ASP.NET Core SignalR에서 스트리밍 사용

2019 년 11 월 12 일 • 읽는 데 9 분 • +7

이 기사에서

으로 브레넌 콘로이

ASP.NET Core SignalR은 클라이언트에서 서버로, 서버에서 클라이언트로 스트리밍을 지원합니다. 이는 시간이 지남에 따라 데이터 조각이 도착하는 시나리오에 유용합니다. 스트리밍 할 때 각 조각은 모든 데이터를 사용할 수있을 때까지 기다리지 않고 사용할 수있게되는 즉시 클라이언트 또는 서버로 전송됩니다.

샘플 코드보기 또는 다운로드 (다운로드 방법)

스트리밍을위한 허브 설정

이 반환 될 때 허브있어서 자동 스트리밍 허브있어서된다 IAsyncEnumerable <T> , ChannelReader <T> , `Task<IAsyncEnumerable<T>>`, 또는 `Task<ChannelReader<T>>`.

서버-클라이언트 스트리밍

스트리밍 허브 방법은 반환 할 수 있습니다 `IAsyncEnumerable<T>`뿐만 아니라 `ChannelReader<T>`. 반환하는 가장 간단한 방법 `IAsyncEnumerable<T>`은 다음 샘플에서 보여주는 것처럼 허브 메서드를 비동기 반복기 메서드로 만드는 것입니다. 허브 비동기 반복기 메서드는 `CancellationToken`클라이언트가 스트림에서 구독을 취소 할 때 트리거 되는 매개 변수를 수락 할 수 있습니다 . 비동기 반복기 메서드는 `ChannelReader`충분히 일찍 반환 하지 않거나 ChannelWriter <T>를 완료하지 않고 메서드를 종료하는 것과 같은 Channels의 일반적인 문제를 방지합니다 .

ⓘ **노트**

다음 샘플에는 C # 8.0 이상이 필요합니다.

씨#

```
public class AsyncEnumerableHub : Hub
{
    public async IAsyncEnumerable<int> Counter(
        int count,
        int delay,
        [EnumeratorCancellation]
        CancellationToken cancellationToken)
    {
        for (var i = 0; i < count; i++)
        {
            // Check the cancellation token regularly so that the server
will stop
            // producing items if the client disconnects.
            cancellationToken.ThrowIfCancellationRequested();

            yield return i;

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
}
```

다음 샘플은 채널을 사용하여 클라이언트로 데이터를 스트리밍하는 기본 사항을 보여줍니다. 개체가 ChannelWriter <T>에 기록 될 때마다 개체는 즉시 클라이언트로 전송됩니다. 마지막 `ChannelWriter` 에는 클라이언트에게 스트림이 단 혔음을 알리기 위해 완료됩니다.

ⓘ **노트**

`ChannelWriter<T>` 백그라운드 스레드에 작성하고 `ChannelReader` 가능한 한 빨리를 반환하십시오 . 다른 허브 호출은 a `ChannelReader` 가 반환 될 때까지 차단 됩니다.

로직을 `try ... catch`. 를 완료 `Channel` 에서 `catch` 와 밖에서 `catch` 확인 허브 메소드 호출이 제대로 완료되었는지 확인 할 수 있습니다.

씨#

```
public ChannelReader<int> Counter(
public ChannelReader<int> Counter(
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    var channel = Channel.CreateUnbounded<int>();

    // We don't want to await WriteItemsAsync, otherwise we'd end up waiting

    // for all the items to be written before returning the channel back to
    // the client.
    _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

    return channel.Reader;
}

private async Task WriteItemsAsync(
    ChannelWriter<int> writer,
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    Exception localException = null;
    try
    {
        for (var i = 0; i < count; i++)
        {
            await writer.WriteAsync(i, cancellationToken);

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
    catch (Exception ex)
    {
        localException = ex;
    }

    writer.Complete(localException);
}
```

Server-to-client streaming hub methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Use this token to stop the server operation and release any resources if the client disconnects before the end of the stream.

Client-to-server streaming

A hub method automatically becomes a client-to-server streaming hub method when it accepts one or more objects of type ChannelReader<T> or IAsyncEnumerable<T>. The following sample shows the basics of reading streaming data sent from the client. Whenever the client writes to the ChannelWriter<T>, the data is written into the `ChannelReader` on the server from which the hub method is reading.

C# Copy

```csharp
public async Task UploadStream(ChannelReader<string> stream)
{
    while (await stream.WaitToReadAsync())
    {
        while (stream.TryRead(out var item))
        {
            // do something with the stream item
            Console.WriteLine(item);
        }
    }
}
```

An IAsyncEnumerable<T> version of the method follows.

Note

The following sample requires C# 8.0 or later.

C# Copy

```csharp
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        Console.WriteLine(item);
    }
}
```

.NET client

Server-to-client streaming

The `StreamAsync` and `StreamAsChannelAsync` methods on `HubConnection` are used to invoke server-to-client streaming methods. Pass the hub method name and arguments defined in the hub method to `StreamAsync` or `StreamAsChannelAsync`. The generic parameter on `StreamAsync<T>` and `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. An object of type `IAsyncEnumerable<T>` or

`ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

A `StreamAsync` example that returns `IAsyncEnumerable<int>`:

C# Copy

```csharp
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var stream = await hubConnection.StreamAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

await foreach (var count in stream)
{
    Console.WriteLine($"{count}");
}

Console.WriteLine("Streaming completed");
```

A corresponding `StreamAsChannelAsync` example that returns `ChannelReader<int>`:

C# Copy

```csharp
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

The `StreamAsChannelAsync` method on `HubConnection` is used to invoke a server-to-client streaming method. Pass the hub method name and arguments defined in the hub method to `StreamAsChannelAsync`. The generic parameter on `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. A `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client

C# Copy

```csharp
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

Client-to-server streaming

There are two ways to invoke a client-to-server streaming hub method from the .NET client. You can either pass in an `IAsyncEnumerable<T>` or a `ChannelReader` as an argument to `SendAsync`, `InvokeAsync`, or `StreamAsChannelAsync`, depending on the hub method invoked.

Whenever data is written to the `IAsyncEnumerable` or `ChannelWriter` object, the hub method on the server receives a new item with the data from the client.

If using an `IAsyncEnumerable` object, the stream ends after the method returning stream items exits.

Note

The following sample requires C# 8.0 or later.

C# Copy

```csharp
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
    //After the for loop has completed and the local function exits the
stream completion will be sent.
}

await connection.SendAsync("UploadStream", clientStreamData());
```

Or if you're using a `ChannelWriter`, you complete the channel with `channel.Writer.Complete()`:

C# Copy

```
var channel = Channel.CreateBounded<string>(10);
await connection.SendAsync("UploadStream", channel.Reader);
await channel.Writer.WriteAsync("some data");
await channel.Writer.WriteAsync("some more data");
channel.Writer.Complete();
```

JavaScript client

Server-to-client streaming

JavaScript clients call server-to-client streaming methods on hubs with `connection.stream`. The `stream` method accepts two arguments:

- The name of the hub method. In the following example, the hub method name is `Counter`.
- Arguments defined in the hub method. In the following example, the arguments are a count for the number of stream items to receive and the delay between stream items.

`connection.stream` returns an `IStreamResult`, which contains a `subscribe` method. Pass an `IStreamSubscriber` to `subscribe` and set the `next`, `error`, and `complete` callbacks to receive notifications from the `stream` invocation.

JavaScript Copy

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;

            document.getElementById("messagesList").appendChild(li);
        },
});
```

To end the stream from the client, call the `dispose` method on the `ISubscription` that's returned from the `subscribe` method. Calling this method causes cancellation of the `CancellationToken` parameter of the Hub method, if you provided one.

Client-to-server streaming

JavaScript clients call client-to-server streaming methods on hubs by passing in a `Subject` as an argument to `send`, `invoke`, or `stream`, depending on the hub method invoked. The `Subject` is a class that looks like a `Subject`. For example in RxJS, you can use the Subject class from that library.

JavaScript Copy

```javascript
const subject = new signalR.Subject();
yield connection.send("UploadStream", subject);
var iteration = 0;
const intervalHandle = setInterval(() => {
    iteration++;
    subject.next(iteration.toString());
    if (iteration === 10) {
        clearInterval(intervalHandle);
        subject.complete();
    }
}, 500);
```

Calling `subject.next(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `subject.complete()`.

Java client

Server-to-client streaming