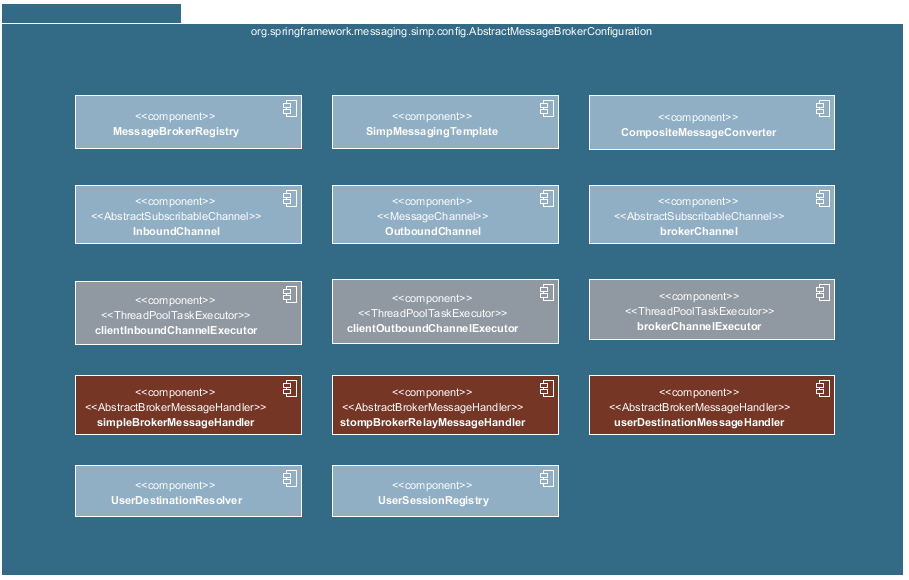
**Code Review**

**CONFIG:**

Con la annotation **@EnableWebSocketMessageBroker** vengono imporatati i beans definiti in **WebSocketMessageBrokerConfigurationSupport**

1. **AbstractMessageBrokerConfiguration**



Permette la configurazione di alcuni degli oggetti che costruisce (con impostazioni default) attraverso i metodi:

configureClientInboundChannel (ChannelRegistration registration)  
configureClientOutboundChannel (ChannelRegistration registration  
configureMessageBroker(MessageBrokerRegistry registry);

1. **WebSocketMessageBrokerConfigurationSupport**

* classe abstract estensione di AbstractMessageBrokerConfiguration.
* Crea un **WebMvcStompEndpointRegistry** che rende configurabile col metodo :
  + abstract void **registerStompEndpoints**(StompEndpointRegistry registry);
  + Di tale Registry espone un UrlHandlerMapping(Vedi sotto)
* Crea un SimpleUrlHandlerMapping (SMVC) che si basa su una Map di   
  <String, HttpRequestHandler> che verra customizzata con dallo sviluppatore  
  con WebMvcStompEndpointRegistry .addEndpoint(String uris)

1. **DelegatingWebSocketMessageBrokerConfiguration**- Estensione di WebSocketMessageBrokerConfigurationSupport .  
   - Definisce una List<WebSocketMessageBrokerConfigurer> configurers

* E’ la class che a runtime esegue la scansione del classpath e popola la sua List interna con le classi @Configuration che implementano WebSocketMessageBrokerConfigurer ?

1. **WebSocketMessageBrokerConfigurationSupport**

**CLASSI MANIPOLABILI DA DEVELOPER:   
(Create in WebSocketMessageBrokerConfigurationSupport)**

**@Bean (**SimpleUrlHandlerMapping )  
  
**@Bean  
SubProtocolWebSocketHandler:**

**@Bean (**SimpleUrlHandlerMapping )  
**WebMvcStompEndpointRegistry**Creatoin DelegatingWebSocketMessageBrokerConfiguration**.**La sua configurazione e’ delegata alle subclassestramite override di

**.registerStompEndpoints(registry);**

**Campi**:

* WebSocketHandler; (in realta e’ il @Bean SubProtocolWebSocketHandler)
* SubProtocolWebSocketHandler (stesso obj di prima unwrapped e casted);
* StompSubProtocolHandler = new StompSubProtocolHandler(); (costr)
* List<WebMvcStompWebSocketEndpointRegistration> (costr lista vuota)

.addEndpoint(String paths...)

* Aggiunge il this.StompSubProtocolHandler alla lista dei gestori subprotocols.
* Crea un un oggetto di configurazione (WebMvcStompWebSocketEndpointRegistration)  
  a cui passa:
  + paths,
  + this.webSocketHandler (tipo SubProtocolWebSocketHandler) ,
  + this.sockJsScheduler);
* Aggiunge l oggetto di configurazione alla list di registrations (vedi getHandlerMapping())

. getHandlerMapping()

* Crea una Map di <url, HttpRequestHandler> urlMap
* Cicla su registrations popolate su .addEndpoint e per ogni registration
  + Estrae una MultiValueMap<HttpRequestHandler, String> con la chiamata   
    registration.getMappings()   
    (vedi WebMvcStompWebSocketEndpointRegistration)
  + Lo aggiunge alla urlMap iniziale
* restituisce sotto forma di SimpleUrlHandlerMapping (SMVC) la urlMap

**WebMvcStompWebSocketEndpointRegistration (usato da WebMvcStompEndpointRegistry)**

.getMappings()

* Collega <pattern url>, <SubProtocolWebSocketHandler>   
  (sotto forma di HttpRequestHandler)
* A seconda che sia stato abilitato o meno il sockJs l oggetto HttpRequestHandler  
  restituito sara’ un wrapper intorno al SubProtocolWebSocketHandler di tipo:
  + SockJsHttpRequestHandler
  + WebSocketHttpRequestHandler

. withSockJS()

predispone l oggetto per utilizzare un SockJsHttpRequestHandler piuttosto che un WebSocketHttpRequestHandler

**@Bean  
SubProtocolWebSocketHandler: (Creata da WebSocketMessageBrokerConfigurationSupport)**

classe che gestisce

* Un insieme di SubProtocolHandler (sulla base dell handshake)
  + SubProtocolHandler default
* Una ConcurrentMap di WebSocketSession (Che si aggiungono con afterConnectionEstablished)
* Metodi per gestire messaggi in IN e OUT:  
  + Gestisce gli **input** da (Attivato da Websocket request)  
    handleMessage(WebSocketSession session, WebSocketMessage<?> message)  
    Delega al StompSubProtocolHandler:
    - protocolHandler.handleMessageFromClient(session, message, this.clientInboundChannel);

* + Gestisce gli **output** (verso client)da  
    handleMessage(Message<?> message)  
    Delega al StompSubProtocolHandler. handleMessageToClient(WebSocketSession session, Message<?> message)

**StompSubProtocolHandler (delegato comunicazioni in STOMP da SubProtocolWebSocketHandler)**.start  
si collega come subscriber del Outbound channel(Attivato da Websocket request -> send Msg to inboundChannel)  
.handleMessageFromClient(session, message, this.clientInboundChannel);

* Esegue un decode dal ByteBuffer del payload del Message
* Fa il parsing del messaggio ed estrae il frame STOMP.
* Esegue le varie operazioni CONNECT, DISCONNECT,
* Outbound.send()   
  Spedisce il messaggio sul Channel Outbound arg   
  (che e’ l INBOUND del **SubProtocolWebSocketHandler)**
* Pubblica eventi di SEND, SUBSCRIBE, UNSUBSCRIBE tramite il suo ApplicationEventPublisher (come evento nell ApplicationContext di Spring)

Recuperabili con un ApplicationListener (Spring Context)

**(Cio che finisce sull OutboundChannel) -> WebsocketSession.send**.handleMessageToClient(WebSocketSession session, Message<?> message)  
 . sendToClient(session, accessor, payload);

* Encode in TextMessage o BinaryMessage
* Session.sendMessage(message)

**CLASSI A RUNTIME: (da verifica in DEBUG)**

**SENDER:  
SimpleMessagingTemplate** collegato al BrokerChannel per send(OUT)

**HANDLERS:**

**Tutti gli Handler sono collegati a** Inbound, OutBound e BrokerChannel

**SimpleBrokerMessageHandler (impl MessageHandler.handleMessage)**

**campi:**Subscribed: Inbound, BrokerChannel  
SubscriptionRegistry

. handleMessageInternal (per msg che arrivano sui suoi subscribed channels)

* + in base a header verifica quale tipo di message (SI SimpleType)
  + SUBSCRIBE e UNSUBSCRIBE: add/remove su SubscriptionRegistry
    - SubscribeId (quale subscription)
    - SessionId (quale client)
    - Destination (nome logico) (AntPatternMatcher per search)
  + CONNECT:
    - Restituisce un ack
  + DISCONNECT:
    - Rimuove tutte le subscriptions per quel SessionId
  + MESSAGE:
    - sendMessageToSubscribers (vedi sotto)

. sendMessageToSubscribers

* + verifica tutte le subscription interessate (data la destination)
    - AntPatternMatch
    - Recupera per ognuna (sessionId, subscriptionId)
    - Add queste info negli Headers del Message
  + Per ogni SessionId
    - Per ogni SubscriptionId
      * Spedisce il messaggio sull OutboundChannel

A seconda di quale enable viene chiamato su

SimpleBrokerMessageHandler (MOM STOMP in memoria)  
StompBrokerRelayMessageHandler (delega la gestione messaggi stomp ad un MOM esterno)

**Configurazione:**

.addEndpoint(url) ->   
associato al **SubProtocolWebSocketHandler e . StompSubProtocolHandler**

**Tests e CallStack:**

**Apertura connessione  
  
1) Comuni:**

* 1. Client SockJs
     1. Invia un primo messaggio per stabilire una connessione websocket
     2. invia un messaggio STOMP CONNECT a uri Endpoint stomp
  2. DispatcherServlet.doDispatch
     1. getHandler
     2. handler.getAdapter() (vedi C)
     3. **handlerAdapter**.handle() (Vedi D)
  3. AbstractUrlHandlerMapping.getHandlerInternal(http request)
     1. Trova l’handler per quel uri -> SockJsHttpRequestHandler
  4. HttpRequestHandlerAdapter.handle (wrapper di SockJsHttpRequestHandler)
  5. SockJsHttpRequestHandler.handleRequest();
  6. DefaultSockJsService(AbstractSockJsService).handleRequest
  7. **DefaultSockJsService.handleTransportRequest**

1. **HTTP Richiesta websocket -> Handshake -> Init Websocket Session:  
     
   SockJs ->**http://localhost:8080/SMVC-800-WebsocketServer/chat/059/wrgkbexp/websocket
   1. **DefaultSockJsService.handleTransportRequest**
   2. WebSocketTransportHandler.handleRequest
   3. DefaultHandshakeHandler(AbstractHandshakeHandler).doHandshake
   4. TomcatRequestUpgradeStrategy.upgrade
      1. Crea **StandardWebSocketHandlerAdapter** passandogli wsHandler (SockJsHttpRequestHandler)
2. **STOMP CONNECT**
   1. **DefaultSockJsService.handleTransportRequest**
   2. XhrReceivingTransportHandler.handleRequestInternal
   3. SubProtocolWebSocketHandler.handleMessage
3. **STOMP SEND**
   1. **DefaultSockJsService.handleTransportRequest**
   2. XhrReceivingTransportHandler.handleRequestInternal
   3. SubProtocolWebSocketHandler**.** handleMessage(WebSocketSession session, WebSocketMessage<?> message)
   4. StompSubProtocolHandler.handleMessageFromClient(WebSocketSession session, WebSocketMessage<?> webSocketMessage, MessageChannel outputChannel)
   5. outputChannel.send() -> inbound
   6. Se trova subscribers per quella destination
      1. SimpleBrokerMessageHandler. handleMessageInternal
      2. SimpleBrokerMessageHandler. sendMessageToSubscribers   
         (se ce ne sono per quella destination)
4. **Send Da Server a Client**
   1. SubProtocolWebSocketHandler**.** handleMessage(WebSocketMessage<?> message)
   2. StompSubProtocolHandler.handleMessageToClient(WebSocketSession session, Message<?> message)
   3. WebSocketSession.send

WebSocketSession si riferisce alla sessione (Tcp) iniziata col client.  
E’ da qui che i messaggi vengono inoltrati al client.

1. **Client SockJS: + Stomp.js**
   1. Connessione verso Stomp Endpoint
   2. Subscribe su una destinazione (’/data’) (IN)
      1. Client.subscribe (destination, function (handler onMessage)
   3. Eventuali Send su una destinazione (OUT)

**CONTROLLO SVILUPPATORE:**

1. Classe configuration per contesto websocket
2. @Configuration:  
   Rende la classe un contesto di Spring che si aggiunge a quelli gia caricati
3. @EnableWebSocketMessageBroker :   
   importa i @Beans da WebSocketMessageBrokerConfigurationSupport
4. Aggiungere endpoints al registro STOMP:
   1. WebMvcStompEndpointRegistry.addEndpoint(url)
5. Abilitare uno StompMessageBroker (Simple o Relay) e dargli un prefisso
   1. Simple (gestore destinations in memoria)
   2. Relay (delega ad un sistema esterno es. RabbitMQ)
6. Abilitare un prefisso per Application Path:
   1. Reindirizzati a Controller per ricezione messaggi via Websocket/Stomp

**WEB UI ad Application: (da SockJS javascript)**

1. Stabilire connessione
   1. HTTP handshake per protocollo WS
   2. Connessione con endpoint STOMP
2. Azioni STOMP (Non passano da Controller):
   1. Subscribe
      1. Callback per ricezione
   2. Unsubscribe
   3. Connect
   4. Disconnect
3. Interazioni con Applicazione (Passano da Controller):
   1. Send (Vedi Mapping)

**Application a UI:**

1. Applicazione spedisce su BrokerChannel
2. BrokerChannel inoltra a StompBrokerHandler (gestore destinations)
3. StompBrokerHandler tramite OutboundChannel inoltra a tutti i client che hanno subscribe per quella destination
4. Se ci sono Clients con una subscribe ed una connessione attiva, riceveranno il messaggio.  
   (Scattera’ callback ricezione)

**TOPOLOGIA:**

?-> **brokerChannel**-> subclasses di AbstractBrokerMessageHandler  
In teoria:  
Sembra che il broker channel sia usato quando c’e’ una connessione ad un STOMP Broker esterno.

**Client to Server:**1) WebsocketRequest -> **InboundChannel** -> subclasses di AbstractBrokerMessageHandler  
2) Creare un MessageTemplate(per test) che scrive sull InboundChannel (da verificare se si prende via @Autowire)

**Server to Client:**1) SimpleBrokerMessageHandler. sendMessageToSubscribers -> **outboundChannel**2) SimpleMessageTemplate (@Bean e @Autowire da contesto) send su **outboundChannel**

**outboundChannel** -> StompSubProtocolHandler.handleMessage -> WebSocketSession.send

@MessageMapping("/chat/{topic}")  
@SendTo("/topic/messages")

**SockJs Stomp:**

1. Connect:
   1. Url = <Host>:<Port>/<WebApp>/<StompEndpoint>
2. Disconnect
   1. stompClient.disconnect
3. Subscribe
   1. stompClient.subscribe(url, function)
      1. url=<Host>:<Port>/<WebApp>/<brokerUri>/
      2. function (onMessage)
4. Unsubsribe
5. Send
   1. url <Host>:<Port>/<WebApp>/<appPrefix>/<urlMessageMapping>/<Topic>