Bonus Chapter 3

WebHooks and SignalR

WHAT'S IN THIS CHAPTER?

- Using WebSockets
- Overview of SignalR
- Creating a SignalR hub
- Creating a SignalR client with HTML and JavaScript
- Creating a SignalR .NET client
- Using groups with SignalR
- Overview of WebHooks
- Creating WebHook receivers for GitHub and Dropbox

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The Wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `SignalRAndWebHooks`.

The code for this chapter is divided into the following major examples:

- WebSocket Server and Client
- Chat Server using SignalR
- Windows App Chat Client using SignalR
- SaaS WebHooks Receiver Sample

OVERVIEW

With .NET you can use events to get notifications. You can register an event handler method with an event, also known as subscribing to events, and as soon as the event is fired from another place, your method gets invoked. Events cannot be used with web applications.

Previous chapters have covered a lot about web applications and web services. What was common with these applications and services is that the request was always started from the client application. The client makes an HTTP request and receives a response.

What if the server has some news to tell? There's nothing like events that you can subscribe to, or are there? With the web technologies you've seen so far, this can be resolved by the client polling for new information. The client has to make a request to the server to ask whether new information is available. Depending on the request interval defined, this way of communication results in either a high load of requests on the network that just result in "no new information is available," or the client misses actual information and when asking for new information receives information that is already old.

If the client is itself a web application, the direction of the communication can be turned around, and the server can send messages to the client. This is how WebHooks work.

With clients behind a firewall, using the HTTP protocol there's no way for the server to initiate a connection to the client. The connection always needs to be started from the client side. Because HTTP connections are stateless, and clients often can't connect to ports other than port 80, WebSockets can help. WebSockets are initiated with an HTTP request, but they're upgraded to a WebSocket connection where the connection stays open. Using the WebSockets protocol, the server can send information to the client over the open connection as soon as the server has new information.

SignalR is an ASP.NET Core web technology that offers an easy abstraction over WebSockets. Using SignalR is a lot easier than programming using the sockets interface, and you get more features right out of the box.

> **NOTE** *At the time of this writing, both SignalR and WebHooks are in a preview state. Check the book's github site for updates on the samples as SignalR is released with ASP.NET Core.*

WebHooks is a technology that is offered by many SaaS (Software as a Service) providers. You can register with such a provider, provide a public Web API to the service provider, and this way the service provider can call back as soon as new information is available.

This chapter starts with a simple WebSocket client and server and covers both SignalR and WebHooks. As these technologies complement each other, they can also be used in combination.

WEBSOCKETS

Before getting into SignalR, let's get at the foundation—creating a server and client using .NET Core. Using WebSockets, the communication starts with an HTTP request:

```
GET /chat HTTP/1.1
Host: sampleserver.cninnovation.com
Upgrade: websocket
Connection: Upgrade
...
```

The server answers with a switch of protocols:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
...
```

From there on, the communication continues using WebSockets. This makes it possible to communicate across firewalls that usually have outgoing ports 80 and 443 enabled for communication.

WebSockets Server

The server application is started as an empty ASP.NET Core Web project. The NuGet package `Microsoft.AspNetCore.WebSockets` is already included with the metapackage `Microsoft.AspNetCore.All`, so no additional package is needed.

With the custom protocol used to communicate via WebSockets, the client needs to send the text `REQUESTMESSAGES:` followed by a term about the message to receive. The server answers with any number of messages containing this message term, and `EOS` (end of sequence) when no longer messages for this term are sent. For closing the communication, the client can send the text `SERVERCLOSE`, which tells the server to close the communication.

In the `Startup` class, to enable the middleware for `WebSockets`, you need to invoke the `UseWebSockets` extension method for the `IApplicationBuilder` (code file `WebSocketsSample/WebSocketsServer/Startup.cs`):

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILoggerFactory loggerFactory)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }

    app.UseWebSockets();
    //...
  }
}
```

> **NOTE** *Middleware is explained in Chapter 30, "ASP.NET Core."*

Communication via WebSockets is implemented with custom middleware. The custom middleware is added to the request pipeline when the URL path starts with `/samplesockets`. In the middleware implementation, a check is done to see whether the HTTP context is a WebSocket request. The `HttpContext` defines the property `WebSockets` that returns a `WebSocketManager`. Using the `WebSocketManager`, you can verify whether the request was a WebSocket request with the property `IsWebSocketRequest`. You can also check for WebSocket subprotocols that have been requested. Here, a custom protocol is used, which means subprotocols are not needed. If the request is a WebSocket request, the method `AcceptWebSocketAsync` transitions the request to a WebSocket connection, and now the communication using WebSockets can continue by calling the method `SendMessagesAsync` (code file `WebSocketsSample/WebSocketsServer/Startup.cs`):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  ILoggerFactory loggerFactory)
{
  //...
  app.Map("/samplesockets", app2 =>
  {
```

```
    app2.Use(async (context, next) =>
    {
      if (context.WebSockets.IsWebSocketRequest)
      {
        var webSocket = await context.WebSockets.AcceptWebSocketAsync();
        await SendMessagesAsync(context, webSocket,
          loggerFactory.CreateLogger("SendMessages"));
      }
      else
      {
        await next();
      }
    });
  });

  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Web Sockets sample");
  });
}
```

The implementation of the `SendMessagesAsync` method sends and receives messages via the `WebSocket` that is received in one parameter. After a message is received that starts the message content `REQUESTMESSAGES:`, the server loops and returns 10 messages with a delay of 1 second using the same connection. With the last message sent, the string `EOS` is added to the message content.

Sending messages via `WebSockets` is done with the `SendAsync` method. `SendAsync` uses an `ArraySegment` that wraps the byte array. With the `WebSocketMessageType`, you define whether to send text or binary data. The third parameter marks the end of the message. If the message is larger than the buffer, you need to set this parameter only with the last `SendAsync`, where the last part of the message is set to `true`, and you mark the other parts of the message with the `SendAsync` as incomplete. The last parameter allows you to send a cancellation token, which allows you to cancel the invocation before the timeout occurs. When the request `SERVERCLOSE` is received, the communication is closed by invoking the `CloseAsync` method of the `WebSocket` (code file `WebSocketsSample/WebSocketsServer/Startup.cs`):

```
private async Task SendMessagesAsync(HttpContext context, WebSocket webSocket,
  ILogger logger)
{
  var buffer = new byte[4096];
  WebSocketReceiveResult result = await webSocket.ReceiveAsync(
    new ArraySegment<byte>(buffer), CancellationToken.None);
  while (!result.CloseStatus.HasValue)
  {
    if (result.MessageType == WebSocketMessageType.Text)
    {
      string content = Encoding.UTF8.GetString(buffer, 0, result.Count);
      if (content.StartsWith("REQUESTMESSAGES:"))
      {
        string message = content.Substring("REQUESTMESSAGES:".Length);
        for (int i = 0; i < 10; i++)
        {
          string messageToSend = $"{message} - {i}";
          if (i == 9)
          {
            messageToSend += ";EOS"; // send end of sequence to not let the
                                     // client wait for another message
          }
          byte[] sendBuffer = Encoding.UTF8.GetBytes(messageToSend);
          await webSocket.SendAsync(new ArraySegment<byte>(sendBuffer),
            WebSocketMessageType.Text, endOfMessage: true,
            CancellationToken.None);
```

```
            logger.LogDebug("sent message {0}", messageToSend);
            await Task.Delay(1000);
          }
        }

        if (content.Equals("SERVERCLOSE"))
        {
          await webSocket.CloseAsync(WebSocketCloseStatus.NormalClosure,
            "Bye for now", CancellationToken.None);
          logger.LogDebug("client sent close request, socket closing");
          return;
        }
        else if (content.Equals("SERVERABORT"))
        {
          context.Abort();
        }
      }

      result = await webSocket.ReceiveAsync(buffer, CancellationToken.None);
    }
  }
```

> **NOTE** *The invocation of the* `SendAsync` *method uses named arguments with a middle parameter. Named arguments are discussed in Chapter 3, "Objects and Types." Using non-trailing named arguments requires C# 7.2. Cancellation tokens are covered in Chapter 21, "Tasks and Parallel Programming."*

WebSockets Client

The WebSockets client is a .NET Core console app. This app uses these namespaces:

`System`

`System.Net.WebSockets`

`System.Text`

`System.Threading`

`System.Threading.Tasks`

This application type needs to initiate a connection to the WebSockets server using `ws:` for the protocol and the path `samplesockets` as defined by the URL mapping in the server (code file `WebSocketsSample/WebSocketClient/Program.cs`):

```
static async Task Main()
{
  Console.WriteLine("Client - wait for server");
  Console.ReadLine();
  await InitiateWebSocketCommunication("ws://localhost:6295/samplesockets");
  Console.WriteLine("Program end");
  Console.ReadLine();
}
```

The communication is started by creating a `ClientWebSocket` and invoking the method `ConnectAsync`. From there (with the server invoking `AcceptWebSocketAsync`), communication flows. The method `SendAndReceiveAsync` is used for the main flow. After this method completes, the client sends a message containing the `SERVERCLOSE` text, where the server initiates a close. The client receives the

CloseStatus and CloseStatusDescription from the server with information about the close (code file WebSocketsSample/WebSocketClient/Program.cs):

```
static async Task InitiateWebSocketCommunication(string address)
{
  try
  {
    var webSocket = new ClientWebSocket();
    await webSocket.ConnectAsync(new Uri(address), CancellationToken.None);

    await SendAndReceiveAsync(webSocket, "A");
    await SendAndReceiveAsync(webSocket, "B");
    await webSocket.SendAsync(
      new ArraySegment<byte>(Encoding.UTF8.GetBytes("SERVERCLOSE")),
      WebSocketMessageType.Text, endOfMessage: true, CancellationToken.None);
    var buffer = new byte[4096];
    var result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
      CancellationToken.None);
    Console.WriteLine($"received for close: {result.CloseStatus} " +
      $"{result.CloseStatusDescription} " +
      $"{Encoding.UTF8.GetString(buffer, 0, result.Count)}");
    await webSocket.CloseAsync(WebSocketCloseStatus.NormalClosure, "Bye",
      CancellationToken.None);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

In the method SendAndReceiveAsync, the client first sends the message containing the text REQUESTMESSAGES: and waits to receive any number of messages until the EOS string is contained (code file WebSocketsSample/WebSocketClient/Program.cs):

```
static async Task SendAndReceiveAsync(WebSocket webSocket, string term)
{
  byte[] data = Encoding.UTF8.GetBytes($"REQUESTMESSAGES:{term}");
  var buffer = new byte[4096];

  await webSocket.SendAsync(new ArraySegment<byte>(data),
    WebSocketMessageType.Text, endOfMessage: true, CancellationToken.None);
  WebSocketReceiveResult result;
  bool sequenceEnd = false;
  do
  {
    result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
      CancellationToken.None);
    string dataReceived = Encoding.UTF8.GetString(buffer, 0, result.Count);
    Console.WriteLine($"received {dataReceived}");
    if (dataReceived.Contains("EOS"))
    {
      sequenceEnd = true;
    }
  } while (!(result?.CloseStatus.HasValue ?? false) && !sequenceEnd);
}
```

When you start both the server and the client, you can see the connection stays opened, and the server sends messages to the client after the client requests:

```
Client - wait for server
received A - 0
received A - 1
received A - 2
```

```
received A - 3
received A - 4
received A - 5
received A - 6
received A - 7
received A - 8
received A - 9;EOS
received B - 0
received B - 1
received B - 2
received B - 3
received B - 4
received B - 5
received B - 6
received B - 7
received B - 8
received B - 9;EOS
received for close: NormalClosure Bye for now
Program end
```

> **NOTE** *When you start the client and server, pay attention to change the port number the client needs to connect to the port number of your server.*

A SIMPLE CHAT USING SIGNALR

With your application, you can use WebSockets directly or use an abstraction layer such as SignalR, which means you don't need to deal with sockets. You have an easy way to return answers to all clients or a group of clients.

The first SignalR sample application is a chat application, which is easy to create with SignalR. With this application, multiple clients can be started to communicate with each other via the SignalR hub (see Figure BC3-1). When one of the client applications sends a message, all the connected clients receive this message in turn.

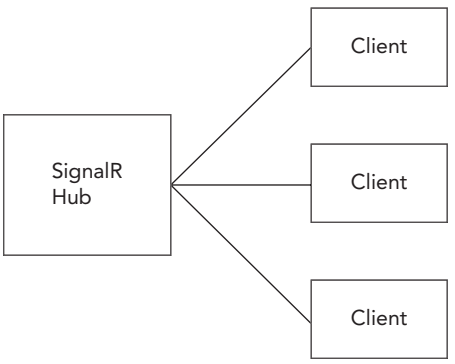

FIGURE BC3-1

The server application is started as an empty ASP.NET Core web application. One of the clients is created with HTML and JavaScript, and the other client application is a Windows app that uses the Universal Windows Platform.

Creating a Hub

The empty ASP.NET Core web project is named `ChatServer`. After you create the project, add a new item and select SignalR Hub class). Adding this item also adds the NuGet packages that are needed server side.

The main functionality of SignalR is defined with the hub. The hub is indirectly invoked by the clients, and in turn the clients are called. The class `ChatHub` derives from the base class `Hub` to get the needed hub functionality. The method `Send` is defined to be invoked by the client applications sending a message to the other clients. You can use any method name with any number of parameters. The client code just needs to match the method name as well as the parameters. To send a message to the clients, the `Clients` property of the `Hub` class is used. The `Clients` property returns an object that implements the interface `IHubCallerClients`. This interface allows you to send messages to specific clients or to all connected clients. To return a message to just a single client, you can use the `Client` method to pass a connection identifier. The sample code sends a message to all clients using the `All` property. The `All` property returns an `IClientProxy`. `IClientProxy` defines the method `InvokeAsync` to invoke a method within the client. The method invoked is the first parameter, which is the method name. The `InvokeAsync` method is overloaded to allow passing up to 10 arguments to the client method. In case you have more than 10, an overload allows passing an object array. With the sample code, the method name defined is `BroadcastMessage`, and two string parameters—the `name` and the `message`—are passed to this method (code file `SignalRSample/ChatServer/Hubs/ChatHub.cs`):

```
public class ChatHub: Hub
{
  public void Send(string name, string message)
  {
    Clients.All.BroadcastMessage(name, message);
  }
}
```

To use SignalR, the interfaces for SignalR need to be registered with the dependency injection container. You do this in the `ConfigureServices` method of the `Startup` class—invoking the `AddSignalR` extension method for the `IServiceCollection` interface (code file `SignalRSample/ChatServer/Startup.cs`):

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddSignalR();
  }
  //...
}
```

Next, you need to add the middleware for SignalR to invoke the `UseSignalR` extension method for the `IApplicationBuilder` interface. The parameter of type `HubRouteBuilder` allows specifying the route for SignalR. With an overload of the method, you can also specify options for the underlying WebSockets (code file `SignalRSample/ChatServer/Startup.cs`):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.UseStaticFiles();

  app.UseSignalR(routes =>
  {
    routes.MapHub<ChatHub>("chat");
  });
```

```
  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("SignalR Sample");
  });
}
```

> **NOTE** *For serving the HTML client from the same web as the SignalR server, the extension method* `UseStaticFiles` *needs to be added to the middleware pipeline.*

Creating a Client with HTML and JavaScript

The previous version of SignalR included a JavaScript library with jQuery extensions. At that time, nearly every website was using jQuery to access the DOM elements of the HTML page. Nowadays, jQuery is just one of many scripting libraries used. That's why SignalR is now built without jQuery.

The SignalR scripting libraries can be retrieved from the NPM server. In case you don't have the npm command-line utility already installed, you can download it from `npmjs.com`.

You can create an NPM configuration file using Visual Studio by adding the item npm Configuration File to the project, or by using the command line

```
> npm init
```

The SignalR client library can be installed to the project using

```
> npm install @aspnet/signalr-client
```

This downloads the library and adds it to the file `package.json` (package file `SignalRSample/ChatServer/package.json`):

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {},
  "dependencies": {
    "@aspnet/signalr-client": "^1.0.0-alpha2-final"
  }
}
```

The downloaded JavaScript libraries from the `node_modules/@aspnet/signalr-client/dist/browser` folder need to be copied to the `wwwroot/js` folder so the libraries are available when you run the application.

For the HTML client, two input fields and a button are defined to allow the user to enter a name and a message and then click a button so send the message to the SignalR server. The messages received will be displayed in the output element (code file `SignalRSample/ChatServer/wwwroot/ChatWindow.html`):

```
<label for="name">Name:</label>
<input type="text" id="name" />
<br />
<label for="message">Message:</label>
<input type="text" id="message" />
<br />
<input id="sendButton" type="button" value="send" />
<p />
<output id="output"></output>
```

The first script element references the JavaScript file from the SignalR library. Remember to use the min file for production. When the DOM tree of the HTML file is loaded, a connection to the Chat server is created.

Using `connection.on`, you define what should happen when a message arrives from the SignalR server. The first parameter is the name of the method used when calling the proxy in the server except the casing changes from C# Pascal casing to JavaScript camel casing. With the second parameter, a function is defined that has the same number of parameters as are sent from the server. When a message arrives, the content of the output element changes to include this message. After registering to this event, the connection to the SignalR server is started, which invokes the `start` function. When the connection completes successfully, the `then` function defines what's next. Here, an event listener is assigned to the `click` event of the button to send the message to the SignalR server. The method that is invoked with the SignalR server is defined with the first parameter of the invoke function—`send` in the sample code—that's the same name as the method is defined in the `ChatHub` (just the casing differs). Again, the same number of arguments is used (code file `SignalRSample/ChatServer/wwwroot/ChatWindow.html`):

```
<script src="js/signalr-client-1.0.0-alpha2-final.js"></script>
<script>
  document.addEventListener("DOMContentLoaded", function () {
    let connection = new signalR.HubConnection('/chat');
    connection.on('broadcastMessage', (name, message) => {
      console.log(message);
      document.getElementById('output').innerHTML +=
        `message from ${name}: ${message}<br />`;
    });

    connection.start().then(function () {
      document.getElementById('sendButton')
        .addEventListener('click', function () {
          let name = document.getElementById('name').value;
          let message = document.getElementById('message').value;

          connection.invoke('send', name, message);
        });
    });
  });
  </script>
```

> **NOTE** *The sample code makes use of the event* `DOMContentLoaded` *instead of* `window.onload`*.* `window.onload` *is fired when the complete page is loaded, including the images and CSS files.* `DOMContentLoaded` *is fired before that—when the DOM tree is ready.* `DOMContentLoaded` *is available with all modern browsers, and with Internet Explorer starting with IE9.*

When you run the application, you can open multiple browser windows—even using different browsers—you can enter names and messages for a chat (see Figure BC3-2).

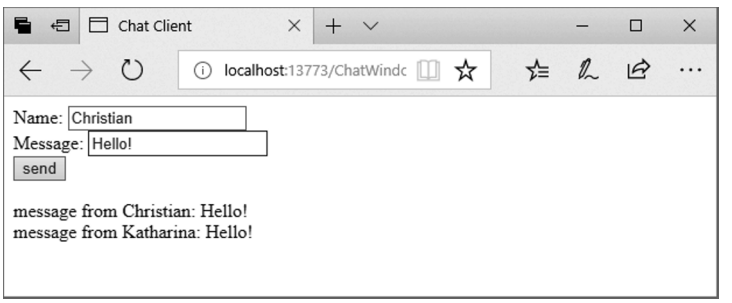

FIGURE BC3-2

When you use the Microsoft Edge Developer Tools (press F12 while Microsoft Edge is open) you can use Network Monitoring to see the upgrade from the HTTP protocol to the WebSocket protocol, as shown in Figure BC3-3.

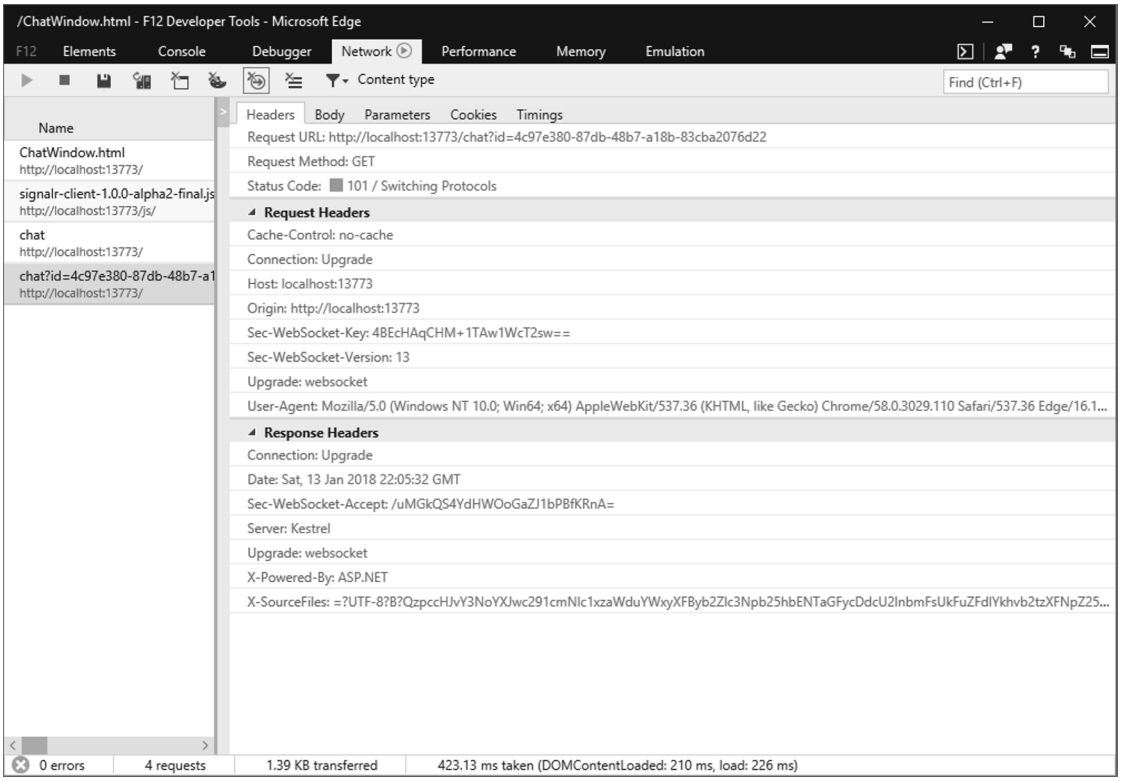

FIGURE BC3-3

Creating SignalR .NET Clients

The sample .NET client application to use the SignalR server is a Windows app. The functionality is similar to the HTML/JavaScript application shown earlier.

The user interface of the UWP application defines two `TextBox`, two `Button`, and one `ListBox` element to enter the name and message, to connect to the service hub, and to show a list of received messages (code file `SignalRSample/WindowsAppChatClient/MainPage.xaml`):

```
<TextBox Header="Name" Text="{x:Bind ViewModel.Name, Mode=TwoWay}"
  Grid.Row="0" Grid.Column="0" />
<TextBox Header="Message" Text="{x:Bind ViewModel.Message, Mode=TwoWay}"
  Grid.Row="1" Grid.Column="0" />
<StackPanel Orientation="Vertical" Grid.Column="1" Grid.RowSpan="2">
  <Button Content="Connect" Command="{x:Bind ViewModel.ConnectCommand}"  />
  <Button Content="Send"
    Command="{x:Bind ViewModel.SendCommand, Mode=OneTime}" />
</StackPanel>
<ListBox ItemsSource="{x:Bind ViewModel.Messages, Mode=OneWay}" Grid.Row="2"
  Grid.ColumnSpan="2" Margin="12" />
```

The `ApplicationServices` class is implemented as a singleton to create the dependency injection (DI) container and register services as well as view models (code file `SignalRSample/WindowsAppChatClient/ApplicationServices.cs`):

```
public class ApplicationServices
{
  private ApplicationServices()
  {
    var services = new ServiceCollection();
    services.AddTransient<ChatViewModel>();
    services.AddTransient<GroupChatViewModel>();
    services.AddSingleton<IDialogService, DialogService>();
    services.AddSingleton<UrlService>();
    services.AddLogging();
    ServiceProvider = services.BuildServiceProvider();
  }

  private static ApplicationServices _instance;
  private static object _instanceLock = new object();
  private static ApplicationServices GetInstance()
  {
    lock (_instanceLock)
    {
      return _instance ?? (_instance = new ApplicationServices());
    }
  }

  public static ApplicationServices Instance => _instance ?? GetInstance();

  public IServiceProvider ServiceProvider { get; }
}
```

The `UrlService` class that is registered with the DI container contains the URL addresses to the chat server. You need to change the `BaseUri` to the address that is shown when starting your SignalR server (code file `SignalRSample/WindowsAppChatClient/Services/UrlService.cs`):

```
public class UrlService
{
  private const string BaseUri = "http://localhost:2952/";
  public string ChatAddress => $"{BaseUri}/chat";
  public string GroupAddress => $"{BaseUri}/groupchat";
}
```

Within the code-behind file of the view, the `ChatViewModel` is assigned to the `ViewModel` property using the DI container (code file `SignalRSample/WindowsAppChatClient/MainPage.xaml.cs`):

```
public partial class MainWindow: Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  public ChatViewModel ViewModel { get; } =
    ApplicationServices.Instance.ServiceProvider.GetService<ChatViewModel>();
}
```

> **NOTE** *Windows apps are covered in detail in Chapter 33, "Windows Apps." The Model-View-View Model (MVVM) pattern is explained in Chapter 34, "Patterns with XAML Apps."*

The hub-specific code is implemented in the class `ChatViewModel`. First, have a look at the bound properties and commands. The property `Name` is bound to enter the chat name, the `Message` property to enter the message. The `ConnectCommand` property maps to the `OnConnect` method to initiate the connection to the server; the `SendCommand` property maps to the `OnSendMessage` method to send a chat message (code file `SignalRSample/WindowsAppChatClient/ViewModels/ChatViewModel.cs`):

```
public sealed class ChatViewModel
{
  private readonly IDialogService _dialogService;
  private readonly UrlService _urlService;

  public ChatViewModel(IDialogService dialogService, UrlService urlService)
  {
    _dialogService = dialogService;
    _urlService = urlService;

    ConnectCommand = new RelayCommand(OnConnect);
    SendCommand = new RelayCommand(OnSendMessage);
  }

  public string Name { get; set; }
  public string Message { get; set; }

  public ObservableCollection<string> Messages { get; } =
    new ObservableCollection<string>();
  public RelayCommand SendCommand { get; }
  public RelayCommand ConnectCommand { get; }
  //...
}
```

The `OnConnect` method initiates the connection to the server. You can create a `HubConnection` with the `HubConnectionBuilder`. This builder uses a fluent API for its configuration. In the sample code, you can see the URL to the server first configured with the `WithUrl` method, followed by the `WithLogger` method to specify a logging provider. After the configuration is done, the `Build` method of the `HubConnectionBuilder` creates a `HubConnection`. When you use these configuration methods, you can also configure the protocol to use. SignalR supports JSON (`WithJsonProtocol`) and MessagePack (`WithMessagePackProtocol`) out of the box, but you can also create and use other providers. To register with messages that are returned from the server, the `On` method is invoked. The first parameter passed to the `On` method defines the method name that is called by the server; the second parameter defines a delegate to the method that is invoked. The method `OnMessageReceived` has the parameters specified with the generic parameter arguments of the `On` method: two strings. To finally initiate the connection, the `StartAsync` method on the `HubConnection` instance is invoked to connect to the SignalR server (code file `SignalRSample/WindowsAppChatClient/ViewModels/ChatViewModel.cs`):

```
private HubConnection _hubConnection;

public async void OnConnect()
{
  await CloseConnectionAsync();
  _hubConnection = new HubConnectionBuilder()
    .WithUrl(_urlService.ChatAddress)
    .WithLogger(loggerFactory =>
    {
    })
    .Build();

  _hubConnection.Closed += HubConnectionClosed;
  _hubProxy.On<string, string>("BroadcastMessage", OnMessageReceived);

  try
```

```
    {
      await _hubConnection.StartAsync();
    }
    catch (HttpRequestException ex)
    {
      _dialogService.ShowMessage(ex.Message);
    }
    _dialogService.ShowMessage("client connected");
}
```

> **NOTE** *The MessagePack protocol is more compact than JSON and is designed for efficiency on the wire.*

Sending messages to SignalR requires only calls to the `SendAsync` method of the `HubConnection`. The first parameter is the name of the method that should be invoked by the server; the following parameters are the parameters of the method on the server (code file `SignalRSample/WindowsAppChatClient/ViewModels/ChatViewModel.cs`):

```
Public async void OnSendMessage()
{
  try
  {
    _hubConnection.SendAsync("Send", Name, Message);
  }
  catch (Exception ex)
  {
    await _dialogService.ShowMessageAsync(ex.Message);
  }
}
```

When receiving a message, the `OnMessageReceived` method is invoked. Because this method is invoked from a background thread, you need to switch back to the UI thread that updates bound properties and collections (code file `WindowsAppChatClient/ViewModels/ChatViewModel.cs`):

```
public void OnMessageReceived(string name, string message)
{
  try
  {
    await CoreApplication.MainView.CoreWindow.Dispatcher
      .RunAsync(CoreDispatcherPriority.Normal, () =>
    {
      Messages.Add($"{name}: {message}");
    });
  }
  catch (Exception ex)
  {
    await _dialogService.ShowMessageAsync(ex.Message);
  }
}
```

When you run the application, you can receive and send messages from the Windows app client as shown in Figure BC3-4. You can also open the web page simultaneously and communicate between them.

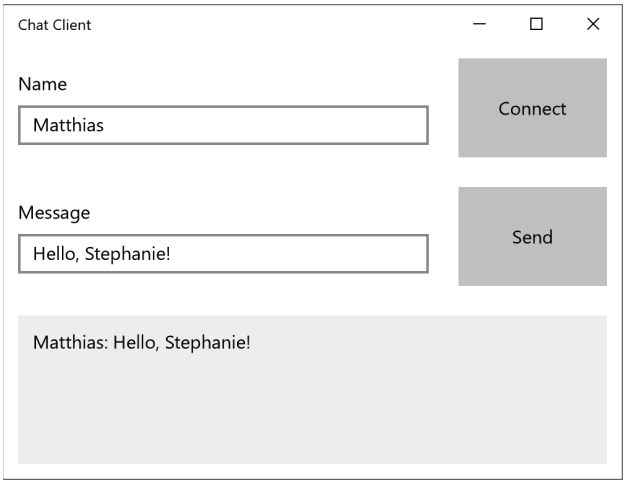

FIGURE BC3-4

GROUPING CONNECTIONS

Usually you don't want to communicate among all clients, but you instead want to communicate among a group of clients. There's support out of the box for such a scenario with SignalR.

In this section, you add another chat hub with grouping functionality and have a look at other options that are possible using SignalR hubs. The Windows app client application is extended to enter groups and send a message to a selected group.

Extending the Hub with Groups

To support a group chat, you create the class `GroupChatHub`. With the previous hub, you saw how to use the `InvokeAsync` method to define the message that is sent to the clients. Instead of using this method, you can also create a custom interface as shown in the following code snippet. This interface is used as a generic parameter with the base class `Hub` (code file `SignalRSample/ChatServer/Hubs/GroupChatHub.cs`):

```
public interface IGroupClient
{
  void MessageToGroup(string groupName, string name, string message);
}

public class GroupChatHub: Hub<IGroupClient>
{
  //...
}
```

`AddGroup` and `LeaveGroup` are methods defined to be called by the client. Registering the group, the client sends a group name with the `AddGroup` method. The `Hub` class defines a `Groups` property where connections to groups can be registered. The `Groups` property of the generic `Hub` class returns `IGroupManager`. This interface defines two methods: `AddAsync` and `RemoveAsync`. Both methods need a group name and a

connection identifier to add or remove the specified connection to the group. The connection identifier is a unique identifier associated with a client connection. The client connection identifier—as well as other information about the client—can be accessed with the `Context` property of the `Hub` class. The following code snippet invokes the `AddAsync` method of the `IGroupManager` to register a group with the connection, and the `RemoveAsync` method to unregister a group (code file `SignalRSample/ChatServer/Hubs/GroupChatHub.cs`):

```
public Task AddGroup(string groupName) =>
  Groups.AddAsync(Context.ConnectionId, groupName);

public Task LeaveGroup(string groupName) =>
  Groups.RemoveAsync(Context.ConnectionId, groupName);
```

> **NOTE** *The* `Context` *property of the* `Hub` *class returns an object of type* `HubCallerContext`. *With this class, you can not only access the connection identifier associated with the connection, but you can access other information about the client, such as the user if the user is authorized.*

Invoking the `Send` method—this time with three parameters including the group—sends information to all connections that are associated with the group. The `Clients` property is now used to invoke the `Group` method. The `Group` method accepts a group string to send the `MessageToGroup` message to all connections associated with the group name. With an overload of the `Group` method you can add connection IDs that should be excluded. Because the `Hub` implements the interface `IGroupClient`, the `Groups` method returns the `IGroupClient`. This way, the `MessageToGroup` method can be invoked using compile-time support (code file `SignalRSample/ChatServer/Hubs/GroupChatHub.cs`):

```
public void Send(string group, string name, string message) =>
  Clients.Group(group).MessageToGroup(group, name, message);
```

Several other extension methods are defined to send information to a list of client connections. You've seen the `Group` method to send messages to a group of connections that's specified by a group name. With this method, you can exclude client connections. For example, the client who sent the message might not need to receive it. The `Groups` method accepts a list of group names where a message should be sent to. You've already seen the `All` property to send a message to all connected clients. Methods to exclude sending the message to the caller are `OthersInGroup` and `OthersInGroups`. These methods send a message to one specific group excluding the caller, or a message to a list of groups excluding the caller.

You can also send messages to a customized group that's not based on the built-in grouping functionality. Here, it helps to override the methods `OnConnectedAsync` and `OnDisconnectedAsync`. The `OnConnectedAsync` method is invoked every time a client connects; the `OnDisconnectedAsync` method is invoked when a client disconnects. Within these methods, you can access the `Context` property of the `Hub` class to access client information as well as the client-associated connection ID. Here, you can write the connection information to a shared state to have your server scalable using multiple instances, accessing the same shared state. You can also select clients based on your own business logic or implement priorities when sending messages to privilege specific clients.

```
public override Task OnConnectedAsync() =>
  base.OnConnectedAsync();

public override Task OnDisconnectedAsync(Exception exception) =>
  base.OnDisconnected(exception);
```

Extending the Windows Client with Groups

After having the grouping functionality with the hub ready, you can extend the Windows app client application. For the grouping features, another XAML page associated with the `GroupChatViewModel` class is defined.

The `GroupChatViewModel` class defines some more properties and commands compared to the `ChatViewModel` defined earlier. The `NewGroup` property defines the group the user registers to. The `SelectedGroup` property defines the group that is used with the continued communication, such as sending a message to the group or leaving the group. The `SelectedGroup` property needs change notification to update the user interface on changing this property; that's why the `INotifyPropertyChanged` interface is implemented with the `GroupChatViewModel` class, and the `set` accessor of the property `SelectedGroup` fires a notification. Commands to join and leave the group are defined as well: the `EnterGroupCommand` and `LeaveGroupCommand` properties (code file `SignalRSample/WindowsAppChatClient/ViewModels/GroupChatViewModel.cs`):

```
public sealed class GroupChatViewModel: INotifyPropertyChanged
{
  private readonly IDialogService _dialogService;
  private readonly UrlService _urlService;

  public GroupChatViewModel(IDialogService dialogService,
    UrlService urlService)
  {
    _dialogService = dialogService;
    _urlService = urlService;

    ConnectCommand = new RelayCommand(OnConnect);
    SendCommand = new RelayCommand (OnSendMessage);
    EnterGroupCommand = new RelayCommand (OnEnterGroup);
    LeaveGroupCommand = new RelayCommand (OnLeaveGroup);
  }

  public event PropertyChangedEventHandler PropertyChanged;
  private void Set<T>(ref T item, T value,
    [CallerMemberName] string propertyName = null)
  {
    if (!EqualityComparer<T>.Default.Equals(item, value))
    {
      item = value;
      PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }
  }

  public string Name { get; set; }
  public string Message { get; set; }
  public string NewGroup { get; set; }
  private string _selectedGroup;
  public string SelectedGroup
  {
    get => _selectedGroup;
    set => Set(ref _selectedGroup, value);
  }

  public ObservableCollection<string> Messages { get; } =
    new ObservableCollection<string>();
  public ObservableCollection<string> Groups { get; } =
```

```
      new ObservableCollection<string>();
    public ICommand SendCommand { get; }
    public ICommand ConnectCommand { get; }
    public ICommand EnterGroupCommand { get; }
    public ICommand LeaveGroupCommand { get; }
    //...
}
```

The handler methods for the `EnterGroupCommand` and `LeaveGroupCommand` commands are shown in the following code snippet. Here, the `AddGroup` and `RemoveGroup` methods are called within the group hub (code file `SignalRSample/WindowsAppChatClient/ViewModels/GroupChatViewModel.cs`):

```
public async void OnEnterGroup()
{
  try
  {
    await _hubConnection.InvokeAsync("AddGroup", NewGroup);
    Groups.Add(NewGroup);
    SelectedGroup = NewGroup;
  }
  catch (Exception ex)
  {
    await _dialogService.ShowMessageAsync(ex.Message);
  }
}

public async void OnLeaveGroup()
{
  try
  {
    await _hubConnection.InvokeAsync("LeaveGroup", SelectedGroup);
    Groups.Remove(SelectedGroup);
  }
  catch (Exception ex)
  {
    _dialogService.ShowMessage(ex.Message);
  }
}
```

Sending and receiving the messages is very similar to the previous sample, with the difference that the group information is added now (code file `SignalRSample/WindowsAppChatClient/ViewModels/GroupChatViewModel.cs`):

```
public async void OnSendMessage()
{
  try
  {
    await _hubConnection.InvokeAsync("Send", SelectedGroup, Name, Message);
  }
  catch (Exception ex)
  {
    _dialogService.ShowMessage(ex.Message);
  }
}

public void OnMessageReceived(string group, string name, string message)
{
  try
```

```
{
  await CoreApplication.MainView.CoreWindow.Dispatcher
    .RunAsync(CoreDispatcherPriority.Normal, () =>
  {
    Messages.Add($"{group}-{name}: {message}");
  });
}
catch (Exception ex)
{
  await _dialogService.ShowMessageAsync(ex.Message);
}}
```

When you run the application, you can send messages for all groups that have been joined and see received messages for all registered groups, as shown in Figure BC3-5.

FIGURE BC3-5

ARCHITECTURE OF WEBHOOKS

WebHooks offer publish/subscribe functionality with web applications. That's the only similarity between WebHooks and SignalR. Otherwise, WebHooks and SignalR are very different and can take advantage of each other. Before discussing how they can be used together, let's get into an overview of WebHooks.

With WebHooks, an SaaS (Software as a Service) service can call into your website. You just need to register your site with the SaaS service. The SaaS service than calls your website (see Figure BC3-6). In your website, the *receiver controller* receives all messages from WebHooks senders and forwards it to the corresponding receiver. The *receiver* verifies security to check whether the message is from the registered sender, and then it forwards the message to the handler. The *handler* contains your custom code to process the request.

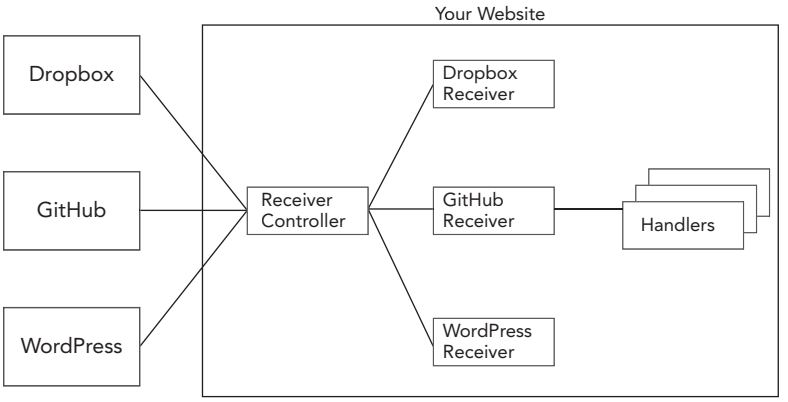

FIGURE BC3-6

Contrary to the SignalR technology, the sender and receiver are not always connected. The receiver just offers a service API that is invoked by the sender when needed. The receiver needs to be available on a public Internet address.

The beauty of WebHooks is the ease of use on the receiver side and the support it receives from many SaaS providers, such as Dropbox, Microsoft Azure, GitHub, WordPress, PayPal, Slack, SalesForce, and others.

Creating a sender is not as easy as creating a receiver, but there's also great support with ASP.NET Core. A sender needs a registration option for WebHook receivers, which is typically done using a Web UI. Of course, you can also create a Web API instead to register programmatically. With the registration, the sender receives a secret from the receiver together with the URL it needs to call into. This secret is verified by the receiver to only allow senders with this secret. As events occur with the sender, the sender fires a WebHook, which in reality involves invoking a web service of the receiver and passing (mostly) JSON information.

Microsoft's ASP.NET Core NuGet packages for WebHooks make it easy to implement receivers for different services by abstracting the differences. It's also easy to create the ASP.NET Core Web API service that verifies the secrets sent by the senders and forward the calls to custom handlers.

To see the ease of use and the beauty of WebHooks, a sample application is shown to create Dropbox and GitHub receivers. When creating multiple receivers, you see what's different between the providers and what functionality is offered by the NuGet packages. You can create a receiver to other SaaS providers in a similar manner.

CREATING DROPBOX AND GITHUB RECEIVERS

To create and run the Dropbox and GitHub receiver example, you need both a GitHub and a Dropbox account. With GitHub you need admin access to a repository. Of course, for learning WebHooks, it's fine to use just one of these technologies. What's needed with all the receivers, no matter what service you use, is for you to have a way to make your website publicly available—for example, by publishing to Microsoft Azure.

Dropbox (`http://www.dropbox.com`) offers a file store in the cloud. You can save your files and directories and share them with others. With WebHooks you can receive information about changes in your Dropbox storage—for example, you can be notified when files are added, modified, and deleted.

GitHub (`http://www.github.com`) offers source code repositories. .NET Core and ASP.NET Core are available in public repositories on GitHub, and so is the source code for this book (`http://www.github.com/ProfessionalCSharp/ProfessionalCSharp7`). With the GitHub WebHook you can receive

information about push events or about all changes to the repository, such as forks, updates to Wiki pages, issues, and more.

Creating a Web Application

Start by creating an ASP.NET Core Web Application named `WebHooksReceiver`. Select Web API with the ASP.NET Core templates.

Next, add the NuGet packages `Microsoft.AspNetCore.WebHooks.Receivers.Dropbox` and `Microsoft.AspNetCore.WebHooks.Receivers.GitHub`. These are the NuGet packages that support receiving messages from Dropbox and GitHub. With the NuGet package manager you'll find many more NuGet packages that support other SaaS services.

Configuring WebHooks for Dropbox and GitHub

You can create services for WebHooks like what you've seen when adding services in the previous chapters. Just make sure you don't add them to the `IServiceCollection`; add them to the result from the `AddMvc` method instead. The methods `AddDropboxWebHooks` and `AddGitHubWebHooks` are extension methods to `IMvcBuilder`, as they configure an ASP.NET Core MVC route and metadata that is used with the action methods of the controllers. The singleton entry for the `StorageQueueService` is a service writing to Azure Storage that will be used from the implementation of the WebHook controllers (code file `WebHooksReceiver/Startup.cs`):

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddMvc()
      .AddDropboxWebHooks()
      .AddGitHubWebHooks();

    services.AddSingleton<IStorageQueueService, StorageQueueService>();
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment)
    {
      app.UseDeveloperExceptionPage();
    }

    app.UseMvc();
  }
}
```

To only allow messages from the defined SaaS services, secrets are used. You can configure these secrets with the application settings. The key for the settings is predefined from the code in the NuGet packages. For Dropbox, you use the key `WebHooks:Dropbox:SecretKey:default`, and for GitHub you use `MWebHooks:GitHub:SecretKey:default`. A secret needs to be at least 15 characters long (configuration file `WebHooksReceiver/appsettings.json`):

```
{
  "WebHooks": {
    "GitHub": {
      "SecretKey": {
        "default": "tHisiStHesEcretkEyfOrgIthub"
      }
    },
    "Dropbox": {
```

```
      "SecretKey": {
        "default": "thIsisthEseCretkeYfoRdrOpbox"
      }
    }
  }
}
```

Implementing the Handler

The functionality of the WebHook is implemented in a controller with attributes for WebHooks. What can be done in this handler? You can write the information to a database, to a file, to invoke other services, and so on. Just be aware that the implementation shouldn't take too long—just a few seconds. In cases where the implementation takes too long, the sender might resend the request. For longer activities, it's best to write the information to a queue and work through the queue after the method is finished—for example, by using a background process.

For the sample application receiving an event, a message is written into the Microsoft Azure Storage queue. For using this queuing system, you need to create a Storage account at `http://portal.azure.com`. With the sample application, the Storage account is named `professionalcsharp`. For using Microsoft Azure Storage from ASP.NET Core, you can add the NuGet package `WindowsAzure.Storage` to the project.

After creating the Azure Storage account, open the portal and copy the account name and primary access key, and add this information to the configuration file (code file `WebHooksReceiver/appsettings.json`):

```
{
  "ConnectionStrings": {
    "AzureStorageConnectionString":
      "Add your Azure Storage connection string"
  }
}
```

> **NOTE** *Pay attention to not save your secrets to a public source code repository. As you're developing, you can write your secrets to User Secrets as shown in Chapter 24, "Security." In production where you store the secrets depends on the technology used. For example, with Microsoft Azure, you can use the Azure Key Vault to store your secret and read it by using the configuration provider* `Microsoft.Extensions.Configuration.AzureKeyVault`.

To send a message to the queue, you create the `StorageQueueService`. In the constructor, you inject the `IConfiguration` interface for reading the configuration from a configuration provider. In the implementation of the `StorageQueueService`, the `CloudStorageAccount` is used to connect to the storage account. From this account, the `CloudQueueClient` is used to access the queues in the storage account. Azure Storage offers queue, table, and blob storage. The property `QueueClient` returns the `CloudQueueClient`. With this client, a new queue is created (if it doesn't already exist), and messages are sent to the queue (code file `WebHooksReceiver/Services/StorageQueueservice.cs`):

```
public class StorageQueueService : IStorageQueueService
{
  private readonly IConfiguration _configuration;
  public StorageQueueService(IConfiguration configuration) =>
    _configuration = configuration;

  private CloudStorageAccount _storageAccount;
  public CloudStorageAccount StorageAccount =>
    _storageAccount ?? (_storageAccount =
      CloudStorageAccount.Parse(
        _configuration.GetConnectionString("AzureStorageConnectionString")));
```

```
    private CloudQueueClient _queueClient;
    public CloudQueueClient QueueClient =>
      _queueClient ?? (_queueClient =
        StorageAccount.CreateCloudQueueClient());

    public async Task WriteToQueueStorageAsync(string queueName, string message)
    {
      CloudQueue queue = QueueClient.GetQueueReference(queueName);
      await queue.CreateIfNotExistsAsync();
      await queue.AddMessageAsync(new CloudQueueMessage(message));
    }
  }
```

Next, let's get into the most important part of the WebHook implementation: the controllers for the Dropbox and GitHub events. For making a Dropbox controller, all you need to do is to add a `DropboxWebHook` attribute to an action method. The parameter with the JObject type contains the JSON message from Dropbox. The `DropboxWebHookAttribute` class derives from `WebHookAttribute`. This attribute offers the `Id` property, which you can use to define different configurations. When you specify the `Id` with the annotated action method, the action method only accepts messages for the specified configuration. The sample code doesn't supply the `Id` property, so every message received from Dropbox applies (code file `WebHooksReceiver/Controllers/DropboxController.cs`):

```
public class DropboxController : Controller
{
  private readonly IStorageQueueService _storageQueueService;
  public DropboxController(IStorageQueueService storageQueueService)
  {
    _storageQueueService = storageQueueService;
  }

  [DropboxWebHook()]
  public async Task<IActionResult> Dropbox(string id, JObject data)
  {
    if (!ModelState.IsValid)
    {
      return BadRequest(ModelState);
    }
    await _storageQueueService.WriteToQueueStorageAsync("dropbox",
      data.ToString());

    return Ok();
  }
}
```

With GitHub, the `GitHubWebHook` attribute defines some more properties. GitHub enables you to send the data in the format of an HTTP request that uses Form data; you can configure this with the GitHub website. The `GitHubWebHook` attribute can be configured to accept or not accept Form data using the `AcceptFormData` property. You can also specify the `EventName` property, like it is done with the `GitHubPushHandler` action method. This action method accepts only push events from GitHub (code file `WebHooksReceiver/Controllers/GitHubController.cs`):

```
[GitHubWebHook(AcceptFormData = false, EventName = "push")]
public async Task<IActionResult> GitHubPushHandler(string id, JObject data)
{
  if (!ModelState.IsValid)
  {
    return BadRequest(ModelState);
  }
  string message = $" id:{id}, push-event, data: {data.ToString()}";
```

```
    await _storageQueueService.WriteToQueueStorageAsync("github", message);

    return Ok();
  }
```

The action method that accepts all the events from GitHub is the method `GitHubHandler`. This just has the `GitHubWebHook` attribute applied without setting any property (code file `WebHooksReceiver/Controllers/GitHubController.cs`):

```
[GitHubWebHook()]
public async Task<IActionResult> GitHubHandler(string id, string eventName,
  JObject data)
{
  if (!ModelState.IsValid)
  {
    return BadRequest(ModelState);
  }
  string message = $" id:{id}, event:{eventName}, data: {data.ToString()}";
  await _storageQueueService.WriteToQueueStorageAsync("github", message);

  return Ok();
}
```

To have a fallback method that is invoked when no other WebHook action method handler applies, you can use the attribute `GeneralWebHook` (code file `WebHooksReceiver/Controllers/GitHubController.cs`):

```
[GeneralWebHook]
public async Task<IActionResult> FallbackHandler(string receiverName,
  string id, string eventName, JObject data)
{
  //...
}
```

With the implementation in a production scenario you can already read information from the JSON object and react accordingly. However, remember that you should do the work within the handler within a few seconds. Otherwise the service can resend the WebHook. This behavior is different based on the providers.

With the handlers implemented, you can build the project and publish the application to Microsoft Azure. You can publish directly from the Solution Explorer in Visual Studio. Select the project, choose the Publish context menu, and select a Microsoft Azure App Service target.

After publishing you can configure Dropbox and GitHub. For these configurations, the site already needs to be publicly available.

Configuring the Application with Dropbox and GitHub

To enable WebHooks with Dropbox, you need to create an app in the Dropbox App Console at `https://www.dropbox.com/developers/apps`, as shown in Figure BC3-7.

To receive WebHooks from Dropbox, you need to register the public URI of your website. When you host the site with Microsoft Azure, the host name is `<hostname>.azurewebsites.net`. The service of the receiver listens at `/api/webhooks/incoming/provider`—for example, with Dropbox at `https://professionalcsharp.azurewebsites.net/api/webhooks/incoming/dropbox`. In case you registered more than one secret, other than the URIs of the other secrets, add the secret key to the URI, such as `/api/webhooks/incoming/dropbox/dropboxsecretkey1`.

Dropbox verifies a valid URI by sending a challenge that must be returned. You can try that out with your receiver that is configured for Dropbox to access the URI `hostname/api/webhooks/incoming/dropbox/?challenge=12345`, which should return the string `12345`.

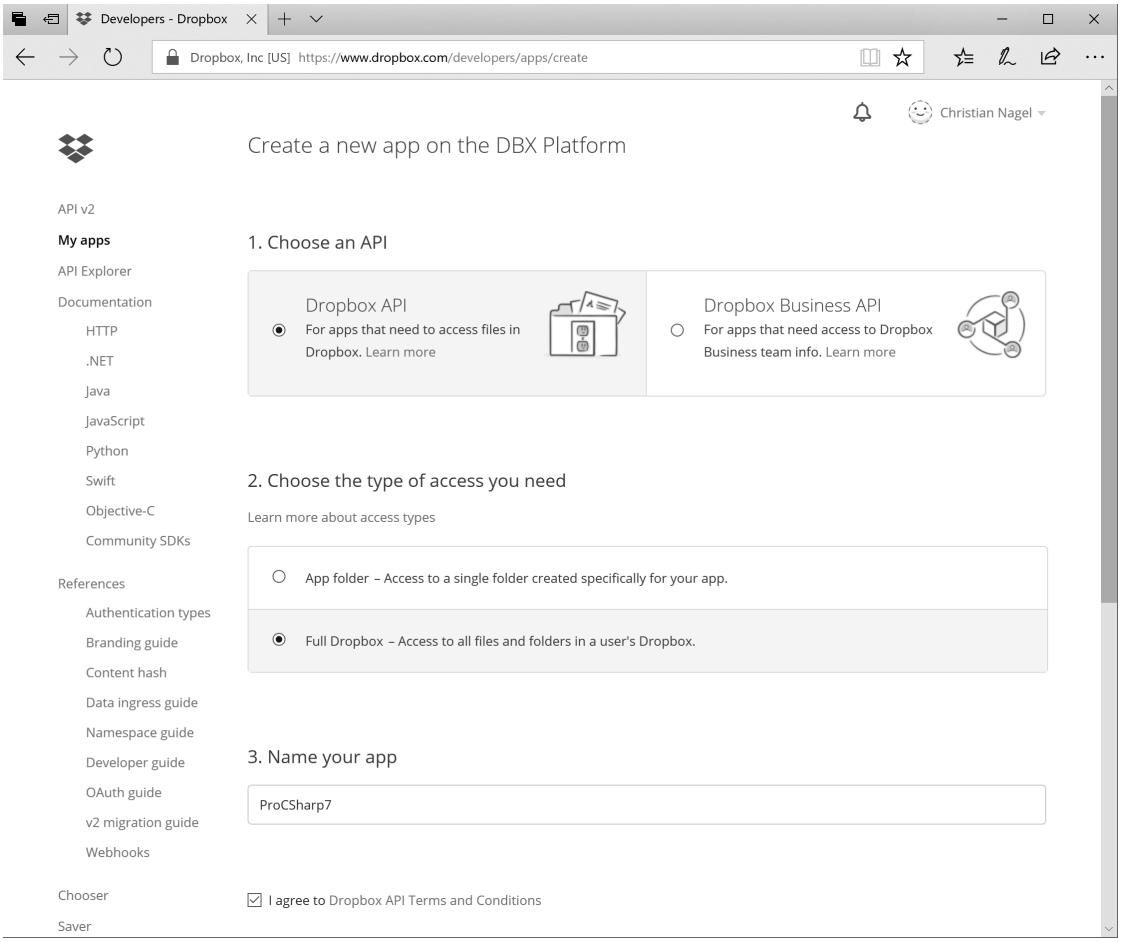

FIGURE BC3-7

To enable WebHooks with GitHub, open the Settings of a GitHub repository (see Figure BC3-8). There you need to add a payload link, which is `http://<hostname>/api/webhooks/incoming/github` with this project. Also, don't forget to add the secret, which must be the same as defined in the configuration file. With the GitHub configuration, you can select either `application/json` or Form-based `application/x-www-form-urlencoded` content from GitHub. With the events, you can select to receive just push events, all events, or select individual events.

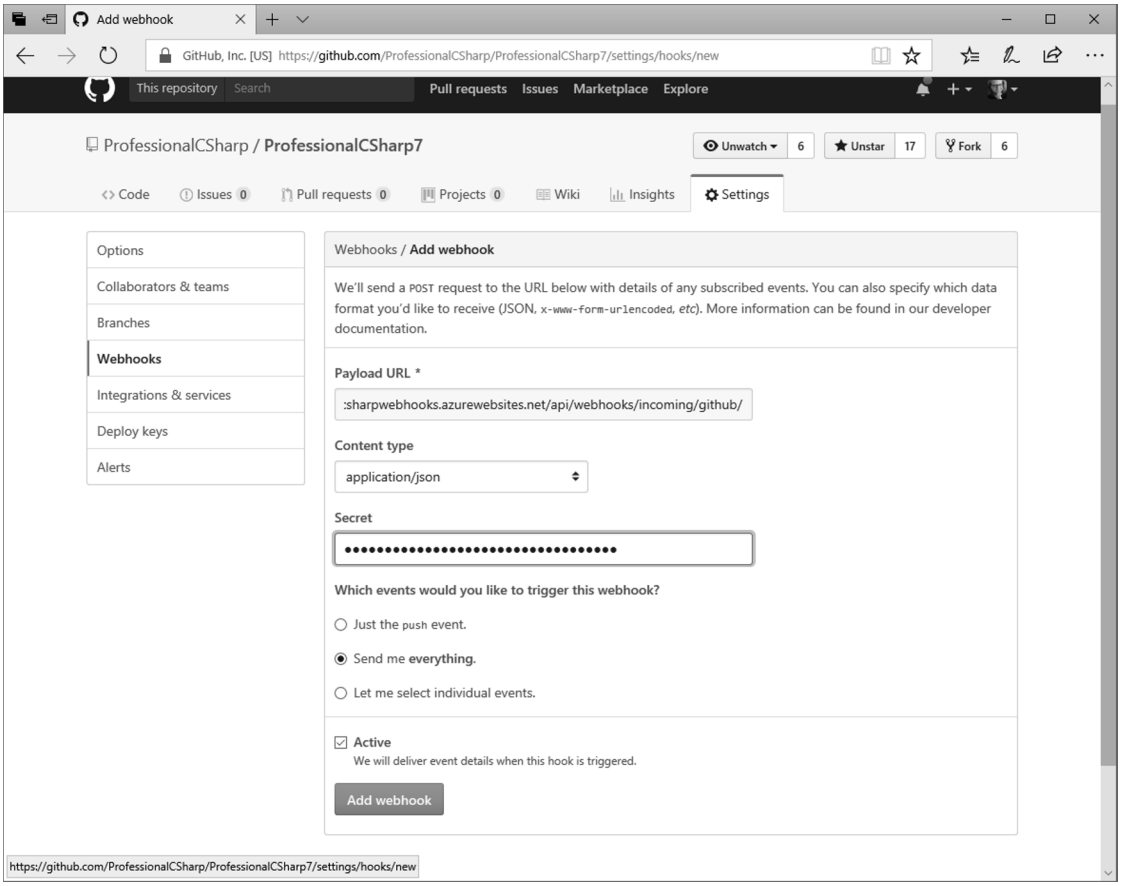

FIGURE BC3-8

Running the Application

With the configured public web application, as you make changes to your Dropbox folder or changes in your GitHub repository, you will find new messages arriving in the Microsoft Azure Storage queue. From within Visual Studio, you can directly access the queue using the *Cloud Explorer*. Selecting your storage account within the Storage Accounts tree entry, you can see the Queues entry that shows all the generated queues. When you open the queue, you can see messages like the one shown in Figure BC3-9.

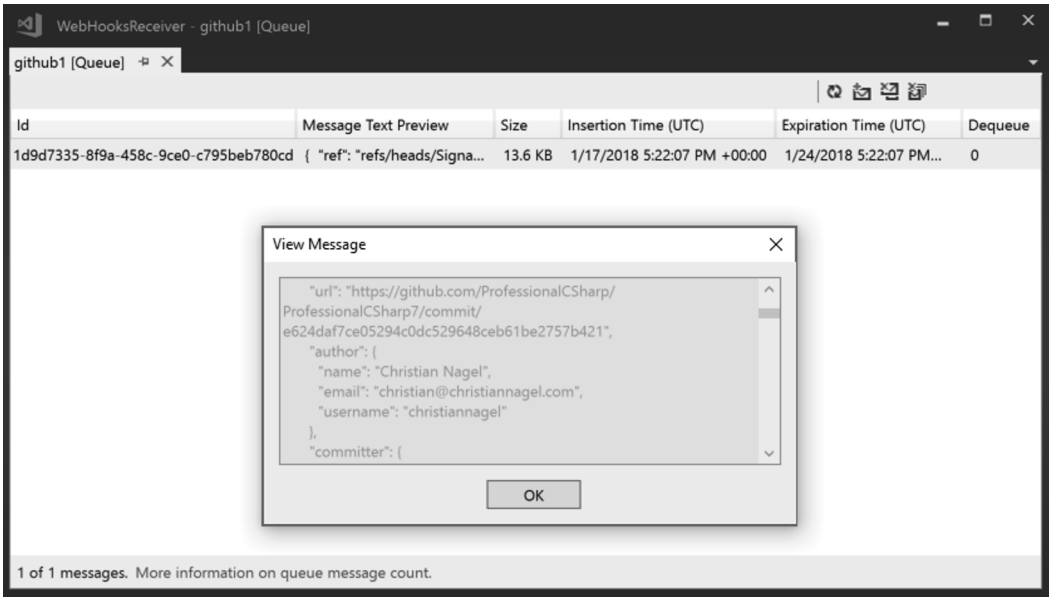

FIGURE BC3-9

SUMMARY

This chapter described publish/subscribe mechanisms with web applications. You've seen the foundation of WebSocket—creating a server and client app that stay connected.

With SignalR there's an easy way to make use of the WebSocket technology that keeps a network connection open to allow passing information from the server to the client, offer abstractions to WebSockets, and provide easy ways to return information to all clients or a specific group. You've seen how to create SignalR hubs and communicate both from a JavaScript as well as a .NET client.

With SignalR's support of groups, you've seen how the server can send information to a group of clients.

With the sample code you've seen how to chat between multiple clients using SignalR. Similarly, you can use SignalR with many other scenarios—for example, if you have some information from devices that call Web APIs with the server, you can inform connected clients with this information.

In the coverage of WebHooks you've seen another technology based on a publish/subscribe mechanism. WebHooks is unlike SignalR in that it can be used only with receivers available with public Internet addresses because the senders (typically SaaS services) publish information by calling web services. With the features of WebHooks, you've seen that many SaaS services provide WebHooks, and it's easy to create receivers that receive information from these services.

To get WebHooks forwarded to clients that are behind the firewall, you can combine WebHooks with SignalR. You just need to pass on WebHook information to connected SignalR clients.

The next bonus chapter introduces you to a new way of using application services: bots. You also learn about using Microsoft Azure Cognitive services, which can be used from bots as well as other apps and services.