EchoHandler & ReactJS 접목하기

소프트웨어공학과 / 201332001 강인성 / hogu9401@gmail.com

0. 추가 배경 지식

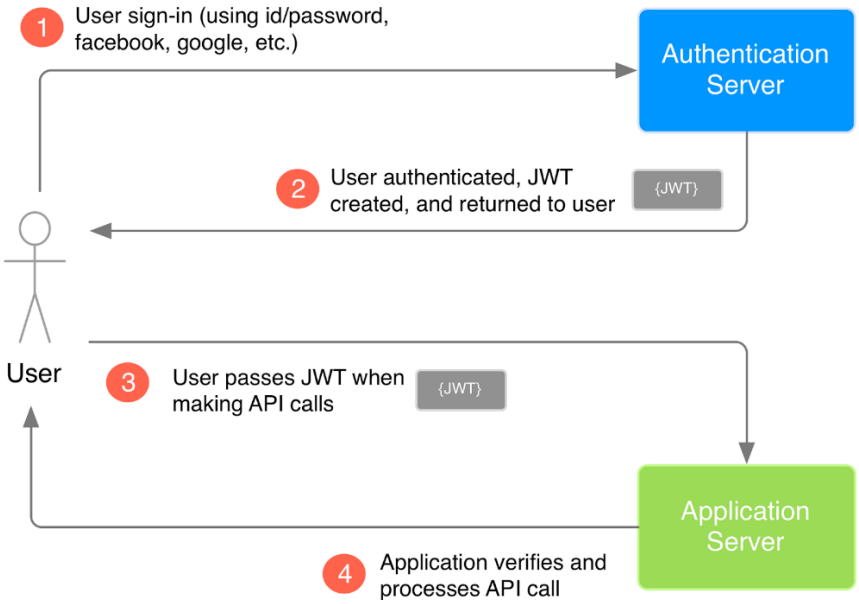
A. EchoHandler 클래스는?

지난 시간에는 STOMP를 기반으로 하는 WebSocket 채팅 프로그램을 구현했다. 그렇지만 STOMP 기반으로 작성하기 위해서는 약간의 자바스크립트 지식을 가지고 있어야 하고, 또한 MessageMapping에 대해서 각각 설정을 해 줘야 하기 때문에 이에 대해서는 대부분 채팅이나 실시간 동영상 등에서 주로 쓰는 기술로 인식을 할 수 있다. 그렇지만 EchoHandler는 STOMP를 이용하지 않고 간단한 Message를 통해서 각 사용자에게 요구하는 기능들에 대해서 바로 응답을 할 수 있도록 하기 위해서 쓰는 기술로 인식을 할 수 있다. 특히 EchoHandler는 WebSocket을 쓰는데 있어서 아주 간단한 알림 창, 각 사용자 별로 보내지는 실시간 정보(예를 들어 지하철 도착 정보, 버스 도착 정보) 등에 적용하는 것이 매우 좋은 방안이다.

지난 주에 언급을 안 하고 넘어갔지만, 본래 Java에서 제공하는 WebSocket의 표준은 SockJS, SocketIO가 절대로 아니라는 사실이다. 실제로는 JSR-356 이라는 Java의 WebSocket의 표준이 존재하는데 실제로 이를 기반으로 구현을 하는 방법은 난해하기 때문에 이의 구현체(예를 들어 상속 클래스, 인터페이스 etc.)를 설정해서 의존적으로 구현을 가능하게 한다는 점을 기억해 두면 좋겠다.

B. JWT Token 소개

JSON Web Token이라고 불린다. 이는 Web 표준을 기반으로 하여 각 서버와 클라이언트에서 JSON 객체를 사용하여 가볍고 자가 수용적으로 정보를 보낼 수 있는 토큰으로 볼 수 있다. JWT는 C언어, Java, Python 등등의 여러 언어에서도 JSON으로 변환을 할 수 있다면 지원을 해 준다. 그리고 JWT Token에는 크게 Header(헤더), Payload(내용), Signature(서명) 3가지 요소로 구성이 되어 있다. 여기서 Header에는 어떠한 알고리즘으로 암호화가 되었는가에 대한 설명(대부분 HS256, HS512 등을 쓴다.)을 쓰고, Payload(내용)에는 사용자에 대한 간략한 정보(예를 들어 이름, 닉네임, 아이디 등이 된다.), 로그인 시간, 유효 시간 등을 작성을 하고, Signature(서명)에는 각 서버 별로 Header, Payload 등의 내용을 암호화 하는 알고리즘을 활용해서 각기 다른 서버 별로 Secret Key를 이용해서 해시 값으로 버물어서 로그인을 한 사람에게 보내주는 입장권과 같은 역할로 보면 된다. Spring Security와 비교를 해 보면 JWT에서는 인증을 하는 방법을 사용자와 서버끼리 토큰을 이용해서 확인을 하는 방면 Spring Security에서는 접근한 사용자 목록에서 정당성을 주로 확인을 하는 방법으로 이용이 된다.



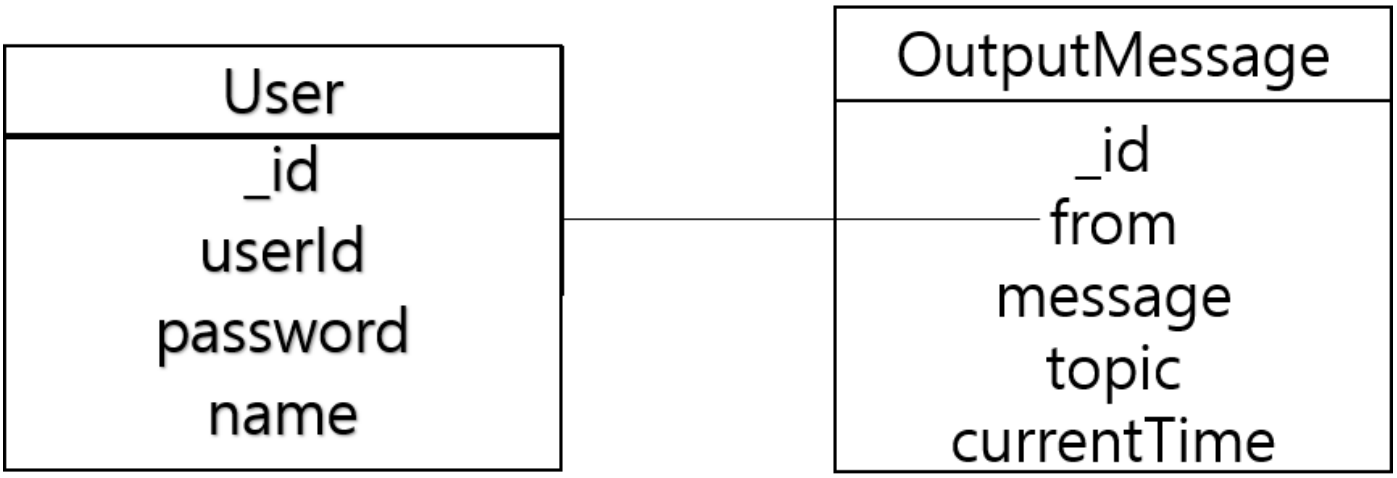
[그림] JWT Token이 인증 서버에서 어플리케이션 서버까지 오는 과정에 대한 Use Case Diagram.

C. Redux 소개

ReactJS는 Facebook에서 구상을 한 Singled Page Application 기반 Library이다. ReactJS에 대한 개념은 여기서 다루지 않고 공식 문서를 참고하면 좋겠다. (그렇지만 추후에 본인이 여름 방학 때 VueJS도 같이 공부하면서 ReactJS와 Redux, VueJS 개념 관련 GitHub에 등재하여 보기 좋게 정리할 예정이 있고, 또한 ReactJS도 새로 추가된 기능들이 있어서 이에 대해서도 반영을 할 것이다.) 그렇지만 React에서는 Component 내에서 놀고 있는 props와 state 두 개의 변수가 있는데 실제로 Component 내부에서는 state 혼자서 값들을 일일이 체크하고 다시 입히는 면에서는 비효율적이고 어플리케이션이 커지는 경우에 state는 값들이 잘 있는지에 대한 보장을 하는가에 대해서 많은 어려움을 가질 수가 있다. 그래서 state를 한 Application 내부에 있는 곳으로 끌어 당겨 쓰는 개념인 Redux를 적용을 해 볼 필요가 있다. Redux는 크게 action, reducer, container 3가지로 나뉜다. Action은 각 데이터들에 대한 움직임을 설정하고, 이를 통해 AJAX 통신으로 데이터를 얻어오거나 WebSocket 등을 이용해서 데이터를 얻어오는 과정 등을 작성을 하고 Action 타입을 설정을 반환하여 이의 유효성을 Reducer에서 다시 Action 타입 별로 재확인을 한다면 React Application에 홀로 떠 있는 큰 개념의 state에 저장을 시켜준다. 그리고 Container에서는 mapStateToProps, mapDispatchToProps 두 가지 함수를 이용해서 우리가 작성한 React Component를 불러와서 Reducer에서 가져온 state와 Action에서 가져온 함수들을 묶어서 React Application에서 Component 대신에 Container를 반영해준다면 본인이 Component에서 불러오는 Action 타입과 Reducer의 state를 통해서 React Application을 효율적으로 테스팅하면서 관리를 할 수 있다. Redux는 또한 ReactJS 뿐만 아니라 VueJS, Angular 등에서도 적용을 할 수 있도록 개선을 하는 중이고, Flux 메커니즘을 기반으로 Singled Page Application에 현존하는 Component들의 상태를 관리를 해 주는 점에서 공부를 해 보면 매우 좋겠다. 또한 한국어 판으로 Redux 공식 문서도 추가가 되어 있으니 JavaScript Singled Page Application이나 Native Application을 개발 할 의향이 있다면 한 번쯤 읽어 보길 권장한다.

D. MongoDB 데이터베이스 구성

MongoDB 데이터베이스는 지난 시간에 작성한 Schema에서 User만 살짝 변경이 있다. 데이터베이스 이름은 mesenger_example_react로 영문 오타 관련에 대해서는 양해를 구한다.



[그림] mesenger_example_react 데이터베이스 구성.

1. JWT Token 클래스 작성

A. JwtFilter.java

```
package net.kang.config;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.filter.GenericFilterBean;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureException;

public class JwtFilter extends GenericFilterBean{ // JWT에 대해서 Filter를 철저하게 할 수 있도록 하기 위해 Bean을 형성함.
	@Override
	public void doFilter(final ServletRequest req, final ServletResponse res, final FilterChain chain) throws IOException, ServletException{
		final HttpServletRequest request=(HttpServletRequest) req;
		final HttpServletResponse response=(HttpServletResponse) res;
		final String authHeader=request.getHeader("authorization");
		// JWT 토큰을 받아올 때 확인하기 위한 함수
		if("OPTIONS".equals(request.getMethod())) { // OPTIONS 관련 RequestMethod를 확인한다.
			response.setStatus(HttpServletResponse.SC_OK);
			chain.doFilter(req, res);
		}else {
	// JWT 토큰은 Bearer 뒤에 붙어야 확인이 가능한데 불가능하면 ServletException을 통해 예외처리한다.
			if(authHeader==null || !authHeader.startsWith("Bearer ")) {
				throw new ServletException("Missing or invalid Authorization header");
			}
			final String token = authHeader.substring(7); // Bearer를 없애기 위한 문장.
			try {
	// secretKey에 대해 확인을 진행하고 난 후에 REST API에서도 보안이 확인됨을 진행한다.
				final Claims claims = Jwts.parser().setSigningKey("secretkey").parseClaimsJws(token).getBody();
				request.setAttribute("claims", claims);
			} catch (final SignatureException e) {
	// JWT 예외 중 대표적으로 제한 시간이 지난다면 ServletException을 통해 예외처리한다.
				throw new ServletException("Invalid token");
			}
			chain.doFilter(req, res);
		}
	}
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > config > JwtFilter.java]

doFilter(final ServletRequest req, final ServletResponse res, final FilterChain chain)

doFilter 매개 변수 내부에 들어 있는 변수들이 전부 final로 구성이 되어 있는데 ServletRequest, ServletResponse, FilterChain 변수들에 대해서 doFilter 함수 내부에서는 HttpRequest, HttpResponse에서 받아온 Servlet 요청들에 대해서 오로지 상수로만 받아서 적용하기 위해서 쓰는 것을 이해할 수 있는데 즉 함수 내부에서 변수에 대해 변경을 하지 않고 Read Only를 위한 개념으로 인지를 해야 한다. 이 함수는 JWT Token에 대해서 session에 저장된 토큰에 대해 받아오고 난 이후에 HTTP에서 Header에 추가를 하고 난 후에 유효성을 판단해주는 역할로 보면 된다. JWT Token에 대해 기본적으로 요청을 해 주는 방안은 아래와 같다. 대부분 로그인 요청은 OPTIONS method를 통해 진행을 하여 확인 절차를 밟고 토큰의 유효성을 확인하고 난 뒤에 인증 토큰에서 다시 반환을 해서 사용자가 이용을 하게끔 끝내 주도록 하기 위해서는 이 함수를 필수로 작성을 하고 넘어가야 한다.

Header Key	Header Context
Authentication	Bearer (JWT Token Context)

B. CorsConfig.java

```
package net.kang.config;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class CorsConfig { // JWT에 대한 RequestMapping에서 이에 대한 보안을 향상하기 위해서 Http Cross Origin 기능을 향상하기 위해 CORS에 대해서 설정하는 Configuration 클래스.
	@Bean
	public FilterRegistrationBean corsFilter() {
		UrlBasedCorsConfigurationSource source=new UrlBasedCorsConfigurationSource();
		CorsConfiguration config=new CorsConfiguration();
		config.setAllowCredentials(true);
		config.addAllowedOrigin("*");
		config.addAllowedHeader("*");
		config.addAllowedMethod("OPTIONS");
		config.addAllowedMethod("HEAD");
		config.addAllowedMethod("GET");
		config.addAllowedMethod("PUT");
		config.addAllowedMethod("POST");
		config.addAllowedMethod("DELETE");
		config.addAllowedMethod("PATCH");
		source.registerCorsConfiguration("/**", config);
		final FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
		bean.setOrder(0);
		return bean;
	}
	@Bean
	public WebMvcConfigurer mvcConfigurer() { // MVC에서 GET, PUT, POST, DELETE, OPTIONS의 Method를 실행이 가능 할 수 있도록 설정하였음.
		return new WebMvcConfigurerAdapter() {
			@Override
```

```
                        public void addCorsMappings(CorsRegistry registry) {
                                registry.addMapping("/**").allowedMethods("GET", "PUT", "POST",
"DELETE", "OPTIONS");
                        }
                };
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > config > CorsFilter.java]

JWT Token에서 이용할 수 있는 Request의 Method들의 목록에 대해서 설정하는 함수로 볼 수 있다. 여기서 FilterRegistrationBean을 통해 설정한 CrossOrigin에 대해 적용 하기 위해서는 Application 클래스에서 corsFilter() 함수에서 작성한 Bean 객체를 통해서 JWT Token에서 인증을 받은 자에게 한해서 REST API 접속을 허용을 할 수 있다. 대부분 Request Method에 대해서는 GET, PUT, POST, DELETE, OPTIONS 5가지 중 하나를 쓴다. 그리고 하단부에 mvcConfigurer() 함수에 대해서는 Spring Web MVC(Controller만 작성하는 REST API로 보낼 필요 없이 Spring을 기반으로 한 JSP 환경에서 Model, View, Controller를 통해 설정함.)에서도 적용할 수 있도록 작성한 문장으로 이해하면 된다.

C. Example032Application.java

```
package net.kang;
import javax.sql.DataSource;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import net.kang.config.JwtFilter;

@SpringBootApplication
public class Example032Application {

        public static void main(String[] args) {
                SpringApplication.run(Example032Application.class, args);
        }

        @Bean
        public FilterRegistrationBean jwtFilter() {
                final FilterRegistrationBean registrationBean=new FilterRegistrationBean();
                registrationBean.setFilter(new JwtFilter());
                registrationBean.addUrlPatterns("/user/*");
                return registrationBean;
        }

        @Bean
        @ConfigurationProperties(prefix = "spring.data.db-main")
        public DataSource mainDataSource() {
                return DataSourceBuilder.create().build();
        }

        @Bean
        @ConfigurationProperties(prefix = "spring.data.db-log")
        public DataSource contractDataSource() {
                return DataSourceBuilder.create().build();
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > Example032Application.java]

FilterRegistrationBean jwtFilter()

이 Bean 객체에 대해서는 Spring Server Application에 방금 전에 작성한 JwtFilter 클래스를 작성을 하고 난 뒤에 토큰의 유효성을 참조하는 Bean 객체를 가져오고 난 이후에 이를 적용을 하도록 도와 주는 함수로 이해하면 된다. 여기서 URL Pattern에서는 http://localhost:8080/example03_2/ 에서 뒤에 붙는 Request 주소의 Prefix가 user로 시작을 한다면 user 뒤에 모든 붙는 데이터들에 대해서는 JWT Token에 대한 인증을 진행을 하고 난 이후에 적용을 할 수 있도록 하기 위해 설정을 한다고 이해 하면 된다.

2. EchoHandler 클래스 작성

A. EchoHandler.java

```
package net.kang.handler;

import java.util.ArrayList;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.CloseStatus;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import net.kang.domain.OutputMessage;
import net.kang.model.Message;
import net.kang.service.OutputMessageService;

@Component // 이를 @Autowired를 할 수 있도록 하기 위해 Component 어노테이션으로 추가한다.
public class EchoHandler extends TextWebSocketHandler{
	private static Logger logger = LoggerFactory.getLogger(EchoHandler.class);
	// Logger에 대해서도 활용을 연습하기 위해 주석 처리를 이용해서 설정을 해 뒀다.
	private List<WebSocketSession> sessions = new ArrayList<WebSocketSession>();
	// 현재 Messenger에 접속된 Session 목록들을 저장한다.
	@Autowired OutputMessageService outputMessageService;
	// EchoHandler에서 MongoDB와 접목하기 위해 Service 클래스 생성

	private String jsonStringFromObject(Object object) throws JsonProcessingException {
	// 이는 Java에서 쓰인 모든 객체들에 대해서 String JSON으로 반환한다.
		ObjectMapper mapper = new ObjectMapper();
		return mapper.writerWithDefaultPrettyPrinter().writeValueAsString(object);
	}

	@Override
	public void afterConnectionEstablished(WebSocketSession session) throws Exception{
	// WebSocket에 접속한 client session에 대해 등록을 한다.
		sessions.add(session);
		System.out.println(" => connected by session : "+session.getId());
		// logger.info("disconnected by session : {0}", session.getId());
	}

	@Override
	public void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception{
 // handleTextMessage 함수에서는 사용자가 보낸 message에 대해서 받아와서 데이터베이스에 저장을 할 수 있는 문장을 작성하였다.
		System.out.println(" => "+session.getId()+" send message -> "+message.getPayload());
		// logger.info("{0} send message -> {1}", session.getId(), message.getPayload());
```

```
                boolean isWrited=false; // 이는 사용자가 채팅 내용을 입력하는 경우에 이를 true로 설정을 하여 모든
사람들에게 새로운 인원이 입력한 채팅 내용을 보내준다.
                for(WebSocketSession ws : sessions) {
                // 각 WebSocket에 접속한 session들에 대해서 각각 순회를 하여 Message를 확인하도록 한다.
                        String messageResult = message.getPayload();
                        if(messageResult.equals("chattingList")) {
                                if(session.getId().equals(ws.getId())) {
                                // 새로고침은 자신만 할 수 있도록 설정을 하였다.
                                        String messageJSON =
jsonStringFromObject(outputMessageService.findAll());
                                        ws.sendMessage(new TextMessage(String.format("%s", messageJSON)));
                                 // 현재 Database에 있는 메신저 목록들을 불러온다.
                                }
                        } else if(messageResult.contains("addMessage")) {
                        // 메시지를 전송하는 사람의 시점을 고려한다.
                                if(session.getId().equals(ws.getId())) {
                        // 메시지를 전송하는 사람이 서로 같은 경우에는...
                                        String[] messageToken = messageResult.split("\n");
                // 메시지에서는 각 요소들에 대해 임시적으로 개행 문자를 통해 구분을 하도록 설정을 하여 각 요소들을 추가한다.
                                        Message msg = new Message();
                                        msg.setText(messageToken[1]);
                                        msg.setFrom(messageToken[2]);
                                        outputMessageService.insert(msg, messageToken[3]);
                                        // 메시지를 1개만 추가를 하도록 형성한다.
                                        isWrited=true;
                                }
                                if(isWrited) break;
                        }
                }
                if(isWrited) {
                // 현재 Session 목록들에서 사용자가 글을 추가한 경우에 각 Message에 대해 JSON으로 반환을 해 준다.
                        for(WebSocketSession ws2 : sessions) {
                                List<OutputMessage> tmpList=new ArrayList<OutputMessage>();
                                tmpList.add(outputMessageService.findTopByOrderByCurrentTimeDesc());
                                String messageJSON = jsonStringFromObject(tmpList);
                                ws2.sendMessage(new TextMessage(String.format("%s", messageJSON)));
                        }
                }
        }

        @Override
        public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception{
        // 각 사용자들 중에서 현재 채팅을 종료하거나 로그아웃을 한다면 이를 선언하여 session 목록에서 제거한다.
                sessions.remove(session);
                System.out.println(" => disconnected by session : "+session.getId());
                // logger.info("disconnected by session : {0}", session.getId());
        }

}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > handler > EchoHandler.java]

```
@Autowired OutputMessageService outputMessageService;
```

지난 시간에 작성한 UserService 클래스와 새로 추가된 Service 클래스에 대해서 일부분 수정 내용이 있다. 이에 대해서는 2-C에서 다시 언급을 하겠다. 물론 Controller 클래스에 대해서도 일부 수정 내용이 있어서 이는 2-D에서 설명을 하겠다.

afterConnectionEstablished, handleTextMessage, afterConnectionClosed 함수

EchoHandler 클래스를 작성하기 위해서는 TextWebSocketHandler Interface를 상속해서 이 함수들에 대한 내용을 꼭 작성을 하고 이용을 하는 것이 상책이다. 각 함수는 Client에서 WebSocket에 접속을 하고 난 후에 connection이 정상적으로 작성된 이후에 실행되는 함수, 사용자가 Client에서 WebSocket Message 요청에 대해 요구할 때 실행되는 함수, 사용자가 Client에서 로그아웃을 하거나 종료를 하는 경우에 실행되는 함수 3가지로 나뉠 수 있다. 이 함수에 대해서는 주석으로 추가를 해서 이해하기 쉽게 작성을 했는데 각 함수 별로 어떤 걸 실행하는지에 대해서 작성을 할 필요가 있다.

➔ afterConnectionEstablished 함수

여기서 사용자가 WebSocket에 접속을 했다는 걸 Session ID를 통해 알리고 난 후에 현재 WebSocketSession에 추가를 해 준다. 여기서 접속된 사용자 목록에 대해서는 Session들의 목록을 통해 알 수 있다. Session은 브라우저 창에서 접속을 하는 경우에 임의의 ID를 보내서 설정을 해 주기 때문에 이에 대한 내부적인 구현을 할 필요까지는 없다.

➔ handleTextMessage 함수

여기서는 각 사용자들에 대해서 WebSocket에서 요청을 하는 메시징에 대해서 실시간으로 보내고 난 후에 이를 받아온 값들에 대해서 현재 데이터베이스에 접속이 가능한 Service 클래스에 대해서 실행을 해 주는 역할을 한다. handleTextMessage 함수에서는 각 Session 별로 사용자들이 보내는 Message는 부지기수 하게 많다. 이에 대해서 적용을 하기 위해서는 각 메시지 별로 어떤 역할을 하는지에 대해 확실하게 설정을 하는 것을 꼭 권장을 하고, 이에 대한 개념을 제대로 이해하기 위해서는 과거에 운영체제론 에서 공부했던 상호 배제와 임계 구역에 대한 개념(WebSocket의 병렬성을 더욱 보장을 하는 방안을 보장하기 위해서)을 복습해 둘 필요가 있다.

★ 상호 배제(Mutual Exclustion)의 정의

동시 프로그래밍에서 공유가 불가능한 자원의 동시 사용을 피하기 위해서 사용이 되는 알고리즘 중에 하나로 임계 구역으로 불리는 코드 영역에 대해서 구현을 한다. 상호 배제에 대한 소프트웨어로 구현을 완료한 데커의 알고리즘이 대표적인 사례로 볼 수 있다. 우리가 현재 단일 프로세서 시스템에서 이를 구현한 사례로는 간단하게 인터럽트만 억제하는 것으로서 이에 대한 메시징의 집합의 수를 최소한으로 해야 더욱 효율적으로 볼 수 있다.

★ 임계 구역(Critical Section)의 정의

임계 구역 혹은 공유 변수 영역은 병렬 컴퓨팅에서 둘 이상의 스레드가 동시에 접근을 하면 안 되는 공유 자원을 접근하는 코드의 일부로 이해를 할 수 있다. 이에 대해서는 입장 구역, 퇴장 구역, 나머지 구역 3가지로 분류가 되는데 입장 구역은 우리가 작성한 소스 코드에서 Session ID를 추가를 해서 WebSocket에 접속한 문장으로

볼 수 있는데 여기서 Thread 작업으로 추후 작업에 대해서 타인이 사용을 종결하기를 기다리고(afterConnectionEstablished 함수로 이해할 수 있다.), 퇴장 구역은 사용자가 WebSocket에서 빠져 나갈 때 Session ID를 삭제하고 난 이후에 Thread 작업을 통해서 종결을 하고 난 후에 대기 중인 사용자들에게 접속을 할 수 있도록 제어를 하는 개념으로 인지를 할 수 있고(afterConnectionClosed 함수로 이해할 수 있다.), 나머지 구역은 입장 구역, 퇴장 구역 이외에 진행을 하는 영역으로 이해를 할 수 있다.

[그림] 상호 배제와 임계 구역의 비유는 쉽게 이야기 해서 사거리에서 좌/우회전할 때를 연상할 수 있다.

➔ afterConnectionClosed 함수

현재 접속한 사용자가 WebSocket이 작동 중인 웹 Client에 대해서 종결(예를 들어 창을 닫거나 로그아웃을 하거나 타 페이지로 라우팅을 하는 경우 등이 되겠다.)을 하는 경우에 이 Client에 지정된 Session에 대해서 종결을 지어준 다음에 그 해당되는 Session ID에 대해서 없애는 역할을 하는 함수로 보면 된다.

B. WebSocketConfig.java

```
package net.kang.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

import net.kang.handler.EchoHandler;

@Configuration
@EnableWebMvc
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer{
        // 이번에는 WebSocket을 그대로 이용하되 EchoHandler를 이용해서 할 수 있도록 작성하였다.
        @Autowired EchoHandler echoHandler;
        // 이는 EchoHandler에 대한 Bean을 따로 Singleton을 이용하여 반환하지 않고 @Autowired를 이용해서
설정을 한다.
```

```
        @Override
        public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
                registry.addHandler(echoHandler, "/echo").setAllowedOrigins("*").withSockJS();
                // WebSocket 중에서 SockJS를 사용할 수 있도록 설정을 하는 문장.
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > config > WebSocketConfig.java]

방금 전에 우리가 작성한 EchoHandler에 대해서 적용을 하기 위해서는 Autowired를 하여 이 Bean에 대해 적용을 시켜주고 난 후에 registerWebSocketHandlers 함수에서 Handler를 추가하고 난 후에 Prefix를 대부분 /echo로 시작을 하고 이 뒤에 대해서는 WebSocket에 접속을 한다는 것을 알리기 위해 *를 추가를 한다. 그러면 http://localhost:8080/example03_2/echo/... 로 시작을 하는 URL은 현재 WebSocket에 대해서 접속을 하는 개념으로 생각을 하면 된다. 뒤에 withSockJS() 함수에 대해서는 지난 주에 언급을 하였으니 큰 설명 없이 넘어 가겠다.

C. Service 클래스 수정

```
package net.kang.service;

import java.io.IOException;
import java.util.Date;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import net.kang.domain.User;
import net.kang.repository.UserRepository;

@Service
public class TokenAuthenticationService {
        private long EXPIRATIONTIME=1000*60*60; // 1시간
        private String secret="secretkey"; // ScrectKey에 대해 임시로 secretkey로 설정.
        @Autowired UserRepository userRepository;
        public String addAuthentication(HttpServletResponse response, User user) throws IOException
{ // 로그인을 진행하고 난 후에 JWT Token을 반환한다.
                Claims claimList=Jwts.claims();
                claimList.put("userId", user.getUserId());
                claimList.put("name", user.getName());
                String jwt=Jwts.builder()
                                .setSubject(user.getId())
                                .setClaims(claimList)
                                .setIssuedAt(new Date())
                                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATIONTIME))
                    .signWith(SignatureAlgorithm.HS256, secret)
                    .compact();
                return jwt;
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > service > TokenAuthenticationService.java]

이 클래스에서는 사용자 별로 정보를 가져와서 그 사용자에 대한 JWT Token을 반환을 하는 역할로 보면 된다.

JWT Token의 구성에 대해서는 Header, Payload, Signature 3개로 나뉘는 사실은 이미 인지하고 있다. 그 중 Claims 목록에 대해서는 각 사용자의 간략한 정보를 넣는 걸 이해할 수 있고, IssuedAt는 현재 사용자가 로그인을 한 시각, Expiration은 유효 기간으로 이해할 수 있다. 이렇게 Claims, IssuedAt, Expiration, Subject(이건 이 토큰에 대한 제목으로 볼 수 있다.) 등은 Payload에 담기게 된다. 그리고 signWith에서는 임의의 Secret Key와 암호화 알고리즘을 담아서 Signature 내부에 저장을 하고, Header에서는 암호화 알고리즘에 대해서 또한 추가를 한다.

```
package net.kang.service;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Date;
import java.util.HashMap;

import javax.servlet.ServletException;

import org.apache.tomcat.util.codec.binary.Base64;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.fasterxml.jackson.databind.ObjectMapper;

import net.kang.domain.User;
import net.kang.repository.UserRepository;
import net.kang.util.Encryption;

@Service
public class UserService {
	@Autowired UserRepository userRepository;
	public User login(String userId, String password) { // 로그인 관련 함수
		User tempUser=userRepository.findByUserId(userId).orElse(new User());
		if(tempUser.equals(new User())) return null;
		String pw=Encryption.encrypt(password, Encryption.MD5);
		if(!tempUser.getPassword().equals(pw)) return null;
		return tempUser;
	}

	public User findByToken(String token) throws UnsupportedEncodingException, ServletException
{ // Token을 통해 사용자를 확인할 때 쓰는 함수
		User findUser = null;
		if(token!="" && token!=null) {
		String[] split_string = token.split("\\.");
		String base64EncodedBody = split_string[1];
		Base64 base64Url = new Base64(true);
		String body = new String(base64Url.decode(base64EncodedBody), "UTF-8");
		try {
				HashMap<String, Object> result = new ObjectMapper().readValue(body,
HashMap.class);
				Date currentTime=new Date();
				int expTime=(int) result.get("exp");
				if(expTime < currentTime.getTime()/1000) {
					throw new ServletException("유효 시간이 만료되었습니다. 다시
로그인을 진행하시길 바랍니다.");
				}
				return userRepository.findByUserId((String)
result.get("userId")).orElse(new User());
			} catch (IOException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}else {
```

```
                        throw new ServletException("유효하지 않은 토큰입니다. 다시 진행하시길 바랍니다.");
                }
                return findUser;
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > service > UserService.java]

login 함수에 대해서는 각 사용자가 LoginForm(userId, password)을 통해 로그인을 진행하고 난 후에 암호화를 통해 저장이 되어 있는 비밀번호에 대해 같이 암호화를 해서 비교를 해서 그 사용자에 대하여 반환을 하는 역할로 볼 수 있고, findByToken은 Authentication Server에서 각 사용자의 정보를 가져오는 함수로 볼 수 있는데 토큰 복호화를 하여 검색을 하는 방안으로 살펴볼 수 있다.

D. Controller 클래스 수정

```
package net.kang.controller;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import net.kang.domain.User;
import net.kang.model.LoginForm;
import net.kang.service.TokenAuthenticationService;
import net.kang.service.UserService;

@RestController
@CrossOrigin
@RequestMapping("guest")
public class GuestRestController {
        @Autowired UserService userService;
        @Autowired TokenAuthenticationService tokenAuthenticationService;
        @PostMapping("login") // Login 함수를 이용해서 사용자가 Messenger에 접근할 수 있도록 하였음.
        public ResponseEntity<?> login(HttpServletResponse response, @RequestBody LoginForm loginForm)
throws IOException{
                User loginUser=userService.login(loginForm.getUserId(), loginForm.getPassword());
                if(loginUser==null) return new ResponseEntity<ServletException>(new
ServletException("존재하지 않은 사용자입니다."), HttpStatus.UNAUTHORIZED);
                return new
ResponseEntity<String>(tokenAuthenticationService.addAuthentication(response, loginUser),
HttpStatus.OK);
        }
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > controller > GuestRestController.java]

GuestRestController에서는 로그인을 진행하지 않은 비회원들에 대해서 MongoDB에 저장된 사용자 정보를 이용해서 사용자 인증을 하기 위해 만든 클래스로 이해할 수 있다.

```java
package net.kang.controller;

import java.io.UnsupportedEncodingException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import net.kang.domain.User;
import net.kang.service.UserService;

@RestController
@CrossOrigin
@RequestMapping("user")
public class UserRestController {
	@Autowired UserService userService;

	@GetMapping("findByToken/{token}") // 현재 User의 Token을 받아서 유효성을 확인하기 위한 함수로 작성.
	public ResponseEntity<?> tokenChecking(@PathVariable("token") String token, HttpServletResponse
response) throws UnsupportedEncodingException, ServletException{
		User user = userService.findByToken(token);
		if(user.equals(new User())) {
			return new ResponseEntity<ServletException>(new ServletException("토큰이
무효합니다."), HttpStatus.UNAUTHORIZED);
		}
		return new ResponseEntity<String>(token, HttpStatus.OK);
	}
}
```

[mongoDB_JPA_Start04 > src > main > java > net > kang > controller > UserRestController.java]

UserRestController 클래스에서는 추후에 Singled Page Application에서 새로 고침을 하는 경우(SPA에서는 새로 고침을 하게 되면 벗어나게 된다. 타 페이지로 이동을 할 때에는 AJAX 통신 이외에 리다이렉트를 따로 거치치 않는다.)에 사용자에 대한 정보를 받아와서 토큰을 확인하는 역할을 한다. 그리고 올바르게 확인이 되었다면 회원정보에 대해서 반환을 하게 된다. 그리고 이 반환된 사용자 정보에 대해서는 Redux에 AJAX를 이용해서 저장을 하는데 이용을 할 수 있다.

3. ReactJS에서 SockJS 활용하기

A. Redux User Action & Reducer

```
import axios from 'axios';
import JWTDecode from 'jwt-decode';

export const USER_LOGIN = 'USER_LOGIN';
export const USER_LOGIN_SUCCESS = 'USER_LOGIN_SUCCESS';
export const USER_LOGIN_FAILURE = 'USER_LOGIN_FAILURE';
export const LOGOUT_USER='LOGOUT_USER';

export const USER_FROM_SERVER='USER_FROM_SERVER';
export const USER_FROM_SERVER_SUCCESS='USER_FROM_SERVER_SUCCESS';
export const USER_FROM_SERVER_FAILURE='USER_FROM_SERVER_FAILURE';
export const RESET_USER_FROM_SERVER='RESET_USER_FROM_SERVER';

const ROOT_URL = 'http://localhost:8080/example03_2';

export function userLogin(loginForm){
    const request=axios({
        method : 'post',
        url : `${ROOT_URL}/guest/login`,
        data : loginForm
    });
    return{
        type : USER_LOGIN,
        payload : request
    };
}

export function loginUserSuccess(userToken){
    const user=JWTDecode(userToken);
    return{
        type : USER_LOGIN_SUCCESS,
        payload : user
    };
}

export function loginUserFailure(error){
    return{
        type : USER_LOGIN_FAILURE,
        payload : error
    };
}

export function logoutUser(){
    return{
        type : LOGOUT_USER
    }
}

export function userFromServer(tokenFromStorage){
    const request=axios({
        method : 'get',
        url : `${ROOT_URL}/user/findByToken/${tokenFromStorage}`,
        headers : {
```

```
            'Authorization' : `Bearer ${tokenFromStorage}`
        }
    });
    return {
        type : USER_FROM_SERVER,
        payload : request
    };
}

export function userFromServerSuccess(userToken) {
    const user=JWTDecode(userToken.data);
    return {
        type : USER_FROM_SERVER_SUCCESS,
        payload : user
    };
}

export function userFromServerFailure(error){
    return{
        type : USER_FROM_SERVER_FAILURE,
        payload : error
    };
}

export function resetUserFromServer(){
    return{
        type : RESET_USER_FROM_SERVER
    }
}
```

[ReactJS_Example04_Client > src > action > user_action.js]

```
import {USER_LOGIN, USER_LOGIN_SUCCESS, USER_LOGIN_FAILURE, LOGOUT_USER,
         USER_FROM_SERVER, USER_FROM_SERVER_SUCCESS, USER_FROM_SERVER_FAILURE, RESET_USER_FROM_SERVER
} from "../action/user_action";
const INITIAL_STATE = {
    current : {user : null, status : 'login', loading : false, error : null},
}

export default function(state=INITIAL_STATE, action) {
    let error;
    switch (action.type) {
        case USER_LOGIN :
            return {...state, current : { user: null, status: 'login', error: null, loading: true }};
        case USER_LOGIN_SUCCESS :
            return {...state, current : { user: action.payload, status: 'authenticated', error: null, loading:
false}};
        case USER_LOGIN_FAILURE :
            error = action.payload.data || {message: action.payload.message};
            return {...state, current : { user: null, status: 'login', error: error, loading: false }};
        case LOGOUT_USER :
            return {...state, current : { user : null, status : 'login', error : null, loading : false}};

        case USER_FROM_SERVER :
            return {...state, current : { user: null, status: 'login', error: null, loading: true }};
        case USER_FROM_SERVER_SUCCESS :
            return {...state, current : { user: action.payload, status: 'authenticated', error: null, loading:
false}};
```

```
        case USER_FROM_SERVER_FAILURE :
            error = action.payload.data || {message: action.payload.message};
            return {...state, current : { user: null, status: 'login', error: error, loading: false }};
        case RESET_USER_FROM_SERVER :
            return {...state, current : { user : null, status : 'login', error : null, loading : false}};

        default :
            return state;
    }
}
```

[ReactJS_Example04_Client > src > reducer > reducer_user.js]

Redux에서 부가한 state 관리는 사용자의 로그인 정보에 대해서 current를 통해서 담았다. 각 Action들을 분석을 한다면 아래와 같이 작성을 할 수 있다.

로그인 진행 관련 Action

- USER_LOGIN : 사용자가 로그인을 실행하는 경우에 나오는 Action Type으로서 LoginForm Component를 통해 입력한 로그인 정보를 방금 전에 작성한 REST API Server에 AJAX 통신으로 ResponseEntity를 받아와서 로그인 진행 상황에 대한 결과 값을 가져온다.
- USER_LOGIN_SUCCESS : 로그인을 진행하고 난 후에 사용자 Token을 받아오고 난 직후, 현재 사용자의 정보를 Reducer에 저장하기 위해 JWT 토큰을 복호화해서 재저장을 한다.
- USER_LOGIN_FAILURE : 로그인 과정 중에 존재하지 않는 사용자에 대해서 예외 결과를 받아오고 난 후에 이에 대한 에러 처리를 하기 위하여 예외를 받았을 때 나오는 message를 통해 가져와서 리턴을 해 준다.
- LOGOUT_USER : 사용자가 로그아웃을 하고 난 이후에 state의 user 값들에 대해 초기화를 시켜주는 역할로 보면 되고 동시에 Client에 있는 session에서 Token을 삭제를 동시에 하여 로그아웃을 처리한다.

로그인 이후 User Token의 유효성 확인

- USER_FROM_SERVER : 로그인을 완료한다면 현재 Client내에 존재하는 Session에 JWT Token을 저장을 하고 난 이후에 Root Component에서 벗어나는 경우에(즉 새로 고침을 하는 경우 등등이 되겠다.)도 Session 내부에는 JWT Token이 그대로 저장이 되어 있어서 이를 가져와서 토큰에 대한 유효성을 확인 시키는 작업을 UserRestController에서 재실행하는 것으로 이해하면 된다.
- USER_FROM_SERVER_SUCCESS : JWT Token에 대해서 확인이 완료되는 경우에 state의 user는 그대로 설정을 해 두도록 하고, Session에 있는 JWT Token도 그 자리에 계속 머물게 된다.
- USER_FROM_SERVER_FAILURE : JWT Token에 대한 유효성을 벗어나는 경우에는 재 로그인 처리를 하기 위해서 Session에 있는 JWT Token을 다시 반환하고, 현재 state에 있는 user 값들에 대하여 에러 처리를 한다.
- RESET_USER_FROM_SERVER : 이는 Root Component에서 벗어나는 경우에 User와 REST API Server 간에 비동기적 설정에 대한 문제가 야기되지 않기 위하여 초기화 값으로 설정을 해주고 난 이후에 Root Component에서 USER_FROM_SERVER Action을 다시 실행시켜주는 것으로 이해하면 된다.

B. ChattingDocument.js

```
import React, {Component} from 'react';
import SockJS from 'sockjs-client';
class ChattingDocument extends Component{
    constructor(props) {
        super(props);
        this.state = {
            messages : [],
            context : "",
            topic : "코딩이야기"
        }
        this.sock = new SockJS('http://localhost:8080/example03_2/echo');
        this.sock.onopen = () => {
            console.log('connection open');
        };
        this.sock.onmessage = (e) => {
            console.log('message received:', JSON.parse(e.data));
            //incoming message from server, store in state
            this.setState( { messages: this.union_arrays(this.state.messages, JSON.parse(e.data)) });
        };
        this.sock.onclose = () => {
            console.log('close');
        };
        this.handleFormSubmit = this.handleFormSubmit.bind(this);
        this.handleChange = this.handleChange.bind(this);
    }

    union_arrays(array01, array02){
        if(array02 === [] && array01 === []) return [];
        var obj = {};
        for (var i = array01.length-1; i >= 0; -- i) {
            obj[array01[i].id] = array01[i];
        }
        for (var i = array02.length-1; i >= 0; -- i){
            obj[array02[i].id] = array02[i];
        }
        var res = [];
        for (var k in obj) {
            if (obj.hasOwnProperty(k))  // <-- optional
                res.push(obj[k]);
        }
        res.sort(function(a, b){
            return a.id > b.id ? -1 : a.id < b.id ? 1 : 0;
        });
        return res;
    }

    handleClick(){
        this.sock.send('chattingList');
    }

    handleChange(e){
        this.setState({ [e.target.name] : e.target.value });
    }

    handleFormSubmit(e) {
        e.preventDefault();
        let text = this.state.context;
```

```
        let topic = this.state.topic;
        let userId = this.props.currentUser.user.userId;
        this.sock.send(`addMessageWn${text}Wn${userId}Wn${topic}`);
        this.setState({
            context : "",
            topic : "코딩이야기"
        });
    }

    render(){
        return (
            <div>
                <div className="container">
                    <form onSubmit={this.handleFormSubmit}>
                        <div className="form-group">
                            <div className="input-group">
                                <select name="topic" value={this.state.topic} onChange={this.handleChange}
className="form-control">
                                    <option value="코딩이야기">코딩이야기</option>
                                    <option value="음악이야기">음악이야기</option>
                                    <option value="먹방이야기">먹방이야기</option>
                                    <option value="일상이야기">일상이야기</option>
                                </select>
                                <input type="text" name="context" value={this.state.context}
onChange={this.handleChange} className="form-control" placeholder="채팅 내용을 입력하세요..." />
                                  
                                <span className="input-group-btn">
                                    <button type="submit" className="btn btn-primary">보내기</button>
                                      
                                    <button type="button" className="btn btn-info" onClick={() =>
this.handleClick()}>새로고침</button>
                                </span>
                            </div>
                        </div>
                    </form>
                </div>
                <ul className="list-group">{
                    this.state.messages.map((message, idx) => {
                        let topicTheme = "";
                        switch(message.topic){
                            case "코딩이야기" :
                                topicTheme = "list-group-item-info";
                                break;
                            case "음악이야기" :
                                topicTheme = "list-group-item-warning";
                                break;
                            case "먹방이야기" :
                                topicTheme = "list-group-item-success";
                                break;
                            case "일상이야기" :
                                topicTheme = "list-group-item-secondary";
                                break;
                            default :
                                topicTheme = "list-group-item-light";
                                break;
                        }
                        return(
                            <li className={`list-group-item ${topicTheme} d-flex justify-content-between align-
items-center`} key={idx}>
```

```
                                [{message.topic}] {message.from.name} -> {message.message}
                                <span class="badge badge-primary badge-pill">{new
Date(message.currentTime).toLocaleString()}</span>
                            </li>
                        );
                    })}
                </ul>
            </div>
        );
    }
}
export default ChattingDocument;
```

[ReactJS_Example04_Client > src > component > user > ChattingDocument.js]

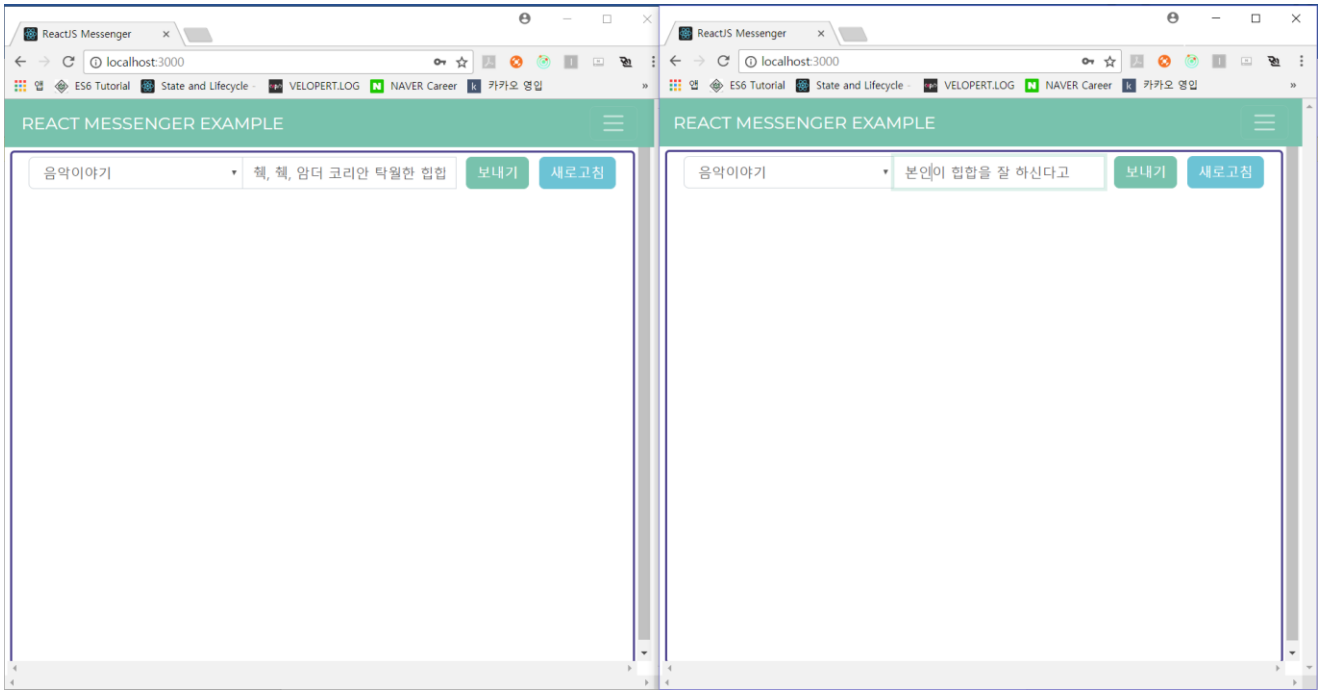
SockJS를 접목하기 위해서 Redux로 작성을 하게 된다면 난해하게 되는 점을 고려하여 순수 ReactJS를 기반으로 하는 채팅에 대해서 작성을 하였다.

ReactJS에서 각 Form 들에 대해서 설정을 하기 위해 ChattingDocument 내부에 있는 Form 들에 대한 이벤트를 handleClick, handleSubmit, handleChange 등을 통해서 설정을 하는 것이 관례인데 여기서 constructor에서 생성한 SockJS의 문장에서는 각각 onopen, onmessage, onclose 3가지로 나뉠 수 있다. onopen은 각 사용자 별로 Client에서 접속을 하였다는 것을 알리고 WebSocket에 Client Session ID를 통해 보고를 하는 기능으로 볼 수 있고, onmessage는 현재 Client에서 요청하는 Message에 대해 보낸 다음(send 함수를 통해 보내겠다.) 그 다음에 Message에 대해서 반환을 해서 현재 클라이언트에 전송되게끔 하는 함수로 볼 수 있고, onclose는 WebSocket에 현재 Client가 나갔다는 사실에 대해서 보고를 하는 함수로 볼 수 있다. 지난 시간에는 STOMP 프로토콜을 이용해서 MessageMapping을 하는 방안에 대해서 구상을 하여 적용했지만 이번에는 EchoHandler를 통해서 간략한 메시징을 보내는 수준으로 보내는 방법으로 작성을 하였기에 여기서는 간단한 문자열 정도들만 통신을 할 수 있다는 점에서 의의를 가지면 되겠다.

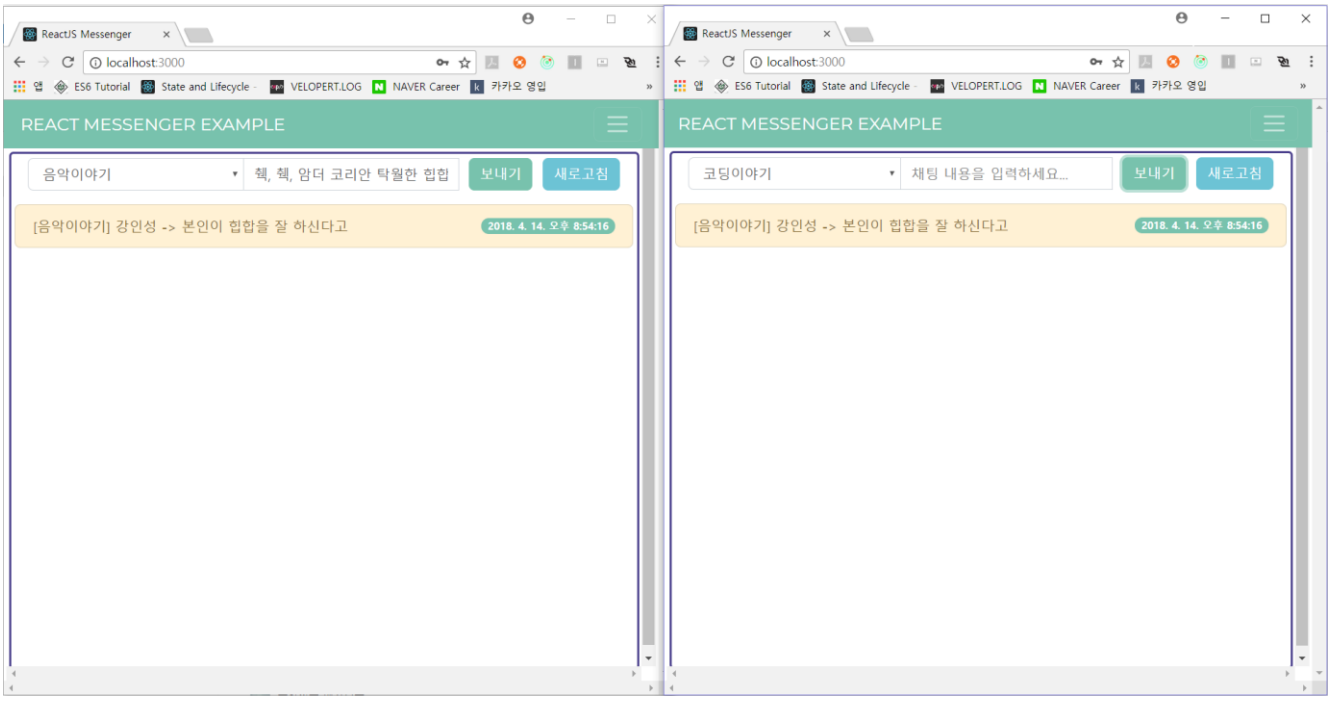
그리고 각 클라이언트에서 사용자가 접속을 하고 난 후에 채팅 내용들을 입력하게 된다면 메시지를 기반으로 통신을 하고 이에 대한 결과 값을 보기 좋게 정리를 하는 작업을 ReactJS의 Virtual DOM을 통해서 정리를 하는 역할로 인지를 하면 되겠다. ChattingDocument 내부에 있는 state의 역할로서는 messages는 현재 가지고 있는 채팅 내용들, context는 입력 내용, topic은 채팅 주제를 선택하고 난 이후에 그 값을 설정하는 것으로 이해할 수 있다. 그리고 새로 고침 버튼에 대해서는 chattingList 메시지를 보내고 이를 받으면 WebSocket에는 MongoDB에 있는 현재 채팅 목록들에 대해서 JSON으로 반환을 해 주고 이 Component 내부에서는 JSON 데이터의 결과를 파싱을 해서 적용을 시켜서 전체 채팅 목록을 보여주도록 하고 그 다음에 addMessage에서는 각 채팅 요소들에 대해서 개행 문자로 구분을 하고 새로운 채팅 내용에 대해서 WebSocket을 통해서 받아서 저장을 하고 채팅을 작성한 본인을 포함하여 모든 인원들에게 목록을 보여주게 되어 실시간으로 채팅을 반영하는 것으로 인지를 하면 되겠다.

4. 실행 결과와 GitHub 주소 참조와 Logger

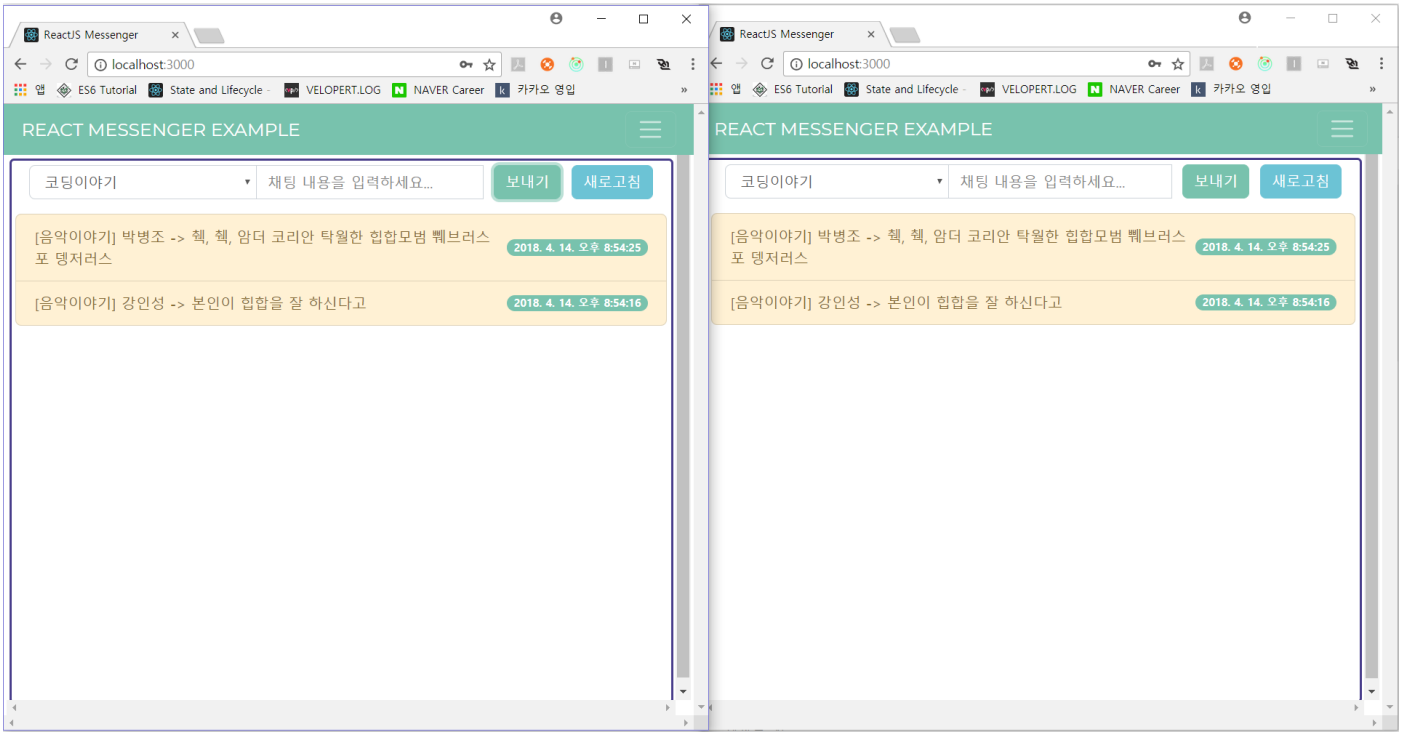
실행 결과



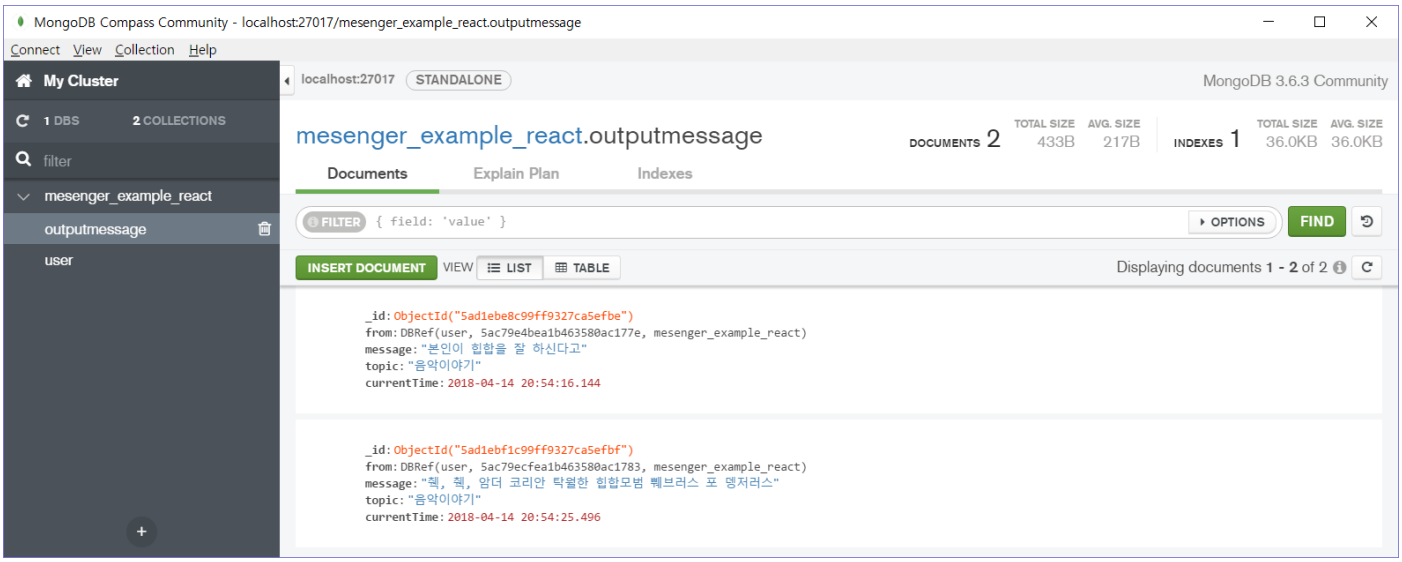
채팅의 원활한 결과를 확인하기 위해서 동시 로그인 기능에 대해서는 배제를 하였다. 우측부터 메시지를 보내면



각각 모든 사람들에게 채팅 내용을 보내게 되어 확인을 할 수 있게 된다. 좌측에서 보내게 된다면...



좌측에 있는 사람도 마찬가지로 보내는 모습을 볼 수 있다. 이 채팅 내용들도 또한 지난 시간과 마찬가지로...



이처럼 MongoDB에 저장을 하게 된다.

GitHub 주소 참조

https://github.com/tails5555/mongoDB_JPA_Start04 - MongoDB + WebSocket 채팅 서버.

https://github.com/tails5555/ReactJS_Example04_Client - ReactJS + Redux로 구현한 SPA Client.

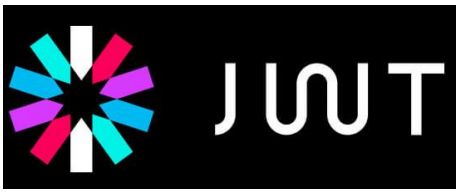
그리고 이 문서는 이 홈페이지에 들어가서(mongoDB_JPA_Start04만 해당) src > doc 파일에 pdf 파일로 추가를 해뒀으니 소스 코드와 같이 참고하면 좋겠다.

(참고) Logger에 대한 개념

EchoHandler 클래스에 주석으로 처리 된 Logger에 대해서 적용을 하는 방법에 대해서도 꼭 알고 넘어가야 하는데 sysout를 입력해서 알아보는 결과에 대해서는 자제를 해야 한다. 본래 Java System에서 출력 작업을 하는데 있어서 의외로 큰 시간을 소요를 한다는 점에 대해서는 인지를 하고 있어야 한다. 그래서 Spring에서 Server 간에 결과를 인지하고 싶다면 Logger를 이용을 하는 것을 추천을 하고 싶다. 그렇지만 Logger의 종류도 생각보다 의외로 많은데 우선 대중적으로 많이 쓰고 있는 slf4j, 그 다음으로 많이 쓰는 log4j, 그리고 최근에 나온 logback 등이 있다. 각 Logging을 하는 방안에 대해서는 이 페이지에서 스스로 공부를 하여 반영을 하면 좋겠다.

http://java-school.net/java/Logging - Logging 관련 소스 코드와 안내

공식문서 안내



https://jwt.io/ - JWT Token 공식 문서. 영어로 작성되어 있다.

https://reactjs.org/ - ReactJS 공식 문서. 영어로 작성되어 있다.

또한 ReactJS 강의나 동영상은 velopert 블로그(JS 개발 관련 이야기도 많이 있어서 Node.js 서버에 대해서 인지를 해 볼 의향이 있다면 꼭 참고하면 좋겠다.)에서 공부하면 더욱 도움 된다.

https://vuejs.org/ - ReactJS는 생각보다 어렵기 때문에 SPA에 대한 개념 중에서 그나마 쉬운 개념을 가지고 있는 VueJS로도 개발을 원한다면 VueJS 공식 문서도 읽어보면 좋다.

https://deminoth.github.io/redux/ - Redux 공식 문서 한국어 번역. 영문판 공식 문서와 함께 읽으면 좋겠다.

출처

https://github.com/bfocht/chat-client - ReactJS 채팅 구현 소스 코드

https://github.com/rajaraodv/react-redux-blog - ReactJS + Redux 접목 구현 소스 코드

https://blog.naver.com/wlguswls89/220325271561 - EchoHandler 개념과 WebSocket 개념

https://ko.wikipedia.org/wiki/%EC%83%81%ED%98%B8_%EB%B0%B0%EC%A0%9C – 상호 배제 개념

https://ko.wikipedia.org/wiki/%EC%9E%84%EA%B3%84_%EA%B5%AC%EC%97%AD – 임계 구역 개념

https://bootswatch.com/minty/ - ReactJS Application CSS Theme. Bootswatch가 의외로 좋은 테마가 많으니 Front 관련해서 같은 Bootstrap에 대해 추가적인 색상을 권장한다면 이 페이지에서 적용시키면 된다. 또한 Bootstrap 4 최신 버전에 대해 반영이 되어 있어서 이를 참고하면 된다.