WebSocket-based Programming

JSR-356

❑ **About**

– 본 문서는 JSR-356 스펙에 따라 WebSocket 프로토콜을 이용한 양방향 통신을 위한 참고 문서임.

❑ **목차**

대덕인재개발원 : 최희연

❑ JSR 356, Java API for WebSocket

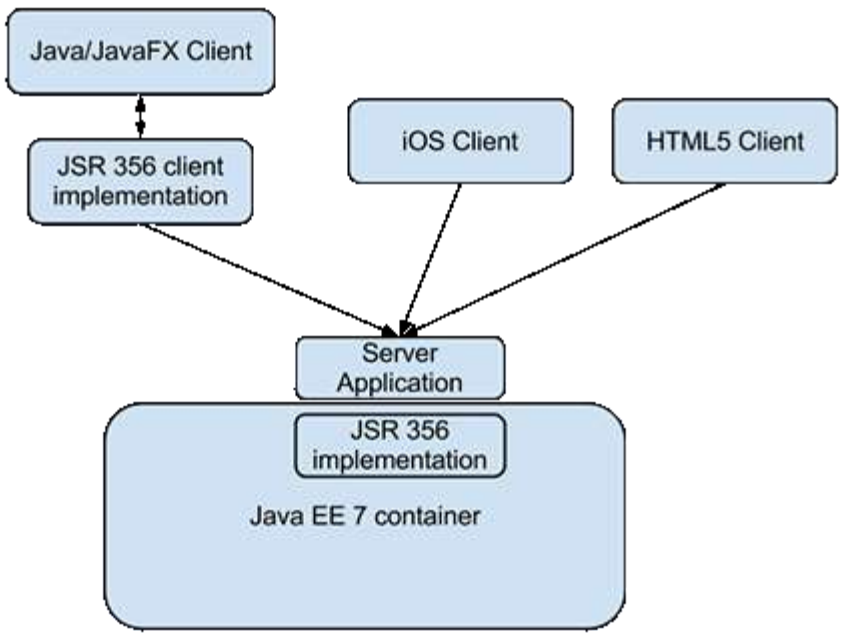
(http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html)

대부분의 웹 기반 클라이언트-서버 어플리케이션에서 HTTP의 요청-응답 모델은 서버로부터 클라이언트로의 정보 전송이 요청이 발생한 경우에 한해서만, 이루어진다는 한계를 지니고 있다. 이로 인해 Long-polling 이나 comet 같은 기술들이 등장했으나, 클라이언트와 서버 사이의 full-duplex(전이중) 통신에 대한 요구는 꾸준히 증가해왔다.

2011년, IETF가 WebSocket 프로토콜에 대한 표준(RFC6455)을 정의한 이후 많은 브라우저들이 웹소켓 프로토콜을 지원하는 클라이언트 API 구현체를 제공하고, 또 웹소켓 프로토콜을 지원하는 자바 라이브러리도 다수 개발되어왔다. 웹소켓 프로토콜은 HTTP 연결을 웹소켓으로 업그레이드하기 위한 기술로, 이를 통해 웹소켓의 양 종단(peer) 간의 독립된 양방향 메시지 전송이 가능해진다. 간혹, 정보전송시 추가되는 헤더나 쿠키들이 페이로드 보다 더 긴 경우가 발생하기도 하는데, 웹소켓은 헤더나 쿠키가 필요없어 낮은 대역폭으로 전송이 가능하고, 따라서 대부분 간단한 단문 메시지 전송에 사용된다.

- JSR 356

JSR 356 은 자바 클라이언트 뿐만 아니라 서버 어플리케이션까지 웹소켓 연결로 통합할 수 있는 웹소켓 프로토콜 지원 스펙을 의미한다. 모든 웹소켓 구현은 JSR 356을 준수하도록 제안하고 있으며, 이에 따라 실제 웹소켓 구현(WebSocket container)에 독립적인 웹소켓 기반 어플리케이션의 구현이 가능해진다. Java EE 7 에 표준 스펙으로 채택되어, 모든 JEE7 서버들은 JSR 356을 준수하는 웹소켓 프로토콜 구현을 제공하고 있다. 웹소켓 연결이 수립되면, 서버와 클라이언트는 완벽하게 대칭을 이루는 peer가 되기때문에, JSR 356은 JEE7 에서 필요한 서버 API뿐만 아니라, 자바 클라이언트 API 까지 제공하고 있으며, 클라이언트 API와 서버 API의 차이는 거의 없다. 다음 그림과 같이 웹소켓을 지원하는 클라이언트-서버 어플리케이션은 하나의 서버 컴포넌트와 여러 개의 클라이언트 컴포넌트들로 구성되는 게 일반적이다.



이 그림의 자바로 작성된 서버 어플리케이션에서 JEE7 컨테이너에 포함된 JSR 356 구현에 의해 웹소켓 프로토콜에 대한 상세 처리가 이루어진다. JavaFX 클라이언트는 JSR 356 스펙을 준수하고, 웹소켓 프로토콜을 지원하는 클라이언트 모델에 해당하고, IOS나 HTML5 클라이언트는 서버 어플리케이션과의 통신에 RFC 6455 표준을 준수하는 non-Java 구현(Javascript WebSocket API)에 해당한다.

- 프로그래밍 방식

JSR 356 스펙은 대부분의 JavaEE 개발자들이 쉽게 적용할 수 있도록 표준패턴과 기술에 따라 어노테이션과 Injection 기법을 지원한다 (annotation-driven and injection).
웹소켓 프로그래밍 방식에는 두가지 모델이 있는데, 두 방식 모두 event-driven 방식에 기반한다.

✓ Annotation-driven 방식은 POJO에 어노테이션을 사용하여 양 피어간에 웹소켓 라이프사이클 이벤트를 처리할 수 있도록 한다.

✓ Interface-driven 방식은 Endpoint 인터페이스의 구현체를 통해 고정된 시그니처의 메소드로 웹소켓 이벤트를 처리할 수 있다.

웹소켓의 라이프사이클 이벤트는 다음과 같은 흐름을 갖는다.

✓ 클라이언트 피어에서 HTTP 핸드쉐이크 요청으로 웹소켓 통신을 시작되고,

✓ 이에 대해 서버 피어에서 이 핸드쉐이크 요청을 받아 핸드쉐이크 응답을 전송하면,

✓ 완전하게 대칭인 전이중 양방향 연결이 수립되고,

✓ 어느 한쪽에서 연결을 종료할 때까지 양 피어는 서로간의 메시지 송수신이 가능하다.

대부분의 웹소켓 라이프사이클 이벤트는 어노테이션 방식이나 인터페이스 지향 방식 모두에서 자바 메소드 단위로 처리될 수 있다.

- Annotation-driven 방식

어노테이션방식은 웹소켓 요청을 받아들일 endpoint를 POJO 지향적으로 작성하고, 이를 endpoint로 지정하기 위해 @ServerEndpoint 어노테이션을 사용한다. 해당 어노테이션의 value 속성으로 웹소켓의 경로(url)를 기술하여, 컨테이너에 의해 해당 경로의 웹소켓 리퀘스트에 대한 endpoint로 사용될 수 있다.

```
@ServerEndpoint("/example/hello")
public class MyClientEndpoint{ //... }
```

위 코드는 "ws://host_name/example/hello" 경로에 대해 endpoint 로 발행되는데, 이러한 경로에는 경로인자(path parameter)가 포함되어 웹소켓 요청에 인자를 포함시킬 수 있다(/example/hello/{user}). 인자를 사용하는 웹소켓에 요청에 대한 핸드쉐이크 응답과 메시지 발송은 @PathParam 파라미터를 포함하고 있는 메소드를 통해서만 처리가 가능하다.

```
@ServerEndpoint("/example/hello/{user}")
public class MyEndpointPathParams{
            @OnMessage
            public void process( @PathParam("user") String user){
                        //............
            }
}
```

웹소켓 연결을 초기화할수 있는 클라이언트 endpoint 는 POJO에 @ClientEndpoint를 사용하여 작성하며, 이 어노테이션은 경로 매핑을 위한 value 속성을 갖지 않는데, 이는 클라이언트 endpoint 에서 들어오는 요청을 대기할 필요가 없기 때문이다.

```
@ClientEndpoint
public class MyClientEndpoint{}
```

어노테이션 방식으로 웹소켓 통신을 초기화하기 위한 클라이언트측 코드는 다음과 같다.

```
javax.websocket.WebSocketContainer container =
              javax.websocket.ContainerProvider.getWebSocketContainer();
container.conntectToServer(MyClientEndpoint.class,
              new URI("ws://host_name/example/hello"));
```

웹소켓 연결(Session)이 성립되면, @ClientEndpoint와 @ServerEndpoint 클래스들 간의 각 endpoint를 호출할 수 있게 된다. 연결 수립 후 클라이언트와 서버의 endpoint 간의 유일한 연결, 즉 Session 이 생성되고, @OnOpen 어노테이션을 가진 endpoint 메소드가 호출되는 event-driven 방식이 사용되는데, open 이벤트 핸들러 메소드는 다음과 같은 파라미터들을 가질 수 있다.
javax.websocket.Session : 생성된 세션 정보를 가진 객체.
javax.websocket.EndpointConfig : endpoint 설정 및 환경에 대한 정보를 가진 객체
@PathParam 파라미터 : endpoint 경로에 포함된 경로인자를 확인하기 위한 파라미터

```
@OnOpen
public void myOnOpen(Session session){
              System.out.println("WebSocket opened : "+session.getId());
}
```

세션 인스턴스는 웹소켓이 close 될 때까지만 유효한 객체로, Session 클래스는 웹소켓 연결에 대한 정보를 조회할 수 있는 여러가지 메소드를 포함하고 있다. 또한 Session의 getUserProperties()를 사용하여 세션이 유효한 동안 어플리케이션 내에서 유지해야 하는 데이터들을 관리하기 위한 공간을 확보할 수도 있다. 이 map을 통해 세션 혹은 어플리케이션에 구체적인 데이터의 이벤트 핸들러 메소드 간 공유가 가능하다.
웹소켓 endpoint 는 @OnMessage 어노테이션을 갖는 메소드를 통해 메시지를 수신하는데, 수신 메소드는 javax.websocket.Session, @PathParam 파라미터들, 텍스트 메시지 수신을 위한 String 파라미터나, 바이너리 메시지 수신을 위한 ByteBuffer 혹은 byte[] 등의 파라미터를 가질 수 있다.

```
@OnMessage
public void myOnMessage(String txt){
              System.out.println("WebSocket received message: "+txt);
}
```

만일 OnMessage 핸들러에 리턴타입이 존재한다면, 웹소켓 컨테이너는 리턴값 자체를 다른 피어로 전송해버린다. 그러나, 웹소켓은 메시지 전송시 chunk 단위로 분리하여 수신되기 때문에 메시지 수신 메소드를 통해 수신된 메시지가 전체 메시지 페이로드 중 일부에 해당한다면 일부 메시지만 반대편 피어로 전송되는 경우가 발생할 수 있기 때문에, 가능하면 메시지 수신 메소드의 리턴타입은 void 로 설정하고, 에코가 필요하다면, remoteEndpoint 를 통해 직접 처리하거나 스트림 전역변수를 선언하여 핸들링하는 편이 안전하다.

```
@OnMessage
public String myOnMessage(String txt){
              return txt.toUpperCase();
}
```

만일 OnMessage 핸들러에 리턴타입이 존재한다면, 웹소켓 컨테이너는 리턴값 자체를 다른 피어로 전송해버린다. 그러나, 웹소켓은 메시지 전송시 chunk 단위로 분리하여 수신되기 때문에 메시지 수신 메소드를 통해 수신된 메시지가 전체 메시지 페이로드 중 일부에 해당한다면 일부 메시지만 반대편 피어로 전송되는 경우가 발생할 수 있기 때문에, 가능하면 메시지 수신 메소드의 리턴타입은 void 로 설정하고, 에코가 필요하다면, RemoteEndpoint 를 통해 직접 처리하거나 스트림 전역변수를 선언하여 처리하는 편이 안전하다.

```
RemoteEndpoint.Basic remoteBasic;
@OnOpen
public void myOnOpen(Session session){
            remoteBasic = session.getBasicRemote();
}

Writer writer;

@OnMessage
public void myOnMessage(String txt, boolean last) throws IOException{
            if(writer==null){
                        writer = session.getBasicRemote().getSendWriter();
            }
            writer.write(txt.toUpperCase());
            if(last){
                        writer.close();
                        writer = null;
            }
            // ---- 혹은 -------------
            remoteBasic.sendText(txt.toUpperCase(), last);
}
```

웹소켓 핸드쉐이크가 형성된 후, open 이벤트 핸들러 내부에서 웹소켓 연결의 반대편 피어에 대한 정보를 가진 RemoteEndpoint 구현을 미리 확보해두면 필요한 코드 부분에서 적절히 사용할 수 있다.

웹소켓 통신이 종료되면 @OnOpen 어노테이션을 가진 메소드가 호출된다.

javax.websocket.Session : 이 세션 객체는 close 이벤트가 발생하고 메소드가 리턴된 이후에는 사용할수 없는 세션 객체가 된다.

javax.websocket.CloseReason : 종료 코드와 종료 메시지를 가진 객체로, 종료 코드의 종류는 CloseCodes가진 상수들로 확인할 수 있다.

더불에 @PathParam 파라미터까지 close 이벤트 핸들러 메소드의 파라미터로 사용될수 있다.

```
@OnClose
public void myOnClose(CloseReason reason){
      System.out.println("Closing a WebSocket due to"+reason.getReasonPhrase());
}
```

이외에 @OnError 어노테이션을 통해 에러가 발생한 경우나 에러 메시지가 수신된 경우에 대한 이벤트 처리도 가능하다.

```
@OnError
public void myOnError(Session session, Throwable e){
      System.out.println(session.getId()+", "+e.getMessage());
}
```

- **Interface-driven 방식**

어노테이션 지향 방식과 다르게 인터페이스 지향 방식은 고전적인 프로그래밍 방법론에 따라, 고정된 시그니처의 메소드들을 가진 인터페이스(Endpoint)를 상속하여 구현하는 방식을 따른다.

```
public class MyOwnEndpint extends javax.websocket.Endpoint{
              public void onOpen(Session session, EndpointConfig config) {}
              public void onClose(Session session, CloseReason closeReason) {}
              public void onError(Session session, Throwable throwable) {}
}
```

수신된 메시지에 대한 핸들러는 MessageHandler.Partial<T> 나 MessageHandler.Whole<T> 구현체로 작성 할 수 있고, 처리할수 있는 메시지의 종류는 각 구현체의 generic type 으로 결정되며, 메시지 핸들러들은 웹소켓 연결직후 open 이벤트 핸들러에서 해당 세션에 등록하는 과정을 거쳐야만 한다.

```
public void onOpen(Session session, EndpointConfig config){
              session.addMessageHandler(new MessageHandler(){...});
}
```

Partial나 Whole 타입은 MessageHandler 의 하위 인터페이스로 분리된 메시지의 chunk 각각에 대한 처리가 필요하다면 Partial를, 메시지 페이로드 전체가 수신된 이후의 핸들러가 필요하다면 Whole 의 구현체를 각기 만들어 적용할수 있다. 만일 동일 메시지 타입에 대해 Partial과 Whole 이 둘다 핸들러로 등록되었다면 Whole 구현체 핸들러가 무시된다.

```
public void onOpen(Session session, EndpointConfig config){
              final RemoteEndpoint.Basic remote = session.getBasicRemote();
              session.addMessageHandler(new MessageHandler.Partial<T>(){
                            public void onMessage(String text, boolean last){
                                          try{

              remote.sendText(text.toUpperCase(), last);
                                          }catch(IOException e){
                                                        // excetpion handling code
                                          }
                            }
              });
}
```

수신메시지는 각 메시지 타입을 직접 핸들링하거나, 메시지 타입 복부호화를 위한 Encoder, Decoder 등이 제공되는 등 Java WebSocket API 등의 여러가지 지원요소들을 통해 다양한 형태의 메시지 송수신이 가능하다.
가장 기본적인 메시지 전송 형태는
텍스트 기반 메시지
바이너리 메시지
Pong 메시지 등이 있는데, 이러한 메시지들은 인터페이스 지향 방식을 사용하는 경우, 각기 다른 타입의 MessageHandler 를 등록하여 처리할 수 있고, 어노테이션 지향 방식을 사용하는 경우, OnMessage 메소드의 파라미터 타입으로 각기 다른 메시지 핸들러를 작성할 수 있다.

@OnMessage의 api 문서를 보면, 다음과 같은 명확한 내용이 명시되어 있다.
(http://docs.oracle.com/javaee/7/api/javax/websocket/OnMessage.html)

@OnMessage 메소드의 파라미터 종류.

1. 메시지 수신을 위한 파라미터

✓ 텍스트 메시지에 대한 핸들러 메소드의 경우.

전체 메시지를 수신하기 위해 String 파라미터.

기본형 데이터 전체 메시지를 수신한다면, primitive type 이나 적절한 Wrapper 타입 파라미터.

부분 메시지 수신시에는, String 과 boolean 타입 파라미터.

블럭킹 방식의 스트림 데이터를 수신한다면, Reader 타입 파라미터.

endpoint 에서 특정 타입의 객체로 메시지를 파싱해야 한다면, Decoder.Text 나 Decoder.TextStream 파라미터.

✓ 바이너리 메시지에 대한 핸들러 메소드의 경우.

전체 메시지 수신은 byte[] 이나 ByteBuffer 파라미터.

부분 메시지 수신은 byte[] 이나 ByteBuffer 파라미터와 boolean 파라미터를 함께 선언.

블럭킹 방식의 스트림 데이터를 수신할 경우는 InputStream 파라미터.

바이너리 디코더를 통해 특정 객체 타입으로 수신할 경우는 Decoder.Binary 나 Decoder.BinaryStream

✓ Pong 메시지에 대한 핸들러 메소드의 경우.

pong 메시지 수신을 위해 PongMessage 타입의 파라미터

2. 서버사이드 엔드포인트에서 경로인자를 확보하기 위한, @PathParam 파라미터들.

3. 필요하다면 Session 인자도 선언 가능함.

WebSocket 라이브러리 내에는 특정 피어에서 다른 피어로 전송되는 텍스트 기반의 메시지나 바이너리 메시지들을 수신 후 처리하기 위해, 자바의 객체 기반의 타입으로 메시지를 복부화 할수 있는 여러가지 Encoder/Decoder들이 포함되어 있다. 이러한 Encoder/Decoder 들을 이용하여, XML 이나 JSON 형태로 송수신되어야 하는 메시지들에 대한 마샬링 혹은 언마샬링 작업을 처리할 수 있다. 다음 예제들은 Encoder 와 Decoder 를 사용하여 텍스트 기반의 메시지를 특정 자바 객체로 파싱하는 경우이다.

```
@ServerEndpoint(value="/endpoint",
              encoders=MessageEncoder.class, decoders=MessageDecoder.class)
public class MyEndpoint{
              //…
}
class MessageEncoder implements Encoder.Text<MyJavaObject>{
              @Override
              public String encode(MyJavaObject obj) throws EncodingException{
                // .... fomatting code
              }
}
class MessageDecoder implements Decoder.Text<MyJavaObject>{
              @Override
              public MyJavaObject decode(String src) throws DecodeException{
                // ..... parsing code
              }
              public boolean willDecode(String src){
                // .. MyJavaObject 타입으로 파싱이 가능한 텍스트 메시지인지 여부를 리턴.
              }
}
```

✓ Encoder의 하위인터페이스 종류

Encoder.Text : 자바 객체를 텍스트 메시지로 변환

Encoder.TextStream : 자바 객체를 문자 스트림에 추가

Encoder.Binary : 자바객체를 바이너리 메시지(BLOB) 로 변환

Encoder.BinaryStream : 자바 객체를 바이너리 스트림에 추가

✓ Decoder의 하위인터페이스 종류

Decoder.Text : 텍스트 메시지를 자바 객체로 변환

Decoder.TextStream : 문자 스트림을 자바 객체로 읽어옴.

Decoder.Binary : 바이너리 메시지를 자바 객체로 변환.

Decoder.BinaryStream : 바이너리 스트림으로부터 자바 객체를 읽어옴.

자바의 웹소켓 스펙은 IETF의 웹소켓 표준안에 따라 웹소켓 프로토콜을 지원하는 어플리케이션을 개발할 수 있는 표준에 준하는 방법을 제공하므로, 웹 클라이언트나 네이티브 클라이언트 모듈이 웹소켓 컨테이너에 독립적으로 동작하면서 자바의 백앤드 모듈과 쉽게 통신할 수 있는 웹소켓 기반 어플리케이션의 구현이 가능하다.

❑ WebSocket Server programming : 에코서버 구현

- javax.websocket-api의 이용

1. 웹 프로젝트 생성

2. 서버사이드 Endpoint 의 작성

1) Interface-driven 방식

```
/**
 * 서버의 Endpoint 등록과정에서 웹소켓 경로 매핑을 위한 어노테이션
 * websocket.MappingPath
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MappingPath {
	public String value();
}
```

```
/**
 * 웹 소켓 통신의 서버측 endpoint 구현체.
 * 고정된 인터페이스로 메시지 이벤트 핸들러를 작성한 클래스.
 * websocket.echo.SampleEchoEndpoint
 */
@MappingPath("/websocket/echoProgrammatic")
public class SampleEchoEndpoint extends Endpoint {

	public SampleEchoEndpoint() {
		System.out.println(this.getClass().getSimpleName() + "생성");
	}

	/**
	 * 단순 텍스트 메시지를 처리하기 위한 핸들러
	 */
	private static class EchoMessageHandlerText implements
			MessageHandler.Partial<String>{
		private RemoteEndpoint.Basic remoteBasic;
		private EchoMessageHandlerText(RemoteEndpoint.Basic remoteBasic)
		{
		 this.remoteBasic = remoteBasic;
		}

		@Override
		public void onMessage(String message, boolean last) {
			try {
				remoteBasic.sendText(message, last);
			} catch (IOException e) {
			e.printStackTrace();
			}
		}
	}
```

```
	/**
	 * 이진 데이터 수신에 사용되는 핸들러.
	 */
	private static class EchoMessageHandlerBinary implements
		MessageHandler.Partial<ByteBuffer> {
		private RemoteEndpoint.Basic remoteBasic;
		private EchoMessageHandlerBinary(RemoteEndpoint.Basic remoteBasic)
		{
			this.remoteBasic = remoteBasic;
		}

		@Override
		public void onMessage(ByteBuffer message, boolean last) {
			try {
				remoteBasic.sendBinary(message, last);
			} catch (IOException e) {
			e.printStackTrace();
			}
		}
	}
	@Override
	public void onOpen(Session session, EndpointConfig endPointConfig) {
		RemoteEndpoint.Basic basicEndPoint = session.getBasicRemote();
		MessageHandler txtHandler =
			new EchoMessageHandlerText(basicEndPoint);
		session.addMessageHandler(txtHandler);
		MessageHandler binHandler =
			new EchoMessageHandlerBinary(basicEndPoint);
		session.addMessageHandler(binHandler);
		System.out.println(session.getRequestURI() + "연결 성공");
	}
	@Override
	public void onClose(Session session, CloseReason closeReason) {
		super.onClose(session, closeReason);
		CloseCode code = closeReason.getCloseCode();
		if (!(code == CloseCodes.NORMAL_CLOSURE ||
			code == CloseCodes.NO_STATUS_CODE)) {
		System.out.println(closeReason.getReasonPhrase() + "로 인해 종료");
		} else {
		System.out.println(session.getRequestURI() + " 연결 종료");
		}
	}
	@Override
	public void onError(Session session, Throwable thr) {
		super.onError(session, thr);
		thr.printStackTrace();
	}


}
```

2) Annotation-driven 방식

```
/**
 * Annotation-driven 방식의 Endpoint 상위로
 * open, error, close 이벤트에 대한 기본적인 메소드를 가지고 있음.
 * websocket.echo.annotation.AbstractEchoAnnotation
 */
public abstract class AbstractEchoAnnotation {

	public AbstractEchoAnnotation(){
		System.out.println(this.getClass().getSimpleName()+"생성");
	}

	@OnOpen
	public void onOpen(Session session, EndpointConfig config){
		System.out.println(session.getUserProperties());
		System.out.println(session.getRequestURI()+"연결 성공, id :
		"+session.getId());
	}

	@OnError
	public void onError(Session session, Throwable e){
		e.printStackTrace();
		try {
			session.close(new CloseReason(CloseCodes.CLOSED_ABNORMALLY,
			e.getMessage()));
		} catch (IOException e1) { }
	}

	@OnClose
	public void onClose(Session session, CloseReason reason){
		CloseCode code = reason.getCloseCode();
		// 비정상 종료
		if(!(code == CloseCodes.NORMAL_CLOSURE ||
				code == CloseCodes.NO_STATUS_CODE)){
			System.out.println(reason.getReasonPhrase()+"로 인해 종료");
		}else{
			// 정상 종료
			System.out.println(session.getRequestURI()+" 연결 종료");
		}
	}
}
```

```java
/**
 * 경로인자(PathParameter)를 받아 처리할수 있는 Endpoint
 * websocket.echo.annotation.SampleEchoAnnotation
 */
@ServerEndpoint("/websocket/echoAnnotation/{sessionId}")
public class SampleEchoAnnotationHasParam extends AbstractEchoAnnotation{

	@OnMessage
	public void echoTextMessage(Session session, @PathParam("sessionId")
		String id, String message, boolean last){
		RemoteEndpoint.Basic remoteBasic = session.getBasicRemote();
		try {
			if(session.isOpen()){
				if(last){
					remoteBasic.sendText(message, false);
					remoteBasic.sendText(" from ["+id+",
						"+session.getId()+"]", true);
				}else{
					remoteBasic.sendText(message, last);
				}
			}
		} catch (IOException e) {
			e.printStackTrace();
			try {
				session.close(
				new CloseReason(CloseCodes.CLOSED_ABNORMALLY,
					e.getMessage()));
			} catch (IOException e1) {}
		}
	}

	@OnMessage
	public void echoBinaryMessage(Session session, @PathParam("sessionId")
		String id, ByteBuffer message, boolean last){
		RemoteEndpoint.Basic remoteBasic = session.getBasicRemote();
		try {
			remoteBasic.sendBinary(message, last);
			if(last){
				remoteBasic.sendText(" from ["+id+"]",  true);
			}
		} catch (IOException e) {
			e.printStackTrace();
			try {
				session.close(
					new CloseReason(CloseCodes.CLOSED_ABNORMALLY,
					e.getMessage()));
			} catch (IOException e1) {}
			}
	}
}
```

3) 각 방식으로 구현된 Endpoint를 WebSocketContainer에 Endpoint 를 등록하기 위한 ServerApplicationConfig 구현

```
public class SamplesCofig implements ServerApplicationConfig {
	/**
	 * websocket.SamplesConfig
	 * Annotation-driven 방식으로 구현된  ServerEndpoint를 찾고,
	 * 이를 통해 웹소켓 peer 를 완성할 수 있도록 {@link ServerEndpoint}를
	마커로 사용.
	 * @see
	javax.websocket.server.ServerApplicationConfig#getAnnotatedEndpointClass
	es(java.util.Set)
	 */
	@Override
	public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> arg0) {
		Set<Class<?>> result = new HashSet<Class<?>>();
		for(Class<?> clz : arg0){
			ServerEndpoint endpoint =
				clz.getAnnotation(ServerEndpoint.class);
			if(endpoint!=null){
				result.add(clz);
			}
		}
		return result;
	}
	/**
	 * 웹소켓 통신의 서버사이드를 담당하는 EndPoint 구현체를 스캐닝하고,
	 * 스캐닝된 EndPoint 집합으로 ServerEndPointConfig 집합을 빌드하는 메소드.
	 * 커스텀 어노테이션인 {@link MappingPath} 를 마커 어노테이션으로 사용함.
	 * @see
	javax.websocket.server.ServerApplicationConfig#getEndpointConfigs(java.u
	til.Set)
	 */
	@Override
	public Set<ServerEndpointConfig> getEndpointConfigs(
		Set<Class<? extends Endpoint>> arg0) {
		Set<ServerEndpointConfig> result =
				new HashSet<ServerEndpointConfig>();
		for(Class<? extends Endpoint> endpoint : arg0){
			MappingPath mPath =
				endpoint.getAnnotation(MappingPath.class);
			if(mPath!=null){
				String path = mPath.value();
				result.add(ServerEndpointConfig.Builder.
						create(endpoint, path).build());
			}
		}
		return result;
	}
}
```

2. 클라이언트 사이드 Endpoint 및 클라이언트 UI 구성 (WebSocket Client Programming 참고)

❑ WebSocket Server programming using Spring 4

(http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#websocket)

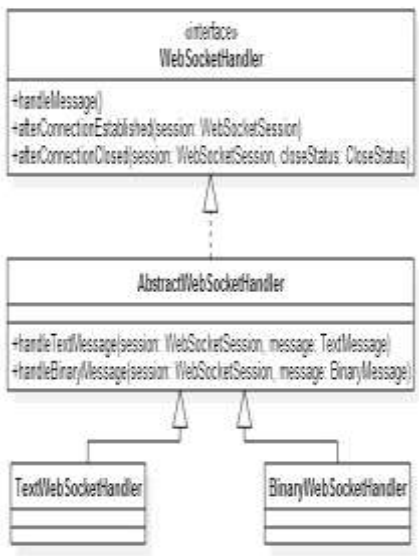
- Spring WebSocket API 사용 단계

1. 스프링 4에서 JSR-356 스펙에 준하여 작성된 Spring-Websocket과 Spring-webmvc 모듈 의존성 추가.

2. WebSocketHandler 의 구현

```
public class TextEchoWebSocketHandler extends TextWebSocketHandler {
	@Override // @OnError 참고
	public void handleTransportError(WebSocketSession session, Throwable exception)
		throws Exception {
	}
	@Override  // @OnMessage 참고
	protected void handleTextMessage(WebSocketSession session, TextMessage message)
		throws Exception {
		session.sendMessage(message);
	}
	@Override //  @OnOpen 참고
	public void afterConnectionEstablished(WebSocketSession session)
		throws Exception {
		InetSocketAddress clientAddr = session.getRemoteAddress();
		System.out.println(clientAddr + " 과의 websocket 연결 성공");
	}
	@Override // @OnClose 참고
	public void afterConnectionClosed(WebSocketSession session,CloseStatus status)
		throws Exception {
		InetSocketAddress clientAddr = session.getRemoteAddress();
		int code = status.getCode();
		String reason = status.getReason();
		System.out.println(clientAddr + " 과의 websocket 연결 종료, 종료코드 : "
		+ code + ", 종료사유 : " + reason);
	}
	@Override
	public boolean supportsPartialMessages() {
		// 리턴값이 true라면, 메시지를 분할 처리할 수 있음.
		return super.supportsPartialMessages();
	}
}
```

1) TextWebSocketHandler : 문자 메시지를 처리하기 위한 구현체.
2) BinaryWebSocketHandler : 바이너리메시지를 처리하기 위한 구현체.
3) 복합 메시지를 처리할때는 AbstractWebSocketHandler를 하위 구현함.
   ① afterConnectionEstablished : 웹소켓 핸드쉐이크 성공이후 호출.
   ② afterConnectionClosed : 웹소켓 연결 종료 후 호출.
   ③ handleTextMessage : 문자메시지 처리 핸들러.
   ④ handleBinaryMessage : 바이너리 메시지 처리 핸들러.
   ⑤ supportsPartialMessage : 전체 메시지를 분할 처리하는지 여부, true이면서 WebSocket container 가 부분 메시지를 지원하는 경우, 데이터가 크거나 미리 데이터의 크기를 알수없을때, handleMessage 메소드를 여러 차례 나눠서 호출함.
   ⑥ handleTransportError : 전송 에러 발생시 처리 핸들러.



3. 구현한 핸들러를 WebSocketEndpoint로 등록 : handshake URL 매핑.

- XML Configuration : <websocket:handlers >

```
<bean id="webSocketHandler"  class="kr.or.ddit.web.spring.TextEchoWebSocketHandler" />
<websocket:handlers>
     <websocket:mapping handler="webSocketHandler" path="/spring/textToEcho"/>
</websocket:handlers>

<mvc:default-servlet-handler />
```

- Java Configuration : WebSocketConfigurer 의 구현체에 @EnableWebSocket 를 사용.

```
@Configuration
@EnableWebMvc
@EnableWebSocket
public class WebAppConfig extends WebMvcConfigurerAdapter
                          implements WebSocketConfigurer{
       @Bean
       public TextEchoWebSocketHandler textEcho(){
              return new TextEchoWebSocketHandler();
       }

       @Override
       public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
              registry.addHandler(textEcho(), "/spring/textEcho");
       }

       @Override
       public void configureDefaultServletHandling(
              DefaultServletHandlerConfigurer configurer) {
              configurer.enable();
       }
}
```

- WebSocketHandler 로 엔드포인트를 등록하는 경우, 반드시 DispatcherServlet 매핑은 루트로 설정함.
  이때 ServletContainer 가 가진 default servlet으로 처리되지 않은 리퀘스트를 위임하기 위한 핸들러가 반드시 필요함.

4. DispatcherServlet 의 등록 및 WebApplicationContext의 생성

- XML Configuration : web.xml 을 통해 등록.

```
<servlet>
       <servlet-name>springDispatcherServlet</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
       <init-param>
              <param-name>contextConfigLocation</param-name>
              <param-value>classpath:webSocket-config.xml</param-value>
       </init-param>
       <load-on-startup>1</load-on-startup>
       <async-supported>true</async-supported> <!-- SockJS  구축을 위해 비동기 지원 필요 -->
</servlet>
<servlet-mapping>
       <servlet-name>springDispatcherServlet</servlet-name>
       <url-pattern>/</url-pattern>
</servlet-mapping>
```

- Java Configuration : WebApplicationInitializer 구현

```
public class CustomWebApplicationInitializer extends
                AbstractDispatcherServletInitializer{
        @Override
        protected WebApplicationContext createServletApplicationContext() {
                AnnotationConfigWebApplicationContext webContext =
                new AnnotationConfigWebApplicationContext();
                webContext.register(WebAppConfig.class);
                return webContext;
        }
        @Override
        protected String[] getServletMappings() {
                return new String[]{"/"};
        }
        @Override
        protected WebApplicationContext createRootApplicationContext() {
                return null;
        }
}
```

- Spring 의 웹소켓 핸들러로 엔드포인트를 구현하는 경우, 반드시 WebApplicationContext를 생성할 DispatcherServlet 이 모든 리퀘스트를 받을 수 있도록 선언해줘야 함.
  따라서 매핑 url 은 "/" 로 전체 매핑을 걸게 되고, 이렇게 모든 리퀘스트가 DispatcherServlet으로 집중되는 경우, 일반 웹리소스에 대한 요청처럼 별도의 핸들러를 구현하지 않는 요청들을 ServletContainer의 default servlet에게 위임할 수 있어야 함.

❖ SockJS 서버 구축

SockJS (https://github.com/sockjs/sockjs-client) : 웹소켓 API 지원여부가 브라우저 혹은 웹서버에 따라 상이한 관계로 다양한 우회기법들을 추상화하여 공통된 인터페이스를 정의한 라이브러리다. SockJS 은 http 프로토콜을 사용하기 때문에 웹소켓 프로토콜을 지원하지 않는 런타임 환경에서도 어플리케이션의 코드를 변경하지 않고, 양방향 통신이 가능하므로, 웹접근성을 고려한다면 아직 웹소켓에 비해 SockJS 의 활용성이 더 좋다 할 수 있다. SockJS 서버는 websocket, long-polling 등 SockJS 클라이언트가 사용하는 다양한 방식을 지원하는데, Node.js, Python, Java Vert.x 등이 SockJS를 지원하고 있으며, 이를 이용하면 단일 서버 API 로 SockJS 클라이언트와 양방향 통신하는 서버를 만들 수 있다. Spring 역시 WebSocket 모듈에 SockJS 서버를 구축할 수 있는 기능을 지원하는데, 웹소켓 핸들러 설정에 간단한 설정 추가로 쉽게 SockJS 서버를 구축할 수 있다.

1) Jackson2 라이브러리 의존성 추가 : SockJS의 메시지 기본 전송 방식이 JSON포맷을 사용하기 때문에, SockJS 서버를 구축하기 위해서는 먼저 Jackson2 라이브러리 의존성이 필요하다.
2) WebsocketHandler 구현
3) 구현한 핸들러를 websocket endpoint 로 등록 & SockJS 지원 설정 추가

```
<websocket:handlers>
     <websocket:mapping handler="webSocketHandler" path="/spring/textEcho"/>
     <websocket:sockjs />
</websocket:handlers>
```

```
        registry.addHandler(textEcho(), "/spring/textEcho").withSockJS();
```

4) 지금까지 예제에서 웹소켓이나 SockJS에 관한 설정을 DispatcherServlet으로 초기화되는 WebAppliationContext 에 설정해왔으나 Spring-WebSocket 모듈 자체가 SpringMVC 에 의존하는 것은 아니다 . Spring MVC 구조가 아닌 Http 서비스 어플리케이션에서 SockJS를 사용해야 하는 경우라면 SockJSHttpRequesHandler를 이용해 SockJS 와 어플리케이션간의 통합이 가능하다.
전이중 양방향 통신을 지원하는 다양한 기술들의 집합이라 할 수 있는 SockJS가 내부적으로 사용하는 통신 기술이 xhr-streaming 이나 Http long polling 방식이다라면 일반적인 웹요청 처리 사이클보다 커넥션을 더 길게 유지해야만 한다. 따라서 서블릿 컨테이너는 비동기 리퀘스트를 처리하기 위한 async 컨텍스트를 지원해야 하고, 이때 스프링 SockJS는 기본적으로 매 25초마다 하트비트 메시지를 전송하여 클라이언트의 연결이 종료되었는지를 확인하게 된다.

```
<servlet>
		<servlet-name>springDispatcherServlet</servlet-name>
		<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<init-param>
			<param-name>contextConfigLocation</param-name>
			<param-value>classpath:webSocket-config.xml</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
		<async-supported>true</async-supported>
</servlet>
<servlet-mapping>
		<servlet-name>springDispatcherServlet</servlet-name>
		<url-pattern>/</url-pattern>
</servlet-mapping>
```

```
public class CustomWebApplicationInitializer extends  AbstractAnnotationConfigDispatcherServletInitializer{
		@Override
		protected String[] getServletMappings() {
			DefaultServletHttpRequestHandler a;
			return new String[]{"/"};
		}
		@Override
		protected Class<?>[] getRootConfigClasses() {
			return null;
		}
		@Override
		protected Class<?>[] getServletConfigClasses() {
			return new Class<?>[]{WebSoConfig.class,};
		}
		@Override
		protected boolean isAsyncSupported() {
			return true;
		}
}
```

❖ **HttpSession 과 WebSocketSession 간의 데이터 공유(Handshaking 커스터마이징)**

웹 어플리케이션을 개발하는 과정에서 http 를 기본 사용하는 모듈에서 생성된 데이터를 웹소켓 모듈에서 사용해야 하거나 혹은 그 반대의 데이터 공유가 필요할 때가 있다. 이때 HandshakeInterceptor 가 사용된다. 예를 들어, HttpSession 내에 포함된 인증 정보를 websocket handler 에서 사용해야 하는 경우, 다음과 같은 인터셉터를 등록한다.

```
<websocket:handlers>
        <websocket:mapping handler="webSocketHandler" path="/spring/textEcho" />
        <websocket:handshake-interceptors>
            <bean class="org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"
            />
        </websocket:handshake-interceptors>
        <websocket:sockjs />
</websocket:handlers>
```

```
HttpSessionHandshakeInterceptor interceptor = new HttpSessionHandshakeInterceptor();
interceptor.setCopyAllAttributes(true);
interceptor.setCopyHttpSessionId(true);
registry.addHandler(textEcho(), "/spring/textEcho").addInterceptors(interceptor).withSockJS();
```

❖ **WebSocket 엔진 설정**

서블릿 컨테이너의 웹 소켓 엔진들은 message buffer size 나 idle timeout 등의 런타임 특성을 조절할 수 있는 configuration properties 를 가지고 있는데, 이는 tomcat, glassfish 등의 각 컨테이너에 맞는 ServletServerContainerFactoryBean 을 WebSocket config 에 추가하여 설정할 수 있다 .

```
<bean id="servletServerContainer"
        class="org.springframework.web.socket.server.standard.ServletServerContainerFactoryBean"
        p:maxSessionIdleTimeout="3000"
        p:maxBinaryMessageBufferSize="8192"
/>
```

```
@Bean
public TextEchoWebSocketHandler textEcho(){
        return new TextEchoWebSocketHandler();
}
@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        HttpSessionHandshakeInterceptor interceptor = new HttpSessionHandshakeInterceptor();
        interceptor.setCopyAllAttributes(true);
        interceptor.setCopyHttpSessionId(true);
        registry.addHandler(textEcho(), "/spring/textEcho").addInterceptors(interceptor).withSockJS();
}
@Bean
public ServletServerContainerFactoryBean createServletServerContainer(){
        ServletServerContainerFactoryBean factory = new ServletServerContainerFactoryBean();
        factory.setMaxSessionIdleTimeout(3000);
        factory.setMaxBinaryMessageBufferSize(8192);
        return factory;
}
```

❑ WebSocket Client Programming

- JavaScript in Browser

웹소켓은 ws 프로토콜에 기반하여 클라이언트와 서버 사이에 지속적인 전이중 연결 스트림을 유지할 수 있는 기술이다. 웹소켓 프로토콜 자체는 플랫폼에 독립적이나 가장 전형적인 웹소켓 클라이언트는 브라우저가 많이 사용된다. 자바스크립트 window 객체가 가진 WebSocket의 구조는 다음과 같다.

✓ **생성자의 종류**

```
WebSocket WebSocket(
  in DOMString url,
  in optional DOMString protocols
);

WebSocket WebSocket(
  in DOMString url,
  in optional DOMString[] protocols
);
// url : ws 프로토콜을 사용해 연결하고자 하는 서버의 경로
// protocols : 하나의 서버에 여러개의 서브 프로토콜을 지원하는 엔드포인트가 있는경우 사용.
// 예외정보
SECURITY_ERR : 연결에 사용되는 포트가 블럭상태인 경우
```

✓ **메소드의 종류**

```
// 웹 소켓 서버와의 연결을 종료
void close(
//(https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent#Status_codes)
  in optional unsigned short code, // 종료 코드
  in optional DOMString reason // 종료 사유
);
// 예외 정보
INVALID_ACCESS_ERR : 정의되지 않은 종료 코드 사용.
SYNTAX_ERR : 종료 사유가 지나치게 긴 경우등의 예외 사항.

// 다른 피어로의 데이터 전송
void send(
  in DOMString data
);

void send(
  in ArrayBuffer data
);

void send(
  in Blob data
);
// 예외 정보
INVALID_STATE_ERR : 메시지 송수신 상태(OPEN)가  아닌 경우
SYNTAX_ERR : 전송 불가능한 데이터를 사용하는 경우 문법에 맞지 않는 문자열 데이터 등..
```

✓ **속성 종류**

속성	타입	설명
binaryType	DOMString	전송데이터의 타입. (blob or arraybuffer or string)
bufferedAmount	unsigned long	전송 데이터 길이**(Read only)**
extensions	DOMString	서버에 의해 선택된 확장자. 비어있는 문자열 이거나 연결과 관련된 확장자를 가짐.
onclose	EventListener	CLOSED 상태가 됐을때 발생하는 close 이벤트 핸들러
onerror	EventListener	error 이벤트 핸들러
onmessage	EventListener	Message 를 수신한 상태에서 발생하는 message 이벤트 핸들러
onopen	EventListener	OPEN 상태가 됐을때 발생하는 open 이벤트 핸들러로 메시지 송수신 대기 상태를 의미함.
protocol	DOMString	생성자를 통해 받은 서브 프로토콜
readyState	unsigned short	현재 연결 상태를 의미하는 상태 속성 **(Read only)**
url	DOMString	생성자를 통해 받은 웹소켓 연결 대상 경로(url). **(Read only)**

✓ **상수의 종류 및 의미**

상수	값	의미
CONNECTING	0	아직 연결 수립 전 핸드쉐이크 시도 중
OPEN	1	연결 수립 완료, 데이터 전송 가능 상태
CLOSING	2	연결 종료 진행 중
CLOSED	3	연결 종료 상태로 재연결 불가

✓ **브라우저에서의 웹소켓 클라이언트 예(ws://echo.websocket.org 에코서버 사용)**

```
<html>
<head>
// SockJS 클라이언트를 사용하기 위한 코드.
<script src="https://cdn.jsdelivr.net/npm/sockjs-client@1/dist/sockjs.min.js"></script>
</head>
<body>
<input type="button" value="connect" id="connBtn" />
<input type="button" value="disconnect" id="disConnBtn" disabled/>
<input type="text" id="msgInput" disabled/>
<div id="msgArea"></div>
<script type="text/javascript">
	let logHandler = function(event){
		console.log(event);
	}
```

WebSocket API 를 사용하는 경우의 코드 조각

```
        let sock = null;
        let urlInput = document.getElementById("urlInput");
        let connBtn = document.getElementById("connBtn");
        let disConnBtn = document.getElementById("disConnBtn");
        let msgInput = document.getElementById("msgInput");
        let msgArea = document.getElementById("msgArea");
        let writeMessage = function(message){
        let pTag = document.createElement("p");
               pTag.innerHTML = message;
               msgArea.appendChild(pTag);
        }
        connBtn.onclick = function (event){
               if(window.WebSocket){
                      let host = location.host;
                      let protocol = location.protocol == "https:" ? "wss:" : "ws:";
                      let port = location.port;
                      let url = protocol+"//"+host+(port?":"+port:"")
                               +"${pageContext.request.contextPath}/spring/textEcho";
                      sock = new WebSocket(url);
               }else{
                      sock = new SockJS(
                             '${pageContext.request.contextPath}/spring/textEcho'
                      );
               }
               sock.onopen=function(event){
                      logHandler(event);
                      connBtn.disabled = true;
                      msgInput.disabled = disConnBtn.disabled = false;
               };
               sock.onclose=function(event){
                      logHandler(event);
                      msgInput.disabled = disConnBtn.disabled = true;
                      connBtn.disabled = false;
               };
               sock.onerror=logHandler;
               sock.onmessage=function(event){
                      let message = event.data;
                      writeMessage(message);
                      logHandler(event);
               };
        }
        disConnBtn.onclick = function(event){
               sock?.close?.();
        }
        msgInput.onchange = function(event){
               let message = msgInput.value;
               if(!message) return false;
               sock?.send?.(message);
               writeMessage(message);
               msgInput.value = "";
        }
        window.addEventListener("unload", function(){          disConnBtn.click();          });
</script>
</html>
```

▪ STOMP(Simple Text Oriented Message Protocol)

1. STOMP frame 구조

stomp 는 스크립트 언어를 위해 만들어진 문자 기반의 프로토콜로 주로 webSocket 과 같은 full duplex connection 프로토콜의 서브프로토콜로 사용되며, 다음과 같은 frame 구조를 갖는다.

```
COMMAND
header1 : value1
header2 : value2

Body^@
```

```
SUBSCRIBE
id:sub-1
destination:/app/biz

...
```

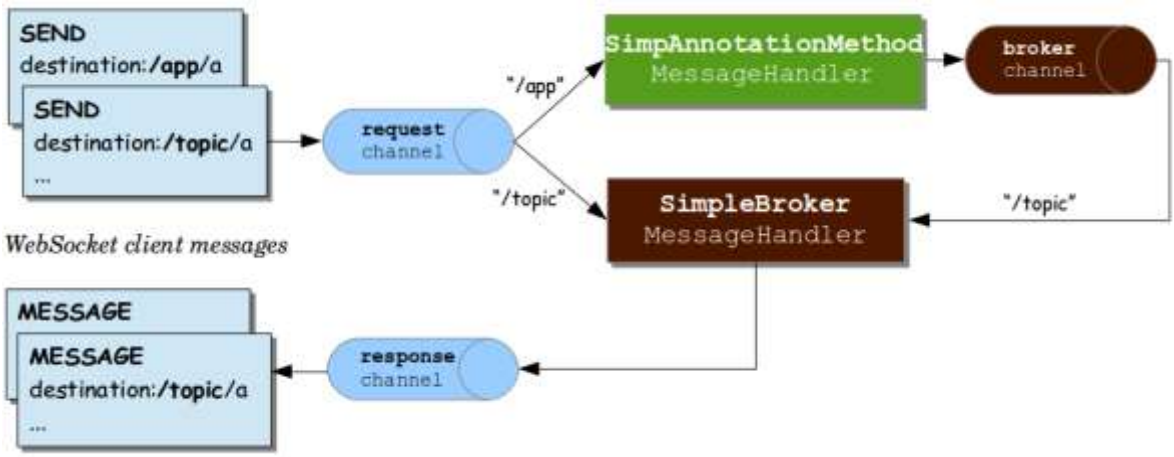
```
SEND[MESSAGE]
destination:/topic/biz
content-type:application/json

...
```

기본적으로 websocket 은 전송 메시지의 규격이 없기 때문에, 양 피어에서 메시지의 형식을 합의하고, 합의된 형식에 따라 메시지를 파싱하기 위한 과정이 수반된다. 반면, stomp 는 상위와 같이 프레임 구조가 명확하게 정의되어 있고, 발행/구독 형태의 메시지 전송 구조를 갖기 때문에 전송에 관련된 양 피어가 독립적으로 동작할 수 있는 구조를 갖는다.

1) Frame 종류 (https://stomp.github.io/stomp-specification-1.2.html#Frames_and_Headers)
   a. CONNECT : STOMP 클라이언트가 연결을 수립하고 스트림을 초기화할 때 사용.
   b. CONNECTED : 클라이언트의 연결 시도를 서버가 수락하면 사용되는 프레임.
   c. SUBSCRIBE : destination 을 대상으로 전송되는 메시지를 청취하기 위해 사용.
   d. UNSUBSCRIBE : destionation 을 대상으로 한 구독을 취소하기 위해 사용.
   e. SEND (body) : 클라이언트가 destination 을 대상으로 메시지를 전송하기 위해 사용.
   f. BEGIN (transaction): 트랜잭션의 시작을 의미하는 프레임
   g. COMMIT (transaction) : 트랜잭션 커밋 프레임
   h. ABORT (transaction) : 트랜잭션 롤백 프레임
   i. DISCONNECT : 연결 종료 프레임.
   j. MESSAGE (body) : 브로커에서 릴레이된 구독 메시지가 전송될때 사용되는 프레임.
   k. RECEIPT : 클라이언트가 보낸 프레임을 서버가 처리 완료하면, 전송되는 프레임.
   l. ERROR (body) : 에러 발생시 전송되는 프레임.

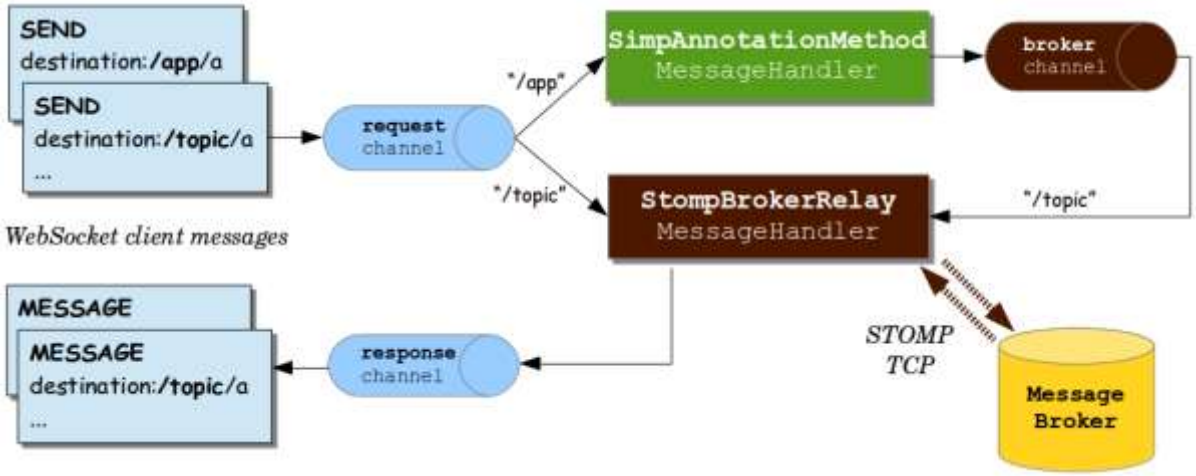
2. 메시지 중계 구조



스프링 빌트인 메시지 브로커를 사용하는 위 그림에는 다음과 같은 3개의 메시지 채널이 있다.

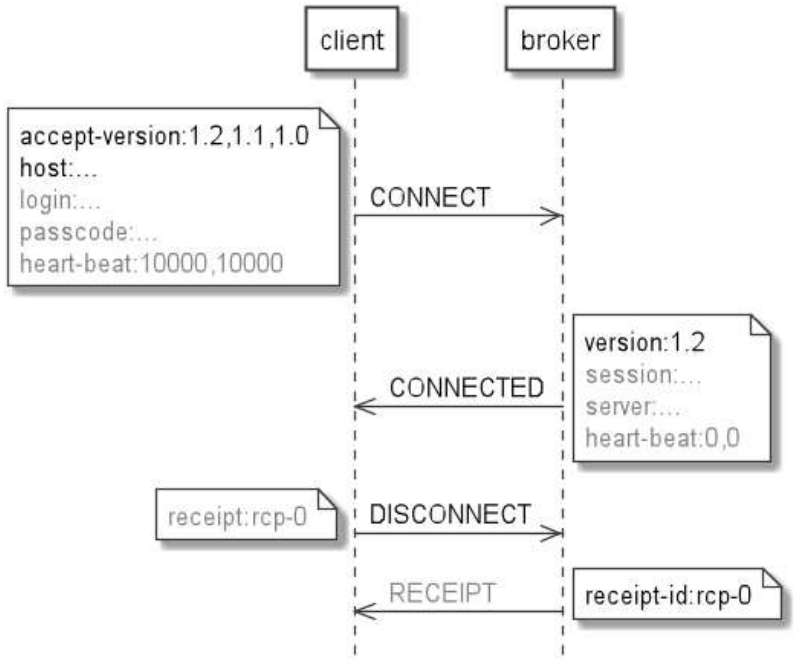
- clientInboundChannel : 웹소켓 클라이언트로부터 수신할 메시지 채널
- clientOutboundChannel : 웹소켓 클라이언트에게 메시지를 송신할 채널
- brokerChannel : 어플리케이션 내에서 메시지 브로커에게 메시지를 송신할 채널

외부의 STOMP 브로커에게 메시지를 전달하고, 브로커에서 구독자에게 메시지를 전달하는 구조로 "브로커 릴레이" 방식을 사용하기도 한다.



메시지를 발행하고, 구독하는 구조는 다음과 같은 단계를 거친다.

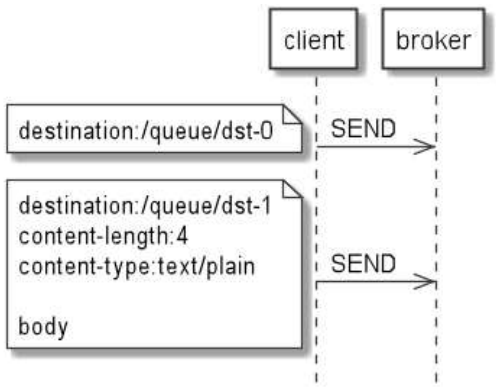
1) STOMP 메시지 브로커로 연결하기 위해 클라이언트는 CONNECT 프레임을 전송하고, 수락되면 브로커에서 클라이언트로 CONNECTED 프레임이 전송된다.



연결 종료를 위해 DISCONNECT 프레임이 전송되면, 서버에서는 이를 확인했다는 의미로 RECEIPT 프레임이 전송되고 소켓은 종료된다.

2) 클라이언트는 SUBSCRIBE 프레임으로 구독하고자 하는 메시지의 카테고리를 결정하는데, 이때 하나의 구독에 대해 id 가 부여되고, 구독 메시지의 카테고리는 destination 헤더로 표현한다.

3) 클라이언트는 SEND 프레임을 사용하여 메시지를 보내고, 서버에서는 MESSAGE 프레임으로 모든 구독자들에게 브로드캐스팅 할 수 있다. 이때 서버는 subscription 헤더를 통해 하나의 구독자를 표현하고, destination 헤더로 메시지의 카테고리를 표현한다.



**** destination : [topic/..] – (one to many publish-subscribe) , [queue/..] – (one to one pont-to-point)**

3. Spring-webSocket + STOMP echo 예제

1) spring-websocket, spring-messaging 모듈 의존성 추가

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-websocket</artifactId>
        <version>${org.springframework-version}</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-messaging</artifactId>
        <version>${org.springframework-version}</version>
</dependency>
```

2) Spring 컨텍스트에 STOMP 관련 설정 추가

```
<websocket:message-broker application-destination-prefix="/app" user-destination-prefix="/user">
        <websocket:stomp-endpoint path="/stomp/echo"  allowed-origins="*">
                <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:simple-broker prefix="/topic, /queue"/>
</websocket:message-broker>
```

① [/ stomp/echo] : Websocket 이나 SockJS client 를 사용시, 직접적으로 양방향 연결이 수립되는 주소.

② [/app/**] : 서버로 전송시 @MessageMapping 기반의 메시지 핸들러로 라우팅할 메시지 경로의 prefix

③ [/topic/**] : 클라이언트나 서버쪽에서 공통 구독자들에게 브로드캐스팅 메시지를 전송할때 사용할 주소

④ [/queue/**] : 1:1 구조로 private 메시지 전송시에 사용할 주소

3) Spring security 를 더한 보안 구조(optional).

① spring-security-messaging 모듈 의존성 추가

```
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-messaging</artifactId>
        <version>${org.springframework.security-version}</version>
</dependency>
```

```
<security:websocket-message-broker same-origin-disabled="true">
        <security:intercept-message type="CONNECT" access="isAuthenticated()"/>
        <security:intercept-message type="DISCONNECT" access="isAuthenticated()"/>
        <security:intercept-message type="SUBSCRIBE" access="isAuthenticated()"/>
        <security:intercept-message type="UNSUBSCRIBE" access="isAuthenticated()"/>
        <security:intercept-message type="MESSAGE" access="isAuthenticated()"/>
        <security:intercept-message pattern="/**" access="permitAll"/>
</security:websocket-message-broker>
```

상위 컨텍스트의 security 설정에 따라 변경 가능하며 메시지 브로커에 id 속성이 없는 경우 다음과 같은 전략들이 자동 등록된다..

a. SimpAnnotationMessageHandler 에 AuthenticationPrincipalArgumentResolver 가 등록됨.
b. clientInboundChannel 에 SecurityContextChannelInterceptor 가 등록됨.
c. clientInboundChannel 에 ChannelSecurityInterceptor 가 등록됨.

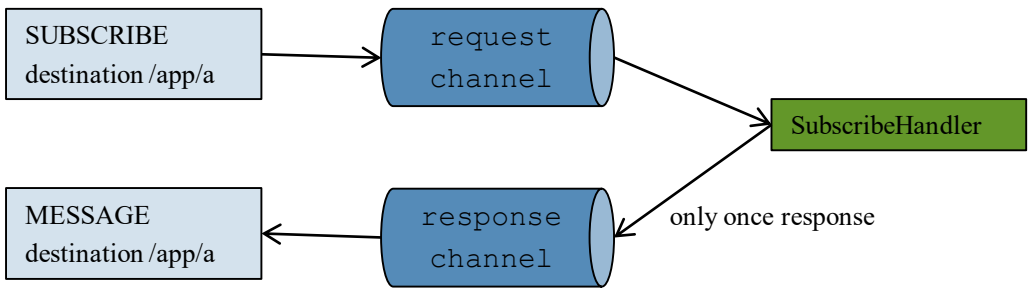
4) 서버사이드 메시지 핸들러 등록

```
@Slf4j
@RestController
public class EchoMessageHandler {
	@Data
	public static class MessageVO{
		private String sender;
		private String message;
	}

	@MessageMapping("/handledEcho")
	@SendTo("/topic/echoed")
	public MessageVO handler(@Payload MessageVO messageVO, @Header String id) {
		log.info("id header : {}", id);
		log.info("sender : {}, message : {}", messageVO.getSender(), messageVO.getMessage());
		return messageVO;
		//messagingTemplate.convertAndSend("/topic/echoed", messageVO);
	}

	// destination 이 /app/handledEcho 인 구독 요청에 대해 동작하며,
	//  한번의 요청에 한번의 응답만을 처리하게 됨.
	@SubscribeMapping("/handledEcho")
	public String subscribeHandler(
		@Headers Map<String, Object> headers
	) {
		log.info("headers : {}", headers);
		// subscription id 를 생성함.
		String sub_id = UUID.randomUUID().toString();
		return sub_id;
	}
}
```

clientInboundChannel 의 메시지는 어노테이션으로 매핑된 메시지 핸들러로 수신되거나 메시지 브로커에게 포워딩된다. @SendTo 어노테이션으로 메시지 핸들러 리턴객체의 destination 을 결정하여 brokerChannel에 전달하면 해당 채널에서 구독자에게 메시지가 전송된다. 이 어노테이션은 메시지 핸들러에서만 동작하기 때문에, 어플리케이션내 다른 위치에서 메시지를 브로드캐스팅할 때는 messagingTemplate 객체를 활용할 수 있다(주석 부분 참고). 또한, @SendTo 나 messagingTemplate 를 이용한 명시적인 destination 이 없는 경우, 메시지 핸들러의 매핑 url 로 메시지를 포워딩하게 되며, 이때 destination 은 매핑 url 에 브로커 prefix 가 붙은 형태가 된다. ex) /topic/handledEcho



@SubscribeMapping 은 클라이언트의 구독 요청을 접수하고, 양방향 통신시 초기 셋팅이 필요한 경우 활용된다. 마치 HTTP 의 요청 처리 핸들러처럼 동작하기 때문에, 한번의 구독 요청이 발생하면, 일정 부분 처리 후 @SendTO 로 새로운 destination 이 없는 한, 다시 메시지 송신자에게 결과를 한번 응답하게 된다. **단, 브로커를 거치지않고 어플리케이션으로 전송되는 구독인 경우만 수신하므로**, destination 이 /app 으로 시작하는 경우만 구독을 수신할 수 있다. 일반적으로 양방향 통신을 위한 UI 가 구성될때 초기 설정을 맞추기 위한 초기화 모듈로 활용할 수 있으며, 상기 예제에서는 차후 브로드캐스팅 되는 에코 메시지를 구독하기 위한 구독 아이디를 서버사이드에서 생성하여 다시 **/app/handledEcho 의 구독자에게 1회 MESSAGE frame 으로 응답하고** 있다.

5) 클라이언트 사이드 echo view

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="UTF-8">
	<title>Insert title here</title>
	<script src="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.5.0/sockjs.min.js"></script>
	<script src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js"></script>
	<style type="text/css">
	.my {
		background-color: yellow;
	}
	</style>
</head>
<body onload="init(event);">
	<input type="text" id="message" onchange="messageSend(event);"/>
	<input type="button" value="종료" onclick="disconnect(event);">
	<div id="messagesArea“></div>
	<script type="text/javascript">
		let client = null;
		let headers = {}
		let messageArea = document.querySelector("#messagesArea");
		let SUB_ID = null;
		function init(event){
			//  stomp-endpoint로 양방향  통신 연결 수립
			  var sockJS =
				new SockJS("${pageContext.request.contextPath}/stomp/echo");
			  // sockJS 연결 기반하에 Stomp client 객체 생성
			  client  = Stomp.over(sockJS);
			  // Stomp CONNECT frame 전송
			  client.connect(headers, function(connectFrame){
				// CONNECTED frame 을 받은 후,
				// echo 메시지 프레임을 수신을 위한
				// SUBSCRIBE frame에서 사용할  구독 아이디를 생성하기 위해
				// 구독 요청 핸들러 쪽으로 전송되는 SUBSCRIBE frame
				// 단 한번의 응답만을 수신함.
				client.subscribe("/app/handledEcho", function(messageFrame){
					SUB_ID = messageFrame.body;
					headers.id=SUB_ID;
```

5) 클라이언트 사이드 echo view (계속)

```
                                        //  Simple Message Broker 로 부터 브로드캐스팅 되는
                                        // 에코 메시지를 구독하기 위한 SUBSCRIBE frame 전송
                                        client.subscribe("/topic/echoed", function(messageFrame){
                                                let body = JSON.parse(messageFrame.body);
                                                let msgTag = document.createElement("p");
                                                if(body.sender==SUB_ID)
                                                        msgTag.classList.add("my");
                                                msgTag.innerHTML=
                                                        body.message+"["+body.sender+"]";
                                                messageArea.appendChild(msgTag);
                                        }, {id:SUB_ID});
                                });
                                let msgTag = document.createElement("p");
                                msgTag.innerHTML="연결수립";
                                messageArea.appendChild(msgTag);
                        }, function(error){
                                console.log(error);
                                alert(error.headers.message);
                        });
                }

                function messageSend(event){
                        if(! client || ! client.connected) throw "stomp 연결 수립 전";
                        let body = {
                                sender : SUB_ID
                                , message : event.target.value
                        }
                        // 서버사이드의 메시지 처리 없이 에코되는 메시지 전송
                        // client.send("/topic/echoed", headers, JSON.stringify(body));
                        // 서버사이드의 메시지 핸들러에서 처리될 메시지 전송
                        client.send("/app/handledEcho", headers, JSON.stringify(body));
                        event.target.value = "";
                        event.target.focus();
                }
                function disconnect(event){
                        if(! client || ! client.connected) throw "stomp 연결 수립 전";
                        client.disconnect();
                        let msgTag = document.createElement("p");
                        msgTag.innerHTML="연결종료";
                        messageArea.appendChild(msgTag);
                }
                window.addEventListener("unload",disconnect);
        </script>
</body>
</html>
```

❑ One To One (DM) 송수신

- Message broker 등록시 설정한 user-destination-prefix(default: user) 에 따라 direct 메시지를 수신하기 위해 ex) '/user/queue/DM' 과 같은 형식의 주소로 구독하며, 이 메시지는 UserDestinationMessageHandler 에 의해 destination 이 /queue/DM-user{sessionID} 와 같은 형태로 변환되어 브로커에 등록된다.
- SEND 프레임의 destination 은 '/user/{username}/queue/DM' 으로 발송하면, broker relay 로 라우팅되어 바로 수신자에게 전송된다.
- "/user/*" 패턴의 destination 을 통해 사용자를 식별하기 위해서는 Principal 기반 인증 객체의 형태로 사용자를 식별할 수 있어야 함. Ex) Filter 및 RequestWrapper 의 활용 권장(GeneratePrincipalFilter).

```
@Inject
private SimpMessagingTemplate  messagingTemplate;

@MessageMapping("/myProcess")
@SendToUser("/queue/myProcessResult") // 접속 유저를 대상으로 한 push 메시지,
public String handlerProcessResult(@Payload String message) {
	log.info("; message : {}", message);
	return message
}
@MessageMapping("/DM")
public void handlerDM(@Payload MessageVO body) {
	log.info("DM message : {}", body);
	messagingTemplate.convertAndSendToUser(body.getReceiver(), "/queue/DM", body);
}
```

```
// 수신 구조
client.subscribe("/user/queue/myProcessResult", function(messageFrame) {
	let message = messageFrame.body;
	alert(message);
});
client.subscribe("/user/queue/DM", function(messageFrame) {
	let message = messageFrame.body;
	alert(message);
});
```

```
// 송신 구조
// 1. 서버측 메시지핸들러로 라우팅 한 형태의 메시지 송신
client.send("/app/DM", {},
	JSON.stringify({
		username : “a001”,
		message : “a001 hello”
	 })
);
// 2. 핸들러로 라우팅하지 않고, broker relay 로 라우팅 한 형태의 메시지 송신
client.send("/user/a001/queue/DM",{},
	JSON.stringify({
		username : “a001”,
		message : “a001 hello”
	})
);
```