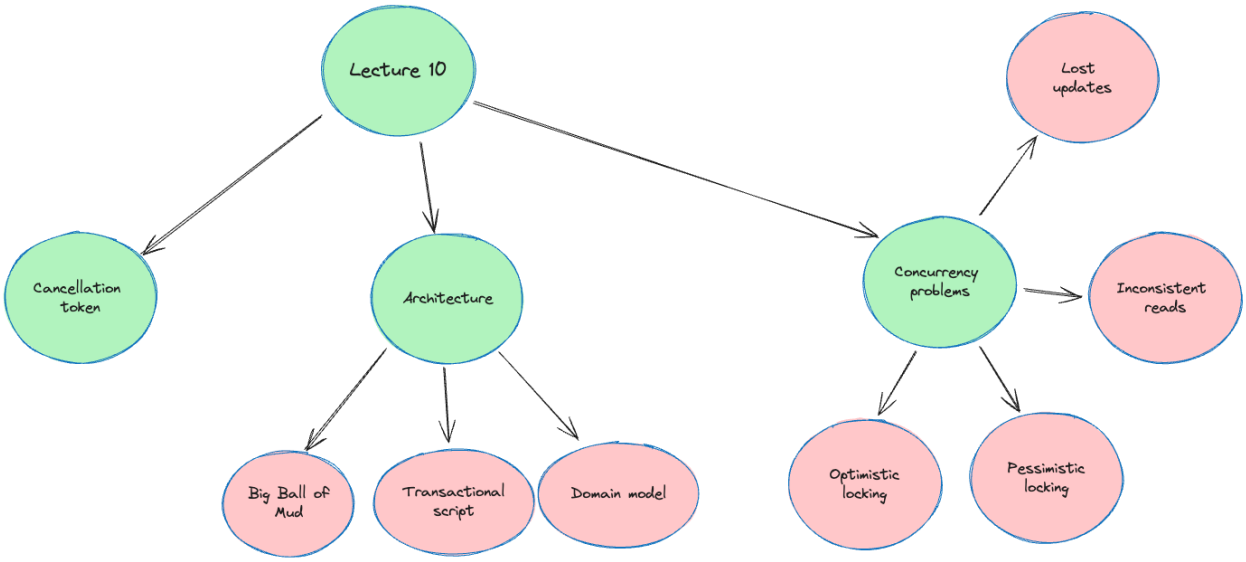
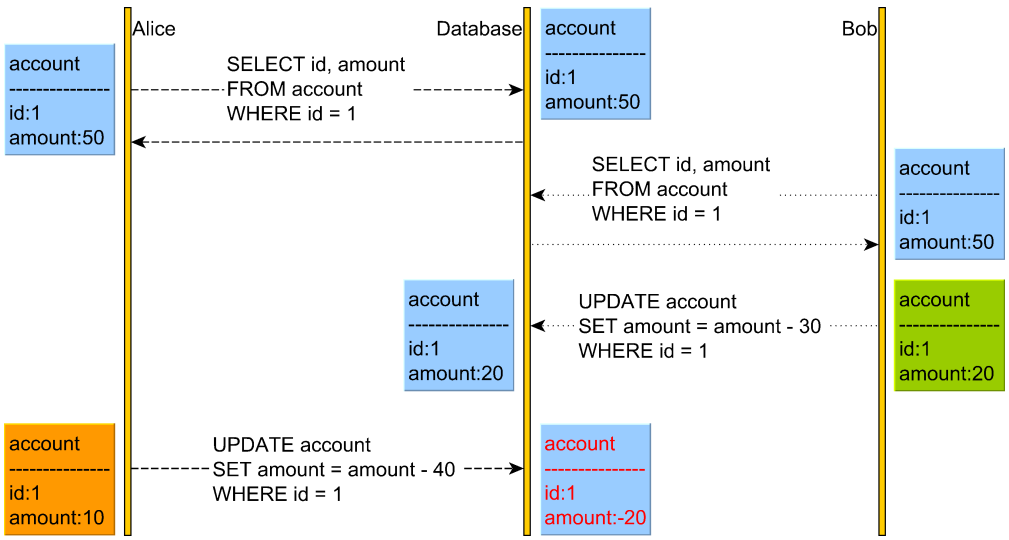


Optimistic and pessimistic locking

Problem - lost updates

One student saves their changes (commit a transaction), and then another student, who started editing before the first save, saves their changes afterward. The second save might overwrite the first student's changes, leading to lost updates.

Another example - here the potential of errors might be much more serious.
What will be the result of the following situation?



Problem - inconsistent reads

While one student is editing a section, another student might read that section. If another transaction commits changes to the data, the first student might see different versions of the same data within the same transaction.

This is not ideal, as the user would like to be sure that the data is not being changed while they are within their transaction.

Solution

We have generally two strategies to solve this problem. Optimistic concurrency and pessimistic concurrency. By default most ORM are using optimistic concurrency locking.

Optimistic locking

Optimistic locking is a strategy used to manage concurrency control in database systems, where it allows multiple transactions to proceed without locking resources upfront but checks for conflicts before the transaction commits.

Steps in Optimistic Locking

1. **Read Phase:**
    - The transaction reads the current state of the data it needs to work with.
    - Along with the data, it also reads a version identifier (or timestamp) associated with the data. This version identifier is used to detect changes.
2. **Work Phase:**

- The transaction performs the necessary operations on the data locally (in memory), without holding any locks on the database.
- During this phase, other transactions are free to access and modify the same data concurrently.

3. **Validation Phase:**
    - Before the transaction is ready to commit, it enters the validation phase.
    - The transaction re-reads the version identifier of the data it initially read.
    - It compares this version identifier with the version identifier currently in the database.
    - If the version identifier has changed, it indicates that another transaction has modified the data since it was initially read, leading to a conflict.
    - If a conflict is detected, the transaction is rolled back. The transaction must be retried from the beginning.
4. **Commit Phase:**
    - If no conflict is detected (i.e., the version identifier has not changed), the transaction proceeds to commit.
    - During the commit, the data is written back to the database, and the version identifier is updated to reflect the new state.

Example in SqlConnection/SqlCommand

```
create table Account2
(
    IdAccount INT PRIMARY KEY IDENTITY,
    Balance INT,
    RowVersion ROWVERSION
);
```

C# code:

```
await using var con = new
SqlConnection(_configuration.GetConnectionString("Defaul
tConnection"));
await using var com = new SqlCommand();

await con.OpenAsync();

SqlTransaction tran = (SqlTransaction) await
con.BeginTransactionAsync();

com.Transaction = tran;
com.Connection = con;
com.CommandText = "SELECT IdAccount, Balance, RowVersion
FROM Account2 WHERE IdAccount=@IdAccount";
com.Parameters.AddWithValue("IdAccount", 1);

var dr = await com.ExecuteReaderAsync();

Account account=null;
if (await dr.ReadAsync())
{
    account = new Account
    {
        IdAccount = (int)dr["IdAccount"],
        Balance = (int)dr["Balance"],
```

```
        RowVersion = (byte[])dr["RowVersion"]
    };}

await dr.CloseAsync();

if (account.Balance < 1000)
{
    return BadRequest("Not enough money");
}

com.Parameters.Clear();
com.CommandText = @"
UPDATE Account2 SET Balance=Balance-1000 WHERE
IdAccount=1 AND RowVersion=@RowVersion;";
com.Parameters.AddWithValue("RowVersion",
account.RowVersion);


int affectedCount=await com.ExecuteNonQueryAsync();

await tran.CommitAsync();

if (affectedCount == 0)
{
    return BadRequest("Conccurency error");
}
```

Example in Entity Framework

Entity Framework Core supports two approaches to concurrency conflict detection: configuring existing properties as concurrency tokens; and adding a "rowversion" property to act as a concurrency token.

ConcurrencyCheck

Properties can be configured as concurrency tokens via data annotations by applying the **ConcurrencyCheck** attribute:

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    [ConcurrencyCheck]
    public string LastName { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

Or

```
public class SampleContext : DbContext
{
    public DbSet<Author> Authors { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
         modelBuilder.Entity<Author>()
            .Property(a =>
a.LastName).IsConcurrencyToken();
    }
}
public class Author
```

```
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

Any existing properties that have been configured as concurrency tokens will be included with their original values in the WHERE clause of an UPDATE or DELETE statement. When the SQL command is executed, EF Core expects to find one row that matches the original values.
If any of the configured columns have had their values changed between the time that the data was retrieved and the time that the changes are sent to the database, EF Core will throw a DbUpdateConcurrencyException with the message.

The concurrency token configuration can be applied to as many non-primary key properties as needed.
Care needs to be taken as this approach can lead to very long `WHERE` clauses, or a lot of data being passed into them especially if any of the properties being configured as concurrency tokens are unlimited string values as is illustrated where, where the `Biography` field has been included as a concurrency token.

RowVersion

The second approach to concurrency management involves adding a column to the database table to store a version stamp for the row of data.
Different database systems approach this requirement in different ways. SQL Server offers the rowversion data type for this purpose. The column stores an incrementing number. Each time the data is inserted or modified, the number increments.

User A might retrieve a row of data, followed by User B.
The `rowversion` value for the row will be the same for both users. If User B submits changes, the `rowversion` value in the table will increment by 1 for that row. If User A subsequently tries to modify the same record, the `rowversion` value in their `WHERE` clause combined with the primary key value will no longer match an existing row in the database and EF Core will throw a `DbUpdateConcurrencyException`.

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public ICollection<Book> Books { get; set; }
    [TimeStamp]
    public byte[] RowVersion { get; set; }
}
```

Or

```
public class SampleContext : DbContext
{
    public DbSet<Author> Authors { get; set; }

    protected override void OnModelCreating(ModelBuilder
```

```
modelBuilder)
    {
        modelBuilder.Entity<Author>()
           .Property(a => a.RowVersion).IsRowVersion();
    }
}
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public ICollection<Book> Books { get; set; }
    public byte[] RowVersion { get; set; }
}
```

We can catch the concurrency error the following way:

```
try
{
    db.SaveChanges();
    // move on
}
catch (DbUpdateException)
{
    // get the latest version of the record for display
}
```

Pessimistic locking

Pessimistic locking is a concurrency control strategy used in database systems where data is locked as soon as it is accessed, and the lock is held for the duration of the transaction.
This approach prevents other transactions from modifying the locked data until the lock is released, thus avoiding conflicts by ensuring that only one transaction can modify the data at a time.

Relational databases

When it comes to relational databases we usually have the option of setting the proper isolation level for our transaction. Depending in the requirement we have to remember that higher isolation level will result in more locks and will hinder our concurrency.

Isolation levels:

- **Read Uncommitted** - Allows for dirty reads, non-repeatable reads, and phantom reads.
- **Read Committed** - Protects us from dirty reads.
- **Repeatable Read** - Protects us from non-repeatable reads.
- **Serializable** - Protects us from all the aforementioned anomalies.

Brief description of each isolation level:

- **Dirty Read**: Reading uncommitted changes from another transaction.
- **Non-Repeatable Read**: Reading the same data twice and getting different results due to another transaction's updates.
- **Phantom Read**: Reading a set of rows that change due to another transaction's inserts or deletes during the transaction.

Steps in Pessimistic Locking

1. **Lock Phase:**
    - When a transaction needs to read or write data, it requests a lock on the data.
    - If the data is already locked by another transaction, the requesting transaction must wait until the lock is released.
    - Once the lock is acquired, the transaction has exclusive access to the data.
2. **Work Phase:**
    - The transaction performs the necessary operations on the locked data.
    - During this phase, no other transactions can read or modify the locked data.
3. **Commit Phase:**
    - The transaction completes its operations and prepares to commit.
    - The changes are written to the database.
    - The lock is released, allowing other transactions to access the data.
4. **Rollback Phase (if necessary):**
   - If an error occurs or the transaction needs to be rolled back, the changes are discarded.
   - The lock is released, reverting the data to its previous state.

Example in SqlConnection/SqlCommand

We can use the pessimistic strategy either by employing proper transaction isolation levels or by using additional hints to explicitly tell SQL Server to acquire a lock.

Using proper transaction isolation level:

```
await using var con = new
SqlConnection(_configuration.GetConnectionString("Defaul
tConnection"));
await using var com = new SqlCommand();

await con.OpenAsync();

SqlTransaction tran = (SqlTransaction) await
con.BeginTransactionAsync(IsolationLevel.Serializable);

com.Transaction = tran;
com.Connection = con;
com.CommandText = "SELECT IdAccount, Balance FROM
Account WHERE IdAccount=@IdAccount";
com.Parameters.AddWithValue("IdAccount", 1);
```

Explicitly acquiring a lock:

```
await using var con = new
SqlConnection(_configuration.GetConnectionString("Defaul
tConnection"));
await using var com = new SqlCommand();

await con.OpenAsync();

SqlTransaction tran = (SqlTransaction) await
con.BeginTransactionAsync();
```

```
com.Transaction = tran;
com.Connection = con;
com.CommandText = "SELECT IdAccount, Balance FROM
Account WITH (UPDLOCK) WHERE IdAccount=@IdAccount";
com.Parameters.AddWithValue("IdAccount", 1);

var dr = await com.ExecuteReaderAsync();
```

Entity Framework

Entity Framework Core provides no support for pessimistic concurrency control.

Architecture - general guidelines

General concerns

We have already discussed that according to the SRP - Single Responsibility principle - class should have "one reason to change". The question is what are those reasons?

We have discussed before that usually that usually we can divide concerns into:

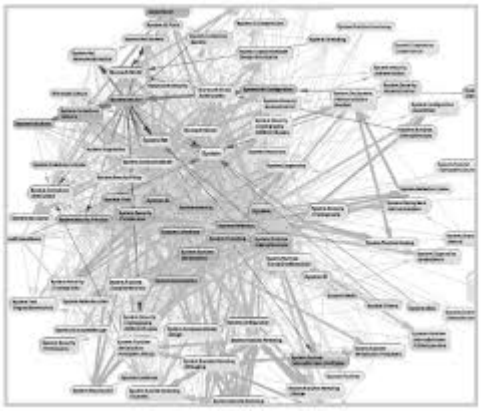
- **Domain (Business Logic) Concerns:** This is logic strictly related to our business requirements.
- **Infrastructure Concerns:** These are all the elements necessary for our application to work correctly but are not important from a business perspective—such as persistence, communication with external services, and reading from or writing to files.
- **UI Concerns:** Everything related to the interface of our application. This interface does not necessarily have to be a GUI. If our application is a REST API, then our interface is really just a JSON file.

These are typical elements that we can find in most applications. In an ideal world, we would like to separate these concerns because they usually change independently. If we change the business logic, we typically modify the domain. If we need to change the database for performance reasons, we adjust elements in the infrastructure.

Investing time in separating these concerns allows our code to be more testable and modifiable in the future.

Of course, the more we think about the architecture, the more complex the code can become. There is no single best architecture available.

Approach 1 - Big Ball of Mud

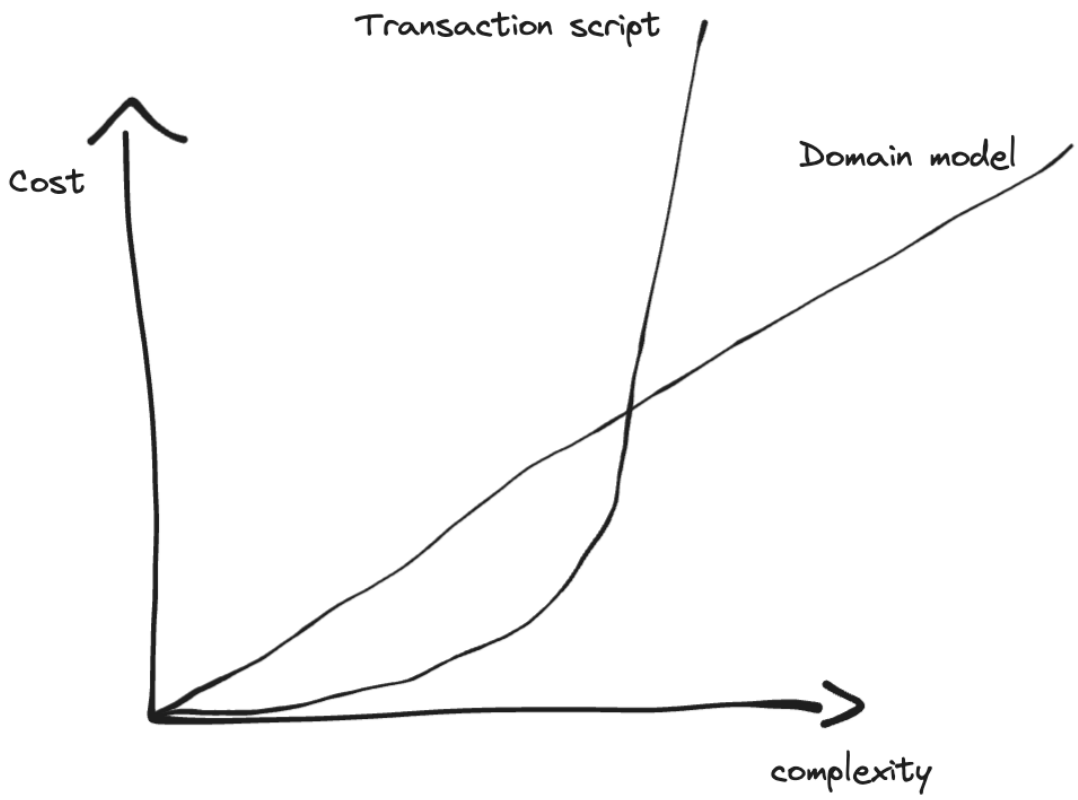


As you might guess, "Big Ball of Mud" means that we do not really have any idea how to divide our code. Maybe we squeeze everything into the controller, or maybe we mix business logic with database logic. A Big Ball of Mud is associated with increasing complex coupling, which over time renders the application very hard to modify and extend.

Approach 2 - Transactional script

A transactional script is a design pattern used to organize business logic in an application, often in the context of a service layer. This is currently the place where we usually include the business logic.
This pattern is straightforward and suitable for applications where the business logic is simple and does not require extensive layering or abstraction.
Making sure that the business logic is separated from other concerns. This way we may try to keep our application testable.

Approach 3 - Domain model



In the domain model approach we try to create a model domain of the business whcih includes our business invariants/rules. The logic is slowly moved towards rich object models - instead of weak classes which only contain properties.

A domain model in ASP.NET (or any software development context) represents the conceptual structure and the primary entities involved in the business logic of an application. It serves as an abstraction of the real-world entities and their relationships within the application domain.

Domain model and EF models

In some of the solutions we might try to use our EF models as domain models. This may be however hard because by definition th edomain model should be persistant ignorant. in other words we do not care how the data are persisted.
Moreover the ORM requires sometimes certain rules that sometimes break the encapsulation of our model - for ex. a lot of ORM solutions require all properties to be publis which is not good from the perspective of the encapsulation.
In other words reusing the EF models as domain models is possible, but it may be limiting.

Persistant ignorant domain model

In this approach, we create a completely persistence-ignorant domain model. In other words, Entity Framework (EF) models are completely separate from domain model classes. Of course, this kind of approach introduces additional complexity and requires additional mapping. Nevertheless, this makes us completely independent of the persistence layer and allows us to be more flexible.

How to model domain model?

Creating a proper domain model is not easy and is outside the scope of this discussion. There are many different approaches and design patterns we can use while working on the model. Many of them are related to the DDD (Domain-Driven Design) approach introduced in 2003 by Eric Evans. Domain-Driven Design, as the name suggests, tries to explain how to create complex systems that match user requirements.
Remember that the main reason for creating a domain model is to concentrate business rules in a single place and be able to easily modify them in the future while keeping the consistency of the system intact.

Some of the example design patterns used when working with domain model:

Entities

Entities are objects that have a distinct identity that runs through time and different states. They are defined by their unique identifier rather than by their attributes.

- **Example:** A `User` with an `ID` attribute. Even if the user's details (name, email) change, the `ID` remains the same, identifying the same user entity.

Value Objects

Value Objects are objects that describe some characteristic or attribute but have no conceptual identity. They are immutable and distinguished only by their attributes.

- **Example:** An `Address` with attributes like `Street`, `City`, and `ZipCode`. Two addresses with the same values for these

attributes are considered equal.

Domain Events

Domain Events are events that are significant to the domain and indicate that something important has happened. They can trigger side effects or notify other parts of the system.

- **Example:** An `OrderPlaced` event that is raised when a new order is created.

Domain Services

Domain Services are operations that don't naturally fit within a single entity or value object. They are used for domain logic that spans multiple entities.

- **Example:** A `CurrencyConversionService` that converts an amount from one currency to another.

Aggregates

Aggregates are clusters of entities and value objects that are treated as a single unit. Each aggregate has a root entity (the aggregate root) that is the only entity accessible from outside.

- **Example:** An `Order` aggregate with `OrderLine` entities. The `Order` is the aggregate root, and all operations on `OrderLine` entities go through the `Order`.

Factories

Factories are objects that handle the creation of complex objects and aggregates. They encapsulate the logic needed to instantiate objects correctly.

- **Example:** An `OrderFactory` that creates an `Order` aggregate with all necessary `OrderLines`.

Repositories

Repositories are mechanisms for encapsulating storage, retrieval, and search behaviour which emulates a collection of objects. They provide methods to add, remove, and query objects.

- **Example:** An `OrderRepository` that provides methods to save, update, delete, and fetch `Order` aggregates.

Architectures using the domain model approach

- **Onion architecture** - Onion Architecture, introduced by Jeffrey Palermo, emphasises the separation of concerns by organising the code into concentric layers, resembling an onion.
- **Hexagonal architecture** - Hexagonal Architecture, also known as Ports and Adapters, was introduced by Alistair Cockburn. It promotes creating a system that is decoupled from its external interfaces.
- **Ports and adapter architecture** - This is essentially another name for Hexagonal Architecture. The terms are used interchangeably.
- **Clean architecture** - Clean Architecture, proposed by Robert C. Martin (Uncle Bob), aims to create a structure where the business logic is independent of frameworks, UI, and external systems.

Additional useful patterns

Below, we present a few useful design patterns that we will be using in the example solutions.

Repository pattern

The Repository Pattern is a design pattern that mediates data access and domain logic. It provides a centralized place for data access logic, making it easier to manage and test. By abstracting the data access layer, the Repository Pattern helps to achieve a cleaner separation of concerns within an application.

Key Concepts:

1. **Abstraction**: Abstracts the data access logic from the business logic.
2. **Encapsulation**: Encapsulates the logic for retrieving and manipulating data.
3. **Testability**: Makes it easier to mock and test data access in unit tests.

UnitOfWork pattern

The Unit of Work (UoW) pattern is a design pattern that maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. Essentially, it helps manage transactions, ensuring that all operations within a single transaction either complete successfully or fail together.

Key Concepts:

1. **Transaction Management**: Ensures that a series of operations are treated as a single transaction. If any operation fails, all

changes are rolled back.

2. **Repository Coordination**: Coordinates changes across multiple repositories.
3. **Change Tracking**: Keeps track of changes to objects so that only those changes are persisted to the database.

Cancellation token

A Cancellation Token in ASP.NET and asynchronous programming is a mechanism that allows you to gracefully manage the cancellation of long-running tasks.

A Cancellation Token is a struct in .NET that is used to signal that an operation should be canceled. It's part of the `System.Threading` namespace and works in conjunction with the `CancellationTokenSource` class. The `CancellationTokenSource` generates the token, which can then be passed to asynchronous methods to monitor for cancellation requests.

Why Use a Cancellation Token?

1. **Graceful Termination**: It allows for a graceful termination of tasks. Instead of abruptly terminating a task, it can be informed to stop, giving it the opportunity to clean up resources properly.
2. **Improved Responsiveness**: By using cancellation tokens, you can make your application more responsive. Users can cancel long-running operations (like file uploads, database queries, or web requests), improving the user experience.
3. **Resource Management**: Canceling tasks that are no longer needed helps in efficient resource utilization. It prevents the system from wasting CPU cycles, memory, or other resources on operations that are not required.
4. **Avoiding Timeouts**: In web applications, long-running requests might timeout if they exceed the server's limits. Cancellation tokens can help to cancel the task before it hits the timeout, allowing you to handle such scenarios more gracefully.

Example in CS

Creating a CancellationTokenSource:

```
var cancellationTokenSource = new
CancellationTokenSource(); var token =
cancellationTokenSource.Token;
```

Passing the Token to an Async Method:

```
await LongRunningOperationAsync(token);
```

Monitoring Cancellation in the Async Method:

```
public async Task
LongRunningOperationAsync(CancellationToken token)
{
    for (int i = 0; i < 10; i++)
    {
        token.ThrowIfCancellationRequested();
        // Simulate work
        await Task.Delay(1000);
    }
}
```

Triggering the Cancellation:

```
cancellationTokenSource.Cancel();
```

Example in ASP.NET

Example of using cancellation token in ASP.NET endpoint.

```
[HttpGet("long-running-operation")]
public async Task<IActionResult>
LongRunningOperation(CancellationToken
cancellationToken)
{
    try
    {
        await
LongRunningOperationAsync(cancellationToken);
        return Ok("Operation completed successfully.");
    }
    catch (OperationCanceledException)
    {
        return StatusCode(499, "Client closed
request.");
    }
    catch (Exception ex)
    {
        return StatusCode(500, $"Internal server error:
{ex.Message}");
    }
}

public async Task
LongRunningOperationAsync(CancellationToken token)
{
    for (int i = 0; i < 10; i++)
    {
        token.ThrowIfCancellationRequested();
        // Simulate work
        await Task.Delay(1000, token);
    }
```

```
}
```