**Основы SignalR**

**SignalR Core. Первое приложение**

### Что такое SignalR

SignalR Core представляет библиотеку от компании Microsoft, которая работает поверх ASP.NET и которая предназначена для создания приложений, работающих в режиме реального времени. В частности, ее можно использовать вместе с ASP.NET Core. SignalR использует двунаправленную связь для обмена сообщениями между клиентом и сервером, благодаря чему сервер может отправлять в режиме реального времени всем клиентам некоторые данные, а клиенты также в режиме реального времени могут передавать данные серверу и другим клиентам.

Где может использоваться SignalR? Прежде всего это приложения, которые получают данные в реальном режиме времени, например, чаты, социальные сети, игровые приложения, карты, приложения для аукционов, голосований и карт, панели управления, приложения для мониторинга данных и так далее.

Для обмена сообщениями между клиентом и сервером SignalR использует ряд механизмов:

* WebSockets
* Server-Side Events
* Long Polling

Исходя из возможностей клиента и сервера инфраструктура SignalR выбирает наилучший механизм для взаимодействия. В частности, наиболее оптимальным является WebSockets, соответственно если и клиент, и сервер позволяют использовать этот механизм, то взаимодействие идет через WebSockets. Однако если WebSockets не поддерживается, то применяется Server-Side Events. И если SSE не поддерживается, то применяется Long Polling.

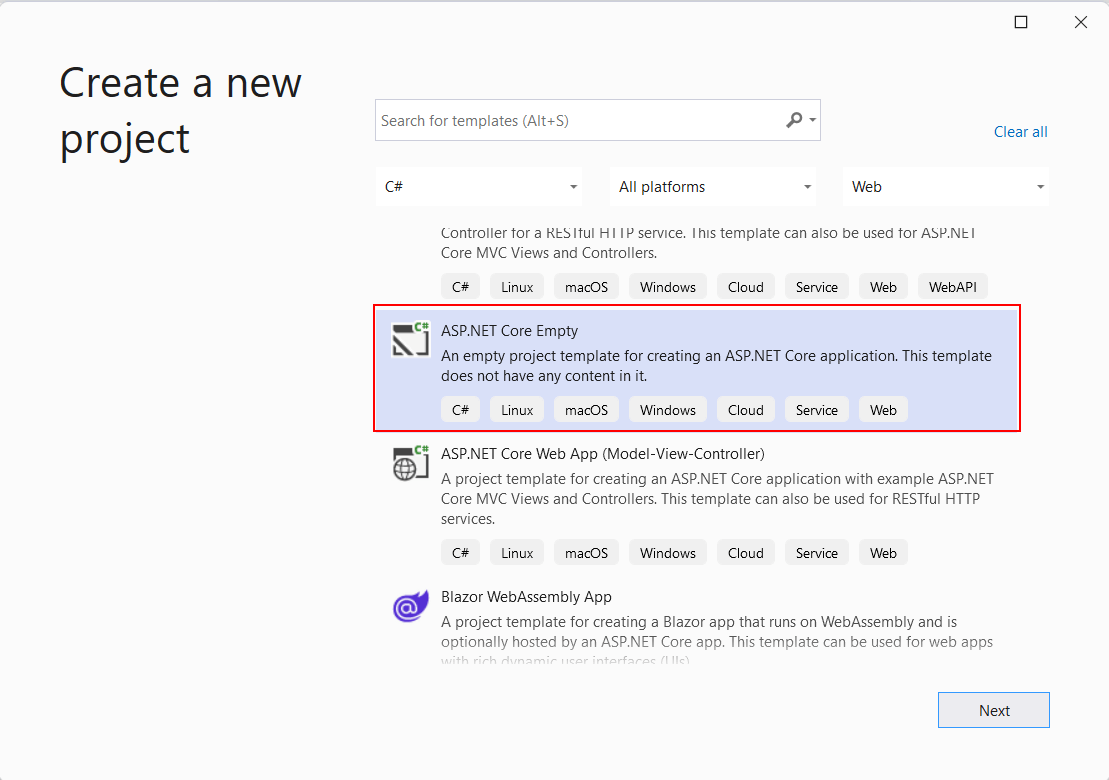
### Поддерживаемые клиенты

SignalR обеспечивает взаимодействие клиента с сервером. Если на стороне сервера ожидаемое это приложение ASP.NET Core, то на стороне клиента все намного интереснее. В частности, в качестве клиента в SignalR может выступать:

* Приложение на JavaScript, запущенное на Node.js (поддерживается версия Node.js 8 и выше)
* Приложение на JavaScript, которое работает в рамках браузеров Google Chrome (в том числе на Android), Microsoft Edge, Mozilla Firefox, Opera, Apple Safari (MacOS/iOS)
* Приложение на .NET. Это может быть десктопное приложение WPF, Windows Forms, мобильное приложение .NET MAUI.
* Приложение на языке Java
* Есть экспериментальная поддержка для приложений на языках C++ и Swift

### Первое приложение

Создадим новый простейший проект ASP.NET Core. Если мы работаем в Visual Studio, то надо создать проект по типу **ASP.NET Core Empty**:



Если мы работаем в текстовом редаторе и используем .NET CLI, то надо создать проект по шаблону web:

dotnet new web

### Определение серверной части

При работе с SignalR на стороне сервера необходимо создать специальную сущность - хаб (hub). По сути хаб представляет класс, наследующийся от класса Hub, который может обрабатывать запросы. Создадим новый хаб. Для этого добавим в проект следующий класс **ChatHub**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message)          {              await this.Clients.All.SendAsync("Receive", message);          }      }  } |

Класс хаба наследуется от класса **Hub**. И здесь определен один метод **Send()**, который получает некоторое отправленное сообщение в виде параметра message и затем с помощью вызова await Clients.All.SendAsync("Receive", message) ретранслирует это сообщение всем подключенным клиентам.

Первый параметр метода SendAsync() указывает на метод, который будет получать ответ от сервера, а второй параметр представляет набор значений, которые посылаются в ответе клиенту. То есть метод Receive на клиенте получит значение параметра message. То есть наш хаб будет просто получать сообщение и транслировать его всем подключенным клиентам.

### Конфигурация приложения

Но чтобы SignalR и хаб ChatHub заработали, необходимо подключить необходимые службы и настроить маршруты в приложении. Для этого откроем файл **Program.cs** и изменим его код следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();      // подключема сервисы SignalR    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");   // ChatHub будет обрабатывать запросы по пути /chat    app.Run(); |

Прежде всего, чтобы добавить в приложение сервисы SignalR, вызывается метод:

|  |  |
| --- | --- |
| 1 | services.AddSignalR(); |

Затем устанавливаем маршруты для хаба ChatHub:

|  |  |
| --- | --- |
| 1 | app.MapHub<ChatHub>("/chat"); |

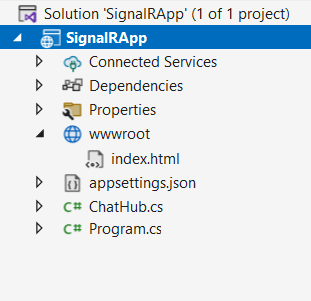
У объекта IEndpointRouteBuilder вызывается метод **MapHub**, который позволяет связать запросы и класс хаба. В данном случае он устанавливает класс ChatHub в качестве обработчика запросов по пути "/chat". То есть, чтобы обратиться к хабу, строка запроса должна иметь вид типа "https://localhost:5000/chat".

Стоит отметить, что если адрес сервера и адрес клиента не будут совпадать, то, возможно, потребуется настроить поддержку CORS. В данном же случае серверная и клиентская части будут работать в рамках одного приложения, поэтому настройки CORS не потребуется.

### Создание клиентской части

Для создания клиентской части можно использовать различные способы. Например, можно использовать javascript, либо же использовать typescript, определить приложение на .NET или Java. В данном случае мы будем использовать JavaScript, который будет выполняться на обычной странице html.

Для хранения статических файлов добавим в проект папку **wwwroot**. Затем добавим в папку **wwwroot** новый файл **index.html**. В итоге проект будет выглядеть следующим образом:



На странице **index.html** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div id="inputForm">          <input type="text" id="message" />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              let message = document.getElementById("message").value;              hubConnection.invoke("Send", message)                  .catch(function (err) {                      return console.error(err.toString());                  });          });            hubConnection.on("Receive", function(message) {                let messageElement = document.createElement("p");              messageElement.textContent = message;              document.getElementById("chatroom").appendChild(messageElement);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

На странице определено текстовое поле для ввода сообщение и кнопка для его отправки. Под ними расположен блок chatroom, в который будут добавляться полученные сообщения.

Внизу страницы подключается скрипт "signalr.min.js". Далее в коде javascript определена основная логика взаимодействия клиента с хабом.

Вначале определяется переменная, с помощью которой устанавливается подключение:

|  |  |
| --- | --- |
| 1  2  3 | const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build(); |

Для взаимодействия с хабом ChatHub с помощью метода build() объекта HubConnectionBuilder создается объект hubConnection - объект подключения. Метод **withUrl** устанавливает адрес, по которому приложение будет обращаться к хабу. Поскольку ChatHub на сервере сопоставляется с адресом "/chat", то именно этот адрес и передается в withUrl.

Далее устанавливается обработчик для кнопки, который вызывается при ее нажатии:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | document.getElementById("sendBtn").addEventListener("click", function () {      let message = document.getElementById("message").value;      hubConnection.invoke("Send", message)          .catch(function (err) {              return console.error(err.toString());          });  }); |

Для отправки данных хабу на сервер вызывается метод hubConnection.invoke("Send", message), первый параметр которого представляет метод хаба, обрабатывающий данный запрос, а второй параметр - данные, отправляемые на сервер.

В случае если при отправке возникнет ошибка, сработает функция, которая передается в метод catch()

Далее метод hubConnection.on устанавливает метод на стороне клиента, который будет получать данные от сервера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | hubConnection.on("Receive", function(message) {        let messageElement = document.createElement("p");      messageElement.textContent = message;      document.getElementById("chatroom").appendChild(messageElement);  }); |

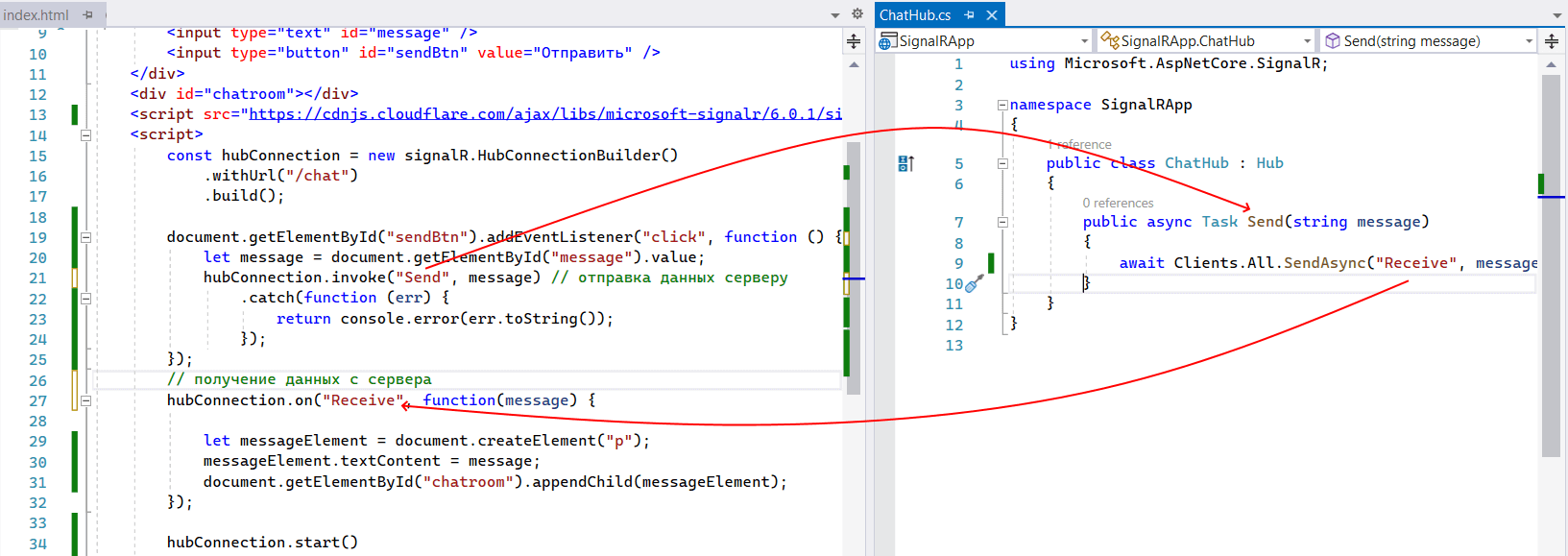
В данном случае метод называется Receive и фактически он представляют функцию, которая передается в качестве второго параметра. Эта функция принимает один параметр - message - те данные, которые в хабе отправляются клиенту. В самой функции с помощью стандартных функций javascript создается элемент. В этот элемент помещается присланное с сервера сообщение. Затем элемент добавляется в начало элемента chatroom.

И для начала соединения с сервером вызывается функция hubConnection.start().

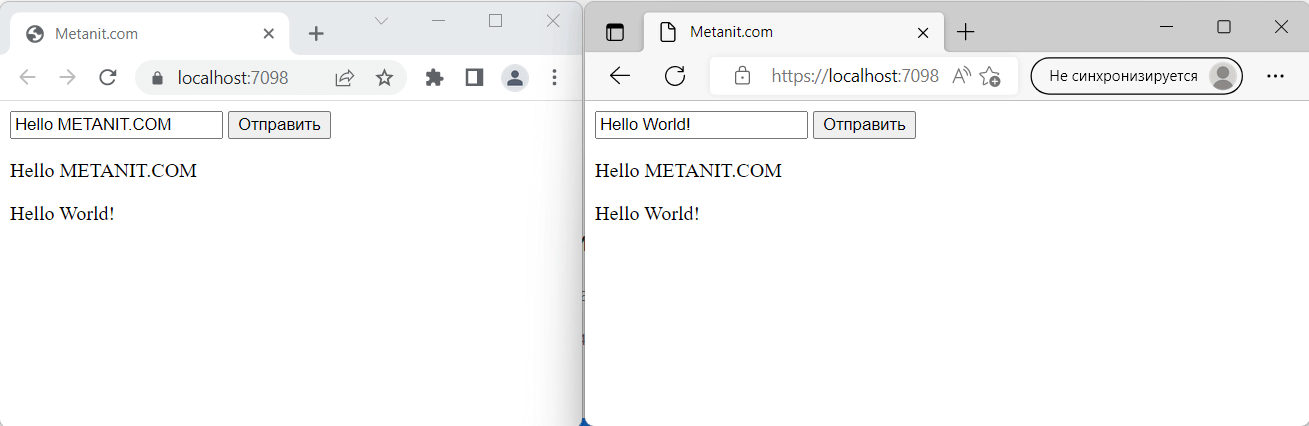
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | hubConnection.start()      .then(function () {          document.getElementById("sendBtn").disabled = false;      })      .catch(function (err) {          return console.error(err.toString());      }); |

Если подключение к хабу успешно установлено, то мы можем использовать метод then, чтобы выполнить некоторые действия. Так, в данном случае при успешном подключении делаем кнопку отправки доступной для нажатия.

Если же в процессе подключения к серверу возникнет ошибка, то сработает функция, которая передается в метод catch и которая получит данные об ошибке и выведет их на консоль браузера.



После запуска приложения в разных браузерах при отправке сообщения каждый браузер будет получать отправленное сообщение и выводить его на веб-страницу:



### Расширение приложения

Теперь модифицируем приложение, что кроме сообщения пользователя также передавалось и его имя. Вначале изменим код класса ChatHub:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message, string userName)          {              await Clients.All.SendAsync("Receive", message, userName);          }      }  } |

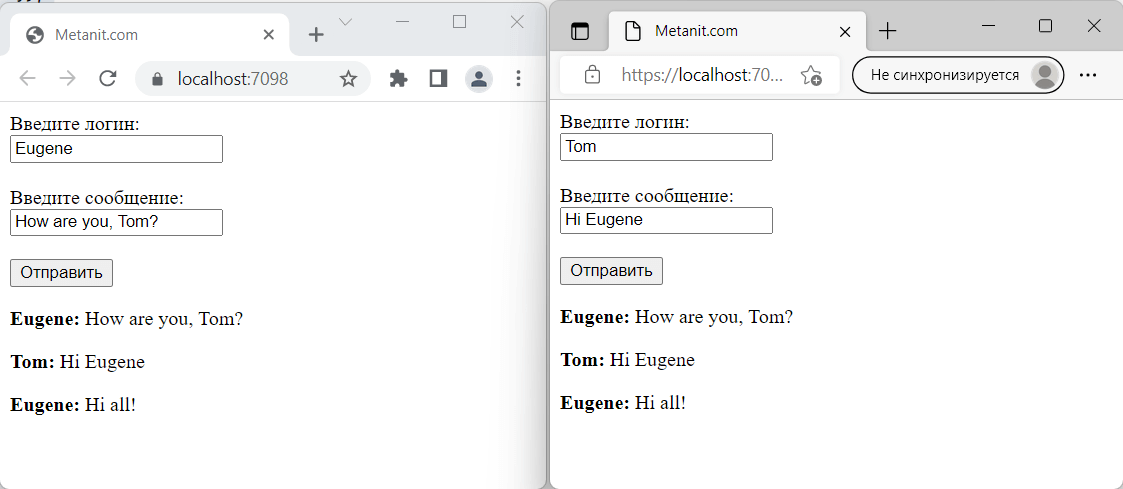
Теперь метод Send принимает два параметра и значения обоих параметров ретранслирует всем подключенным клиентам.

И изменим страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          Введите логин:<br />          <input id="userName" type="text" /><br/><br/>          Введите сообщение:<br />          <input type="text" id="message" /><br/><br/>          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const userName = document.getElementById("userName").value;   // получаем введенное имя              const message = document.getElementById("message").value;                hubConnection.invoke("Send", message, userName) // отправка данных серверу                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (message, userName) {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

Так как хаб на сервере отправляет клиентам два значения - имя пользователя и его сообщение, то соответственно на стороне клиента в функции, которая регистрируется в методе hubConnection.on мы можем получить оба этих значения.

И теперь мы условно можем войти под разными пользователями в различных браузерах и отправлять друг другу сообщения:



## Создание и конфигурация хабов

Ключевой сущностью в SignalR, через которую клиенты обмениваются сообщеними с сервером и между собой, является **хаб** (hub). Хаб представляет некоторый класс, который унаследован от абстрактного класса **Microsoft.AspNetCore.SignalR.Hub**.

Простейший хаб:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message)          {              await Clients.All.SendAsync("Receive", message);          }      }  } |

Класс хаба может определять различные методы с различными параметрами, однако клиент может обращаться только к публичным методам хаба. Например, в данном случае определен публичный метод Send, который принимает параметр типа string. А это значит, что клиент может вызвать на хабе этот метод и передать ему некоторую строку. Метод может содержать различную логику, но в данном случае метод просто транслирует полученную строку всех подключенным клиентам.

### **Конфигурация хаба**

Для использования функциональности библиотеки SignalR, в приложении надо зарегистрировать соответствующие сервисы с помощью метода **AddSignalR()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using SignalRApp; // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();      // подключаем сервисы SignalR    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");    app.Run(); |

Перегузка этого метода принимает в качестве параметра делегат Action, который имеет параметр типа **HubOptions**.

|  |  |
| --- | --- |
| 1  2 | ISignalRServerBuilder AddSignalR();  ISignalRServerBuilder AddSignalR(Action<HubOptions> configure); |

Параметр configure позволяет сконфигурировать подключение хаба. В частности, класс HubOptions определяет ряд свойств:

* **ClientTimeoutInterval**. Определяет время, в течение которого клиент должен отправить серверу сообщение. Если в течение данного времени никаких сообщенй от клиента на сервер не пришло, то сервер закрывает соединение.

По умолчанию равно 30 секунд.

* **HandshakeTimeout**. После подключения к серверу клиент должен отправить серверу в качестве самого первого сообщения специальное сообщение - HandshakeRequest. Это свойство устанавливает допустимое время таймаута, которое может пройти до получения от клиента первого сообщения об установки соединения. Если в течение этого периода клиент не отправит первое сообщение, то подключение закрывается.

По умолчанию равно 15 секунд.

* **KeepAliveInterval**: если в течение этого периода сервер не отправит никаких сообшений, то автоматически отправляется ping-сообщение для поддержания подключения открытым. При изменении этого свойства Microsoft рекомендует изменить на стророне клиента параметр serverTimeoutInMilliseconds (клиент javascript)/ ServerTimeout(клиент .NET), которое рекомендуется устанавливать в два раза больше, чем KeepAliveInterval.

По умолчанию равно 15 секунд.

* **SupportedProtocols** определяет поддерживаемые протоколы. По умолчанию поддерживаются все протоколы.
* **EnableDetailedErrors** при значении **true** возвращает клиенту детальное описание возникшей ошибки (при ее возникновении). Поскольку подобные сообщения могут содержать критически важную для безопасности информацию, то по умолчанию имеет значение **false**.
* **StreamBufferCapacity** определяет максимальный размер буфера для входящего потока клиента. По умолчанию равно 10.
* **MaximumReceiveMessageSize** определяет максимальный размер для входящего сообщения. По умолчанию - 32 кб
* **MaximumParallelInvocationsPerClient** определяет максимальное количество методов хаба, которые клиент может вызвать параллельно. По умолчанию равно 1.

Мы можем настроить параметры глобально для всех хабов в приложении, либо для хабов отдельного типа. Глобальная настройка хабов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR(hubOptions =>  {      hubOptions.EnableDetailedErrors = true;      hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);  }); |

Настройка только для хаба ChatHub:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var builder = WebApplication.CreateBuilder(args);  builder.Services.AddSignalR().AddHubOptions<ChatHub>(options =>  {      options.EnableDetailedErrors = true;      options.KeepAliveInterval = System.TimeSpan.FromMinutes(1);  }); |

Можно комбинировать оба вида настроек:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | builder.Services      .AddSignalR(hubOptions =>      {          hubOptions.EnableDetailedErrors = true;          hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);      })      .AddHubOptions<ChatHub>(options =>      {          options.EnableDetailedErrors = false;          options.KeepAliveInterval = TimeSpan.FromMinutes(5);      }); |

При этом настройки для отдельного хаба переопределяют глобальные настройки.

### **MapHub и маршрутизация**

Чтобы сервер мог сопоставить запросы с определенным хабом вызывается метод **MapHub()**:

|  |  |
| --- | --- |
| 1 | app.MapHub<ChatHub>("/chat"); |

С помощью метода **MapHub()** можно сопоставить URL с хабом. В данном случае все запросы по адресу "/chat" будут обрабатываться хабом ChatHub.

Перегрузка метода MapHub дополнительно принимает делегат с параметром типа **HttpConnectionDispatcherOptions**, благодаря чему мы можем настроить различные настройки подключения

|  |  |
| --- | --- |
| 1 | HubEndpointConventionBuilder MapHub<THub>(string pattern, Action<HttpConnectionDispatcherOptions>? configureOptions) where THub : Hub; |

В частности, в HttpConnectionDispatcherOptions есть следующие свойства:

* **ApplicationMaxBufferSize**: максимальный размер буфера в байтах, в который сервер помещает получаемые от клиента данные.

По умолчанию равно 64 килобайта.

* **AuthorizationData**: представляет список (объект IList) объектов IAuthorizeData, которые определяют, авторизован ли клиент для подключения к хабу.
* **TransportMaxBufferSize** максимальный размер буфера в байтах, в который сервер помещает данные для отправки клиенту.

По умолчанию равно 64 килобайта.

* **MinimumProtocolVersion**: минимальная версия протокола. Применяется, чтобы отсеить клиентов определенных версий
* **Transports** представляет битовую маску из значений перечисления **HttpTransportType**, которая устанавливает допустимые типы транспорта. По умолчанию применяются все типы транспорта
* **LongPolling** представляет объект **LongPollingOptions**, который настраивает транспорт LongPolling. Этот класс имеет только одно свойство **PollTimeout**, которое устанавливает периодичность опроса. По умолчанию равно 90 секунд.
* **WebSockets** представляет объект **Microsoft.AspNetCore.Http.Connections.WebSocketOptions**, который настраивает транспорт WebSocket. У данного объекта можно установить два свойства:
  + CloseTimeout: временный интервал после закрытия сервера, в течение которого клиент должен закрыть подключение. Если клиенту не удастся закрыть подключение, то соединение с сервером автоматически завершается.
  + SubProtocolSelector: делегат, который устанавливает заголовок Sec-WebSocket-Protocol.

Применим некоторые свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using Microsoft.AspNetCore.Http.Connections; // пространство имен перечисления HttpTransportType  using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat",      options => {          options.ApplicationMaxBufferSize = 128;          options.TransportMaxBufferSize = 128;          options.LongPolling.PollTimeout = TimeSpan.FromMinutes(1);          options.Transports = HttpTransportType.LongPolling | HttpTransportType.WebSockets;  });  app.Run(); |

## Клиент javascript

### Установка

Клиент javascript является одним из распространенных типов клиентов в SignalR, который может взаимодействовать с хабом на сервере. Для создания клиента js на веб-страницу необходимо подключить специальную библиотеку.

#### Получение из CDN

Для подключения SignalR можно использовать сети CDN, которые содержат данную библиотеку. В частности, подключить SignalR на страницу можно по одному из следующих адресов:

|  |  |
| --- | --- |
| 1  2  3 | <https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>  <https://unpkg.com/@microsoft/signalr@6.0.1/dist/browser/signalr.min.js>  <https://cdn.jsdelivr.net/npm/@microsoft/signalr/dist/browser/signalr.min.js> |

### Варианты добавления библиотеки в проект

Выше рассматривалась работа с библиотекой SignalR, когда она подключается через CDN:

|  |  |
| --- | --- |
| 1 | <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script> |

Либо при необходимости мы можем сохранить данный файл локально в проект и подключать уже из проекта. Это самый простой вариант, однако есть также и другие способы добавления SignalR в проект.

#### Добавление через NPM

В качестве альтернативы, особенно, если мы работаем не в Visual Studio, можно было бы загрузить пакет "@microsoft/signalr" через пакетный менеджер NPM с помощью команды:

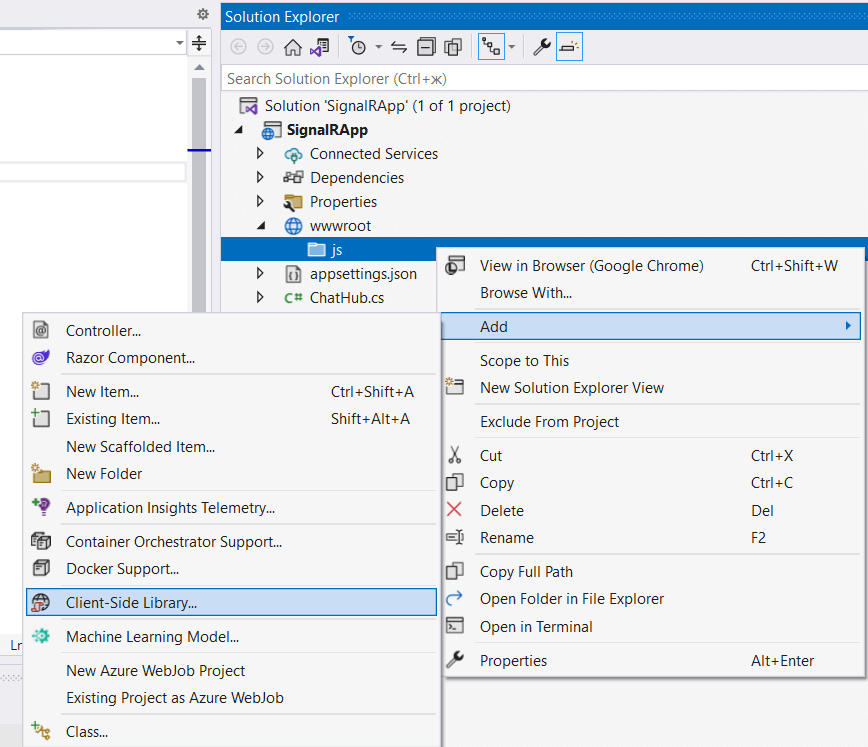
|  |  |
| --- | --- |
| 1 | npm install @microsoft/signalr |

либо добавить библиотеку через файл **package.json**:

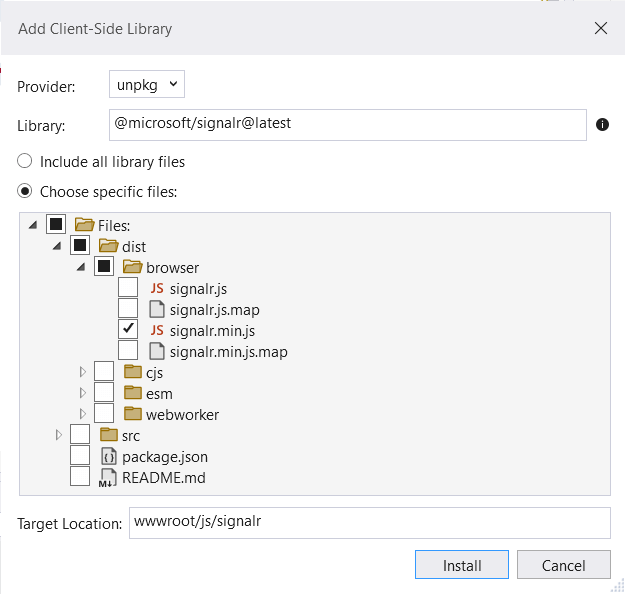
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {    "version": "1.0.0",    "name": "asp.net",    "private": true,    "devDependencies": {      "@microsoft/signalr": "6.0.0"    }  } |

#### Добавление через менеджер libman

Другая альтернатива состоит в добавлении библиотеки SignalR в проект через пакетный менеджер libman. Для хранения всех необходимых javascript вначале создадим в папке **wwwroot** новый подкаталог **js**. Далее нажмем на эту папку правой кнопкой мыши и в контекстном меню выберем **Add -> Client Side Library**



Далее нам откроется окно добавления клиентских библиотек. Укажем в нем следующие опции:



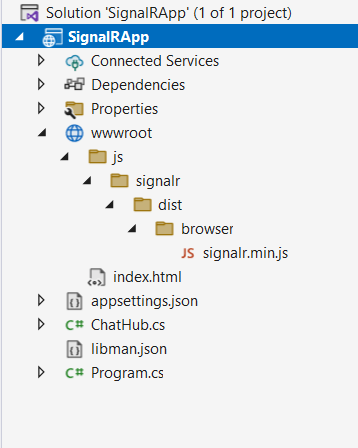
В поле **Provider** укажем значение **unpkg**.

В поле **Library** в качестве названия пакета введем **@microsoft/signalr@latest**.

Поскольку пакет содержит много файлов, которые нам могут не понадобиться, то отметим пункт **Choose specific files:** и затем из списка файлов отметим только **signalr.min.js**, то есть минимизированную версию библиотеки. Но при желании и необходимости можно выбрать и другие файлы.

И в конце в поле **Target location** укажем расположение, по которому будет сохранена библиотека, то есть путь **wwwroot/js/signalr**.

В итоге в проекте по пути *wwwroot/js/signalr/dist/browser/* мы сможем найти файл *signalr.min.js*



И затем мы сможем подключить библиотеку на веб-страницу:

|  |  |
| --- | --- |
| 1 | <script src="js/signalr/dist/browser/signalr.min.js"></script> |

### Определение клиента на javascript

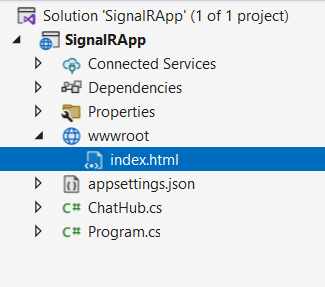
Допустим, на стороне сервера определен следующий хаб **ChatHub**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message, string userName)          {              await Clients.All.SendAsync("Receive", message, userName);          }      }  } |

В файле Program.cs для хаба определена следующая конфигурация:

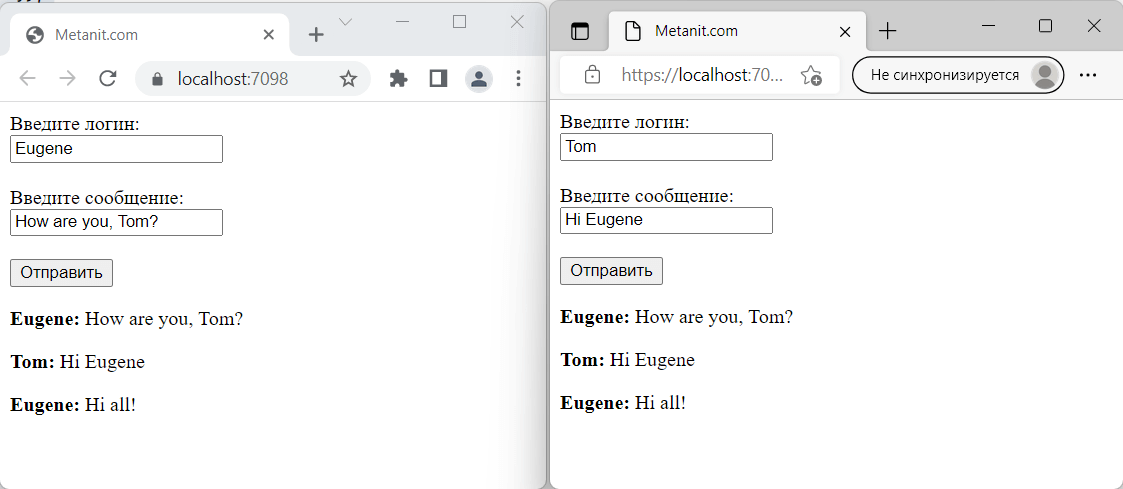
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();      // подключаем сервисы SignalR    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");   // ChatHub будет обрабатывать запросы по пути /chat    app.Run(); |

И определим в проекте в папке **wwwroot** страницу **index.html**:



И определим в ней следующий код

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          Введите логин:<br />          <input id="userName" type="text" /><br/><br/>          Введите сообщение:<br />          <input type="text" id="message" /><br/><br/>          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const userName = document.getElementById("userName").value;   // получаем введенное имя              const message = document.getElementById("message").value;                hubConnection.invoke("Send", message, userName) // отправка данных серверу                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (message, userName) {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |



Теперь рассмотрим подробнее основные моменты работы клиента javascript.

### Подключение к хабу

Чтобы подключиться к хабу, применяется объект **signalR.HubConnectionBuilder**:

|  |  |
| --- | --- |
| 1  2  3 | const hubConnection = new signalR.HubConnectionBuilder()      .withUrl("/chat")      .build(); |

Через его метод **withUrl** передается адрес, по которому доступен хаб. А метод **build()** собственно создает объек подключения - объект **HubConnection**, через который мы можем взаимодействовать с хабом.

После получения объекта HubConnection для подключения к хабу надо вызвать метод **start()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | hubConnection.start()             .then(function () {                 document.getElementById("sendBtn").disabled = false;             })             .catch(function (err) {                 return console.error(err.toString());             }); |

Для обработки подключения или ошибки мы можем в методу start добавить вызовы методов then и catch. В метод then передается функция, которая вызывается при успешном подключении. А в метод catch передается функция, вызываемая при возникновении ошибки. Параметр этой функции как раз содержит информацию об оишбке.

### Отправка данных хабу

Для отправки данных хабу у объекта HubConnection применяется метод **invoke()**. Например, выше в классе хаба ChatHub был определен метод Send, который принимает два параметра. Обратимся к этому методу из кода javascript:

|  |  |
| --- | --- |
| 1  2  3  4 | hubConnection.invoke("Send", "Hello SignalR", "Tom")              .catch(function (err) {                  return console.error(err.toString());              }); |

Первый параметр метода представляет имя метода хаба, к которому идет обращения. Например, в данном случае это метод Send. Все последующие параметры передают данные для параметров метода хаба. Так, метод Send в хабе ChatHub принимает два строковых параметра. Соответственно в методе invoke мы можем передать для них данные.

Параметры передаются по позиции: второй аргумент метода invoke передает значение для первого параметра метода Send, третий аргумент в invoke - для второго параметра в Send и так далее.

Сам метод invoke представляет объект Promise языка javascript. Соответственно для обработки ошибок, которые могут произойти в процессе отправки, к методу invoke мы можем добавить вызов метода catch. В метод catch в нее передается функция, которая вызывается при возникновении ошибки. Параметр этой функции (в примере выше параметр err) содержит информацию об ошибке.

### Вызов метода клиента из хаба

Клиент может обращаться к методу хаба, но и хаб может вызывать метод на клиенте. Для этого объект HubConnection с помощью метода **on** должен зарегистрировать функцию, которую сможет вызывать хаб. Например, выше в методе Send в хабе полученные данные транслировались на функцию Receive всем подключенным клиентам:

|  |  |
| --- | --- |
| 1 | await Clients.All.SendAsync("Receive", message, userName); |

Первый аргумент функции SendAsync как раз представляет название функции в коде клиента, которая будет вызываться. Последующие аргументы передают данные, которые получит функция на клиенте.

В этом случае на клиенте мы могли бы определить функцию Receive и получить данные от хаба:

|  |  |
| --- | --- |
| 1  2  3  4 | hubConnection.on("Receive", function (message, userName) {        // обработка данных  }); |

Первый параметр функции on - это название вызываемой хабом функции - в данном случае Receive. Второй параметр - функция, которая получает переданные хабом данные.

### Логгирование

У объекта signalR.HubConnectionBuilder есть метод **configureLogging()**, в который передается уровень логгирования:

|  |  |
| --- | --- |
| 1  2  3  4 | const connection = new signalR.HubConnectionBuilder()      .withUrl("/chat")      .configureLogging(signalR.LogLevel.Information)      .build(); |

Все возможные уровни логгирования:

* signalR.LogLevel.None: логгирование отключено
* signalR.LogLevel.Critical: логгирование сообщений об ошибках, которые относятся ко всему приложению в целом
* signalR.LogLevel.Error: логгирование сообшений об ошибках, которые относятся к текущей операции
* signalR.LogLevel.Warning: логгирование сообщений, которые не представляют ошибки
* signalR.LogLevel.Information: логгирование информационных сообщений
* signalR.LogLevel.Debug: логгирование диагностических сообщений, используемых при отладке
* signalR.LogLevel.Trace: логгирование диагностических сообщений с детальной информацией

### Конфигурация клиента

#### Настройка типа транспорта

Для настройки транспорта в методе **withUrl()** у дополнительного параметра устанавливается свойство transport, которому передаются значения объекта **HttpTransportType**: WebSockets, LongPolling или ServerSentEvents. По умолчанию используются все эти виды транспорта, но можно указать только один из них:

|  |  |
| --- | --- |
| 1  2  3  4 | const hubConnection = new signalR.HubConnectionBuilder()      .withUrl("/chat", { transport: signalR.HttpTransportType.WebSockets })      .configureLogging(signalR.LogLevel.Information)      .build(); |

Установка нескольких типов транспорта:

|  |  |
| --- | --- |
| 1  2  3  4 | const hubConnection = new signalR.HubConnectionBuilder()      .withUrl("/chat", { transport: signalR.HttpTransportType.WebSockets | signalR.HttpTransportType.LongPolling })      .configureLogging(signalR.LogLevel.Information)      .build(); |

Единственное, что надо учитывать, что сервер тоже должен поддерживать устанавливаемые виды транспорта.

#### Установка времени таймаута

Для установки таймаута для объекта **HubConnection** определено два свойства:

* **serverTimeoutInMilliseconds** представляет таймаут, в течение которого подключение считается активным. Если в течение этого периода сервер не присылает никакого сообщения, то клиент считает, что подключение к серверу разорвано. И в этом случае вызывается метод **onclose()**. Рекомендуется устанавливать это значение в два раза большим, чем параметр KeepAliveInterval на стороне сервера. По умолчанию равно 30 секунд (30000 миллисекунд).
* **keepAliveIntervalInMilliseconds** определяет интервал, в течение которого клиент посылает пинг-сообщения серверу. Отправка любого сообщения от клиента сбрасывает таймер для отслеживания этого интервала до нуля. Если клиент не отправит никакого сообщения в течении интервала, который устанавливается свойством ClientTimeoutInterval класса хаба на сервере, то сервер считает, что клиент отключился. Значение по умолчанию - 15 секунд

Например, установим таймаут в виде 10 минут:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var connection = new signalR.HubConnectionBuilder()      .withUrl("/chat")      .configureLogging(signalR.LogLevel.Trace)      .build();    connection.serverTimeoutInMilliseconds = 1000 \* 60 \* 10; |

## Клиент на .NET

наряду с клиентом для javascript библиотека SignalR также поддерживает создание клиентских приложений для .NET. Рассмотрим основные моменты клиента для .NET на примере простейшего приложения на WPF.

### Создание сервера

Сначала определим код сервера, с которым будет взаимодействовать клиент на .NET MAUI. Для этого создадим проект ASP.NET Core по типу Empty.

Определим в проекте следующий простейший класс хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string username, string message)          {              await this.Clients.All.SendAsync("Receive", username, message);          }      }  } |

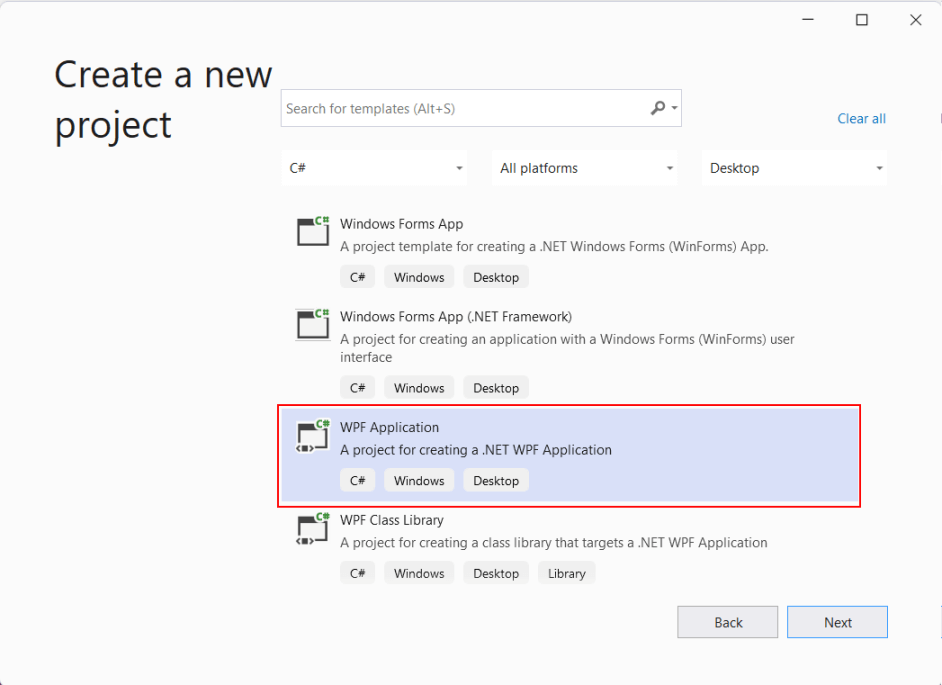
В методе Send хаб будет принимать имя пользователя и его сообщение и транслировать его на функцию Receive всех подключенных клиентов.

В файле **Program.cs** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();    var app = builder.Build();      app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");  app.Run(); |

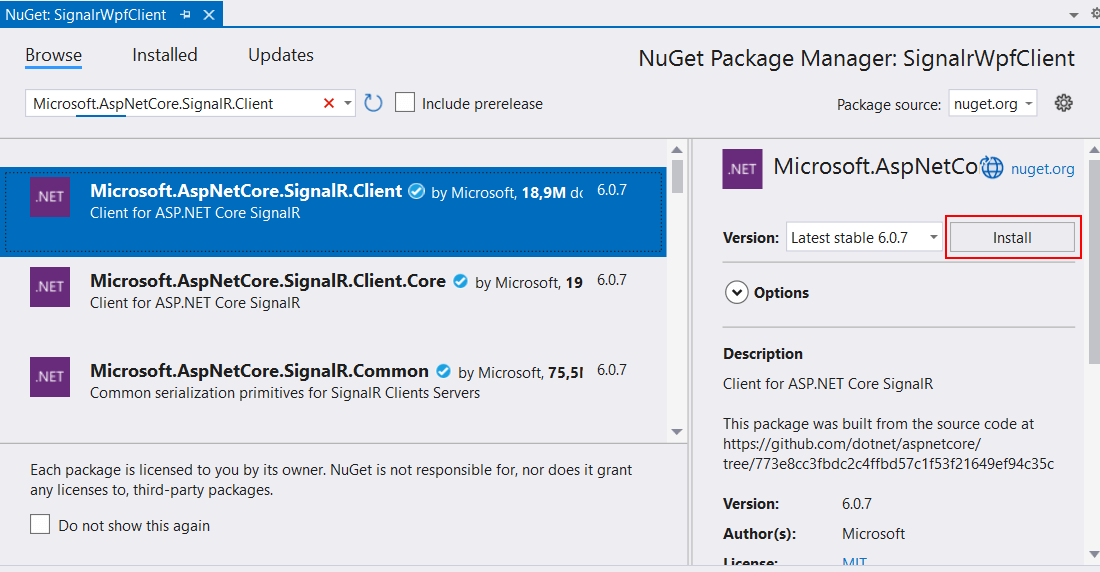
### Создание клиента .NET MAUI

Прежде всего, создадим новый проект по типу **WPF Application**:



Допустим, проект будет называться **SignalrWpfClient**.

После создания проекта в первую очередь добавим в него Nuget-пакет **Microsoft.AspNetCore.SignalR.Client**.



По умолчанию проект WPF содержит определение главного окна в файлах **MainWindow.xaml** и **MainWindow.xaml.cs**. В файле **MainWindow.xaml** определим следующий интерфейс окна:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <Window x:Class="SignalrWpfClient.MainWindow"          xmlns="<http://schemas.microsoft.com/winfx/2006/xaml/presentation>"          xmlns:x="<http://schemas.microsoft.com/winfx/2006/xaml>"          xmlns:d="<http://schemas.microsoft.com/expression/blend/2008>"          xmlns:mc="<http://schemas.openxmlformats.org/markup-compatibility/2006>"          mc:Ignorable="d"          Title="METANIT.COM" Height="450" Width="300" Loaded="Window\_Loaded">      <StackPanel Margin="4">          <Label Content="Введите логин"/>          <TextBox x:Name="userTextBox" />          <Label Content="Введите сообщение"/>          <TextBox x:Name="messageTextBox" />          <Button x:Name="sendBtn" Content="Отправить" Click="Button\_Click" IsEnabled="False" />          <ListBox x:Name="chatbox" />      </StackPanel>  </Window> |

Здесь определены два текстовых поля для ввода имени пользователя и сообщения. Также здесь определена кнопка, по нажатию на которую введенные в текстовые поля данные будут отправляться хабу.

Кроме того, здесь определен элемент ListBox, который будет отображать полученные сообщения.

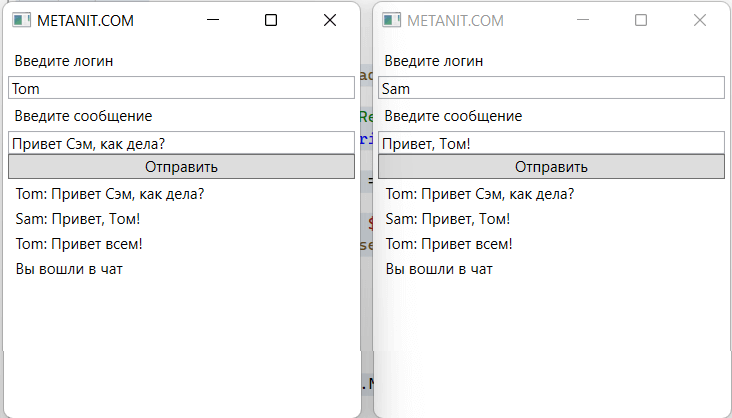
Теперь изменим код в файле **MainWindow.xaml.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | using Microsoft.AspNetCore.SignalR.Client;  using System;  using System.Windows;    namespace SignalrWpfClient  {      public partial class MainWindow : Window      {          HubConnection connection; // подключение для взаимодействия с хабом          public MainWindow()          {              InitializeComponent();                // создаем подключение к хабу              connection = new HubConnectionBuilder()                  .WithUrl("<https://localhost:7098/chat>")                  .Build();                  // регистрируем функцию Receive для получения данных              connection.On<string, string>("Receive", (user, message) =>              {                  Dispatcher.Invoke(() =>                  {                      var newMessage = $"{user}: {message}";                      chatbox.Items.Insert(0, newMessage);                  });              });          }            // обработчик загрузки окна          private async void Window\_Loaded(object sender, RoutedEventArgs e)          {              try              {                  // подключемся к хабу                  await connection.StartAsync();                  chatbox.Items.Add("Вы вошли в чат");                  sendBtn.IsEnabled = true;              }              catch (Exception ex)              {                  chatbox.Items.Add(ex.Message);              }          }          // обработчик нажатия на кнопку          private async void Button\_Click(object sender, RoutedEventArgs e)          {              try              {                  // отправка сообщения                  await connection.InvokeAsync("Send", userTextBox.Text, messageTextBox.Text);              }              catch (Exception ex)              {                  chatbox.Items.Add(ex.Message);              }          }      }  } |

В данном случае у меня сервер запущен по адресу "https://localhost:7098/", поэтому я указываю этот адрес для подключения к хабу:

|  |  |
| --- | --- |
| 1  2 | connection = new HubConnectionBuilder()                  .WithUrl("<https://localhost:7098/chat>") |

Сначала запустим сервер, а затем скомпилируем приложение WPF и запустим несколько его копий. И мы сможем в разных приложениях, подключенных к хабу, посылать в чат сообщения:



Теперь рассмотрим подробнее основные моменты работы клиента Signalr для .NET.

### **Создание подключения**

Чтобы подключиться к хабу, применяется объект **HubConnection**:

|  |  |
| --- | --- |
| 1 | HubConnection connection; |

Для создания данного объекта применяется специальный класс-строитель **HubConnectionBuilder**:

|  |  |
| --- | --- |
| 1  2  3 | connection = new HubConnectionBuilder()          .WithUrl("<https://localhost:7098/chat>")          .Build(); |

Через его метод **WithUrl** передается адрес, по которому доступен хаб. А метод **Build()** собственно создает объект подключения - объект **HubConnection**, через который мы можем взаимодействовать с хабом.

### Получение данных от хаба

Далее регистрируем метод, которая будет получать данные от хаба с помощью метода connection.On()

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | connection.On<string, string>("Receive", (user, message) =>  {      Dispatcher.Invoke(() =>      {          var newMessage = $"{user}: {message}";          chatbox.Items.Insert(0, newMessage);      });  }); |

Первый параметр метода **connection.On()** представляет имя метода, который будет получать данные от хаба, а второй параметр - собственно определение этого метода.

Например, выше в методе Send в хабе полученные данные транслировались на метод Receive всем подключенным клиентам:

|  |  |
| --- | --- |
| 1 | await Clients.All.SendAsync("Receive", username, message); |

Первый аргумент вызова SendAsync как раз представляет название метод в коде клиента, которая будет вызываться. Последующие аргументы передают данные, которые получит метод на клиенте.

Поэтому первый параметр метода connection.On представляет строка "Receive", а второй параметр - метод или точнее лямбда-выражение получает отправленные хабом данные в виде параметров. А поскольку хаб посылает две строки, то метод connection.On() типизируется двумя типами string.

В данном случае при получении данных от хаба с помощью метода Dispatcher.Invoke мы можем обратиться к пользовательскому интерфейсу и добавить полученные данные в список chatbox, который представляет элемент ListBox.

### Подключение к хабу

После получения объекта HubConnection для подключения к хабу надо вызвать метод **StartAsync()**. В пример выше это делается в обработчике события загрузки окна:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private async void Window\_Loaded(object sender, RoutedEventArgs e)  {      try      {          // подключемся к хабу          await connection.StartAsync();          chatbox.Items.Add("Вы вошли в чат");          sendBtn.IsEnabled = true;      }      catch (Exception ex)      {          chatbox.Items.Add(ex.Message);      }  } |

В случае успешного выполнения метода StartAsync пользователь может начать взаимодействовать с хабом.

### **Отправка данных хабу**

Для отправки данных хабу у объекта HubConnection применяется метод **InvokeAsync()**. Например, выше в классе хаба ChatHub был определен метод Send, который принимает два параметра. В клиенте на WPF отправка этому методу происходит в обработчике нажатия кнопки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | private async void Button\_Click(object sender, RoutedEventArgs e)  {      try      {          await connection.InvokeAsync("Send", userTextBox.Text, messageTextBox.Text);      }      catch (Exception ex)      {          chatbox.Items.Add(ex.Message);      }  } |

Первый параметр метода представляет имя метода хаба, к которому идет обращения. Например, в данном случае это метод Send. Все последующие параметры передают данные для параметров метода хаба. Так, метод Send в хабе ChatHub принимает два строковых параметра. Соответственно в методе InvokeAsync мы можем передать для них данные.

Параметры передаются по позиции: второй аргумент метода InvokeAsync передает значение для первого параметра метода Send, третий аргумент в InvokeAsync - для второго параметра в Send и так далее.

### Отключение от хаба

При необходимости после подключения к хабу мы можем отключиться от него с помощью метода **StopAsync()** объекта HubConnection. Например, можно было бы определить для события Closing - события, которое возникает перед закрытием окна, следующий обработчик:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private async void Window\_Closing(object sender, System.ComponentModel.CancelEventArgs e)  {      await connection.InvokeAsync("Send", "",$"Пользователь {userTextBox.Text} выходит из чата");      await connection.StopAsync();   // отключение от хаба  } |

### 

### **Свойства и события HubConnection**

HubConnection имеет ряд свойств, которые позволяют получить информацию о подключении или сконфигурировать клиент:

* **ConnectionId** представляет индификатор текущего подключения
* **ServerTimeout** представляет таймаут, в течение которого подключение считается активным. Если в течение этого периода сервер не присылает никакого сообщения, то клиент считает, что подключение к серверу разорвано. И в этом случае вызывается событие **Closed()**.
* **KeepAliveInterval** определяет интервал, в течение которого клиент посылает пинг-сообщения серверу. Отправка любого сообщения от клиента сбрасывает таймер для отслеживания этого интервала до нуля. Если клиент не отправит никакого сообщения в течении интервала, который устанавливается свойством ClientTimeoutInterval класса хаба на сервере, то сервер считает, что клиент отключился

События HubConnection:

* **Closed** возникает после закрытия подключения
* **Reconnected** возникает после переподключения к хабу.
* **Reconnecting** возникает перед переподключением к хабу

### Переподключение

Возможна ситуация, что соединение с хабом будет потеряно. Если мы хотим, чтобы подключение было восстановлено, то мы можем сделать это автоматически, используя у HubConnectionBuilder метод **WithAutomaticReconnect()**:

|  |  |
| --- | --- |
| 1  2  3  4 | connection = new HubConnectionBuilder()                  .WithUrl("<https://localhost:7098/chat>")                  .WithAutomaticReconnect()   // автопереподключение                  .Build(); |

При необходимости можно указать временные интервалы для переподключения с помощью массива TimeSpan:

|  |  |
| --- | --- |
| 1  2  3  4 | connection = new HubConnectionBuilder()                  .WithUrl("<https://localhost:7098/chat>")                  .WithAutomaticReconnect(new[] { TimeSpan.FromSeconds(10),  TimeSpan.FromSeconds(20) })                  .Build(); |

В данном случае клиент пытается переподключить сначала через 10, а потом через 20 секунд. Можно задать произвольное количество подобных интервалов.

Перед переподключением возникает событие **Reconnecting**, а после переподключения - событие **Reconnected**. Соответственно, если небходимо отследить переподключение, то можно обработать данные события:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | connection.Reconnecting += error =>  {      // обработка события        return Task.CompletedTask;  };  connection.Reconnected += connectionId =>  {      // обработка события        return Task.CompletedTask;  }; |

В обработчик события Reconnecting передается информация об ошибке в виде объекта Exception, а в обработчик события Reconnected передается новый идентификатор подключения.

Также можно вручную переподключиться с помощью обработки события **Closed**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | connection.Closed += async (error) =>  {      await Task.Delay(1000); // черех секунду переподключаемся      await connection.StartAsync();  }; |

### Логгирование

У объекта HubConnectionBuilder есть метод **ConfigureLogging()**, в который передается делегат Action<Microsoft.Extensions.Logging.ILoggingBuilder>. Объект ILoggingBuilder позволяет настроить ряд параметров логгирования, в частности, с помощью метода SetMinimumLevel() устанавливается уровень логгирования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | connection = new HubConnectionBuilder()                  .WithUrl("<https://localhost:7098/chat>")                  .ConfigureLogging(logging =>                  {                      logging.SetMinimumLevel(LogLevel.Information);                  })                  .Build(); |

Все возможные уровни логгирования:

* LogLevel.None: логгирование отключено
* LogLevel.Critical: логгирование сообщений об ошибках, которые относятся ко всему приложению в целом
* LogLevel.Error: логгирование сообшений об ошибках, которые относятся к текущей операции
* LogLevel.Warning: логгирование сообщений, которые не представляют ошибки
* LogLevel.Information: логгирование информационных сообщений
* LogLevel.Debug: логгирование диагностических сообщений, используемых при отладке
* LogLevel.Trace: логгирование диагностических сообщений с детальной информацией

## Контекст хаба, подключение и отключение клиентов

В классе Hub определено свойство **Context**, которое представляет тип **HubCallerContext** и который позволяет нам получить некоторую информацию о подключении к хабу. Данная информация хранится в следующих его свойствах:

* **ConnectionId**: возвращает уникальный идентификатор подключения в виде строки.
* **ConnectionAborted**: возвращает объект CancellationToken, который извещает о закрытии подключения
* **User**: возвращает объект **ClaimsPrincipal**, ассоциированный с текущим пользователем. По сути это аналог свойства HttpContext.User, которое доступно в констроллерах
* **UserIdentifier**: возвращает идентификатор пользователя. По умолчанию индентификатор пользователя представляет клейм **ClaimTypes.NameIdentifier** объекта **ClaimsPrincipal**, который ассоциирован с данным подключением.
* **Items**: возвращает словарь значений, ассоциированных с текущим подключением. Данный словарь позволяет хранить данные для определенного клиента между его запросами
* **Features**: возвращает коллекцию возможностей HTTP, ассоциированных с текущим подключением

Также класс HubCallerContext определяет пару методов:

* **Abort()**: принудительно завершает текущее подключение
* **GetHttpContext()**: возвращает объект **HttpContext** для текущего подключения или **null**, если подключение не ассоциировано с запросом HTTP.

### **Подключение и отключение клиентов**

Класс Hub определяет два метода, которые мы можем переопределить в классах-наследниках:

* **OnConnectedAsync()**: срабатывает при подключении нового клиента
* **OnDisconnectedAsync(Exception exception)**: срабатывает при отключении клиента, в качестве параметра передается сообщение об ошибке, которая описывает, почему произошло отключение.

Реализуем эти методы и определим следующий класс хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message)          {              await Clients.All.SendAsync("Receive", message, Context.ConnectionId);          }            public override async Task OnConnectedAsync()          {              await Clients.All.SendAsync("Notify", $"{Context.ConnectionId} вошел в чат");              await base.OnConnectedAsync();          }          public override async Task OnDisconnectedAsync(Exception? exception)          {              await Clients.All.SendAsync("Notify", $"{Context.ConnectionId} покинул в чат");              await base.OnDisconnectedAsync(exception);          }      }  } |

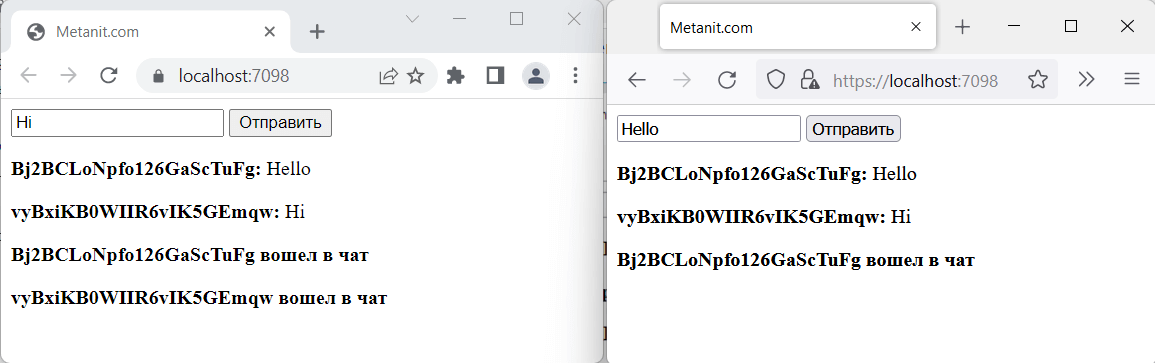
В данном случае вместо условного ника пользователя будет использоваться идентификатор подключения. В методах OnConnectedAsync и OnDisconnectedAsync мы будем уведомлять клиентов, о том, что какой-то клиент подключился или отключился от приложения.

Для теста определим следующую веб-страницу index.html:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          <input type="text" id="message" />          <input type="button" id="sendBtn" value="Отправить" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const message = document.getElementById("message").value;                hubConnection.invoke("Send", message) // отправка данных серверу                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (message, connectionId) {                // создаем элемент <b> для connectionId              const userNameElem = document.createElement("b");              userNameElem.textContent = `${connectionId}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.on("Notify", function (message) {                // добавляет элемент для диагностического сообщения              const notifyElem = document.createElement("b");              notifyElem.textContent = message;              const elem = document.createElement("p");              elem.appendChild(notifyElem);              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

Здесь в javascript в функции "Send" получаем обычные сообщения от клиентов, а в функции "Notify" диагностические уведомления.

Таким образом, при подключение и отключение мы сможем уведомлять всех клиентов:



Стоит отметить, что методы OnConnectedAsync срабатывает еще до первой отправки сообщения пользователем. Метод OnDisconnectedAsync срабатывает, если клиент закроет вкладку браузера или перезагрузит страницу.

Также стоит отметить, что если мы откроем в одном браузере две вкладки со страницей, то у нас будут два отдельных подключенных клиента, которые будут иметь разные идентификаторы.

### **Получение информации о контексте HTTP**

С помощью метода GetHttpContext() контекста хаба мы можем получить различную информацию о запросе, например, информацию о браузере пользователя, его ip, куки и т.д., вообщем различные заголовки http. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string message)          {              await Clients.All.SendAsync("Receive", message, Context.ConnectionId);          }            public override async Task OnConnectedAsync()          {              var context = Context.GetHttpContext();              if(context is not null)              {                  // получаем кук name                  if (context.Request.Cookies.ContainsKey("name"))                  {                      if (context.Request.Cookies.TryGetValue("name", out var userName))                      {                          Console.WriteLine($"name = {userName}");                      }                  }                  // получаем юзер-агент                  Console.WriteLine($"UserAgent = {context.Request.Headers["User-Agent"]}");                  // получаем ip                  Console.WriteLine($"RemoteIpAddress = {context.Connection?.RemoteIpAddress?.ToString()}");                    await base.OnConnectedAsync();              }          }      }  } |

Следует отметить, что получить заголовки http таким образом мы можем, а вот установить заголовки или поменять их значение или установить те же куки нельзя.

## Взаимодействие с клиентами

Для взаимодействия с клиентами в классе Hub определено свойство **Clients**. Оно представляет всех подключенных клиентов в виде объекта IHubCallerClients. Если в классе хаба необходимо вызвать на клиенте функцию и отправить ей некоторые данные, то мы можем использовать данный объект. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ChatHub : Hub  {      public async Task Send(string message)      {          await Clients.All.SendAsync("Receive", message);      }  } |

В данном случае у всех подключенных клиентов вызывается функция Receive, которой передается строка message.

У объекта Clients есть следующие свойства:

* **All**: представляет всех подключенных клиентов
* **Caller**: представляет только текущего клиента, который обратился к хабу
* **Others**: представляет всех клиентов за исключением текущего клиента, который обратился к хабу

Кроме свойств объект Clients имеет ряд методов:

* **AllExcept(IReadOnlyList<string> connectionIds)**: представляет всех клиентов за исключением тех, id которых передаются в метод
* **Client(string connectionId)**: вызывает метод у клиента по id подключения
* **Clients(IReadOnlyList<string> connectionIds)**: вызывает метод у клиентов, id которых передаются в метод
* **Group(string groupName)**: вызывает метод у клиентов определенной группы
* **GroupExcept(string groupName, IReadOnlyList<string> connectionIds)**: вызывает метод у клиентов группы по имени groupName за исключением тех клиентов, id которых передаются в качестве второго параметра
* **Groups(IReadOnlyList<string> groupNames)**: вызывает метод у клиентов групп, названия которых передаются в метод
* **OthersInGroup(string OthersInGroup)**: вызывает метод у клиентов определенной группы за исключением текущего клиента
* **User(string userId)**: вызывает метод у пользователя по id
* **Users(IReadOnlyList<string> userIds)**: вызывает метод у пользователей, id которых передаются в метод

Обратите внимание, что концепция клиентов и концепция пользователей в данном случае отличается. Клиент по сути представляет отдельное подключение, с которым ассоциируется определенный идентификатор подключения. Если мы откроем в браузере две вкладки, у нас будет два подключения. А пользователи относятся к системе аутентификации приложения.

Например, определим следующий хаб:

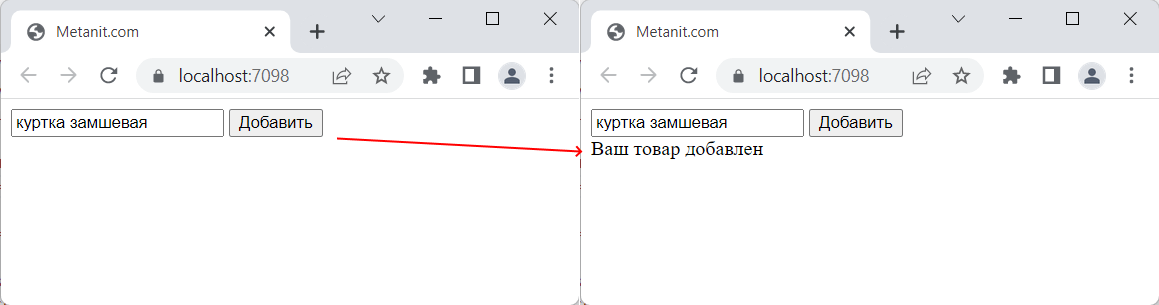
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string product)          {              await Clients.Caller.SendAsync("Notify", "Ваш товар добавлен");              await Clients.Others.SendAsync("Receive", $"Добавлено: {product} в {DateTime.Now.ToShortTimeString()}");          }      }  } |

При вызове метода Send у текущего клиента будет вызываться функция Notify, а у всех остальных клиентов - функция Receive.

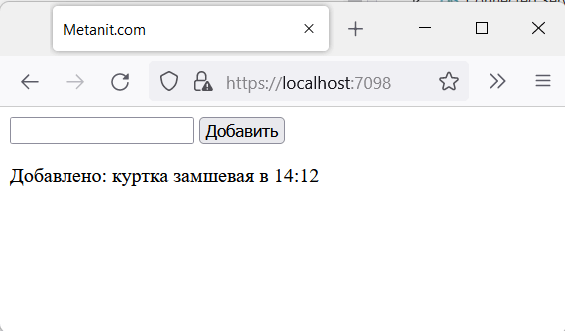
Для теста определим следующую страницу index.html:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          <input type="text" id="product" />          <input type="button" id="sendBtn" value="Добавить" />      </div>      <div id="status"></div>      <div id="info"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const product = document.getElementById("product").value;                hubConnection.invoke("Send", product)                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (message) {                const messageElem = document.createElement("p");              messageElem.textContent = message;                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("info").firstChild;              document.getElementById("info").insertBefore(messageElem, firstElem);          });            hubConnection.on("Notify", function (message) {              document.getElementById("status").innerText = message;          });              hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

И после условного добавления товара клиент получит одно сообщение:



А все остальные подключенные клиенты получат своего рода push-уведомление о произведенных действиях:



Также мы могли бы переписать метод Send в хабе следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string product)          {              await Clients.Client(Context.ConnectionId).SendAsync("Notify",  "Ваш товар добавлен");              await Clients.AllExcept(new List<string> { Context.ConnectionId }) .SendAsync("Receive", $"Добавлено: {product} в {DateTime.Now.ToShortTimeString()}");          }      }  } |

В данном случае метод Send выполняет все те же действия, только теперь для определения клиентов для отправки сообщений применяется идентификатор клиента, который получаем из контекста хаба.

## Отправка сложных объектов

В прошлых темах из хаба в клиент и из клиента в хаб передавались довольно простые данные - одна или две строки. Однако SignalR позволяет связывать наборы данных со сложными объектами. Это значит, что мы можем получать в хабе данные со сложной комплексной структурой.

Например, определим следующий класс хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(User user)          {              user.Age += 5;              await Clients.Caller.SendAsync("Receive", user);          }      }      public class User      {          public string Name { get; set; } = "";          public int Age { get; set; }      }  } |

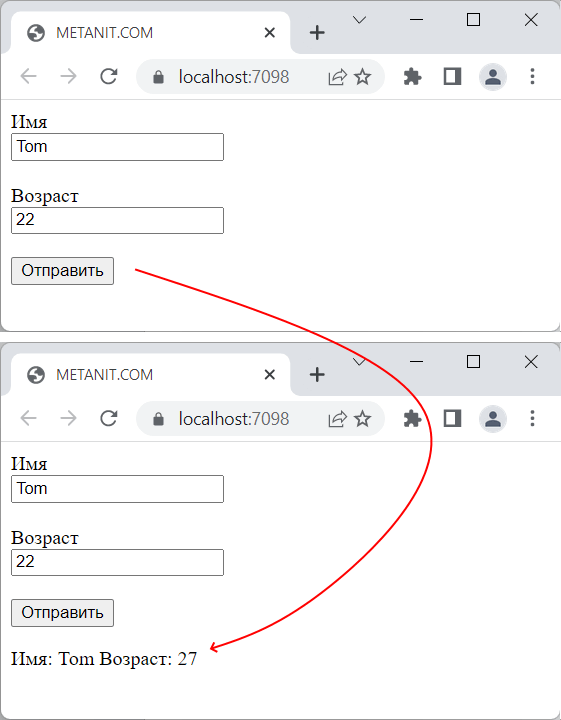
Для демонстрации здесь хаб получает данные в виде объекта User, изменяет значение его свойства Age и отправляет обратно текущему клиенту.

Также определим следующую веб-страницу для взаимодействия с хабом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <div>          <label>Имя</label><br />          <input type="text" id="name" /><br /><br />          <label>Возраст</label><br />          <input type="number" id="age" /><br /><br />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div><p id="response"></p></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            // отправка сообщения на сервер          document.getElementById("sendBtn").addEventListener("click", function () {                const name = document.getElementById("name").value;              const age = document.getElementById("age").value;                hubConnection                  .invoke("Send", { "name": name, "age": parseInt(age) })                  .catch(function (err) {                      return console.error(err.toString());              });          });          // получение сообщения от сервера          hubConnection.on("Receive", function (user) {                document.getElementById("response").innerText = `Имя: ${user.name} Возраст: ${user.age}`;          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

Код javascript получает введенные в два поля на странице данные, формирует из них объект со свойствами "name" и "age" и отправляет хабу. Стоит обратить внимание, что отправляемые объект соответствует определению класса User, благодаря чему SignalR сможет связать эти данные с объектом User.

Обратите внимание на то, что при отправке для значения age выполняется преобразование parseInt(age).



Подобным образом можно отправлять массивы и коллекции, в том числе коллекции сложных данных. Например, определим хаб, который принимает список строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(List<string> items)          {              await Clients.Caller.SendAsync("Receive", items);          }      }  } |

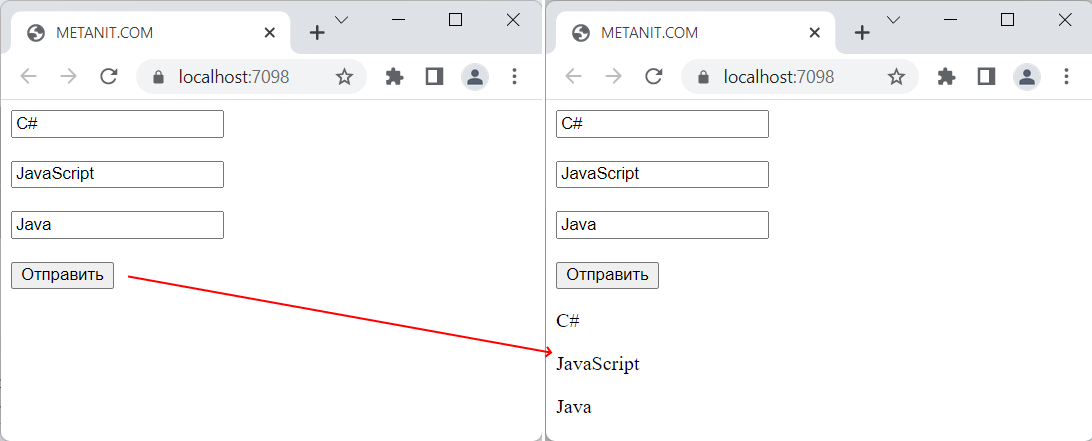
Полученный список отправляет обратно клиенту.

На клиенте определим отправку и получение набора строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <div>          <input type="text" name="items" /><br /><br />          <input type="text" name="items" /><br /><br />          <input type="text" name="items" /><br /><br />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div><p id="response"></p></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            // отправка сообщения на сервер          document.getElementById("sendBtn").addEventListener("click", function () {                const itemElems = document.getElementsByName("items");              const items = [];              for (let i = 0; i < itemElems.length; i++) {                  items.push(itemElems[i].value);              }                hubConnection                  .invoke("Send", items)                  .catch(function (err) {                      return console.error(err.toString());              });          });          // получение сообщения от сервера          hubConnection.on("Receive", function (data) {                document.getElementById("response").innerHTML = "";              for (let i = 0; i < data.length; i++) {                    document.getElementById("response").innerHTML += `<p>${data[i]}</p>`;              }          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

На странице три поля ввода, имя которых равно "items". В коде javascript считываем значения этих полей, добавляем их в массив и отправляем хабу.

В функции Receive, наоборот, получаем пришедшие данные и выводим на страницу.



## IHubContext и отправка сообщений вне хаба

Класс хаба - не единственное место, где мы можем отправлять сообщения клиентам. SignalR позволяет это делать также в контроллерах, страницах Razor Pages, компонентах middleware, сервисах благодаря внедрению зависимости **IHubContext**. Объект IHubContext реализует часть функциональности стандартного класса хаба, в частности, в нем определены такие же свойства **Groups** и **Clients**, которые позволяют манипулировать группами и отправлять сообщения всем подключенным клиентам.

Например, определим хаб ChatHub:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {      }  } |

Несмотря на то, что класс хаба пустой, он нам понадобится для типизиации объекта IHubContext, кроме того, взаимодействие с клиентами будет идти через данный хаб.

Далее в файле **Program.cs** определим конечную точку, которая будет принимать данные и, используя функционал хаба SignalR, регтранслировать эти данные всем подключенным клиентам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using Microsoft.AspNetCore.SignalR;     // для IHubContext  using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapPost("create", async (Message message, IHubContext<ChatHub> hubContext) =>  {      await hubContext.Clients.All.SendAsync("Receive", $"{message.Text} - {DateTime.Now.ToLongTimeString()}");  });    app.MapHub<ChatHub>("/chat");  app.Run();    record class Message(string Text); |

Итак, здесь по прежнему клиенты подключаются к хабу ChatHub по пути "/chat". Но теперь данные получает конечная точка, которая определена с помощью метода MapPost и которая соответственно обрабатывает POST-запросы.

В качестве первого параметра обработчик конечной точки принимает собственно отправленные пользователем данные в виде объекта Message, который имеет строковое свойство Text.

В качестве второго параметра в обработчик конечной точки передается объект IHubContext, который мы можем получить благодаря встроенному механизму внедрения зависимостей. Основное ограничение в данном случае - этот объект должен быть типизирован классом хаба.

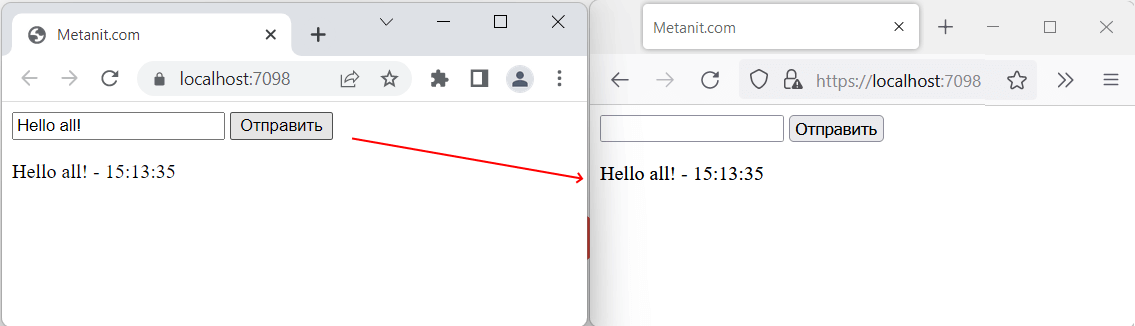
Затем мы можем использовать данный IHubContext, например, отправив сообщение подключенным клиентам. К примеру, в данном случае конечная точка получает отправленные данные в post-запросе по адресу "/create", добавляет к ним текущее время и отправляет их всем подключенным клиентам.

Для тестирования определим в папке **wwwroot** следующую веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div id="inputForm">          <input type="text" id="message" />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click",()=> {              const message = document.getElementById("message").value;                fetch("create", {                  method: "POST",                  headers: { "Content-Type": "application/json" },                  body: JSON.stringify({                      "text": message                  })              })              .catch(error => console.error(error));          });            hubConnection.on("Receive",  message =>{                const messageElement = document.createElement("p");              messageElement.textContent = message;                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(messageElement, firstElem);          });            hubConnection.start()              .then(() => document.getElementById("sendBtn").disabled = false)              .catch((err) =>console.error(err) );      </script>  </body>  </html> |

На данной веб-странице определено поле для ввода некоторого сообщения, и после нажатия на кнопку это сообщение с помощью функции fetch отправляется на сервер по адресу "/create". Причем стоит отметить, что также идет подключение идет по адресу "/chat" - то есть фактически к хабу ChatHub. При получении уведомлений от хаба будет вызываться функция receive.

И после отправки формы все остальные подключенные клиенты увидят информацию об отправленных данных.



Подобным образом можно получать сервис IHubContext также как и любyю другую зависимость в других сервисах, компонентах middleware, контроллерах MVC и страницах Razor Pages.

Однако стоит отметить, что данный способ отправки сообщений клиентам имее некоторые недостатки. Например, у свойства Clients доступно только свойство All - то есть мы можем отправить сообщение только всем клиентам и нету таких свойств как Other или Caller, которые доступны при использовании в классе хаба. Также мы не можем получить из IHubContext id подключения. Поэтому если нам потребуется в контроллерах или других классах получить id подключения, то его можно получить при подключении на клиенте и затем пересылать тем или иным образом, например, через куки или через параметры запроса.

К примеру, определим следующий код в файле **Program.cs**:

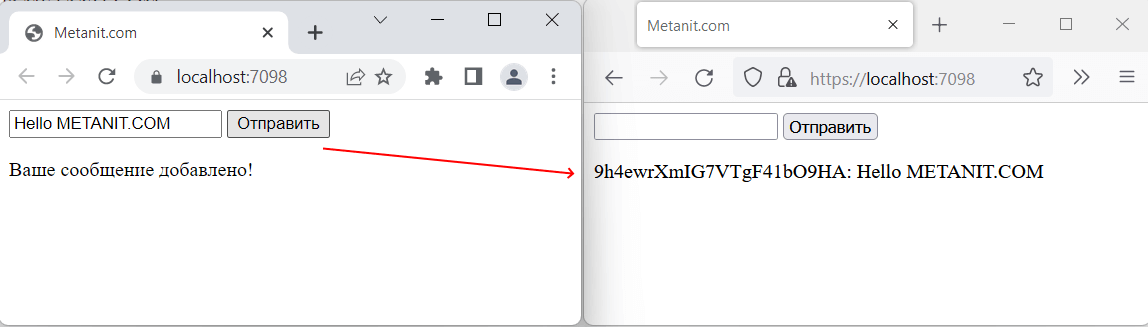
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using Microsoft.AspNetCore.SignalR;     // для IHubContext  using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddSignalR();    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapPost("create", async (Message message, IHubContext<ChatHub> hubContext) =>  {      await hubContext.Clients.AllExcept(message.ConnectionId).SendAsync("Receive", $"{message.ConnectionId}: {message.Text}");      await hubContext.Clients.Client(message.ConnectionId).SendAsync("Receive", "Ваше сообщение добавлено!");  });    app.MapHub<ChatHub>("/chat");  app.Run();    record class Message(string Text, string ConnectionId); |

Теперь конечная точка также получает объект Message, но который теперь содержит свойство ConnectionId для хранения id подключение отправителя. И в зависимости от того, кто отправитель, отправляет то или иное сообщение.

Также изменим веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div id="inputForm">          <input type="text" id="message" />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          let connectionId = ""; // id подключения          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click",()=> {              const message = document.getElementById("message").value;                fetch("create", {                  method: "POST",                  headers: { "Content-Type": "application/json" },                  body: JSON.stringify({                      "text": message,                      "connectionid": connectionId                  })              })              .catch(error => console.error(error));          });            hubConnection.on("Receive",  message =>{                const messageElement = document.createElement("p");              messageElement.textContent = message;                const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(messageElement, firstElem);          });            hubConnection.start()              .then(() => {                  document.getElementById("sendBtn").disabled = false;                  // после соединения получаем id подключения                  console.log(hubConnection.connectionId);                  connectionId = hubConnection.connectionId;              })              .catch((err) =>console.error(err) );      </script>  </body>  </html> |

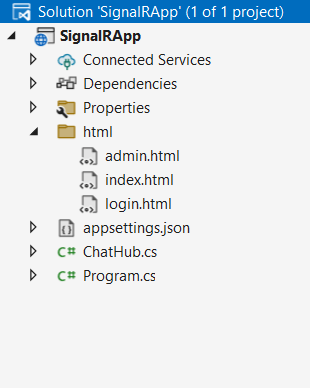
Теперь страница отправляет ajax-запросы к методу Create контроллера Home. После подключения к хабу через функцию hubConnection.start() получаем id подключения. И затем при отправке включаем его в тело запроса.



## Аутентификация и авторизация на основе куки

SignalR позволяет использовать встроенные механизмы аутентификации и авторизации ASP.NET Core для разграничения доступа к хабам и их функциональности. В данном случае рассмотрим применение аутентификации на основе куки и авторизации по ролям в приложении с SignalR. Подробнее же про аутентификацию на основе куки и авторизацию ролей в asp.net core можно посмотреть в соответствующих статьях: [Аутентификация с помощью куки](https://metanit.com/sharp/aspnet6/13.4.php) и [Авторизация по ролям](https://metanit.com/sharp/aspnet6/13.7.php)

Возьмем простейший проект, где применяется простейшая аутентификация и авторизация на основе куки со следующей структурой:



### Определение хаба SignalR

Для тестирования аутентификации и авторизации в SignalR определим следующий класс хаб **ChatHub**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using Microsoft.AspNetCore.Authorization; // для атрибута Authorize  using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      [Authorize]      public class ChatHub : Hub      {          public async Task Send(string message, string userName)          {              await Clients.All.SendAsync("Receive", message, userName);          }          [Authorize(Roles = "admin")]          public async Task Notify(string message)          {              await Clients.All.SendAsync("Receive", message, "Администратор");          }      }  } |

Для организации доступа к хабу и его методам можно применять атрибут **AuthorizeAttribute**. В данном случае функциональность хаба доступна только для аутентифицировнных пользователей, то есть тех, кто условно залогинился в приложении. Таким образом, анонимные пользователи не смогут вызвать методы хаба, даже если в их распоряжении будет код javascript, который обращается к этим методам.

Кроме того, для метода Notify указано еще одно ограничение - доступ только для тех пользователей, которые принадлежат роли "admin". Поэтому когда у хаба будет вызван данный метод, приложение будет знать, что метод вызван администратором. В остальном оба метода одинаковы вызывают на клиенте функцию "Receive".

### Определение клиентской части

Для определения клиентской части создадим в проекте новую папку, которую назовем **html** - она будет хранить файлы html. Прежде всего добавим в нее новый файл **login.html**, который будет содержать форму для входа в приложение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <h2>Вход в приложение</h2>      <form method="post">          <p>              <label>Email</label><br />              <input name="email" />          </p>          <p>              <label>Password</label><br />              <input type="password" name="password" />          </p>          <input type="submit" value="Login" />      </form>  </body>  </html> |

По нажатию на кнопку введенные email и пароль будут отправляться обратно приложению в POST-запросе.

Также добавим в папку **html** новую страницу **index.html**, которая будет предназначена для обычных пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <h2>Личный кабинет пользователя</h2>      <p><a href="logout">Выйти</a></p>      <div id="userForm">          <p>              Введите ник:<br />              <input id="username" type="text" />          </p>          <p>              Введите сообщение:<br />              <input type="text" id="message" />          </p>          <input type="button" id="sendBtn" value="Отправить" />      </div>        <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            // отправка сообщения от простого пользователя          document.getElementById("sendBtn").addEventListener("click", () => {              const message = document.getElementById("message").value;              const username = document.getElementById("username").value;              hubConnection.invoke("Send", message, username)                  .catch(error => console.error(error));          });          // получение сообщения от сервера          hubConnection.on("Receive", (message, userName) => {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                var firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(() => document.getElementById("sendBtn").disabled = false)              .catch((err) => console.error(err));      </script>  </body>  </html> |

По нажатию на кнопку на странице хабу будет отправлено сообщение с ником пользователя.

И также добавим в папку **html** новую страницу **admin.html**, которая будет предназначена для администратора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <h2>Личный кабинет администратора</h2>      <p><a href="logout">Выйти</a></p>      <div id="userForm">          <p>              Введите ник:<br />              <input id="username" type="text" />          </p>          <p>              Введите сообшение:<br />              <input type="text" id="message" />          </p>          <input type="button" id="sendBtn" value="Отправить" />      </div>        <br />      <div id="adminForm">          <input type="text" id="notify" />          <input type="button" id="notifyBtn" value="Уведомление" />      </div>        <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            // отправка сообщения от простого пользователя          document.getElementById("sendBtn").addEventListener("click", () => {              const message = document.getElementById("message").value;              const username = document.getElementById("username").value;              hubConnection.invoke("Send", message, username)                  .catch(error => console.error(error));          });          // отправка сообщения от администратора          document.getElementById("notifyBtn").addEventListener("click", () => {              const message = document.getElementById("notify").value;              hubConnection.invoke("Notify", message);          });            // получение сообщения от сервера          hubConnection.on("Receive", (message, userName) => {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                var firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(() => document.getElementById("sendBtn").disabled = false)              .catch((err) => console.error(err));      </script>  </body>  </html> |

По сути тут тот же код, что и на странице index.html за тем исключением, что администратор также может отправить некоторое специальное уведомление, которое будет отправлено в хаб методу Notify.

### Определение базовой части

Тепеь определим в файле **Program.cs** основную логику приложения:

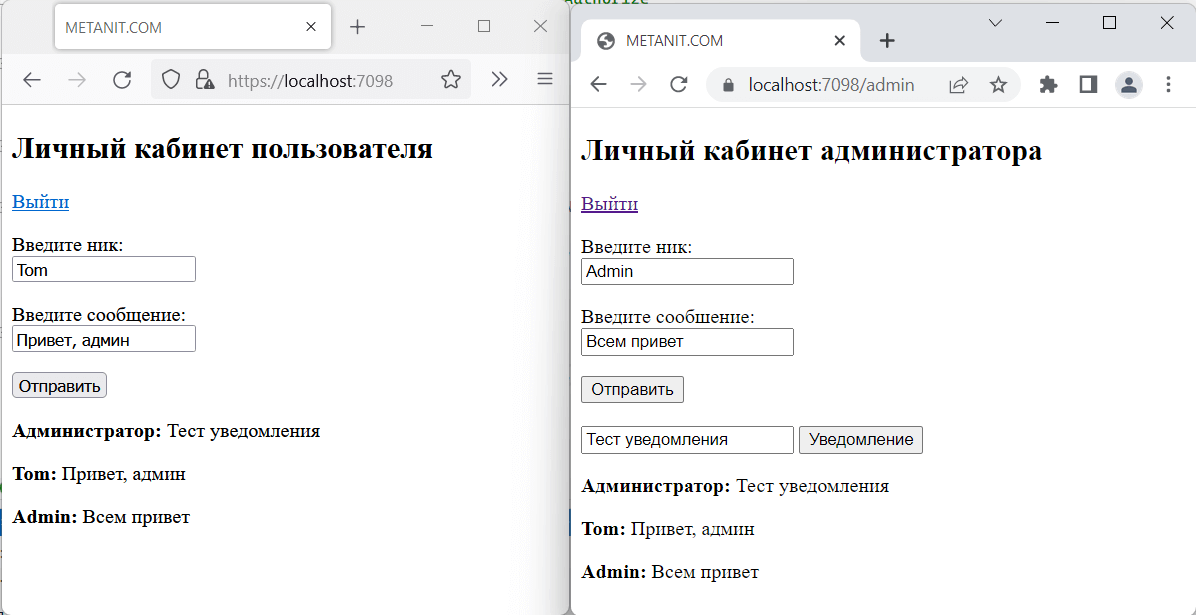
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | using Microsoft.AspNetCore.Authentication.Cookies;  using System.Security.Claims;  using Microsoft.AspNetCore.Authentication;  using Microsoft.AspNetCore.Authorization;  using SignalRApp;   // пространство имен класса ChatHub      var adminRole = new Role("admin");  var userRole = new Role("user");  var people = new List<Person>  {      new Person("tom@gmail.com", "12345", adminRole),      new Person("bob@gmail.com", "55555", userRole),  };    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)      .AddCookie(options => options.LoginPath = "/login");  builder.Services.AddAuthorization();    builder.Services.AddSignalR();    var app = builder.Build();    app.UseAuthentication();   // добавление middleware аутентификации  app.UseAuthorization();   // добавление middleware авторизации    app.MapGet("/login", async context =>      await SendHtmlAsync(context, "html/login.html"));    app.MapPost("/login", async (string? returnUrl, HttpContext context) =>  {      // получаем из формы email и пароль      var form = context.Request.Form;      // если email и/или пароль не установлены, посылаем статусный код ошибки 400      if (!form.ContainsKey("email") || !form.ContainsKey("password"))          return Results.BadRequest("Email и/или пароль не установлены");      string email = form["email"];      string password = form["password"];        // находим пользователя      Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);      // если пользователь не найден, отправляем статусный код 401      if (person is null) return Results.Unauthorized();        var claims = new List<Claim>      {          new Claim(ClaimsIdentity.DefaultNameClaimType, person.Email),          new Claim(ClaimsIdentity.DefaultRoleClaimType, person.Role.Name)      };      var claimsIdentity = new ClaimsIdentity(claims, "Cookies");      var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);      await context.SignInAsync(claimsPrincipal);      return Results.Redirect(returnUrl ?? "/");  });    app.MapGet("/", [Authorize] async (HttpContext context) =>      await SendHtmlAsync(context, "html/index.html"));    app.MapGet("/admin", [Authorize(Roles = "admin")] async (HttpContext context) =>      await SendHtmlAsync(context, "html/admin.html"));      app.MapGet("/logout", async (HttpContext context) =>  {      await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);      return Results.Redirect("/login");  });      app.MapHub<ChatHub>("/chat");  app.Run();    async Task SendHtmlAsync(HttpContext context, string path)  {      context.Response.ContentType = "text/html; charset=utf-8";      await context.Response.SendFileAsync(path);  }  record class  Person(string Email, string Password, Role Role);  record class Role(string Name); |

Для упрощения примера здесь учетные данные пользователей определены и хранятся непосредственно в коде. Пользователь представлен классом Person, а роль пользователя - классом Role.

При обращении по адресу "/login" сработает конечная точка app.MapGet("/login"), который отправит пользователю страницу login.html для ввода логина и пароля. После отправки этих данных их получит конечная точка app.MapPost("/login"), который, в случае если пользователь найден, установит аутентификацирнные куки.

Конечная точка app.MapGet("/") возвращает пользователям страницу index.html. Если же пользователь имеет роль администратора, то ему также будет доступен функционал конечной точки app.MapGet("/admin"), которая отправит страницу для администратора.

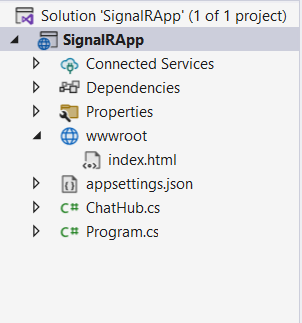
Пример работы хаба:



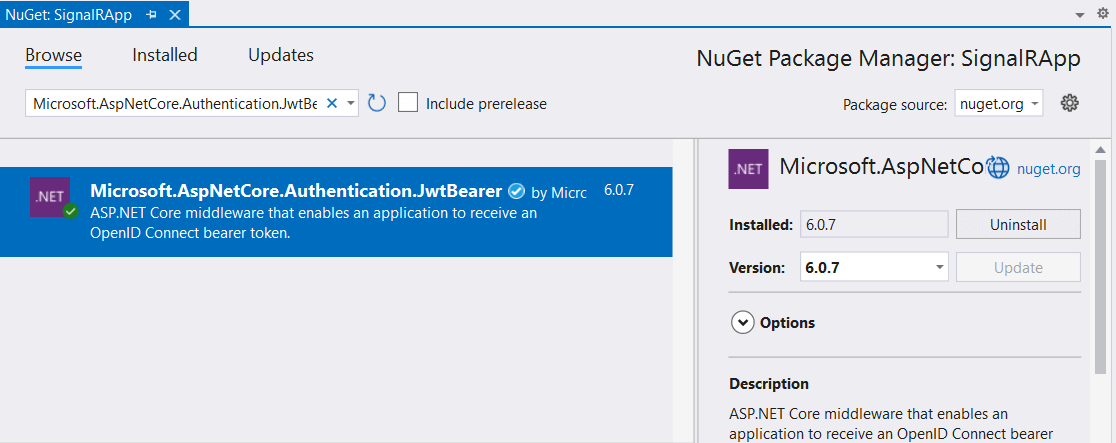
## Аутентификация и авторизация с помощью токенов

В прошлой статье рассматривалась аутентификация и авторизация на основе кук. Другим расспространенным способом аутентификации и авторизации представляет использование токенов. Рассмотрим, как использовать токены в приложении SignalR.

Подробнее про аутентификацию и авторизацию токенов в asp.net можно посмотреть в соответствующих статьях: [Аутентификация с помощью JWT-токенов](https://metanit.com/sharp/aspnet6/13.2.php) и [Авторизация с помощью JWT-токенов в клиенте JavaScript](https://metanit.com/sharp/aspnet6/13.3.php). В данном же случае рассмотрим непосредственное взавимодействие SignalR и механизма аутентификацию с помощью токенов. И для этого возьмем следующий проект:



Прежде всего для использования JWT-токенов в проект ASP.NET Core необходимо добавить Nuget-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**.



### Определение хаба

Пусть у нас будет следующий хаб **ChatHub**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Authorization; // для атрибута Authorize  using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      [Authorize]      public class ChatHub : Hub      {          public async Task Send(string message, string userName)          {              await Clients.All.SendAsync("Receive", message, userName);          }      }  } |

Поскольку к классу применяется атрибут Authorize, то доступ к хабу установлен только для аутентифицировнных пользователей. В самом хабе метод Send получает сообщение и имя отправившего его пользователя и ретранслирует его всем подключенным клиентам.

### **Определение основной части**

В файле **Program.cs** определим основную логику приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100 | using Microsoft.AspNetCore.Authentication.JwtBearer;  using Microsoft.IdentityModel.Tokens;  using System.IdentityModel.Tokens.Jwt;  using System.Security.Claims;  using System.Text;  using SignalRApp;   // пространство имен класса ChatHub    // условная бд с пользователями  var people = new List<Person>   {      new Person("tom@gmail.com", "12345"),      new Person("bob@gmail.com", "55555")  };    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddAuthorization();  builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)      .AddJwtBearer(options =>      {          options.TokenValidationParameters = new TokenValidationParameters          {              ValidateIssuer = true,              ValidIssuer = AuthOptions.ISSUER,              ValidateAudience = true,              ValidAudience = AuthOptions.AUDIENCE,              ValidateLifetime = true,              IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),              ValidateIssuerSigningKey = true          };            options.Events = new JwtBearerEvents          {              OnMessageReceived = context =>              {                  var accessToken = context.Request.Query["access\_token"];                    // если запрос направлен хабу                  var path = context.HttpContext.Request.Path;                  if (!string.IsNullOrEmpty(accessToken) && path.StartsWithSegments("/chat"))                  {                      // получаем токен из строки запроса                      context.Token = accessToken;                  }                  return Task.CompletedTask;              }          };      });    builder.Services.AddSignalR();    var app = builder.Build();      app.UseDefaultFiles();  app.UseStaticFiles();    app.UseAuthentication();   // добавление middleware аутентификации  app.UseAuthorization();   // добавление middleware авторизации      app.MapPost("/login",  (Person loginModel) =>  {      // находим пользователя      Person? person = people.FirstOrDefault(p => p.Email == loginModel.Email && p.Password == loginModel.Password);      // если пользователь не найден, отправляем статусный код 401      if (person is null) return Results.Unauthorized();        var claims = new List<Claim> { new Claim(ClaimTypes.Name, person.Email) };      // создаем JWT-токен      var jwt = new JwtSecurityToken(              issuer: AuthOptions.ISSUER,              audience: AuthOptions.AUDIENCE,              claims: claims,              expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)),              signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));      var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);        // формируем ответ      var response = new      {          access\_token = encodedJwt,          username = person.Email      };        return Results.Json(response);  });    app.MapHub<ChatHub>("/chat");  app.Run();    record class Person(string Email, string Password);  public class AuthOptions  {      public const string ISSUER = "MyAuthServer"; // издатель токена      public const string AUDIENCE = "MyAuthClient"; // потребитель токена      const string KEY = "mysupersecret\_secretsecretsecretkey!123";   // ключ для шифрации      public static SymmetricSecurityKey GetSymmetricSecurityKey() =>          new SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));  } |

Для представления отдельного пользователя здесь определен класс Person с двумя свойствами, а для целей тестирования в приложении опреден список объектов Person. Для хранения настроек токена определен вспомогательный класс AuthOptions.

В отличие от стандартной работы с токеном в webapi здесь нам еще надо получить токен из строки запроса, если запрос обращен к хабу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | options.Events = new JwtBearerEvents  {      OnMessageReceived = context =>      {          var accessToken = context.Request.Query["access\_token"];            // если запрос направлен хабу          var path = context.HttpContext.Request.Path;          if (!string.IsNullOrEmpty(accessToken) && path.StartsWithSegments("/chat"))          {              // получаем токен из строки запроса              context.Token = accessToken;          }          return Task.CompletedTask;      }  }; |

То есть, если запрос идет по пути "/chat", а в строке запроса передан параметр "access\_token" (который хранит токен), то токен извлекается и устанавливается для свойства context.Token. Таким образом, мы сможем идентифицировать текущего пользователя.

Для непосредственной аутентификации пользователя и выдачи токена определена конечная точка app.MapPost("/login"), которая через post-запрос получает логин и пароль пользователя и и возвращает токен и имя пользователя.

### Создание клиентской части

Для тестирования определим в проекте в папке **wwwroot** следующую веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <div id="loginBlock">          <p>              Введите логин:<br />              <input id="email" type="text" />          </p>          <p>              Введите пароль:<br />              <input id="password" type="password" />          </p>          <input id="loginBtn" type="button" value="Войти" />      </div><br />        <div id="inputForm">          <input type="text" id="message" />          <input type="button" id="sendBtn" disabled value="Отправить" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          let token;      // токен          let username;   // имя пользователя          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat", { accessTokenFactory: () => token })              .build();            // аутентификация          document.getElementById("loginBtn").addEventListener("click", async ()=>{                // отправляем запрос на аутентификацию              // посылаем запрос на адрес "/login", в ответ получим токен и имя пользователя              const response = await fetch("/login", {                  method: "POST",                  headers: { "Content-Type": "application/json" },                  body: JSON.stringify({                      email: document.getElementById("email").value,                      password: document.getElementById("password").value                  })              });                // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const data = await response.json();                  token = data.access\_token;                  username = data.username;                  document.getElementById("loginBtn").disabled = true;                    hubConnection.start()       // начинаем соединение с хабом                      .then(() =>  document.getElementById("sendBtn").disabled = false )                      .catch(err =>  console.error(err.toString()));              }              else {                  // если произошла ошибка, получаем код статуса                  console.log(`Status: ${response.status}`);              }          });            // отправка сообщения от простого пользователя          document.getElementById("sendBtn").addEventListener("click", () => {              const message = document.getElementById("message").value;              hubConnection.invoke("Send", message, username)                  .catch(error => console.error(error));          });          // получение сообщения от сервера          hubConnection.on("Receive", (message, user) => {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${user}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                var firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });      </script>  </body>  </html> |

Изначально токен не установлен, он будет установлен только после прохождении аутентификации на сервере. Для этого по нажатию на кнопку loginBtn посылается POST-запрос на сервер по адресу "/login" и взамен получаем токен и имя пользователя, которые сохраняются в глобальные переменные на веб-странице.

И только после получения токена запускатся соединение:

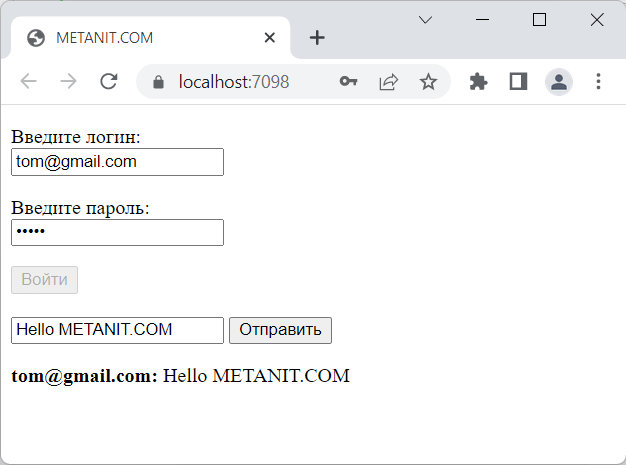
|  |  |
| --- | --- |
| 1 | hubConnection.start() |

При этом при установке соединения в SignalR в клиенте javascript необходимо дополнительно передавать токен:

|  |  |
| --- | --- |
| 1  2  3 | const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat", { accessTokenFactory: () => token })              .build(); |

То есть SignalR на стороне клиент не добавляет токен в заголовки запроса. И при обращении к хабу токен фактически будет посылаться как параметр строки запроса.

Результат работы программы:



## Пользователи

Каждый пользователь в SignalR представляет по умолчанию объект **ClaimsPrincipal**. По сути это аутентифицированный пользователь, который осуществил вход в приложение. И посредством аутентификационного тикета в куки или jwt-токена хаб может узнать, что это за пользователь. Используя идентификатор ClaimTypes.NameIdentifier из ClaimsPrincipal, можно отправлять сообщение определенным пользователям.

Концепция пользователя и концепция клиента(подключения) в SignalR отличаются. Пользователь - это учетная запись, под которой осуществлен вход в приложение. Можно одновременно выполнить вход в приложение сразу на нескольких устройствах, допустим, на ПК в нескольких браузерах и на мобильном устройстве. И если отправить этому пользователю сообщение, то этот пользователь увидит отправленное сообщение во всех браузерах и устройствах, в которых он зашел в приложение.

Подключение, в отличие от пользователя, создается даже на каждую отдельную вкладку браузера. Например, если в одном браузере открыть две вкладки и на обеих вкладках подключиться к хабу, то на каждой вкладке будут разные подключения с разными идентификаторами.

### Получение текущего пользователя в хабе

Если пользователь авторизован, то внутри хаба мы можем получить этого пользователя через свойство **Context.User**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using Microsoft.AspNetCore.Authorization; // для атрибута Authorize  using System.Security.Claims;   // для ClaimTypes  using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      [Authorize]      public class ChatHub : Hub      {          [Authorize]          public async Task Send(string message, string to)          {              var user = Context.User;              var userName = user?.Identity?.Name;              // получаем роль              var userRole = user?.FindFirst(ClaimTypes.Role)?.Value;              // принадлежит ли пользователь роли "admins"              var isAdmin = user?.IsInRole("admin");              //..........          }      }  } |

Это свойство представляет объект ClaimsPrincipal, поэтому мы можем также, как и в представлениях или контроллерах, получить из него формальное имя пользователя (user.Identity.Name), получить роль или значения других клеймов, если они сохранены в аутентификационных куках или токене, узнать, принадлежит ли пользователь определенной роли, а также получить другую сопутствующую информацию.

Но важно учитывать, что некоторые пользователи, которые обращаются к хабу, могут быть аутентифицированы, некоторые могут быть анонимными. Поэтому если мы предоставляем доступ вообще всем пользователям, то при обращении к функционала в Context.User необходимо проверять этот объект на null. Либо применять атрибут авторизации, который исключает обращения анонимных пользователей.

В хабе мы можем не только получить текущего пользователя, но и отправить сообщения определенным пользователям. Для этого можно использовать ряд методов:

* **Clients.User(string userId)**: вызывает метод у пользователя по id
* **Clients.Users(IReadOnlyList<string> userIds)**: вызывает метод у пользователей, id которых передаются в метод

В оба метода передается некий id пользователя. Что это за id? В реальности мы сами можем определить, что будет использоваться в качестве id. Для этого добавим в проект новый класс **CustomUserIdProvider**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.SignalR;  using System.Security.Claims;    // для ClaimTypes    namespace SignalRApp  {      public class CustomUserIdProvider : IUserIdProvider      {          public virtual string? GetUserId(HubConnectionContext connection)          {              return connection.User?.FindFirst(ClaimTypes.NameIdentifier)? .Value;          }      }  } |

Данный класс реализует интерфейс **IUserIdProvider**, который определяет метод **GetUserId()**. Этот метод как раз и возвращает id пользователя. В данном случае он возвращает значение клейма ClaimTypes.NameIdentifier, который хранит имя пользователя. Здесь мы можем возвратить любое другое значение, например, роль пользователя, все зависит от конкретного приложения и конкретной задачи.

### Определение хаба

Определим следующий класс хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | using Microsoft.AspNetCore.Authorization; // для атрибута Authorize  using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      [Authorize]      public class ChatHub : Hub      {          public async Task Send(string message, string to)          {              // получение текущего пользователя, который отправил сообщение              //var userName = Context.UserIdentifier;              if (Context.UserIdentifier is string userName)              {                  await Clients.Users(to, userName).SendAsync("Receive", message, userName);              }          }            public override async Task OnConnectedAsync()          {              await Clients.All.SendAsync("Notify", $"Приветствуем {Context.UserIdentifier}");              await base.OnConnectedAsync();          }      }  } |

При подключение пользователя у всех подключенных клиентов будет вызываться функция Notify, которой передается сообщение общего характера. В методе Send получаем сообщение и идентификатор пользователя, которому предназначено это сообщение. В самих методах мы можем получить идентификатор текущего пользователя через свойство **Context.UserIdentifier**. По сути это будет значение клейма ClaimTypes.NameIdentifier. Далее сообщение отправляется получателю, которому предназначено сообщение, и текущему пользователю, которой собственно и отправляет сообщение.

### Определение Program.cs

В файле **Program.cs** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105 | using Microsoft.AspNetCore.Authentication.JwtBearer;  using Microsoft.IdentityModel.Tokens;  using System.IdentityModel.Tokens.Jwt;  using System.Security.Claims;  using System.Text;  using SignalRApp;   // пространство имен класса ChatHub  using Microsoft.AspNetCore.SignalR; // для IUserIdProvider      // условная бд с пользователями  var people = new List<Person>   {      new Person("tom@gmail.com", "11111"),      new Person("bob@gmail.com", "55555"),      new Person("sam@gmail.com", "22222")  };    var builder = WebApplication.CreateBuilder(args);   // Устанавливаем сервис для получения Id пользователя  builder.Services.AddSingleton<IUserIdProvider, CustomUserIdProvider>();    builder.Services.AddAuthorization();  builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)      .AddJwtBearer(options =>      {          options.TokenValidationParameters = new TokenValidationParameters          {              ValidateIssuer = true,              ValidIssuer = AuthOptions.ISSUER,              ValidateAudience = true,              ValidAudience = AuthOptions.AUDIENCE,              ValidateLifetime = true,              IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),              ValidateIssuerSigningKey = true          };            options.Events = new JwtBearerEvents          {              OnMessageReceived = context =>              {                  var accessToken = context.Request.Query["access\_token"];                    // если запрос направлен хабу                  var path = context.HttpContext.Request.Path;                  if (!string.IsNullOrEmpty(accessToken)  && path.StartsWithSegments("/chat"))                  {                      // получаем токен из строки запроса                      context.Token = accessToken;                  }                  return Task.CompletedTask;              }          };      });    builder.Services.AddSignalR();    var app = builder.Build();      app.UseDefaultFiles();  app.UseStaticFiles();    app.UseAuthentication();   // добавление middleware аутентификации  app.UseAuthorization();   // добавление middleware авторизации      app.MapPost("/login",  (Person loginModel) =>  {      // находим пользователя      Person? person = people.FirstOrDefault(p => p.Email == loginModel.Email && p.Password == loginModel.Password);      // если пользователь не найден, отправляем статусный код 401      if (person is null) return Results.Unauthorized();        var claims = new List<Claim> { new Claim(ClaimTypes.NameIdentifier, person.Email) };      // создаем JWT-токен      var jwt = new JwtSecurityToken(              issuer: AuthOptions.ISSUER,              audience: AuthOptions.AUDIENCE,              claims: claims,              expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)),              signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));      var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);        // формируем ответ      var response = new      {          access\_token = encodedJwt,          username = person.Email      };        return Results.Json(response);  });    app.MapHub<ChatHub>("/chat");  app.Run();    record class  Person(string Email, string Password);  public class AuthOptions  {      public const string ISSUER = "MyAuthServer"; // издатель токена      public const string AUDIENCE = "MyAuthClient"; // потребитель токена      const string KEY = "mysupersecret\_secretsecretsecretkey!123";   // ключ для шифрации      public static SymmetricSecurityKey GetSymmetricSecurityKey() =>          new SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));  } |

В данном случае применяется аутентификация на основе токенов, которая была рассмотрена в прошлой теме. Соответственно для ее работы в проект необходимо добавить Nuget-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**.

Для представления пользователя определен класс Person, и в приложении для тестирования определена условная база данных - список с тремя объектами Person.

При обращении по адресу "/login" клиенту отправляет токен для последующего взаимодействия с хабом.

При этом следует отметить два момента. Во-первых, установка сервиса получения id пользователя в хабах SignalR:

|  |  |
| --- | --- |
| 1 | builder.Services.AddSingleton<IUserIdProvider, CustomUserIdProvider>(); |

Во-вторых, при получении данных пользователя при логине в приложение и установке токена логин пользователя будет связан с клеймом ClaimTypes.NameIdentifier

|  |  |
| --- | --- |
| 1 | var claims = new List<Claim> { new Claim(ClaimTypes.NameIdentifier, person.Email) }; |

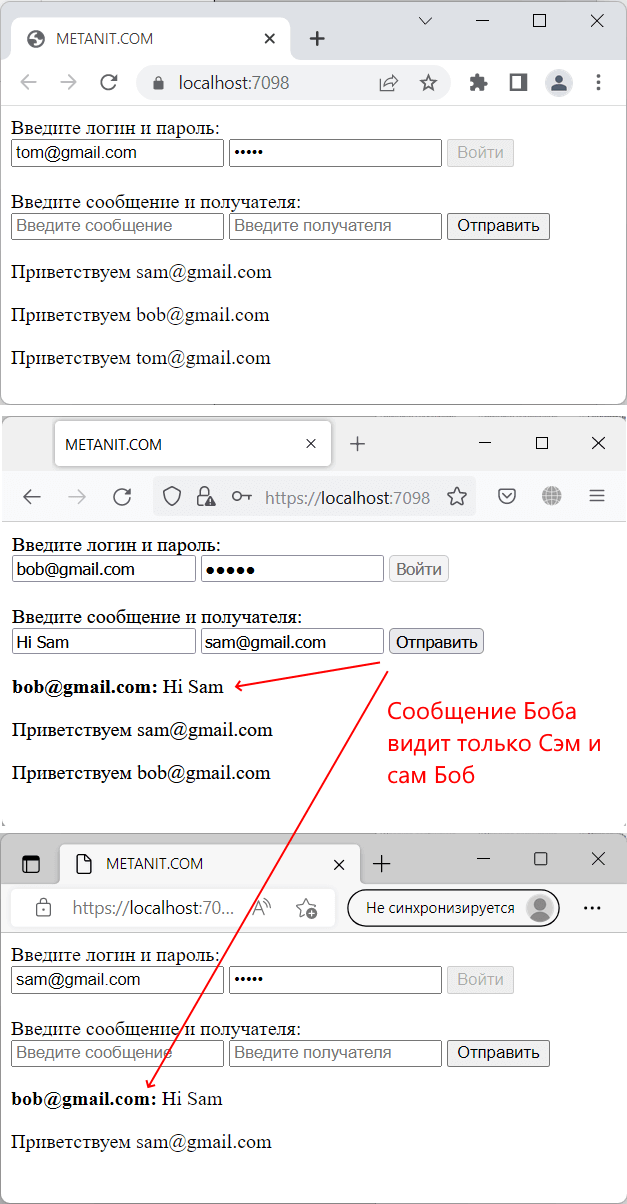
### **Определение клиентской части**

Для тестирования определим в проекте в папке **wwwroot** следующую веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <div id="loginBlock">          Введите логин и пароль:<br />          <input id="email" type="text" />          <input id="password" type="password" />          <input id="loginBtn" type="button" value="Войти" />      </div><br />        <div id="inputForm">          Введите сообщение и получателя:<br />          <input type="text" id="message" placeholder="Введите сообщение" />          <input type="text" id="receiver" placeholder="Введите получателя" />          <input type="button" id="sendBtn" disabled value="Отправить" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          let token;      // токен          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat", { accessTokenFactory: () => token })              .build();            // аутентификация          document.getElementById("loginBtn").addEventListener("click", async () => {                const response = await fetch("/login", {                  method: "POST",                  headers: { "Content-Type": "application/json" },                  body: JSON.stringify({                      email: document.getElementById("email").value,                      password: document.getElementById("password").value                  })              });                // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const data = await response.json();                  token = data.access\_token;                  username = data.username;                  document.getElementById("loginBtn").disabled = true;                    hubConnection.start()       // начинаем соединение с хабом                      .then(() => document.getElementById("sendBtn").disabled = false)                      .catch(err => console.error(err.toString()));              }              else {                  // если произошла ошибка, получаем код статуса                  console.log(`Status: ${response.status}`);              }          });            // отправка сообщения от простого пользователя          document.getElementById("sendBtn").addEventListener("click", () => {                const message = document.getElementById("message").value;              const receiver = document.getElementById("receiver").value;              hubConnection.invoke("Send", message, receiver)                  .catch(error => console.error(error));          });            // получение сообщения от пользователя          hubConnection.on("Receive", (message, user) => {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${user}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            // получени общего уведомления          hubConnection.on("Notify", message => {                const elem = document.createElement("p");              elem.textContent = message;                const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });      </script>  </body>  </html> |

На веб-странице вначале осуществляем вход в приложение, как было описано в прошлой статье, затем отправляем сообщение определенному пользователю.

Таким образом, пользователи смогут отправлять сообщения только определенным пользователям:



## Группы

Группы в SignalR представляют коллекцию подключений, которые ассоциированы с именем группы. SignalR позволяет добавлять и удалять клиентов из группы. Один клиент может входить сразу в несколько групп.

Для работы с группами в классе Hub определено свойство **Groups**, которое представляет объект **IGroupManager**. Он имеет два метода:

* **AddToGroupAsync(connectionId, groupName)**: добавляет клиента с идентификатором connectionId в группу с именем groupName
* **RemoveFromGroupAsync(connectionId, groupName)**: удаляет клиента с идентификатором connectionId из группы

Также у объекта IHubCallerClients определено ряд методов, которые позволяют выбрать определенные группы для отправки им сообщения. Для вызова метода у клиентов в группах можно использовать ряд методов:

* **Group(string groupName)**: вызывает метод у клиентов определенной группы
* **GroupExcept(string groupName, IReadOnlyList<string> connectionIds)**: вызывает метод у клиентов группы по имени groupName за исключением тех клиентов, id которых передаются в качестве второго параметра
* **Groups(IReadOnlyList<string> groupNames)**: вызывает метод у клиентов групп, названия которых передаются в метод
* **OthersInGroup(string OthersInGroup)**: вызывает метод у клиентов определенной группы за исключением текущего клиента

Например, определим следующий хаб:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Enter(string username, string groupName)          {              await Groups.AddToGroupAsync(Context.ConnectionId, groupName);              await Clients.All.SendAsync("Notify", $"{username} вошел в чат в группу {groupName}");          }          public async Task Send(string message, string userName, string groupName)          {              await Clients.Group(groupName).SendAsync("Receive", message, userName);          }      }  } |

В хабе определено два метода. В методе Enter хаб получает имя пользователя и группы, в которую он вошел. Имя групп может быть любым, важно только учитывать, что оно регистрозависимое. Пользователь добавляется в эту группу, а всем клиентам отправляется соответствующее сообщение.

В методе Send ретранслируют сообщение клиента всем участникам определенной группы.

Для тестирования можно определить следующую веб-страницу **index.html**:

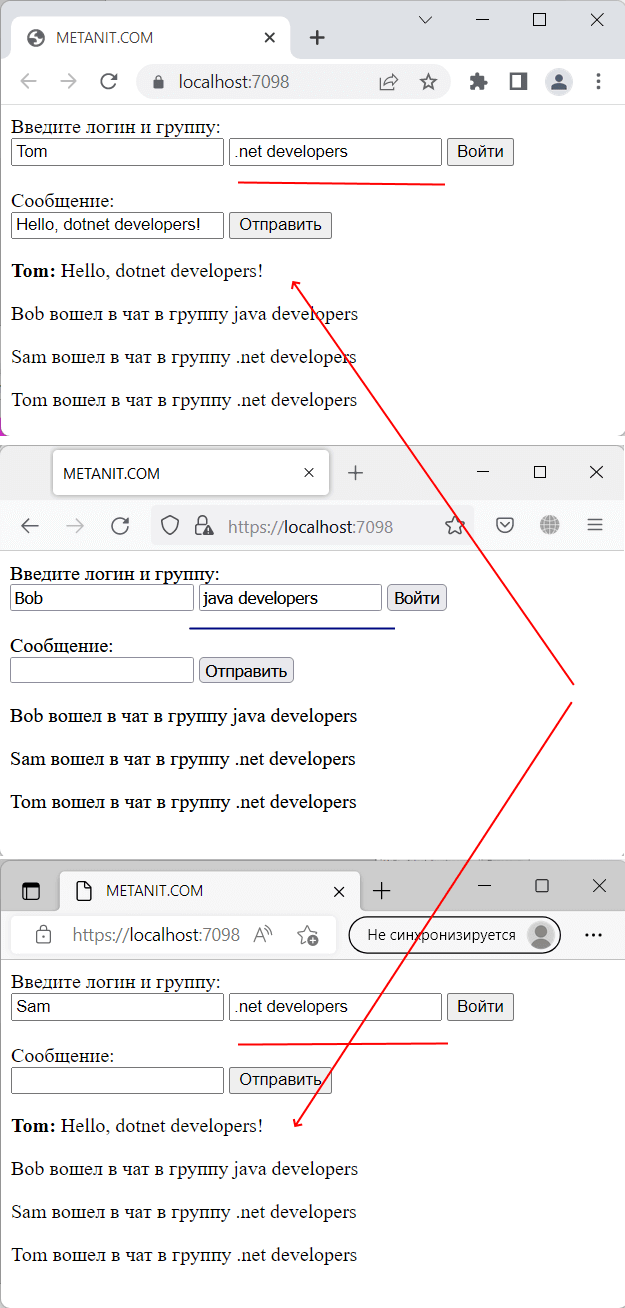
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>METANIT.COM</title>  </head>  <body>      <div id="loginBlock">          Введите логин и группу:<br />          <input id="userName" type="text" placeholder="Введите логин" />          <input id="userGroup" type="text" placeholder="Введите группу" />          <input id="loginBtn" type="button" value="Войти" />      </div><br />          <div id="inputForm">          Сообщение:<br />          <input type="text" id="message" />          <input type="button" id="sendBtn" value="Отправить" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          let userName = "";          let userGroup = "";          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            // установка имени пользователя и вход в группу          document.getElementById("loginBtn").addEventListener("click", function (e) {              userName = document.getElementById("userName").value;              userGroup = document.getElementById("userGroup").value;              hubConnection.invoke("Enter", userName, userGroup);          });            // отправка сообщения в группу          document.getElementById("sendBtn").addEventListener("click", () => {                const message = document.getElementById("message").value;              hubConnection.invoke("Send", message, userName, userGroup)                  .catch(error => console.error(error));          });            // получение сообщения для определенной группы          hubConnection.on("Receive", (message, user) => {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${user}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            // получение общего уведомления          hubConnection.on("Notify", message => {                const elem = document.createElement("p");              elem.textContent = message;                const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(() => {                  document.getElementById("sendBtn").disabled = false;                  document.getElementById("loginBtn").disabled = false              })              .catch ((err) => console.error(err));      </script>  </body>  </html> |

Здесь по нажатию на кнопку loginBtn хабу отправляется имя пользователя и его группы. А по нажатию на кнопку sendBtn отправляется сообщение с указанием имени пользователя и его группы.

Для тестирования определим в файле **Program.cs** следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);  builder.Services.AddSignalR();  var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");  app.Run(); |

Пример работы:



Так, в примере выше пользователи Tom и Sam принадлежат группе .net developers, поэтому только они видят сообщения, предназначенные для этой группы. А пользователь Bob, который принадлежит группе "java developers", не видит эти сообщения. Ему доступны только общие сообщения.

При работе с группами стоит учитывать, группа никак не связана с аутентификацией и авторизацией. Например, в примере хаба выше любой подключившийся к хабу и не состояющий в определенной группе, может обратиться к методу Send и отправить сообщение в эту группу, только это сообщение будет видно только членам этой группы. Группа в SignalR - это не более чем условное объединение клиентов, подключенных к хабу. И мы не можем узнать, какие пользователи состоят в той или иной группе, получить всех пользователей определенной группы.

И если, к примеру, необходимо, создавать какие-нибудь групповые чаты, где могут послать сообщения и соответственно видеть эти сообщения только участники данного чата, необходимо использовать стандартные механизмы авторизации и аутентификации. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Authorize(Roles="admin")]  public async Task Send(string message, string username)  {      await Clients.Group("admins").SendAsync("Send", message, username);  } |

В данном случае в группу "admins" могут посылать сообщения только авторизованные пользователи, у которых роль - "admin".

## Фильтры хабов

Фильтры хабов представляют специальные компоненты, которые выполняют до и после выполнения хабов. В этом плане фильтры хабов напоминают стандартные компоненты middleware, которые встраиваются в конвейер обработки запроса в ASP.NET Core. Фильтры хабов могут быть полезны, если необходимо перед и/или после выполнения хаба определить некоторую программную логику.

Допустим у нас есть следующий хаб ChatHub:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string username, string message)          {              await this.Clients.All.SendAsync("Receive", username, message);          }      }  } |

Определим для него фильтр.

### **Определение фильтра**

Функционально фильтры представляют классы, которые реализуют интерфейс **IHubFilter**. Данный интерфейс определяет три метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface IHubFilter  {      ValueTask<object?> InvokeMethodAsync(HubInvocationContext invocationContext, Func<HubInvocationContext, ValueTask<object?>> next);      Task OnConnectedAsync(HubLifetimeContext context, Func<HubLifetimeContext, Task> next);      Task OnDisconnectedAsync(HubLifetimeContext context, Exception? exception, Func<HubLifetimeContext, Exception?, Task> next);  } |

Этот интерфейс определяет три метода:

* **InvokeMethodAsync**: вызывается при каждом обращении к хабу. Первый параметр - объект HubInvocationContext представляет контекст вызова хаба и хранит всю связанную информацию в виде свойств:
  + Context: контекст хаба - объект **HubCallerContext**
  + Hub: вызываемый хаб в виде объекта класса Hub
  + HubMethod: вызываемый метод хаба - объект MethodName
  + HubMethodName: название вызываемого метода хаба
  + HubMethodArguments: аругменты вызываемого метода хаба - объект IReadOnlyList<object?>
  + ServiceProvider: провайдер сервисов - объект IServiceProvider

Второй параметр - next представляет следующий в конвейере фильтр хаба. Если в конвейере больше нет фильтров, то представляет вызов метода хаба.

Метод возвращает результат работы метода хаба.

* **OnConnectedAsync**: вызывается при обращении к методу OnConnectedAsync хаба. Первый параметр - объект HubLifetimeContext представляет контекст вызова метода OnConnectedAsync и имеет следующие свойства:
  + Context: контекст хаба - объект **HubCallerContext**
  + Hub: вызываемый хаб в виде объекта класса Hub
  + ServiceProvider: провайдер сервисов - объект IServiceProvider

Второй параметр - next представляет следующий в конвейере фильтр хаба. Если в конвейере больше нет фильтров, то представляет вызов метода хаба.

* **OnDisconnectedAsync**: вызывается при обращении к методу OnDisconnectedAsync хаба. Первый параметр - объект HubInvocationContext представляет контекст вызова метода OnDisconnectedAsync

Второй параметр представляет возможную ошибку, которая может произойти при отключении.

Третий параметр - next представляет следующий в конвейере фильтр хаба. Если в конвейере больше нет фильтров, то представляет вызов метода хаба.

Например, определим следующий фильтр хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class MyHubFilter : IHubFilter      {          public async ValueTask<object?> InvokeMethodAsync(              HubInvocationContext invocationContext,              Func<HubInvocationContext, ValueTask<object?>> next)          {              // получаем вызываемый метод хаба              Console.WriteLine($"Вызов метода {invocationContext.HubMethodName}");              try              {                  return await next(invocationContext);   // вызываем следующий фильтр или метод хаба              }              catch (Exception ex)              {                  Console.WriteLine($"Не удалось вызвать метод {invocationContext.HubMethodName}: {ex.Message}");                  throw;              }          }      }  } |

В данном случае просто логгируем на консоль имя метода хаба, который обрабатывает запрос. После логгирования вызываем метод хаба (так как у нас больше нет фильтров хаба)

|  |  |
| --- | --- |
| 1 | await next(invocationContext); |

Результат вызова метода хаба собственно и будет представлять возвращаемый результат.

Чтобы задействовать фильтр, его надо зарегистрировать. Для регистрации фильтра изменим код файла **Program.cs** следующим образом:

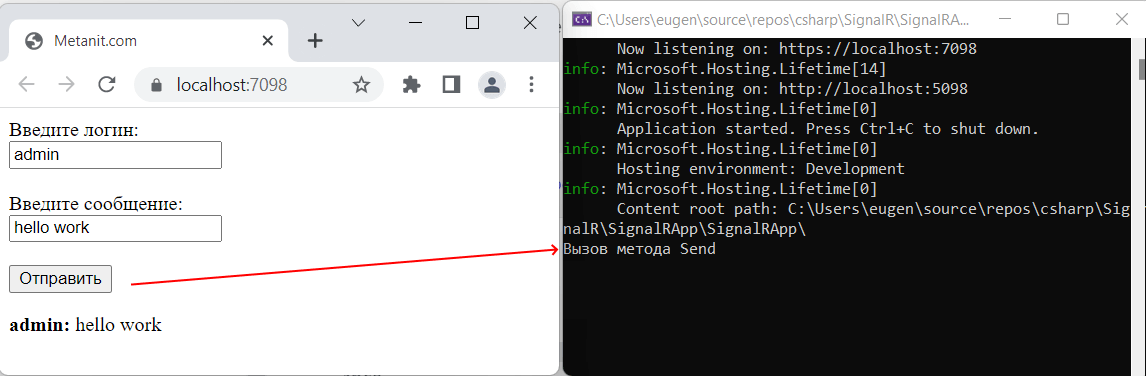
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.AspNetCore.SignalR;  using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    // регистрация фильтра  builder.Services.AddSignalR(options => options.AddFilter<MyHubFilter>());    var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");  app.Run(); |

Для регистрации фильтра применяется перегрузка метода AddSignalR(), который в качестве параметра принимает делегат. Параметр этого делегата - объект HubOptions с помощью метода **AddFilter()** добавляет фильтр в качестве сервиса. Для добавления фильтра метод AddFilter типизируется типом фильтра.

Для тестирования этого хаба в проекте в папке **wwwroot** следующую страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          Введите логин:<br />          <input id="userName" type="text" /><br /><br />          Введите сообщение:<br />          <input type="text" id="message" /><br /><br />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const userName = document.getElementById("userName").value;   // получаем введенное имя              const message = document.getElementById("message").value;                hubConnection.invoke("Send", userName, message) // отправка данных серверу                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (userName, message) {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

И при обращении к хабу сначала сработает метод InvokeMethodAsync() в фильтре хаба, который выведет на консоль уведомление:



Подобным образом можно реализовать и другие методы фильтра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class MyHubFilter : IHubFilter      {          public async ValueTask<object?> InvokeMethodAsync(              HubInvocationContext invocationContext,              Func<HubInvocationContext, ValueTask<object?>> next)          {              // получаем вызываемый метод хаба              Console.WriteLine($"Вызов метода {invocationContext.HubMethodName}");              return await next(invocationContext);   // вызываем следующий фильтр или метод хаба          }            public Task OnConnectedAsync(HubLifetimeContext context, Func<HubLifetimeContext, Task> next)          {              Console.WriteLine("Вызов метода OnConnectedAsync");              return next(context);          }            public Task OnDisconnectedAsync(              HubLifetimeContext context, Exception? exception, Func<HubLifetimeContext, Exception, Task> next)          {              Console.WriteLine("Вызов метода OnDisconnectedAsync");              return next(context, exception!);          }      }  } |

### **Модификация входных значений хаба**

Благодаря объекту HubInvocationContext, который передается в метод InvokeMethodAsync() в фильтре хаба, мы можем проинспектировать передаваемые в метод хаба значения и при необходимости изменить их. Например, нам надо исключить отправку определенных слов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class MyHubFilter : IHubFilter      {          public async ValueTask<object?> InvokeMethodAsync(              HubInvocationContext invocationContext,              Func<HubInvocationContext, ValueTask<object?>> next)          {              // получаем второй параметр метода хаба в переменную message              if (invocationContext.HubMethodArguments.Count == 2 &&                  invocationContext.HubMethodArguments[1] is string message)              {                  // заменяем некоторые слова                  message = message.Replace("блин", "б\*\*\*");                  var arguments = invocationContext.HubMethodArguments.ToArray();                  arguments[1] = message;                  // пересоздаем объект HubInvocationContext                  invocationContext = new HubInvocationContext(invocationContext.Context,                      invocationContext.ServiceProvider,                      invocationContext.Hub,                      invocationContext.HubMethod,                      arguments);              }              // передаем этот объект в вызов последующих фильтров или метода хаба              return await next(invocationContext);          }      }  } |

Так, в методе Send хаба ChatHub второй параметр представлял некоторое сообщение. В данном случае мы получаем это сообщение, заменяем в нем подстроку "блин" на "б\*\*\*". Пересоздаем объект HubInvocationContext, который принимает измененные значения параметров, и передаем его в вызов метода хаба.

### Регистрация глобальных и локальных фильтров

Чтобы задействовать фильтр, его надо зарегистрировать. Это можно сделать глобально - для всех хабов, либо локально - для определенного хаба. Так, в примере выше применялась глобальная регистрациия. Но можно также применить локальную регистрацию - для определенного класса хабов или сочетать оба вида фильтров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using Microsoft.AspNetCore.SignalR;  using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);    builder.Services  // глобальная регистрация фильтра      .AddSignalR(options => options.AddFilter<MyHubFilter>()) // локальная регистрация фильтра     .AddHubOptions<ChatHub>(options => options.AddFilter<MyHubFilter>());  var app = builder.Build();    app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");  app.Run(); |

Для глобальной установки фильтра применяется перегрузка метода AddSignalR().

Для локальной установки фильтра применяется метод AddHubOptions(), который типизируется типом хаба, к которому будут применться настройки. И также в качестве параметра метод принимает делегат, в котором через параметр типа HubOptions и его метода **AddFilter()** добавляется тип фильтра.

Следует учитывать, что сначала выполняются глобальные фильтры, а потом локальные.

Кроме типизации метода AddFilter можно напрямую передать в этот метод объект фильтра:

|  |  |
| --- | --- |
| 1  2  3 | builder.Services      .AddSignalR(options => options.AddFilter(new MyHubFilter())) // глобальная регистрация фильтра      .AddHubOptions<ChatHub>(options => options.AddFilter(new MyHubFilter())); // локальная регистрация фильтра |

### **Время жизни фильтров**

Фильтры внедряются в виде сервисов в приложение. Однако в зависимости от типа регистрации сервис будет иметь определенный тип жизненного цикла. Например, при регистрации в виде типизации метода AddFilter при каждом отдельном обращении к хабу будет создаваться новый объект фильтра:

|  |  |
| --- | --- |
| 1  2  3 | builder.Services      .AddSignalR(options => options.AddFilter<MyHubFilter>()) // глобальная регистрация фильтра      .AddHubOptions<ChatHub>(options => options.AddFilter<MyHubFilter>()); // локальная регистрация фильтра |

Так, в этом примере регистрируется два фильтра одного типа. И при обращении к хабу будет создавать два разных объекта этого фильтра. При новом обращении к хабу опять будет создаваться два новых объекта фильтров. То есть это будет transient-сервис.

Если же в метод AddFilter передается объект фильтра, то будет создваться один фильтр, который будет существовать на протяжении всей жизни приложения, то есть это будет signleton-сервис:

|  |  |
| --- | --- |
| 1  2  3 | builder.Services  // глобальная регистрация фильтра     .AddSignalR(options => options.AddFilter(new MyHubFilter())) // локальная регистрация фильтра     .AddHubOptions<ChatHub>(options => options.AddFilter(new MyHubFilter())); |

Так, в этом примере регистрируется два фильтра одного типа. И при обращении к хабу будет создавать два разных объекта этого фильтра, два singleton-объекта. Но при новом обращении к хабу будут использоваться ранее созданные объекты фильтров.

## Клиент на .NET MAUI

.NET MAUI позволяет создавать клиентские приложения, которые могут взаимодействовать с хабом SignalR на стороне сервера. .NET MAUI применяет тот же клиент, что применяется вцелом в .NET для взаимодействия с хабом SignalR. Рассмотрим, как сделать подобное приложение.

### Создание сервера

Сначала определим код сервера, с которым будет взаимодействовать клиент на .NET MAUI. Для этого создадим проект ASP.NET Core по типу Empty.

Определим в проекте следующий простейший класс хаба:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.SignalR;    namespace SignalRApp  {      public class ChatHub : Hub      {          public async Task Send(string username, string message)          {              await this.Clients.All.SendAsync("Receive", username, message);          }      }  } |

В методе Send хаб будет принимать имя пользователя и его сообщение и транслировать его на функцию Receive всех подключенных клиентов.

В файле **Program.cs** определим следующий код:

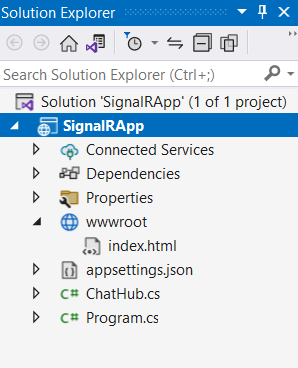
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using SignalRApp;   // пространство имен класса ChatHub    var builder = WebApplication.CreateBuilder(args);  builder.WebHost.UseUrls("[http://0.0.0.0:8080](http://0.0.0.0:8080/)");    builder.Services.AddSignalR();    var app = builder.Build();      app.UseDefaultFiles();  app.UseStaticFiles();    app.MapHub<ChatHub>("/chat");  app.Run(); |

Здесь следует отметить, что для того, чтобы наше приложение было доступно извне в локальной сети, применяется вызов builder.WebHost.UseUrls("http://0.0.0.0:8080");, который устанавливает для приложения адрес "http://0.0.0.0:8080". Это значит, что мы можем обратиться к приложения, используя адрес локального компьютера в сети и порт 8080. Обратите внимание, что порт должен быть не занят, иначе следует выбрать другой порт.

И также для теста определим в папке **wwwroot** простейшую веб-страницу **index.html**:

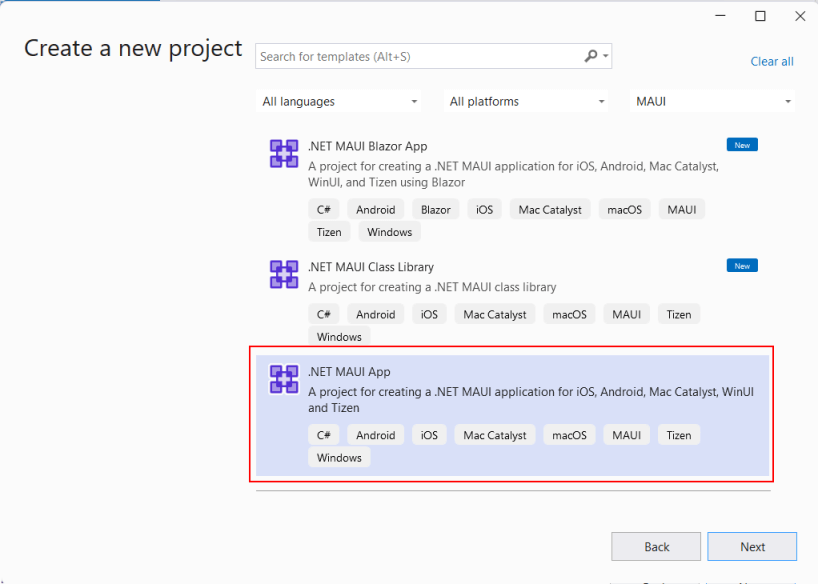
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div>          Введите логин:<br />          <input id="userName" type="text" /><br /><br />          Введите сообщение:<br />          <input type="text" id="message" /><br /><br />          <input type="button" id="sendBtn" value="Отправить" disabled="disabled" />      </div>      <div id="chatroom"></div>      <script src="<https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.js>"></script>      <script>          const hubConnection = new signalR.HubConnectionBuilder()              .withUrl("/chat")              .build();            document.getElementById("sendBtn").addEventListener("click", function () {              const userName = document.getElementById("userName").value;   // получаем введенное имя              const message = document.getElementById("message").value;                hubConnection.invoke("Send", userName, message) // отправка данных серверу                  .catch(function (err) {                      return console.error(err.toString());                  });          });          // получение данных с сервера          hubConnection.on("Receive", function (userName, message) {                // создаем элемент <b> для имени пользователя              const userNameElem = document.createElement("b");              userNameElem.textContent = `${userName}: `;                // создает элемент <p> для сообщения пользователя              const elem = document.createElement("p");              elem.appendChild(userNameElem);              elem.appendChild(document.createTextNode(message));                // добавляем новый элемент в самое начало              // для этого сначала получаем первый элемент              const firstElem = document.getElementById("chatroom").firstChild;              document.getElementById("chatroom").insertBefore(elem, firstElem);          });            hubConnection.start()              .then(function () {                  document.getElementById("sendBtn").disabled = false;              })              .catch(function (err) {                  return console.error(err.toString());              });      </script>  </body>  </html> |

Общий проект сервера:

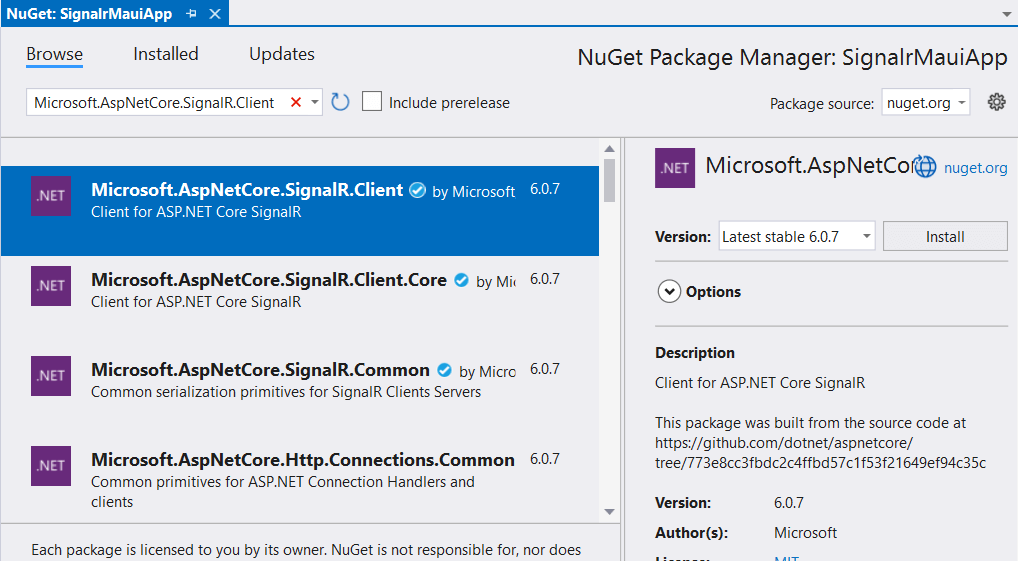


### Создание клиента .NET MAUI

Теперь создадим клиентское приложение на .NET MAUI для взаимодействия с вышеопределенным хабом SignalR. Для этого создадим новый проект по типу **.NET MAUI App**, который путь будет называться **SignalrMauiApp**.



После создания проекта в первую очередь добавим в него Nuget-пакет **Microsoft.AspNetCore.SignalR.Client**.



Сначала добавим в проект MAUI класс **ChatViewModel**, который будет выполнять роль модели представления и через который будет идти взаимодействие с сервером:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129 | using Microsoft.AspNetCore.SignalR.Client;  using System.Collections.ObjectModel;  using System.ComponentModel;    namespace SignalrMauiApp  {      public class ChatViewModel : INotifyPropertyChanged      {          HubConnection hubConnection;            public string UserName { get; set; }          public string Message { get; set; }          // список всех полученных сообщений          public ObservableCollection<MessageData> Messages { get; } = new();            // идет ли отправка сообщений          bool isBusy;          public bool IsBusy          {              get => isBusy;              set              {                  if (isBusy != value)                  {                      isBusy = value;                      OnPropertyChanged("IsBusy");                  }              }          }          // осуществлено ли подключение          bool isConnected;          public bool IsConnected          {              get => isConnected;              set              {                  if (isConnected != value)                  {                      isConnected = value;                      OnPropertyChanged("IsConnected");                  }              }          }          // команда отправки сообщений          public Command SendMessageCommand { get; }            public ChatViewModel()          {              // создание подключения              hubConnection = new HubConnectionBuilder()                  .WithUrl("<http://192.168.0.116:8080/chat>")                  .Build();                IsConnected = false;    // по умолчанию не подключены              IsBusy = false;         // отправка сообщения не идет                SendMessageCommand = new Command(async () => await SendMessage(), () => IsConnected);                hubConnection.Closed += async (error) =>              {                  SendLocalMessage(string.Empty, "Подключение закрыто...");                  IsConnected = false;                  await Task.Delay(5000);                  await Connect();              };                hubConnection.On<string, string>("Receive", (user, message) =>              {                  SendLocalMessage(user, message);              });          }          // подключение к чату          public async Task Connect()          {              if (IsConnected)                  return;              try              {                  await hubConnection.StartAsync();                  SendLocalMessage(string.Empty, "Вы вошли в чат...");                    IsConnected = true;              }              catch (Exception ex)              {                  SendLocalMessage(string.Empty, $"Ошибка подключения: {ex.Message}");              }          }            // Отключение от чата          public async Task Disconnect()          {              if (!IsConnected) return;                await hubConnection.StopAsync();              IsConnected = false;              SendLocalMessage(string.Empty, "Вы покинули чат...");          }            // Отправка сообщения          async Task SendMessage()          {              try              {                  IsBusy = true;                  await hubConnection.InvokeAsync("Send", UserName, Message);              }              catch (Exception ex)              {                  SendLocalMessage(string.Empty, $"Ошибка отправки: {ex.Message}");              }                IsBusy = false;          }          // Добавление сообщения          private void SendLocalMessage(string user, string message)          {              Messages.Insert(0, new MessageData(user, message));          }          public event PropertyChangedEventHandler PropertyChanged;          public void OnPropertyChanged(string prop = "")          {              if (PropertyChanged != null)                  PropertyChanged(this, new PropertyChangedEventArgs(prop));          }      }        public record MessageData(string User, string Message);  } |

Раберем данный код. Прежде всего в конце этого кода определен класс MessageData, который представляет одно сообщение. Он имеет два свойства: Message (для хранения собственно текста сообщения) и User (для хранения отправителя сообщения).

Для взаимодействия с хабом в классе ChatViewModel нам потребует класс **HubConnection**, который предоставляет нам функционал для подключения к хабу и отправки сообщений.

Свойства UserName и Message представляют соответственно имя пользователя и текст сообщения, которые будут отправляться на сервер. Свойство Messages представляет объект ObservableCollection<MessageData> - полученные с сервера сообщения.

Чтобы извещать пользователя о процессе отправки, определено свойство IsBusy - если оно равно true, то приложение находится в процессе оправки сообщения.

Свойство IsConnected указывает, подключено ли приложение к хабу.

Непосредственно для отправки сообщений определена команда SendMessageCommand.

В конструкторе ChatViewModel с помощью класса HubConnectionBuilder создается объект HubConnection. Для его инициализации через метод WithUrl() передается адрес хаба:

|  |  |
| --- | --- |
| 1  2  3 | hubConnection = new HubConnectionBuilder()      .WithUrl("<http://192.168.0.116:8080/chat>")      .Build(); |

В каждом конкретном случае адрес будет **отличаться**. В моем случае адрес компьютера, на котором запущено приложение asp.net - http://192.168.0.116. Поскольку это приложение использует порт 8080, а хаб ChatHub доступен по адресу "/chat", то полный адрес выглядит так: "http://192.168.0.116:8080/chat".

(Чтобы узнать адрес компьютера в локальной сети на Windows в командной строке можно выполнить команду **ipconfig**)

Затем определяется команда отправки сообщений:

|  |  |
| --- | --- |
| 1 | SendMessageCommand = new Command(async () => await SendMessage(), () => IsConnected); |

При выполнении команды будет вызываться метод SendMessage. Кроме того, команда будет доступна, если свойство IsConnected равно true, то есть если мы подключены к хабу.

Получив объект HubConnection, мы можем выполнить его настройку. Так, далее устанавливается обрабатчик события завершения подключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | hubConnection.Closed += async (error) =>  {      SendLocalMessage(String.Empty, "Подключение закрыто...");      IsConnected = false;      await Task.Delay(5000);      await Connect();  }; |

При закрытии подключения, которое может происходить по самым разным причинам, коллекцию Messages добавляется диагностическое сообщение для пользователя (поэтому вместо имени пользователя используется пустая строка string.Empty) и затем через 5 секунд мы повторно пытаемся подключиться к хабу.

Кроме того, нам надо настроить прием сообщений. Для этого применяется метод On:

|  |  |
| --- | --- |
| 1  2  3  4 | hubConnection.On<string, string>("Receive", (user, message) =>  {      SendLocalMessage(user, message);  }); |

В классе хаба мы транслируем всем подключенным клиентам на функцию Receive две строки: Clients.All.SendAsync("Receive", username, message);. Поэтому в данном случае метод On() типизирован двумя объектами string - для получения имени пользователя и его сообщения. Первый парамет метода указавает название функции - Receive, а второй параметр представляет лямбда-выражение, в котором мы получаем от сервера данные.

В методе Connect() осуществляется подключение к хабу. Для этого применяется вызов hubConnection.StartAsync(). После его успешного выполнения мы можем взаимодействовать с сервером.

В методе Disconnect() происходит отключение от сервера. Для этого применяется вызов hubConnection.StopAsync()

Метод SendMessage() предназначен для отправки сообщений хабу. Это осуществляется посредством вызова hubConnection.InvokeAsync("Send", UserName, Message); - на хабе вызывается метод Send, которому передаются значения UserName и Message.

Теперь используем этот класс. Для этого определим к следующий интерфейс на странице **MainPage.xaml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | <?xml version="1.0" encoding="utf-8" ?>  <ContentPage xmlns="<http://schemas.microsoft.com/dotnet/2021/maui>"               xmlns:x="<http://schemas.microsoft.com/winfx/2009/xaml>"               x:Class="SignalrMauiApp.MainPage">        <StackLayout>          <ActivityIndicator IsRunning="{Binding IsBusy}" IsVisible="{Binding IsBusy}"                             HorizontalOptions="CenterAndExpand" />          <StackLayout Padding="5">              <Label FontSize="18" Text="Логин" />              <Entry x:Name="userNameBox"  Text="{Binding UserName}" HorizontalOptions="FillAndExpand"/>              <Label FontSize="Small" Text="Сообщение" />              <Entry HorizontalOptions="FillAndExpand" Text="{Binding Message}"/>              <Button Text="Отправить" IsEnabled="{Binding IsConnected}" Command="{Binding SendMessageCommand}"/>          </StackLayout>          <ListView ItemsSource="{Binding Messages}">              <ListView.ItemTemplate>                  <DataTemplate>                      <ViewCell>                          <ViewCell.View>                              <StackLayout Orientation="Horizontal">                                  <Label Text="{Binding User}" FontAttributes="Bold" />                                  <Label Text="{Binding Message}" Margin="10, 0, 0, 0" />                              </StackLayout>                          </ViewCell.View>                      </ViewCell>                  </DataTemplate>              </ListView.ItemTemplate>          </ListView>      </StackLayout>  </ContentPage> |

Элемент ActivityIndicator извещает пользователя об процессе отправки сообщений. Для ввода данных определены два текстовых поля. И по нажатию на кнопку вызывается команда SendMessageCommand, которая оправляет введенные данные.

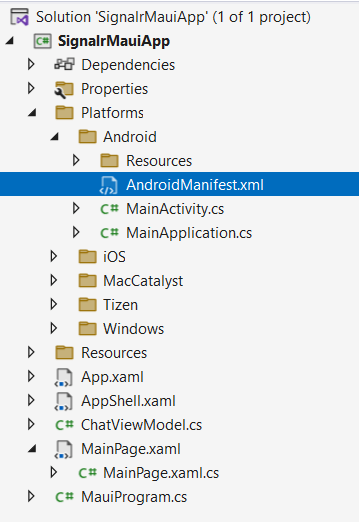
Для отображения сообщений определен элемент ListView.

В файле **MainPage.xaml.cs** определим привязку ChatViewModel к странице:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | namespace SignalrMauiApp;    public partial class MainPage : ContentPage  {      ChatViewModel viewModel;      public MainPage()      {          InitializeComponent();          viewModel = new ChatViewModel();          BindingContext = viewModel;      }        protected override async void OnAppearing()      {          base.OnAppearing();          await viewModel.Connect();      }        protected override async void OnDisappearing()      {          base.OnDisappearing();          await viewModel.Disconnect();      }  } |

В методе OnAppearing, то есть когда начинается отображение страницы, осуществляется подключение к хабу. В методе OnDisappearing, когда пользователь покидает страницу или приложение переходит в фоновый режим, то выполняется отключение от хаба.

Важный момент, который надо учитывать при тестировании: использование протокола http. В реальной применении приложении asp.net core с хабом SignalR рекомендуется запускать по протоколу https. Однако при тестирование гораздо проще использовать протокол http, что позволяет избежать возни с сертификатами. И в примере выше с хабом SignalR приложение ASP.NET как раз запускается по протоколу http. В приложении для Windows проблем с использованием http не возникнет. А вот приложение для Android по умолчанию использует только протокол https. Чтобы в подключить для Android протокол http, необходимо перейти к проекте MAUI к папке **Platforms/Android** к файлу **AndroidManifwest.xml**:



По умолчанию он выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <?xml version="1.0" encoding="utf-8"?>  <manifest xmlns:android="<http://schemas.android.com/apk/res/android>">      <application android:allowBackup="true" android:icon="@mipmap/appicon"                   android:roundIcon="@mipmap/appicon\_round" android:supportsRtl="true"      ></application>      <uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE" />      <uses-permission android:name="android.permission.INTERNET" />  </manifest> |

Для подключения поддержки http необходимо в элемент **application** добавить атрибут:

|  |  |
| --- | --- |
| 1 | android:usesCleartextTraffic="true" |

То есть в итоге код файла будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?xml version="1.0" encoding="utf-8"?>  <manifest xmlns:android="<http://schemas.android.com/apk/res/android>">      <application android:allowBackup="true" android:icon="@mipmap/appicon"                   android:roundIcon="@mipmap/appicon\_round" android:supportsRtl="true"                   android:usesCleartextTraffic="true"      ></application>      <uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE" />      <uses-permission android:name="android.permission.INTERNET" />  </manifest> |

Запустим сначала приложение ASP.NET Core, а затем приложение на .NET MAUI. И все пользователи, которые подключены к хабу, вне зависимости через javascript-клиент или maui-клиент, смогут взаимодействовать с хабом.

