ekklo dale

BIM SOFTWARE
DEVELOPMENT
AND CONSULTING

Schritt-für-Schritt-Anleitung

KEINE DATENBANK…

In viele Applikationen wird eine Möglichkeit zur Speicherung von Daten benötigt. In vielen Fällen wird dazu eine Datenbank benutzt. Die Einrichtung einer Datenbank ist aber mit Aufwand verbunden, und der Einsatz der Datenbank "überdimensioniert"; man möchte vielleicht "nur ein paar Werte" speichern und diese Daten dem Kunden überreichen. Dazu muss man die Daten aus der Datenbank in ein Format konvertieren, welches der Kunde auch lesen kann, z.B. Excel.

Wie wäre es die Daten direkt in Excel zu speichern, statt den Umweg über die Datenbank zu machen? Hier tritt schon mal ein Problem auf: die Daten könnten "gleichzeitig" über eine API an die Applikation geschickt werden, das Schreiben in eine Datei müsste man synchronisieren. Mit einer relationalen Datenbank hätte man das Problem in dieser Form nicht – eine Datenbank kann mit konkurrierendem Zugriff umgehen.

Mit Hilfe von DataFlow, der Task Parallel Library von Microsoft, lassen sich aber solche konkurrierenden Zugriffe sehr elegant lösen. An einem einfachen Beispiel soll dies demonstriert werden.

Szenario

Wir nehmen an, wie haben eine Asp.Net Core Web API. Die API hat nur einen Endpunkt, über den ein Name in eine Exceldatei (CSV-Format) auf dem Server abgespeichert werden soll. Der Server ergänzt zudem noch den Datensatz um den Zeitpunkt, wann der Datensatz gespeichert worden ist.

Auch wenn die Speicheranfragen "gleichzeitig" eintreffen, sollen alle Datensätze korrekt in der Exceldatei gespeichert werden.

DataFlow

Die Task Parallel Library (TPL) wird verwendet, um die Robustheit von nebenläufigen Anwendungen zu erhöhen. Sie stellt zu diesem Zweck verschiedene Klassen zur Verfügung.

Die Klasse BufferBlock stellt eine FIFO-Warteschlange für Nachrichten dar. Die Nachrichten können in diese Warteschlange geschrieben und aus dieser wieder gelesen werden. Ein BufferBlock kann somit als Nachrichtenquelle oder Nachrichtenziel fungieren.

Mit der Klasse ActionBlock wird die Aktion definiert, welche beim Eintreffen einer Nachricht ausgeführt wird.

Mit Hilfe der Methode LinkTo(..) kann man einen Datenfluss zwischen Quelle und Ziel erzeugen. D.h. ist die Quelle mit dem Ziel verlinkt und kommt eine Nachricht in der Quelle an, wird diese an das Ziel weitergereicht.

Dieses Wissen reicht schon aus, um unserem Szenario gerecht zu werden.

Umsetzung

Beim Start der Anwendung prüfen wir, ob die gewünschte Datei existiert. Wenn nicht, legen wir diese an schreiben die Kopfzeile rein.

```
public void ConfigureServices(IServiceCollection services)
{
   if (!File.Exists("data.csv"))
   {
      File.AppendAllLines("data.csv", new[] { $"Name,CreatedAt" });
   }

   // . . .

   services.AddControllers();
}
```

Im zweiten Schritt erstellen wir den "Konsumenten". Wir definieren diesen als ActionBlock und geben gleich an, wie die Daten in die Excelliste geschrieben werden sollen. Immer wenn der Konsument eine Nachricht empfängt, wird dieser Codeblock ausgeführt.

```
public void ConfigureServices(IServiceCollection services)
{
   if (!File.Exists("data.csv"))
   {
      File.AppendAllLines("data.csv", new[] { $"Name,CreatedAt" });
   }

   var dataWriter = new ActionBlock<DataEntry>(entry =>
   {
      var data = $"{entry.Name},{entry.CreatedAt}";
      File.AppendAllLines("data.csv", new[] { data });
   });

   // . . .

   services.AddControllers();
}
```

Im nächsten Schritt definieren wir den "Produzenten", den Erzeuger der Nachricht. Diesen definieren wir als BufferBlock und verlinken ihn mit dem "Konsumenten". Durch die Verlinkung wird eine Nachricht, welche in den Buffer reingestellt wird, an den Konsumenten weitergereicht.

```
public void ConfigureServices(IServiceCollection services)
{
   if (!File.Exists("data.csv"))
   {
      File.AppendAllLines("data.csv", new[] { $"Name,CreatedAt" });
   }

   var dataWriter = new ActionBlock<DataEntry>(entry =>
   {
      var data = $"{entry.Name},{entry.CreatedAt}";
      File.AppendAllLines("data.csv", new[] { data });
   });

   var dataBuffer = new BufferBlock<DataEntry>();
   dataBuffer.LinkTo(dataWriter);

   // . . .

   services.AddControllers();
}
```

Dann stellen wir den dataBuffer per DependencyInjection zur Verfügung.

```
public void ConfigureServices(IServiceCollection services)
{
   if (!File.Exists("data.csv"))
   {
      File.AppendAllLines("data.csv", new[] { $"Name,CreatedAt" });
   }

   var dataWriter = new ActionBlock<DataEntry>(entry =>
   {
      var data = $"{entry.Name},{entry.CreatedAt}";
      File.AppendAllLines("data.csv", new[] { data });
   });

   var dataBuffer = new BufferBlock<DataEntry>();
   dataBuffer.LinkTo(dataWriter);

   services.AddSingleton(dataBuffer);

   // . . .
   services.AddControllers();
}
```

Zuletzt folgt der Controller mit der Post-Methode. Wir erstellen aus den an die API gesendeten Daten die Nachricht und schreiben sie mit der Methode SendAsync(…) in die Warteschlange.

```
public class DataController : ControllerBase
{
   private readonly BufferBlock<DataEntry> _bufferBlock;

   public DataController(BufferBlock<DataEntry> bufferBlock)
   {
      _bufferBlock = bufferBlock;
   }

   [HttpPost]
   public async Task<IActionResult> SubmitEntry([FromBody] DataEntryDto entry)
   {
      entry.Name = entry.Name.Trim();
      var dataEntry = new DataEntry()
      {
         Name = entry.Name.Trim(),
         CreatedAt = DateTime.UtcNow,
      };
      await _bufferBlock.SendAsync(dataEntry);
      return Ok();
   }
}
```

Das ist auch schon alles. Zu beachten ist nur noch die Größe und die Konfiguration der Warteschlange. Diese Informationen können aber der Dokumentation entnommen werden.