# **Top 10 Manager Round Interview Questions**

1. Tell Me Something about your self

Thank you swetha,

Sure

I am rajkumar jadi having 4.6 years of experience in implementing web apps , Generally we build the business applications for our internal and external clients to do this we use Microsoft technologies such as c#,.net core, asp.net core web API , asp.net core entity framework ,dapper and RDBMS we are using SQL server 2014 and coming to ui technologies we use html5, css3 , bootstrap5,java script and angular 16

Thank you.

1. Tell me About your current project?

Currently I am working in a transport management system projetc application and which is a school related travel domain project the purpose of the application to maintain transport system and tracking the system for generating reports based on school bus live tracking information and fees information

**Key Responsibilities:**

* **Bus Management Module:** Led the design and development of the Bus Management module, overseeing fleet management, driver assignments, real-time route tracking, and vehicle maintenance monitoring.
* **Driver Management Module:** Managed the Driver Management module, ensuring driver compliance with safety regulations, conducting regular safety assessments, and tracking performance metrics.
* **Passenger Management Module:** Spearheaded the integration of online booking, reservation handling, and feedback systems in the Passenger Management module to improve customer satisfaction and streamline passenger operations.
* **Request Management Module:** Directed the development of the Request Management module, optimizing transportation scheduling, enhancing resource allocation, and improving service delivery for transportation requests.
* **Vendor Management:** Led the Vendor Management process, establishing strong relationships with external vendors for vehicle procurement, maintenance services, and other critical resources, ensuring cost-effective and timely deliveries.
* **Transport Executive Management:** Developed and managed the Transport Executive Management module, overseeing executives responsible for daily operations, addressing issues, and ensuring compliance with organizational policies and standards.

The technology we are using angular for fronted dot net core for backend and sql server for the database

And we are following a hybrid agile methodology for our project

3.what are your roles and responsibilities in your project ?

As a dot net developer my role is to

· Attend daily status calls with the project manager to discuss progress and upcoming tasks.

* Receive user stories and Figma links via Monday.com from the design team.
* Participate in knowledge transfer (KT) sessions for each user story with the design team.
* Analyze screen designs and design the database structure accordingly.
* Write code based on the designed database structure and debug as needed.
* Generate SQL Server dynamic queries as required by the application.
* Monitor and respond to incidents and requests promptly.
* Troubleshoot issues reported and provide resolutions or workarounds.
* Attend project meetings to discuss progress, blockers, and next steps.
* Prepare documentation for each API using Postman, including detailed explanations and examples.
* Share daily status updates with the Team Lead to keep them informed of project status and any issues that need attention.
* Follow Agile practices throughout the development lifecycle.
* Engage in sprint planning, daily stand-ups, sprint reviews, and retrospectives as part of Agile ceremonies.
* Adapt to changes and prioritize tasks based on Agile principles and feedback received.

4.Explain the technical design of your project ?

My project is a

· **ASP.NET Core Web API** with a **repository pattern** to ensure scalability, flexibility, and independent service management.

· Utilized **Dependency Injection** for service management and **Entity Framework Core** along with **Dapper** for efficient database interactions.

· Implemented **JWT tokens** for robust authentication and authorization mechanisms.

· Integrated **AWS App Insights** for logging and monitoring application performance.

· Employed **XUnit** for comprehensive unit testing, ensuring code quality and reliability.

· Followed **SOLID principles** to achieve a loosely coupled, maintainable, and testable system design.

Study Guide: Repository Pattern in C# .NET

The Repository Pattern is a design pattern used to separate the data access logic from the business logic in an application. It is commonly used in applications where there is a need to manage data from different sources and want to abstract away the complexity of querying and storing data. By using this pattern, your application becomes more maintainable, testable, and flexible.

Benefits of the Repository Pattern

1. Separation of Concerns: Business logic does not depend on the data access code, making the code cleaner.

2. Testability: Easier to write unit tests as dependencies on data access can be mocked.

3. Decoupling: Your application is decoupled from the database, meaning you can change the data source with minimal code changes.

4. Centralized Data Logic: The data access logic is centralized, making it easier to change or add new ways to retrieve or modify data.

5. Scalability: You can easily scale the repository to manage more complex querying, caching, or other data concerns.

# Key Components

1. Domain Model: Represents the business entities.

2. Repository Interface: Declares the methods for data access, e.g., GetAll(), Add(), Update(), Delete().

3. Repository Implementation: Implements the interface and contains the actual code to interact with the data source.

4. Service Layer: Utilizes the repository to retrieve and save data, without needing to understand how the repository interacts with the data source.

**how to identify the bugs in production environments and how to fix it by using asp.net core web api and angular**

In my experience working with ASP.NET Core Web API and Angular, identifying and fixing bugs in production environments is a structured process. First, I rely heavily on **monitoring and logging**. On the backend, I use tools like **AWS App Insights** to track exceptions and performance issues. On the frontend, I use **Angular's ErrorHandler** to catch client-side errors and integrate it with monitoring tools like **Sentry**.

Once an issue is detected, I look into structured logs, often set up with **Serilog** or similar logging libraries, to trace the root cause. If necessary, I use **remote debugging** tools to live-debug production issues. After pinpointing the bug, I apply the necessary fixes and ensure that we write **unit tests** and **e2e tests** to prevent future occurrences.

For deployments, I typically use a **Blue-Green deployment** strategy to minimize user impact and carefully monitor the system post-deployment to confirm the issue is resolved.

This approach ensures quick bug resolution with minimal impact on the production environment.

5.Suppose it is the delivery time and therefore , you have to work on weekends ?

Yes if needed I will be there with the project at any point

1. extra work load ,learn new technology , project with tight deadline

I will be there with the project in difficult situations

### TELL ME ABOUT CHALLENGING PROJECT YOU WORKED ON

### ****Situation****: "In a recent project, I worked on building a ****live bus tracking system**** that would track the location of buses in real-time and generate reports based on the historical data of bus routes, schedules, and delays. The challenge was to handle continuous location updates from multiple buses while ensuring the reports were accurate and timely."

### ****Task****: "My primary responsibility was to design the backend system, ensuring real-time updates were efficiently handled and stored. I also had to ensure that users, like bus operators and passengers, could see the live positions of buses and that reports could be generated to analyze bus performance and schedule adherence."

**Action**:  
"To achieve real-time tracking, I used **WebSocket** in **ASP.NET Core** to create persistent connections between the server and the clients. This allowed the server to push live location updates to the clients (like a dashboard or mobile app) as soon as the bus sent a new location.

For report generation, I designed a mechanism that stored all location data into **SQL Server** in a structured format, which allowed for efficient querying and report generation. I used **Entity Framework** to interact with the database and created queries to generate reports on metrics like bus delays, total travel time, and route deviations.

To handle scalability and avoid performance bottlenecks, I implemented **message queues** using kafka, ensuring that location data processing and report generation were decoupled. This allowed the system to process real-time data asynchronously without affecting report generation."

**Result**:  
"As a result, the system was able to provide real-time location updates with minimal latency, even when tracking hundreds of buses simultaneously. The generated reports provided insights into bus performance, leading to a 15% improvement in schedule adherence. The system's architecture also ensured it could scale easily as more buses and routes were added."

WHAT ROLES DOES GIT AND AWS PLAY IN YOUR DEVELOPMENT PROCESS AND HOW DO YOU USE IT EFFECTIVELY

In my development process, **Git** and **AWS** play crucial roles in version control, collaboration, and cloud infrastructure management:

### ****Git****

1. **Version Control**: Git allows me to track changes in the codebase, making it easy to revert to previous versions when needed or compare code across different points in time.
2. **Branching and Merging**: I create feature branches for new tasks and merge them into the main branch after testing. This ensures that experimental or feature-specific code doesn't disrupt the stable production code.
3. **Collaboration**: Git facilitates team collaboration by enabling multiple developers to work on different parts of the project simultaneously without overwriting each other's work. Pull requests and code reviews improve code quality.
4. **CI/CD Integration**: Git repositories can be integrated with Continuous Integration (CI) tools like Jenkins or GitHub Actions to automate testing and deployment processes.
5. **Effective Use**:
   1. Frequent commits with meaningful messages help maintain a clean project history.
   2. U”sing .gitignore to avoid tracking unnecessary files ensures that the repository remains clean.
   3. Regular pull requests and reviews to maintain code quality.

### ****AWS (Amazon Web Services)****

1. **Cloud Infrastructure**: AWS provides scalable and reliable cloud infrastructure. I use services like EC2 for hosting, RDS for databases, and S3 for object storage.
2. **Deployment and Scaling**: By using services like Elastic Beanstalk or ECS (Elastic Container Service), I can easily deploy, manage, and scale applications based on demand.
3. **Security**: AWS Identity and Access Management (IAM) helps manage permissions and access to resources, ensuring that applications and developers only have the permissions they need.
4. **Monitoring and Logging**: AWS CloudWatch and CloudTrail are invaluable for monitoring the performance of applications and tracking changes made in the AWS environment.
5. **Effective Use**:
   1. Utilizing **auto-scaling** for cost-effective resource management.
   2. Following **best security practices** by limiting public access, using IAM roles, and regularly rotating keys and credentials.
   3. Automating deployments using AWS services like **CodeDeploy** and **CodePipeline**.

Git manages the source code lifecycle, while AWS handles the hosting, scaling, and management of the production environment, making both indispensable for modern development workflows.

**How to optimize the query perfomance**

Optimizing query performance is essential for building efficient and scalable applications. Several techniques can be applied depending on the database design, query structure, and the specific use case. Below are some key strategies with examples for optimizing query performance:

### 1. ****Indexing****

Indexes allow the database to quickly locate the data without scanning the entire table.

#### Example:

sql

Copy code

-- Create an index on a column frequently used in WHERE clausesCREATE INDEX idx\_employee\_name ON Employees(Name);

-- Query using the indexed columnSELECT \* FROM Employees WHERE Name = 'Rajkumar Jadi';

**Explanation:** Without an index, the database would have to scan every row to find matching entries. By indexing the Name column, the database can quickly find the relevant rows.

### 2. ****Avoiding SELECT \* (Selecting Specific Columns)****

Selecting only the necessary columns reduces the amount of data being transferred, improving performance.

#### Example:

sql

Copy code

-- Inefficient query SELECT \* FROM Employees;

-- Optimized query (select only necessary columns)SELECT Name, Department FROM Employees;

**Explanation:** Selecting only the required columns reduces the load on the database and network by fetching only what’s needed.

### 3. ****Use JOINs Instead of Subqueries****

In many cases, JOIN operations perform better than subqueries because databases are optimized for them.

#### Example:

sql

Copy code

-- Subquery (less efficient)SELECT Name FROM EmployeesWHERE DepartmentId IN (SELECT Id FROM Departments WHERE DepartmentName = 'IT');

-- Optimized query using JOINSELECT e.Name FROM Employees eJOIN Departments d ON e.DepartmentId = d.IdWHERE d.DepartmentName = 'IT';

**Explanation:** JOIN tends to be more efficient than subqueries as the database can handle the join operation more effectively, especially with proper indexing.

### 4. ****Use WHERE Instead of HAVING (Where Possible)****

The HAVING clause is used to filter results after aggregation, but it is less efficient than using WHERE when filtering rows before aggregation.

#### Example:

sql

Copy code

-- Less efficient querySELECT Department, COUNT(\*) FROM EmployeesGROUP BY DepartmentHAVING Department = 'IT';

-- Optimized query using WHERE SELECT Department, COUNT(\*) FROM EmployeesWHERE Department = 'IT'GROUP BY Department;

**Explanation:** WHERE filters rows before aggregation, while HAVING filters after, so using WHERE can reduce the data that needs to be processed.

### 5. ****Limit the Number of Rows Returned****

Using LIMIT or TOP in queries can prevent fetching unnecessary data, improving performance.

#### Example:

sql

Copy code

-- Fetch all rows (inefficient if you only need a few)SELECT \* FROM Employees ORDER BY JoiningDate;

-- Optimized query with LIMITSELECT \* FROM Employees ORDER BY JoiningDate LIMIT 10;

**Explanation:** Limiting the number of rows returned reduces both the load on the server and the amount of data sent to the client.

### 6. ****Optimize with Proper Use of Indexes in Composite Keys****

When using composite indexes, ensure that the query filters on the leading index column(s) to take advantage of the index.

#### Example:

sql

Copy code

-- Create a composite index on (DepartmentId, JoiningDate)CREATE INDEX idx\_dept\_join\_date ON Employees(DepartmentId, JoiningDate);

-- Query that can use the composite indexSELECT \* FROM Employees WHERE DepartmentId = 1 AND JoiningDate > '2023-01-01';

**Explanation:** The query benefits from the composite index because it uses both columns in the condition, with DepartmentId being the leading column.

### 7. ****Use EXISTS Instead of IN for Large Datasets****

EXISTS often performs better than IN when dealing with subqueries that return large datasets.

#### Example:

sql

Copy code

-- Less efficient query using INSELECT Name FROM Employees WHERE DepartmentId IN (SELECT Id FROM Departments WHERE Location = 'New York');

-- Optimized query using EXISTSSELECT Name FROM Employees eWHERE EXISTS (SELECT 1 FROM Departments d WHERE e.DepartmentId = d.Id AND d.Location = 'New York');

**Explanation:** EXISTS stops searching once a match is found, making it more efficient for large datasets compared to IN, which processes all results.

### 8. ****Use Pagination for Large Result Sets****

For large datasets, it is better to paginate the results rather than fetching everything at once.

#### Example:

sql

Copy code

-- Inefficient query (fetches all rows)SELECT \* FROM Employees ORDER BY Name;

-- Optimized query with paginationSELECT \* FROM Employees ORDER BY Name LIMIT 10 OFFSET 0;

**Explanation:** Pagination allows the application to fetch data in smaller, more manageable chunks, reducing the load on the database and improving user experience.

### 9. ****Optimize Query Plans****

Analyze query execution plans to identify bottlenecks such as full table scans or inefficient joins. This can be done using EXPLAIN in most databases.

#### Example:

sql

Copy code

-- Using EXPLAIN to analyze the query

EXPLAIN SELECT \* FROM Employees WHERE Name = 'Rajkumar Jadi';

**Explanation:** EXPLAIN provides insights into how a query is executed, showing whether an index is used or if any parts of the query can be optimized.

### 10. ****Partitioning Large Tables****

Partitioning large tables can improve performance by dividing them into smaller, more manageable pieces.

#### Example:

sql

Copy code

-- Partition the Employees table by JoiningYearCREATE TABLE Employees (

Id INT,

Name VARCHAR(100),

DepartmentId INT,

JoiningYear INT

) PARTITION BY RANGE (JoiningYear);

**Explanation:** Partitioning allows the database to scan only the relevant partition of a table, improving performance when dealing with large datasets.

##### ****How do you implement continuous integration and continuous deployment (CI/CD) pipelines for ASP.NET Core?****

Utilize tools like Azure DevOps, Jenkins, or GitHub Actions to automate the build, test, and deployment process of ASP.NET Core applications.

Set up pipelines to include steps for code compilation, running tests, and deploying to various environments (development, staging, production) based on triggers like code commits or manual approvals.

##### ****What challenges do you face when developing distributed systems with ASP.NET Core?****

* ****Complexity:**** Increased complexity in managing multiple services, inter-service communication, and data consistency.
* ****Deployment:**** Coordinating deployment across multiple services.
* ****Monitoring and Logging:**** Centralizing logs and monitoring from disparate services.
* ****Latency:**** Increased latency due to network calls between services.

##### ****How do you optimize performance in ASP.NET Core applications?****

* ****Response Caching:**** Use response caching to reduce the load on the server and speed up responses.
* ****Asynchronous Programming:**** Leverage async/await to improve scalability and responsiveness.
* ****Minimize Resource Usage:**** Optimize database queries, minimize the use of blocking calls, and use efficient algorithms.

##### ****What are the different types of tests you can write for ASP.NET Core applications?****

In ASP.NET Core applications, we can write unit tests, integration tests, and functional tests. Unit tests focus on testing individual components or methods for correctness. Integration tests verify the interaction between components or systems, such as database access and API calls. Functional tests, or end-to-end tests, validate the application as a whole, ensuring that the user experience is as expected.

##### ****How do you unit test controllers and services?****

To unit test controllers and services, I use a testing framework like xUnit or NUnit, along with a mocking library like Moq. For controllers, I mock the services they depend on to isolate the controller logic. For services, I mock external dependencies like database contexts or external APIs. This approach allows me to test the behavior of my code in isolation from its dependencies.

EXPLANATION OF PROJECT ARCHETECTURE

### 1. ****Overall Architecture Approach****

Your project follows a **microservices architecture** with an emphasis on **modularity, scalability, and maintainability**. The architecture consists of distinct services that interact with each other via well-defined APIs. Each service is independently managed, and the **repository pattern** ensures separation of concerns between data access and business logic.

### 2. ****Backend - ASP.NET Core Web API****

At the core of the application is an **ASP.NET Core Web API**, which serves as the backend and exposes RESTful services. This architecture provides:

* **Scalability**: Each service can be independently scaled based on demand.
* **Flexibility**: Services can be modified without affecting the whole system.
* **Separation of Concerns**: The use of **Repository Pattern** keeps the data access layer separate from the business logic, ensuring cleaner code and easier testing.

### 3. ****Service Management - Dependency Injection****

The architecture leverages **Dependency Injection (DI)**, which makes the system more modular and easier to manage. With DI:

* Services and dependencies are injected at runtime.
* It promotes loose coupling by ensuring components are not tightly bound to one another.
* This allows for easy testing and better maintainability, especially when scaling the system with more services.

### 4. ****Database Layer - Entity Framework Core and Dapper****

For the database interactions, the project employs:

* **Entity Framework Core** for ORM, which simplifies database management and enables LINQ-based queries.
* **Dapper** is used where performance and direct control over SQL queries are needed.
* The **Repository Pattern** further abstracts these technologies, making it easy to swap or enhance database interactions without affecting the business logic.

### 5. ****Authentication & Authorization****

The application implements **JWT (JSON Web Token) tokens** for secure and scalable authentication and authorization:

* **JWT tokens** ensure that each request is stateless and securely validated.
* This allows user identities to be confirmed and roles assigned without maintaining a persistent session on the server.

### 6. ****Logging & Monitoring - AWS App Insights****

To ensure that the system is reliable and trackable, **AWS App Insights** is integrated for:

* **Performance monitoring**: It tracks the application’s performance, request patterns, and potential bottlenecks.
* **Logging**: Provides logs that help in debugging, monitoring user activities, and identifying issues in production.

### 7. ****Unit Testing - XUnit****

For ensuring the quality and reliability of the system, the architecture employs **XUnit** for unit testing:

* Each service, repository, and controller has corresponding test cases.
* This ensures the system is always tested against regressions when new features or updates are introduced.
* Test cases also help ensure the business logic is implemented as expected.

### 8. ****SOLID Principles****

The entire architecture follows **SOLID design principles** to achieve:

* **Single Responsibility**: Each service/module is responsible for a single task, making it easy to maintain and test.
* **Open/Closed Principle**: Modules are open for extension but closed for modification, ensuring future-proofing of the design.
* **Liskov Substitution**: Services or components can be replaced with others without breaking functionality.
* **Interface Segregation**: Small, client-specific interfaces are preferred over large ones, ensuring better separation of responsibilities.
* **Dependency Inversion**: Higher-level modules do not depend on lower-level modules but rather on abstractions, ensuring loose coupling.

### 9. ****Communication Between Components****

* **API Layer**: The Web API serves as the gateway for all external communication. Clients or other services interact with the system using HTTP requests.
* **Service Layer**: Each module within the backend (e.g., Bus Management, Passenger Management) communicates through well-defined APIs, ensuring modularity.
* **Data Access Layer**: The repository pattern isolates the business logic from the underlying database technologies.

### 10. ****Deployment & DevOps****

* The architecture is designed to be deployed on cloud platforms (such as **Azure or AWS**), leveraging **App Insights** and cloud-native services for monitoring, scaling, and deploying updates seamlessly.

### Diagram (optional)

If possible, illustrate the architecture with a **diagram** showing:

* Frontend → API Gateway → Microservices (Bus Management, Driver Management, etc.) → Repository Layer → Database
* Include logging (App Insights), testing (XUnit), and authentication (JWT) in the diagram for better understanding.