# 50 Advanced Entity Framework Core Interview Questions and Answers for Senior Developer Roles

## Introduction

Entity Framework Core (EF Core) is a highly flexible, powerful Object-Relational Mapper (ORM) for .NET developers. As the backbone for data access in many modern ASP.NET Core applications, EF Core's extensible architecture supports diverse database providers, complex LINQ queries, migrations, advanced change tracking, performance optimization, concurrency control, and integration with the latest patterns such as dependency injection and domain-driven design.

For senior developer positions at leading companies, interviews often focus on not only deep EF Core expertise but also on real-world scenarios, performance benchmarking, maintainability, debugging, and cross-platform considerations. This resource compiles **50 in-depth, scenario-driven interview questions**-each paired with comprehensive answers, explanations, and C# code snippets-covering the most frequently encountered and advanced aspects of EF Core as identified in recent web sources, expert blogs, official documentation, and technical articles.

## Table of Questions and Section Structure

Questions are organized into themed sections for focused learning:

* **Performance Optimization**
* **Migrations and Versioning**
* **LINQ Queries and Projections**
* **Change Tracking**
* **Concurrency Control**
* **Integration with ASP.NET Core**
* **Best Practices and Patterns**
* **Bulk Operations and Third-party Libraries**
* **Caching and Second-Level Cache**
* **Debugging and Logging**
* **Testing Strategies**
* **Security and Data Validation**
* **EF Core vs Dapper/Raw SQL**
* **Tools and Profilers**
* **Database Providers and Cross-Platform**
* **Extensibility and Custom Conventions**
* **Release Notes and Breaking Changes**
* **Real-world Scenarios and Company Questions**

Each question includes a concise answer, a detailed explanation, and, where appropriate, relevant code samples.

## Performance Optimization

### 1. **What is**

**Answer:**  
AsNoTracking() is a LINQ extension method in EF Core that disables change tracking for the retrieved entities. When you apply AsNoTracking(), EF Core does not keep entities in the change tracker, leading to better performance and less memory usage for read-only queries.

**Explanation:**  
By default, EF Core tracks all retrieved entities, which is necessary when you intend to update them later. However, for queries where you only need to read data and have no intention of modifying entities, the tracking feature is unnecessary overhead. Using AsNoTracking() can make such queries 3-4 times faster, especially with large result sets. It's critical to avoid AsNoTracking() if updates will follow, as changes will not be recognized by SaveChanges()12.

**Code Snippet:**

|  |
| --- |
| using var context = new MyDbContext(); var products = await context.Products  .AsNoTracking()  .Where(p => p.IsActive)  .ToListAsync(); |

### 2. **How do you address the N+1 query problem in EF Core and why is it important?**

**Answer:**  
The N+1 query problem occurs when an application issues one query for the root entities and then one query for each related entity (N additional queries). EF Core can avoid this inefficiency through *eager loading* with .Include() or by using explicit joins or projections.

**Explanation:**  
Lazy loading can cause numerous database round-trips, which degrades performance significantly when handling collections. Eager loading retrieves all necessary data in a single query using .Include() or .ThenInclude(). For more complex scenarios, projections into DTOs or explicit loading can further optimize query plans. Tools like MiniProfiler or EF Core logging can help detect N+1 issues2.

**Code Snippet:**

|  |
| --- |
| // Eager loading var blogs = context.Blogs  .Include(b => b.Posts)  .Include(b => b.Owner)  .ToList(); |

### 3. **What is query splitting, and when should you use**

**Answer:**  
Query splitting (.AsSplitQuery()) tells EF Core to generate multiple SQL queries-one for each included collection-when processing a LINQ query with nested includes. This prevents 'Cartesian explosion' and minimizes duplicate data in the result set, especially for deeply nested relationships.

**Explanation:**  
Without query splitting, EF Core joins all included tables in one query, which can result in large, slow-running statements and excessive duplication of parent data for every related child. With .AsSplitQuery(), each include fetches data with a separate query, reducing memory pressure and improving performance in complex graph scenarios. You can configure all queries to use split behavior with UseQuerySplittingBehavior(QuerySplittingBehavior.SplitQuery) in your DbContext configuration3.

**Code Snippet:**

|  |
| --- |
| var order = dbContext.Orders  .Include(o => o.LineItems)  .ThenInclude(li => li.Dimensions)  .AsSplitQuery()  .First(o => o.Id == orderId); |

### 4. **How can you limit the data returned and mapped by EF Core to maximize performance?**

**Answer:**  
Project only the required fields using LINQ .Select() clauses to avoid loading unnecessary columns or navigation properties.

**Explanation:**  
When you use entity queries without projection, EF Core returns all columns mapped to the entity, which can be wasteful if only a few columns are needed. By projecting into DTOs or anonymous types, you reduce both the amount of data fetched and the memory consumed by the application. This practice is particularly beneficial for API endpoints or large tables. Libraries like AutoMapper can simplify projection mapping4.

**Code Snippet:**

|  |
| --- |
| var postSummaries = context.Posts  .Select(p => new PostSummaryDto { Id = p.Id, Title = p.Title })  .ToList(); |

### 5. **What is a Compiled Query in EF Core, and when would you use it?**

**Answer:**  
A compiled query is a LINQ query that’s compiled only once and then executed repeatedly with different parameter values, reducing query parsing and translation overhead.

**Explanation:**  
Compiled queries are especially beneficial in scenarios where the same query structure is executed multiple times-such as high-traffic web APIs or background processing tasks. EF Core allows precompiling queries via EF.CompileQuery() or EF.CompileAsyncQuery(). While modern EF Core versions are highly optimized, compiled queries can still reduce CPU cycles for hot paths.

**Code Snippet:**

|  |
| --- |
| private static readonly Func<MyDbContext, int, Post>  \_getPostById = EF.CompileQuery(  (MyDbContext ctx, int id) => ctx.Posts.FirstOrDefault(p => p.Id == id)  );  var post = \_getPostById(context, postId); |

### 6. **How do you leverage database indexing through EF Core to improve query performance?**

**Answer:**  
You define indexes using the Fluent API in OnModelCreating with .HasIndex(), indicating columns that should be indexed for frequent lookups or uniqueness enforcement.

**Explanation:**  
Indexing improves performance of queries involving filtering, sorting, and joining on indexed columns. You can specify single or composite indexes, including uniqueness. Indexes should be thoughtfully selected based on query patterns identified via database query analysis, as excessive indexing can slow down writes.

**Code Snippet:**

|  |
| --- |
| modelBuilder.Entity<User>()  .HasIndex(u => u.Email)  .IsUnique(); |

### 7. **Describe the role of EF Core’s DbContext pooling and how it enhances performance in ASP.NET Core applications.**

**Answer:**  
DbContext pooling enables the reuse of DbContext instances, reducing garbage collection pressure and object construction overhead on each request. Instead of creating a new instance for every scope, EF Core retrieves an existing instance from the pool.

**Explanation:**  
Pooling is highly effective in high-throughput web APIs and microservices. However, pooled contexts must not hold state across requests. Proper scoping guarantees thread safety and removes risk of cross-request contamination.5

**Code Snippet:**

|  |
| --- |
| builder.Services.AddDbContextPool<MyDbContext>(  options => options.UseSqlServer(Configuration.GetConnectionString("Default"))); |

### 8. **Compare EF Core’s query performance versus Dapper or raw SQL in high-volume read scenarios.**

**Answer:**  
Dapper (a micro-ORM) outperforms EF Core on large data reads by skipping entity tracking, relationship resolution, and query translation. While EF Core's AsNoTracking() narrows the gap, Dapper’s manual SQL execution is still faster and uses less memory.

**Explanation:**  
EF Core is well-suited for maintainability and complex domain models but adds abstraction overhead, especially when tracking is enabled. For analytics, dashboards, or pure reporting, Dapper or direct SqlDataReader are preferable. Consider hybrid architectures using EF Core for core domain CRUD and Dapper for heavy reporting.6

**Code Snippet (Dapper):**

|  |
| --- |
| using (var conn = new SqlConnection(connectionString)) {  var posts = conn.Query<Post>("SELECT Id, Title FROM Posts WHERE IsPublished = 1"); } |

## Migrations and Versioning

### 9. **How does EF Core migration work, and what are the typical migration workflows?**

**Answer:**  
EF Core migrations maintain schema versioning by representing database changes as code. A migration consists of Up() (apply changes) and Down() (rollback) methods. Common lifecycle: make model changes → add migration → apply/update database.

**Explanation:**  
Migrations provide a robust, repeatable way to evolve your schema in sync with domain classes. Use commands like Add-Migration to scaffold code, and Update-Database to apply changes. In collaborative or CI/CD pipelines, SQL scripts generated from migrations are preferred for production. Always aim for small, focused, and descriptive migration steps.78

**Code Snippet:**

|  |
| --- |
| dotnet ef migrations add AddProductsTable dotnet ef database update |

**Migration Class Example:**

|  |
| --- |
| public partial class AddEmployeesTable : Migration {  protected override void Up(MigrationBuilder builder)  {  builder.CreateTable(  name: "Employees",  columns: table => new  {  EmployeeId = table.Column<int>(nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  FirstName = table.Column<string>(maxLength: 100, nullable: false),  // ... other columns  },  constraints: table => { table.PrimaryKey("PK\_EmployeeId", x => x.EmployeeId); });  }   protected override void Down(MigrationBuilder builder)  {  builder.DropTable("Employees");  } } |

### 10. **Explain the difference between automatic and code-based migrations in EF Core.**

**Answer:**

* **Automatic migrations:** Schema changes are applied implicitly by the framework; little to no developer intervention.
* **Code-based (explicit) migrations:** Developers manually scaffold migration files using tooling, offering fine-grained control over changes, custom SQL, and upgrade/downgrade processes.

**Explanation:**  
Automatic migrations can be quick but lack transparency and control, risking data loss or production mishaps. Code-based migrations allow versioning, reviews, script generation, and safer operations, which are preferable in team and production settings.9

### 11. **Describe best practices for seeding data in EF Core migrations.**

**Answer:**  
Use the Fluent API's HasData() within OnModelCreating() to specify initial static or reference data. For more complex, transactional seeding (possibly conditional or dependent on runtime state), implement logic during startup after applying migrations.

**Explanation:**  
Always separate static reference data (roles, categories, etc.) from user or environment-specific data. Do not seed sensitive or environmental data in migrations. Prefer database initialization logic in application startup for dynamic/environment-dependent scenarios. Seed data is automatically inserted or updated during migrations if changed.10

**Code Snippet:**

|  |
| --- |
| modelBuilder.Entity<Category>().HasData(  new Category { CategoryId = 1, CategoryName = "Electronics" },  new Category { CategoryId = 2, CategoryName = "Books" } ); |

### 12. **How would you handle breaking changes and rollbacks during EF Core migrations in a production environment?**

**Answer:**

* **Review generated migration scripts carefully before applying to production.**
* **Use idempotent SQL scripts for production deployment** to handle unknown database states safely.
* **Backup the database before applying destructive changes (column/type drops).**
* **Keep destructive changes (like column drops) in a separate migration after data has been moved or copied.**
* **Rollback with**

**Explanation:**  
In production, direct migration scripts provide more transparency and the opportunity for code reviews and manual validation, mitigating accidental data loss or downtime. Automating migrations in a CI/CD pipeline should be coupled with robust tests and backups. Rollbacks may not always be straightforward; consider the feasibility of reversals in your migration design.9

### 13. **What are idempotent SQL scripts in EF Core migrations and why are they important for deployment?**

**Answer:**  
An idempotent SQL script adapts its actions based on the current state of the database-applying changes only if necessary. EF Core can generate such scripts using the dotnet ef migrations script --idempotent command.

**Explanation:**  
Idempotent scripts allow the same script to be safely run multiple times or on databases at different migration stages, critical for environments where the schema may not be synchronized or has unknown state (e.g., blue-green deployments or multi-tenant systems).7

**Command Example:**

|  |
| --- |
| dotnet ef migrations script --idempotent |

### 14. **How do migrations differ between PostgreSQL and SQL Server using EF Core, particularly regarding timestamps and timezones?**

**Answer:**  
PostgreSQL and SQL Server handle timestamp/timezone types differently. For PostgreSQL (with the Npgsql provider), EF Core 6+ maps .NET DateTime (with Kind=UTC) to PostgreSQL's timestamptz (timestamp with time zone) and applies Kind.Local/Unspecified to timestamp without time zone. A migration may alter timestamp columns and require explicit timezone declarations for correct conversion.

**Explanation:**  
When migrating or upgrading versions, carefully handle timestamp conversions and literal seeding with explicit UTC indicators to avoid losing accurate datetime data. Use the recommended Npgsql approach in migration scripts: migrationBuilder.Sql("SET TimeZone='UTC';"); for correct conversions.11

## LINQ Queries and Projection

### 15. **How do projection queries optimize performance in EF Core and what are your options for defining projections?**

**Answer:**  
Projection queries, crafted via the .Select() method in LINQ, retrieve only specified columns into either anonymous or explicit DTO types, reducing transfer and memory overhead.

**Explanation:**  
Full entity queries fetch all mapped columns, including navigation properties, whereas projections limit transfer to required fields. This technique is vital for high-performance APIs and UIs. Define projections as anonymous types for local consumption or concrete DTOs for API responses and mapping layers. Libraries such as AutoMapper’s ProjectTo<> can automate DTO mapping at the query level.4

**Anonymous Projection Example:**

|  |
| --- |
| var customers = dbContext.Customers  .Select(c => new  {  c.FirstName,  c.LastName,  c.Email  })  .ToList(); |

**DTO Projection Example:**

|  |
| --- |
| public class CustomerDto {  public string FirstName { get; set; }  public string LastName { get; set; } }  var dtos = dbContext.Customers  .Select(c => new CustomerDto  {  FirstName = c.FirstName,  LastName = c.LastName  })  .ToList(); |

### 16. **How do you implement dynamic filtering with LINQ and EF Core, such as for advanced search pages?**

**Answer:**  
Build dynamic LINQ expressions using either Expression Trees or helper libraries so that filters can be composed at runtime.

**Explanation:**  
Dynamic querying is essential for advanced filtering scenarios. You can leverage System.Linq.Expressions to compose queries based on user-specified filters, or use libraries like Entity Framework Dynamic LINQ, or roll custom builders. This approach is invaluable for admin dashboards, reporting, and user-defined searches.12

**Code Snippet Example:**

|  |
| --- |
| // See DynamicQueryBuilderDemo for full implementation var filters = new List<FilterCondition> {  new FilterCondition { Field = "Name", Operator = FilterOperator.Contains, Value = "Laptop" },  new FilterCondition { Field = "Price", Operator = FilterOperator.GreaterThan, Value = 1000M } };  var query = DynamicQueryBuilder.Build(products.AsQueryable(), filters); var results = query.ToList(); |

### 17. **How can you map SQL functions, such as JSON functions or database-specific scalar functions, in EF Core queries?**

**Answer:**  
You can map database scalar functions using the [DbFunction] attribute, then use the mapped function in your LINQ queries.

**Explanation:**  
For advanced scenarios (e.g., filtering on a JSON column in SQL Server), write a static .NET method decorated with [DbFunction], and optionally configure it in OnModelCreating. EF Core will use this method’s SQL translation in query generation.13

**Code Snippet:**

|  |
| --- |
| public static class MyDbFunctions {  [DbFunction("JSON\_VALUE", "")]  public static string JsonValue(string source, string path) => throw new NotSupportedException(); }  // Usage in query var query = db.Set<Setting>()  .Where(s => Convert.ToInt32(MyDbFunctions.JsonValue(s.Value, "lax $.Name")) > 1); |

### 18. **What are the pros and cons of explicit joins versus navigational property queries in EF Core LINQ?**

**Answer:**  
Navigational property queries allow clear, maintainable code and fully benefit from EF’s change tracking and relationship management. Explicit joins offer more flexible query shapes and control, sometimes resulting in more efficient raw SQL, but at the cost of increased complexity and more manual mapping.

**Explanation:**  
Use navigational properties for regular domain traversal and relationship loading. Use explicit joins for aggregations, complex filtering, or when optimizing performance or limiting the result set. Navigational properties are subject to tracking-related overhead (unless projected or AsNoTracking).

## Change Tracking

### 19. **Explain the different entity states in EF Core change tracking.**

**Answer:**  
EF Core tracks entities in one of the following states:

* **Unchanged:** No modifications since retrieval.
* **Added:** New entity to insert.
* **Modified:** Properties have changed since loading.
* **Deleted:** Entity marked for removal.
* **Detached:** Not tracked by the DbContext.

**Explanation:**  
Entity state changes are managed by the ChangeTracker and represented by EntityState. State transitions dictate what SQL commands (INSERT, UPDATE, DELETE) are generated on SaveChanges(). Understanding these states, and how and when they change, is vital for accurate data persistence and for scenarios involving disconnected contexts (e.g., web APIs).142

### 20. **How can you globally configure EF Core to disable change tracking by default on all queries within a DbContext?**

**Answer:**  
Configure the context’s query tracking behavior using:

|  |
| --- |
| options.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking) |

inside your DbContext configuration.

**Explanation:**  
This approach prevents unintentional tracking, which can lead to performance improvements for read-intensive applications and API endpoints that do not require entity updates.15

### 21. **How do you perform explicit change tracking or handle disconnected entities (e.g., in web APIs)?**

**Answer:**  
You attach the entity to the DbContext instance and manually set its state:

**Code Snippet:**

|  |
| --- |
| context.Entry(entity).State = EntityState.Modified; |

**Explanation:**  
When changes are made outside the scope of the original context (e.g., in a web API, where entities are reconstructed from incoming JSON), you need to attach the entity and specify which state corresponds to the update scenario, so that SaveChanges() produces the right SQL command.14

### 22. **What is**

**Answer:**  
SaveChanges(false) submits changes to the database but maintains their tracked state. AcceptAllChanges() must be called later to mark them as “accepted.”

**Explanation:**  
This technique is useful in distributed transaction scenarios where multiple contexts participate in a single transaction. It allows you to coordinate commits/rollbacks across different contexts and only finalize state after all succeed.2

**Code Example:**

|  |
| --- |
| using (var scope = new TransactionScope()) {  context1.SaveChanges(false);  context2.SaveChanges(false);  scope.Complete();  context1.AcceptAllChanges();  context2.AcceptAllChanges(); } |

## Concurrency Control Strategies

### 23. **How does optimistic concurrency work in EF Core, and how do you implement it?**

**Answer:**  
EF Core supports optimistic concurrency by using a concurrency token property (typically a rowversion column or a property decorated with [Timestamp]). On SaveChanges(), EF includes the token’s original value in the WHERE clause of UPDATE/DELETE. If the row has changed since reading, the command affects zero rows and EF throws DbUpdateConcurrencyException.

**Implementation:**

* Add a [Timestamp] property.
* Handle DbUpdateConcurrencyException.

**Explanation:**  
This system allows multiple users to work with data, only reconciling conflicts if/when updates collide. Approaches to resolve conflicts include “Store wins” (reload), “Client wins” (overwrite), or prompting the user to address mismatches.162

**Code Snippet:**

|  |
| --- |
| public class Department {  public int DepartmentID { get; set; }  [Timestamp]  public byte[] RowVersion { get; set; } } |

### 24. **What’s the difference between optimistic and pessimistic concurrency in EF Core?**

**Answer:**

* **Optimistic concurrency (EF Core’s default):** No locks are applied. All users can read data, but updates may fail if the data has changed since loading (conflict is detected at SaveChanges()).
* **Pessimistic concurrency:** Locks are applied during read, blocking others until the lock is released. EF Core does not natively support pessimistic locking.

**Explanation:**  
Optimistic concurrency is scalable and suited to stateless, high-volume web apps, while pessimistic locking is more appropriate for niche, stateful, or two-tier scenarios, often using custom stored procedures in EF.2

### 25. **How would you handle concurrency conflicts in an ASP.NET Core MVC controller action?**

**Answer:**  
Catch DbUpdateConcurrencyException, reload current values, and prompt the user to reapply changes (the “store-wins” approach). Compare client and database values and update the RowVersion accordingly.

**Code Snippet Example:**

|  |
| --- |
| [HttpPost] public async Task<IActionResult> Edit(int id, byte[] rowVersion) {  var dept = await \_context.Departments.FirstOrDefaultAsync(d => d.Id == id);  \_context.Entry(dept).Property("RowVersion").OriginalValue = rowVersion;  try  {  await \_context.SaveChangesAsync();  return RedirectToAction("Index");  }  catch (DbUpdateConcurrencyException ex)  {  var dbValues = ex.Entries.Single().GetDatabaseValues();  // Compare dbValues with clientValues  ModelState.AddModelError("Name", $"Current value: {((Department)dbValues)["Name"]}");  // ... other custom logic  }  return View(dept); } |

16

### 26. **How can you customize concurrency tokens using the Fluent API instead of data annotations?**

**Answer:**  
Use the IsRowVersion() method in OnModelCreating:

|  |
| --- |
| modelBuilder.Entity<Department>()  .Property(p => p.RowVersion)  .IsRowVersion(); |

16

### 27. **Give two examples of conflict resolution strategies when handling optimistic concurrency exceptions.**

**Answer:**

* **Store Wins:** Database values are kept; client is prompted to reload and retry.
* **Client Wins:** Client’s values overwrite those in the store.

**Explanation:**  
Choose the strategy according to application requirements for data consistency vs. user convenience.16

## Integration with ASP.NET Core

### 28. **How do you register and use EF Core's DbContext in ASP.NET Core Dependency Injection?**

**Answer:**  
Use the AddDbContext<T> or AddDbContextPool<T> extension methods in the Program.cs or Startup.cs file.

**Code Snippet:**

|  |
| --- |
| builder.Services.AddDbContext<MyDbContext>(options =>  options.UseSqlServer(Configuration.GetConnectionString("Default"))); |

8

### 29. **Describe how you would use an in-memory database in EF Core for development or testing scenarios.**

**Answer:**  
In-memory databases are set up with .UseInMemoryDatabase("DbName") and are suitable for unit tests, prototypes, and quick experimentation.

**Explanation:**  
This approach is ideal for unit/integration tests and rapid development without a real database. However, the in-memory provider does not support raw SQL or full feature parity with production databases.1718

**Code Snippet:**

|  |
| --- |
| services.AddDbContext<MyDbContext>(options =>  options.UseInMemoryDatabase("TestDb")); |

### 30. **How do you inject additional dependencies into your custom services or, in advanced cases, entity objects when using EF Core?**

**Answer:**  
You use standard constructor injection in services. For advanced scenarios (e.g., domain-driven design), EF Core 7+ supports interceptors such as IMaterializationInterceptor to inject dependencies into entities as they are loaded from the database.

**Example:**  
Implement IMaterializationInterceptor and assign dependencies after entity instantiation within its InitializedInstance method.19

## Best Practices and Patterns

### 31. **What is the Repository pattern and how do you implement it with EF Core? When should it be avoided?**

**Answer:**  
The Repository pattern abstracts data access behind interfaces, decoupling business logic and persistence. Implement a repository interface and a concrete class using EF Core’s DbContext.

**Code Snippet:**

|  |
| --- |
| public interface IEmployeeRepository {  Task<IEnumerable<Employee>> GetAllAsync();  // Other CRUD signatures }  public class EmployeeRepository : IEmployeeRepository {  private readonly MyDbContext \_context;   public EmployeeRepository(MyDbContext context) => \_context = context;   public async Task<IEnumerable<Employee>> GetAllAsync()  => await \_context.Employees.ToListAsync(); } |

**Explanation:**  
Use the Repository pattern for abstraction, testability, or when supporting multiple backends. Avoid when all logic is simple CRUD, as DbContext itself already acts as a unit of work and repository. Abstracting DbContext for trivial code can be unnecessary overhead.20

### 32. **What are POCO classes, and why are they important in EF Core?**

**Answer:**  
POCO (Plain Old CLR Object) classes are simple C# classes with no dependence on EF Core or base classes. They allow separation of domain logic from persistence framework details.

**Explanation:**  
They foster maintainability, testability, and architecture adherence. EF Core fully supports POCOs for all entity types.2

### 33. **How do you enforce validation logic in EF Core models?**

**Answer:**

* **Data annotations:** Apply attributes like [Required], [StringLength], [Range].
* **Fluent validation:** Use FluentValidation or implement IValidatableObject.

**Code Snippet:**

|  |
| --- |
| public class Product {  [Required]  [StringLength(50, MinimumLength = 2)]  public string Name { get; set; }  [Range(0, 100)]  public int Stock { get; set; } } |

21

### 34. **How do you implement and share custom conventions in EF Core?**

**Answer:**  
Use data annotations, Fluent API, or by writing custom convention classes. For global conventions (e.g., all string properties must be required and max 255 chars), you can set up conventions in OnModelCreating.

**Explanation:**  
Global conventions make maintenance and consistency easier, but should be used judiciously. EF Core 5+ has improved convention APIs for such scenarios.10

## Bulk Operations and Third-party Libraries

### 35. **How do you insert, update, or delete large numbers of records efficiently in EF Core?**

**Answer:**  
Standard EF Core operations are inefficient for batches. Use libraries such as EFCore.BulkExtensions or Z.EntityFramework.Plus for batch operations.

**Code Snippet:**

|  |
| --- |
| using EFCore.BulkExtensions; // Insert context.BulkInsert(entities); // Update context.BulkUpdate(entities); // Delete context.BulkDelete(entities); |

22

**Explanation:**  
Batching operations minimizes round trips and minimizes overhead in the DbContext change tracker. Bulk extensions generate optimized SQL statements.

### 36. **What issues might you encounter when using bulk extension libraries with EF Core?**

**Answer:**

* Slower-than-expected performance if database/network is the true bottleneck.
* Incomplete feature parity across providers.
* Transactions and special handling for output (e.g., database-generated IDs may require manual mapping).

**Explanation:**  
For best results, test with datasets, tweak configuration, and, if necessary, compare with provider-specific alternatives (SqlBulkCopy in SQL Server).22

## Caching and Second-Level Cache

### 37. **How does second-level caching work in EF Core, and what libraries are available?**

**Answer:**  
Second-level caching stores the results of queries, so repeat queries fetch data from memory or distributed cache instead of the database. Libraries such as <https://github.com/VahidN/EFCoreSecondLevelCacheInterceptor> enable this feature.

**Code Snippet to Enable:**

|  |
| --- |
| services.AddEFSecondLevelCache(options =>  options.UseMemoryCacheProvider().ConfigureLogging(true) );  services.AddDbContextPool<ApplicationDbContext>(  (sp, optionsBuilder) => optionsBuilder  .UseSqlServer(connectionString)  .AddInterceptors(sp.GetRequiredService<SecondLevelCacheInterceptor>()) );  // Usage: var posts = context.Posts  .Where(...).Cacheable().ToList(); |

23

### 38. **Explain automatic and manual cache invalidation in EF Core second-level caching.**

**Answer:**  
The cache provider auto-invalidates cached entries when a call to SaveChanges() or SaveChangesAsync() occurs. For operations bypassing EF's interception mechanisms (e.g., ExecuteUpdate), you must manually clear or invalidate relevant cache entries via the provider’s API.

**Code Snippet:**

|  |
| --- |
| \_cacheServiceProvider.ClearAllCachedEntries(); |

23

## Debugging and Logging

### 39. **How can you log SQL commands generated by EF Core?**

**Answer:**

* Use the ToQueryString() method on a query for ad-hoc inspection.
* Use LogTo() when configuring DbContext for global logging.
* Set up logging providers like Microsoft.Extensions.Logging.Console.

**Code Snippet Examples:**

|  |
| --- |
| var sql = query.ToQueryString();  optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information); |

24

### 40. **How do you analyze EF Core performance and memory allocations using BenchmarkDotNet?**

**Answer:**  
Use [MemoryDiagnoser] and BenchmarkRunner.Run() on benchmark classes to collect performance metrics. Open .diagsession files in Visual Studio for deep analysis.

**Code Snippet:**

|  |
| --- |
| [MemoryDiagnoser] public class EfBenchmark {  // Benchmark methods... } // In Main: BenchmarkRunner.Run<EfBenchmark>(); |

25

## Testing Strategies

### 41. **Describe approaches for unit and integration testing code dependent on EF Core.**

**Answer:**

* **In-memory provider:** Useful for simple, fast tests but can diverge in behavior from real databases.
* **SQLite in-memory mode:** Better parity for relational behavior but lacks full compatibility.
* **Mocking DbContext:** Useful for mocking individual behaviors, but complex for query logic; recommended to mock only repositories.

**Explanation:**  
Test most business logic with real or integrated test databases for best coverage. Use repository abstraction if you need data access to be fully unit testable.2018

### 42. **How do you mock DbSet and DbContext for testing in EF Core?**

**Answer:**  
Use frameworks like Moq and Moq.EntityFrameworkCore or MockQueryable to mock DbSet behavior for specific query and command scenarios.

**Snippet:**

|  |
| --- |
| var dbSetMock = new Mock<DbSet<Employee>>(); dbSetMock.As<IQueryable<Employee>>().Setup(m => ...);  var dbContextMock = new Mock<MyDbContext>(); dbContextMock.Setup(c => c.Employees).Returns(dbSetMock.Object); |

18

## Security and Data Validation

### 43. **Discuss approaches to securely validate input data in EF Core models, both via annotations and Fluent API.**

**Answer:**

* **Annotations:** [Required], [StringLength], [Range], etc.
* **FluentValidation or IValidatableObject:** For decoupled, reusable validation logic.

**Best Practice:**  
Validate incoming DTOs in controllers, then map only valid input to entity models.21

### 44. **How do you enforce unique constraints and foreign keys using both data annotations and Fluent API in EF Core?**

**Answer:**

* [Index(nameof(Email), IsUnique = true)] (EF Core 5+)
* .HasIndex(e => e.Email).IsUnique()
* .HasOne(...).WithMany().HasForeignKey(...)

**Explanation:**  
Proper constraints enforce data integrity at the database level in addition to application checks.8

## EF Core vs Dapper and Raw SQL

### 45. **List the trade-offs between EF Core and Dapper for .NET data access.**

|  |  |  |
| --- | --- | --- |
| Feature | Dapper | EF Core |
| Performance | Higher on simple queries | Slower due to abstraction |
| Change tracking | Manual | Automatic |
| Migrations | No | Built-in support |
| Bulk Operations | Manual (SqlBulkCopy) | Extensions available |
| Transaction handling | Manual | Built-in |
| Query flexibility | Manual SQL | LINQ & SQL |
| Relationship management | Manual | Automated |
| Maintainability | Harder | Higher |

**Recommendations:**

* Use Dapper for high-performance, “read-heavy” microservices or dashboards.
* EF Core for maintainability, domain complexity, and business workflows.6

## Tools and Profilers

### 46. **Which tools and profilers can help you diagnose EF Core performance issues?**

**Answer:**

* **MiniProfiler**-HTTP and SQL profiling, detects N+1.
* **Visual Studio Profiler**-CPU, memory, allocation analysis.
* **BenchmarkDotNet**-Code-level performance testing.
* **EF Core’s own logging**-ToQueryString(), LogTo().
* **Database-level profiling**-SQL Server Profiler, pgAdmin for PostgreSQL.

**Explanation:**  
These tools help identify inefficient queries, track memory and CPU usage, measure allocation, and optimize against real workloads.2

## Database Providers and Cross-Platform

### 47. **Describe EF Core’s cross-platform database support and major providers’ maturity.**

**Answer:**

* **SQL Server:** First-class support, most robust feature list.
* **PostgreSQL:** Npgsql is robust, with support for advanced PostgreSQL features, custom types, and timestamp handling.
* **MySQL:** Pomelo.EntityFrameworkCore.MySql and Oracle's MySQL provider are available but less commonly used.
* **SQLite:** Embedded, great for desktop/mobile and testing.
* **Others:** CosmosDB, Oracle, DB2, MongoDB (community/extensions).

**Explanation:**  
Provider maturity affects feature support; PostgreSQL and SQL Server are best-maintained for EF Core 6+.2611

## Extensibility and Custom Conventions

### 48. **How do you create and register a custom convention in EF Core?**

**Answer:**  
Override OnModelCreating to iterate and configure model properties/entities. For advanced scenarios, define classes implementing conventions and register them if supported.

**Example:**

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder) {  foreach (var entity in modelBuilder.Model.GetEntityTypes())  {  foreach (var property in entity.GetProperties().Where(p => p.ClrType == typeof(string)))  {  property.SetIsUnicode(false);  property.SetMaxLength(255);  }  } } |

10

## Release Notes and Breaking Changes

### 49. **Describe a high-impact breaking change between EF Core 5 and 6, and a mitigation strategy.**

**Example:**  
EF Core 6 disallows nested optional owned dependent entities sharing the same table with no required properties. Mitigation options: make dependents required, map to own table, or ensure at least one required property.

**Explanation:**  
Breaking changes may surface during upgrades-read release notes carefully, run tests, and choose appropriate mitigations for production upgrade paths.27

## Real-world Scenarios and Company-specific Questions

### 50. **Describe a real-world scenario migrating from EF6 to EF Core. What challenges must you address?**

**Answer:**

* **Analyze feature parity:** EF6 features like lazy loading, in-memory data seeding, and certain stored procedure mappings may not be identical in EF Core.
* **Update data access code:** Rewrite parts of the repository/unit-of-work layers if direct context access is used in controllers.
* **Handle breaking changes:** Conventions, migration approaches, and provider-specific features differ.
* **Test comprehensively:** Validate with integration tests against the production database provider.
* **Leverage new features:** Benefit from .NET cross-platform support, performance improvements, and migration scripts.
* **Address third-party dependencies:** Replace EF6 extensions and interceptors with EF Core equivalents.

**Explanation:**  
Successful migration requires careful code analysis, incremental adoption, parity testing, and retraining if necessary. Benefits include performance and .NET ecosystem improvements.27

## Conclusion

Passing senior-level EF Core interviews means demonstrating deep technical expertise complemented by real-world reasoning, awareness of the latest best practices, security, testing, debugging, and architectural skills. Mastering the topics above-using the examples provided-prepares you to discuss not just *how* EF Core works, but also *why* certain choices matter in high-stakes, production-grade environments. Regularly review official documentation, release notes, and open-source library updates to stay current as EF Core and the broader .NET ecosystem continuously evolve.

*This compilation draws on a wide array of current resources, expert blogs, technical documentation, and real interview experiences to ensure its practical relevance for top-tier senior developer interviews in 2025.*

# References (40)

1. *Entity Framework Tracking and AsNoTracking query performance comparison ...*. <https://davecallan.com/entity-framework-tracking-and-asnotracking-query-performance-comparison-with-benchmarks/>

2. *35 Entity Framework Interview Questions (ANSWERED - fullstack.cafe*. <https://www.fullstack.cafe/blog/entity-framework-interview-questions>

3. *How To Improve Performance With EF Core Query Splitting*. <https://www.milanjovanovic.tech/blog/how-to-improve-performance-with-ef-core-query-splitting>

9. *Entity Framework Migrations: Step-by-Step Guide & Best Practices - .Net ...*. <https://amarozka.dev/entity-framework-migrations/>

4. *EF Core Projection Queries - Tektutorialshub*. <https://www.tektutorialshub.com/entity-framework-core/projection-queries-in-ef-core>

5. *BenchmarkDotNet and EF Core vs EF6 - Part 2 - briangetsbinary.com*. <https://www.briangetsbinary.com/software%20engineering/dotnet/csharp/performance/2022/09/18/benchmarkdotnet-ef-core-vs-ef-6-part-2.html>

6. *Dapper vs. Entity Framework Core: Performance & Use Cases*. <https://blog.devart.com/dapper-vs-entity-framework-core.html>

7. *EF Core Migrations: A Detailed Guide - Milan Jovanovic*. <https://www.milanjovanovic.tech/blog/efcore-migrations-a-detailed-guide>

8. *Real-time Ecommerce Application using ASP.NET Core Web API - Dot Net ...*. <https://dotnettutorials.net/lesson/e-commerce-real-time-application-using-asp-net-core-web-api/>

10. *Implementing Seeding, Custom Conventions and Interceptors in EF Core 1.0*. <https://devblogs.microsoft.com/dotnet/implementing-seeding-custom-conventions-and-interceptors-in-ef-core-1-0/>

11. *6.0 Release Notes* . <https://www.npgsql.org/efcore/release-notes/6.0.html>

12. *GitHub - mhd-ahmadi/DynamicQueryBuilderDemo: A practical demo ...*. <https://github.com/mhd-ahmadi/DynamicQueryBuilderDemo>

13. *c# - Expression tree to SQL with EF Core - Stack Overflow*. <https://stackoverflow.com/questions/52017204/expression-tree-to-sql-with-ef-core>

14. *ChangeTracker, EntityEntry & Entity States in Entity Framework Core*. <https://www.tektutorialshub.com/entity-framework-core/change-tracker-entity-states-in-entity-framework-core/>

15. *Entity Framework Core: To Track or Not To Track?*. <https://amarozka.dev/ef-core-asnotracking-performance/>

16. *Tutorial: Handle concurrency - ASP.NET MVC with EF Core*. <https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/concurrency?view=aspnetcore-9.0>

19. *Injecting Service Dependencies to Entities with Entity Framework ... - ABP*. <https://abp.io/community/articles/injecting-service-dependencies-to-entities-with-entity-framework-core-7.0-db6vdh4s>

26. *Microsoft SQL Server Database Provider - EF Core*. <https://learn.microsoft.com/en-us/ef/core/providers/sql-server/>

24. *3 Ways to View and Log the SQL Generated by Entity Framework Core 5*. <https://eamonkeane.dev/3-ways-to-view-sql-generated-by-entity-framework-core-5/>

25. *Analyze BenchmarkDotNet data in Visual Studio*. <https://learn.microsoft.com/en-us/visualstudio/profiling/profiling-with-benchmark-dotnet?view=vs-2022>

18. *How to Mock EF Core DbContext - Code Maze*. <https://code-maze.com/ef-core-mock-dbcontext/>

23. *NuGet Gallery* . <https://www.nuget.org/packages/EFCoreSecondLevelCacheInterceptor/4.4.1>

21. *Fluent API Validation in ASP.NET Core Web API*. <https://dotnettutorials.net/lesson/fluent-api-validation-in-asp-net-core-web-api/>

27. *Breaking changes in EF Core 6.0 - EF Core* . <https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-6.0/breaking-changes>

17. *Overview of testing applications that use EF Core - EF Core*. <https://learn.microsoft.com/en-us/ef/core/testing/>

20. *Testing Repository Pattern Using Entity Framework - Code Maze*. <https://code-maze.com/testing-repository-pattern-entity-framework/>

22. *EF Bulk Extensions is really slow for me, but why?*. <https://stackoverflow.com/questions/79252117/ef-bulk-extensions-is-really-slow-for-me-but-why>