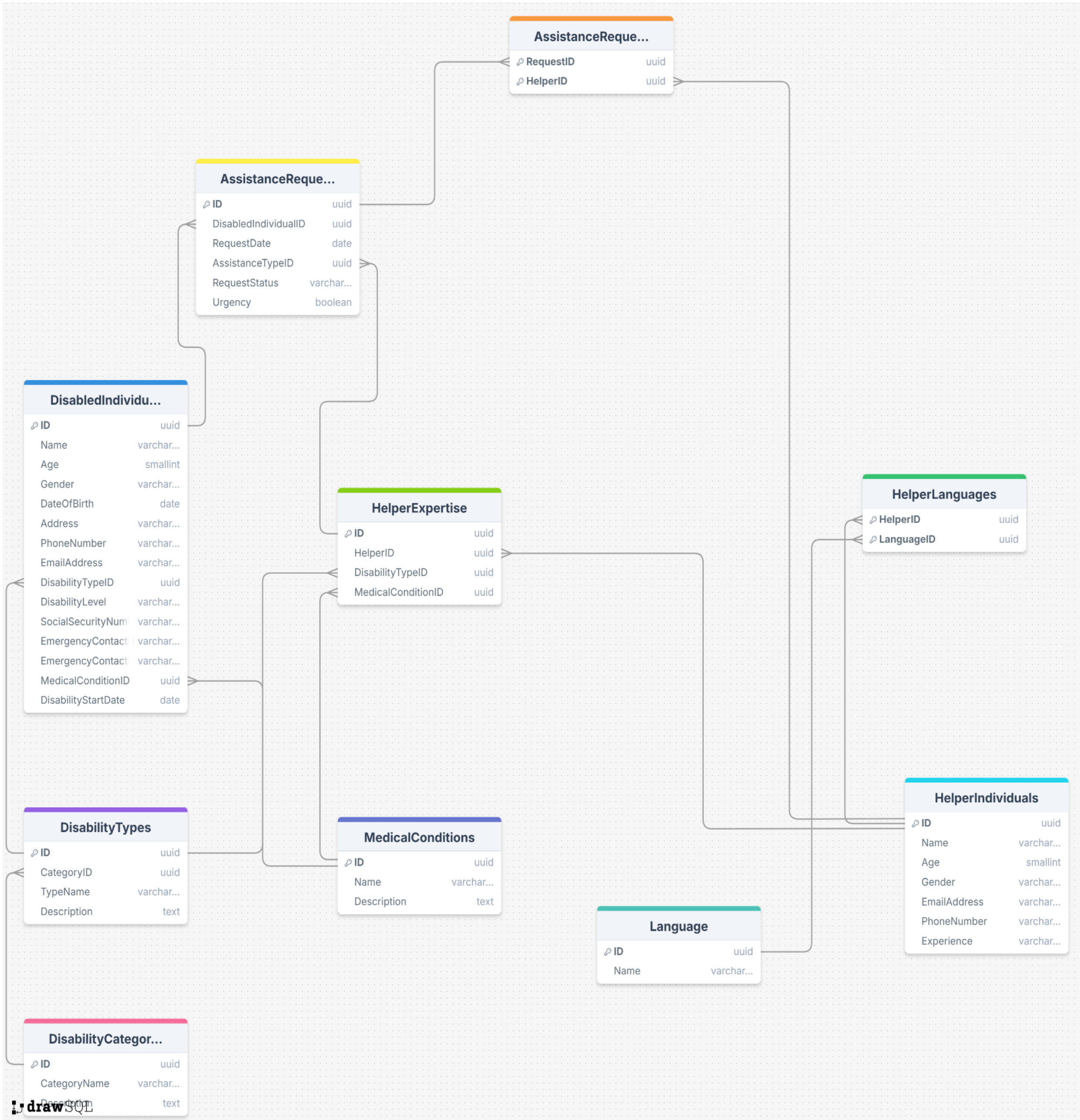
Our team collaborated virtually using Google Docs and utilizing Discord to communicate progress. Roles were divided as follows:

- **Project Lead**: Matilda Nilsson. Coordinated milestones and client vision
- **Database Designer**: Kanan Akbarov. Drafted schema and ER model
- **Researcher**: Faiz Saiyed. Investigated platform needs and user accessibility

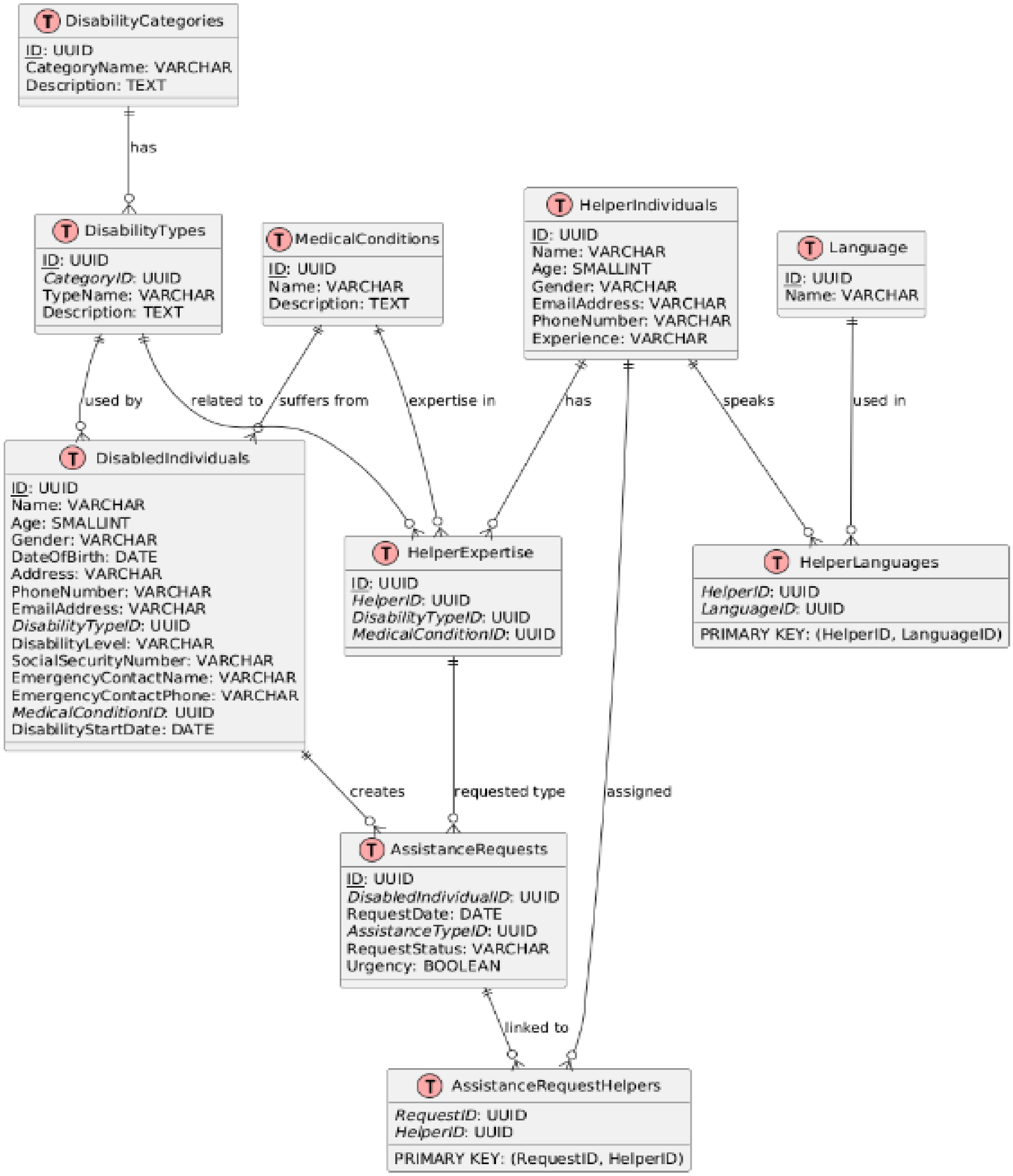
Frequent chats and shared documentation tools helped ensure alignment and version control throughout development. Together, the team wrote the documentation.

Conclusion

ZeroTrace provides a scalable, privacy-preserving platform for digital support services. Our PostgreSQL-based relational model and structured pipeline offers a secure, efficient data environment suited to the needs of individuals with disabilities. The logical design supports expansion and integration, aligning with the evolving expectations of digital social care services.



Visual 1: ER diagram



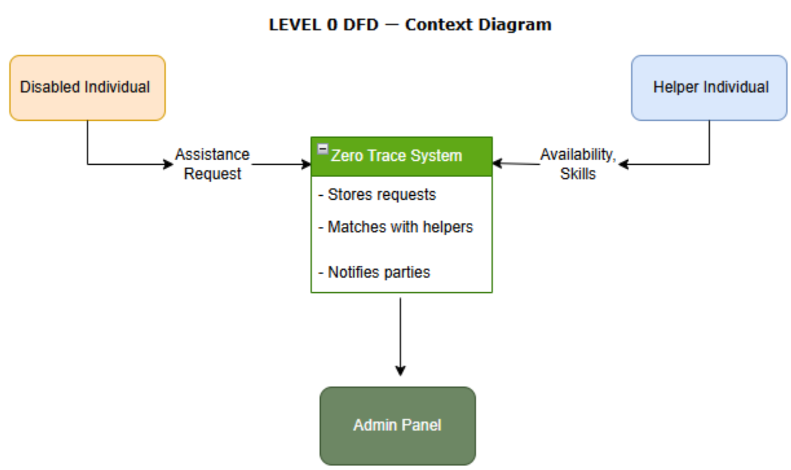
Visual 2: ER Flow

Entity	Primary Key	Foreign Keys	Relationships
DisabilityCategories	ID	—	1 → Many DisabilityTypes
DisabilityTypes	ID	CategoryID → DisabilityCategories.ID	1 → Many DisabledIndividuals1 → Many HelperExpertise
MedicalConditions	ID	—	1 → Many DisabledIndividuals1 → Many HelperExpertise
DisabledIndividuals	ID	DisabilityTypeID → DisabilityTypes.IDMedicalConditionID → MedicalConditions.ID	1 → Many AssistanceRequests
HelperIndividuals	ID	—	1 → Many HelperExpertiseM ↔ N Language (via HelperLanguages)M ↔ N AssistanceRequests (via AssistanceRequestHelpers)
HelperExpertise	ID	HelperID → HelperIndividuals.IDDisabilityTypeID → DisabilityTypes.IDMedicalConditionID → MedicalConditions.ID	1 → Many AssistanceRequests
Language	ID	—	M ↔ N HelperIndividuals (via HelperLanguages)
HelperLanguages	(HelperID, LanguageID)	HelperID → HelperIndividuals.IDLanguageID → Language.ID	Join table: HelperIndividuals ↔ Language
AssistanceRequests	ID	DisabledIndividualID → DisabledIndividuals.IDAssistanceTypeID → HelperExpertise.ID	M ↔ N HelperIndividuals (via AssistanceRequestHelpers)
AssistanceRequestHelpers	(RequestID, HelperID)	RequestID → AssistanceRequests.IDHelperID → HelperIndividuals.ID	Join table: AssistanceRequests ↔ HelperIndividuals

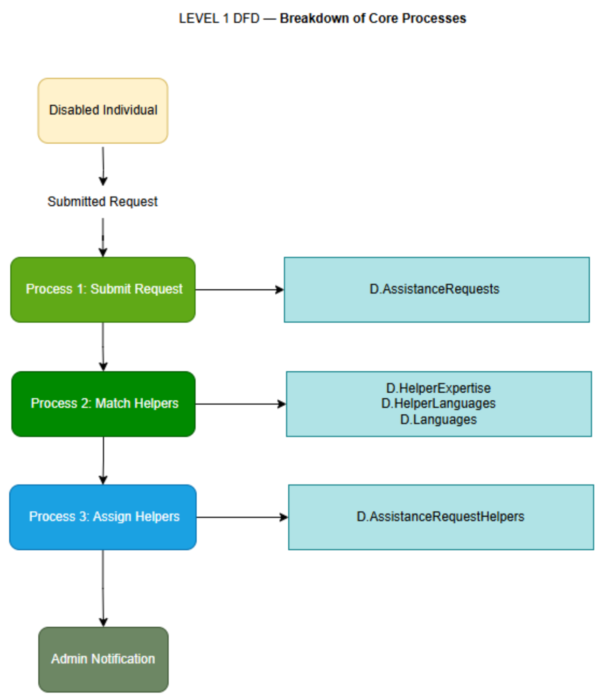
Table 1: Entity Relationship Diagram

Entity	Attributes	Description
DisabilityCategories	ID (PK), CategoryName, Description	Categories of disabilities, e.g., physical, cognitive
DisabilityTypes	ID (PK), **CategoryID** (FK), TypeName, Description	Specific types of disabilities linked to a category
MedicalConditions	ID (PK), Name, Description	Medical conditions potentially associated with disabilities
DisabledIndividuals	ID (PK), Name, Age, Gender, DateOfBirth, Address, PhoneNumber, EmailAddress, **DisabilityTypeID** (FK), DisabilityLevel, SocialSecurityNumber, EmergencyContactName, EmergencyContactPhone, **MedicalConditionID** (FK), DisabilityStartDate	Stores personal and health details of individuals with disabilities
HelperIndividuals	ID (PK), Name, Age, Gender, EmailAddress, PhoneNumber, Experience	People who provide assistance, with demographic and experience data
HelperExpertise	ID (PK), **HelperID** (FK), **DisabilityTypeID** (FK), **MedicalConditionID** (FK)	Expertise mapping for helpers based on conditions and disability types
Language	ID (PK), Name	Languages supported by helpers
HelperLanguages	HelperID, **LanguageID** → Composite PK (FKs from HelperIndividuals and Language)	Join table mapping helpers to languages
AssistanceRequests	ID (PK), **DisabledIndividualID** (FK), RequestDate, **AssistanceTypeID** (FK), RequestStatus, Urgency	Requests for help initiated by disabled individuals
AssistanceRequestHelpers	RequestID, **HelperID** → Composite PK (FKs from AssistanceRequests and HelperIndividuals)	Join table for assigning helpers to specific assistance requests

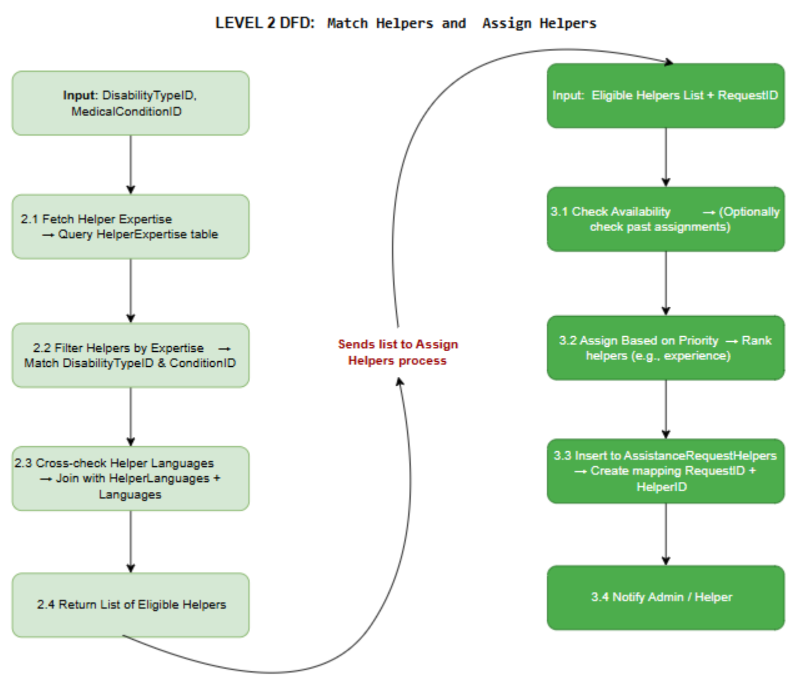
Table 2: Detailed view of database



Visual 3: Level 0 DF Diagram



Visual 4: Level 1 of DF Diagram



Visual 5: Level 2 of DF Diagram

Name	Gender	Email	Phone	DOB	Disability Level	Experience	Emergency Contact Name	Contact Phone
john DOE	male	johndoe(at)mailcom	123456789	12/5/1990	severe	EXPERT	mary	98765432
SARAH connor	Female	sarah.connor@mail.com	+91 99999X99	7/8/1992	MODERATE	intermediate	luis	567-567-567
Alex	Other		123123123	1/1/1995			Not Provided	

Table 3: Example of Dirty Data

```sql
CREATE OR REPLACE PROCEDURE clean_user_data()
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE "DisabledIndividuals"
    SET
        "Name" = INITCAP(TRIM("Name")),
        "Gender" = CASE
            WHEN LOWER("Gender") IN ('male') THEN 'Male'
            WHEN LOWER("Gender") IN ('female') THEN 'Female'
            WHEN LOWER("Gender") = 'other' THEN 'Other'
            ELSE 'Other'
        END,
        "EmailAddress" = CASE
            WHEN POSITION('@' IN "EmailAddress") > 0 THEN LOWER(TRIM("EmailAddress"))
            ELSE 'NotProvided@domain.com'
        END,
        "PhoneNumber" = CASE
            WHEN LENGTH(REGEXP_REPLACE("PhoneNumber", '\D', '', 'g')) >= 10
            THEN '+91' || RIGHT(REGEXP_REPLACE("PhoneNumber", '\D', '', 'g'), 10)
            ELSE 'Invalid'
        END,
        "EmergencyContactName" = INITCAP(TRIM("EmergencyContactName")),
        "EmergencyContactPhone" = CASE
            WHEN LENGTH(REGEXP_REPLACE("EmergencyContactPhone", '\D', '', 'g')) >= 10
            THEN '+91' || RIGHT(REGEXP_REPLACE("EmergencyContactPhone", '\D', '', 'g'), 10)
            ELSE 'Invalid'
        END,
        "DisabilityLevel" = CASE
            WHEN LOWER("DisabilityLevel") IN ('mild', 'moderate', 'severe') THEN INITCAP("DisabilityLevel")
            ELSE 'Not Provided'
        END;
    UPDATE "HelperIndividuals"
    SET
        "Name" = INITCAP(TRIM("Name")),
        "Gender" = CASE
            WHEN LOWER("Gender") IN ('male') THEN 'Male'
            WHEN LOWER("Gender") IN ('female') THEN 'Female'
            WHEN LOWER("Gender") = 'other' THEN 'Other'
            ELSE 'Other'
        END,
        "EmailAddress" = CASE
            WHEN POSITION('@' IN "EmailAddress") > 0 THEN LOWER(TRIM("EmailAddress"))
            ELSE 'NotProvided@domain.com'
        END,
        "PhoneNumber" = CASE
            WHEN LENGTH(REGEXP_REPLACE("PhoneNumber", '\D', '', 'g')) >= 10
            THEN '+91' || RIGHT(REGEXP_REPLACE("PhoneNumber", '\D', '', 'g'), 10)
            ELSE 'Invalid'
        END,
        "Experience" = CASE
            WHEN LOWER("Experience") = 'beginner' THEN 'Beginner'
            WHEN LOWER("Experience") = 'intermediate' THEN 'Intermediate'
            WHEN LOWER("Experience") = 'expert' THEN 'Expert'
            ELSE 'Not Provided'
        END;

    RAISE NOTICE 'Data cleaning completed successfully.';
END;
$$;
```

Visual 6: Data cleaning script

Name	Gender	Email	Phone	DOB	Disability Level	Experience	Emergency Contact Name	Contact Phone
John Doe	Male	johndoe@mail.com	+91123456789	5/12/1990	Severe	Expert	Mary	91098765432
Sarah Connor	Female	sarah.connor@mail.com	+919999900099	7/8/1992	Moderate	Intermediate	Luis	91567567567
Alex	Other	Not Provided	+91123123123	1/1/1995	Not Provided	Not Provided	Not Provided	Not Provided

Table 4: Cleaned data after procedure

References

Huxley, K., 2020. *Data Cleaning*. In: Paul Atkinson, ed., Sage Research Methods Foundations. London: SAGE Publications Ltd. Available at: <https://doi.org/10.4135/9781526421036842861> [Accessed 8 Jun 2025].

Woodie, A. (2019). *Data Pipeline Automation: The Next Step Forward in DataOps*. [online] BigDATAwire. Available at: https://www.bigdatawire.com/2019/04/24/data-pipeline-automation-the-next-step-forward-in-dataops/ [Accessed 8 Jun. 2025].