ASP.NET Core SignalR

Real-Time Applications, Live Communication, SignalR, Hubs

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Software
University

sli.do

#csharp-web

Table of Contents

1. Real-Time Applications
2. Web Communication Fundamentals
3. Remote Procedure Calls
4. ASP.NET Core SignalR

Candidates, Live Communication, Benefits

Real-Time Applications

Real-Time Applications

- **Real-Time Applications** are essentially **live communication** apps
  - They function within a time frame usually sensed as immediate
  - They have two-way communication between client and server
- **Real-Time Application** are used in many cases:
  - Gaming
  - Auctions, Betting
  - Stock quotes, Crypto
  - Email clients
  - Social media, Chats

Real-Time Applications

- **RTAs** use live communication to optimize functionality

    - This makes them more interactive and comfortable to use

- **Live communication** often requires additional web protocols

    - Like the **WebSocket** communication protocol for example

- **Live communication** requires some, unnatural for **HTTP**, processes

    - Server **sending** data **without** the Client **requesting** it

        - This feature is also known as **Server Push** – it is included in **HTTP/2**

    - Two-way data transfer over a single connection (**Full-Duplex**)

# Real-Time Applications

- **RTAs** solve many problems in the web applications world

  - **Fast Information delivery**

    - Imagine having to refresh a page to check actual live data

    - This would be catastrophic for crypto apps / betting apps

  - **Interactivity**, **Comfort** and **Usability**

    - Imagine having to refresh a chat to check if your friend sent a message

    - Your clients won't be satisfied with such functionality needs

Real-Time Applications

- **Live communication** is currently a very common thing

  - In fact, you are probably watching a **live-stream** at the moment

  - This is also considered live communication

- **Live communication** is probably used the most in the gaming industry

  - **Multiplayer Games** need a live connection with players

    - There are many in-game events that require this feature

    - It won't be very appropriate for players to restart the game just to realize they lost

 - Even the simplest online game of **Chess** uses **live communication**

Polling, Server-Sent Events, Remote Procedure Calls

Polling

- **Polling** is a technique by which the client requests data regularly
  - New data is **requested** at **frequent intervals**
  - Works with **HTTP requests** and **responses**
- There are generally two ways of **Polling**
  - **Short Polling**
    - An AJAX-based timer, that calls at **fixed delays**
  - **Long Polling**
    - The server **holds** the **request open** until **new data** is available
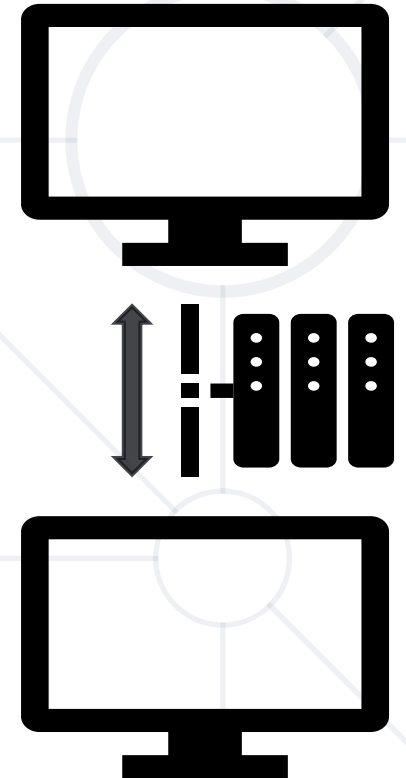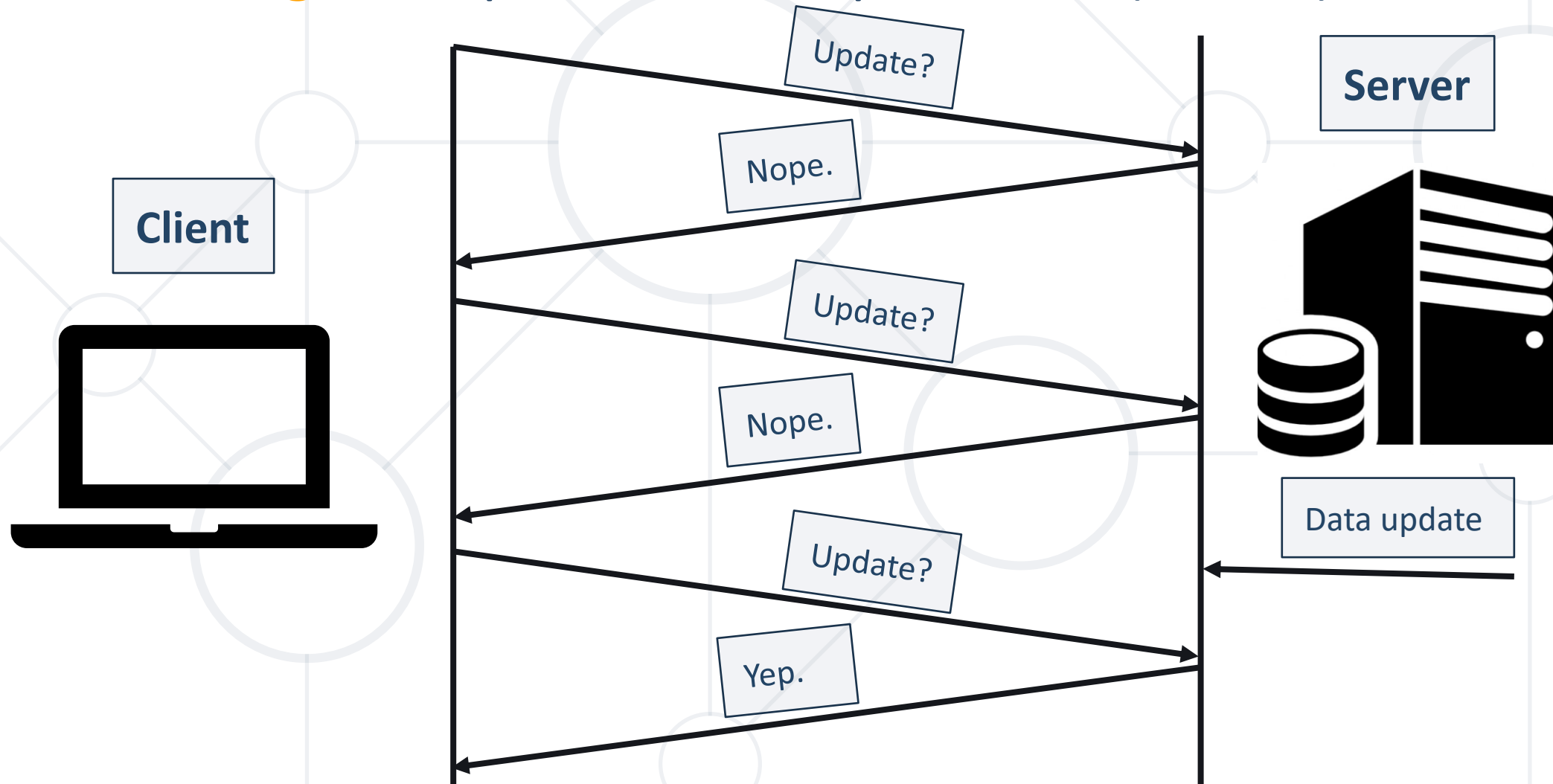
Polling

- Each of the 2 ways of **Polling** has its advantages and disadvantages

- **Short Polling** requires less server resource consumption

  - Practically useless if you need server event notification with **no delay**

  - Blasts your clients' **internet data** (if its limited)

- **Long Polling** notifies you about server events with **no delay**

  - Good for the bandwidth

  - More complex to develop and manage
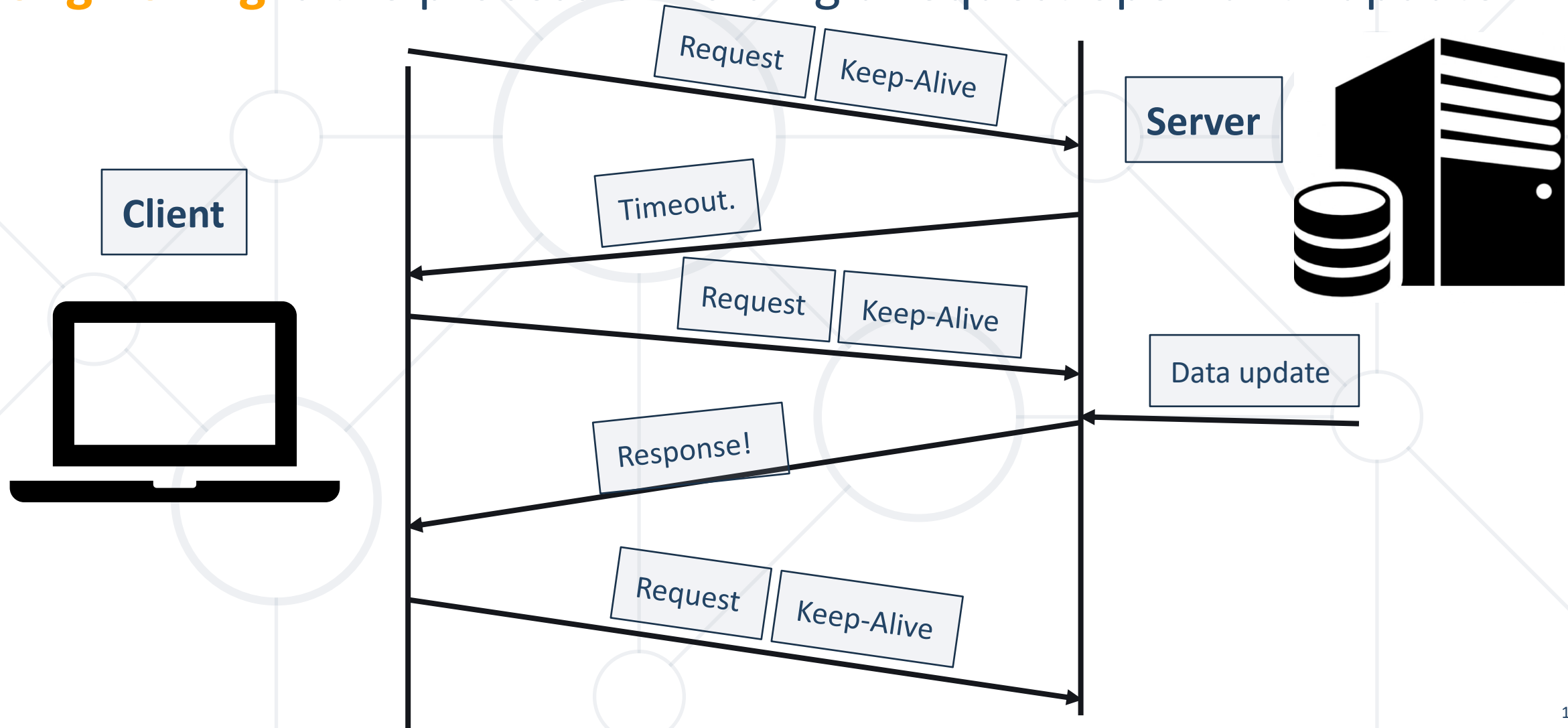
  - Requires more server resources
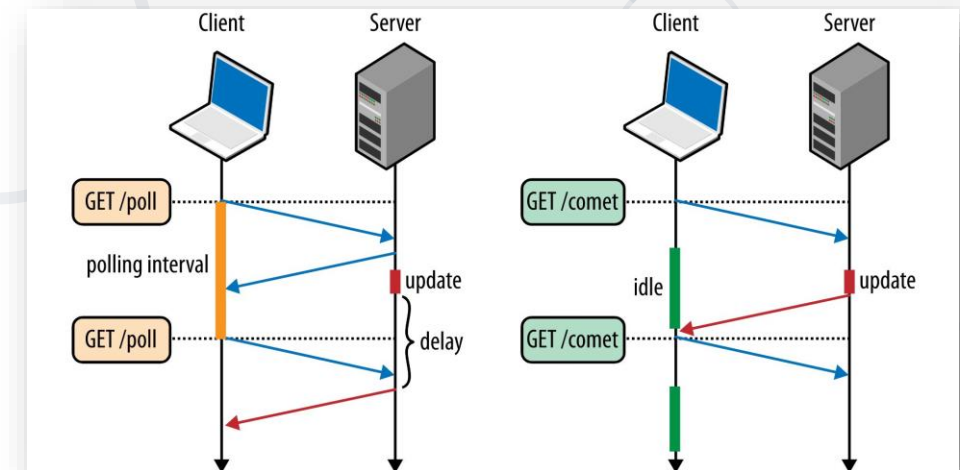
Short Polling – Process Explained

- **Short Polling** is the process of frequent calls (checks) for actual data
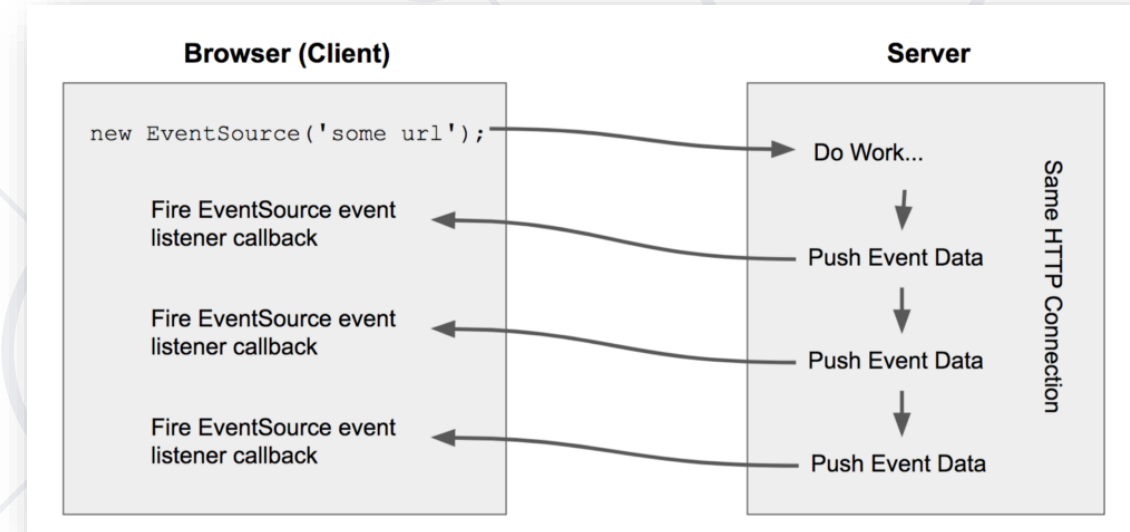

Update?

Nope.

Update?

Nope.

Update?

Yep.

Client

Server

Data update

- **Long Polling** is the process of holding a request open until update

Server-Sent Events

- **SSE** is a technology enabling a browser to receive automatic updates from a server via HTTP connection

  - Client doesn't have to make a request to check if updates are available

  - It is a **one-way** channel

- **SSE** requires an initial handshake

  - Server leaves the response open

    - Until there are no more events

    - Until the connection has been considered stale

    - Until the client explicitly closes the initial request

Browser (Client)

```
new EventSource('some url');
```

Fire EventSource event
listener callback

Fire EventSource event
listener callback

Fire EventSource event
listener callback

Server

Do Work...

Push Event Data

Push Event Data

Push Event Data

Same HTTP Connection

WebSocket

- **WebSocket** is a computer communication protocol
  - Provides **full-duplex** (**two-way**) communication channels
  - Channels are provided over a single **TCP connection**
- **WebSocket** is different from **HTTP**, although they are compatible
  - Works over a standard connection (**ws://**) and SSL (**wss://**)
  - Supports **HTTP** proxies and intermediaries
  - **WebSocket** enables **streams** of **messages** on top of **TCP**
  - Supported in most modern browsers, nowadays

WebSocket

- **WebSocket** introduces a whole **new way** of two-way communication

  - Two-way communication (**browser-server**) is certainly a convenience

- Before **WebSocket**, this was achieved in a rather non-standardized way

  - Using stopgap technologies such as **Comet**, for example

- **WebSocket** communication is initiated trough a casual handshake

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```
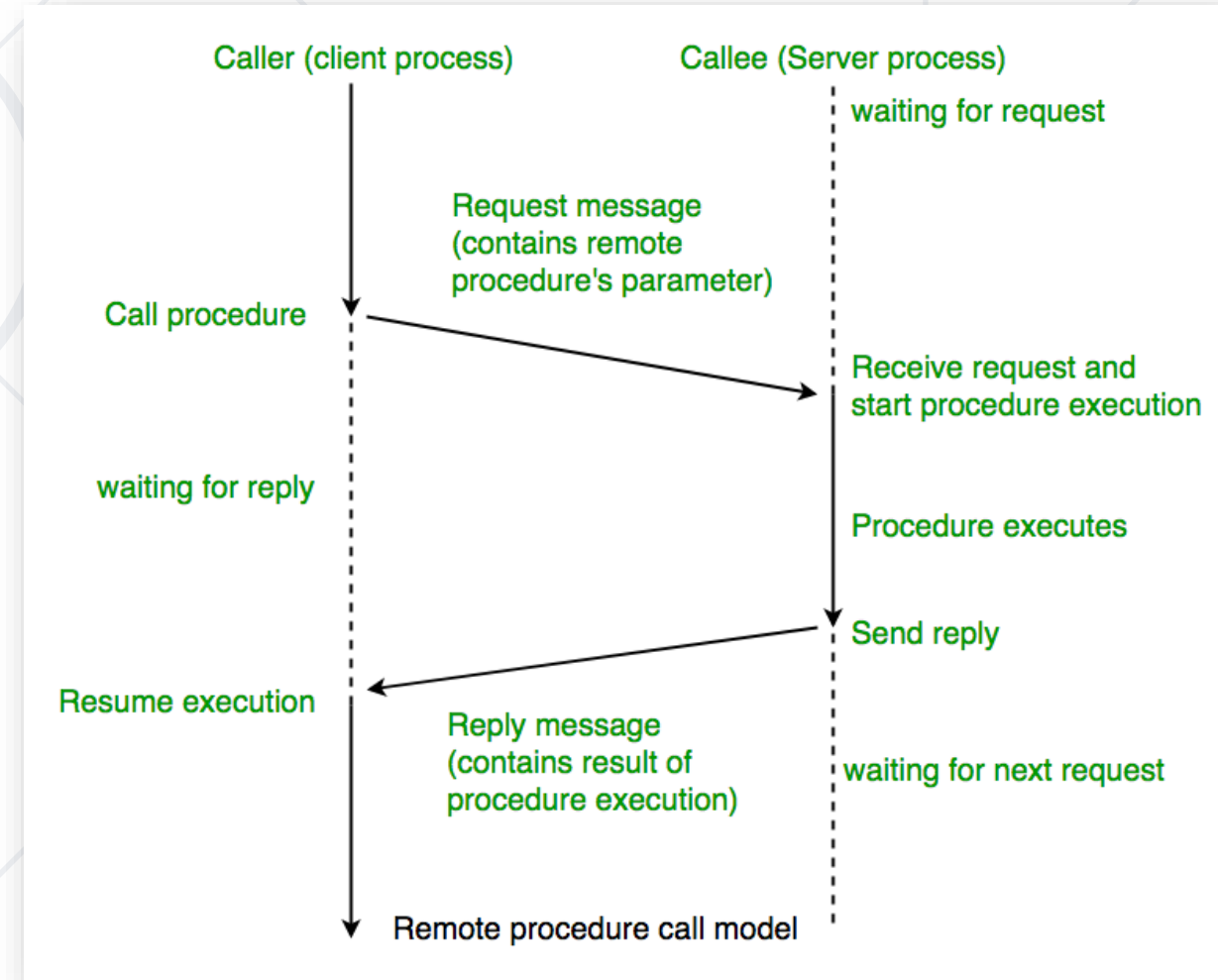Client Request

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```
Server Response

Remote Procedure Call

- **Remote Procedure Call** is an interprocess communication technique
    - A software causes a **procedure execution** in a different address space
    - Uses the **Client-Server** model
    - Used for **point-to-point** communication between apps
    - Sometimes called a **function** call / **subroutine** call
- **Remote Procedure Calls** are naturally **synchronous**
    - The **Sender** (Client) must wait for the **Executor** (Server)'s result
    - There are some ways of achieving **concurrency** though

Remote Procedure Call

- When making a **RPC**
  - Procedure **name** and **parameters** are transferred over the **network**
    - **Serialized**
  - Procedure is **executed** and produces **results**
  - **Results** are transferred **back**
    - **Serialized**
 - Execution process **resumes**

Caller (client process)   Callee (Server process)

waiting for request

Request message
(contains remote
procedure's parameter)

Call procedure

Receive request and
start procedure execution

waiting for reply

Procedure executes

Send reply

Resume execution

Reply message
(contains result of
procedure execution)

waiting for next request

Remote procedure call model

Adding Real-Time Functionality to Your Apps

ASP.NET Core SignalR

ASP.NET Core SignalR

- **ASP.NET Core SignalR** is a library that simplifies adding **RTC** in web apps
  - **RT** functionality enables **server content** to be **pushed** to clients **instantly**
- Good candidates for **SignalR** implementation include:
  - Apps that require **high-frequency updates** from the server
    - Gaming, Social Networks, Chat, Voting, Auction, Maps & GPS, etc.
  - **Dashboards** and **monitoring** apps – travel alerts, sales updates, etc.
  - **Collaborative** apps – Agile apps, Team Meeting apps, etc.
  - **Apps** that **require notifications** – Email, Chat, Social networks

ASP.NET Core SignalR

- **SignalR** provides **API** for creating **server-to-client RPCs**
    - The **RPCs** call **JavaScript** functions on **clients** and vice-versa
    - The **RPCs** are called from **server-side** .NET Code
- Some of the core features of **SignalR** include:
    - Handling **connection management** automatically
    - Sending messages to all connected clients simultaneously (**broadcast**)
    - Sending messages to a specific client or **group** of **clients**
    - Scaling to handle increasing traffic (**Azure SignalR Service**)

ASP.NET Core SignalR

- **SignalR** supports 3 techniques for handling RTC
  - WebSockets
  - Server-Sent Events
  - Long Polling
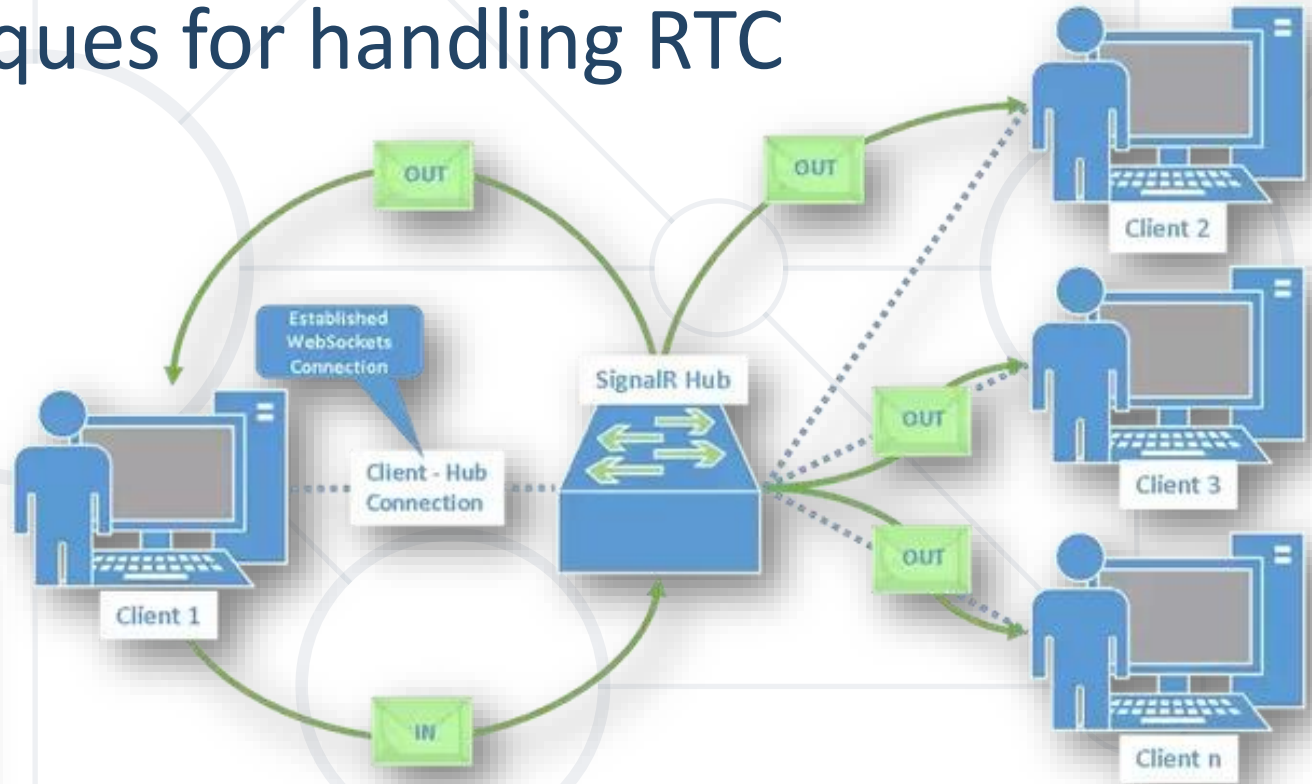
- **SingalR** automatically chooses the best transport method
  - Chosen within the **capabilities** of the **Server** and **Client**

ASP.NET Core SignalR Hubs

- **SignalR** uses **hubs** to establish communication client ↔ server

  - A **hub** is **high-level pipeline**

  - Allows a **Client** and **Server** to **call methods** on each other

- **Hubs** call **client-side** code by **sending messages**

  - These messages contain the **method name** and the **parameters**

  - Objects sent as **method parameters** are **deserialized**

  - The client tries to find a method, matching the **given name**

  - When the **Client** finds the method, it **passes** the **parameters** to it

ASP.NET Core SignalR Hubs
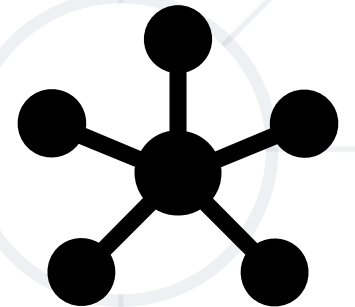
- You can pass **Strongly-typed** parameters to methods

    - This enables model binding on the server and vice-versa

    - These parameters are **deserialized**, using a **configured protocol**

- **SignalR** provides 2 built-in hub protocols

    - A **text protocol** based on **JSON**

    - A **binary protocol** based on **MessagePack**

    - **MessagePack** generally creates smaller messages

        - Compared to the JSON format

`{"compact":true}`

MessagePack

- **MessagePack** is an efficient, **binary** serialization format

  - It's like JSON, but fast and small

- **MessagePack** lets you exchange data among multiple languages

  - Small integers are encoded into a **single byte**

  - Typical short strings require only **one extra byte**

    - In addition to the strings themselves

Hub

- **SignalR Hubs API** provides a **Hub** class for applications to consume

  - The **Hub** class manages connections, groups, messaging

  - It also automatically manages the **Identity** system

  - This provides convenience and comfort when developing Hubs

- The **Hub** class contains several helpful members

  - The Hub **Context**, The Hub **Clients**, The Group **Manager**

- The **Hub** class contains methods for capturing Connection **events**

  - **OnConnectedAsync()** & **OnDisconnectedAsync()**

Context

- The **Hub Context** is represented by the **Context** property
  - It contains properties with information about the connection
- The most important properties provided by the **Context** are:
  - **ConnectionId** – Gets the unique Id for the connection
  - **UserIdentifier** – Gets the user identifier – **ClaimTypes.NameIdentifier**
  - **User** – Gets the **ClaimsPrincipal** associated with the current user
  - **Items** – Gets a **key-value-pair** collection, used for data sharing
- The **Context** also holds the methods **GetHttpContext** and **Abort**

Clients

- The **Hub Clients** is represented by the **Clients** property
  - It contains properties for communication between server and client
- The most important members provided by the **Clients** are:
  - **All** – Calls a method on **all** connected clients
  - **Caller** – Calls a method on the **client** that **invoked** the **hub method**
  - **Others** – Calls a method on all connected clients **except** the **caller client**
- The **Clients** also holds many other methods for **filtering clients**
  - These methods helps specify **particular clients**

GroupManager

- The **GroupManager** is represented by the **Groups** property
  - It contains properties for managing client clustering
  - You can add **Clients** to a specific **Group**
    - Using **AddToGroupAsync()**
  - You can remove **Clients** from a specific **Group**
    - Using **RemoveFromGroupAsync()**
- **Grouping Clients** helps you **broadcast** messages to specific auditories
  - In **real-time apps**, it is **rarely** needed to **broadcast** a message to **all clients**

SignalR Notes

- **SignalR** code is asynchronous to ensure maximum scalability
  - Use await when calling **async** methods that depend on the **Hub**
  - **Async** methods can fail if the hub method completes first
- **Hub** methods can return any type and receive any parameters
  - **SignalR** handles serialization and deserialization
- **Hubs** are transient
  - **Don't store state** in a property on the hub class
  - Every hub method call is executed on a **new hub instance**

Creating a Very Simplistic Chat Application

Including SignalR in ASP.NET Core

- Simple **Chat** application in **ASP.NET Core**

  - Create an empty project

  - In [**Solution Explorer**], right-click on the project, [**Add**] → [**Client-Side Library**]

Including SignalR in ASP.NET Core
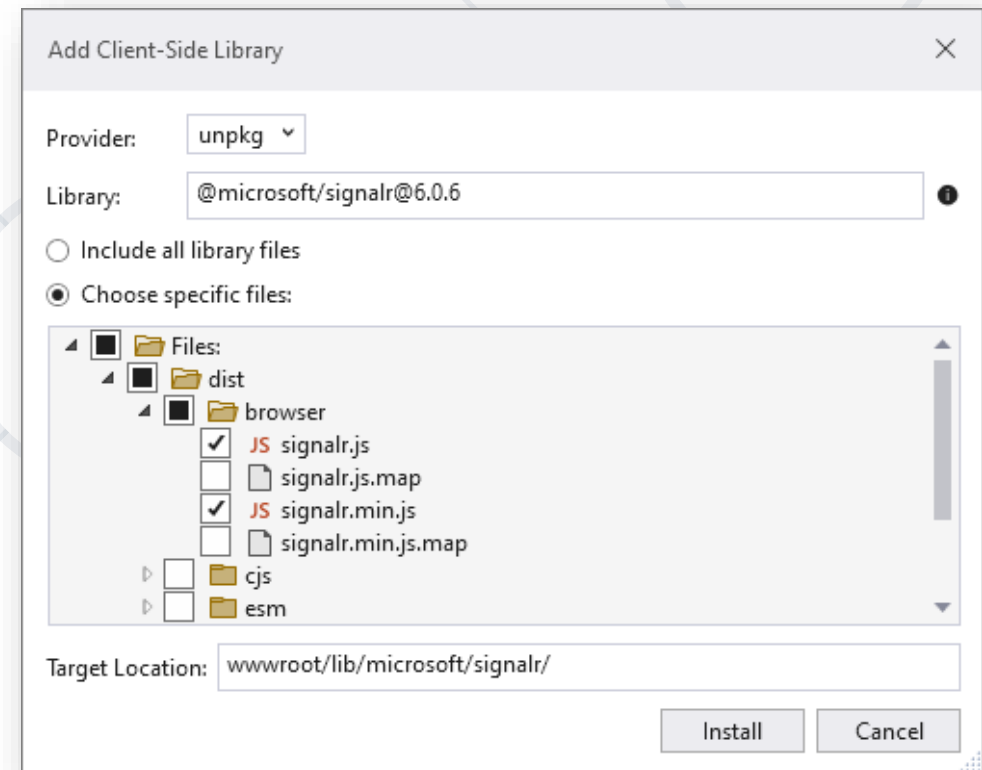
- Simple **Chat** application in **ASP.NET Core**

  - Configuring **SignalR**

```csharp
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();

var app = builder.Build();

app.MapHub<ChatHub>("/chatHub");
```

Including SignalR in ASP.NET Core

- Simple **Chat** application in **ASP.NET Core**
  - **ChatHub.cs** class file

```csharp
public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All
                .SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

Including SignalR in ASP.NET Core

- The **ChatHub** class inherits from the **SignalR Hub** class

  - The **Hub** class manages connections, groups and messaging

- The **SendMessage** method can be called by any connected client

  - It sends the received message to all clients

Including SignalR in ASP.NET Core

- Simple **Chat** application in **ASP.NET Core**

  - **JavaScript** (Client) code - **chat.js**

```javascript
var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();
// The Receive Message Client event. This will trigger, when the Back-End calls the ReceiveMessage
method
connection.on("ReceiveMessage", function (user, message) { ... });

//An error handler for connection errors
connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

// The Send Message DOM event. This will trigger the Back-End SendMessage method
document.getElementById("sendButton").addEventListener("click", function (event) { ... });
```

Including SignalR in ASP.NET Core

- Simple **Chat** application in **ASP.NET Core**

```javascript
connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);

    li.textContent = `${user} says ${message}`;
});
```

```javascript
document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

Summary

- **Real-Time Applications**
- **Web Communication** Fundamentals
  - **Short Polling, Long Polling**
  - **Server-Sent Events, WebSockets**
- **Remote Procedure Calls**
- ASP.NET Core **SignalR**
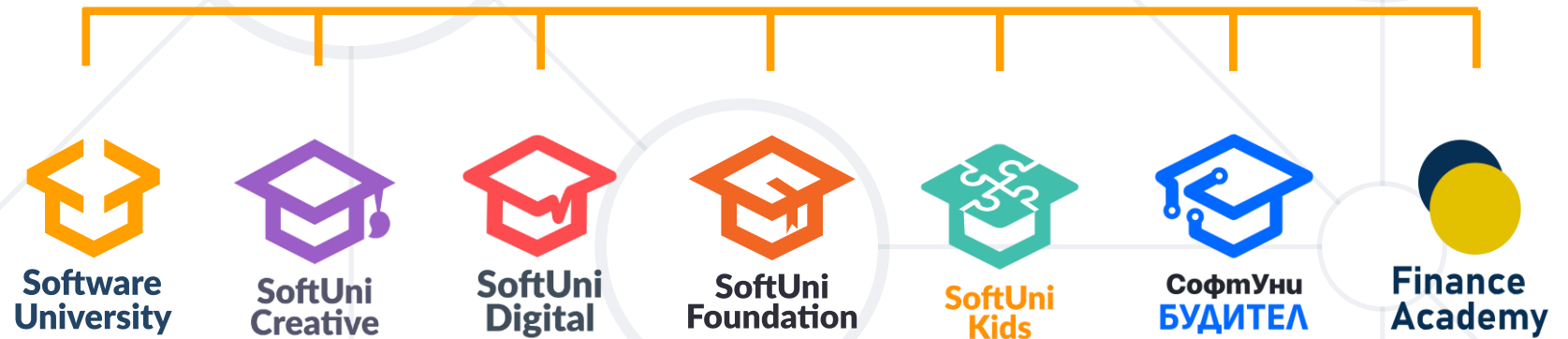 - **Real-time chat application with SignalR**

SoftUni Diamond Partners

Questions?

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers

  - softuni.bg, softuni.org

- Software University Foundation

  - softuni.foundation

- Software University @ Facebook

  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**

- Unauthorized copy, reproduction or use is illegal

- © SoftUni – https://softuni.org

- © Software University – https://softuni.bg