# WebSockets

Szolgáltatásorientált rendszerintegráció

Service-Oriented System Integration

Dr. Balázs Simon

BME, IIT

# Outline

- WebSocket technology
- Java API for WebSocket
- .NET APIs for WebSocket
- JavaScript API for WebSocket

# WebSocket protocol

# Original web philosophy

- Client-server model
- Half-duplex:
  - initiator is always the client
  - the client waits for the response of the server
- No server initiation:
  - the client has to poll the server
- HTTP headers for metadata and context information

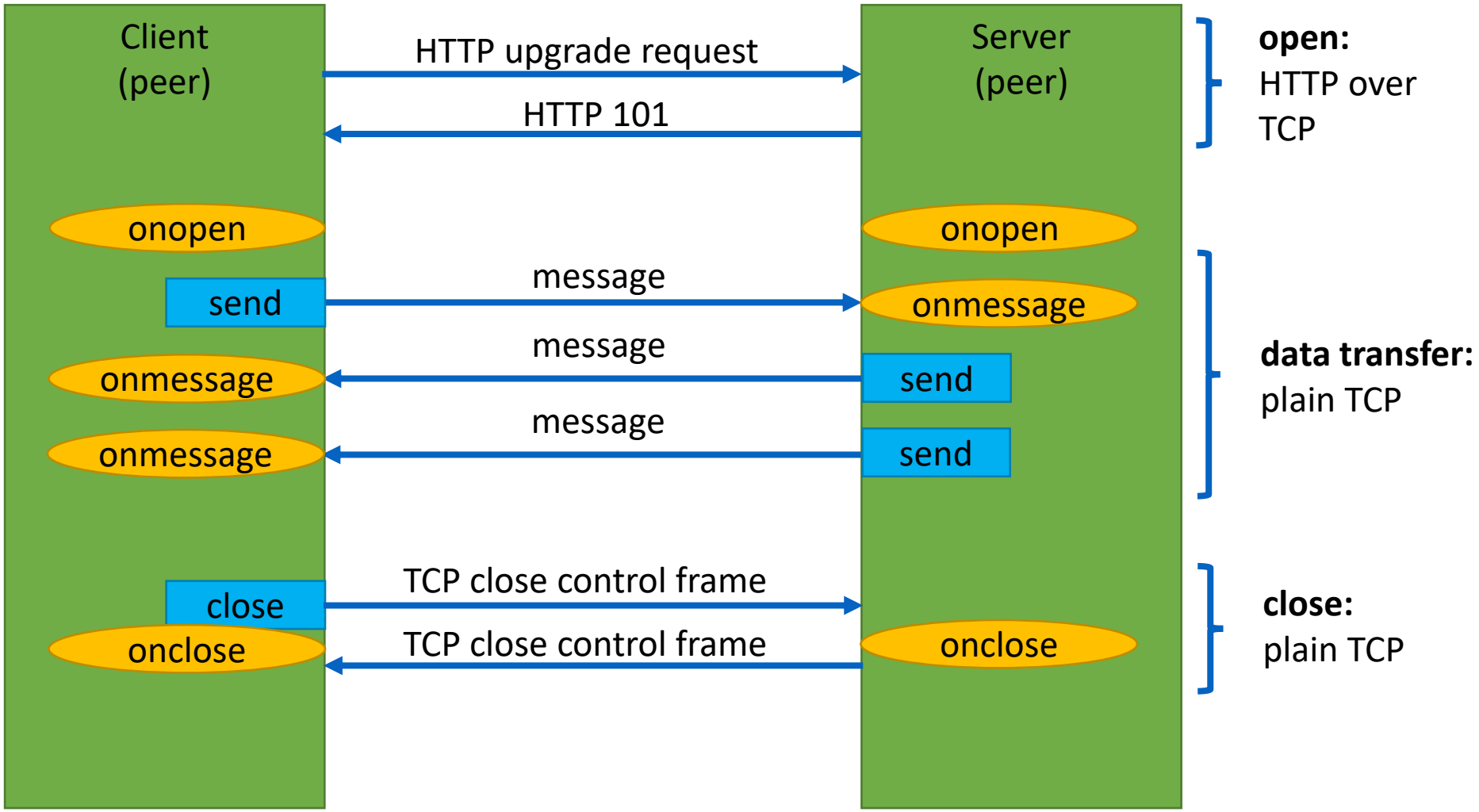
# WebSocket protocol

- A single unique plain TCP connection between the two peers
  - HTTP: each request-response requires a new TCP connection
- No HTTP header overhead
- Bidirectional
  - the client can send messages to the server
  - the server can send messages to the client
- Full-duplex:
  - each peer can send multiple messages to the other peer without waiting for a response

# WebSocket protocol

- Protocol URI scheme: ws:// or wss://
- Opening a WebSocket connection:
  - an HTTP upgrade request (handshake)
- The same underlying TCP connection used for the HTTP upgrade request will be used for WebSocket communication
- But after the upgrade:
  - no more HTTP headers
  - just plain TCP
  - the application can decide what to send through the TCP connection
- Closing a WebSocket connection:
  - just simply closing the TCP connection

# WebSocket protocol



# HTTP upgrade request

ws://echo.websocket.org/echo

```
GET /echo HTTP/1.1
Host: echo.websocket.org
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/36.0
Accept: text/html, application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: https://www.websocket.org
Sec-WebSocket-Key: gIcmfo3+pI2x3W4i6uT+ig==
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

# HTTP upgrade response

```
HTTP/1.1 101 Web Socket Protocol Handshake
access-control-allow-credentials: true
access-control-allow-headers: content-type,
    authorization, x-websocket-extensions,
    x-websocket-version, x-websocket-protocol
access-control-allow-origin: https://www.websocket.org
Connection: Upgrade
Date: Wed, 18 Mar 2015 10:35:35 GMT
Server: Kaazing Gateway
Sec-Websocket-Accept: 8XV19zYSfbKMh+ZnY8LkmDrJKpY=
Upgrade: websocket
```

# Data transfer

- When the HTTP handshake is over
- It is no longer possible to use HTTP communication
- Any peer (client or server) can send any number of messages
- Data transfer is plain TCP
- Data format: can be text or binary
  - standard subprotocol: e.g. SOAP
  - custom subprotocol: application-specific protocol

# Java API for WebSocket

# Java API for WebSocket

- Introduced in JavaEE 7
- Annotation based
- Features:
  - create a server or client endpoint
  - send and receive messages
  - text or binary protocol
  - custom encoders and decoders
  - session management

# Java annotations

- @ServerEndpoint("/...")
  - declared on a class
  - specifies a relative URL from the web application root
- @ClientEndpoint
  - declared on a class
  - specifies a client endpoint
- both of them can specify:
  - custom message encoders/decoders
  - subprotocols

# Java annotations (client and server side)

- @OnOpen
  - declared on a method
  - **onopen** event
- @OnMessage
  - declared on a method
  - **onmessage** event
- @OnError
  - declared on a method
  - **onerror** event
- @OnClose
  - declared on a method
  - **onclose** event

# WebSocket server sample

```
@ServerEndpoint("/hello")
public class HelloEndpoint {
    @OnMessage
    public String hello(String message) {
        System.out.println("Received: " + message);
        return message;
    }

    @OnOpen
    public void helloOnOpen() {
        System.out.println("WebSocket opened.");
    }

    @OnClose
    public void helloOnClose() {
        System.out.println("Closing a WebSocket.");
    }
}
```

# Method parameters

- @OnMessage parameter:
  - a single String or primitive type parameter
    - if the passed value is simple
  - a single complex type parameter
    - if the passed value is complex
    - must be deserialized by a decoder
- @OnMessage result:
  - a single String or primitive type parameter
  - a single complex type parameter
    - must be serialized by an encoder
- @OnError parameter:
  - a single Throwable typed parameter (mandatory)

# Optional method parameters

- Optional:
  - Session typed
  - EndpointConfig typed
- Zero or more:
  - String or primitive typed parameter annotated with @PathParam
  - not the same annotation as the JAX-RS @PathParam (different package), although it has the same purpose

# WebSocket server sample

```
@ServerEndpoint("/hello/{id}")
public class HelloEndpoint {
  @OnMessage
  public String hello(String message, @PathParam("id") int id) {
    System.out.println("Received: "+ message);
    return "Hello: " + message + "-" + id;
  }

  @OnError
  public void helloOnError(Throwable error) {
    System.out.println(error.getMessage());
  }
}
```

# WebSocket server sample

```
@ServerEndpoint("/hello")
public class HelloEndpoint {
  @OnOpen
  public void helloOnOpen(Session session) {
    System.out.println("WebSocket opened: "+session.getId());
  }

  @OnClose
  public void helloOnClose(Session session) {
    System.out.println("Closing a WebSocket: "+session.getId());
  }
}
```

# WebSocket client

- Defined for a client running on a server
- Not for standalone applications
- But the reference implementation (GlassFish) provides a standalone library
  - maven dependencies:
    - tyrus-client, tyrus-container-grizzly-server
- Annotation on the client endpoint class: @ClientEndpoint
- All the other annotations are the same

# WebSocket client example

```
@ClientEndpoint
public class HelloClient {
    @OnMessage
    public void message(String message){
        System.out.println(message);
    }
}
```

# Executing the client application

```
public static void main(String[] args) {
  try {
    String url = "ws://localhost:8080/WebSocketTest/hello/13";
    WebSocketContainer c =
        ContainerProvider.getWebSocketContainer();
    Session session = c.connectToServer(HelloClient.class,
                                        new URI(url));
    session.getBasicRemote().sendText("me");
    Thread.sleep(5000);
    session.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

# Encoders-decoders

- Used for serializing and deserializing complex types
- Encoders and decoders can convert between Java objects and:
  - text representation: String
  - binary representation: ByteBuffer
- Interfaces to implement:
  - text encoder interface: Encoder.Text<T>
  - binary encoder interface: Encoder.Binary<T>
  - text decoder interface: Decoder.Text<T>
  - binary decoder interface: Decoder.Binary<T>
- There is also a streamed version of these interfaces

# Coord class

```
public class Coord {
  private double x;
  private double y;

  public Coord(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() {
    return x;
  }
  public double getY() {
    return y;
  }
}
```

# Coord encoder: text

```
public class CoordEncoder implements Encoder.Text<Coord> {
  public void destroy() {
  }

  public void init(EndpointConfig config) {
  }

  public String encode(Coord coord) throws EncodeException {
    return "("+coord.getX()+","+coord.getY()+")";
  }
}
```

# Coord decoder: text

```
public class CoordDecoder implements Decoder.Text<Coord> {
  public void destroy() {
  }
  public void init(EndpointConfig config) {
  }

  public Coord decode(String text) throws DecodeException {
    String tmp = text.substring(1, text.length()-1);
    String[] xy = tmp.split(",");
    double x = Double.parseDouble(xy[0]);
    double y = Double.parseDouble(xy[1]);
    return new Coord(x, y);
  }

  public boolean willDecode(String text) {
    return text != null && text.startsWith("(");
  }
}
```

# Coord server

```
@ServerEndpoint(
  value = "/coord",
  decoders = { CoordDecoder.class },
  encoders = { CoordEncoder.class }
)
public class CoordEndpoint {
  @OnMessage
  public Coord mirror(Coord c) {
    return new Coord(c.getX(), -c.getY());
  }
}
```

# Coord client

```
@ClientEndpoint(
  decoders = { CoordDecoder.class },
  encoders = { CoordEncoder.class }
)
public class CoordClient {
  @OnMessage
  public void mirror(Coord c) {
    System.out.println(c.getX());
    System.out.println(c.getY());
  }
}
```

# JSON serialization

- For strongly typed objects:
  - use JAXB annotated classes
- For dynamically creating JSON objects:
  - JSON-P API (JSON-Processing API)
  - JsonObject: represents a JSON object
  - JsonObjectBuilder: builds a JSON object dynamically
  - JsonReader: deserialize from a stream
  - JsonWriter: serialize to a stream

# .NET APIs for WebSocket

# .NET APIs for WebSockets

- Similar interface for the server and the client:
  - Server: WebSocket API for ASP.NET Core
  - Client: ClientWebSocket class
- Very low level API
- Have to write everything manually:
  - opening a connection
  - receiving a message
  - decoding from a byte array
  - encoding to a byte array
  - sending a message
  - closing a connection
  - maintaining a list of clients

# Registering a ServerEndpoint service and a WebSocket middleware

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<ServerEndpoint>();

var app = builder.Build();
app.UseWebSockets();
app.UseMiddleware<ServerMiddleware>();

app.Run();
```

# Server endpoint

```
public class ServerEndpoint
{
  public async Task OnOpen(WebSocket socket)
  {
    Console.WriteLine($"WebSocket opened.");
  }

  public async Task<string?> OnMessage(WebSocket socket, string message)
  {
    Console.WriteLine($"Received: {message}");
    return $"Hello: {message}";
  }

  public async Task OnClose(WebSocket socket)
  {
    Console.WriteLine($"WebSocket closed.");
  }
}
```

# WebSocket middleware

```
public class ServerMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ServerEndpoint _server;

    public ServerMiddleware(RequestDelegate next, ServerEndpoint server)
    {
        _next = next;
        _server = server;
    }

    public async Task Invoke(HttpContext context)
    {
        if (!context.WebSockets.IsWebSocketRequest) return;

        var socket = await context.WebSockets.AcceptWebSocketAsync();
        await _server.OnOpen(socket);

        // ...
    }
}
```

# WebSocket middleware: Invoke

```
public async Task Invoke(HttpContext context)
{
    if (!context.WebSockets.IsWebSocketRequest) return;

    var socket = await context.WebSockets.AcceptWebSocketAsync();
    await _server.OnOpen(socket);

    try
    {
        while (socket.State == WebSocketState.Open)
        {
            await HandleMessage(socket);
        }
    }
    catch (Exception ex)
    {
        await socket.CloseAsync(WebSocketCloseStatus.InternalServerError,
                                ex.Message, CancellationToken.None);
        throw;
    }
}
```

# WebSocket middleware: HandleMessage

```
private async Task HandleMessage(WebSocket socket)
{
    var request = await StringEncoder.ReceiveAsync(socket);
    if (request.message is not null)
    {
        var response = await _server.OnMessage(socket, request.message);
        if (response is not null)
        {
            await StringEncoder.SendAsync(socket, response);
        }
    }
    else if (request.result.MessageType == WebSocketMessageType.Close)
    {
        await _server.OnClose(socket);
        await socket.CloseAsync(WebSocketCloseStatus.NormalClosure,
                                null, CancellationToken.None);
    }
}
```

# StringEncoder: Receive

```
public static class StringEncoder
{
  public static async
     Task<(WebSocketReceiveResult result, string? message)>
  ReceiveAsync(WebSocket socket)
  {
    var buffer = new byte[1024 * 4];

    var result = await socket.ReceiveAsync(
          buffer: new ArraySegment<byte>(buffer),
          cancellationToken: CancellationToken.None);

    if (result.MessageType == WebSocketMessageType.Text)
    {
      var text = Encoding.UTF8.GetString(buffer, 0, result.Count);
      return (result, text);
    }
    return (result, null);
  }
}
```

# StringEncoder: Send

```
public static class StringEncoder
{
  public static async Task SendAsync(WebSocket socket,
                                     string message)
  {
    var buffer = new ArraySegment<byte>(
       Encoding.ASCII.GetBytes(message), 0, message.Length);

    await socket.SendAsync(buffer: buffer,
                           messageType: WebSocketMessageType.Text,
                           endOfMessage: true,
                           cancellationToken: CancellationToken.None);
  }
}
```

# Client

```
using System.Net.WebSockets;
using WebSocketClient;

using (var socket = new ClientWebSocket())
{
    await socket.ConnectAsync(new Uri("wss://localhost:8080"),
                              CancellationToken.None);
    await StringEncoder.SendAsync(socket, "me");
    var response = await StringEncoder.ReceiveAsync(socket);
    if (response.message is not null)
    {
        Console.WriteLine(response.message);
    }
    else if (response.result.MessageType == WebSocketMessageType.Close)
    {
        await socket.CloseAsync(WebSocketCloseStatus.NormalClosure,
                                string.Empty, CancellationToken.None);
    }
    await socket.CloseAsync(WebSocketCloseStatus.NormalClosure,
                            string.Empty, CancellationToken.None);
}
```

# JavaScript API for WebSocket

# JavaScript API for WebSocket

- W3C recommendation
- Similar to the Java API
- Event-based:
  - onopen
  - onmessage
  - onerror
  - onclose

# JavaScript code in HTML

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script type="text/javascript">
        // here comes the JavaScript code
    </script>
</head>
<body>

</body>
</html>
```

# Testing browser support

```
function WebSocketTest() {
    if ("WebSocket" in window) {
        alert("WebSocket is supported by your Browser!");
    }
    else {
        alert("WebSocket NOT supported by your Browser!");
    }
}
```

# Creating a WebSocket

```
// Determine the URI of the server:
function getRootUri() {
    return "ws://" +
      (document.location.hostname == "" ? "localhost"
                  : document.location.hostname)
      + ":" +
      (document.location.port == "" ? "8080"
                  : document.location.port);
}
// Create the full URI:
var wsUri = getRootUri() + "/WebSocketTest/hello";
// Open a web socket:
var ws = new WebSocket(wsUri);
```

# Open event

```
ws.onopen = function()
{
    // Web Socket is connected, send data using send()
    ws.send("Message to send");
    alert("Message is sent...");
};
```

or:

```
ws.onopen = function (event) { onOpen(event) };

function onOpen(event) {
    // Web Socket is connected, send data using send()
    ws.send("Message to send");
    alert("Message is sent...");
};
```

# Message event

```
ws.onmessage = function (event)
{
    // Receiving message:
    var msg = event.data;
    alert("Message is received: "+msg);
};
```

# Error event

```
ws.onerror = function (event)
{
    // Receiving error:
    var err = event.data;
    alert("Error is received: "+err);
};
```

# Close event

```
ws.onclose = function()
{
    // WebSocket is closed:
    alert("Connection is closed...");
};
```

closing the WebSocket connection:

```
ws.close();
```

# Serializing data in JavaScript

- Simple messages can be sent as text
- Complex data can be serialized into JSON
  - JSON = JavaScript Object Notation
- JSON serialization/deserialization:
  - defined in the JavaScript standard
  - serialization: JSON.stringify()
  - deserialization: JSON.parse()

# JSON serialization

```
var coord = { "x": 3.5, "y": 4.7 };
var data = JSON.stringify(coord);
ws.send(data);
```

# JSON deserialization

```
ws.onmessage = function(msg) {
    try {
        var coord = JSON.parse(msg.data);
        alert(coord.x+","+coord.y);
    } catch (exception) {
        data = msg.data;
        console.log(data);
    }
}
```