**Performance Checklist**

1. **Asynchronous Programming**:
   * Ensure all I/O-bound operations (e.g., database calls, file I/O) are asynchronous using async and await to prevent thread-blocking.

|  |  |
| --- | --- |
| **Without Async/Await (Blocking):** | **With Async/Await (Non-Blocking):** |
| [ApiController]  [Route("api/products")]  public class ProductsController : ControllerBase  {  private readonly ApplicationDbContext \_context;  public ProductsController(ApplicationDbContext context)  {  \_context = context;  }  [HttpGet]  public IActionResult GetProducts()  {  var products = \_context.Products.ToList(); // Synchronous database call (blocking)  return Ok(products);  }  }  \_context.Products.ToList() is **synchronous**, which means the thread will be blocked until the database operation completes | [ApiController]  [Route("api/products")]  public class ProductsController : ControllerBase  {  private readonly ApplicationDbContext \_context;  public ProductsController(ApplicationDbContext context)  {  \_context = context;  }  [HttpGet]  public async Task<IActionResult> GetProducts()  {  var products = await \_context.Products.ToListAsync(); // Asynchronous database call (non-blocking)  return Ok(products);  }  }  **Improved Throughput**: The API can handle more concurrent requests without running out of threads because it does not block on I/O-bound operations. |

**Efficient Database Access**:

Use **Entity Framework Core** with **lazy loading** and **eager loading** carefully, and prefer **no-tracking** queries when you don’t need to update entities

**Lazy Loading**

Lazy loading in EF Core automatically loads related entities when they are accessed for the first time. This can be convenient but can lead to performance issues, especially when navigating through many relationships.**Disadvantages of Lazy Loading:**

**N+1 Query Problem**: For each navigation property, EF Core will issue a separate database query. This can result in many unnecessary queries being executed, which increases load time and overhead.

**Eager Loading**

Eager loading allows you to load related entities at the same time as the main entity in a single query. This can improve performance when you know you will need related data.

|  |
| --- |
| public class Product  {  public int ProductId { get; set; }  public string Name { get; set; }  publi**c ICollection<Order> Orders { get; set; } //** Lazy-loaded navigation property  }  public class A**pplicationDbContext : DbContext**  {  public DbSet<Product> Products { get; set; }  public DbSet<Order> Orders { get; set; }  }  public async Task<IActionResult> GetProduct(int id)  {  var product = await \_context.Products.FindAsync(id);  var orders = product.Orders; // This triggers an additional query for orders.  return Ok(product);  } |
| **Eager Loading**  public async Task<IActionResult> GetProductWithOrders(int id)  {  var product = await \_context.Products  .Include(p => p.Orders) // Eager loading orders  .FirstOrDefaultAsync(p => p.ProductId == id);  return Ok(product);  } |

1. **Caching**:
   * Implement **caching** strategies (e.g., MemoryCache, Redis) to reduce repetitive calls to the database or external APIs.
   * Ensure caching is used for data that does not change frequently (e.g., configuration, static content).
2. **Compression**:
   * Enable HTTP response **compression** (e.g., GZIP) to reduce response payload sizes, particularly for large JSON responses.
3. **Database Indexing**:
   * Ensure that frequently queried database fields are properly indexed.
   * Analyze SQL queries to avoid unnecessary joins or subqueries.
4. **Optimize SQL Queries**:
   * Use **parameterized queries** to prevent SQL injection and ensure optimal query plans.
   * Avoid **N+1 query problems** (use eager loading where necessary).
5. **Reduce API Response Size**:
   * Return only the necessary data in API responses. Avoid over-fetching by using **pagination** and **filtering**.
   * Use **Projection** (select specific fields) instead of returning full entities.
6. **Minimize Middleware**:
   * Only include essential middleware in the pipeline to reduce overhead.
   * Ensure that custom middleware does not perform unnecessary work.
7. **Optimize Serialization/Deserialization**:
   * Use System.Text.Json for faster JSON serialization/deserialization over Newtonsoft.Json, unless you need advanced features.
8. **Connection Pooling**:
   * Use **connection pooling** for database connections to minimize the overhead of opening and closing connections.
9. **Minimize Use of Reflection**:
   * Reflection can be slow. Avoid it unless absolutely necessary.
10. **Service and Dependency Injection (DI) Optimization**:
    * Avoid long-lived services being injected into short-lived services.
    * Ensure that **Singletons** are not holding onto resources unnecessarily.
    * Ensure **scoped services** are properly disposed of by the DI container.
11. **Optimize Logging**:
    * Avoid excessive logging at high levels (e.g., debug or info) in production.
    * Use structured logging (e.g., Serilog) for better performance and insights.
12. **Parallelism**:
    * Ensure that tasks which can be run concurrently are properly parallelized using Task.WhenAll or Parallel.ForEach.
13. **Memory Usage**:
    * Avoid excessive memory consumption by monitoring object allocations and use of memory-heavy operations (e.g., large collections, strings).
14. **Load Testing**:
    * Use tools like **Apache JMeter** or **Visual Studio Load Testing** to identify and test performance bottlenecks.
15. **Thread Pool Management**:
    * Avoid blocking threads in web applications. Use asynchronous operations to prevent thread pool exhaustion.
16. **Avoid Large ViewModels**:
    * Keep ViewModel sizes minimal, and transfer only necessary data to the client.
17. **Timeouts for External Requests**:
    * Set proper timeout values for external API calls to avoid unnecessary waits.
18. **Scalability**:
    * Ensure that the API can scale horizontally by implementing stateless services and leveraging cloud-based scalability features (e.g., Azure scaling).

**Security Checklist**

1. **Authentication & Authorization**:
   * Use **JWT tokens** or **OAuth 2.0** for secure authentication.
   * Ensure that role-based access control (RBAC) or claims-based authorization is in place for different user access levels.
2. **Cross-Site Scripting (XSS)**:
   * Validate and sanitize user inputs, especially in HTML and JavaScript.
   * Use output encoding to mitigate XSS vulnerabilities.
3. **Cross-Site Request Forgery (CSRF)**:
   * Use anti-forgery tokens (via [ValidateAntiForgeryToken]) for all state-changing operations (e.g., POST, PUT, DELETE).
   * Ensure CSRF tokens are transmitted correctly in headers or cookies.
4. **Cross-Origin Resource Sharing (CORS)**:
   * Properly configure CORS to restrict access to trusted domains only.
   * Avoid AllowAnyOrigin in production environments.
5. **SQL Injection**:
   * Use **parameterized queries** or **ORMs** (e.g., Entity Framework) to protect against SQL injection.
   * Avoid raw SQL queries if possible.
6. **Sensitive Data Exposure**:
   * Always use **HTTPS** (SSL/TLS) for data in transit.
   * Encrypt sensitive data at rest and use secure algorithms like AES.
7. **Password Security**:
   * Store passwords securely using **bcrypt**, **PBKDF2**, or **Argon2**.
   * Implement **password complexity rules** and enforce multi-factor authentication (MFA) for sensitive actions.
8. **Logging Sensitive Information**:
   * Do not log sensitive information such as passwords, tokens, or credit card numbers.
   * Ensure logs are secured and accessible only by authorized personnel.
9. **Man-in-the-Middle (MITM) Protection**:
   * Ensure that the API is only accessible via **HTTPS** to prevent MITM attacks.
   * Enforce HTTP Strict Transport Security (HSTS) to prevent protocol downgrades.
10. **Session Management**:
    * Implement **short session lifetimes** for sensitive operations.
    * Use secure cookies with the **HttpOnly** and **SameSite** flags set.
11. **Certificate Management**:
    * Regularly rotate SSL/TLS certificates and avoid weak ciphers (e.g., RC4, DES).
    * Use **strong algorithms** such as **SHA256** for signing certificates.
12. **Input Validation**:
    * Always validate and sanitize user input to prevent **buffer overflows** and **injection attacks**.
    * Use whitelist validation whenever possible.
13. **File Uploads**:
    * Validate and sanitize file uploads, ensuring files are checked for dangerous extensions and file types.
    * Limit file sizes and ensure temporary files are securely deleted after use.
14. **API Rate Limiting**:
    * Implement **rate limiting** (e.g., via middleware) to protect the API from abuse and **denial-of-service (DoS)** attacks.
    * Use services like **Azure API Management** or custom middlewares to throttle requests.
15. **Security Headers**:
    * Implement **security headers** like:
      + Strict-Transport-Security (HSTS)
      + X-Content-Type-Options: nosniff
      + X-Frame-Options: DENY
      + Content-Security-Policy
      + X-XSS-Protection
      + Referrer-Policy
16. **Data Encryption**:
    * Ensure sensitive data is encrypted both in transit (HTTPS) and at rest.
    * Use **strong encryption algorithms** like AES-256 and RSA-2048.
17. **API Security Testing**:
    * Use automated security testing tools (e.g., **OWASP ZAP**, **Burp Suite**) to identify vulnerabilities.
18. **Exception Handling**:
    * Ensure **detailed error messages** are not exposed to end-users in production. Log them internally and use generic error messages for the client.
    * Use centralized exception handling with proper logging.
19. **Dependency Scanning**:
    * Regularly scan third-party dependencies for vulnerabilities using tools like **OWASP Dependency-Check** or **SonarQube**.
20. **Application Security Updates**:
    * Regularly update your .NET Core framework and dependencies to patch known security vulnerabilities.
21. **Privilege Escalation Protection**:
    * Ensure that sensitive operations are properly protected with role-based authorization.
    * Apply the **Principle of Least Privilege (PoLP)**.
22. **Ensure Secure Headers**:
    * Use proper headers like X-Content-Type-Options: nosniff and X-Frame-Options: DENY to mitigate content sniffing and clickjacking attacks.
23. **Security Logging**:
    * Ensure sensitive data is **never logged**.
    * Log authentication failures and unusual access patterns for auditing.
24. **OAuth/OpenID Connect Integration**:
    * Ensure **OAuth 2.0** or **OpenID Connect** is correctly implemented for external authentication (e.g., Google, Facebook).
25. **Token Expiry & Revocation**:
    * Ensure that **JWT tokens** or other authentication tokens have an **expiration time** and can be revoked if necessary.
26. **Service-to-Service Authentication**:
    * Use **mutual TLS** or **API keys** for service-to-service authentication in a microservices architecture.
27. **Nonces & CSRF Protection for APIs**:
    * Ensure anti-CSRF measures are in place for state-changing API requests.
28. **Network Segmentation & Firewalling**:
    * Ensure that the application server is behind a firewall and is not directly accessible from the internet (e.g., use a load balancer).
29. **Security Incident Response**:
    * Have a **Security Incident Response Plan** in place in case of a breach.
    * Regularly test and update the response plan.
30. **Data Minimization**:
    * Only collect and retain the minimum amount of sensitive data necessary to perform the required business operations.

** CORS Configuration:**

* **Check trusted origins in CORS settings.**
* **No AllowAnyOrigin in production.**

** API Versioning:**

* **Verify versioning strategy.**
* **API versions should be specified in URLs, headers, or query strings.**

** OWASP Security:**

* **Validate inputs to prevent SQL injection.**
* **Use secure authentication methods (e.g., JWT).**
* **Encrypt sensitive data.**

** XSS Protection:**

* **Sanitize user inputs.**
* **Ensure output encoding is used when rendering HTML.**

** Anti-Forgery Tokens:**

* **Include and validate anti-forgery tokens for state-changing actions.**

** Security:**

* **Implement secure authentication (JWT, OAuth2).**
* **Use HTTPS for secure data transmission.**

** Logging & Monitoring:**

* **Implement logging and monitoring solutions.**
* **Log key events and errors, but avoid logging sensitive information.**

**1. CORS (Cross-Origin Resource Sharing)**

What to check:

* Is CORS properly configured to allow trusted origins only?
* Avoid AllowAnyOrigin for production.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddCors(options =>  {  options.AddPolicy("AllowTrustedOrigins", builder =>  {  builder.WithOrigins("https://example.com", "https://trusted-site.com") // Trusted origins  .AllowAnyMethod()  .AllowAnyHeader();  });  });  }  public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {  app.UseCors("AllowTrustedOrigins"); // Apply CORS policy  } |

What to verify:

* Ensure only specific domains are allowed, not AllowAnyOrigin.

2. API Versioning

What to check:

* Is API versioning implemented?
* Are versioning strategies clear (e.g., URI, query string, or header-based)?

Example (API Versioning via URL):

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddApiVersioning(options =>  {  options.ReportApiVersions = true;  options.AssumeDefaultVersionWhenUnspecified = true;  options.DefaultApiVersion = ApiVersion.Default;  });  }  [ApiController]  [Route("api/v{version:apiVersion}/products")]  public class ProductsController : ControllerBase  {  [HttpGet]  public IActionResult Get() => Ok(new { Message = "Version 1" });  } |

What to verify:

* Ensure versions are explicitly included in routes (v1, v2, etc.).

3. OWASP Security Best Practices

What to check:

* Proper input validation (SQL injection prevention).
* Authentication and authorization.
* Sensitive data exposure.

Example (SQL Injection Prevention):

|  |
| --- |
| public IActionResult GetProducts(string category)  {  var products = \_context.Products  .Where(p => p.Category == category)  .ToList();  return Ok(products);  } |

* Avoid raw SQL queries that could expose the application to injection attacks.
* Use ORM (like Entity Framework) to parameterize queries and prevent injection.

What to verify:

* Inputs are validated and sanitized.
* Authentication uses JWT/OAuth for API access.

4. Cross-Site Scripting (XSS) Protection

What to check:

* Ensure user input is sanitized and encoded properly.
* Use output encoding to prevent malicious scripts from being injected into HTML.

Example (Output Encoding):

csharp

CopyEdit

@Html.Encode(Model.UserInput)

What to verify:

* Always sanitize inputs and escape data when rendering it in HTML, JavaScript, or CSS.

5. Anti-Forgery Token (CSRF Protection)

What to check:

* Are anti-forgery tokens included for every state-changing request (POST, PUT, DELETE)?

Example:

In ASP.NET Core MVC, use the [ValidateAntiForgeryToken] attribute:

|  |
| --- |
| [HttpPost]  [ValidateAntiForgeryToken]  public IActionResult CreateProduct(Product product)  {  // Code to create the product  return RedirectToAction("Index");  } |
| fetch('/api/products', {  method: 'POST',  headers: {  'Content-Type': 'application/json',  'RequestVerificationToken': document.querySelector('input[name="\_\_RequestVerificationToken"]').value  },  body: JSON.stringify(productData)  }); |

What to verify:

* Anti-forgery tokens are included in POST requests.
* The token is validated on the server-side.

6. Security

What to check:

* Are JWT tokens or OAuth2 used for authentication?
* Are sensitive endpoints protected?

Example (JWT Authentication):

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  .AddJwtBearer(options =>  {  options.Authority = "https://your-auth-server";  options.Audience = "your-api";  });  }  public void Configure(IApplicationBuilder app)  {  app.UseAuthentication(); // Enable Authentication  app.UseAuthorization(); // Enable Authorization  } |

What to verify:

* Sensitive data is not exposed via unsecured channels.
* Use HTTPS everywhere.
* Implement role-based authorization (e.g., [Authorize(Roles = "Admin")]).

7. Logging and Monitoring

What to check:

* Are logging mechanisms implemented (e.g., Serilog, NLog)?
* Are logs written for key events (authentication failures, errors, etc.)?

Example (Serilog Setup):

csharp

CopyEdit

public void ConfigureServices(IServiceCollection services)

{

services.AddLogging(logging =>

{

logging.AddSerilog();

});

}

public void Configure(IApplicationBuilder app)

{

app.UseSerilogRequestLogging(); // Log HTTP request details

}

What to verify:

* Logs include appropriate levels (e.g., Information, Warning, Error).
* Sensitive information (e.g., passwords) is not logged.
* Enable application insights or similar tools for monitoring in production.

Summary Checklist for Junior Developer:

1. CORS Configuration:
   * Check trusted origins in CORS settings.
   * No AllowAnyOrigin in production.
2. API Versioning:
   * Verify versioning strategy.
   * API versions should be specified in URLs, headers, or query strings.
3. OWASP Security:
   * Validate inputs to prevent SQL injection.
   * Use secure authentication methods (e.g., JWT).
   * Encrypt sensitive data.
4. XSS Protection:
   * Sanitize user inputs.
   * Ensure output encoding is used when rendering HTML.
5. Anti-Forgery Tokens:
   * Include and validate anti-forgery tokens for state-changing actions.
6. Security:
   * Implement secure authentication (JWT, OAuth2).
   * Use HTTPS for secure data transmission.
7. Logging & Monitoring:
   * Implement logging and monitoring solutions.
   * Log key events and errors, but avoid logging sensitive information.