SECURE YOUR ASP.NET CORE API

100+ .Net & SQL & Angular Security

Interview Questions and Answers

Sai Reddy 5/18/25 Interview Tips

100+ Security Related Interview Q&A

Rest API .Net Core 0-2 Exp

1. What is authentication and authorization in .NET Core?

Authentication is the process of verifying the identity of a user or client, while authorization determines what resources or actions the authenticated user is allowed to access. In .NET Core, authentication confirms who the user is, and authorization enforces access control based on roles or policies.

Example:

```
// Authentication config in Startup.cs
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => { /* JWT options */ });

// Authorization in controller
[Authorize(Roles = "Admin")]
public IActionResult GetAdminData() { ... }
```

2. How do you secure a .NET Core Web API?

We secure a .NET Core Web API by implementing authentication and authorization mechanisms, enabling HTTPS, validating input to prevent injection attacks, configuring CORS properly, and using security headers. JWT tokens are commonly used for stateless authentication.

Example:

```
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
```

3. What is the difference between [Authorize] and [AllowAnonymous] attributes?

[Authorize] restricts access to authenticated users, optionally based on roles or policies, while [AllowAnonymous] explicitly allows access to certain actions or controllers without authentication, even if the global policy requires authentication.

Example:

```
[Authorize]
public IActionResult GetUserData() { ... }

[AllowAnonymous]
public IActionResult PublicInfo() { ... }
```

4. How does JWT (JSON Web Token) authentication work in .NET Core?

JWT authentication works by issuing a signed token after a user logs in, which the client sends with subsequent requests. The server validates the token's signature and claims to authenticate the user without storing session state.

Sai Reddy
saireddy-dotnetfs

Example:

```
// Token generation example
var token = new JwtSecurityToken(
   issuer: "yourIssuer",
   audience: "yourAudience",
   claims: claims,
   expires: DateTime.Now.AddHours(1),
   signingCredentials: creds);
var tokenString = new JwtSecurityTokenHandler().WriteToken(token);
```

5. How do you validate a JWT token in .NET Core API?

JWT validation is handled automatically by the JwtBearer middleware in the authentication pipeline, which checks the token's signature, expiration, issuer, and audience against configured parameters.

Example:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
      .AddJwtBearer(options =>
      {
         options.TokenValidationParameters = new TokenValidationParameters
         {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidIssuer = "yourIssuer",
            ValidAudience = "yourAudience",
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey"))
         };
      });
```

6. What is CORS and why is it important for APIs?

CORS (Cross-Origin Resource Sharing) is a security feature implemented by browsers to restrict web pages from making requests to a different domain than the one that served the web page. It's important for APIs to configure CORS to allow legitimate client domains while blocking unauthorized ones.

Example:

```
services.AddCors(options =>
{
   options.AddPolicy("AllowSpecificOrigin",
      builder => builder.WithOrigins("https://example.com")
                  .AllowAnyMethod()
                  .AllowAnyHeader());
});

app.UseCors("AllowSpecificOrigin");
```

7. How do you enable HTTPS redirection in .NET Core Web API?

HTTPS redirection forces all HTTP requests to be redirected to HTTPS, ensuring secure communication. In .NET Core, it is enabled using the UseHttpsRedirection() middleware.

Example:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   app.UseHttpsRedirection();
   // Other middlewares
}
```

8. What is appsettings.json and how can secrets be stored securely?

appsettings.json is the default configuration file in .NET Core apps used to store settings like connection strings or API keys. Secrets should not be stored in this file for production; instead, use secure storage like Azure Key Vault or the Secret Manager tool during development.

Example:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=MyDb;Trusted_Connection=True;"
  }
}
For secure secrets:
dotnet user-secrets set "Jwt:Secret" "your_secret_key"
```

9. What are common security headers used in Web APIs?

Common security headers include Strict-Transport-Security (enforces HTTPS), Content-Security-Policy (restricts resources the client can load), X-Content-Type-Options (prevents MIME type sniffing), and X-Frame-Options (prevents clickjacking).

Example:

```
app.Use(async (context, next) =>
{
   context.Response.Headers.Add("X-Content-Type-Options", "nosniff");
   context.Response.Headers.Add("X-Frame-Options", "DENY");
   await next();
});
```

10. How do you protect your API from basic attacks like SQL Injection or Cross-Site Scripting (XSS)?

To prevent SQL Injection, use parameterized queries or ORM frameworks like Entity Framework. For XSS, sanitize all user inputs and outputs, and rely on built-in encoding or sanitization features on both client and server sides.

Example:

```
// Parameterized query example with EF Core
```

```
var user = await _context.Users
                .Where(u => u.Username == inputUsername)
                .FirstOrDefaultAsync();
```

Rest API .Net 3-7 Exp

1. Explain how to implement role-based and policy-based authorization in .NET Core.

Answer:

Role-based authorization restricts access based on user roles, such as "Admin" or "User". Policy-based authorization uses custom requirements and handlers to enforce complex rules beyond roles. You define policies during startup and apply them using the [Authorize] attribute.

Example:

```
// Startup.cs - ConfigureServices
services.AddAuthorization(options =>
{
   options.AddPolicy("RequireAdminRole", policy => policy.RequireRole("Admin"));
   options.AddPolicy("Over18Only", policy => policy.RequireClaim("Age", "18"));
});

// Controller usage
[Authorize(Policy = "RequireAdminRole")]
public IActionResult AdminOnly() { ... }
```

2. What is IdentityServer and how does it help in API security?

Answer:

IdentityServer is an OpenID Connect and OAuth 2.0 framework for .NET that enables centralized authentication and authorization for APIs and apps. It issues tokens, manages user identity, supports single sign-on, and secures APIs by delegating authentication.

3. What are Claims and how are they used in authentication and authorization?

Answer:

Claims are key-value pairs that represent user information (like email, roles, or permissions). They are embedded in tokens and used to make authorization decisions by checking user attributes rather than just roles.

Example:

```
var claims = new List<Claim>
{
   new Claim(ClaimTypes.Name, "JohnDoe"),
   new Claim(ClaimTypes.Role, "Admin")
};
var identity = new ClaimsIdentity(claims, "Jwt");
var userPrincipal = new ClaimsPrincipal(identity);
```

4. How do you secure sensitive data in configuration files?

Answer:

Sensitive data like API keys or connection strings should not be stored in plaintext. Use environment

variables, Azure Key Vault, AWS Secrets Manager, or .NET Core's Secret Manager tool during development. Encrypt sensitive config sections if necessary.

5. What is Data Protection API in .NET Core?

Answer:

The Data Protection API provides cryptographic services to protect data, such as encrypting cookies, tokens, or any sensitive information. It helps keep data safe at rest or in transit within an application.

Example:

```
var protector = dataProtectionProvider.CreateProtector("MyPurpose");
string protectedData = protector.Protect("sensitive data");
string unprotectedData = protector.Unprotect(protectedData);
```

6. How do you implement refresh tokens in JWT authentication?

Answer:

Refresh tokens are long-lived tokens stored securely and used to obtain new short-lived JWT access tokens without requiring the user to reauthenticate. The server validates refresh tokens and issues new JWT tokens.

7. What are the best practices for API key management?

Answer:

Best practices include generating unique keys per client, limiting scope and lifetime, using HTTPS, storing keys securely (not in client-side code), rotating keys regularly, and monitoring usage for anomalies.

8. How do you handle authentication in microservices architecture using .NET Core?

Answer:

Use centralized authentication services like IdentityServer to issue tokens, then validate those tokens in each microservice. Use API gateways to handle security and pass tokens to downstream services. Keep microservices stateless.

9. How can you throttle or rate-limit requests in a .NET Core API?

Answer:

Use middleware or third-party libraries like AspNetCoreRateLimit to limit the number of requests a client can make within a time window, protecting the API from abuse or denial-of-service attacks.

Example:

```
services.AddInMemoryRateLimiting();
services.Configure<IpRateLimitOptions>(options => {
   options.GeneralRules = new List<RateLimitRule> {
      new RateLimitRule { Endpoint = "*", Limit = 100, Period = "1m" }
   };
});
app.UseIpRateLimiting();
```

10. How do you perform logging and auditing in a secure way in .NET Core Web API?

Answer:

Use built-in logging providers with structured logging (e.g., Serilog) to capture important security

events like login attempts and access denials. Protect logs from unauthorized access, avoid logging sensitive data, and implement audit trails with timestamps and user IDs.

Rest API .Net 7-14 years Exp

1. How do you design a secure multi-tenant architecture for APIs in .NET Core?

Answer:

In multi-tenant APIs, isolate tenant data logically (or physically) and enforce tenant-specific access controls. Use tenant identifiers in claims or headers, validate them on each request, and apply authorization policies accordingly. Also, encrypt tenant data and isolate tenant configurations.

Example:

```
public class TenantMiddleware
{
   public async Task Invoke(HttpContext context)
   {
      var tenantId = context.Request.Headers["X-Tenant-ID"].FirstOrDefault();
      context.Items["TenantId"] = tenantId;
      await _next(context);
   }
}
// Use TenantId in services to filter data by tenant.
```

2. Compare OAuth2 and OpenID Connect. How would you implement each in .NET Core?

Answer:

OAuth2 is an authorization framework to delegate access, while OpenID Connect (OIDC) extends OAuth2 to provide authentication (identity) with standardized user info. Use Microsoft.AspNetCore.Authentication.JwtBearer for OAuth2 tokens and OpenIdConnect middleware for OIDC.

Example:

```
// OAuth2 Bearer token validation
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
   .AddJwtBearer(options => { /* configure options */ });

// OpenID Connect client
services.AddAuthentication(options =>
{
   options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
   options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddOpenIdConnect(options => { /* configure OIDC options */ });
```

3. How would you architect a secure API Gateway for microservices in .NET Core?

Answer:

Use API Gateway as a centralized security layer that handles authentication, authorization, SSL termination, rate limiting, logging, and request routing. Validate tokens, enforce policies, and use mutual TLS for downstream communication.

4. Explain Zero Trust Security and how to implement it in .NET Core APIs.

Answer:

Zero Trust means "never trust, always verify" — authenticate and authorize every request regardless of network location. Implement strict identity verification, least privilege access, continuous monitoring, and network segmentation in .NET Core using authentication middleware and policies.

5. How do you prevent and mitigate CSRF in a token-based authentication system?

Answer:

CSRF is less common in token-based (JWT) APIs since tokens are usually sent in headers, not cookies. To mitigate further, use SameSite cookies, check origin/referrer headers, and implement anti-CSRF tokens if cookies are involved.

6. What is certificate-based authentication and how is it implemented in .NET Core?

Answer:

Certificate-based authentication uses client certificates to verify identity. In .NET Core, configure middleware to require and validate client certificates during TLS handshake.

Example:

```
services.AddAuthentication(CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
       options.AllowedCertificateTypes = CertificateTypes.All;
       options.RevocationMode = X509RevocationMode.NoCheck;
    });
```

7. How do you secure communication between microservices?

Answer:

Use mutual TLS (mTLS) to authenticate both client and server, encrypt communication, and use short-lived tokens or certificates for authorization. Network policies and firewalls also help isolate services.

8. Explain custom authorization handlers and use-cases in complex business logic.

Answer:

Custom authorization handlers evaluate fine-grained access logic beyond roles/claims, e.g., checking if a user owns a resource or meets dynamic conditions. They implement IAuthorizationHandler and handle specific requirements.

Example:

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
   public int MinimumAge { get; }
   public MinimumAgeRequirement(int age) => MinimumAge = age;
}

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
```

```
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
MinimumAgeRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth))
        {
            var dob = DateTime.Parse(context.User.FindFirst(c => c.Type ==
ClaimTypes.DateOfBirth).Value);
            int age = DateTime.Today.Year - dob.Year;
            if (age >= requirement.MinimumAge) context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

9. What is penetration testing, and how do you make your APIs test-ready for security audits?

Answer:

Penetration testing simulates attacks to find vulnerabilities. To prepare, follow secure coding practices, enable detailed logging, configure test environments, sanitize inputs, and use static analysis tools. Document security controls and fix identified issues promptly.

10. How do you manage secrets in production environments securely (e.g., Azure Key Vault, AWS Secrets Manager)?

Answer:

Use dedicated secret management services like Azure Key Vault or AWS Secrets Manager to store sensitive info securely. Access secrets at runtime via SDKs or environment variables, avoid hardcoding secrets, and rotate them regularly.

Example:

```
var keyVaultClient = new SecretClient(new Uri("https://myvault.vault.azure.net/"), new
DefaultAzureCredential());
KeyVaultSecret secret = await keyVaultClient.GetSecretAsync("MySecret");
string secretValue = secret.Value;
```

SQL Server security-related questions 0-2 exp

1. What are the authentication modes in SQL Server?

Answer:

SQL Server supports two authentication modes:

- **Windows Authentication**: Uses Active Directory accounts.
- **SQL Server Authentication**: Uses SQL Server-specific username and password.

These are configured during installation or via SQL Server Management Studio (SSMS) under **Server Properties > Security**.

2. What is the difference between SQL Server authentication and Windows authentication?

Answer:

- **Windows Authentication** is more secure and uses Kerberos protocol; credentials are managed by Windows.

- **SQL Server Authentication** is used when Windows Auth is not possible, but passwords must be managed and stored separately.

3. What is a login and a user in SQL Server?

Answer:

- **Login**: Used for connecting to SQL Server (server-level).
- **User**: Used for accessing a specific database (database-level).
  A login can map to multiple users in different databases.

Example:

```
CREATE LOGIN MyLogin WITH PASSWORD = 'StrongP@ssword123';
USE MyDatabase;
CREATE USER MyUser FOR LOGIN MyLogin;
```

4. How do you grant and revoke permissions on a table?

Answer:

You use the GRANT, REVOKE, and DENY commands to manage permissions.

Example:

```
GRANT SELECT, INSERT ON dbo.Employee TO MyUser;
REVOKE INSERT ON dbo.Employee FROM MyUser;
```

5. What are roles in SQL Server and why are they used?

Answer:

Roles group users together to manage permissions collectively. SQL Server includes fixed roles (e.g., db_datareader, db_owner) and allows custom roles.

Example:

```
CREATE ROLE SalesTeam;
GRANT SELECT ON Orders TO SalesTeam;
EXEC sp_addrolemember 'SalesTeam', 'MyUser';
```

6. What is the purpose of the sp_change_users_login stored procedure?

Answer:

It's used to fix orphaned users—users in a database not linked to a login, usually after a restore. This procedure is deprecated in favor of ALTER USER.

Example:

```
-- Deprecated way
EXEC sp_change_users_login 'Auto_Fix', 'MyUser', null, 'Password';

-- Recommended
ALTER USER MyUser WITH LOGIN = MyLogin;
```

7. What is the difference between GRANT, DENY, and REVOKE?

Answer:

- **GRANT**: Gives permission to a user or role.
- **DENY**: Explicitly prevents access, even if permission is granted elsewhere.
- **REVOKE**: Removes previously granted or denied permission.

8. How can you secure your connection string in an application?

Answer:

- Use **Windows Authentication** where possible.
- Avoid storing connection strings in code.
- Store in configuration files with **encryption**, or use secure stores like **Azure Key Vault** or **AWS Secrets Manager**.

9. What are SQL Injection attacks and how do you prevent them?

Answer:

SQL Injection occurs when user input is embedded directly into SQL queries, allowing attackers to execute malicious SQL.

Prevention:

- Use **parameterized queries** or **stored procedures**.
- Validate and sanitize input.

Example:

```
var cmd = new SqlCommand("SELECT * FROM Users WHERE Username = @Username", conn);
cmd.Parameters.AddWithValue("@Username", username);
```

10. What is the use of the WITH ENCRYPTION clause in stored procedures?

Answer:

It hides the definition of stored procedures, functions, or views from users, helping to protect intellectual property or sensitive logic.

Example:

```
CREATE PROCEDURE SecretProc
WITH ENCRYPTION
AS
BEGIN
   SELECT * FROM SensitiveData;
END
```

SQL Server Security interview questions 3-7 y Exp

1. What are fixed server roles and fixed database roles in SQL Server?

Answer:

- **Fixed Server Roles** operate at the server level (e.g., sysadmin, securityadmin, serveradmin).
- **Fixed Database Roles** operate at the database level (e.g., db_datareader, db_owner, db_ddladmin).

These predefined roles help manage permissions efficiently.

Example:

```
EXEC sp_addsrvrolemember 'MyLogin', 'securityadmin';
EXEC sp_addrolemember 'db_datareader', 'MyUser';
```

2. How do you implement row-level security in SQL Server?

Answer:

Row-Level Security (RLS) allows filtering rows based on the user accessing the data. It uses **security policies** and **predicate functions**.

Example:

```
CREATE FUNCTION dbo.fnRLS()
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS fn_result
WHERE USER_NAME() = 'Manager1';

CREATE SECURITY POLICY SalesFilter
ADD FILTER PREDICATE dbo.fnRLS() ON dbo.Sales
WITH (STATE = ON);
```

3. What is Transparent Data Encryption (TDE)?

Answer:

TDE encrypts the entire database at rest, protecting data files and backups. It uses a **Database Encryption Key (DEK)** protected by a **certificate** stored in the master database.

Example:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'StrongPassword';
CREATE CERTIFICATE TDECert WITH SUBJECT = 'TDE Cert';
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE TDECert;
ALTER DATABASE MyDatabase SET ENCRYPTION ON;
```

4. Explain Always Encrypted in SQL Server.

Answer:

Always Encrypted protects sensitive data (like credit card numbers) using **client-side encryption**. Encryption keys never reach SQL Server, ensuring end-to-end protection.

It supports:

- **Deterministic** encryption (for joins, indexing).
- **Randomized** encryption (more secure but less functional).

Use SQL Server Management Studio (SSMS) or PowerShell to configure it.

5. How do you audit database activity in SQL Server?

Answer:

Use **SQL Server Audit** to track access and changes. Audits can log to files, event logs, or audit tables.

Example:

```
CREATE SERVER AUDIT Audit_Security
TO FILE (FILEPATH = 'C:\AuditLogs\');
CREATE SERVER AUDIT SPECIFICATION AuditLogins
FOR SERVER AUDIT Audit_Security
ADD (FAILED_LOGIN_GROUP);
ALTER SERVER AUDIT Audit_Security WITH (STATE = ON);
```

6. How would you handle orphaned users after restoring a database?

Answer:

An orphaned user exists without a matching login. Use ALTER USER to fix.

Example:

```
ALTER USER MyUser WITH LOGIN = MyLogin;
```

Previously, sp_change_users_login was used (now deprecated).

7. What is SQL Server Agent Proxy account and when would you use it?

Answer:

A **Proxy Account** allows SQL Agent jobs to execute steps under a security context different from SQL Server Agent's service account. Useful for running SSIS, CmdExec, or PowerShell tasks securely.

Example:

- Create a credential
- Associate it with a proxy
- Assign the proxy to a job step

8. What is the difference between symmetric and asymmetric encryption in SQL Server?

Answer:

- **Symmetric Encryption**: Uses the same key to encrypt/decrypt (faster, used for large data).
- **Asymmetric Encryption**: Uses a public-private key pair (slower, used for secure key exchange).

Example:

```
-- Symmetric
CREATE SYMMETRIC KEY MyKey WITH ALGORITHM = AES_256 ENCRYPTION BY PASSWORD =
'Pass@123';
-- Asymmetric
CREATE ASYMMETRIC KEY MyAsymKey WITH ALGORITHM = RSA_2048;
```

9. How do you restrict access to sensitive columns in a table?

Answer:

- Use **column-level permissions**: GRANT, DENY on specific columns.
- Use **views** to expose only required columns.
- Apply **Always Encrypted** for client-side protection.

Example:

```
DENY SELECT ON dbo.Employees(Salary) TO AnalystUser;
```

10. How do you monitor failed login attempts or security-related events?

Answer:

Options include:

- **SQL Server Audit**
- **SQL Server Error Log**
- **Login triggers**
- **Extended Events**

Example:

```
EXEC xp_readerrorlog 0, 1, 'Login failed';
```

Also, configure SQL Server to log failed logins:

In SSMS: Server > Properties > Security > Login Auditing > "Failed logins only"

SQL Server security questions (7–14 years experience)

1. How would you design a secure multi-tenant database system in SQL Server?

Answer:

Approaches:

- **Shared Database, Shared Schema**: Add a TenantId column in every table; secure with **Row-Level Security (RLS)**.
- **Shared Database, Separate Schemas**: Each tenant gets its own schema; restrict access via schema-level permissions.
- **Isolated Databases per Tenant**: Each tenant has its own database; more secure and scalable.

Security Best Practices:

- Enforce tenant isolation via **predicate functions (RLS)**.
- Encrypt tenant data using **Always Encrypted**.
- Manage tenants using metadata and secure APIs.

2. How do you manage encryption keys securely in a large environment (e.g., EKM, Azure Key Vault integration)?

Answer:

- Use **Extensible Key Management (EKM)** to store encryption keys outside SQL Server (e.g., **HSM** or **Azure Key Vault**).
- Rotate keys regularly and enforce **key hierarchy**: TDE protects DEK, which encrypts data.
- Monitor and audit key usage with **SQL Audit** and Azure logging.

Azure Key Vault Integration Example:

```
-- In Azure SQL: Transparent Data Encryption with Key Vault key
ALTER DATABASE MyDb SET ENCRYPTION ON;
```

3. Describe a comprehensive SQL Server security hardening strategy.

Answer:

1. **Least privilege principle** for all accounts.
2. Disable unused features (e.g., xp_cmdshell).
3. Enforce **strong password policies**.
4. Encrypt data at rest (TDE) and in transit (SSL/TLS).
5. Implement **firewall rules**, **auditing**, **RLS**, and **encryption**.
6. Regularly patch SQL Server and OS.

Tools: Microsoft Defender for SQL, SQL Vulnerability Assessment, CIS Benchmark.

4. How would you implement auditing and alerting on privileged user activity?

Answer:

- Use **SQL Server Audit** for actions by sysadmin or db_owner.
- Log to files or Windows Event Log.
- Create alerts via **SQL Agent** or **Extended Events**.

Example:

```
CREATE SERVER AUDIT PrivilegedAudit TO FILE (FILEPATH = 'C:\AuditLogs\');
CREATE SERVER AUDIT SPECIFICATION PrivilegedSpec
FOR SERVER AUDIT PrivilegedAudit
```

```
ADD (SCHEMA_OBJECT_CHANGE_GROUP)
WHERE (server_principal_name = 'sa');
```

5. What are the trade-offs between TDE and Always Encrypted?

Feature	TDE	Always Encrypted
Scope	Entire DB	Column-level
Data in transit	Not encrypted	Encrypted
Key location	SQL Server	Client-side (app key store)
Use cases	Compliance, backups	Highly sensitive fields (SSNs)
Joins/Filters	Supported	Limited in Always Encrypted

6. How do you ensure secure data access across linked servers?

Answer:

- Use **mapped logins** instead of Be made using this security context.
- Enable **RPC** only if necessary.
- Restrict access via **firewalls**, and **least privilege**.
- Encrypt traffic using **TLS**.

Example:

```
EXEC sp_addlinkedserver ...;
EXEC sp_addlinkedsrvlogin 'MyLinkedServer', 'false', NULL, 'RemoteUser', 'Password';
```

7. Explain how to implement a custom security model using application roles.

Answer:

Application Roles provide a way to assume specific permissions from the application rather than user identity.

Steps:

1. Create an application role.
2. Securely pass the password to the application.
3. Activate the role using sp_setapprole.

Example:

```
EXEC sp_setapprole 'SalesAppRole', 'StrongAppRolePassword';
This switches the context to the application role, restricting actions to only what that role allows.
```

8. How would you isolate environments (Dev/Test/Prod) from a security perspective in SQL Server?

Answer:

- **Separate Instances or VMs** for each environment.
- **Firewall and network isolation** (e.g., different subnets/VLANs).
- No production data in lower environments unless masked or anonymized.
- Use **role-based access control (RBAC)** to restrict access.
- Enforce **different encryption keys** and **access audit trails** per environment.

9. How do you manage access to SQL Server in a hybrid cloud setup securely?

Answer:

- Use **Azure Active Directory Authentication** where possible.
- Enable **TLS encryption** for all SQL endpoints.

- Restrict access using **firewall rules**, **network security groups**, and **Private Link** in Azure.
- Use **Just-In-Time (JIT)** access and **multi-factor authentication (MFA)** for administrative access.

10. What compliance standards (e.g., GDPR, HIPAA) have you implemented security features for, in SQL Server?

Answer:

For **GDPR/HIPAA/PCI-DSS**, I've implemented:

- **Data encryption** (TDE, Always Encrypted).
- **Data masking** and **row-level security**.
- **Auditing and logging** of data access.
- **Data classification** using SSMS tools.
- Ensured **least privilege** access, secure backups, and disaster recovery plans.

Also used **SQL Vulnerability Assessment** and **Microsoft Purview** (Azure) for compliance audits.

Angular security-related interview questions 0-2 y exp

1. What are common security threats in Angular applications?

Answer:

Common threats include:

- **Cross-Site Scripting (XSS)**
- **Cross-Site Request Forgery (CSRF)**
- **Insecure JWT storage**
- **Exposing sensitive information in the frontend**
- **Insecure API communication (HTTP instead of HTTPS)**

Angular provides **built-in XSS protection**, content sanitization, and security practices like route guards.

2. What is Cross-Site Scripting (XSS)? How does Angular help prevent it?

Answer:

XSS is when an attacker injects malicious scripts into a web page. Angular automatically **sanitizes data bindings** to neutralize scripts in the DOM.

Example – Safe Binding:

```
<p>{{ userInput }}</p> <!-- Angular escapes scripts -->
```

3. How does Angular sanitize HTML content?

Answer:

Angular uses its **built-in sanitizer** to clean potentially dangerous content like HTML, URLs, styles, etc. This prevents injection attacks when using properties like [innerHTML].

Example:

```
<div [innerHTML]="someHtmlString"></div> <!-- Angular sanitizes it -->
```

4. What is the DomSanitizer service in Angular?

Answer:

DomSanitizer is a service used to **bypass Angular's built-in sanitization** safely when you trust the content source.

Example:

```
constructor(private sanitizer: DomSanitizer) {}

safeHtml = this.sanitizer.bypassSecurityTrustHtml('<div>Safe Content</div>');
Use this only when you're sure the content is safe.
```

5. What is the difference between [innerHTML] and {{ binding }} in terms of security?

Answer:

- {{ binding }} escapes HTML, so it's safe from XSS.
- [innerHTML] renders raw HTML and is **more dangerous** if used with untrusted content.

Example:

```
<!-- Safe -->
<p>{{ '<b>Bold</b>' }}</p>

<!-- Risky -->
<div [innerHTML]="userInputHtml"></div>
```

6. What are safe navigation operators (?.) and why are they useful?

Answer:

The safe navigation operator (?.) prevents **null/undefined reference errors** in templates, improving code safety and security.

Example:

```
<p>{{ user?.profile?.name }}</p>
If user or profile is null, it avoids runtime errors.
```

7. What is CORS and how is it handled in Angular applications?

Answer:

CORS (Cross-Origin Resource Sharing) is a **browser security feature** that blocks requests to different domains unless allowed.

Handled at API level (backend) – Angular sends a request, and the backend must allow it via headers.

Example Angular call:

```
this.http.get('https://api.example.com/data').subscribe();
Server must respond with:
Access-Control-Allow-Origin: *
```

8. How do you handle authentication in an Angular application?

Answer:

Common steps:

- Login via HTTP request to backend
- Receive **JWT token**
- Store it (e.g., localStorage)
- Send it in headers for secure API access
- Use **Route Guards** for protected routes

Example – Sending Token:

```
const headers = new HttpHeaders().set('Authorization', `Bearer ${token}`);
this.http.get('/api/data', { headers });
```

9. How do you store JWT tokens securely in an Angular app?

Answer:

- Prefer **HttpOnly cookies** for sensitive tokens (best security).
- If stored on client:
  - Use sessionStorage (less persistent)
  - Avoid localStorage if possible
  - Never store sensitive data in plain JS

Avoid exposing tokens in browser-accessible locations.

10. What are security best practices for Angular forms?

Answer:

- **Use Angular's built-in form validation** to prevent invalid input.
- Avoid binding form fields directly to critical data.
- Sanitize values before sending to API.
- Use **form-level validation**, e.g., email format, min/max length.
- Prevent **autofill** for sensitive fields using autocomplete="off".

Example:

```
<input type="password" formControlName="password" autocomplete="off">
```

Angular security questions 3- 7 y exp

1. What are Angular route guards? How do you use them for security?

Answer:

Route guards control access to routes based on conditions like authentication or roles. They protect sensitive pages from unauthorized users.

Example:

```
canActivate(): boolean {
  return this.authService.isLoggedIn();
}
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }
```

2. Explain the different types of route guards (CanActivate, CanDeactivate, etc.).

Answer:

- **CanActivate**: Checks if a route can be accessed.
- **CanActivateChild**: Checks access to child routes.
- **CanDeactivate**: Confirms navigation away from a component.
- **Resolve**: Pre-fetches data before loading a route.
- **CanLoad**: Prevents lazy-loaded module loading.

Use-case example – CanDeactivate:

```
canDeactivate(component: FormComponent): boolean {
  return component.hasUnsavedChanges() ? confirm("Discard changes?") : true;
}
```

3. How do you implement role-based access control in Angular?

Answer:

You restrict access to routes and components by checking the user's role from a token or service.

Example:

```
canActivate(): boolean {
  const user = this.authService.getUser();
  return user?.role === 'admin';
}
Also store role-based conditions in route.data:
{ path: 'admin', component: AdminComponent, canActivate: [RoleGuard], data: { roles: ['admin'] }
}
```

4. What are the security risks of using localStorage/sessionStorage for storing tokens?

Answer:

- They are **accessible via JavaScript**, making them vulnerable to **XSS attacks**.
- Data **persists across sessions** (localStorage), increasing exposure risk.
- No **automatic transmission** of tokens like cookies with HttpOnly flag.

Best practice: Use **HttpOnly cookies** managed by the backend where possible.

5. How do you handle CSRF (Cross-Site Request Forgery) in Angular applications?

Answer:

Angular itself doesn't protect against CSRF, but when using **cookies for tokens**, CSRF becomes a concern.

Mitigation strategies:

- Use **anti-CSRF tokens** sent in headers.
- Use SameSite=Strict or SameSite=Lax in cookie settings.
- Use **HttpOnly cookies** for storing tokens (if supported by backend).

6. How can you prevent unauthorized access to REST APIs from Angular frontend?

Answer:

- Implement **authentication** (JWT or OAuth2).
- Secure endpoints with server-side authorization.
- Always verify token server-side.
- Avoid exposing sensitive logic in frontend code.

Example in Angular (Interceptor):

```
req = req.clone({
  headers: req.headers.set('Authorization', `Bearer ${token}`)
});
```

7. What is the best way to implement HTTP Interceptors for security (e.g., adding tokens, handling 401)?

Answer:

Use **HttpInterceptor** to:

- Attach tokens to outgoing requests
- Handle 401 Unauthorized errors globally

Example:

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  const token = this.authService.getToken();
  if (token) {
    req = req.clone({ setHeaders: { Authorization: `Bearer ${token}` } });
```

```
  }
  return next.handle(req).pipe(
    catchError(err => {
      if (err.status === 401) this.authService.logout();
      return throwError(err);
    })
  );
}
```

8. How do you log out a user securely and invalidate a token?

Answer:

- Clear token from client storage.
- Invalidate the token on the server if possible (e.g., via a token blacklist).
- Redirect to login.

Example:

```
logout() {
  localStorage.removeItem('token');
  this.router.navigate(['/login']);
}
```

9. How do you secure Angular components and modules from unauthorized users?

Answer:

- Use **route guards** for access control.
- Use **ngIf** or **hidden attributes** to hide UI elements.
- Lazy-load modules and protect with CanLoad.

Example:

```
<button *ngIf="user.role === 'admin'">Delete</button>
```

Module route:

```
{ path: 'reports', loadChildren: () => ReportsModule, canLoad: [AuthGuard] }
```

10. How do you use environment variables securely in Angular?

Answer:

Angular uses environment.ts for build-time variables. These are exposed in the frontend, so **never store secrets there**.

Best practices:

- Store only **non-sensitive config** (e.g., API URLs).
- Secure secrets on the **backend**, not frontend.
- Use different environment.ts for dev, prod.

Example:

```
export const environment = {
  production: false,
  apiUrl: 'https://api.example.com'
};
Use it in services:
this.http.get(`${environment.apiUrl}/data`);
```

Angular security questions (7–14 years) exp:

1. How would you architect a secure enterprise-grade Angular application?

Answer:

- **Modular architecture** with lazy loading and role-based modules.
- **Route guards** and **interceptors** for access control and secure communication.
- Use **environment configs** for URLs, not secrets.
- **JWT-based authentication** with token refresh.
- **XSS protection** using Angular's sanitization and DomSanitizer.
- Backend must implement **rate limiting, input validation, and CORS**.

Example:

```
{ path: 'admin', loadChildren: () => AdminModule, canLoad: [RoleGuard] }
```

2. How do you handle security in Angular for microfrontend architecture?

Answer:

- **Isolate microfrontends** using sandboxed iframes or Webpack Module Federation.
- **Single source of truth for auth**, often at the shell app level.
- Share JWT/session via secure means (e.g., **cookie + SameSite**).
- Apply **Content Security Policy (CSP)** headers to prevent cross-app XSS.

Example Concept:

Shell app authenticates, sets HttpOnly cookie → microfrontends read user roles via shared service or API.

3. How do you perform penetration testing or static code analysis on Angular code?

Answer:

- Use **tools like SonarQube**, **ESLint**, and **Retire.js** to detect vulnerable dependencies.
- Perform **OWASP ZAP** testing on deployed app.
- Check for:
  - Insecure innerHTML
  - Token leakage
  - Unpatched third-party libraries

Example:

```
npm audit
npx retire
```

4. Explain how you secure data transmission in Angular apps (HTTPS, SSL pinning in mobile versions, etc.)

Answer:

- Always deploy Angular over **HTTPS**.
- Use **SSL pinning** in mobile wrappers (e.g., with Capacitor or Cordova plugins).
- Angular communicates with secure APIs using https://.

Angular usage:

```
this.http.get('https://api.example.com/data')
```

Mobile (SSL pinning): Use plugins like cordova-plugin-advanced-http.

5. How do you prevent sensitive data exposure on the client side in Angular?

Answer:

- Never store secrets (API keys, passwords) in Angular.
- Use **environment.ts** only for non-sensitive info.
- Store tokens securely and avoid console logs or localStorage for sensitive data.
- Remove debug data in production builds (ng build --prod).

6. What are the implications of third-party libraries on Angular app security?

Answer:

- They may introduce **vulnerabilities or outdated packages**.
- Always **audit dependencies** before adding.
- Use **Subresource Integrity (SRI)** when using CDN libraries.
- Regularly run:

```
npm audit fix
```

7. How do you secure Angular apps that consume APIs protected with OAuth2 / OpenID Connect?

Answer:

- Use **OAuth2 authorization code flow with PKCE** for SPAs.
- Secure token in memory or HttpOnly cookie.
- Use libraries like angular-oauth2-oidc.

Example:

```
this.oauthService.configure(authConfig);
this.oauthService.loadDiscoveryDocumentAndTryLogin();
Auth Config:
export const authConfig: AuthConfig = {
  issuer: 'https://auth.example.com',
  clientId: 'spa-client',
  responseType: 'code',
  redirectUri: window.location.origin,
  scope: 'openid profile email api',
  usePkce: true
};
```

8. What are security considerations for Angular apps deployed on cloud platforms (e.g., Azure, AWS)?

Answer:

- Use **HTTPS** via CDN (Azure Front Door, AWS CloudFront).
- Apply **WAF** (Web Application Firewall).
- Use **environment variables** via Azure App Config or AWS Parameter Store.
- Enable **CORS** only for trusted domains.
- Disable directory listing/static file access unless needed.

9. How do you handle multi-factor authentication (MFA) in an Angular SPA?

Answer:

- Integrate with identity providers like **Azure AD B2C**, **Auth0**, or **Okta** that support MFA.

- After primary login, prompt user for OTP or biometric (depending on IdP).
- Use **step-up authentication** for sensitive actions.

Example flow:

```
this.authService.loginWithRedirect({ screen_hint: 'mfa' });
```

10. What policies and tools do you use for ongoing vulnerability scanning and patching in Angular apps?

Answer:

- **CI/CD integration** with GitHub Actions or Azure DevOps to run:
  - npm audit
  - ESLint
  - OWASP Dependency-Check
- Regular **npm dependency upgrades**.
- Monitor CVEs via **Snyk**, **Dependabot**, or **WhiteSource Bolt**.
- Enforce code review policies to block known vulnerabilities.

Entity Framework Security Questions (0–2 years)

1. What is Entity Framework and how does it help in database security?

Entity Framework (EF) is an ORM tool that maps .NET objects to database tables. It promotes security by using parameterized queries internally, reducing SQL injection risks. It also enforces access through LINQ queries, abstracting direct SQL interaction.

```
var user = dbContext.Users.FirstOrDefault(u => u.Email == email);
```

2. How can you prevent SQL Injection in EF Core?

EF Core inherently uses parameterized queries, preventing SQL injection. Avoid raw SQL unless necessary, and never concatenate strings.

```
var user = dbContext.Users
   .FromSqlRaw("SELECT * FROM Users WHERE Email = {0}", email)
   .FirstOrDefault();
```

3. What are parameterized queries and how does EF use them?

Parameterized queries pass user input as parameters instead of injecting directly into SQL. EF Core generates such queries automatically from LINQ.

```
var products = dbContext.Products
   .Where(p => p.Name == "Laptop")
   .ToList(); // EF converts this to a parameterized SQL query
```

4. What is the role of DbContext in security?

DbContext is the bridge between your application and the database. It enforces access rules, manages entity tracking, and ensures queries are executed safely through EF's infrastructure.

```
public class AppDbContext : DbContext
{
   public DbSet<User> Users { get; set; }
}
```

5. How do you restrict access to certain data in EF using LINQ?

You can filter data using LINQ based on user roles, permissions, or ownership, ensuring users only see what's allowed.

```
var userData = dbContext.Orders
   .Where(o => o.UserId == currentUserId)
   .ToList();
```

6. Can EF be used with authentication and authorization in .NET Core?

Yes, EF Core works with ASP.NET Identity to implement authentication and role-based or claim-based authorization.

```
// Example: get orders only for logged-in user
var orders = dbContext.Orders
   .Where(o => o.UserId == userManager.GetUserId(User))
   .ToList();
```

7. What is the difference between FirstOrDefault() and Find() in terms of safe querying?

FirstOrDefault() executes a query immediately, while Find() searches the context first before querying the database. Find() is safer when you know the key and want to avoid unnecessary queries.

```
var user = dbContext.Users.Find(5); // Uses primary key
```

8. How does EF Core track changes and why is that important for security?

EF Core tracks entity changes to apply only necessary updates. This prevents unintentional data modification and supports audit logging for accountability.

```
dbContext.Entry(user).State = EntityState.Modified;
dbContext.SaveChanges();
```

9. What are migrations and do they pose any security concerns?

Migrations are used to manage schema changes. They can be risky if auto-applied in production without review. Always review and control them via CI/CD.

```
dotnet ef migrations add AddSecureColumn
dotnet ef database update
```

10. How do you encrypt sensitive data before saving via EF?

Manually encrypt fields before saving and decrypt after reading, as EF doesn't support encryption natively.

```
user.Password = EncryptHelper.Encrypt("mypassword");
dbContext.Users.Add(user);
dbContext.SaveChanges();
```

Entity Framework Core Security Questions (3–6 years experience)

1. How do you implement role-based data access using EF Core?

Use user claims or roles to filter data at the query level. Avoid loading data the user shouldn't see by enforcing restrictions in service/repository layers.

```
if (User.IsInRole("Admin"))
   return dbContext.Users.ToList();
else
```

```
return dbContext.Users.Where(u => u.Id == currentUserId).ToList();
```

2. What are the security risks of using Include() or lazy loading?

They can unintentionally expose sensitive navigation data. Use explicit projections instead of Include() when possible to avoid over-fetching.

```
var user = dbContext.Users
   .Select(u => new { u.Name, u.Email }) // Avoid exposing related entities
   .ToList();
```

3. How do you ensure secure connection strings in EF Core?

Never hardcode connection strings. Store them in appsettings.json, and use secure providers like Azure Key Vault in production.

```
// appsettings.json
"ConnectionStrings": {
  "DefaultConnection": "Server=.;Database=SecureDB;Trusted_Connection=True;"
}
```

4. How do you protect against over-posting in EF models?

Use view models or DTOs to limit fields exposed for binding. Avoid binding directly to entity models in controllers.

```
public class UpdateUserDto { public string Name { get; set; } }

[HttpPost]
public IActionResult Update(UpdateUserDto dto) {
   var user = dbContext.Users.Find(id);
   user.Name = dto.Name;
   dbContext.SaveChanges();
}
```

5. How do you audit changes made to the database through EF Core?

Intercept SaveChanges() to log changes to an audit table. Track EntityState to know what's been modified.

```
foreach (var entry in dbContext.ChangeTracker.Entries())
{
   if (entry.State == EntityState.Modified)
      auditLogger.Log(entry.Entity.ToString());
}
```

6. Explain how to use DbContextOptionsBuilder to enforce security settings.

You can configure connection encryption, timeouts, and logging using DbContextOptionsBuilder.

```
optionsBuilder.UseSqlServer(connectionString, opt =>
   opt.EnableRetryOnFailure().CommandTimeout(30));
```

7. What is the impact of tracking vs. no-tracking queries on data security?
AsNoTracking() improves performance but disables change tracking. Use it for read-only access to avoid accidental data exposure or updates.

```
var readOnlyData = dbContext.Products.AsNoTracking().ToList();
```

8. How can you encrypt/decrypt fields at the model level using EF?
Use shadow properties or override getters/setters for encryption logic manually.

```
public string EncryptedSSN { get; set; }
[NotMapped]
public string SSN
{
   get => Decrypt(EncryptedSSN);
   set => EncryptedSSN = Encrypt(value);
}
```

9. How do you prevent exposing internal model properties in APIs when using EF?
Use DTOs in API responses. Avoid returning full EF entities directly, especially with navigation properties.

```
public class UserDto { public string Name { get; set; } }

var dto = dbContext.Users.Select(u => new UserDto { Name = u.Name }).ToList();
```

10. How do you log and monitor SQL generated by EF for security auditing?
Enable logging in DbContextOptionsBuilder and write logs to a secure location or audit tool.

```
optionsBuilder.LogTo(Console.WriteLine); // or write to a file or logging service
```

Entity Framework Core Security Questions (7–14 years experience)

1. How do you implement multitenant database security using EF Core?
Implement multitenancy by adding a TenantId property to your entities and filtering queries by this ID. Use global query filters in OnModelCreating to enforce tenant isolation automatically.

```
modelBuilder.Entity<Order>().HasQueryFilter(o => o.TenantId == _currentTenantId);
```

2. How do you design row-level security using EF Core?
Use global query filters based on user claims or roles to restrict data rows dynamically. Combine with database-level RLS for defense in depth.

```
modelBuilder.Entity<Document>()
   .HasQueryFilter(d => d.OwnerId == _currentUserId);
```

3. Explain best practices for securing large-scale data access using EF in microservices.
Use scoped DbContexts per request, enforce strong authentication and authorization at service boundaries, and implement caching carefully to avoid data leaks.

4. How do you manage data ownership and access control in multi-user applications with EF?
Store ownership metadata (e.g., OwnerId) and enforce access via LINQ filters. Combine with ASP.NET Core authorization policies.

```
var docs = dbContext.Documents.Where(d => d.OwnerId == userId);
```

5. How do you integrate EF with identity providers (e.g., Azure AD, OAuth) for fine-grained access control?

Use claims from identity tokens to filter data queries and apply authorization policies before EF queries execute.

```
var userId = User.FindFirst(ClaimTypes.NameIdentifier).Value;
var data = dbContext.Data.Where(d => d.UserId == userId);
```

6. How would you ensure secure migrations across environments (Dev/Test/Prod)?

Review migration scripts carefully, use separate migration branches, apply migrations in controlled CI/CD pipelines, and avoid auto-applying on production.

7. How do you implement dynamic query filtering based on user roles or claims in EF Core?

Inject user context into DbContext and use conditional global filters or method filters based on roles/claims.

```
if (User.IsInRole("Admin"))
   return dbContext.Items;
else
   return dbContext.Items.Where(i => i.OwnerId == userId);
```

8. How do you avoid performance leaks or data overexposure in projection queries (e.g., via Select) using EF?

Only select required properties using DTOs, avoid eager loading large navigation properties unnecessarily, and avoid returning entire entities.

```
var data = dbContext.Users.Select(u => new UserDto { Name = u.Name }).ToList();
```

9. What are some security hardening techniques for EF Core in enterprise applications?

Disable lazy loading, use explicit loading, enable logging and monitoring, restrict database permissions per service, and encrypt sensitive fields manually.

10. How do you design and enforce secure transactional boundaries using EF in distributed systems?

Use explicit transactions with TransactionScope or distributed transaction coordinators, ensure idempotency, and keep transaction scope minimal to reduce lock duration.

```
using var transaction = await dbContext.Database.BeginTransactionAsync();
// perform operations
await transaction.CommitAsync();
```

CI/CD Security interview questions for 0–3 years experience

1. What is CI/CD and why is security important in it?

CI/CD stands for Continuous Integration and Continuous Deployment. Security is important to ensure that automated builds, tests, and deployments don't introduce vulnerabilities or expose sensitive data.

2. How do you secure your pipeline secrets like API keys or passwords?

Store secrets in secure vaults like Azure Key Vault, AWS Secrets Manager, or pipeline-specific secret managers, and never hardcode them in code or pipeline scripts.

Sai Reddy
saireddy-dotnetfs

3. What is the principle of least privilege in CI/CD?
It means giving the pipeline and its components only the permissions they absolutely need to perform tasks, reducing the risk if a credential is compromised.

4. How do you prevent unauthorized code from being deployed in CI/CD?
Use branch protection rules, require code reviews and approvals before merges, and enable pipeline triggers only on trusted branches.

5. What are some common vulnerabilities in CI/CD pipelines?
Exposed secrets, insecure artifact storage, insufficient access control, lack of pipeline auditing, and running pipelines with excessive permissions.

6. How can you enforce code quality and security checks in CI/CD?
Integrate static code analysis tools (SAST), dependency vulnerability scanners, and automated tests into the pipeline to block insecure code.

7. What is the role of artifact signing in CI/CD security?
Artifact signing ensures the integrity and authenticity of build outputs, so only trusted artifacts are deployed.

8. How do you handle rollback securely in CI/CD?
Maintain versioned and signed artifacts, automate rollback procedures, and audit rollback operations to prevent unauthorized rollbacks.

9. Why is auditing important in CI/CD pipelines?
Auditing tracks who triggered builds or deployments and helps detect and respond to suspicious activity or mistakes.

10. What practices help secure container images in CI/CD?
Use minimal base images, scan images for vulnerabilities during build, sign images, and pull images only from trusted registries.

API Gateway questions 0-2 exp

1. What is an API Gateway and why is it used?
An API Gateway is a server that acts as an entry point for client requests to backend services or microservices. It handles request routing, composition, and protocol translation. It simplifies client communication by providing a unified API surface. It also offers features like authentication, rate limiting, and caching. This helps improve security, scalability, and manageability.

2. How does an API Gateway differ from a reverse proxy?
A reverse proxy primarily forwards requests to backend servers but lacks service-specific logic. An API Gateway includes additional capabilities like authentication, rate limiting, request transformation, and aggregation. It is designed specifically for APIs, managing microservices complexity. A reverse proxy is more generic, mainly focusing on routing and load balancing. API Gateways provide more business-centric controls.

3. What are some common features provided by API Gateways?
Common features include request routing, load balancing, authentication and authorization, rate limiting/throttling, request and response transformation, caching, logging and monitoring, protocol translation (e.g., HTTP to gRPC), and support for API versioning. These help in managing and securing APIs efficiently.

4. How does an API Gateway help in securing microservices?
API Gateways enforce centralized authentication and authorization, usually integrating with OAuth2 or JWT tokens. They apply rate limiting to prevent abuse and throttle excessive requests. Gateways can also validate and sanitize requests to block malicious payloads. SSL termination is often handled here, ensuring encrypted traffic. This consolidates security at a single layer.

5. What is request throttling/rate limiting in API Gateways?
Throttling limits the number of API calls a client can make in a given time frame. This prevents server overload, protects against denial-of-service (DoS) attacks, and ensures fair resource usage. API Gateways can enforce rate limits per user, IP, or API key. Violations typically result in HTTP 429 (Too Many Requests) responses.

6. How do API Gateways handle authentication and authorization?
API Gateways integrate with identity providers (IdPs) and accept tokens like JWT or OAuth2 access tokens. They validate these tokens before forwarding requests. Gateways can enforce role-based or policy-based authorization to restrict API access. They also support API keys for simpler authentication methods.

7. What is the role of API Gateway in load balancing?
API Gateways distribute incoming requests across multiple backend instances or microservices to ensure availability and scalability. This improves performance by preventing any one service from becoming a bottleneck. Load balancing algorithms can be round-robin, least connections, or custom-defined.

8. Can you explain the difference between an API Gateway and a service mesh?
An API Gateway manages north-south traffic (client to services), handling request routing and security at the edge. A service mesh manages east-west traffic (service-to-service communication) within a microservices environment. Service meshes provide fine-grained control over internal service interactions, including retries, circuit breakers, and telemetry.

9. How do API Gateways support protocol translation?
API Gateways can translate requests between different protocols, such as from HTTP to gRPC or WebSocket, enabling clients to interact with services that use different communication protocols. This allows backend services to evolve independently of client protocols and facilitates integration.

10. What are the challenges of using an API Gateway?
Challenges include potential latency overhead as requests pass through the gateway, a single point of failure if not highly available, increased complexity in configuration and management, and the risk of overloading the gateway if not properly scaled. Monitoring and troubleshooting can also become more difficult.

Example: Basic Rate Limiting Policy in Azure API Management (API Gateway)

```
<rate-limit calls="10" renewal-period="60" />
```

Sai Reddy
saireddy-dotnetfs

This XML snippet in Azure API Management limits a client to 10 calls per 60 seconds, enforcing throttling to protect backend services.

API Gateway questions 3-6 exp

1. How do you implement caching strategies in an API Gateway?

API Gateways implement caching by storing frequently requested responses to reduce backend load and improve latency. Caches can be configured based on URL, headers, or query parameters. Time-to-live (TTL) settings control how long cached data remains valid. Cache invalidation policies ensure data freshness. This improves performance and scalability.

2. Explain how API Gateways handle request routing and versioning.

API Gateways route incoming requests to appropriate backend services based on URL patterns, headers, or query parameters. They support versioning by mapping different API versions to separate backend endpoints or services. This allows backward compatibility and controlled upgrades. Routing rules can be dynamic or static. Versioning helps clients migrate smoothly.

3. What is circuit breaker pattern and how does it relate to API Gateways?

The circuit breaker pattern prevents cascading failures by detecting failing services and halting requests temporarily. API Gateways implement this to monitor backend health and avoid overwhelming unstable services. When a service is down, the gateway returns fallback responses or errors quickly. This increases system resilience and uptime.

4. How does an API Gateway support monitoring and logging?

API Gateways collect metrics like request count, latency, error rates, and throughput. They generate detailed logs for requests and responses, including headers and status codes. Integration with monitoring tools or dashboards enables real-time analysis. Logs help troubleshoot issues and enforce security audits. This visibility improves operational health.

5. How can you implement rate limiting policies in an API Gateway?

Rate limiting controls how many requests a client can make in a time window, preventing abuse and DoS attacks. Policies can be configured per API key, IP, or user. Gateways reject excess requests with HTTP 429 status. Limits can be global or granular by endpoint. Rate limiting ensures fair resource distribution.

6. Explain the use of JWT and OAuth2 in API Gateway security.

API Gateways use OAuth2 for delegated authorization, allowing clients to access resources with tokens issued by identity providers. JWT tokens carry claims securely and are validated by the gateway to authenticate users. This enables stateless, scalable security without session management. Tokens also support role-based access control.

7. What is API Gateway throttling and how do you configure it?

Throttling limits the rate of incoming requests to protect backend services from overload. It's configured by setting maximum allowed requests per second or minute. When exceeded, the

gateway rejects requests with HTTP 429. Throttling complements rate limiting and ensures service stability during traffic spikes.

8. How do API Gateways integrate with identity providers?
Gateways connect with identity providers (IdPs) like Azure AD, Okta, or Auth0 using protocols like OAuth2 or OpenID Connect. They validate tokens issued by IdPs to authenticate and authorize API calls. This integration centralizes identity management and supports SSO. It simplifies secure access control.

9. Describe how to handle failures and retries in an API Gateway.
API Gateways implement retries with configurable policies for transient errors, avoiding immediate failures. They use exponential backoff and jitter to prevent thundering herd problems. For persistent failures, fallback mechanisms or circuit breakers can be triggered. This improves reliability and user experience.

10. What is the impact of API Gateway on latency and how do you mitigate it?
API Gateways introduce extra network hops, adding some latency. To mitigate this, caching, request aggregation, and efficient routing are used. Horizontal scaling of gateways reduces bottlenecks. Monitoring helps identify latency causes. Optimizing transformation and security checks also improves performance.

API Gateway questions 7-14 exp

1. How would you architect an API Gateway for a large-scale microservices ecosystem?
Design a scalable, fault-tolerant gateway cluster with load balancing and auto-scaling. Use routing rules to direct requests to appropriate microservices. Implement security features like authentication, authorization, rate limiting, and caching. Integrate with service discovery for dynamic backend resolution. Ensure observability with logging and metrics.

2. Explain Zero Trust Security model implementation in API Gateways.
Zero Trust means never trusting any request by default, verifying every access continuously. API Gateways enforce strict authentication, granular authorization, and continuous monitoring. They validate identity, device posture, and context before granting access. Every API call is inspected for compliance. This reduces attack surfaces significantly.

3. How do you manage multi-tenancy and tenant isolation in API Gateways?
Use tenant-specific routing and authentication to isolate traffic. Implement strict authorization policies per tenant. Enforce resource quotas and rate limits to prevent noisy neighbors. Encrypt tenant data separately and apply logging scoped per tenant. This ensures data privacy and fair resource use.

4. What strategies do you use for API Gateway high availability and disaster recovery?
Deploy gateways across multiple availability zones or regions for failover. Use health probes and automatic failover mechanisms. Implement data replication for configuration consistency. Use backup and restore processes for disaster recovery. Regularly test failover and recovery procedures.

5. How do you secure communication between the API Gateway and backend services?

Use mutual TLS (mTLS) to authenticate both gateway and backend services. Encrypt all traffic with TLS to prevent eavesdropping. Implement IP whitelisting and firewall rules. Use OAuth2 tokens or API keys for backend service authentication. Regularly rotate certificates and keys.

6. Describe the role of API Gateway in service mesh architecture.
The API Gateway acts as the ingress point, handling external client requests. It enforces global policies like authentication and rate limiting. The service mesh manages internal service-to-service communication with sidecars. The gateway and mesh together provide end-to-end security, observability, and routing control.

7. How do you implement custom authorization policies in API Gateways?
Create policy scripts or plugins that evaluate user claims, roles, or attributes. Integrate with external policy engines like OPA for fine-grained control. Policies can check request context, headers, or payloads. Deny or allow access based on business rules. This allows flexible, dynamic access control.

8. What are best practices for scaling API Gateways in a cloud environment?
Use container orchestration (e.g., Kubernetes) for automatic scaling. Enable horizontal scaling behind load balancers. Use stateless gateway designs for easy scaling. Monitor performance metrics and autoscale proactively. Optimize caching to reduce backend calls.

9. How do you perform API analytics and threat detection at the gateway level?
Collect detailed logs and metrics on API usage, latency, and errors. Integrate with SIEM and threat detection tools. Use anomaly detection to spot unusual patterns like spikes or repeated failed auth attempts. Implement alerting for suspicious activities. Use dashboards for continuous monitoring.

10. Discuss how you would implement dynamic request transformation and response aggregation.
Use gateway policies or middleware to modify headers, payloads, or URLs dynamically. Aggregate responses from multiple backend services into one consolidated API response. Use scripting or workflow engines for complex transformations. This improves client experience and reduces round trips.

11. How do you handle versioning and backward compatibility in APIs managed by a gateway?
Route different API versions to separate backend services or endpoints. Support URL path or header-based versioning. Maintain deprecated versions with limited support. Use API contracts and documentation for smooth migration. Gateways can rewrite requests for compatibility.

12. What are the security considerations for API Gateway when exposing third-party APIs?
Validate all incoming data to prevent injection attacks. Implement strict rate limiting and quotas. Use API keys or OAuth tokens to control access. Monitor third-party API behavior for anomalies. Encrypt communication and use secure protocols.

13. How do you integrate API Gateways with CI/CD pipelines for automated deployments?
Automate gateway configuration and policy deployment via Infrastructure as Code (IaC). Use pipeline stages for validation, testing, and rollback. Integrate gateway deployment with app release cycles. Enable version control on gateway configs. This ensures consistent, error-free updates.

14. Explain how you can enforce SLA and QoS policies in an API Gateway.

Sai Reddy
saireddy-dotnetfs

Define SLAs with rate limits, quotas, and response time objectives. Implement QoS with priority routing or throttling. Monitor SLA adherence in real time. Automatically throttle or reject requests violating SLAs. Report SLA violations for accountability.

Load Balancer interview questions (0–3 years experience)

1. What is a load balancer and why is it used?

A load balancer distributes incoming network traffic across multiple servers. It improves application availability, fault tolerance, and performance. It ensures no single server is overwhelmed. Load balancers can be hardware or software-based. They support various algorithms like round-robin or least connections.

Example:

Client --> Load Balancer --> Server1 / Server2 / Server3

2. What are the types of load balancers?

1. **Hardware** (e.g., F5, Cisco) – physical devices
2. **Software** (e.g., NGINX, HAProxy) – flexible, cloud-friendly
3. **Cloud-based** (e.g., Azure Load Balancer, AWS ELB) – scalable
4. **Layer 4 (Transport)** – works on TCP/UDP
5. **Layer 7 (Application)** – works on HTTP/HTTPS and content-aware

Example:

AWS ELB routes HTTP requests to multiple EC2 instances.

3. What is the difference between Layer 4 and Layer 7 load balancers?

Layer 4 works at TCP/UDP level, routing traffic based on IP and port.
Layer 7 works at the HTTP/HTTPS level and routes based on content.
Layer 7 enables smart routing (e.g., URL-based).
Layer 4 is faster, Layer 7 is more flexible.
Layer 7 allows SSL termination, header rewrites, etc.

Example:

Layer 7: /api/* → API servers, /web/* → Web servers

4. What is round-robin load balancing?

In round-robin, requests are sent to servers one by one in a loop.
It assumes all servers have equal capacity.
Simple and fast but not resource-aware.
Can lead to uneven loads if servers differ.
Supported in most software load balancers.

Example:

Req1 → Server A, Req2 → Server B, Req3 → Server C, Req4 → Server A...

5. What happens if a server fails behind a load balancer?

Most load balancers support health checks.
If a server fails the health check, it's removed from the pool.
Traffic is rerouted to healthy servers.
Once restored, it's added back automatically.
Ensures high availability and fault tolerance.

Sai Reddy
saireddy-dotnetfs

Example (NGINX health check):

```
upstream backend {
   server server1.com;
   server server2.com;
}
```

6. How does session persistence (sticky sessions) work in load balancers?

Sticky sessions route a user's requests to the same server.
Useful for stateful applications like shopping carts.
Implemented via cookies or IP-based affinity.
Prevents session loss during multi-request workflows.
Can reduce load distribution effectiveness.

Example:

User A → Server 1 for all requests in that session

7. What is SSL termination in load balancers?

SSL termination decrypts HTTPS traffic at the load balancer.
Back-end servers receive plain HTTP, reducing their load.
Improves performance, centralizes certificate management.
Can be a security concern if internal network isn't secured.
Use with re-encryption if full end-to-end encryption is required.

Example:

Client → HTTPS → Load Balancer → HTTP → Server

8. How do you perform health checks with a load balancer?

Health checks ping servers at intervals (e.g., via HTTP GET).
They check if the server responds within a timeout.
If not, it's marked unhealthy and removed temporarily.
Types: TCP, HTTP, HTTPS, command-line checks.
Custom scripts can be used for advanced checks.

Example (AWS ELB):

Ping path: /health, Interval: 30s, Timeout: 5s, Unhealthy threshold: 2

9. What is load balancing algorithm Least Connections?

Least Connections sends traffic to the server with the fewest active connections.
Ideal for uneven request loads or long-lived sessions.
More dynamic than round-robin.
Requires connection tracking.
Helps improve response time in real-time.

Example:

Server A (2 conns), B (5 conns) → Next request goes to A

10. How does DNS load balancing work?

DNS load balancing rotates multiple IPs for a domain name.
Simple to set up but lacks real-time health checking.
Client-side caching may reduce effectiveness.
Often combined with other load balancing techniques.
Not ideal for dynamic failover.

Example:

DNS A records: www.example.com → 10.0.0.1, 10.0.0.2, 10.0.0.3