FIVE TIPS & TRICKS TO ENHANCE ENTITY FRAMEWORK PERFORMANCE (PART5)

Welcome to the exciting fifth episode of our Entity Framework Performance Tips & Tricks series! In this installment, we'll dive into the world of implementing optimistic concurrency control in Entity Framework to ensure data consistency and handle concurrent updates effectively.

Get ready for a thrilling exploration of techniques that will empower you to build robust and scalable applications with confidence..

DBCONTEXT POOLING

DbContext pooling allows you to reuse existing DbContext instances instead of creating new ones for each request.

By reusing the DbContext instances, you can reduce the overhead of initializing the context, which can significantly improve performance, especially in scenarios with high request loads.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add DbContext with pooling enabled
    services.AddDbContextPool<ApplicationDbContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString(
           "DefaultConnection"));
    });
}
```

ASYNCHRONOUS METHODS

Entity Framework provides asynchronous methods for data retrieval and modification operations.

By using async/await pattern, you can execute database queries and updates asynchronously, which improves scalability and responsiveness of your application.

Asynchronous methods allow the application to perform other tasks while waiting for the database operations to complete, leading to better resource utilization..

```
public async Task<List<Customer>> GetCustomersAsync()
{
    using (var dbContext = new ApplicationDbContext())
    {
        // Use asynchronous method to query customers
        return await dbContext.Customers.ToListAsync();
    }
}
```

UTILIZE TRANSACTIONS

Transactions ensure data consistency and integrity in multi-step database operations.

By wrapping multiple database operations within a transaction, you can guarantee that either all the operations succeed or none of them take effect.

This is particularly important when dealing with complex business logic that involves multiple database modifications.

Using transactions correctly can prevent data inconsistencies and improve the reliability of your application.

```
public void TransferFunds(Account sourceAccount, Account destinationAccount,
decimal amount)
{
    using (var dbContext = new ApplicationDbContext())
    {
        using (var transaction = dbContext.Database.BeginTransaction())
        {
            try
            {
                // Retrieve source and destination accounts
                var source = dbContext.Accounts.Find(sourceAccount.Id);
                var destination = dbContext.Accounts.Find(destinationAccount.Id);
```

```
                // Perform the funds transfer within the transaction
                source.Balance -= amount;
                destination.Balance += amount;

                // Save changes to the database
                dbContext.SaveChanges();

                // Commit the transaction if all operations succeed
                transaction.Commit();
            }
            catch (Exception ex)
            {
                // Handle any exceptions and rollback the transaction if
necessary
                transaction.Rollback();
                throw ex;
            }
        }
    }
}
```

VALIDATION LOGIC

Entity Framework provides two common approaches for implementing validation logic: data annotations and Fluent API.

Data annotations allow you to define validation rules directly on your entity classes, while Fluent API provides a more flexible and centralized way to define validation rules.

By enforcing data validation at the database level, you can ensure the integrity and quality of your data, leading to more robust and reliable applications.

```
public class Product
{
    [Key]
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    [Range(0, 9999)]
    public decimal Price { get; set; }

    [EmailAddress]
    public string Email { get; set; }
}
```

OPTIMISTIC CONCURRENCY

Optimistic concurrency control is a technique used to handle concurrent updates to the same database entity.

Entity Framework offers built-in support for optimistic concurrency control, allowing you to detect and handle conflicts when multiple users try to modify the same data simultaneously.

By implementing optimistic concurrency control, you can prevent data corruption and ensure that changes are applied consistently across all users.

```
public class Product
{
    [Key]
    public int Id { get; set; }

    [ConcurrencyCheck]
    public string Name { get; set; }

    public int UnitsInStock { get; set; }
}
```

Or

```
public class Product
{
    [Key]
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    public int UnitsInStock { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

CONTINUE YOUR LEARNING JOURNEY!

Thank you for joining us on this incredible journey through the Tick and Trip series on Entity Framework! We hope you found these tutorials helpful and insightful. Don't miss out on the next exciting tutorials and valuable content! Follow us on social media to stay connected, get updates, and continue your learning journey. We look forward to sharing more knowledge with you!

Follow us:

- LinkedIn: KHORGAMI
- medium: @Khourgami

Stay tuned for more tutorials, tips, and tricks!

Vahid Khorgami