.NET Mentoring Program Intermediate+ Practice_Q2_2024

PERSISTENCE LEVEL (SQL, NOSQL) + DAL

Legal Notice: This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM®.

What are your steps to start designing database

Define the Purpose and Scope:

- Identify the purpose and functionalities of your database. What information will it store and manage?
- Determine the scope of the data you need to store. This helps define the tables and their relationships.

Identify Entities and Attributes:

- Break down the information into real-world entities (e.g., customers, products, orders).
- Define the attributes (data points) associated with each entity (e.g., customer name, product ID, order date).

Establish Relationships Between Entities:

- Identify how entities relate to each other. Common relationships include one-to-one, one-to-many, and many-to-many.
- Use these relationships to determine how tables will be linked (e.g., foreign keys).

Normalize the Database:

- Normalization is a process of organizing your tables to minimize data redundancy and improve data integrity.
- There are different normalization levels (1NF, 2NF, 3NF, etc.) that progressively reduce redundancy.

Choose a Data Model:

- Select a data model that suits your needs. Common models include relational (uses tables with rows and columns), hierarchical (data organized in a tree-like structure), and NoSQL (flexible schema for non-relational data).

Create a Logical Data Model:

- This is a high-level representation of the database schema using diagrams (Entity-Relationship Diagrams - ERDs) or notations.
- It visually depicts entities, attributes, and relationships.

Physical Database Design:

- Translate the logical model into the specific syntax of your chosen database management system (DBMS) like MySQL, SQL Server, or Oracle.
- This involves defining table structures, data types, constraints (primary keys, foreign keys, etc.).

Implement and Test the Database:

- Create the database schema in your chosen DBMS.
- Populate the database with sample data.
- Test the database functionality by querying and manipulating data.

When can we say that our database is modeled correctly?

A well-designed database meets several criteria that indicate it's modeled correctly. Here are some key aspects:

Data Integrity:

- Minimized Redundancy: Data should be stored in a single place to avoid inconsistencies and the need for manual updates across multiple locations.
- Data Accuracy: The database should enforce data validation rules (e.g., data types, constraints) to ensure accurate information.
- Referential Integrity: Relationships between tables should be maintained using foreign keys to prevent orphaned or dangling data.

Efficiency and Performance:

- Appropriate Normalization: The database schema should be normalized to a level that balances data integrity with efficient querying and retrieval.
- Indexing: Indexing relevant columns can significantly improve query performance for frequently used search criteria.
- Optimized Queries: Queries should be written efficiently to minimize resource usage and avoid unnecessary data processing.

Legal Notice: This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM®.

Maintainability and Scalability:

- Clear and Consistent Naming Conventions: Using consistent naming for entities, attributes, and tables improves readability and understanding of the schema.
- Documentation: Documenting the database schema, relationships, and constraints is crucial for future maintenance and modifications.

Flexibility:

- The design should allow for future growth by accommodating additional data or changes in data requirements without major restructuring.

Additional Considerations:

- Meets User Needs: The database effectively supports the functionalities and information retrieval needs of the application or system it serves.
- Security: Security measures are in place to protect sensitive data from unauthorized access or modification.
- Backup and Recovery: A proper backup and recovery strategy exists to ensure data availability in case of failures or accidents.

What is a Data Access Layer (DAL), and how does it simplify database interactions?

A Data Access Layer (DAL) is a component in software architecture that acts as a mediator between your application logic and the underlying database. It simplifies database interactions by providing a layer of abstraction and offering several advantages:

Abstraction:

- The DAL hides the complexities of specific database technologies (like SQL syntax or database APIs) from your application code.
- Your application logic interacts with the DAL using a consistent interface, regardless of the actual database system being used. This allows for easier switching between databases if needed.

Encapsulation:

- The DAL encapsulates all the database access logic within a dedicated layer. This improves code organization, separation of concerns, and maintainability.
- Developers working on the application logic don't need to worry about the specifics of database interactions, focusing on business functionalities.

Improved Reusability:

- The DAL code can be reused across different parts of your application that need to interact with the database. This reduces code duplication and promotes consistency in data access patterns.

Centralized Management:

- The DAL becomes a central point for managing database connections, pooling, and security configurations. This simplifies database access management and reduces the risk of errors or inconsistencies scattered throughout the application code.

Additional Benefits:

- The DAL can potentially improve performance by caching frequently accessed data or implementing connection pooling for efficient database resource utilization.
- It can integrate with security mechanisms to control access to sensitive data and enforce data integrity rules.

Overall, using a DAL promotes:

- Cleaner and more maintainable application code.
- Easier database switching or technology upgrades.
- Improved development efficiency and reusability.

Here's an analogy: Imagine a library. The library (DAL) provides a central location (interface) to access books (data). You don't need to know how the library is organized or where each book is physically stored (database specifics). You just interact with the librarians (DAL methods) to find and retrieve the books (data) you need.

Legal Notice: This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM®.

You need to implement a new service for a customer. How would you select database (SQL or NoSQL)

Choosing between a relational database (SQL) and a NoSQL database depends on the specific characteristics of your new service and the data it will handle. Here's a breakdown of key factors to consider when making this decision:

Data Structure:

- Structured Data: If your data has a well-defined schema with fixed relationships between entities (e.g., user accounts, product catalogs), an SQL database is a good choice. Its structured tables and querying capabilities (like JOINs) are well-suited for relational data.
- Unstructured or Semi-structured Data: If your data is less rigid or has a more fluid schema that might evolve over time (e.g., sensor readings, social media posts, user preferences), a NoSQL database might be more flexible. These databases can handle data with varying structures without strict schema definitions.

Scalability:

- Horizontal Scalability: If you anticipate high data volume or a large number of concurrent users, NoSQL databases often excel in horizontal scalability. You can easily add more nodes (servers) to handle increased load by distributing data across them.
- Vertical Scalability: If your primary concern is scaling compute resources (CPU, memory) to handle complex queries on a single server, SQL databases might be sufficient. You can upgrade your server hardware to accommodate heavier workloads.

Performance:

- Complex Queries: For applications requiring frequent complex queries involving joins across multiple tables, SQL databases can be efficient due to their optimized query engines.
- Simple Queries and High Availability: If your service involves mostly simple queries (e.g., inserts, updates, deletes) and prioritizes high availability (uptime), NoSQL databases can offer fast performance and resilience to server failures.

Other Considerations:

- Development Experience: If your team has more experience with SQL and familiar with querying languages like SQL, an SQL database might be a quicker development choice.
- Real-time Needs: If your service requires real-time data updates or analytics, some NoSQL databases offer features specifically designed for these scenarios.

Factor	More Suitable for SQL	More Suitable for NoSQL
Data Structure	Structured, Fixed Schema	Unstructured, Semi-structured, Evolving Schema
Scalability	Vertical Scaling	Horizontal Scaling
Performance	Complex Queries	Simple Queries, High Availability
Development Experience	Familiarity with SQL	Familiarity with NoSQL Models
Real-time Needs	Not a primary focus	Can be a strength for certain NoSQL options

Legal Notice: This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM®.