FHV
Vorarlberg University
of Applied Sciences

Application Integration and Security

Valmir Bekiri

Philipp Scambor

Bidirectional communication

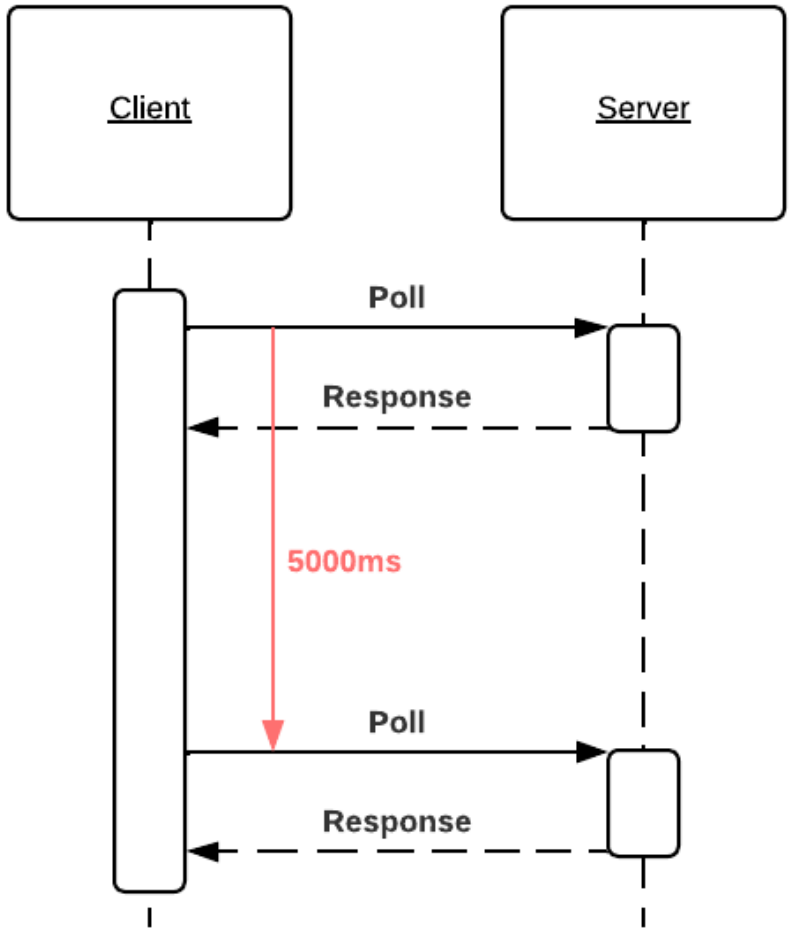
Bidirectional communication

- HTTP is a stateless and unidirectional communication protocol for communication on the Web which fulfils the requirements of a Client-Server Architecture, where the client requests data from the server and the server responds the corresponding data = **Unidirectional request-response communication**!
- However, in certain applications, such as real-time messaging, gaming, voting and auction platform, dashboard, monitoring or collaborative apps, it is important to have a two-way communication channel between the client and server.
- **Bidirectional communication** on the web refers to the ability of a web application
  - to **send and receive data in real-time** between the client (browser) and the server.
  - client and server can initiate communication and exchange data without waiting for the other to start.
- Ways to solve the need for bidirectional communication:
  Polling (Short and Long), **Server-Sent Events**, **WebSockets** (see on next slides)

Bidirectional communication

Short Polling

- Polling is the **simplest solution** to display automatically updated data **but has drawbacks**.
- In **Short Polling**, the **client** uses a timer to send asynchronous requests (using fetch) at regular intervals, which the server responds with the appropriate data.



Advantages

- Simple solution: does only require periodical calls on the client
- Server can be completely unaware

Disadvantages

- Delay due to polling intervall
- Many unnecessary calls: overhead on network, client and server side
- Not really scaleable

Bidirectional communication

Short Polling

Example implementation on a JavaScript client to the right:

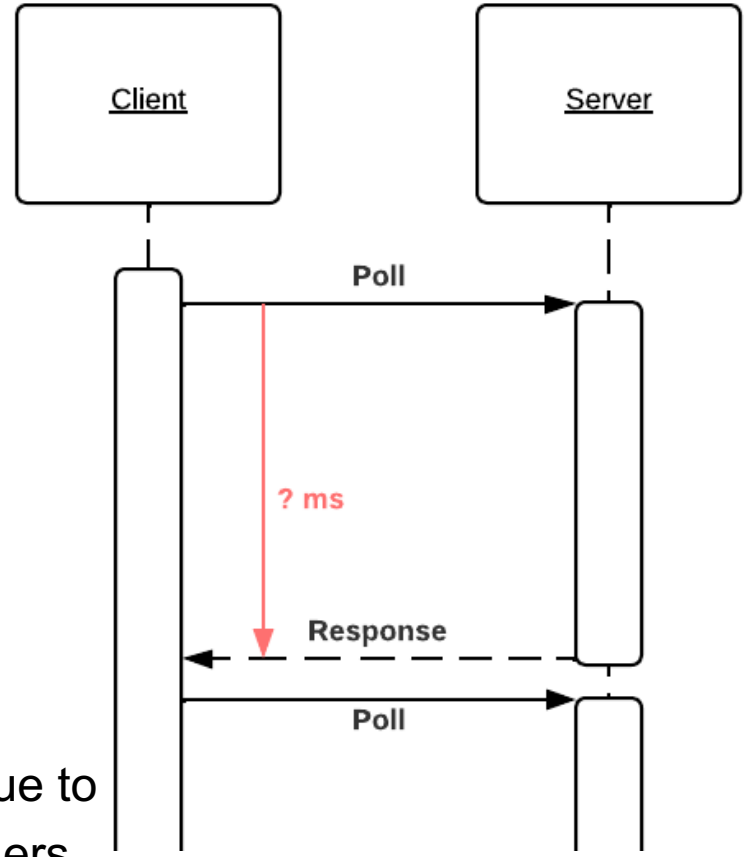
- Fetches an api in 5000ms intervals
- updates the DOM based on the received data

```
const randomNumber = document.getElementById('random-number');
const creationDate = document.getElementById('creation-date');

function updateRandomNumber() {
  fetch('/api/Random')
    .then((response) => response.json())
    .then((data) => {
      randomNumber.innerHTML = data.random;
      creationDate.innerHTML = data.created;
    });
}

// call function manually at the beginning, because
setInterval will call it after 5000ms for the first time
updateRandomNumber();

// calls the updateRandomNumber function every 5000 ms
setInterval(updateRandomNumber, 5000);
```

Bidirectional communication

Long Polling

- In **Long Polling** the client sends a request to the server.
  - The server keeps the connection open until new data is available.
  - When the server has new data, it sends a response to the client and the connection is closed.
  - Immediately after receiving the response, the client sends another request to open a new connection and the process starts again.



Advantages

- Data is delivered instantly
- Server does not waste resources for requests delivering no new data
- Based on standard HTTP (can be a requirement)

Disadvantages

- Each request comes with an overhead due to the nature of HTTP (authentication, headers, etc.) => more data than necessary is transmitted
- Depending on the implementation long polling can be resource intensive on the server (CPU, memory)

Bidirectional communication

Long Polling

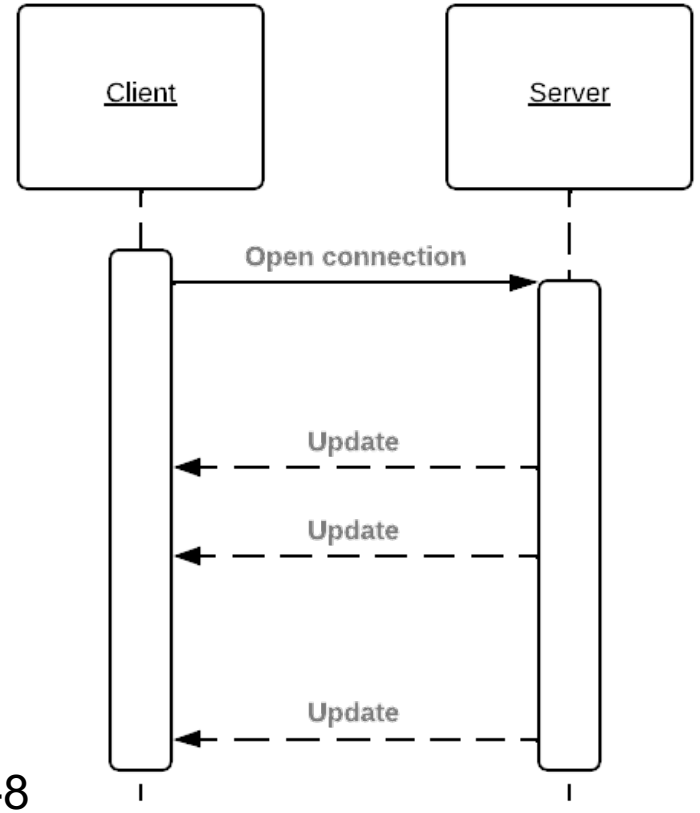
Example implementation on a JavaScript client to the right:

- Fetches an api and updates the DOM based on the received data
- Api response is sent if there is an update (server has the control)
- Client implementation is unaware of the update interval => could be immediate, could be a few seconds or longer

```
1  const randomNumber = document.getElementById("random-number");
2  const creationDate = document.getElementById("creation-date");
3
4  function subscribeRandomNumber() {
5    fetch("/api/Random")
6      .then((response) => response.json())
7      .then((data) => {
8        randomNumber.innerHTML = data.random;
9        creationDate.innerHTML = data.created;
10
11       // after response has been handled a new request is
12       // sent and the server will delay the response
13       subscribeRandomNumber(); // until new data is available
14     });
15 }
16 subscribeRandomNumber();
```

Bidirectional communication

Server-sent events (SSE)



- **Server-sent events (SSE)** have been introduced to address the drawbacks of the polling approach. It allows a client to open an HTTP connection to receive push notifications in form of DOM events.
- Main difference to long polling is that the **connection stays open** after an update has been sent. This is possible due to a streaming mechanism that can send data continuously.

Advantages

- Solves the problem to send data from the server to the client
- Client can still send its own request if it want to talk to the server
- Efficient and relatively easy to implement

Disadvantages

- Limited to transport UTF-8 messages; binary data is not supported
- Limited concurrent connections on browsers
- One way (server to client)

Bidirectional communication

Server-sent events (SSE)

Example implementation on a JavaScript client to the right:

- Uses the EventSource interface to open a connection to an HTTP server
- Server sends events in **text/event-stream** format
- If there is an event field in the incoming message, the triggered event is the same as the event field value (“random” in the code to the right).
- If no event field is present, then a generic “message” event is fired.

```
event: random
data: {"random":2,"created":"..."}
```

```
1  const randomNumber = document.getElementById("random-number");
2  const creationDate = document.getElementById("creation-date");
3
4  const eventSource = new EventSource("/Random");
5
6  eventSource.addEventListener("random", (event) => {
7    const data = JSON.parse(event.data);
8
9    randomNumber.innerHTML = data.random;
10   creationDate.innerHTML = data.created;
11 });
```

https://developer.mozilla.org/en-US/docs/Web/API/EventSource

Bidirectional communication

Websockets

- **WebSockets** were the **real breakthrough regarding bidirectional communication**.
- WebSockets changed the uni-directional way of polling and server-sent events by introducing a **completely new protocol**. The goal was to build something that is almost as powerful as TCP/IP sockets but targeted for the web.
- WebSockets are a **built on top of TCP/IP**
  - adds a few abstractions regarding the way the web works.
  - establishing a connection with it should make use of a URL

Bidirectional communication

Websockets

- HTTP is used to establish a connection between client and server but instead of closing it right after the response, client and server agree on “upgrading” the connection.
  - The client tells the server with the **Connection header** that the current connection should be upgraded. The **Upgrade header** tells the server to which protocol the connection should be upgraded.
  - If the server agrees, it returns the **101 Switching Protocols status code** along with the protocol to which it was upgraded
  - After upgrade, the connection is persistent and can be used for communication in both directions. As the data being exchanged are strings, the client and server must agree on a protocol.

Request for websocket connection upgrade

```
GET /index.html HTTP/1.1
Host: www.example.com
Connection: Upgrade
Upgrade: websocket
```

Response for the upgrade request

```
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
```

Bidirectional communication

Websockets

- Due to being a low-level protocol, a single WebSocket connection can handle a high bandwidth on a single connection. Headers are not sent every time we need to get more information from the server.
- Websockets were introduced in 2011 and are supported across all major browsers

Advantages

- Resource efficiency
- Low latency
- Bidirectional full-duplex communication
- Flexible data format (UTF-8 or binary)

Disadvantages

- Firewall blocking: some block WebSockets
- No built in support for reconnection

https://ably.com/blog/websockets-vs-sse

Bidirectional communication

Websockets

The HTML standard provides an API

for Websockets (WebSockets). Main tasks are:

- open the connection and
- send and receive data in the form of
- plain strings / JSON-Data

WebSocket makes use of DOM events too (e.g. addEventListener for)

- **open** event: connection is established
- **message** event
- **close** event: connection terminated (can be used to reestablish the connection)

```
1  const randomNumber = document.getElementById("random-number");
2  const creationDate = document.getElementById("creation-date");
3  const updateButton = document.getElementById("update-button");
4
5  // create WebSocket connection - absolute URL is necessary
6  const webSocket = new WebSocket(`wss://${location.host}/WebSocket`);
7
8  // connection opened
9  webSocket.addEventListener("open", function (event) {
10   updateButton.addEventListener("click", function () {
11     webSocket.send(JSON.stringify({ type: "update" }));
12   });
13 });
14
15 // listen for messages
16 webSocket.addEventListener("message", function (event) {
17   const data = JSON.parse(event.data);
18   randomNumber.innerHTML = data.random;
19   creationDate.innerHTML = data.created;
20 });
```

https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

Bidirectional communication

Authentication

- **Short and long polling** are based upon the standard HTTP protocol so they can implement **authentication for every poll** (HTTP headers, cookies, etc.)
- **Server-sent events** and **Websockets** use a single **initial HTTP connection** that stays open (or is switched for Websockets). So again one can use standard HTTP authentication for the connection request.
- For some reason the WebSockets browser API does not offer a way to set custom HTTP headers. Because of this authentication for Websockets from browsers is limited to:
  - Cookie based authentication (works automatically as the cookie is sent with the upgrade request)
  - Access token via query parameter with the WebSocket connection upgrade request
    NOTE: query parameters might get logged on the server!

Bidirectional communication

SignalR

- WebSockets allow to have a connection enabling bidirectional communication, but only offer a very low-level API. Nevertheless, WebSockets misses certain features like reconnecting when an error happens or support for channels to address a group of clients.
- **SignalR** solves the above-mentioned issues. Main advantages are
  - **Automatic connection management** reconnects when the connection was dropped
  - **Supports groups** allows to send messages to all or just a group of clients
  - **Scaling** allows to scale to support increasing traffic
- SignalR is not just Websockets - it is a higher level abstraction that needs to be implemented on both the client and the server (see later)

Bidirectional communication

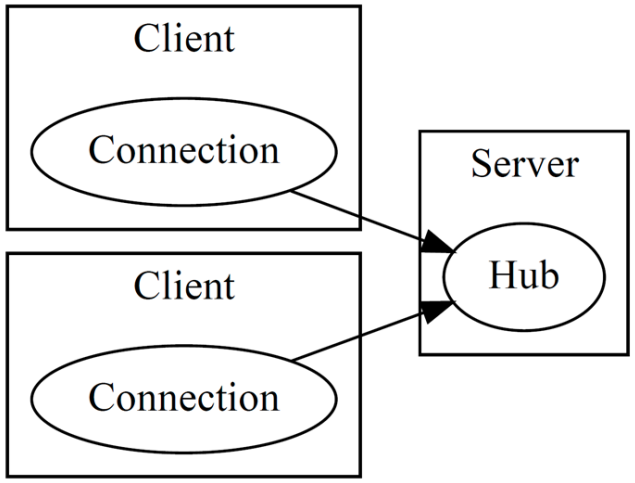
SignalR

- **Automatic connection management** also includes **choosing** the optimal transport method for each connection, i.e. the best technology is chosen based on the capabilities of client and server.
- **SignalR supports WebSockets, server-sent events and long polling**. The selection of the best technology happens automatically, which means that the exact same code can be used to support all the previously mentioned technologies, and SignalR does all the work behind the scenes. Order of graceful fallback:
  - Websockets
  - Server-Sent Events
  - Long Polling

Bidirectional communication

SignalR

- SignalR uses Hubs to allow server and client to call methods
  - Hub implementation inherits from the framework Hub class, which provides access to connected clients using the **Clients** property
  - There are different ways to address clients: **All** property (= easiest way), which addresses all clients currently connected to the hub. **Others** property to address all other clients.
  - **Clients** can call any **publicly defined methods** (e.g. GenerateNumber) on the registered hub.
  - Data (e.g., a randomly generated number) can be sent to connected clients using the SendAsync method, which basically calls a method defined on the client (e.g, ReceiveNumber).



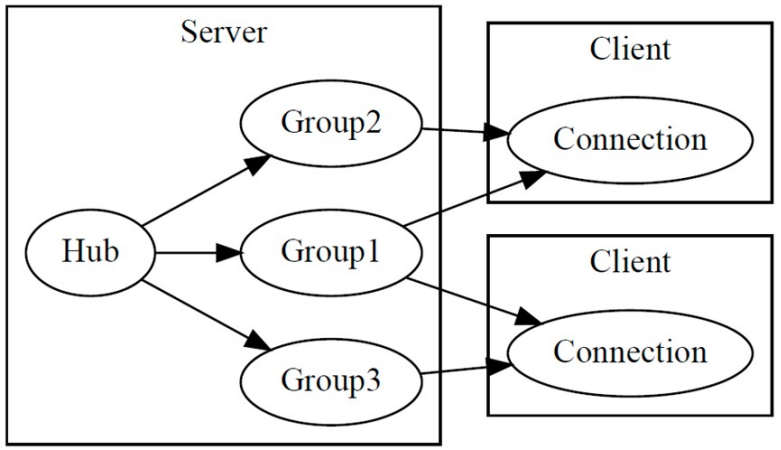

```
1  public class RandomHub : Hub
2  {
3    private Random _random { get; }
4
5    public RandomHub(Random random)
6    {
7      _random = random;
8    }
9
10   public async Task GenerateNumber()
11   {
12     await Clients.All.SendAsync(
13       "ReceiveNumber",
14       _random.Next(0, 100)
15     );
16   }
17 }
```

https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction

Bidirectional communication

SignalR - Server

- **Groups** allow to address only some clients instead all of them. A common feature even in simple applications (e.g. chat)
  - **Groups** are **named** and can **contain multiple connections**.
  - One connection can also be listed in multiple groups. Connections are added to a specific group with the `Groups.AddToGroupAsync` function. Connections are added to a group using the **connection ID**, that can be determined via the `Context` property of the Hub.
  - `OnDisconnectedAsync` method is automatically called when a client disconnects (note: receives an exception parameter in case the disconnection was caused by an error and not by intention). Use the `RemoveFromGroupAsync` method of the Groups property of the hub to remove the connection from a group.
  - Messages can be sent to all connections belonging to a group




```
1  public override async Task
   OnConnectedAsync()
2  {
3    await Groups.AddToGroupAsync(
4      Context.ConnectionId,
5      "SignalR Users"
6    );
7    await base.OnConnectedAsync();
8  }
9
10 public override async Task
   OnDisconnectedAsync(Exception ex)
11 {
12   await Groups.RemoveFromGroupAsync(
13     Context.ConnectionId,
14     "SignalR Users"
15   );
16
17   await base.OnDisconnectedAsync(ex);
18 }
```

https://learn.microsoft.com/en-us/aspnet/core/signalr/groups

ASP.NET Core SignalR

ASP.NET Core SignalR

Basic Setup

- The SignalR library is already included in every .Net Core Web Template. So, it can be included in any .Net Core Web application or service without installing any other library.
- Steps to develop an application using SignalR from scratch:
  - **Create an ASP.Net Core Web API** (minimal template)
  - Add **RandomHub** class (see previous slides)
  - and **wwwroot** folder including client side code (see following slides)

```
1  var builder = WebApplication.CreateBuilder(args); /* Program.cs */
2
3  // Add services to the container.
4  builder.Services.AddSingleton<Random>(); // Random number generator service
5  builder.Services.AddSignalR(); // add SignalR functionality
6
7  var app = builder.Build();
8
9  // Configure the HTTP request pipeline.
10 app.UseHttpsRedirection();
11 app.UseStaticFiles(); // access to wwwroot folder
12 app.MapHub<RandomHub>("/RandomHub"); // map SignalR Hub
13 app.Run();
```

ASP.NET Core SignalR

Hub Improvement: strongly typed interface

- Using the `SendAsync` method and passing the name of the function as a string **may lead to errors**. **Extract the methods** that can be called (e.g. ReceiveNumber) into a separate interface and use the strongly typed Hub. The interface restricts the methods available to be called on the clients!
  - Add IRandomClient interface and inherit the RandomHub from strongly typed Hub.

```
1  public interface IRandomClient
2  {
3    Task ReceiveNumber(int number);
4  }
5
6  public class RandomHub : Hub<IRandomClient>
7  {
8    ...
9    public async Task GenerateNumber()
10   {
11     await Clients.All.ReceiveNumber(_random.Next(0, 100));
12   }
13 }
```

ASP.NET Core SignalR

Hub Improvement: using object parameters

- SignalR will check the number of parameters and throw an error if they do not match, it is best practice to use a single custom object parameters instead of multiple parameters to ensure backwards-compatibility. Missing object properties will be filled with null and at least do not cause any SignalR errors, although these null values then must be checked in the code (client and server side).
  - Add **Parameter** classes to **IRandomClient** and adapt **GenerateNumber** method using these classes

```
1  public interface IRandomClient /* IRandomClient.cs */
2  {
3    public class ReceiveNumberParameters
4    {
5      public int Number { get; set; }
6    }
7
8    Task ReceiveNumber(ReceiveNumberParameters parameters);
9  }
```

```
1  /* RandomHub.cs */
2  public async Task GenerateNumber()
3  {
4    await Clients.All.ReceiveNumber(
5      new ReceiveNumberParameters() {
6        Number = _random.Next(0, 100)
7      }
8    );
9  }
```

ASP.NET Core SignalR

Client side

- Requires npm package `@microsoft/signalr`

```
1  const updateButton = document.getElementById('update-button');
2  const randomNumber = document.getElementById('random-number');
3
4  const connection =
5    new signalR.HubConnectionBuilder().withUrl('/RandomHub').build();
6
7  // use strongly typed hub and parameter objects
8  connection.on('ReceiveNumber',
9    ({ number }) => { randomNumber.innerHTML = number; });
10
11 updateButton.addEventListener('click', () => {
12   connection.invoke('GenerateNumber');
13 });
14
15 connection.start();
```

ASP.NET Core SignalR

Send messages from outside the hub

- As described, every time when a method on the hub is called a new Hub instance is created. This makes it hard to send messages to clients from outside a hub. SignalR solves that issue by
  - introducing the interface called **IHubContext** that takes a generic type parameter describing the Hub class.
  - Use the **IHubContext** as a parameter in a constructor (e.g. Controller) to inject the hub. Note that the **IHubContext** interface only provides the public methods. So, **GenerateNumber** method is not call-/available!

```
1  public class RandomController : Controller
2  {
3    private IHubContext<RandomHub, IRandomClient> _randomHub;
4
5    public RandomController(IHubContext<RandomHub, IRandomClient> randomHub)
6    {
7      _randomHub = randomHub;
8    }
9
10   [Route("api/random")]
11   public void Get()
12   {
13     _randomHub.Clients.All.ReceiveNumber(
14       new IRandomClient.ReceiveNumberParameters() { Number = 50 }
15     );
16   }
17 }
```

That‘s it ☺
…for now