External Documentation – Project

Overview

The Homeless Support System is a Java-based application designed to manage and support homeless shelters, implemented with a PostgreSQL database for data storage. The system leverages Java's capabilities for object-oriented programming and JDBC for database connectivity.

Technology Stack

- **Java**: The core programming language used for the application.
- **MariaDB**: The relational database management system used to store and manage data.
- **JDBC (Java Database Connectivity)**: A Java-based API for interacting with relational databases.
- **JUnit**: Utilized for writing and running test cases.

Class Structure

The system is organized into several classes, each serving a specific purpose:

1. **HomelessSupport Class**: This serves as the central class, orchestrating the entire application. It manages services, shelters, occupancy, donations, funds disbursement, and various reports. This class integrates instances of other service-related classes.
2. **Service Class**: Responsible for defining and managing services, including their inspection frequencies.
3. **ShelterServiceManager Class**: Manages the association of services with shelters.
4. **OccupancyManager Class**: Handles the declaration of shelter occupancies.
5. **Donation Class**: Manages the receipt of donations.
6. **FundsDisbursement Class**: Manages the disbursement of funds to shelters.
7. **LocationManager Class**: Manages the storage and retrieval of location information for shelters, donors, and central offices.
8. **Staff Class**: Represents staff members associated with specific services, tracking their roles, and volunteer status.
9. **Donor Class**: Represents donors contributing to funding programs.
10. **Shelter Class**: Represents shelters with attributes such as ID, name, location, and maximum capacity.

Design Justification

The application follows a modular design, where each class focuses on a specific aspect of homeless support. Integration is achieved through the **HomelessSupport** class, serving as the orchestrator. The design decision to encapsulate related functionalities within individual classes promotes maintainability, reusability, and separation of concerns.

Database Interaction: JDBC is employed to establish a connection to the PostgreSQL database, allowing the system to persistently store and retrieve data. Prepared statements are used to execute SQL queries securely.

Optimization Algorithms: The **HomelessSupport** class includes optimization algorithms for scheduling inspections, ensuring efficient use of staff resources while meeting specified constraints.

Testing Strategy: JUnit test cases are employed to validate the functionality of key methods in the **HomelessSupport** class. Test scenarios cover a range of use cases, ensuring the reliability and correctness of the implemented features.

Files and External Data

Project Structure

The project is structured to facilitate easy navigation through its components. Here's guidance on where a reader can find the important parts of the implementation:

1. **HomelessSupport Class:**
    - Primary class managing various aspects of homeless support services.
    - Key functionalities related to services, shelters, occupancy, donations, funds disbursement, and location management are handled here.
    - Integration with manager classes ensures a modular and organized approach.
2. **DatabaseConnection Class:**
    - Responsible for establishing a connection to the MariaDB database.
    - Database credentials are sourced from the **Property.prop** file, adding a layer of security and configurability.
    - File path to the **Property.prop** file: **C:\Users\sdhru\OneDrive\Documents\SDC Project\shah14\src\Property.prop**.
3. **Main Class:**
    - Main method serves as an entry point for testing various functionalities of the **HomelessSupport** class.
    - Demonstrates usage patterns and scenarios for thorough testing.

Property File

The project utilizes a properties file, **Property.prop**, to securely store database credentials. Details on the file structure:

Property File Structure

plaintextCopy code

Property.prop username=<your_database_username> password=<your_database_password>

Database Connection

In the **DatabaseConnection** class, the direct path to the properties file is provided for secure retrieval of credentials:

Properties identity = new Properties(); String username = ""; String password = ""; String propertyFilename = "C:\\Users\\sdhru\\OneDrive\\Documents\\SDC Project\\shah14\\src\\Property.prop"; try { InputStream stream = new FileInputStream(propertyFilename); // ... (rest of the code) } catch (Exception e) { throw new RuntimeException("Database connection failed"); }

This direct path ensures the correct retrieval of credentials, adding a layer of security to the database connection process.

Navigating the Code

To find specific implementations:

- Refer to the **HomelessSupport** class for high-level functionalities.
- Explore the **DatabaseConnection** class for database connection-related operations.
- Use the **Main** class to see how different functionalities are tested and executed.

This guidance aims to assist readers in locating crucial elements within the codebase for a better understanding of the project.

Data Structures and Their Relations

HomelessSupport Class

1. **Services:**
    - **Data Structure:** List
    - **Data Type:** Service
    - **Relation:** Manages a list of available services for homeless support.
2. **Shelters:**
    - **Data Structure:** List
    - **Data Type:** Shelter
    - **Relation:** Stores information about shelters, including occupancy and location details.
3. **Occupancy:**
    - **Data Structure:** Map
    - **Data Type:** Shelter ID (String) -> Occupancy Count (Integer)
    - **Relation:** Tracks the occupancy count for each shelter.
4. **Donations:**
    - **Data Structure:** List
    - **Data Type:** Donation
    - **Relation:** Keeps a record of donations made to the homeless support services.
5. **Funds Disbursement:**
    - **Data Structure:** Priority Queue (or List)
    - **Data Type:** Disbursement Request
    - **Relation:** Manages funds disbursement requests, prioritizing urgent cases.
6. **Locations:**
    - **Data Structure:** Map
    - **Data Type:** Location ID (String) -> Location Information
    - **Relation:** Stores information about different locations relevant to homeless support services.

Database Connection

1. **Connection Information:**

- **Data Structure:** Properties
- **Data Type:** Key-Value pairs
- **Relation:** Stores database connection credentials securely in the **Property.prop** file.

2. **Database Connection Pooling:**
    - **Data Structure:** Connection Pool (implicitly managed by JDBC)
    - **Data Type:** Database Connection
    - **Relation:** Manages and optimizes a pool of database connections for efficient usage.
3. **Database Queries:**
    - **Data Structure:** SQL Queries
    - **Data Type:** String
    - **Relation:** Represents queries for creating, updating, and retrieving data from the database.
4. **Result Sets:**
    - **Data Structure:** Result Set (implicitly managed by JDBC)
    - **Data Type:** Result Set
    - **Relation:** Stores the results of executed SQL queries for further processing.

Overall Database

1. **MariaDB Database:**
    - **Data Structure:** Tables (e.g., Services, Shelters, Donations)
    - **Data Type:** Rows and Columns
    - **Relation:** Persists and organizes data for homeless support services.
2. **Database Relationships:**
    - **Data Structure:** Foreign Keys
    - **Data Type:** References to Primary Keys
    - **Relation:** Establishes connections between different tables in the database.

These data structures collectively form a robust framework for managing and organizing data within the Homeless Support application, ensuring efficient operations and interactions with the underlying MariaDB database.

Assumptions

1. **Unique Names:**
    - **Assumption:** It is assumed that all names used within the application, such as service names, shelter names, and location names, are unique. This assumption simplifies the process of identifying and managing different entities within the system.
2. **Auto-Incremented IDs:**
    - **Assumption:** Most ID fields, such as primary keys in database tables, are assumed to be auto-incremented. This means that when a new record is added to a table, the system automatically assigns a unique identifier to that record. This simplifies the process of managing and referencing records within the database.
3. **Service Availability:**
    - **Assumption:** It is assumed that services listed in the application are available by default unless explicitly marked as unavailable. This simplifies the user experience by assuming that all services are accessible unless specified otherwise.
4. **Location Identification:**
    - **Assumption:** Locations are identified uniquely by their IDs. Each location ID is assumed to be unique and serves as a reliable identifier for different locations. This assumption facilitates the unambiguous referencing of locations within the application.
5. **Data Integrity:**
    - **Assumption:** The application assumes that the data integrity constraints in the database, such as foreign key relationships, are maintained. It is expected that references between tables are valid, ensuring the consistency and reliability of the stored data.
6. **Standard Database Connectivity:**
    - **Assumption:** The application assumes standard database connectivity and that the necessary drivers and configurations are correctly set up. Any deviations from standard configurations may require adjustments to the database connection settings.

These assumptions contribute to the simplicity and efficiency of the implementation while aligning with common practices in database design and application development. It is essential to validate these assumptions based on specific use cases and requirements.

Choices

1. **Database Management System:**
    - **Choice:** The application uses MariaDB as the database management system.
    - **Impact of Incorrect Choice:** Choosing a different database management system without ensuring compatibility may lead to connectivity issues, SQL syntax disparities, and overall system instability.
2. **Direct Path to Property File:**
    - **Choice:** Database connection credentials (username and password) are stored in a properties file, and the file path is directly specified in the code.
    - **Impact of Incorrect Choice:** Storing credentials directly in the code or specifying an incorrect file path can pose security risks. It is essential to secure sensitive information and ensure the correct path for accessing credentials.
3. **Assumption of Unique Names:**
    - **Choice:** The design assumes that all names used in the application are unique.
    - **Impact of Incorrect Choice:** If names are not unique as assumed, it could lead to confusion, errors in data retrieval, and challenges in identifying and managing different entities.
4. **Auto-Incremented IDs:**
    - **Choice:** Most ID fields, such as primary keys, are assumed to be auto-incremented.
    - **Impact of Incorrect Choice:** If IDs are not auto-incremented as assumed, manual management of unique identifiers becomes necessary, potentially leading to ID conflicts and data inconsistency.
5. **Standardized Database Connectivity:**
    - **Choice:** Standard JDBC drivers and configurations are used for connecting to MariaDB.
    - **Impact of Incorrect Choice:** Choosing incompatible drivers or configurations may result in connection failures and disrupt the overall database interaction.
6. **Service Availability Handling:**
    - **Choice:** Services are assumed to be available by default unless marked as unavailable.
    - **Impact of Incorrect Choice:** If the assumption is not valid, and services are not available by default, it may lead to unexpected behavior and user confusion.

These key choices significantly influence the behavior and performance of the application. It is crucial to adhere to these design decisions for the proper functioning of the system and to mitigate potential issues arising from incompatible choices.

Key Algorithms and Design Elements

1. **Database Connection Algorithm:**
    - **Location: DatabaseConnection** class
    - **Description:** Establishes a MariaDB database connection using JDBC, involving loading credentials from a properties file, initializing the JDBC driver, and creating a connection object.
2. **Homeless Support Services Retrieval Algorithm:**
    - **Location: HomelessSupport** class
    - **Description:** Retrieves support services from the database based on specified criteria, executing SQL queries, processing result sets, and populating a list of support services.
3. **User Input Validation Algorithm:**
    - **Location:** Various classes handling user input
    - **Description:** Validates user inputs (e.g., names, addresses) for length, format, and allowable characters, ensuring data integrity.
4. **Service Availability Check Algorithm:**
    - **Location: HomelessSupport** class
    - **Description:** Checks the availability of support services; if a service is marked as unavailable, it is excluded from the results.

Limitations

1. **Single-User Interaction:**
    - The current implementation assumes a single-user scenario. Simultaneous interactions by multiple users might result in data integrity issues.
2. **Minimal Security Measures:**
    - Limited security features are implemented. For instance, user authentication and authorization mechanisms are not robustly established. Future versions should include enhanced security measures.
3. **Static Database Connection Credentials:**
    - Database connection credentials are stored in a properties file with a static path. This poses a security risk, and future versions should implement a more secure and dynamic credential management system.
4. **Limited Error Handling:**
    - While basic error handling is in place, it may not cover all possible scenarios. Future iterations should include comprehensive error handling mechanisms to provide informative feedback to users.
5. **Non-Optimized Database Queries:**
    - Database queries, while functional, may not be fully optimized. Future enhancements should focus on refining queries for improved performance, especially as the database grows.
6. **Assumed Unique Names:**
    - The system assumes that all names are unique. In reality, this assumption might lead to conflicts in scenarios where non-unique names are valid.
7. **Auto-Incremented ID Dependency:**
    - The assumption that most ID fields are auto-incremented might limit compatibility with databases where such a feature is not prevalent.

These limitations highlight areas where the current implementation may fall short in terms of scalability, security, and flexibility. Addressing these in future versions would enhance the overall robustness and usability of the system.