**Http의 한계**

1 요청-응답 기반이이다 : 클라가 요청해야만 서버가 응답할 수 있다. 클라가 요청하기 전까진 서버는 뭘 할 수 없음.

2 비연결성 : 요청마다 TCP 연결이 끊겼다 다시 열린다.

3 일방향 통신 : 실질적으로 클라에서 서버 방향으로 통신한다. 반대도 하지만 여튼 단방향임.

🡺실시간 알림, 실시간 데이터, 채팅, 온라인 게임 서비스처럼 서버가 먼저 데이터를 보내고 실시간 양방향 통신하는 상황에 적합하지 않다.

**WebSocket의 특징**

1 양방향 통신이다 : 클라와 서버가 자유롭게 메시지를 주고 받을 수 있다.

2 지속적 연결 : 한 번 연결되면 끊기지 않고 계속 유지. 다시 연결해야하는 오버헤드 없음. 그로 인한 딜레이도 없음!

3 낮은 오버헤드 : http 헤더 없이 프레임으로 통신하므로 네트워크 효율이 좋음. 그렇구만.

4 이벤트 기반 메시징 가능 : 이벤트 발생시 실시간 메시지 전송 기능

**🡺 spring에서 WebSocket을 사용하는 이유**

1 실시간 데이터 처리 : 채팅, 알림, 주식 시세, 게임, 실시간 모니터링 등 실시간 서비스에 적합

2 효율적인 서버 리소스 사용

3 양방향 통신이 필요할 때 : 서버🡪클라, 클라🡪서버 모두 자유롭게 가능. 서버에서 먼저 통신 할 수 있다.

4 STOMP 프로토콜과 메시지 브로커를 활용하면 추상화 수준이 높고 관리를 쉽게 할 수 있다.

5 보안/확장성 좋은 구조 : 인증 처리, 사용자 구분, 토픽 분리, 토픽 구독 등 편리하게 구성할 수 있다.

🡺서버가 클라에게 실시간으로 데이터를 푸시할 수 있게 해주고/클라도 서버에게 “언제든지” 메시지를 전송할 수 있게 해준다.

**WebSocket이 활용되는 분야 (대표 사례)**

실시간 채팅 : 카톡, 슬랙, 디코 … : 사용자 간 메시지를 지연없이 주고 받아

실시간 알림 : 이메일 알림, 웹 알림, 푸시 알림 : 서버에서 이벤트 발생 시 클라에게 즉시 전송

데이터 스트리밍 : 주식 시세, 환율, 비트 코인 등 // 동영상 스트리밍은 다른 기술.

온라인 게임

실시간 위치 : 실시간 차량 위치 추적, 물류/배송 추적 시스템/친구 위치 공유 앱 : 실시간 위치 반영

콜라보 도구 : 실시간 문서 편집(google docs) : 여러 사용자의 편집 정보를 실시간 반영

**🡺WebSocket을 써야 하는 상황 판단 기준! 아래 중에 하나라도 충족되면 WebSocket고려**

서버가 먼저 클라에게 데이터를 보내야 한다. 예) 알림, 방송

데이터가 자주/빠르게 바뀌며 이를 즉시 반영해야 한다. 예) 대시보드

지연에 민감하다. 예) 온라인 게임

사용자가 서로에게 실시간으로 데이터를 주고 받는다. 예) 채팅

**번외1) SSE == Server-Sent Events**

서버에서 클라로 단방향 실시간으로 http를 통해 데이터를 푸시할 수 있는 표준 기술.

websocket까지 필요하지 않은 경우에 매우 유용. **서버🡪클라.**

EventSource객체로 쉽게 구현 가능하다.

예시) 실시간 알림, 뉴스 피드

**번외2) 실시간 영상 스트리밍**

SSE나 WebSocket을 사용하지 않고 HttpLiveStreaming이나 MPEG-DASH 기술을 사용한다.

**WebSocket의 한계**

연결은 잘 만들어주지만, 그 위에서 직접 메시지를 주고 받을 때는 :

어떤 메시지가 어떤 목적(구독, 전송 등)인지 명확하지 않고, 메시지 형식을 개발자가 직접 설계해야하고, 메시지 라우팅(구독자 관리, 주제별 발행 등)을 모두 수동으로 구현해야 한다.

🡺즉, WebSocket은 파이프는 만들어주지만, 그 파이프 안의 통신 규칙은 알아서 만들어야 합니다.

**STOMP**

STOMP : Simple Text Oriented Messaging Protocol : WebSocket같은 이진 기반 전송 기술 위에서 텍스트 기반 메시징을 쉽고 구조적으로 하기 위해서 사용한다.

WebSocket위에서 구조화된 메시지를 주고 받기 위한 메시징 프로토콜이다.

기능들 :

메시지 목적 정의 🡺 명령 프레임으로 지원

메시지 라우팅 🡺 destination으로 자동화

클라이언트 관리 🡺 클라이언트 구독/해지 모델 지원. destination으로 자동화.

에러 처리 🡺 ERROR 프레임 제공

메타데이터 🡺 헤더 기반 지원

텍스트 명령어로 처리.

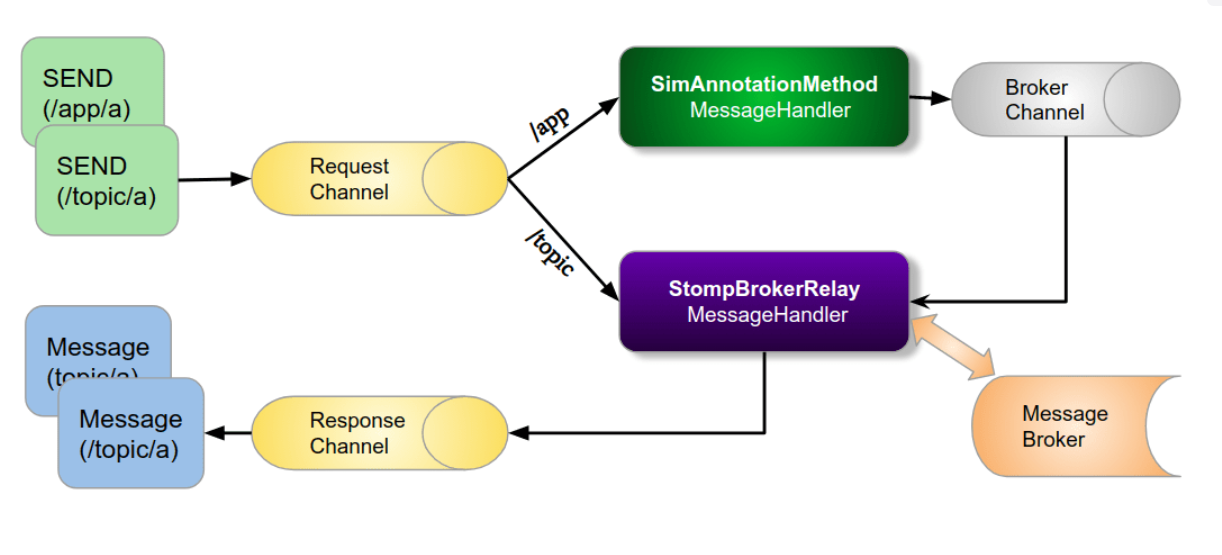
🡺 STOMP는 WebSocket통신을 HTTP처럼 구조화된 메시징 방식을 만들기 위해 존재.  
🡺 STOMP는 WebSocket의 단순한 통신 채널에 메시지 목적, 라우팅, 구독, 헤더 등을 명확히 정한 프로토콜을 입힌 것.

🡺 STOMP는 복잡한 메시징 시스템을 안정적이고 일관된 방식으로 구현하게 해주는 것.

**STOMP 개발자 생산성 향상**

클라이언트는 단순히 SUBSCRIBE, SEND명령만 보내고 되고, 서버(spring 등)은 목적지(destination)만 매핑하면  
되며 서로 메시지 구조를 맞추느라 생기는 혼란이 줄어든다.

**방식 :**



**클라 (프론트의 js)**

클라가 특정 토픽을 구독. //subscribe(토픽명)

클라가 “/app/~~”로 메시지 발행.

클라가 “/topic/~~”로 메시지 발행.

특정 토픽을 구독한 다른 클라가 메시지 응답 받음

**서버 (백의 spring controller)**

메시지 브로커에게 메시지요청 접두어 설정.

“/topic/~~”경로로 요청이 오면 해당 메시지를 같은 토픽 구독자들에게 브로드캐스트 전달.

“/app/~~”경로로 요청이 오면 해당 경로를 처리하는 컨트롤러 메서드를 통해서 메시지를 가공하고 같은 토픽   
구독자들에게 브로드캐스트 전달.

같은 토픽 구독자들에게 전달할 때 메시지 브로커에 등록된 접두어 경로라면 메시지브로커가 메시지 전송.

STOMP 용어

Public : 메시지 전송자

Subscriber : 메시지 수신자

Broker : publisher가 발행한 메시지를 subscriber에게 전달.

우리가 할일 : websocekt & stomp를 사용할 구성 및 준비를 하고  
메시지 브로커가 어떤 경로에 대해서 보내고, 컨트로러가 어떤 경로에 대해서 반응하는지 지정하고  
STOMP 프레임 구조의 바디에 어떤 내용(JSON타입)을 넣을지 정의하면 된다.

**구성 및 준비**

Build.gradle  
implementation "org.springframework:spring-websocket:${springVersion}"//웹소켓  
implementation "org.springframework:spring-messaging:${springVersion}"//메세지관련  
implementation "org.springframework.integration:spring-integration-stomp:5.5.20"//stomp관련

Src>main>java>org.scoula>config>WebSocketConfig.java  
@Configuration  
@EnableWebSocketMessageBroker//웹소켓 메시지 브로커 활성화  
public classWebSocketConfig implementsWebSocketMessageBrokerConfigurer {  
 @Override  
 public void configureMessageBroker(MessageBrokerRegistry config) {  
 config.enableSimpleBroker("/topic"); // 구독시 사용할 토픽 접두어. 처리할 토픽을 인자로 지정.  
 // 클라이언트가 발행 시 사용해야하는 접두어. 메모리기반 브로커 활성화.  
 config.setApplicationDestinationPrefixes("/app");//구별하기 위한 접두어 설정. 메시지 경로 접두어.  
 }  
 @Override  
 public void registerStompEndpoints(StompEndpointRegistry registry) {  
 registry.addEndpoint("/chat-app") // 접속 엔드포인트, ws://localhost:8080/chat-app. 클라가 접속할 브로커URL  
 .setAllowedOrigins("\*"); // CORS 허용 //origin이 달라도 웹소켓 연결을 허용하겠다.  
 }  
}

Src>main>java>org.scoula>config>WebConfig.java  
…  
@Override  
protected Class<?>[] getServletConfigClasses( ) {  
return new Class[] { ServletConfig.class,**WebSocketConfig.class** }; //적용할 config 추가.  
}  
…

데이터 준비(메시지) :

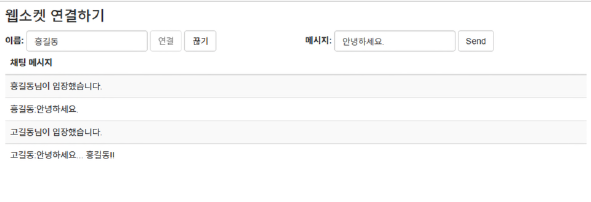
Src>main>java>org.scoula>ex>domain>GreetingMessage.java  
@Data @AllArgsConstructor @NoArgsConstructor //입장시 메시지   
public class GreetingMessage { private String name; }//가공 필요. name님이 입장하셨습니다!

Src>main>java>org.scoula>ex>domain>ChatMessage.java  
@Data @AllArgsConstructor @NoArgsConstructor  
public class ChatMessage { //일반 메시지  
 private String name; //송신자 이름  
 private String content;//내용  
}

**활용 :**

**백에서는**  
Src>main>java>org.scoula>ex>controller>ChatController.java  
@Controller @Log4j2  
public class ChatController{  
@MessageMapping("/hello") //실제 요청 경로는 “/app/hello” //아까 지정한 접두어 제외하고 지정.  
@SendTo("/topic/greetings") //해당 토픽 경로의 구독자들에게 보내겠다.  
public GreetingMessage greeting(GreetingMessage message) throws Exception {  
 log.info("greeting: " + message); //매개변수에 @RequsetBody를 붙여야 하지만 디폴트가 JSON이므로 생략가능.  
 return message; //아까 정의한 형식을 리턴. “/topic/greetings”구독자들에게 전송  
}  
@MessageMapping("/chat") //실제 요청 경로는 “/app/chat” //메시지의 body를 매개변수의 객체로 변환 전달.  
@SendTo("/topic/chat") //메시지를 컨트롤러에서 수신했는데 누구한테 보낼것인가.  
public ChatMessage chat(ChatMessage message) throws Exception {  
 log.info("chat received: " + message); //가공  
 return message;  
}  
  
**프론트에서는**  
**Js :** STOMP 라이브러리 사용. VUE가 아닌 JSP사용 전제. Webapp>resources>js>stomp.js  
// StompJs.Client 객체 생성  
const stompClient = new StompJs.Client({//설정객체를 인자로  
 brokerURL: 'ws://localhost:8080/chat-app' //아까 백에서 지정한 stomp엔드포인트  
});  
// 웹 소켓 에러 발생시 콜백 등록  
stompClient.onWebSocketError = (error) => { //에러 관련 로그 출력  
 console.error('Error with websocket', error);  
};  
// Stomp 에러 발생시 콜백 등록  
stompClient.onStompError = (frame) => { //에러 관련 로그 출력  
 console.error('Broker reported error: ' + frame.headers['message']);  
 console.error('Additional details: ' + frame.body);  
};  
// 연결 성공시 콜백  
// 구독 토픽 등록  
stompClient.onConnect = (frame) => {//연결이 성공했을 때 행동 지정  
 console.log(frame)  
 setConnected(true); //뷰에서의 버튼을 끄고 키는 함수  
 // 구독 토픽 등록 및 수신 처리 핸들러 등록  
 // 토픽 문자열: '/topic/greetings' - 입장 메시지 //토픽 경로는 @SendTo와 같게  
 stompClient.subscribe('/topic/greetings', (greeting) => {//어떤 토픽을 구독할건지.  
 console.log('/topic/greetings', greeting.body)//메시지 수신시 행동들  
 showMessage(JSON.parse(greeting.body).name + '님이 입장했습니다.');  
 }); //해당 토픽에 메시지가 응답으로 왔을 때 해당 핸들러를 호출해 처리해줘  
 // 토픽 문자열: '/topic/chat' - chat 메시지 //토픽 경로는 @SendTo와 같게  
 stompClient.subscribe('/topic/chat', (chat) => {{//어떤 토픽을 구독할건지  
 console.log('/topic/chat', chat.body)  
 const message = JSON.parse(chat.body); )//메시지 수신시 행동들  
 showMessage(`${message.name}:${message.content}`);  
});  
// 연결 성공시 입장 메시지 보내기  
const name = document.getElementById('name').value; //뷰에서 이름 추출  
stompClient.publish({ //전송 메소드  
 destination: '/app/hello', //해당 요청경로로 보낸다.  
 body: JSON.stringify({name}) // GreetingMessage에 대응 //보낼 내용  
});  
};  
// 연결됬을 때 엘리먼트 프로퍼티 변경 //뷰의 버튼 UI 비활성화  
function setConnected(connected) { //뷰의 구성 동적 변경하는 메서드  
 const connectBtn = document.getElementById('connect');  
 const disconnectBtn = document.getElementById('disconnect');  
 const messages = document.getElementById('chat-messages');  
 connectBtn.disabled = connected;  
 disconnectBtn.disabled = !connected;  
 messages.innerHTML = '';  
}  
// 연결하기  
function connect( ) {  
 stompClient.activate( );//설정된 객체를 연결  
}  
// 연결 끊기  
function disconnect() {  
 stompClient.deactivate( );//설정된 객체를 연결끊기  
 setConnected(false);  
 console.log('Disconnected');  
}  
// 메시지 전송하기  
function sendMessage( ) {//뷰에서 이름과 내용 추출 후 메시지 송신 요청  
 const name = document.getElementById('name').value;  
 const content = document.getElementById('content').value;  
 console.log({name, content})  
 stompClient.publish({  
 destination: '/app/chat',//어디로 보낼건지  
 body: JSON.stringify({name, content}); // ChatMessage에 대응 //보낼 내용  
 });  
}  
// 수신 메시지 출력하기  
function showMessage(message) {//받은 메시지 뷰에 노출  
 const messages = document.getElementById('chat-messages');  
 messages.innerHTML += '<tr><td>' + message + '</td></tr>'  
}  
// 이벤트 핸들러 설정 //이벤트 핸들러와 뷰 설정  
window.addEventListener("DOMContentLoaded", (event) => {  
 const forms = document.querySelectorAll('.form-inline');  
 const connectBtn = document.getElementById('connect');  
 const disconnectBtn = document.getElementById('disconnect');  
 const sendBtn = document.getElementById('send');  
 connectBtn.addEventListener('click', ( ) => connect( ));//connect메서드 호출  
 disconnectBtn.addEventListener('click', ( ) => disconnect( )); //disconnect 호출  
 sendBtn.addEventListener('click', ( ) => sendMessage( ));//sendMessage 호출  
 for(const form of forms) {  
 console.log(form)  
 form.addEventListener('submit', (e) => e.preventDefault( ));  
 }  
});

**뷰 : jsp webapp>WEB-INF>views>???.jsp**



CDN 준비 :

<head>  
… <script src="https://cdn.jsdelivr.net/npm/@stomp/stompjs@7.0.0/bundles/stomp.umd.min.js"></script> …   
</head>

아까 작성한 js 준비 :

<body> … <script src="/resources/js/stomp.js"></script> </body>

바디 내용 :

이름 : <input type="text" id="name" class="form-control" placeholder="이름을 이력하세요."> //입력값 js로  
메시지 : <input type="text" id="content" class="form-control" placeholder="메시지를 입력하세요...">

연결 버튼 : <button id="connect" class="btn btn-default" type="submit">연결</button>//핸들러 연결  
연결 끊기 버튼 : <button id="disconnect" class="btn btn-default" type="submit" disabled="disabled">끊기 </

메시지 출력 화면 : <tbody id="chat-messages">//showMessage함수로인해 여기에 추가됨 </tbody>