Avoiding Unnecessary Work with Laziness

Mark Heath
SOFTWARE DEVELOPER

@mark_heath www.markheath.net

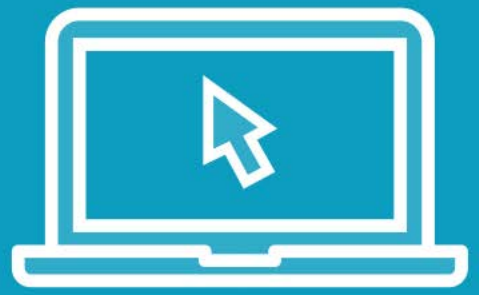
Overview

Avoid doing any more work than necessary

Three ways to be lazy

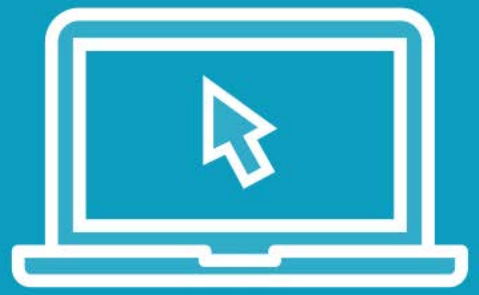
- Don’t start iterating until you need to
- Don’t iterate through more elements than you need to
- Avoid iterating through more than once

Demo



Deferred Execution

Demo



RSS Downloader

Breaking Out Early

The “any” pattern:

```
bool anyRefunded = false;
foreach (var order in orders)
{
    if (order.Status == "Refunded")
    {
        anyRefunded = true;
        break;
    }
}
```

With LINQ:

```
orders.Any(o => o.Status == "Refunded")
```

Other LINQ short-circuiting methods:

- First
- FirstOrDefault
- Take
- All

Some LINQ methods will always evaluate the entire sequence. e.g.

- ToList
- Max
- ToArray
- Last

Avoiding Multiple Enumeration

Reasons to avoid iterating through an IEnumerable<T> more than once

Performance

Especially if the pipeline contains long-running methods

Correctness

You can get different results each time you iterate

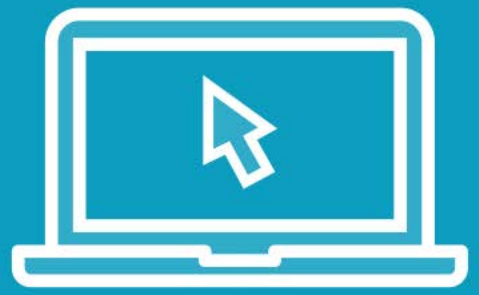
Should I Use ToList?

Only if you know you need the entire sequence cached in memory

If you want to safely enumerate multiple times

Avoid if you have a huge data set

Demo



Multiple Enumeration and Databases

ToList and Databases

Let the database do the hard work for you

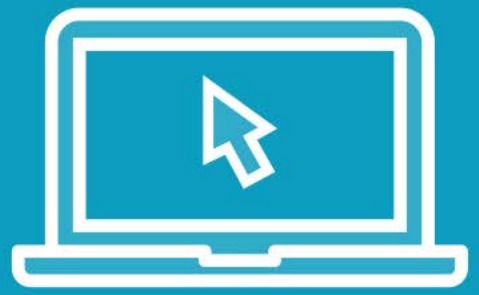
(e.g. sorting, grouping, paging, filtering)

Avoid retrieving more data than you need

Understand when your SQL statements will be executed

ToList will cause immediate evaluation

Demo



Multiple Enumeration and Correctness

Returning IEnumerable<T>

```
public ??? GetOrdersForDelivery()
```

Return Type	Implications
`Order[]`	The results are already in memory and we can safely multiply enumerate.
`List<Order>`	In memory but … do we own this list? May wish to call ToList again.
`IReadOnlyCollection<Order>`	An in-memory list that we know won't change.
`IEnumerable<Order>`	Might take advantage of deferred execution. Not safe to multiply enumerate.
`IQueryable<Order>`	Likely to be a deferred execution database query. Can chain on additional clauses before executing.

IEnumerable<T> Function Parameters

Make it easy for the caller by accepting IEnumerable<T>

Don’t require them to pass an Array or List<T>

```
void ShipOrders(IEnumerable<Order> orders)
{
   // can cache for ourselves if we want to
   orders.ToList()
}
```

Summary

Three ways to be lazy

- Don’t start iterating until you need to
- Don’t iterate through more elements than you need to
- Avoid iterating through multiple times

Let the database do the hard work

- Avoid pulling down more data than you need to

Next up: Performance