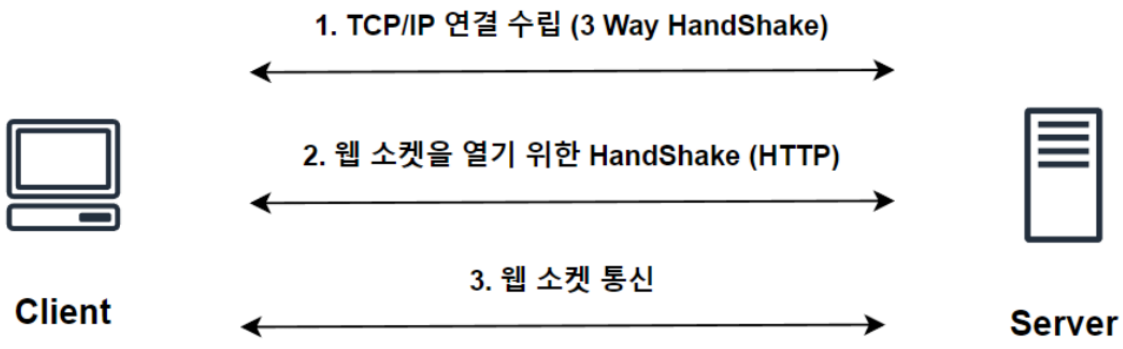
채팅 서비스

✨WebSocket 동작방식

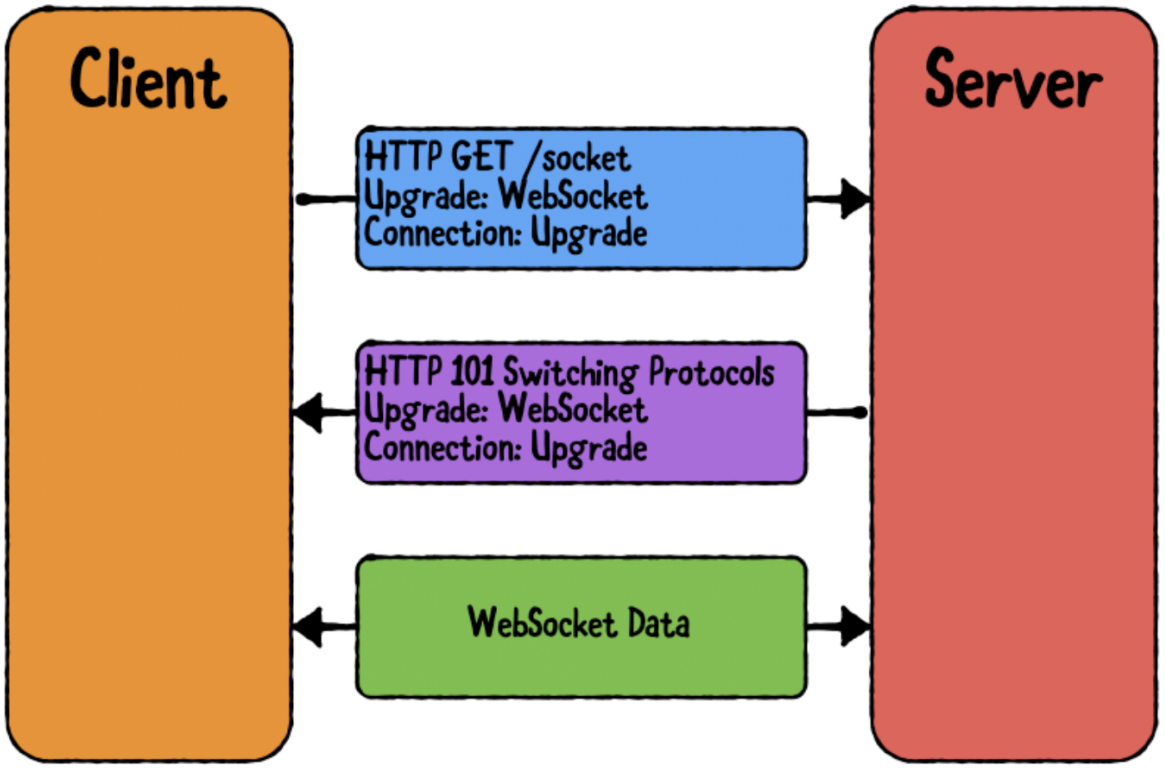


> **WebSocket**
>
> - 서버와 클라이언트 사이에 소켓 커넥션을 유지하며, 양방향 통신이 가능한 기술
>
> **3 Way HandShake**
>
> - 연속적인 데이터 전송의 신뢰성을 보장하기 위해 이러한 과정 진행

💡 기존 TCP는 **TCP layer** 에서 HandShake 통해 연결을 수립

WebSocket은 **HTTP 요청 기반**으로 연결

- Upgrade 헤더, Connection 헤더 포함하는 HTTP 요청
- 웹소켓 연결 아래와 같이 101 으로 응답 코드를 보내줌



✨WebSocket (서버가 1대일 경우)

WebSocketConfiguration.java

```
package practice.websocket.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.Enabl
```

```
import org.springframework.web.socket.config.annotation.WebSo
import org.springframework.web.socket.config.annotation.WebSo
import practice.websocket.handler.WebSocketHandler;

@Configuration
@EnableWebSocket
public class WebSocketConfiguration implements WebSocketConfi
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerReg
        registry
                // webSocket server 의 endpoint => url:port/ro
                .addHandler(signalingSocketHandler(), "/room"
                .setAllowedOrigins("*");
    }

    @Bean
    public WebSocketHandler signalingSocketHandler() {
        return new WebSocketHandler();
    }
}
```

WebSocketHandler.java

```
package practice.websocket.handler;

import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.socket.CloseStatus;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHa
import practice.websocket.entity.Message;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
```

```
@Slf4j
public class WebSocketHandler extends TextWebSocketHandler {

    // 예를 들어, 채팅방에 이미 접속 해 있던 유저들에게 신규 유저가 들어온
    // 그러면 채팅방에 접속 해 있던 기존 접속 사용자의 웹소켓 세션을 전부
    // <세션 Id => String, 세션 => WebSocketSession> key - valu
    private final Map<String, WebSocketSession> sessions = new
    private final ObjectMapper objectMapper = new ObjectMappe

    // 웹소켓 연결
    @Override
    public void afterConnectionEstablished(WebSocketSession s
        String sessionId = session.getId();
        sessions.put(sessionId, session);

        // 입장하였을 때, 보낼 메세지
        Message message = Message.builder()
                .sender(sessionId)
                .receiver("all")
                .build();
        message.newConnect();

        String jsonMessage = objectMapper.writeValueAsString(

        sessions.values().forEach(s -> {
            try {
                if (!s.getId().equals(sessionId)) {
                    s.sendMessage(new TextMessage(jsonMessage
                }
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });
    }

    // 양방향 데이터 통신
    @Override
    protected void handleTextMessage(WebSocketSession session
```

```
        // client 가 보낸 json 문자열 메세지
        Message message = objectMapper.readValue(textMessage.
        message.setSender(session.getId());

        WebSocketSession receiver = sessions.get(session.getI

        if (receiver != null && session.isOpen()) {
            receiver.sendMessage(new TextMessage(objectMapper
        }
    }

    // 소켓 연결 종료
    @Override
    public void afterConnectionClosed(WebSocketSession sessio
        String sessionId = session.getId();
        // 세션 저장소에서 연결이 끊긴 유저 삭제
        sessions.remove(sessionId);

        // 종료 메세지 생성
        final Message message = new Message();
        message.closeConnect();
        message.setSender(session.getId());

        // 남은 유저에게 메세지 전송
        sessions.values().forEach(s -> {
            try {
                s.sendMessage(new TextMessage(objectMapper.wr
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });
    }

    // 소켓 통신 에러
    @Override
    public void handleTransportError(WebSocketSession session
        // 브라우저를 그냥 종료하면 1001 코드가 날라옴 (명시적으로 종료
```

```
    }
}
```

Message.java

```
package practice.websocket.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;

@Getter
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Message {
    private String type;
    private String sender;
    private String receiver;
    private Object data;

    public void setSender(String sender) {
        this.sender = sender;
    }

    public void newConnect() {
        this.type = "new";
    }

    public void closeConnect() {
        this.type = "close";
    }
}
```

💡 현재 코드는 세션을 서버에서 따로 관리할 수 있도록 MAP 자료구조로 정의하였다.

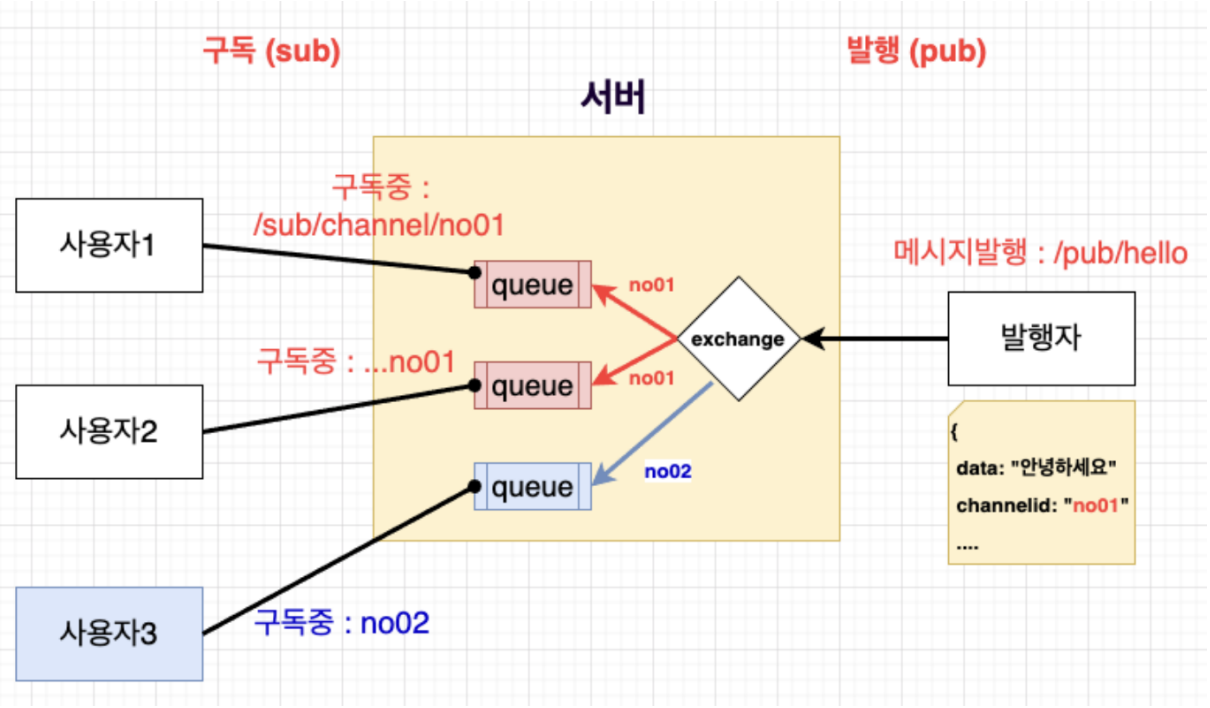
또한 메세지를 어떻게 처리할 지도 직접 구현하였음

하지만, 웹소켓 서버가 2대 이상이라면 **메모리 기반**으로 관리하는 세션 정보를 서로 알아야 함!

✨WebSocket STOMP (서버가 2대 이상일 경우)

> **STOMP (Simple Text oriented Messaging Protocol)**
>
> - 메세징 전송을 효율적으로 하기 위한 프로토콜
> - pub / sub 기반으로 동작
> - 메세지 송신 / 수신에 대한 처리가 명확하게 정의되어 있음
> - WebSocketHandler 를 직접 구현 할 필요 없이 @MessageMapping를 사용하여 메세지 발행 시 엔드포인트를 별도로 분리해서 관리

✨ Pub(발행) / Sub(구독) 에 대한 이해



사용자 구분

- 사용자 1, 2: 조선일보 신문(no01) 구독
- 사용자 3: 한겨레 신문(no02) 구독

서버 구조

- Exchange: 메시지 분배를 담당하는 중앙 처리기
- Queue: 각 구독자에게 할당된 메시지 대기열
- 발행자: '/pub/hello' 경로로 메시지 전송

작동 방식

메시지 전달 과정

- 발행자가 채널 ID를 지정하여 메시지를 전송
- Exchange는 해당 채널의 구독자 Queue로 메시지를 분배
- 구독자는 자신의 Queue에서 메시지를 수신

주의사항

- 지정된 채널 ID에 구독자가 없으면 메시지 전송이 실패
- 구독자는 자신이 구독한 채널의 메시지만 수신 가능
- 각 구독자의 Queue는 독립적으로 운영