



**UNIVERSITÀ
DEGLI STUDI
DI PADOVA**



**DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE**

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

**CORSO DI LAUREA MAGISTRALE IN
ICT FOR INTERNET AND MULTIMEDIA**

**“ANALYSIS OF ROBUST INTERNET INSTANT MESSAGING
PROTOCOLS FOR CHAT APPLICATIONS”**

Relatore: Prof. / Dott. ERSEGHE TOMASO

Laureando/a: VENKATA AVINASH JAKKAMPUDI

ANNO ACCADEMICO 2021 – 2022

Data di laurea 11/07/2022

Table of Contents	Page
Introduction	11
Literature Part	12
Chapter 1 -- WebRTC	13
1.1 Introduction to WebRTC	14
1.2 Major Components in WebRTC	14
1.2.1 API: getUserMedia	14
1.2.2 API: RTCPeerConnection	14
1.2.3 API: RTCDataChannel	15
1.3 WebRTC in the Real World	15
1.3.1 Signaling	16
1.3.1.1 How does Signaling work	17
1.3.1.2 Technologies Available for signaling servers	18
1.3.2 ICE & STUN/TURN	18
1.4 Typical Architecture Topologies for WebRTC	19
1.4.1 Mesh Topology	20
1.4.2 MCU	20
1.4.3 Routing	21
1.5 Use Cases	22
1.6 Security Considerations	22
1.7 Why WebRTC is so important	22
Chapter 2 -- XMPP	23
2.1 Introduction to XMPP	24
2.2 History to XMPP	25
2.3 How XMPP works	26

2.4 Client Server architecture in XMPP	26
2.5 Anatomy of JID	28
2.6 Core protocol defined in RFC 6120 i.e., about Streams	28
2.7 XMPP Stanzas	29
2.7.1 Presence Stanza	29
2.7.2 Message Stanza	29
2.7.3 IQ Stanza	29
2.8 XMPP features	30
2.8.1 Asynchronous Protocol	30
2.8.2 Persistent Connection	30
2.8.3 Decentralization	30
2.9 Few Common Extensions in XEP series	30
2.9.1 XEP-0280 : Message Carbons	30
2.9.2 XEP-0310 : Message Archive Management (MAM)	30
2.9.3 XEP-0352 : Client State Indication (CSI)	31
2.9.4 XEP-0268 : Mobile Considerations	31
2.9.5 XEP-0198	31
2.10 Services built on top XMPP	31
Chapter 3 -- WebSockets	32
3.1 What is HTTP	33
3.2 Drawbacks in HTTP	34
3.3 What are WebSockets	34
3.3.1 WebSocket Architecture	34
3.3.2 Protocol Handshake	35
3.3.3 Relationship to TCP and HTTP from WebSockets	38

3.3.4 Security Considerations	38
3.3.5 Use Cases	39
3.3.5.1 Gaming Applications	39
3.3.5.2 Chat Applications	39
3.3.5.3 Live feed	39
3.3.6 Advantages of WebSockets	39
3.4 Introduction to STOMP	40
3.4.1 Latest version updates	41
3.4.2 Message brokers available for STOMP	42
3.4.3 Connecting clients to a broker	42
3.4.3.1 Connect	42
3.4.3.2 Disconnect	42
3.4.4 Sending message from clients to a broker	43
3.4.5 Subscribing clients to message from a broker	44
3.4.5.1 Subscribe	44
3.4.5.2 Message	45
3.4.5.3 Unsubscribe	45
3.4.6 Acknowledgement	45
3.4.6.1 Client message acknowledgement	45
3.4.6.2 Broker Commands Acknowledgement	46
3.4.7 Examples and Implementations of STOMP	47
Chapter 4 -- AMQP	48
4.1 Introduction to AMQP	49
4.2 How AMQP works	49
4.3 Consumer driven Messaging Approach	49

4.4 Message Broker	50
4.5 Overview of AMQP Protocol	51
4.5.1 Components of AMQP	51
4.5.1.1 Message Queues	51
4.5.1.2 Exchange and Exchange types	52
4.5.1.3 Binding	52
4.5.1.4 Message and Content	52
4.5.1.5 Connection	52
4.5.1.6 Channel	52
4.5.1.7 Virtual hosts	53
4.6 AMQP Architecture	53
4.6.1 Producers	53
4.6.2 Exchange	53
4.6.3 Routing Key	54
4.6.4 Consumers	54
4.6.5 Subscriptions	54
4.6.6 Publishing	55
4.6.6.1 Direct Exchange	56
4.6.6.2 Fanout Exchange	57
4.6.6.3 Topic Exchange	57
4.6.6.4 Header Exchange	58
4.7 Key Features of AMQP	59
4.7.1 Security	59
4.7.2 Reliability	59
4.7.3 Interoperability	60

Implementation Part	61
Chapter 5 – Introduction to Spring Boot	62
5.1 Spring Boot	63
5.2 Advantages of Spring Boot	63
5.3 Specifications of Spring Boot	64
5.4 Features of Spring Boot	64
5.4.1 Web Development	64
5.4.2 Spring Application	64
5.4.3 Application Events and Listeners	65
5.4.4 Admin Support	65
5.4.5 Externalized Configuration	65
5.4.6 Properties Files	65
5.4.7 YAML Support	66
5.4.8 Logging	66
5.4.9 Security	66
5.5 Spring Boot Annotation	66
5.5.1 @Configuration	66
5.5.2 @Enable Auto Configuration	67
5.5.3 @ComponentScan	67
5.5.4 @SpringBootApplication	67
5.6 Spring Boot Starters	67
5.6.1 Web Starter	68
5.6.2 Data JPA Starter	68
5.6.3 Mail Starter	69
5.6.4 Test Starter	69

5.7 Problems of Spring Boot	70
Chapter 6 – WebSocket in Spring Boot	71
6.1 Dependencies Required	72
6.2 Detail of application.properties files	73
6.2.1 Log Files	73
6.3 Project Structure	74
6.4 Main Method	74
6.5 WebSocket Config	75
6.6 Echoing Handler	76
6.7 Running the application	77
6.8 Results of the application	77
6.9 Log Files of the results	79
Chapter 7 – WebSockets with STOMP in Spring Boot	80
7.1 Dependencies Required	81
7.2 Details of application.properties file	82
7.3 Project Structure	82
7.4 Main Method of the Application	83
7.5 WebSocket Config	84
7.6 Chat Controller	85
7.7 Event Listener	86
7.8 Connecting and Subscribing the client	86
7.9 Sending message from JS client	87
7.10 Results of the chat application	87
7.10.1 Start the server	87
7.10.2 UI of the chat application	88

7.10.3 New User Notification	88
7.10.4 Communications in the group	89
7.10.5 Log files	90
7.10.6 Everyday log files	90
Chapter 8 – WebSockets with STOMP using RabbitMQ as Message Broker	91
8.1 WebSocket Config	92
8.2 Results of the application	93
8.2.1 Starting the server	93
8.2.2 Checking the server connection to RabbitMQ in RabbitMQ portal	93
8.2.3 New user	94
8.2.4 RabbitMQ Monitoring	94
8.3 POM.XML	96
Conclusion	97
Bibliography	98

List of Figures	Page
Figure 1: API getUserMedia	14
Figure 2: RTCPeerConnection	15
Figure 3: RTCDataChannel	15
Figure 4: WebRTC in the Real World: Signaling	16
Figure 5: WebRTC in the Real World: Signaling with server	16
Figure 6: Signaling server	17
Figure 7: Signaling server	17
Figure 8: ICE and STUN/TURN	19
Figure 9: Peer to Peer Connection	19
Figure 10: Mesh Topology	20
Figure 11: MCU Topology	21
Figure 12: Routing	21
Figure 13: XMPP workflow	26
Figure 14: Client Server Architecture in XMPP	27
Figure 15: XMPP client server flow	27
Figure 16: HTTP workflow	33
Figure 17: WebSocket Architecture	35
Figure 18: WebSocket protocol switching	36
Figure 19: WebSocket Connection	37
Figure 20: Connecting clients to a broker	43
Figure 21: Sending messages from clients to a broker	44
Figure 22 : Subscribing clients to messages from a broker	44
Figure 23: Client messages acknowledgement	46
Figure 24: Broker commands acknowledgement	47
Figure 25: RabbitMQ implementing AMQP	50
Figure 26: Overview of AMQP Protocol	51
Figure 27: AMQP Architecture	53
Figure 28: Direct Exchange	56
Figure 29: Fanout Exchange	57
Figure 30: Topic Exchange	58

Figure 31: Header Exchange	59
Figure 32: Spring Boot framework	63
Figure 33: Main method example	65
Figure 34: Application Example	67
Figure 35: Web Starter dependency	68
Figure 36: data JPA and H2 dependencies	68
Figure 37: mail starter dependency	69
Figure 38: starter test dependency	69
Figure 39: Required dependencies	72
Figure 40: application.properties	73
Figure 41: Project structure	74
Figure 42: Main method	75
Figure 43: WebSocketConfig	75
Figure 44: Echoing Handler	76
Figure 45: Start of the application	77
Figure 46: Establishing the connection	77
Figure 47: Sending and Receiving json messages	78
Figure 48: Logging the messages	78
Figure 49: Log file	79
Figure 50: Everyday logs	79
Figure 51: Dependencies required	81
Figure 52: application.properties file	82
Figure 53: Project structure	82
Figure 54: Main method	83
Figure 55: WebSocketConfig	84
Figure 56: Chat controller	85
Figure 57: Event listener	86
Figure 58: Connecting and subscribing the JS client	86
Figure 59: Sending message from JS client	87
Figure 60: Start of the application	87
Figure 61: UI of localhost:8001	88

Figure 62: New User notification	88
Figure 63: Communications in the group	89
Figure 64: Log file	90
Figure 65: Everyday log files	90
Figure 66: WebSocketConfig	92
Figure 67: Start of the application and connection to RabbitMQ	93
Figure 68: Server connection to RabbitMQ	93
Figure 69: New user notification	94
Figure 70: New connection to RabbitMQ	94
Figure 71: Message Rate	94
Figure 72: Global Counts	95
Figure 73: Each User Queues	95
Figure 74: Download details JSON file	95
Figure 75: POM.xml	96

Introduction

Chat applications have become one of the most important parts of everyone's daily routine. Be it to chat with colleagues, friends, family and also for the discussions among the companies or organizations, and importantly for the clients to get several clarifications and also to raise their queries and issues during the company's product usage.

In this research that has been done among several protocols that are useful for creating such chat applications, there are two parts firstly the theoretical part which consists of the research of four top protocols i.e., WebRTC, XMPP, WebSockets with STOMP, and AMQP.

WebRTC (Web Real-Time Communication) is a technology that enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary.

XMPP, Short for Extensible Messaging and Presence Protocol, XMPP is an open standard that supports near-real-time chat and instant messaging by governing the exchange of XML data over a network.

WebSockets, A WebSocket is a persistent connection between a client and server. WebSockets provide a bidirectional, full-duplex communications channel that operates over HTTP through a single TCP/IP socket connection.

AMQP is a message protocol that deals with publishers and consumers. The publishers produce the messages, the consumers pick them up and process them. It's the job of the message broker (such as RabbitMQ) to ensure that the messages from a publisher go to the right consumers.

Secondly, the implementation part, where chat applications have been developed with WebSockets with having any protocol over it in Java using Spring Boot and WebSockets having STOMP over it and also adding RabbitMQ as message broker for the application.

Literature Part

Chapter 1

WebRTC

1.1 Introduction to WebRTC

WebRTC is an open framework for the web that allows Real-Time communications within the browser. It includes the fundamental building blocks for the high-quality communications on the web such as network, audio, and video components used in voice and video chat applications. WebRTC allows to build real-time communication applications for the browser.

How WebRTC different from other social apps such as skype, and face time is that WebRTC provides those features without requiring users to install any additional plugins or software other than the browser.

1.2 Major Components in WebRTC

The major components in WebRTC are JavaScript APIs. These components, when implemented in the browser, can be accessed through JavaScript APIs, enabling developers to easily implement their own RTC web app. WebRTC effort is being standardized on an API level at thew3c and at the protocol level at the IETF and is supported by Google Mozilla and opera.

1.2.1 API: getUserMedia

getUserMedia enables your application to access users' media devices. The user will see a prompt asking for permissions and after granting them, the streams of such devices will be available to be used from the code, Figure 1 is the sample code snippet for the above explanation.

```
navigator.mediaDevices.getUserMedia(streamConstraints).then(function (stream) {  
    localStream = stream;  
    localVideo.srcObject = stream;  
})
```

Figure 1: API getUserMedia

1.2.2 API: RTCPeerConnection

From the figure 2 we can understand that, once we have the user's local media streams, we create an RTC build connection object to enable audio and video communication between peers. Connection is made peer-to-peer using the SRTP protocol, this means that media goes straight to

the other browser without any storage in the middle and is encrypted in transit this enables security by default.



Figure 2: *RTCPeerConnection* (Source: *WebRTC.ventures*)

1.2.3 API: *RTCDataChannel*

Audio and video are not the only content that the WebRTC is able to transmit we can also send any kind of arbitrary data the possibilities of this include online video game chat and any kind of application that requires the exchange of information in real-time.

Figure 3 illustrates the *RTCDataChannel* API

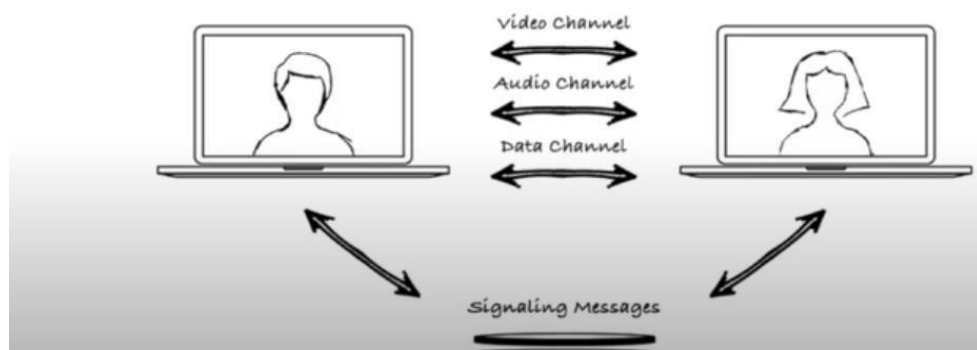


Figure 3: *RTCDataChannel* (Source: *WebRTC.ventures*)

1.3 WebRTC in the Real World

Peer to Peer(almost) and no servers involved but that is not completely true. They are some components that require a server. They are:

- Signaling
- ICE & STUN/TURN
- Media servers

1.3.1 Signaling

Figure 4 is the depiction of how WebRTC works but it has a fault. It does not work the problem is you cannot communicate with something you do not know. Both the peers need to know where the other is defined. To establish a connection, signaling enters. A signaling server sits in the middle of both the servers and allows them to exchange information about themselves, prior to initiating the call. After that exchange, both peers know how to find each other and hence can connect peer to peer. So, a signaling server is only required before the call.

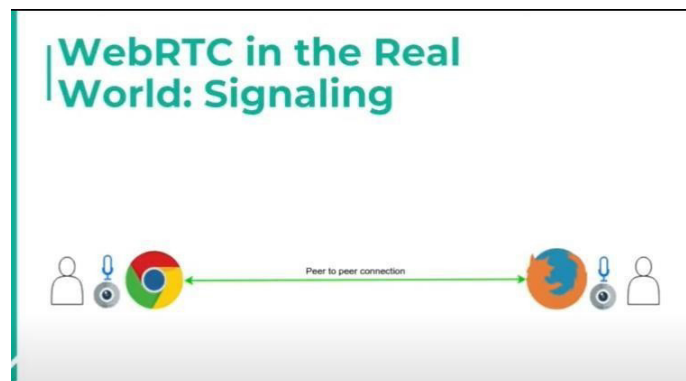


Figure 4: WebRTC in the Real World: Signaling (Source: WebRTC.ventures)

As per the figure 5, the first step is that the browsers get access to users' media devices using getUserMedia API. After that, they exchange some information about themselves using a signaling server, and then they can connect peer to peer.

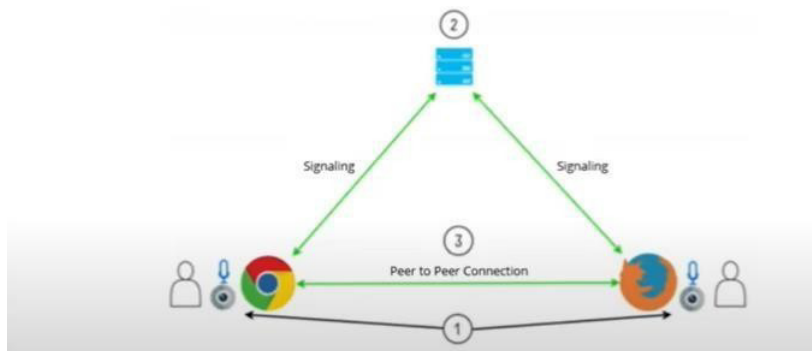


Figure 5: WebRTC in the Real World: Signaling with server (Source: WebRTC.ventures)

Signaling is the process of coordinating communication through an exchange of messages, below is the information contained in those messages

- Codecs

- Bandwidth
- Media types
- Ip Addresses

1.3.1.1 How does Signaling work

As shown in the figure 6, after getting user media and creating a WebRTC connection the browsers create an offer that contains information about itself, such an offer is sent to the signaling server to the other peer and it receives it and creates an answer, and that answer is returned to the first peer through the signaling server. These messages are forwarded using the session description protocol and contain all the information that we mentioned before. After both the peers have exchanged the information, they are able to initiate the call.

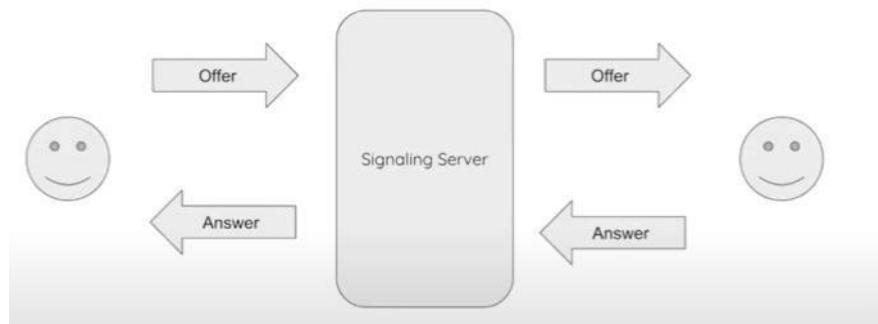


Figure 6: Signaling server (Source: WebRTC.ventures)

Figure 7 shows another process that takes place in the signaling server is the exchange of ice candidates. It works like the offer and answer mechanisms

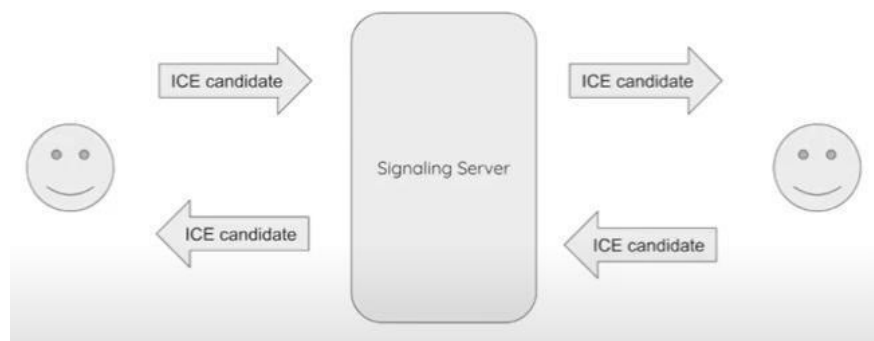


Figure 7: Signaling server (Source: WebRTC.ventures)

Signaling is not part of WebRTC standard, so you can build it the way you want. There are even

some 3rd party commercial messaging platforms that provide that service (Eg: pusher, kaazing, and PubNub).

1.3.1.2 Technologies Available for signaling servers

Some of the technologies you can use in setting up a signaling server are:

- WebSocket
- SIP
- XMPP/JINGLE

1.3.2 ICE & STUN/TURN

In the real-world peers are likely behind networks or restricting firewalls in such cases it is impossible to make a direct connection at least not without the traversal strategy that is where the ice framework and the STUN and TURN servers enter. WebRTC applications will use the ICE framework to overcome the difficulties of real-world networking. To enable this, your application must pass the ICE server and URLs to `RTCPeerConnection`.

ICE finds the most effective path to connect with peers. It tries all options available and chooses the best option that works.

1. The first path is it tries to make a connection using the host address obtained from a device's operating system.
2. If it does not work, the STUN server is used to get an external(public) network address.
3. If it also doesn't work TURN servers are used to relay traffic if a direct (peer-to-peer) connection fails.

Now we have the whole picture, as shown in figure 8, first, each client will get access to the user's media devices then an RTC paid connection is created and the signaling process begins such process consists of two concurrent tasks first both peers will exchange the offer and answer messages and also both peers will ask a STUN server for their external IP addresses and will be sending those as ice candidates to the other peer if for some reason such connection is not possible then a TURN server is used as a relay and that's the whole picture of how WebRTC work.

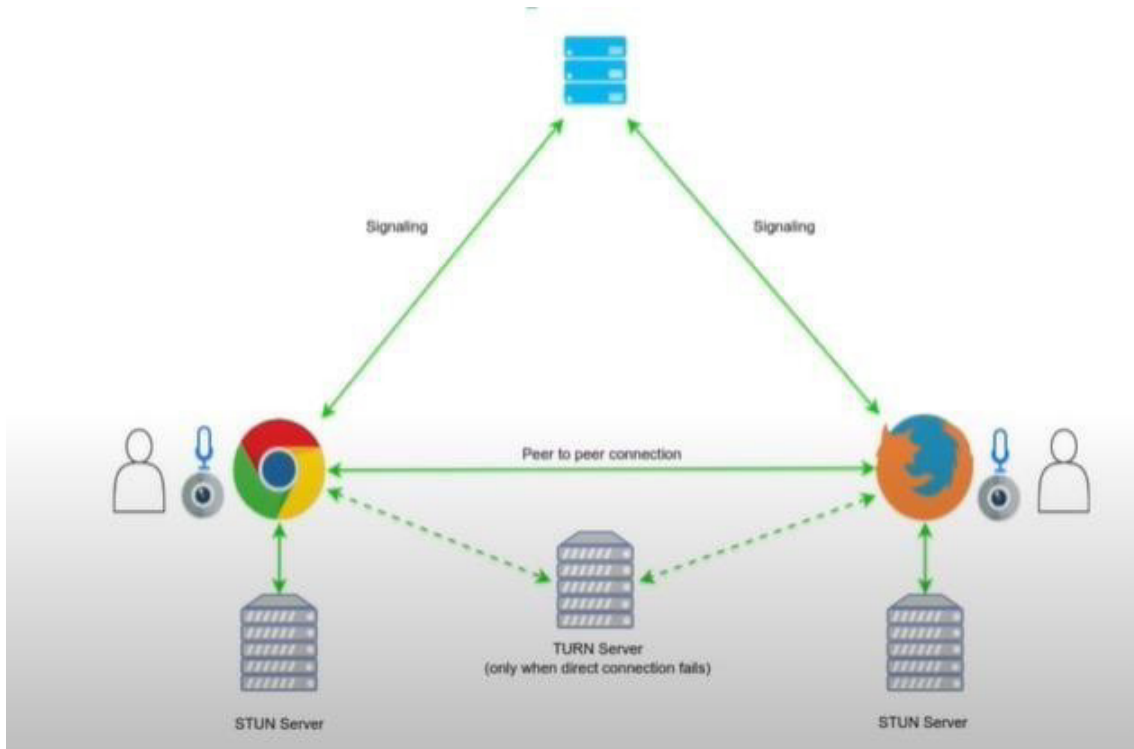


Figure 8: ICE and STUN/TURN (Source: WebRTC.ventures)

1.4 Typical Architecture Topologies for WebRTC

There are different architectures of WebRTC depending on the requirement.

Figure 9 is a simple approach for two users. It can get complicated when more users want to join.



Figure 9: Peer to Peer Connection(Source: WebRTC.ventures)

1.4.1 Mesh Topology

Figure 10 topology is called mesh and under this approach, each browser is responsible for managing as many RTC build connections as users in the call. This brings a huge resource load to each client and eventually as more users join the call will simply break.

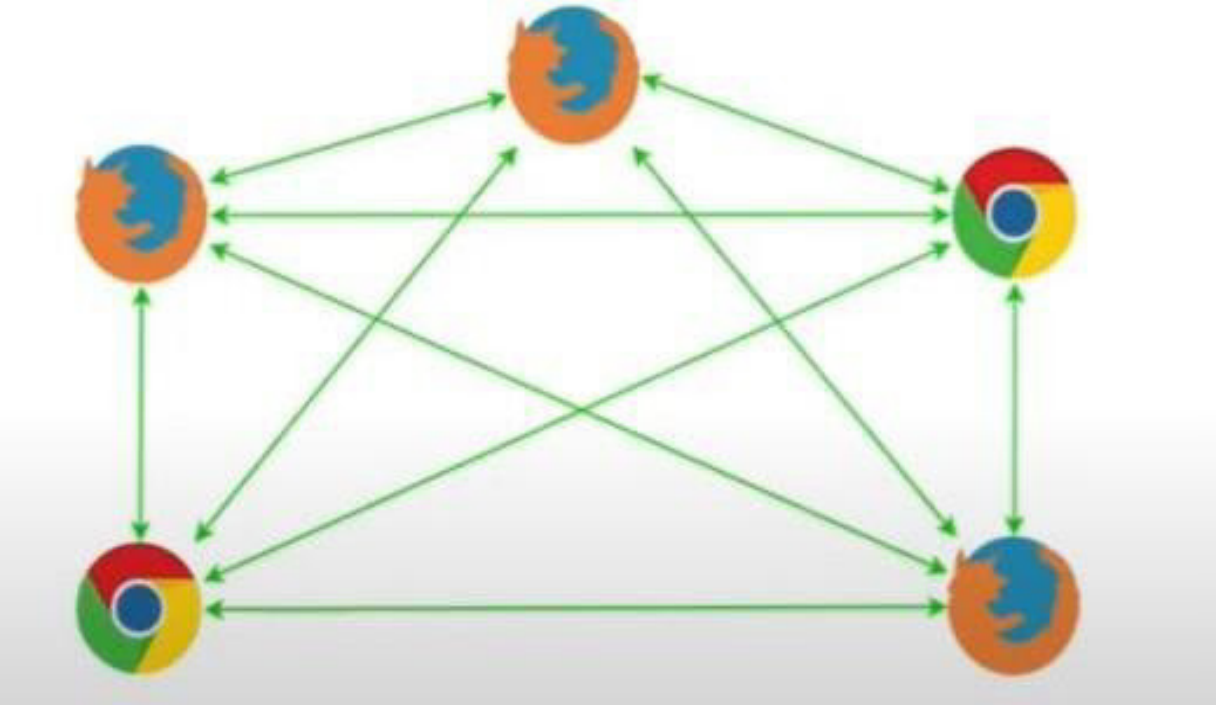


Figure 10: Mesh Topology (Source: WebRTC.ventures)

1.4.2 MCU

Figure 11 shows the next topology is the mixer also known as MCU which stands for multi- point conferencing unit. There is a server in the middle this is a media server and it's responsible for receiving all the streams from all the users and sending them to the others and a single connection here we don't have the problem of the previous approach users can join without having to worry about breaking the session the only constraint here is the media cerebral capacity and MCU media server is usually expensive.

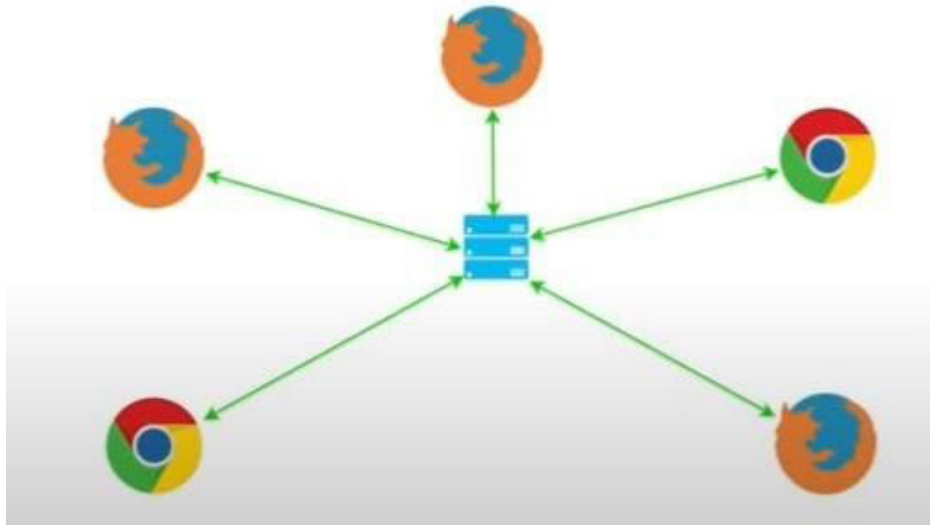


Figure 11: MCU Topology (Source: WebRTC.ventures)

1.4.3 Routing

Figure 12 shows the other topology for multi-party calls is Routing also called a selective forwarding unit. In this topology we have a media server that receives all the streams from the users however instead of making all the processing in the server it simply routes the streams to all the users this approach distributes the load between the server and the client this allows for the server to be less expensive and at the same time offer a comparable performance to MCU.

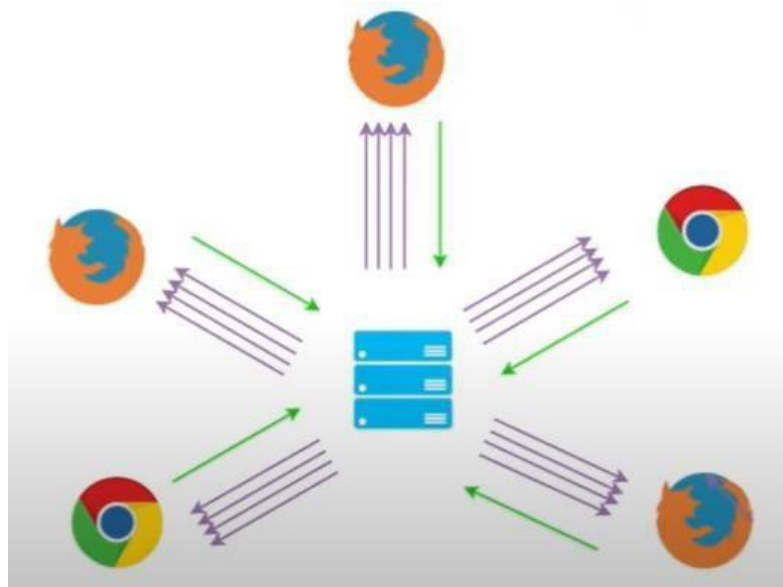


Figure 12: Routing
(Source: WebRTC.ventures)

1.5 Use Cases

WebRTC is used in audio and video calls for peer-to-peer communication over the internet. Many video chat and meeting platforms like Google Meet, Microsoft Teams, and Zoom use WebRTC.

1.6 Security Considerations

Connection is made peer-to-peer using the SRTP (Secure Real-Time Protocol) protocol, this means that media goes straight to the other browser without any storage in the middle and is encrypted in transit this enables security by default. Every WebRTC component should be encrypted and meet all the security standards and needs a secure origin like HTTPS. An unencrypted WebRTC session is not allowed by IETF (Internet Engineering Task Force).

1.7 Why WebRTC is so important

The first reason is that it enables communication in the browser and right now there is no other technology capable of doing such a thing at least not without having to install any additional software or plugin.

The second reason is that it considers some of the most common issues in today's networking picture which are NAT and firewall traversals.

The third reason is that most browsers already support it now.

Chapter 2

XMPP

2.1 Introduction to XMPP

XMPP stands for Extensible Messaging and Presence Protocol. XMPP is a network protocol to exchange data between two network endpoints. It is a standard messaging protocol used to build chat applications.

Extensible: XMPP uses XML (Extensible Markup Language) to exchange data between client and server. It is an open communication protocol designed for instant messaging (IM), presence information, near real-time messaging, and message-oriented middleware. XMPP is defined as a free open-source protocol and designed to be extensible with new features so that anyone can build their implementations. Hence it has been used for publish-subscribe systems, file transfer, and communication in embedded Internet-of-Things networks. XML is a data format that has one strength which is namespacing. Namespacing allows users to extend XMPP to do things it was not originally designed to do.

Messaging: It sends one-to-one messages (group messages). Just like any other protocol, It transfers data from client to server and vice-versa.

Presence: Can see the status of your contact's like away, busy, online, offline.

Protocol: XMPP is a protocol with a set of standards that allows systems to communicate with each other.

XMPP Standards:

XMPP standardization is managed by the Internet Engineering Task Force (IETF), which maintains the core protocol, the instant messaging subprotocol, the address format, and things of that nature.

Below are the XMPP standardizations managed by IETF

- RFC 6120: XMPP Core
- RFC 6121: XMPP IM
- RFC 7590: Use of TLS in XMPP
- RFC 7622: XMPP Address Format

The XMPP Standards Foundation (XSF) maintains all the extensions. XMPP stands for Extensible Messaging and Presence Protocol, the extensible part is what XSF manages.

Below are the extensions managed by XSF

- XEP-0045: Multi-User Chat
- XEP-0198: Stream Management
- XEP-0367: Message Attaching

2.2 History to XMPP

In 1999 XMPP was created by the Jabber open-source community to provide an open alternative to the closed instant messaging services, and it was known as Jabber then.

In 2002 the Jabber community starts turning its core protocol into an Internet Standard, IETF forms the XMPP working group. They presented the Jabber protocol to Internet Engineering Task Force (IETF). The jabber community submitted the core Jabber protocol to the Internet Standards Process and the control over jabber protocol is taken over by the IETF.

In 2004, RFC 3920, RFC 3921, RFC 3922, RFC 3923 with the XMPP standards is approved. RFC (Request for Comments) is a document published by the IETF to make design and architecture; technical standards available for everyone.

In 2008 Cisco acquires Jabber, which is why the jabber name kind of fell out of favor.

In 2011 XMPP specifications, the RFCs are superseded by RFC 6120, RFC 6121, RFC 6122. In 2014 WebSocket subprotocol RFC 7395 was created for connecting from web clients once WebSockets became a thing.

In 2015 Address format was superseded by RFC 7622 and the XMPP standard foundation started developing open XMPP extensions. There are a few other specifications like RFC 7590, which have TLS (Transport Layer Security) updates. TLS has been updated to keep up with the security requirements end-to-end kind of internationalization changes, but overall, the basic core XMPP protocol has remained largely like what it was in the late 90s.

2.3 How XMPP works

XMPP works on a client-server architecture. In XMPP a message is sent to a server first and then the server routes it to the correct client. From the figure 13, we can see that client and server are exchanging data in XML format.

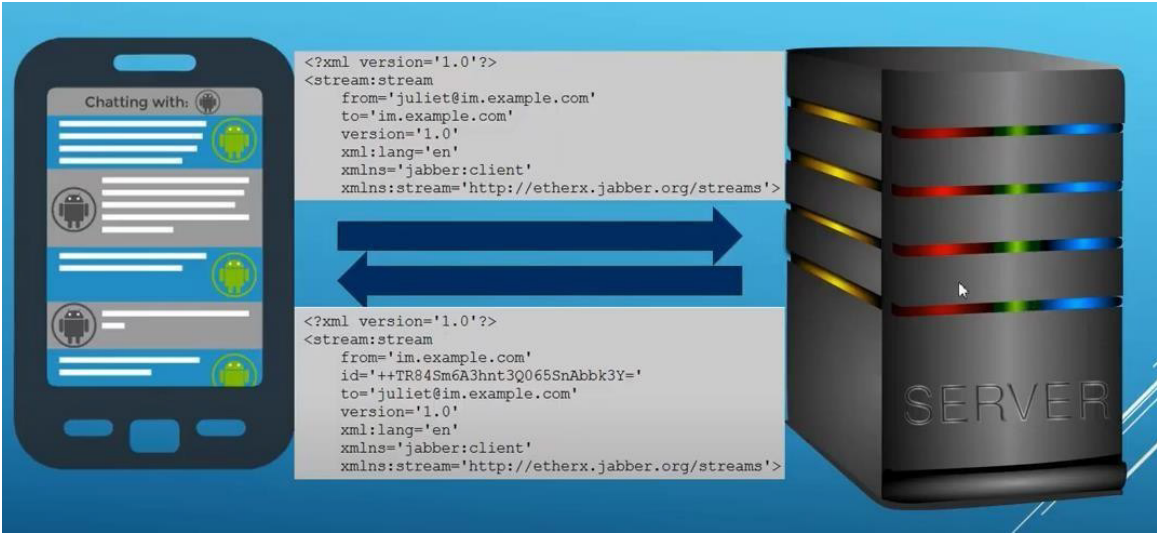


Figure 13: XMPP workflow

XMPP protocol uses XML streams to send data but not documents unlike a lot of chat protocols we are not sending individually consumable documents. From the moment we connect to the moment we disconnect we have one kind of continuous stream. By opening a tag (<) messages can be sent, and it should have a closing tag (>). But that will not break the integrity, it does not have to wait for the whole document to be sent perfectly to send the data to the other end we can just stream it.

2.4 Client Server architecture in XMPP

Communication among different XMPP servers creates a global communication network known as a "federation". From the figure 14, C1 send data to client c2, C1 is connected to server S1, and a Message is sent from C1 to S1. Whereas C2 is connected to Server 2, C1 and C2 are connected to different servers, C1 sends messages to S1, and Server S1 forwards that message to Server 2 and further forwards it to C2 from Server 2.

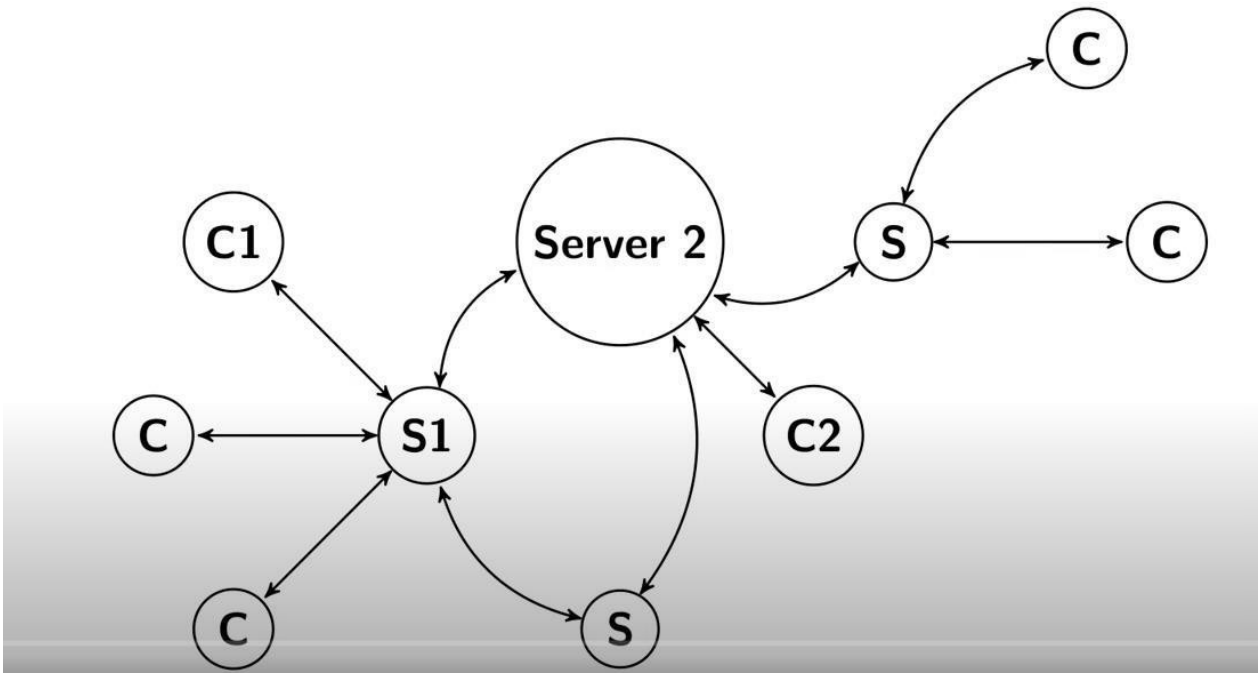


Figure 14: Client Server Architecture in XMPP

So, we end up with this broad federated network like email. A federation allows communication with a client on a different XMPP server effortlessly. So, we need an address format that supports this sort of federated network. To route, each message to the correct client XMPP uses a unique identifier called as jid. It stands for jabber id or jabber identifier. Figure 15 shows the XMPP client server flow.

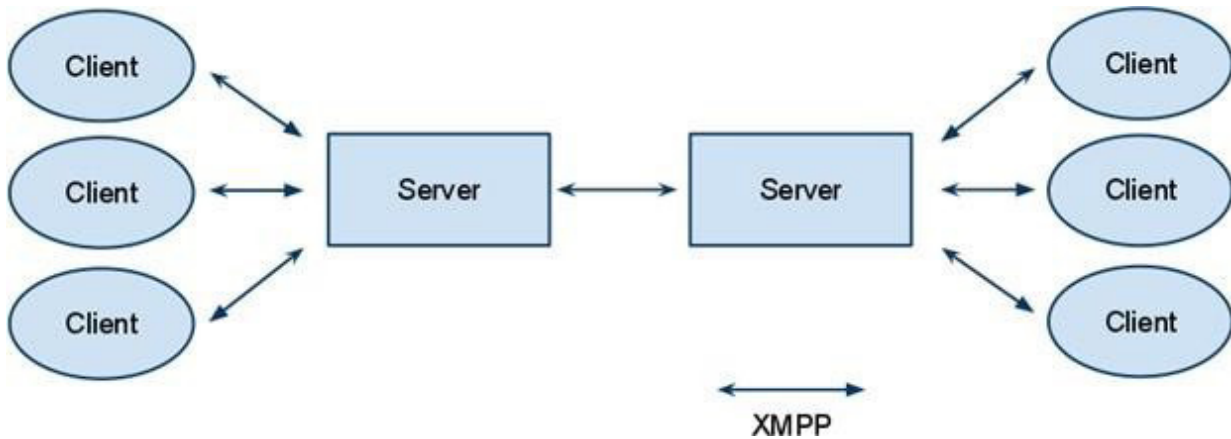


Figure 15: XMPP client server flow (Source:easyatm.com.tw/wiki/xmpp)

2.5 Anatomy of JID

The Anatomy of JID is pretty much like the email format, but here we have some extra part called resource. The unique identifier has a specific format as user@domain.com/resource. The user represents the username of the person, and the domain represents the domain of the client sending the message. The resource represents the type of device from which the message is sent for example mobile. This part is optional and is defined only when the client does not support every device.

2.6 Core protocol defined in RFC 6120 i.e., about Streams

Using this unique identifier client always initiates that connection with the XMPP server. The client ignores the TLS, and DNS for a second and sends an XML stream to the server. Once the client is identified by the server and the connection is accepted, the server then opens an additional XML stream, and this stream goes back to the client. This results in the stream of XML data bi-directionally.

There are two streams input and an output stream, and these occur over a persistent TCP connection.

As a security measure, streams are restarted when their state changes. For Example, if we are going to upgrade from plain text to TLS or from uncompressed to compressed connection streams are to be restarted from scratch throwing out all the states we had before and starting a new one inside that new layer. This sort of security precaution is advantageous, as we have a sort of stateful stream.

These streams are event-based and pipelined so this means that XMPP is an asynchronous protocol as opposed to HTTP where HTTP is synchronous when we make a request there will be a response. But in XMPP we might make any number of requests get a response back slowly. If we make three requests, then we get responses back with some other requests interspersed and the communication is entirely asynchronous.

2.7 XMPP Stanzas

As we start sending messages XMPP transmits all the little XML payloads between the client and server which are used for basic communication are called Stanzas. Stanzas are sort of basic primitive units of the XMPP stream. In XMPP stanzas are basically of 3 types and each stanza has its own purpose. These are the presence, message, and IQ stanzas and these are the only routable elements in XMPP.

2.7.1 Presence Stanza

The status of the other users can be known with the Presence stanza. Can see your contact's onlinestatus like away, busy, online, offline can be known.

2.7.2 Message Stanza

Chat messages can be shared between the users using the Message stanza. Information can be sent from one XMPP server to another XMPP seer without requiring a response. They are not just used for chat messages but also where you only send a message to one address and there is no acknowledgment. Any sort of server does not confirm that it was delivered or anything like that you just send it you forget about it so they are kind of useful for anything that doesn't require a response like chats, alerts, logging.

2.7.3 IQ Stanza

IQ stands for information query is the other type of stanzas they are also one-on-one you only send them to a single recipient however unlike messages these are acknowledged by the recipient. The recipient always sends back something that may be a message or iq saying we do not understand the payload you just sent. so, this gives you the option that you do not get the response within a certain amount of time you can retry because you know you are expecting a response. Each IQ request stanza has an ID value that corresponds to the response stanza.

2.8 XMPP features

2.8.1 Asynchronous Protocol

XMPP streams are event-based and pipelined. This means XMPP is an asynchronous protocol, we might make any number of requests, and we get responses back slowly. There is no immediate response to every request we make.

2.8.2 Persistent Connection

The XMPP server and client have a persistent TCP connection, so once there is a request from the client the connection remains open and persistent, and the exchange of data can be done without re-establishing the connection for the other request. Since the connection is open and persistent it has lower latency.

2.8.3 Decentralization

In XMPP the clients do not communicate with each other directly, there is a server in between. Anyone can have their own XMPP server, there is no one main server. XMPP network architecture is like email.

2.9 Few Common Extensions in XEP series

2.9.1 XEP-0280 : Message Carbons

It copies incoming messages to various resources like to the mobile, desktop and have a consistent view from every device. In the same way it also copies all the outgoing messages to other connected devices to have consistent view from every device. It is going to be replaced Message Archive Management (MAM) though both these extensions are in use as of now.

2.9.2 XEP-0310 : Message Archive Management (MAM)

MAM is just a chat history; it stores outgoing messages on the server so that the new resources can access history. Even if the client is offline for bit once they are online can catch up and receive all the messages that were sent when they are offline. MAM can replace Message Carbons sooner.

2.9.3 XEP-0352 : Client State Indication (CSI)

Clients indicate when they become inactive or active with simple top-level elements. With respect to the client's status server does what is required with the data. When the client is offline it does not send presence or typing notifications instead it starts sending push notifications.

2.9.4 XEP-0268 : Mobile Considerations

This extension attempts to save users battery by implementing CSI. This XEP allows to send or receive as much as you can send at once but when it detects the data is already sent or received it allows the modem to go back to sleep.

2.9.5 XEP-0198

Stream Management: Stream management lets you do the stream resumption; it allows very fast reconnects. It also does stanza acknowledgements like tracking of data packets. It tracks the number of packets that have been sent and resuming something which is missed is possible since there are data packets tracking.

2.10 Services built on top XMPP

- Nintendo Switch notifications
- WhatsApp
- Zoom
- Cisco Jabber
- Google cloud Print

Chapter 3

Web Sockets

3.1 What is HTTP

HTTP is a protocol belonging to the application layer used to exchange data between client and server. It is used to deliver contents videos, audio, images, etc. HTTP stands for Hypertext Transfer Protocol, and it is a TCP/IP-based protocol. We can think of a protocol as a language or mechanism for communication, HTTP is a way to communicate or exchange information online, and the stuff we transfer here in HTTP is called hypertext and the language of this hypertext is HTML.

As we can observe in the figure 16, The client sends an HTTP request, and the server returns an HTTP response where clients can be browsers, programs, or devices and servers are computers on the cloud.

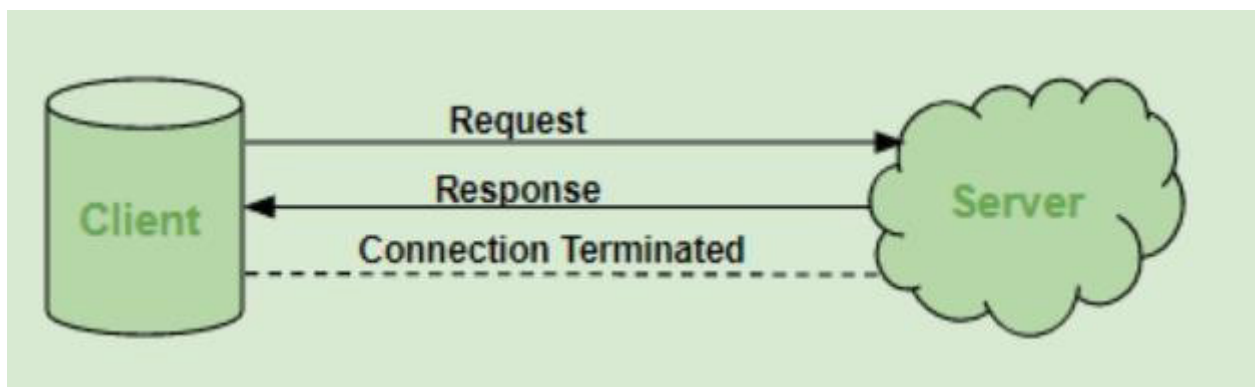


Figure 16: HTTP workflow (Source: geeksforgeeks.org)

HTTP protocol exchanges any kind of data if both the client and server understand it. Communication is done by requests and responses. A client sends an HTTP request to the server, Server receives and processes the request and returns an HTTP response to the client. When the request is sent, after DNS resolution and IP address are identified, a TCP connection is established between the client and server. HTTP runs on a TCP connection and after making the request transfer of the data packet is guaranteed. The client disconnects from the server, then when response is the ready server will re-establish the connection and deliver the response. In HTTP we have different methods to transfer information, the most common methods are GET, PUT, POST, DELETE.

3.2 Drawbacks in HTTP

- HTTP is a unidirectional protocol, when there is a request from the client after getting the response, the connection is closed.
- HTTP is a stateless protocol
- HTTP is slower because it requires transferring a lot of data in every single request as it is a stateless protocol.

The main purpose of WebSockets is to provide bi-directional communication which is the major drawback of HTTP.

3.3 What are WebSockets

Websockets allow asynchronous bidirectional and provide full-duplex communication between a client and a server. Communication is done by requests and responses. A client sends an HTTP request to the server, after DNS resolution and IP address are identified, a TCP connection is established between the client and server.

3.3.1 WebSocket Architecture

As we can understand from figure 17, to establish a WebSocket connection the client should send an HTTP request to upgrade the HTTP protocol to the WebSocket protocol and the server sends the confirmation after the upgrade and the connection between the client and the server is open and persistent. They keep exchanging the messages or frames over the connection until onside decides to close the connection.

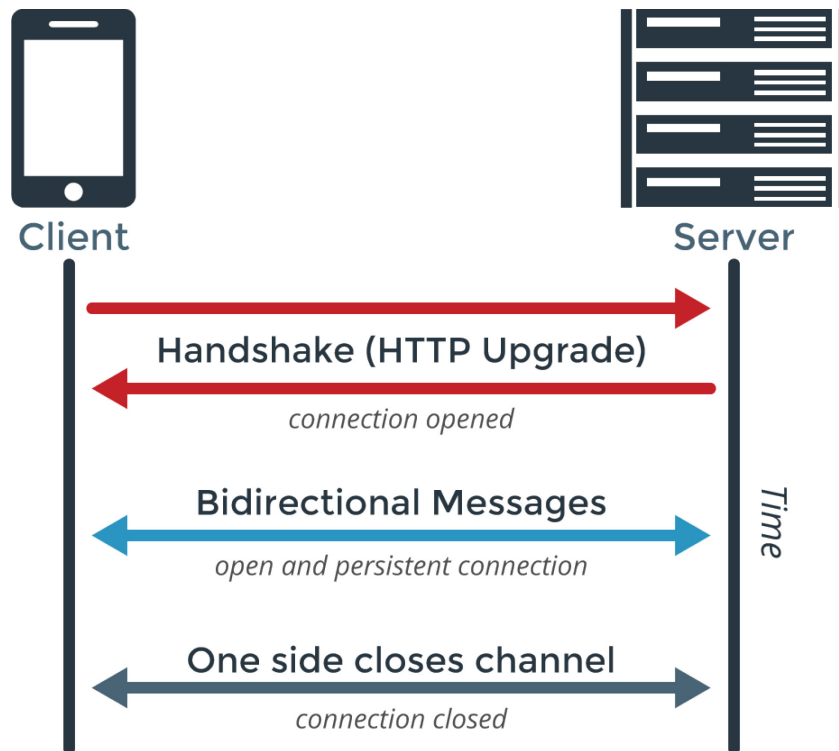


Figure 17: WebSocket Architecture (Source: medium.com)

3.3.2 Protocol Handshake

To establish a WebSocket connection between client and server, the client sends a WebSocket handshake request, for which the server returns a WebSocket handshake response. This handshake is necessary to keep the connection open as a WebSocket connection and to make sure that both client and server are speaking to the same protocol. When both the client and server have sent their handshakes and if was successful, then an exchange of data can be done bi-directionally between both.

The Websocket handshake from the client looks as follows:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat,
```

Sec-WebSocket-Version: 13

To have a handshake with the server, the client sends an HTTP GET request to the WebSocket path. It also sets headers like Connection: Upgrade, Upgrade: Websocket, Sec-WebSocket-Key: <random_key>

Since the request sent to establish a WebSocket connection is HTTP itself, the single port can be used by both HTTP clients talking to that server and WebSocket clients talking to that server. In the end, the WebSocket client's handshake is an HTTP Upgrade request. In the handshake header files can be sent in any order, the order in which these header files are received is not significant. The first request that we make is an HTTP normal get request, the request is an HTTP 1.1 which is going to establish a persistent connection between the two. There is an upgrade header in the request which is a powerful thing. The server consumes the request, and it knows the client is trying to upgrade this request to WebSocket and replies with the status code 101 which says switching protocol. According to figure 18 the WebSocket endpoint is identified using the request URI of the GET method.

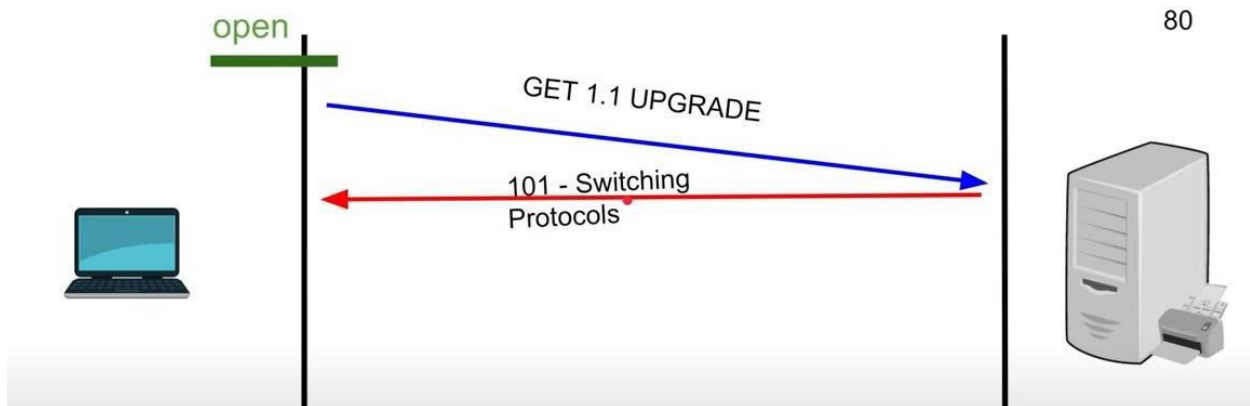


Figure 18: WebSocket protocol switching

The client handshake request has a host header so that both the client and the server can verify that they agree on which host is in use. To avoid unauthorized use the origin header is used. As the server is informed about the origin header, if the request is from an unauthorized origin, it rejects the connection by sending an appropriate HTTP error code.

To avoid other than WebSocket connections the server should send an acknowledgment to the client that it received a handshake request. The server must send the response handshake to the

client to prove that handshake is received. The server must concatenate the Sec-WebSocket-Key value from the client with Globally Unique Identifier (GUID). This value would be then returned by the server in the Sec-WebSocket-Accept header field.

The response handshake from the server looks as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade:      WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=Sec-
WebSocket-Protocol: chat
```

From the figure 19 the response handshake from the server is simpler than the request handshake from the client. The response handshake has an HTTP status code 101. If the status code is anything other than 101 indicates that the handshake is not completed. The Upgrade and Connection headers complete the HTTP upgrade. If the Sec-WebSocket-Accept header value does not match the expected value, or if the HTTP status code is not 101, the WebSocket connection will not be established, and WebSocket messages/frames will not be sent.

Closing the handshake is much simpler than opening the handshake, either client or server can close the connection.



Figure 19: WebSocket Connection (Source: geeksforgeeks.org)

Browser – WebSockets

To set up a connection from the browser, create a new WebSocket object with the host/path to connect to and set up the server to accept WS requests on that path.

3.3.3 Relationship to TCP and HTTP from WebSockets

The WebSocket Protocol is an independent TCP-based protocol. The relation between WebSocket and HTTP is that the initial request sent by the client is interpreted as an HTTPrequest.

Websocket URI's

ws://www.google.com

wss://www.google.com

The URI is like HTTP:// and HTTPS://. Prefix ws:// indicates WebSocket connection and wss:// indicates Websocket secure connection.

Websockets feature an HTTP-compatible handshake, thus allowing HTTP servers to share their default HTTP and HTTPS ports. Port 80 is used for regular WebSocket connections and port 443 is used for secured WebSocket connections in WebSocket Protocol.

3.3.4 Security Considerations

WebSocket protocol uses the Origin header to restrict which web pages can contact a WebSocket server when the WebSocket Protocol is used from a web page. Data transfer over the WebSocket protocol is done in plain text, like HTTP. Therefore, this data is vulnerable. It is better to use tokens or similar protection mechanisms to authenticate the WebSocket connection when sensitive data is being transferred over the WebSocket. Authenticate requests are made to the HTTP endpoint /authenticate/token with the internal authentication token passed in the header of the request. A temporary external authentication token is generated by the server, and it is stored in the Authentication Cache, and it is returned to the client.

The client sends a handshake request with the external authentication token handshake endpoint URL. The server validates the token with the authentication cache. The handshake is established if the token is valid, and the HTTP request upgrades to the WebSocket connection. The client is now authenticated and can have bidirectional communication.

3.3.5 Use Cases

3.3.5.1 Gaming Applications

In Gaming applications data is continuously received from the server without refreshing the UI and the effect is seen on the screen. The UI gets automatically affected without establishing a new connection. So, WebSocket is widely used here.

3.3.5.2 Chat Applications

In chat applications, the WebSocket connection is established only once for exchange, publishing, and broadcasting the message among the users. The same WebSocket connection is reused for sending and receiving one-to-one messages.

3.3.5.3 Live feed

WebSocket is widely used in live feed where data should be continuously published at the client end from the backend server. Since the connection is already established data is continuously pushed into the WebSocket connection which is already established.

3.3.6 Advantages of WebSockets

The main advantage of WebSocket is it allows bi-directional communication/ full-duplex communication between the client and server.

WebSocket is a stateful protocol, it does not require transferring a lot of data in every single request as the connection is persistent open till either client or server decides to close the connection.

WebSocket has low latency and hence it is faster than HTTP.

WebSocket can be used where continuous data streaming is required. HTTP can be used when the data fetching is only once.

3.4 Introduction to STOMP

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol.

STOMP is a simple interoperable protocol designed for asynchronous message passing between clients via mediating servers. It defines a text-based wire-format for messages passed between these clients and servers.

STOMP has been in active use for several years and is supported by many message brokers and client libraries.

STOMP is a frame-based protocol, with frames modelled on HTTP. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based but also allows for the transmission of binary messages. The default encoding for STOMP is UTF-8, but it supports the specification of alternative encodings for message bodies.

A STOMP server is modelled as a set of destinations to which messages can be sent. The STOMP protocol treats destinations as opaque string, and their syntax is server implementation specific. Additionally, STOMP does not define what the delivery semantics of destinations should be. The delivery, or “message exchange”, semantics of destinations can vary from server to server and even from destination to destination. This allows servers to be creative with the semantics that they can support with STOMP.

A STOMP client is a user-agent which can act in two (possibly simultaneous) modes:

- as a producer, sending messages to a destination on the server via a SEND frame
- as a consumer, sending a SUBSCRIBE frame for a given destination and receiving messages from the server as MESSAGE frames.

3.4.1 Latest version updates

Latest version as per the date of this report is STOMP 1.2.

STOMP 1.2 is mostly backwards compatible with STOMP 1.1. There are only two incompatible changes:

- it is now possible to end frame lines with carriage return plus line feed instead of only line feed
- message acknowledgment has been simplified and now uses a dedicated header

Apart from these, STOMP 1.2 introduces no new features but focuses on clarifying some areas of the specification such as:

- repeated frame header entries
- use of the content-length and content-type headers
- required support of the STOMP frame by servers
- connection lingering
- scope and uniqueness of subscription and transaction identifiers
- meaning of the RECEIPT frame about previous frames.

3.4.2 Message brokers available for STOMP

STOMP is a messaging protocol because clients can produce messages (send messages to a brokerdestination) and consume them (subscribe to and receive messages from a broker destination).

STOMP is an interoperable protocol because it can work with multiple message broker, and clientswritten in many languages and platforms.

There are several brokers that can be used with STOMP, few of them are ActiveMQ,RabbitMQ, HornetQ, OpenMQ, etc.

3.4.3 Connecting clients to a broker

3.4.3.1 Connect

From figure 20, we can understand that to connect to a broker, a client sends a CONNECT frame with two mandatory headers:

- accept-version — the versions of the STOMP protocol the client supports
- host — the name of a virtual host that the client wishes to connect to

To accent the connection, the broker sends to the client a CONNECTED frame with themandatory header:

- version — the version of the STOMP protocol the session will be using

3.4.3.2 Disconnect

From figure 20, we can understand that a client can disconnect from a broker at any time by closing the socket, but there is no guarantee that the previously sent frames have been received bythe broker.

To disconnect properly, where the client is assured that all previous frames have been received by the broker, the client must:

1. send a DISCONNECT frame with a receipt header
2. receive a RECEIPT frame
3. close the socket

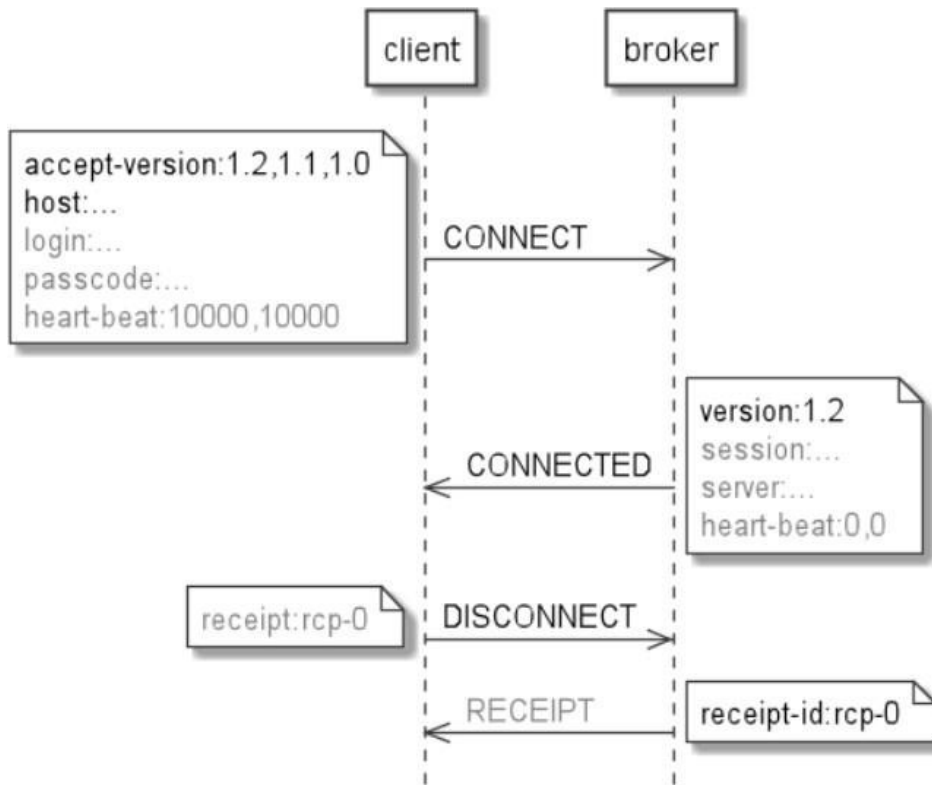


Figure 20: Connecting clients to a broker (Source: medium.com)

3.4.4 Sending message from clients to a broker

From figure 21, we can understand that to send a message to a destination, a client sends a SEND frame with the mandatory header:

- Destination — the destination to which the client wants to send

If the SEND frame has a body, it must include the content-length and content-type headers.

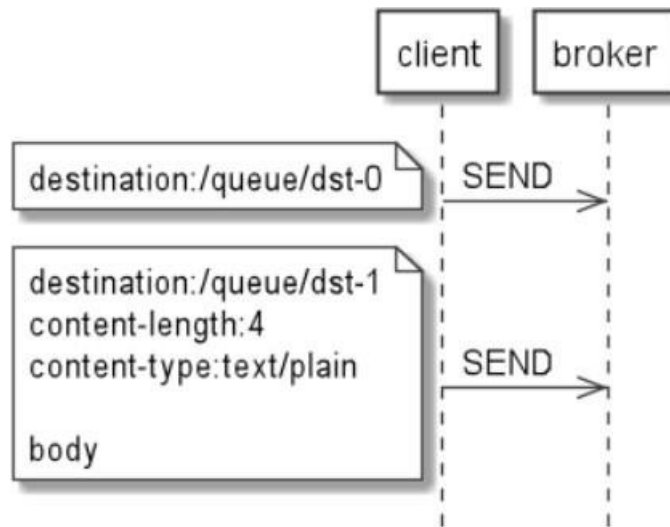


Figure 21: Sending messages from clients to a broker(Source: medium.com)

3.4.5 Subscribing clients to message from a broker

3.4.5.1 Subscribe

From figure 22, we can understand that to subscribe to a destination a client sends a SUBSCRIBE frame with two mandatory headers:

- destination — the destination to which the client wants to subscribe
- id — the unique identifier of the subscription

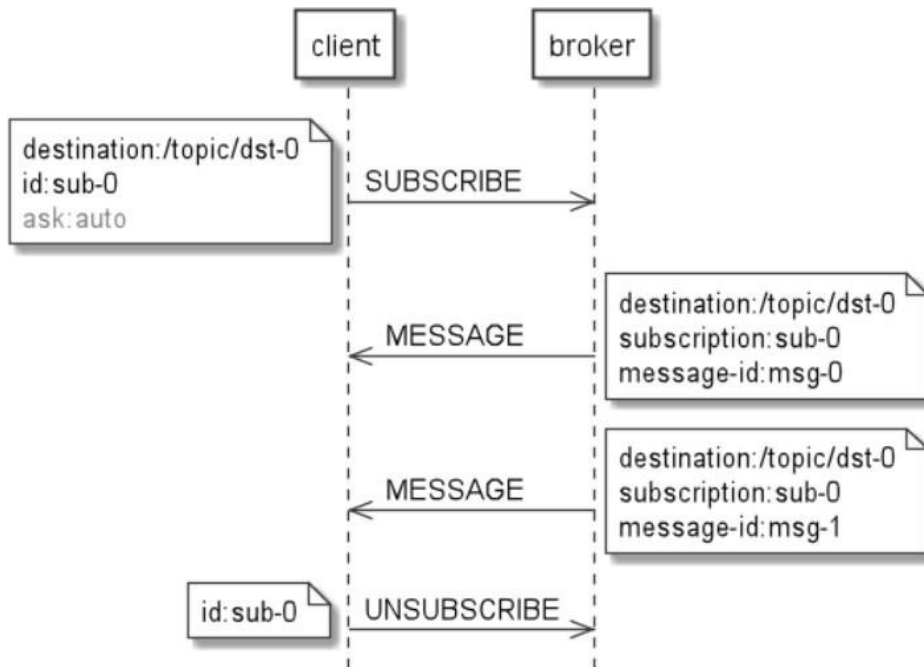


Figure 22 : Subscribing clients to messages from a broker(Source: medium.com)

3.4.5.2 Message

To transmit messages from subscriptions to the client, the server sends a MESSAGE frame with three mandatory headers:

- destination — the destination the message was sent to
- subscription — the identifier of the subscription that is receiving the message
- message-id — the unique identifier for that message

3.4.5.3 Unsubscribe

To remove an existing subscription, the client sends an UNSUBSCRIBE frame with the mandatory header:

- id — the unique identifier of the subscription

3.4.6 Acknowledgement

To avoid lost or duplicated frames, if a client and a broker are parts of a distributed system, it is necessary to use frames acknowledgment.

3.4.6.1 Client message acknowledgement

The SUBSCRIBE frame may contain the optional ack header that controls the message acknowledgment mode: auto (by default), client, client individual.

When the acknowledgment mode is auto, then the client does not need to confirm the messages it receives. The broker will assume the client has received the message as soon as it sends it to the client.

When the acknowledgment mode is client, then the client must send the server confirmation for all previous messages: they acknowledge not only the specified message but also all messages sent to the subscription before this one.

When the acknowledgment mode is client-individual, then the client must send the server confirmation for the specified message only as shown in figure 23.

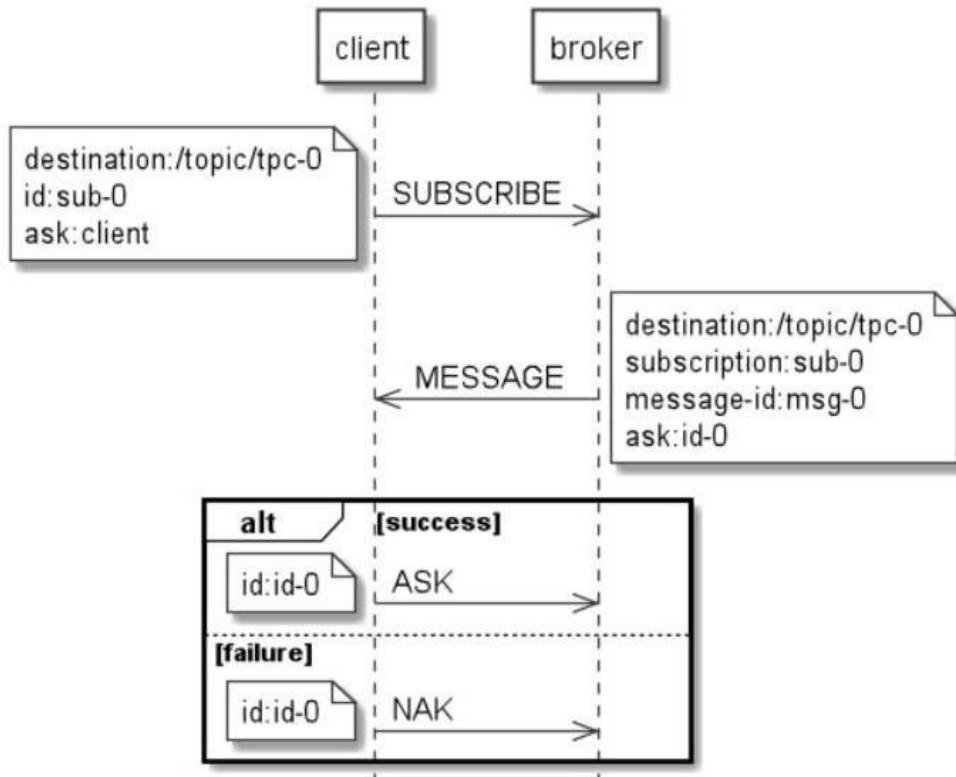


Figure 23: Client messages acknowledgement(Source: medium.com)

The client uses an ACK frame to confirm the consumption of a message from a subscription using the client or client-individual acknowledgment modes. The client uses a NACK frame to negate the consumption of a message from a subscription. The ACK and NAK frames must include the id header matching the ack header of the MESSAGE frame being acknowledged.

3.4.6.2 Broker Commands Acknowledgement

As we can see from the figure 24, a broker sends a RECEIPT frame to a client once the broker has successfully processed a client frame that requests a receipt. The RECEIPT frame includes the receipt-id header matching the receipt header of the command being acknowledged.

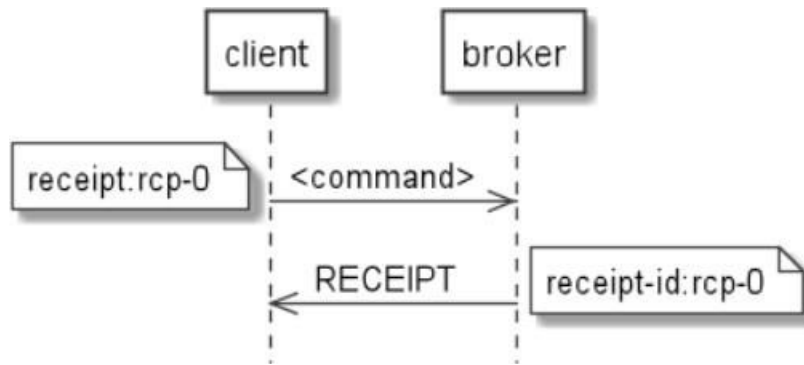


Figure 24: Broker commands acknowledgement (Source: medium.com)

3.4.7 Examples and Implementations of STOMP

In the implementation part good examples have been explained with implementations of STOMP with default broker and RabbitMQ.

Chapter 4

AMQP

4.1 Introduction to AMQP

AMQP is an application layer protocol for Message Oriented Middleware. MOM allows the exchange of messages between any distributed system irrespective of the technology used. AMQP stands for advanced messaging queuing protocol. In AMQP exchange of messages between the producer and consumer over TCP connections is done through the message brokers. AMQP standardizes messages using producers, brokers, and consumers.

4.2 How AMQP works

AMQP protocol deals with the producers which produce the message and consumers who receive the message. We have a producer, and a consumer, and there is exchange and queue in between. The message broker consists of this exchange and queue. Message broker decouples the communication between producers and consumers. The producer sends a message to the exchange then depending on the type of exchange the message is directed to the queue. The producer sends a message to the exchange, depending on the type of exchange the message will be sent to the queue and then the queue further sends the message to the consumer. The message broker which contains Exchange and queue will make sure that the message is received. We have multiple queues in the complex application. Either producers or consumers can bind the messages with the queues and make it available for the consumers. The queues can be either public or private and the messages are kept buffering in the queues till the consumers consume it.

4.3 Consumer driven Messaging Approach

AMQP follows consumer driven messaging approach. In a traditional point to point communication between producer and consumer, the producer will publish a message and it will be sent to queue and producer should be aware of which queue is going to receive this message. Here only one consumer will receive the messages, the queue acts as an end point to only one consumer. In traditional publish subscribe approach the queue acts as an endpoint to any number of consumers. These consumers can access the queues and search for their own copies. Some unique messages are shared with only few consumers in a round-robin fashion. In AMQP any kind of exchange of data is possible, data can be transferred between the producer and consumer

it will allow the consumers to search all the queues for their messages that are to be received. AMQP allows different patterns to exchange data based on the exchange type.

4.4 Message Broker

Figure 25 explains that, when there is exchange of data between two applications messages are not directly sent between producer and consumer. It is required that both the applications should know each other, and they must be online at the same time. It becomes more complex when the number of applications that share the information. But by using the message broker the message producers do not need to know about the consumers. Using message broker between the producers and consumers act as decoupling layer as the messages are not directly sent to the consumers. The message producing applications does not need to know about the receiving applications as the messages are sent to third party message broker. The message broker provides exchange type and queue, either consumers or producers create the queues and bind them to the exchange to direct messages to the respective consumers. RabbitMQ is one message broker that widely implements AMQP protocol.

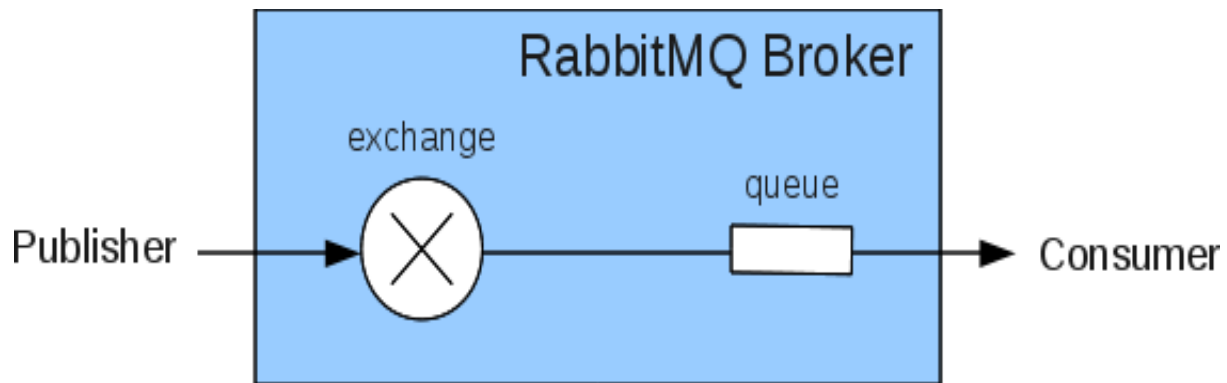


Figure 25: RabbitMQ implementing AMQP(Source: rabbitmq.com)

4.5 Overview of AMQP Protocol

Figure 26 is the overview of AMQP protocol.

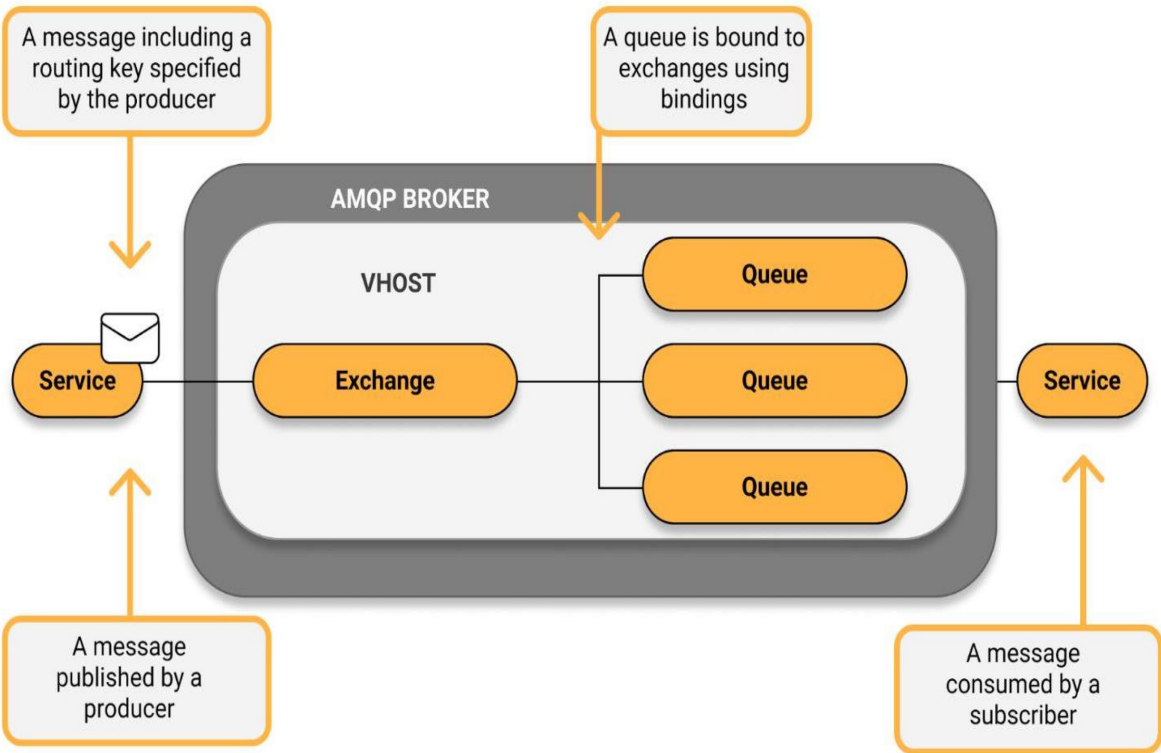


Figure 26: Overview of AMQP Protocol(Source: cloudamqp.com)

4.5.1 Components of AMQP

4.5.1.1 Message Queues

Messages are sent to queues by the exchange and these messages are kept buffered until consumers receives it. Consumers can reject the messages or return the messages to the queues as the AMQP is a full duplex communication protocol. A queue can be shared by one or more consumers. Producers send the message to the exchange and bind it to the queue and make it available for the consumers to consume it or it will just send messages to the exchange, and it is consumers responsibility to create queues and bind the message to the queue. Temporary queue scan be created for the private message exchange and the message queues can be deleted once the application is disconnected.

4.5.1.2 Exchange and Exchange types

Exchange lies between producers and consumers, and it receives messages from the producers along with the routing key. Once the message is received depending on the exchange type message is directed to the respective queue.

4.5.1.3 Binding

Binding is a link to bind a queue to an exchange. While sending a message a routing key should be sent along by the producer while publishing the message. The Exchange compares the two keys depending on the type of the exchange. Each exchange type has a different kind of algorithm to direct messages to the queues.

4.5.1.4 Message and Content

A message consists of information that must be exchanged with the other application. This information is wrapped into a message so that it becomes transportable and exchanged over a TCP connection. The message consists of the data to be exchanged, message headers and a routing key which must be compared with the binding key at the exchange. All this helps the message to be routed and delivered to the correct queue and consumed by the correct consumer.

4.5.1.5 Connection

Exchange of data in AMQP is done over the TCP connection.

4.5.1.6 Channel

When a TCP connection is established between two AMQP servers a virtual connection is established called as channel through which exchange of information is done. One single connection can have multiple channels in it as it is a multiplexed connection.

4.5.1.7 Virtual hosts

Virtual hosts are used to host multiple domains on a single server. It lets different domains to share the resources like memory and CPU cores. It provides specific privileges to specific users to access applications by setting user permissions.

4.6 AMQP Architecture

We can clearly understand from figure 27 that AMQP architecture mainly involves Publishers/Producers, Exchange, Queues, and Subscribers/Consumers.

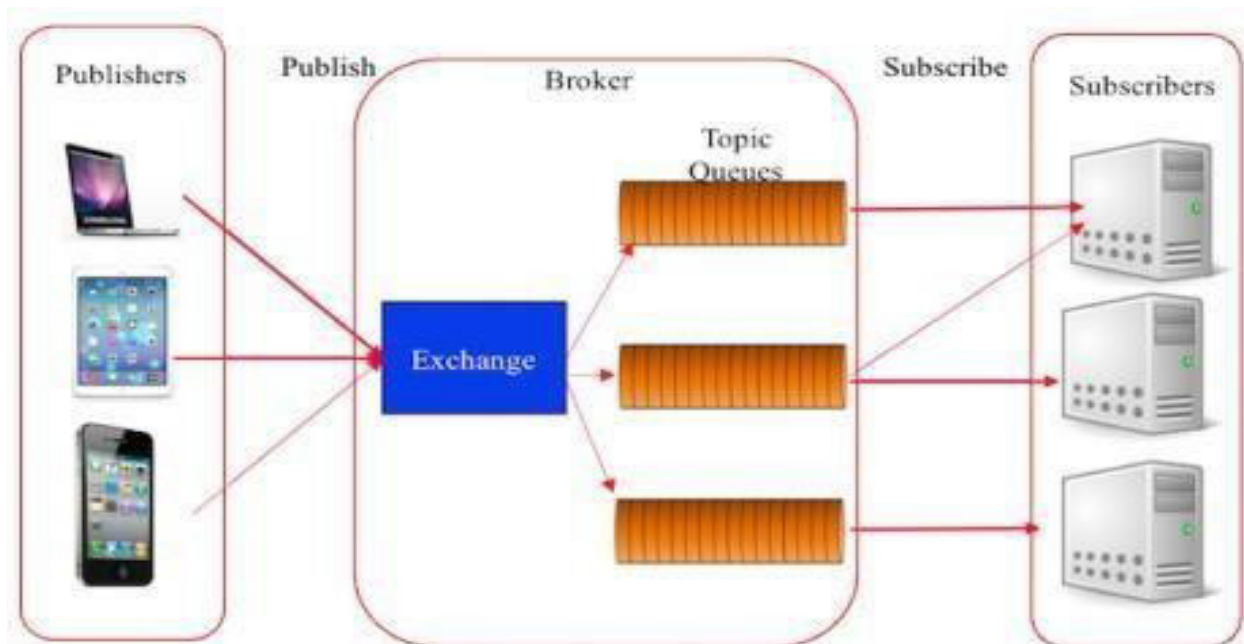


Figure 27: AMQP Architecture(Source: researchgate.net)

4.6.1 Producers

Publishers or Producers are the clients that create messages which are then given to brokers. The process of sending messages by the Producers/Publishers to the exchange is called publishing.

4.6.2 Exchange

Publishers deliver messages to Exchange. These messages contain routing keys that are used by the exchange module to route them. Queues created either by the consumers or by the publishers

are bind with exchange. When the exchange receives the message, depending on the type of exchange it will send it to the selected queue.

4.6.3 Routing Key

The messages which are sent by the producer to the exchange contains routing key. The exchange compares the binding and routing key depending on the type of exchange and the messages are further directed to the queues and the consumers browse for the required messages in the queues. It is responsibility of the producer to send a routing key when publishing a message.

4.6.4 Consumers

The concept of AMQP is producer emits messages to exchange and consumer receives messages from the queue. Consumers create queues and bind them to an exchange with a binding key.

4.6.5 Subscriptions

AMQP protocol uses a message broker as a decoupling layer between message producers and consumers. The message brokers consist of exchange and queues. Consumers subscribe to the exchange by creating queues and binding them to the exchange. There are many differences between exchange and queue. Queues are basically used to queue the messages and are buffered till the consumers receive it. Whereas an exchange sends messages to the queue when the messages are published but not store them and if there is no queue available to store the message the messages will be removed. Since the queues allow the messages to buffer the message producer need not be online or stay connected till the message is received by the consumer.

The message will be available in the queue even if the message producer is not connected.

4.6.6 Publishing

A publisher can send messages using different publishing strategies. The first way is that message producing application will send a message to the application and the message consuming application will create their own queues and subscribe to the exchange. We can use different exchange types depending on how the message must be distributed. When there are no consumers subscribed to the exchange, then the messages sent to the exchange are no longer available.

Exchange does not support storage of messages. If the messages must be stored even if there are no message consuming applications, then we need to send the messages to the queues. Publishers can directly send messages to the queues by creating queues and subscribing them to an exchange. The messages sent to the exchange are directed to the queue depending on the exchange type and consumers will receive them. When the published messages are to be stored on the broker even if the consumers do not receive them, this type of publishing is used. Even this method allows consumers to subscribe to the exchange and receive messages. We can publish messages only to a one queue by binding this queue with an exchange using an exclusive binding key. This can be done with direct exchange which is one of the exchange types. This does not let another queue to bind with the exchange to receive those messages.

We can use fanout exchange or topic exchange to bind a queue with the exchange and to publish message to the queue and these exchange type lets consuming applications to create queues and bind with the exchange to receive the published messages.

Message Distribution to the queue depends on the exchange type. There are four different types of Exchange: Fanout, Direct, Topic, Header.

4.6.6.1 Direct Exchange

Direct Exchange sends messages to the queues only when the binding key is the same as the routing key. The routing key is added to the message by the producer while sending a message to the producer to specify a routing key. The routing key and binding key must be the same to send the message to the queue. The exchange sends messages only to the one queue; hence it supports exclusive binding of queues with the exchange with the exclusive binding key. It is like topic exchange where it allows partial match of message to transfer the data but in direct exchange it is not possible.

From figure 28, order-create and order-create-log are the binding keys, when a message is sent with the order-create routing key, the routing key and binding key are the same, so the message is now sent to the order_create_queue. If there is no matching routing key to the binding key, then that message will not be sent to any queue. If the routing key is matching with more than one binding key, then the message will be broadcast to all the matching queues. Since the direct exchange allows exclusive binding of queue with the exchange there is only one recipient who receives these messages.

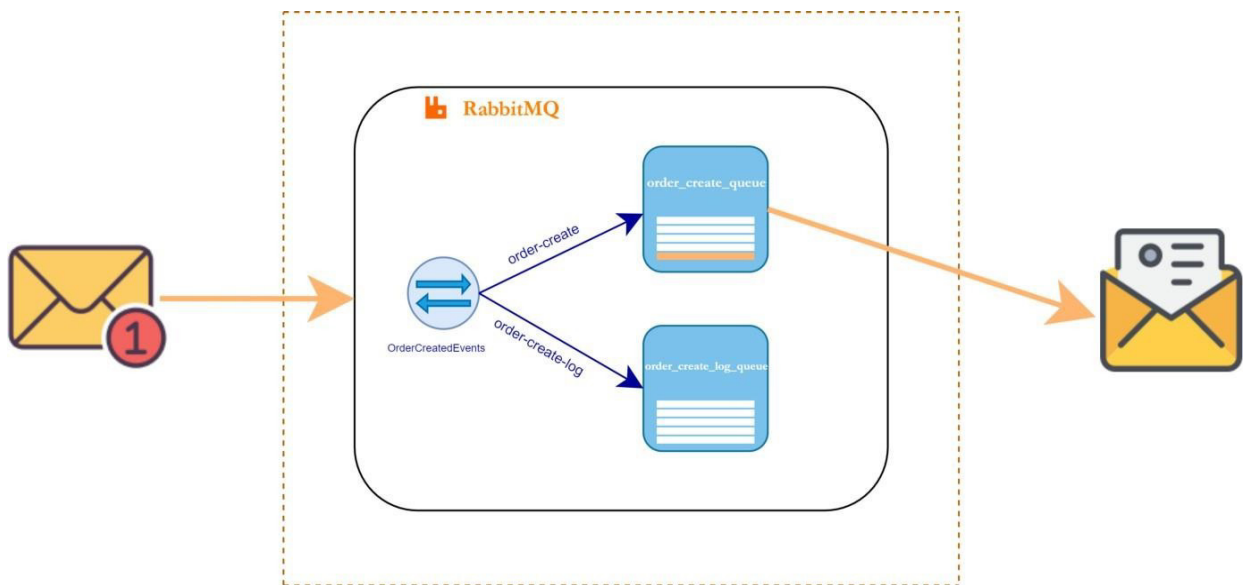


Figure 28: Direct Exchange(Source: medium.com)

4.6.6.2 Fanout Exchange

When the message is sent to the fanout exchange it simply ignores the routing key and sends the messages to all the queues it knows about, and it ignores all the keys sent while the message is published. It simply copies the message to all the queues that are bound to the exchange. Fanout exchange is used when the same message must be sent and stored to one or more queues, and it must be used in different ways. Figure 29 represents the Fanout Exchange model.

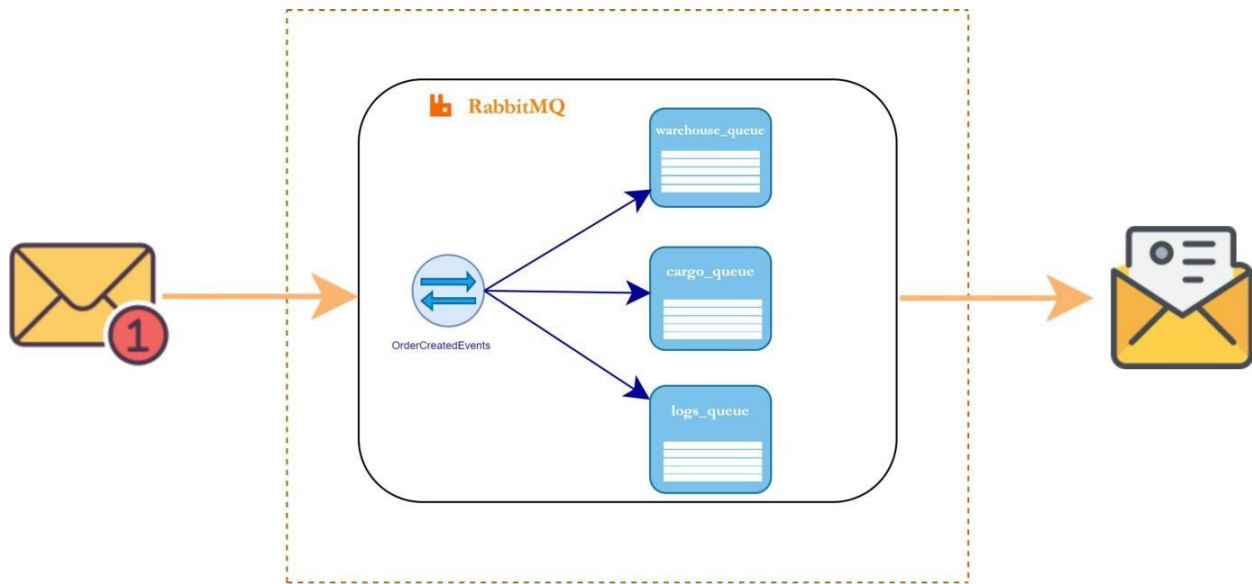


Figure 29: Fanout Exchange(Source: medium.com)

4.6.6.3 Topic Exchange

As shown in Figure 30 topic exchange allows partial matches of the keys when the message is published. The routing from the exchange to the queue can be done when the routing key sent by the producer along with the message partially matches the binding pattern.

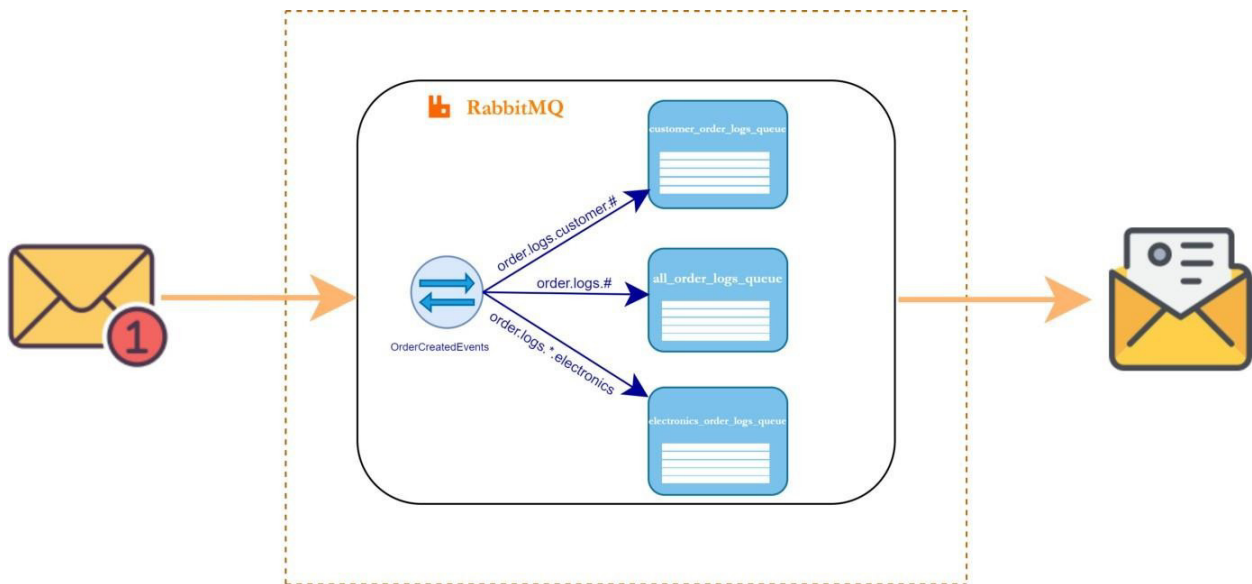


Figure 30: Topic Exchange(Source: medium.com)

All the routing keys which are sent along with the message should have words separated by the delimiter(.). Messages are sent to different queues depending on the routing key and the pattern. The pattern consists of * when the matching should be done only in the specific locations. For example “order.*.*.electronics”, this pattern and the routing key should have the same words at the same position. The first position of the routing key should be order and the fourth position should be electronics. If the routing pattern consists of # at the end, then the routing key should consist of all the words in the same order as in the pattern. The routing key should be order.logs.customer to match with the routing pattern “order.logs.customer.#” and then it sends messages to the respective queue.

4.6.6.4 Header Exchange

As shown in Figure 31, Header Exchange uses a message header instead of a routing key and is the most powerful exchange type in AMQP. The routing keys are ignored in this header exchange and the messages are sent to the queues using the header properties. A message is sent to one or more queues depending on the header properties. The producer sends a header with the message while publishing, if that header matches the header properties the message is forwarded to the queue. Header matching can be done in two ways

using ANY, ALL. Header properties are represented as key values in the binding like {"x-match","any"} or {"x-match","all"} and x-match is added between exchange and queue.

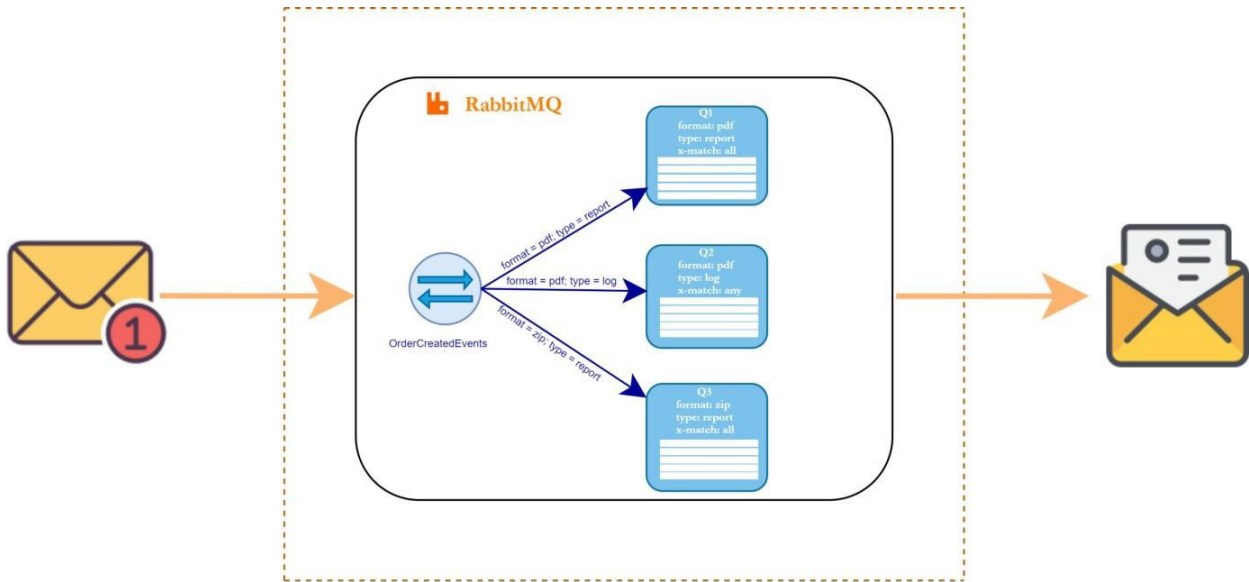


Figure 31: Header Exchange(Source: medium.com)

The message sent to the exchange should match with one of the headers linked with the queue when the x-match value is any. If it matches, then the message is sent to the corresponding queue. If the x-match value is all then the message sent to the exchange should have all the headers linked with the queue. Only then the message is forwarded to the queue.

4.7 Key Features of AMQP

4.7.1 Security

AMQP supports authentication, authorization, LDAP, and TLS through RabbitMQ plugins.

4.7.2 Reliability

When the message is sent it confirms that the message is delivered to the message broker and when it is processed successfully by the consumer.

4.7.3 Interoperability

In AMQP message is transferred as a stream of bytes so that any client can operate on it irrespective of any language.

Implementation Part

Chapter 5

Introduction to Spring Boot

5.1 Spring Boot

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications. Spring boot is a tool that lets you create stand-alone, production grade spring-based applications that you can just run because it needs minimal Spring configuration. It is a Spring module that provides the **RAD (Rapid Application Development)** feature to the Spring Framework. Spring is an application framework that lets you build enterprise java applications. It also has programming and configurational model and provides infrastructure support. As shown in the figure 32, SpringBoot is the combination of Spring Framework and embedded servers without the xml configuration (deployment descriptor).

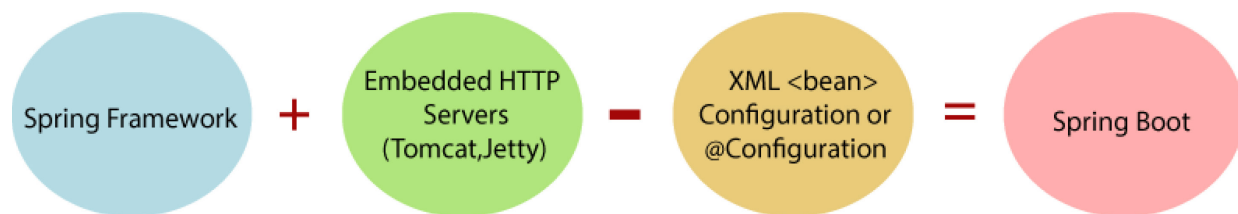


Figure 32: Spring Boot framework

5.2 Advantages of Spring Boot

- It creates standalone spring applications that lets program to run as a separate process.
- It has embedded HTTP servers like tomcat, jetty to test web applications easily. We donot need to deploy WAR files.
- It provides starter dependencies to simplify maven configuration.
- It supports programming and configurational model that automatically configure spring and third-party libraries.
- It provides production ready features such as metrics, health checks and externalized configuration.
- There is no need for XML configuration, and it offers number of plugins.
- Since spring boot is build using spring framework dependency injection

approach is possible.

- It contains powerful database transaction management capabilities.
- It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
- It reduces the cost and development time of the application.

5.3 Specifications of Spring Boot

- Spring Boot provides an opinionated approach it makes certain decisions and changes can be made if required.
- It features convention over configuration concept
- It is a stand-alone application; the application is ready to run without the requirement of external servers
- Avoids defining more Annotation Configuration
- Avoids writing lots of import statements
- Avoids XML Configuration.

5.4 Features of Spring Boot

5.4.1 Web Development

Spring boot is widely used for web application development because it has embedded HTTP servers like tomcat, jetty to test web applications easily. We do not need to deploy WAR files. Spring-boot-starter-web module can be used to start and run the application quickly.

5.4.2 Spring Application

Figure 33 is an example code snippet of spring application. SpringApplication is a class that provides a run method to bootstrap a spring application as a stand-alone application from the main method.

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

Figure 33: Main method example

5.4.3 Application Events and Listeners

Spring Boot uses events to handle the variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it to using the **ApplicationListener** key. Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and `ApplicationListener` interface. Hence, if a bean implements the `ApplicationListener`, then every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean is notified.

5.4.4 Admin Support

Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using `spring.application.admin.enabled` property.

5.4.5 Externalized Configuration

Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments. The application uses YAML files to externalize configuration.

5.4.6 Properties Files

Spring Boot provides a rich set of Application Properties. So, we can use that in the properties file of our project. The properties file is used to set properties like `server-port=8082` and many others. It helps to organize application properties.

5.4.7 YAML Support

It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The `SpringApplication` class automatically supports YAML. It is an alternative of properties file.

5.4.8 Logging

Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default. We should not change logging dependencies if no customization is needed.

5.4.9 Security

Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

5.5 Spring Boot Annotation

The three most frequently annotations used in Spring Boot are:

`@Configuration`

`@EnableAutoConfiguration`

`@ComponentScan`

5.5.1 @Configuration

When `@Configuration` annotation is used, it indicates that the class can be used as source of bean definitions by the IoC container. The class may contain one or more `@Bean` methods and can be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

5.5.2 @Enable Auto Configuration

It is used for auto-configuring beans present in the classpath in Spring Boot applications.

5.5.3 @ComponentScan

This annotation is used to enables to scan spring components automatically to scan web controller components and register them as beans.

5.5.4 @SpringBootApplication

This is the most important and core annotation of spring boot. As show in the figure 34, the mainclass of the spring boot application is marked as @SpringBootApplication

```
@SpringBootApplication
class VehicleFactoryApplication {

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

Figure 34: Application Example

@SpringBootApplication annotation is the equivalent and recommended annotation over @Configuration, @EnableAutoConfiguration @ComponentScan

5.6 Spring Boot Starters

By including correct Spring Boot Starters in the pom.xml file, spring boot will make sure that all the required dependencies are there in the class path for the application to run successfully.

Pom.xml file can be easily managed by using spring boot starters. It helps in developing applications that are production-ready, tested, and supported dependency configurations. It helps in decrease the overall configuration time for the project.

Below are the few most used starters in the spring boot application.

5.6.1 Web Starter

As shown in the figure 35, by adding web starter start up in the pom.xml will make sure that all the dependencies required including REST for developing web application are added in the classpath. It also adds the embedded tomcat server in the class path to run the HTTP web application easily.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Figure 35: Web Starter dependency

5.6.2 Data JPA Starter

As shown in the figure 36, most of the spring boot applications require persistence mechanisms hence spring boot starters come with Data JPA starter without the requirement of external configuration. Data JPA starter supports H2, Derby and Hsqldb.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Figure 36: data JPA and H2 dependencies

5.6.3 Mail Starter

As shown in figure 37, in most applications there is a requirement to send emails from the system. Spring Boot Mail starter provides an easy way to handle this feature by hiding all complexities. We can enable email support by adding mail starter in our application.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

Figure 37: mail starter dependency

5.6.4 Test Starter

As shown in figure 38, Test Starter automatically adds the Junit, Mockito or Spring Test libraries to the classpath to test the spring boot applications.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Figure 38: starter test dependency

Test Starter automatically adds the Junit, Mockito or Spring Test libraries to the classpath to test the spring boot applications.

The other most used spring boot starters are:

- spring-boot-starter-security
- spring-boot-starter-web-services
- spring-boot-starter-integration

- spring-boot-starter-validation
- spring-boot-starter-actuator

5.7 Problems of Spring Boot

By using spring boot starters spring boot might use many dependencies that are not used in the application. This increases the size of the application.

Chapter 6

WebSocket in Spring Boot

We need to follow few steps to do a sample implementation of WebSocket in spring boot, As a sample example, which is good to understand the flow and the process of implementation of WebSocket in Spring Boot, we are going to develop a small echo application.

This application at the end is going to have a WebSocket endpoint where a client can connect and send some messages which will be sent back as response.

6.1 Dependencies Required

As we can see from the figure 39, we need to add three dependencies to develop an echoing application with works with JSON message.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20180130</version>
  </dependency>
</dependencies>
```

Figure 39: Required dependencies

The first dependency in the figure i.e., spring-boot-starter-websocket and the third dependency i.e., JSON, are the dependencies which the developer should be added.

The second dependency i.e., spring-boot-starter-test is added by default to the pom.xml when we create the spring boot project.

6.2 Detail of application.properties files

As we know that application.properties file is very important to run the application in different environments.

As we can see from the figure 40, we are setting the properties such as the port that where application should be running on and the properties that are required by the logs and logfiles.

```
server.port = 8001

logging.level.root=DEBUG
logging.level.thesis.without.stomp=DEBUG
logging.level.org.springframework=DEBUG
logging.file.name=thesis_websocket_without_stomp.log
```

Figure 40: application.properties

6.2.1 Log Files

Log Files are the files which has all the details about the running application. When the server is up and running, there will be several requests and queues and lot of other things that will be up and running, the log files are useful to understand all these things.

The most important thing where log files are most useful is when the errors has been occurred and need to be resolved then log files make the developer's life much easier, to identify the exact point and the reason in some cases which is the cause of the error.

In this sample application, a new log file will be created each day. But this can be changed according to the developer's requirements.

6.3 Project Structure

We understand from the figure 41 that we need to create a config class file and a handler for echoing. Here we have named our sample application as withoutstomp project.

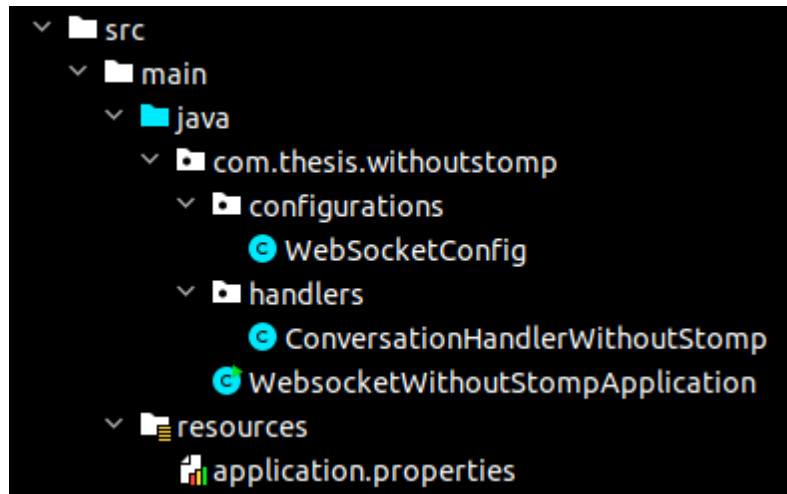


Figure 41: Project structure

6.4 Main Method

As we know about the importance of main method in every java project which is also discussed in the Introduction to Spring Boot chapter, here is the main method of our application.

From the figure 42, we can understand that we are declaring a spring boot application with its annotation. We are also starting the server with the port which is given with server.port in the application.properties file i.e., 8001 in our current application.

```

@SpringBootApplication
public class WebSocketWithoutStompApplication {

    private static final Logger log = LoggerFactory.getLogger(WebSocketWithoutStompApplication.class);

    public static void main(String[] args) throws UnknownHostException {
        SpringApplication app = new SpringApplication(WebSocketWithoutStompApplication.class);
        Environment env = app.run(args).getEnvironment();

        log.info("-----
APPLICATION STARTED on port {}
-----",
env.getProperty("server.port")
);
    }
}

```

Figure 42: Main method

6.5 WebSocket Config

The `WebSocketConfig` class file implements `WebSocketConfigurer` and has two annotations `@Configuration` and `@EnableWebSocket` as shown in figure 43.

The `registerWebSocketHandlers()` method is the key to registering a message handler. By overriding it, you're given a `WebSocketHandlerRegistry` through which you can call `addHandler()` to register a message handler.

```

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    private static final Logger logger = LoggerFactory.getLogger(WebSocketConfig.class);

    @Autowired
    ConversationHandlerWithoutStomp conversationHandlerWithoutStomp;

    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(conversationHandlerWithoutStomp, ...paths: "/conversation/messages");
    }
}

```

Figure 43: `WebSocketConfig`

6.6 Echoing Handler

We have named our echoing handler class as `ConversationHandlerWithoutStomp` which extends `TextWebSocketHandler`, which exchanges text messages.

Also, whenever there is a request then we are also logging it to the logfile, initially the program is checking whether the message that has been sent is a json or not.

```
String payload = message.getPayload();
```

From the below code shown in figure 44, if it is not a json message then an exception will be thrown asking to check the message.

If the message is validated as a correct json message then, `session.sendMessage();` will be sending back the json message.

```
@Component
public class ConversationHandlerWithoutStomp extends TextWebSocketHandler {

    private static final Logger logger = LoggerFactory.getLogger(ConversationHandlerWithoutStomp.class);

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws InterruptedException, IOException {

        logger.info("Calling handleTextMessage with parameter:{}", message.getPayload());

        try {

            String payload = message.getPayload();
            JSONObject jsonObject = new JSONObject(payload);
            session.sendMessage(new TextMessage(jsonObject.toString()));

        } catch (Exception e) {
            logger.debug("Error" + e.getMessage());
            session.sendMessage(new TextMessage(payload: "check message"));
        }

    }

}
```

Figure 44: Echoing Handler

6.7 Running the application

We can see from the figure 45, that the application has been started on the port 8001.

```
2022-06-11 12:35:00.570 INFO 30409 --- [          main] c.t.w.WebsocketWithoutStompApplication : Started WebsocketWith
2022-06-11 12:35:00.573 DEBUG 30409 --- [          main] o.s.b.a.ApplicationAvailabilityBean   : Application availabi
2022-06-11 12:35:00.581 DEBUG 30409 --- [          main] o.s.b.a.ApplicationAvailabilityBean   : Application availabi
2022-06-11 12:35:00.582 INFO 30409 --- [          main] c.t.w.WebsocketWithoutStompApplication :
-----
APPLICATION STARTED on port 8001
-----
```

Figure 45: Start of the application

6.8 Results of the application

As we can see from the figure 46, a connection has been established with the server of port 8001 and with the handler /conversation/messages.

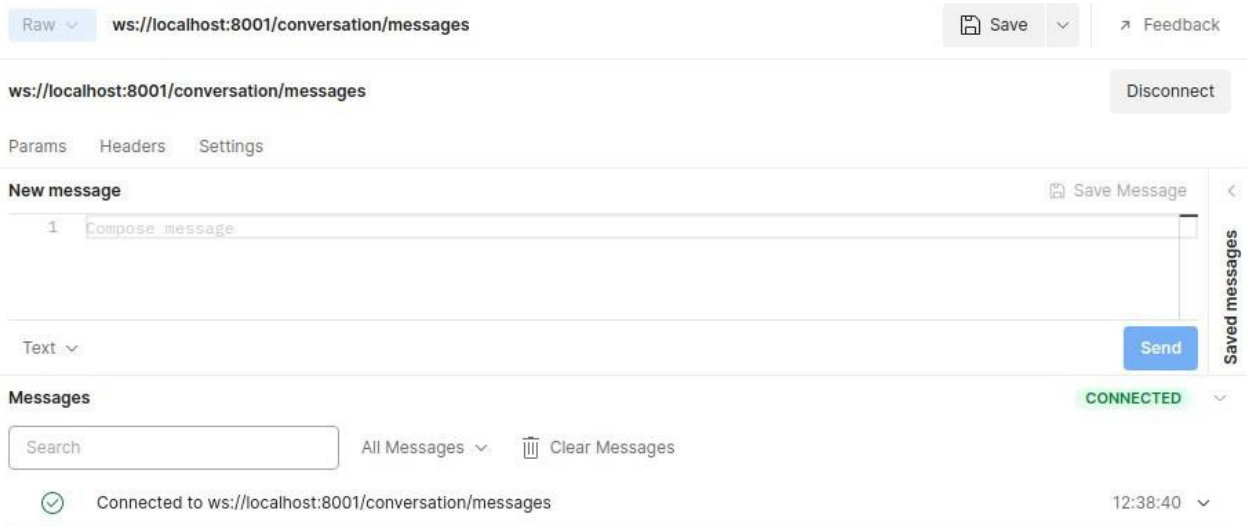


Figure 46: Establishing the connection

A message has been sent to the server, which is a json text message, we can also see in the figure 47, that the message has also been echoed back to the same session

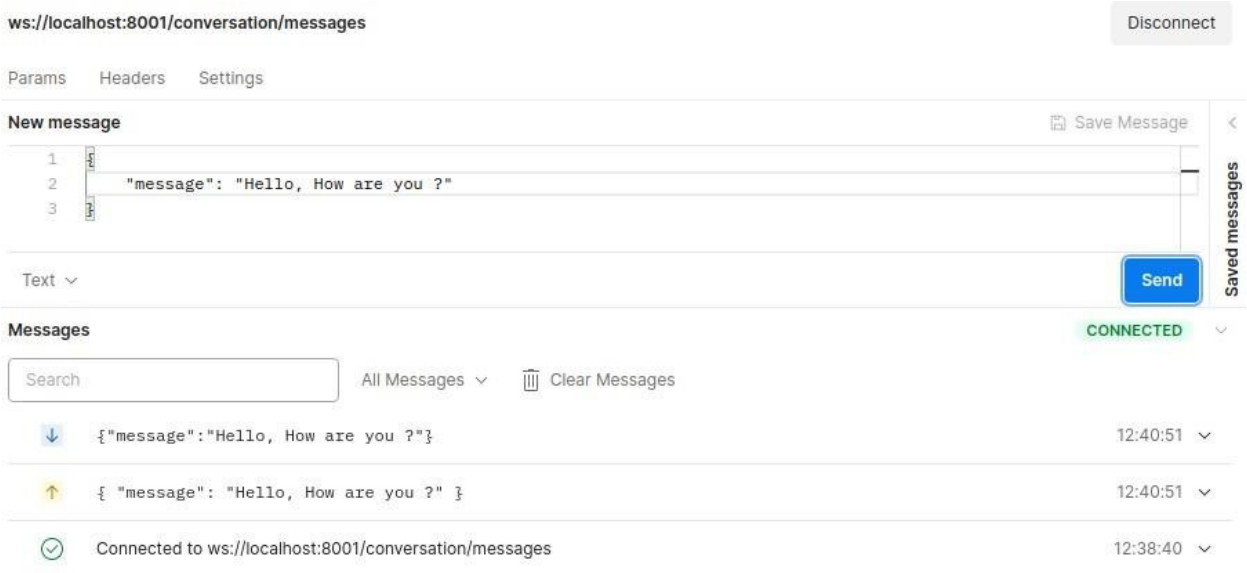


Figure 47: Sending and Receiving json messages

We can observe in the figure 48, that the message which is sent from the previous established connection from the postman has also been logged in the terminal of the server which is up and running.

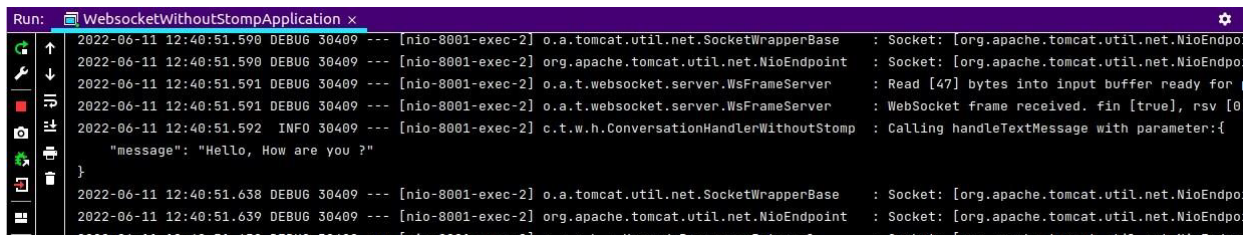
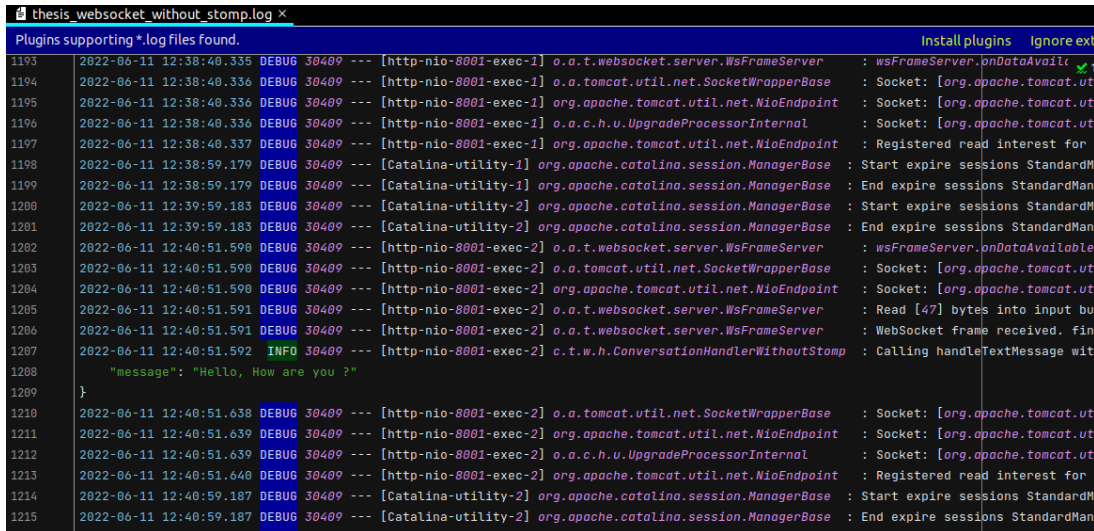


Figure 48: Logging the messages

6.9 Log Files of the results

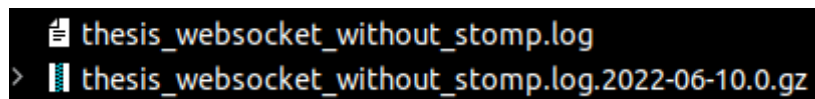
The figure 49 is the log file which has the details of the running server and requests received, etc which includes the detailed information of everything i.e., most importantly the incoming and outgoing messages, date and time, and the issues which are debugged if there occurs an unencountered error.



```
thesis_websocket_without_stomp.log x
Plugins supporting *.log files found. Install plugins Ignore ext
1193 2022-06-11 12:38:40.335 DEBUG 30409 --- [http-nio-8001-exec-1] o.a.t.websocket.server.WsFrameServer : wsFrameServer.onDataAvail(
1194 2022-06-11 12:38:40.336 DEBUG 30409 --- [http-nio-8001-exec-1] o.a.tomcat.util.net.SocketWrapperBase : Socket: [org.apache.tomcat.ut
1195 2022-06-11 12:38:40.336 DEBUG 30409 --- [http-nio-8001-exec-1] org.apache.tomcat.util.net.NioEndpoint : Socket: [org.apache.tomcat.ut
1196 2022-06-11 12:38:40.336 DEBUG 30409 --- [http-nio-8001-exec-1] o.a.c.h.u.UpgradeProcessorInternal : Socket: [org.apache.tomcat.ut
1197 2022-06-11 12:38:40.337 DEBUG 30409 --- [http-nio-8001-exec-1] org.apache.tomcat.util.net.NioEndpoint : Registered read interest for
1198 2022-06-11 12:38:59.179 DEBUG 30409 --- [Catalina-utility-1] org.apache.catalina.session.ManagerBase : Start expire sessions StandardM
1199 2022-06-11 12:38:59.179 DEBUG 30409 --- [Catalina-utility-1] org.apache.catalina.session.ManagerBase : End expire sessions StandardMan
1200 2022-06-11 12:39:59.183 DEBUG 30409 --- [Catalina-utility-2] org.apache.catalina.session.ManagerBase : Start expire sessions StandardM
1201 2022-06-11 12:39:59.183 DEBUG 30409 --- [Catalina-utility-2] org.apache.catalina.session.ManagerBase : End expire sessions StandardMan
1202 2022-06-11 12:40:51.590 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : wsFrameServer.onDataAvailable
1203 2022-06-11 12:40:51.590 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.tomcat.util.net.SocketWrapperBase : Socket: [org.apache.tomcat.ut
1204 2022-06-11 12:40:51.590 DEBUG 30409 --- [http-nio-8001-exec-2] org.apache.tomcat.util.net.NioEndpoint : Socket: [org.apache.tomcat.ut
1205 2022-06-11 12:40:51.591 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : Read [47] bytes into input bu
1206 2022-06-11 12:40:51.591 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : WebSocket frame received, fin
1207 2022-06-11 12:40:51.592 INFO 30409 --- [http-nio-8001-exec-2] c.t.w.h.ConversationHandlerWithoutStomp : Calling handleTextMessage with
1208 "message": "Hello, How are you ?"
1209 }
1210 2022-06-11 12:40:51.638 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.tomcat.util.net.SocketWrapperBase : Socket: [org.apache.tomcat.ut
1211 2022-06-11 12:40:51.639 DEBUG 30409 --- [http-nio-8001-exec-2] org.apache.tomcat.util.net.NioEndpoint : Socket: [org.apache.tomcat.ut
1212 2022-06-11 12:40:51.639 DEBUG 30409 --- [http-nio-8001-exec-2] o.a.c.h.u.UpgradeProcessorInternal : Socket: [org.apache.tomcat.ut
1213 2022-06-11 12:40:51.640 DEBUG 30409 --- [http-nio-8001-exec-2] org.apache.tomcat.util.net.NioEndpoint : Registered read interest for
1214 2022-06-11 12:40:59.187 DEBUG 30409 --- [Catalina-utility-2] org.apache.catalina.session.ManagerBase : Start expire sessions StandardM
1215 2022-06-11 12:40:59.187 DEBUG 30409 --- [Catalina-utility-2] org.apache.catalina.session.ManagerBase : End expire sessions StandardMan
```

Figure 49: Log file

The figure 50 has two log files which one of them has the date that the log file is created and details of the server on that day, and the log file with no date is the log file of the current day's log file.



```
thesis_websocket_without_stomp.log
> thesis_websocket_without_stomp.log.2022-06-10.0.gz
```

Figure 50: Everyday logs

Chapter 7

WebSocket with STOMP in Spring Boot

In this chapter, a chat application has been developed using WebSockets with STOMP in SpringBoot. This helps us to get more clear idea about the STOMP implementation in WebSockets using Spring Boot.

7.1 Dependencies Required

By looking at figure 51, we can understand the required dependencies for the implementation of chat application.

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator-core</artifactId>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>sockjs-client</artifactId>
  <version>1.0.2</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>stomp-websocket</artifactId>
  <version>2.3.3</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.7</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.1.1-1</version>
</dependency>
```

Figure 51: Dependencies required

The most important dependencies are stomp-websocket and sockjs-client.

WebJars are client-side web libraries (e.g., jQuery & Bootstrap) packaged into JAR (Java Archive) files.

Explicitly and easily manage the client-side dependencies in JVM-based web applications

Use JVM-based build tools (e.g., Maven, Gradle, sbt, ...) to download your client-side dependencies

Transitive dependencies are automatically resolved and optionally loaded via RequireJS

7.2 Details of application.properties file

The application.properties shown in figure 52, are the same properties that we have used in chapter – 6.

```
server.port = 8001

logging.level.root=DEBUG
logging.level.thesis.without.stomp=DEBUG
logging.level.org.springframework=DEBUG
logging.file.name=thesis_websocket_without_stomp.log
```

Figure 52: application.properties file

7.3 Project Structure

We can see from the figure 53, that a html, CSS, and js files have also been developed. This is because the WebSockets with STOMP cannot be tested in postman. So, we have also developed a client application.

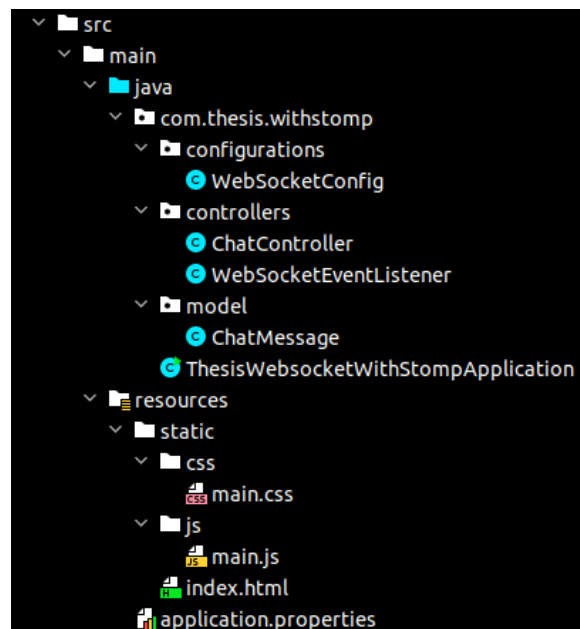


Figure 53: Project structure

ChatMessage is the model of the message, ChatController is the controller which receives and sends the message to appropriate people who have subscribed to the end point.

WebSocketEventListener is the event listener which gives the notification of a new person who has subscribed to the endpoint.

In simple words, the output of the chat application can be imagined as a group chatting.

7.4 Main Method of the Application

The main method of this application shown in figure 54 is almost same as the chapter- 6's main method.

```
@SpringBootApplication
public class ThesisWebSocketWithStompApplication {

    private static final Logger log = LoggerFactory.getLogger(ThesisWebSocketWithStompApplication.class);

    public static void main(String[] args) throws UnknownHostException {
        SpringApplication app = new SpringApplication(ThesisWebSocketWithStompApplication.class);
        Environment env = app.run(args).getEnvironment();

        log.info("""
        -----
        APPLICATION STARTED on port {}
        -----""",
            env.getProperty("server.port")
        );
    }
}
```

Figure 54: Main method

7.5 WebSocket Config

Here comes our one of the important parts of our application.

The `configureMessageBroker()` method implements the default method in `WebSocketMessageBrokerConfigurer` to configure the message broker. It starts by calling `enableSimpleBroker()` to enable a simple memory-based message broker to carry the greeting messages back to the client on destinations prefixed with

`/topic`. It also designates the `/app` prefix for messages that are bound for methods annotated with `@MessageMapping`

The `registerStompEndpoints()` method registers the `/ws` endpoint, enabling SockJS fallback options so that alternate transports can be used if `WebSocket` is not available. The SockJS client will attempt to connect to `/ws` and use the best available transport (`websocket`, `xhr-streaming`, `xhr-polling`, and so on). An example code snippet of `WebSocketConfig` is shown in figure 55.

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint(...paths: "/ws").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker(...destinationPrefixes: "/topic");
    }
}
```

Figure 55: `WebSocketConfig`

7.6 Chat Controller

As we can see from the figure 56, there are two message mappings i.e., `/chat.sendMessage` and `/chat.addUser`, and both message mappings are replying to the subscribers who are subscribed to `/topic/public`.

```
@Controller
public class ChatController {

    @RequestMapping("/chat.sendMessage")
    @SendTo("/topic/public")
    public ChatMessage sendMessage(@Payload ChatMessage chatMessage) { return chatMessage; }

    @RequestMapping("/chat.addUser")
    @SendTo("/topic/public")
    public ChatMessage addUser(@Payload ChatMessage chatMessage,
                               SimpMessageHeaderAccessor headerAccessor) {
        // Add username in web socket session
        headerAccessor.getSessionAttributes().put("username", chatMessage.getSender());
        return chatMessage;
    }
}
```

Figure 56: Chat controller

Annotation Type `RequestMapping`. Annotation for mapping a Message onto a message-handling method by matching the declared patterns to a destination extracted from the message.

Annotation that indicates a method's return value should be converted to a Message if necessary and sent to the specified destination.

In a typical request/reply scenario, the incoming Message may convey the destination to use for the reply. In that case, that destination should take precedence.

This annotation may be placed class-level in which case it is inherited by methods of the class.

7.7 Event Listener

The event listener which we can see in the figure 57, whenever the user is disconnected from the session, then the other people who are subscribed will be notified.

```
@EventListener
public void handleWebSocketDisconnectListener(SessionDisconnectEvent event) {
    StompHeaderAccessor headerAccessor = StompHeaderAccessor.wrap(event.getMessage());

    String username = (String) headerAccessor.getSessionAttributes().get("username");
    if(username != null) {
        logger.info("User Disconnected : " + username);

        ChatMessage chatMessage = new ChatMessage();
        chatMessage.setType(ChatMessage.MessageType.LEAVE);
        chatMessage.setSender(username);

        messagingTemplate.convertAndSend(destination: "/topic/public", chatMessage);
    }
}
```

Figure 57: Event listener

7.8 Connecting and Subscribing the client

Whenever there is a new client for connection, the stomp client makes the user to be subscribed to `/topic/public` as shown in figure 58.

```
function onConnected() {
    // Subscribe to the Public Topic
    stompClient.subscribe('/topic/public', onMessageReceived);

    // Tell your username to the server
    stompClient.send("/app/chat.addUser",
        {},
        JSON.stringify({sender: username, type: 'JOIN'})
    )

    connectingElement.classList.add('hidden');
}
```

Figure 58: Connecting and subscribing the JS client

7.9 Sending message from JS client

From the figure 59, the json message is being sent to the end point of STOMP i.e., `/app/chat.sendMessage`.

```
function sendMessage(event) {
  var messageContent = messageInput.value.trim();
  if(messageContent && stompClient) {
    var chatMessage = {
      sender: username,
      content: messageInput.value,
      type: 'CHAT'
    };
    stompClient.send("/app/chat.sendMessage", {}, JSON.stringify(chatMessage));
    messageInput.value = '';
  }
  event.preventDefault();
}
```

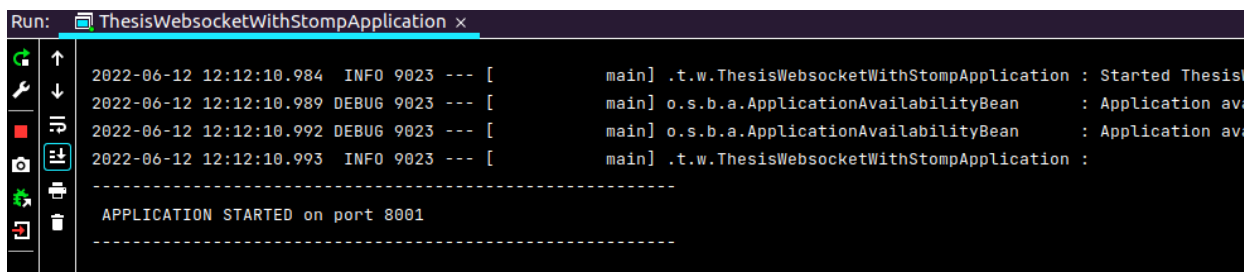
Figure 59: Sending message from JS client

This is part of the stomp client which is developed in JavaScript. For the UI, we have used HTML and CSS.

7.10 Results of the chat application

7.10.1 Start the server

As shown in figure 60 the application has started on the port 8001.

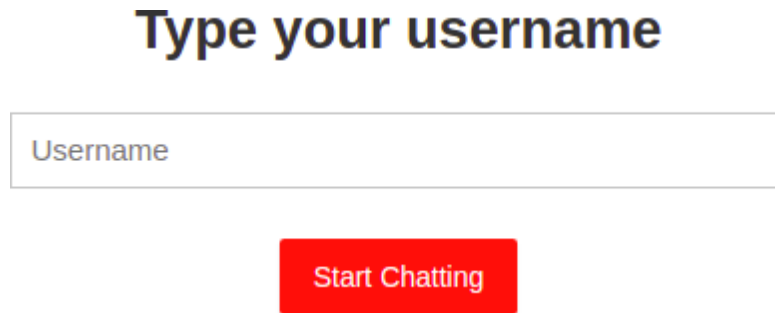


```
Run: ThesisWebsocketWithStompApplication x
2022-06-12 12:12:10.984 INFO 9023 --- [main] .t.w.ThesisWebsocketWithStompApplication : Started ThesisWebsocketWithStompApplication
2022-06-12 12:12:10.989 DEBUG 9023 --- [main] o.s.b.a.ApplicationAvailabilityBean : Application availability check succeeded
2022-06-12 12:12:10.992 DEBUG 9023 --- [main] o.s.b.a.ApplicationAvailabilityBean : Application availability check succeeded
2022-06-12 12:12:10.993 INFO 9023 --- [main] .t.w.ThesisWebsocketWithStompApplication :
-----
APPLICATION STARTED on port 8001
-----
```

Figure 60: Start of the application

7.10.2 UI of the chat application

When the application is started successfully, then the UI of localhost:8001 is the figure 61.

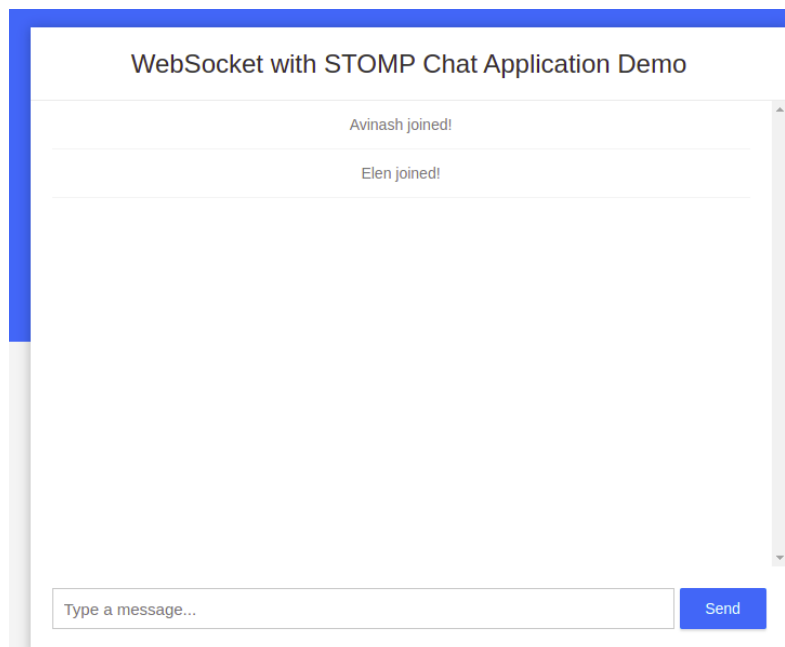


The screenshot displays a simple web interface. At the top, the text "Type your username" is centered in a large, bold, black font. Below this text is a white rectangular input field with a thin grey border, containing the placeholder text "Username". Centered below the input field is a prominent red rectangular button with the white text "Start Chatting".

Figure 61: UI of localhost:8001

7.10.3 New User Notification

Whenever new users join the group, then the existing connected users will be notified as shown in the figure 62.



The screenshot shows a chat application window titled "WebSocket with STOMP Chat Application Demo". The window has a blue border. Inside, there is a white chat area with a vertical scrollbar on the right. Two messages are visible: "Avinash joined!" and "Elen joined!". At the bottom of the window, there is a white input field with the placeholder text "Type a message..." and a blue "Send" button to its right.

Figure 62: New User notification

7.10.4 Communications in the group

This is the sample discussion in the group which is shown in figure 63. We can see the notifications of the new users who are joining and the users who are leaving the group.

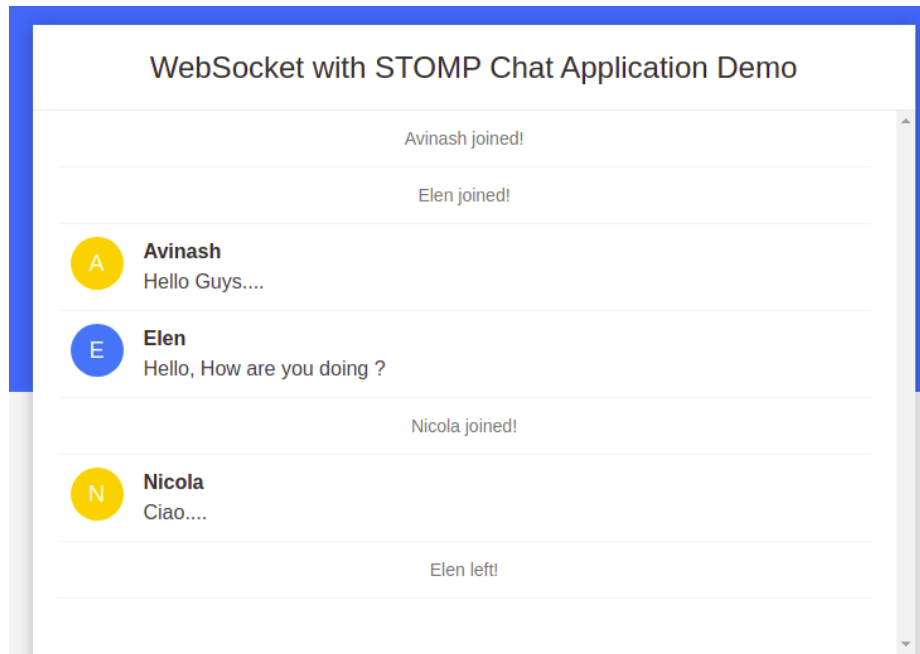
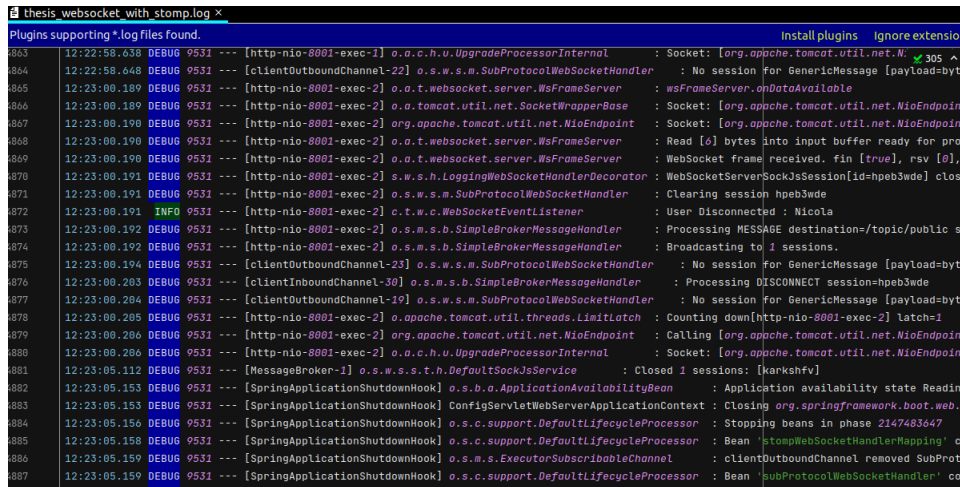


Figure 63: Communications in the group

7.10.5 Log files

This figure 64 is the sample log file which has the details of the server and the works that the server has done for the application



```
thesis_websocket_with_stomp.log x
Plugins supporting *.log files found. Install plugins Ignore extension
883 12:22:58.638 DEBUG 9531 --- [http-nio-8001-exec-1] o.a.c.h.u.UpgradeProcessorInternal : Socket: [org.apache.tomcat.util.net.NioEndpoint]
884 12:22:58.648 DEBUG 9531 --- [clientOutboundChannel-22] o.s.w.s.m.SubProtocolWebSocketHandler : No session for GenericMessage [payload=byte
885 12:23:00.189 DEBUG 9531 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : wsFrameServer.onDataAvailable
886 12:23:00.189 DEBUG 9531 --- [http-nio-8001-exec-2] o.a.tomcat.util.net.SocketWrapperBase : Socket: [org.apache.tomcat.util.net.NioEndpoint]
887 12:23:00.190 DEBUG 9531 --- [http-nio-8001-exec-2] org.apache.tomcat.util.net.NioEndpoint : Socket: [org.apache.tomcat.util.net.NioEndpoint]
888 12:23:00.190 DEBUG 9531 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : Read [6] bytes into input buffer ready for prod
889 12:23:00.190 DEBUG 9531 --- [http-nio-8001-exec-2] o.a.t.websocket.server.WsFrameServer : WebSocket frame received, fin [true], rsv [0],
890 12:23:00.191 DEBUG 9531 --- [http-nio-8001-exec-2] s.w.s.h.LoggingWebSocketHandlerDecorator : WebSocketServerSocketSession[id=hpb3wde] close
891 12:23:00.191 DEBUG 9531 --- [http-nio-8001-exec-2] o.s.w.s.m.SubProtocolWebSocketHandler : Clearing session hpb3wde
892 12:23:00.191 INFO 9531 --- [http-nio-8001-exec-2] c.t.w.c.WebSocketEventListener : User Disconnected : Nicola
893 12:23:00.192 DEBUG 9531 --- [http-nio-8001-exec-2] o.s.m.s.b.SimpleBrokerMessageHandler : Processing MESSAGE destination=/topic/public se
894 12:23:00.192 DEBUG 9531 --- [http-nio-8001-exec-2] o.s.m.s.b.SimpleBrokerMessageHandler : Broadcasting to 1 sessions.
895 12:23:00.194 DEBUG 9531 --- [clientOutboundChannel-23] o.s.w.s.m.SubProtocolWebSocketHandler : No session for GenericMessage [payload=byte
896 12:23:00.203 DEBUG 9531 --- [clientInboundChannel-30] o.s.m.s.b.SimpleBrokerMessageHandler : Processing DISCONNECT session=hpb3wde
897 12:23:00.204 DEBUG 9531 --- [clientOutboundChannel-19] o.s.w.s.m.SubProtocolWebSocketHandler : No session for GenericMessage [payload=byte
898 12:23:00.205 DEBUG 9531 --- [http-nio-8001-exec-2] o.apache.tomcat.util.threads.LimitLatch : Counting down[http-nio-8001-exec-2] latch=1
899 12:23:00.206 DEBUG 9531 --- [http-nio-8001-exec-2] org.apache.tomcat.util.net.NioEndpoint : Calling [org.apache.tomcat.util.net.NioEndpoint]
900 12:23:00.206 DEBUG 9531 --- [http-nio-8001-exec-2] o.a.c.h.u.UpgradeProcessorInternal : Socket: [org.apache.tomcat.util.net.NioEndpoint]
901 12:23:05.112 DEBUG 9531 --- [MessageBroker-1] o.s.w.s.s.t.h.DefaultSocketJsService : Closed 1 sessions: [karkshfv]
902 12:23:05.153 DEBUG 9531 --- [SpringApplicationShutdownHook] o.s.b.a.ApplicationAvailabilityBean : Application availability state Reading
903 12:23:05.153 DEBUG 9531 --- [SpringApplicationShutdownHook] ConfigServletWebServerApplicationContext : Closing org.springframework.boot.web.s
904 12:23:05.156 DEBUG 9531 --- [SpringApplicationShutdownHook] o.s.c.support.DefaultLifecycleProcessor : Stopping beans in phase 2147483647
905 12:23:05.158 DEBUG 9531 --- [SpringApplicationShutdownHook] o.s.c.support.DefaultLifecycleProcessor : Bean 'stompWebSocketHandlerMapping' cd
906 12:23:05.159 DEBUG 9531 --- [SpringApplicationShutdownHook] o.s.m.s.ExecutorSubscribableChannel : clientOutboundChannel removed SubProtoc
907 12:23:05.159 DEBUG 9531 --- [SpringApplicationShutdownHook] o.s.c.support.DefaultLifecycleProcessor : Bean 'subProtocolWebSocketHandler' com
```

Figure 64: Log file

7.10.6 Everyday log files

These are the sample everyday log files shown in figure 65, the log file with date is the log file of that day and the log file which does not have date and having only the name is the current day's log file. But according to the developer's requirements log files can also be created for every hour.

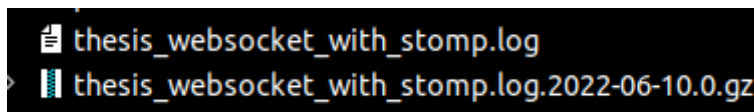


Figure 65: Everyday log files

Chapter 8

WebSocket using STOMP with RabbitMQ as Message Broker

RabbitMQ is a message broker that implements Advanced Message Queuing Protocol (AMQP) which we have learnt about in the specific chapter of AMQP.

Here we are going to implement RabbitMQ as message broker for communications between the clients.

In simple words, we will be replacing the simple broker that we have used in the previous chapter with RabbitMQ. This helps us to understand the queues and connection and a lot of things.

We are going to use the same project and code, but the main difference between the previous chapter that is with the simple broker of stomp, and this is the configuration.

8.1 WebSocket Config

As we can see from the figure 66, the stomp broker has been enabled to port 61613, which is the port of RabbitMQ for stomp.

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint(...paths: "/ws").withSockJS();
    }

    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");

        // Use this for enabling a Full featured broker like RabbitMQ
        registry.enableStompBrokerRelay(...destinationPrefixes: "/topic")
            .setRelayHost("localhost")
            .setRelayPort(61613)
            .setClientLogin("guest")
            .setClientPasscode("guest");
    }
}
```

Figure 66: WebSocketConfig

The credentials of the RabbitMQ that are being used are of guest.

8.2 Results of the application

8.2.1 Starting the server

The server has started, and it is connected to RabbitMQ, which we can see the log in the figure 67, that the connection has been established.

```
Run: ThesisStompWithRabbitMqApplication x
2022-06-12 12:36:46.721 DEBUG 10780 --- [ent-scheduler-2] io.netty.util.Recycler : -Dio.netty.recycler.maxCapac
2022-06-12 12:36:46.721 DEBUG 10780 --- [ent-scheduler-2] io.netty.util.Recycler : -Dio.netty.recycler.ratio: 8
2022-06-12 12:36:46.721 DEBUG 10780 --- [ent-scheduler-2] io.netty.util.Recycler : -Dio.netty.recycler.chunkSiz
2022-06-12 12:36:46.721 DEBUG 10780 --- [ent-scheduler-2] io.netty.util.Recycler : -Dio.netty.recycler.blocking
2022-06-12 12:36:46.767 DEBUG 10780 --- [ent-scheduler-3] o.s.m.s.s.StompBrokerRelayMessageHandler : Received CONNECTED heart-bea
2022-06-12 12:36:46.767 INFO 10780 --- [ent-scheduler-3] o.s.m.s.s.StompBrokerRelayMessageHandler : "System" session connected.
2022-06-12 12:36:46.768 DEBUG 10780 --- [ent-scheduler-3] reactor.netty.ReactorNetty : [5afe1f30, L:/127.0.0.1:4964
2022-06-12 12:36:46.769 DEBUG 10780 --- [ent-scheduler-3] reactor.netty.ReactorNetty : [5afe1f30, L:/127.0.0.1:4964
2022-06-12 12:36:46.772 DEBUG 10780 --- [ent-scheduler-3] reactor.netty.ReactorNetty : [5afe1f30, L:/127.0.0.1:4964
2022-06-12 12:36:46.773 DEBUG 10780 --- [ent-scheduler-3] reactor.netty.ReactorNetty : [5afe1f30, L:/127.0.0.1:4964
```

Figure 67: Start of the application and connection to RabbitMQ

8.2.2 Checking the server connection to RabbitMQ in RabbitMQ portal

We can see in the figure 68, that a new connection which is a server of the application has connected.

The screenshot shows the RabbitMQ management interface. At the top, there is a navigation bar with tabs for Overview, Connections (selected), Channels, Exchanges, Queues, and Admin. The main content area is titled 'Connections' and shows 'All connections (1)'. Below this, there is a pagination section with 'Page 1 of 1' and a search filter. A table displays the connection details:

Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
127.0.0.1:49642	guest	running	o	STOMP 1.2	1	0B/s	0B/s

At the bottom of the interface, there is an 'Update' dropdown set to 'every 5 seconds' and a timestamp 'Last update: 2022-06-12 12:38:54'.

Figure 68: Server connection to RabbitMQ

8.2.3 New user

New user has been joined in the group as shown in figure 69, which means a new RabbitMQ connection and should also be established.

WebSocket STOMP with RabbitMQ Chat Application Demo

Avinash joined!

Figure 69: New user notification

Here in the figure 70, we can see that when there is a new user joined in the group, a new RabbitMQ connection is also established.

Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
127.0.0.1:49642	guest	running	o	STOMP 1.2	1	0B/s	0B/s
127.0.0.1:49754	guest	running	o	STOMP 1.1	1	0B/s	0B/s

Figure 70: New connection to RabbitMQ

8.2.4 RabbitMQ Monitoring

The following are the few monitoring results of the RabbitMQ during the exchanging of messages in the group among users.

RabbitMQ comes with a management UI and HTTP API which exposes a number of RabbitMQ metrics for nodes, connections, queues, message rates as shown in figure 71 and so on. This is a convenient option for development and in environments where external monitoring is difficult or impossible to introduce.

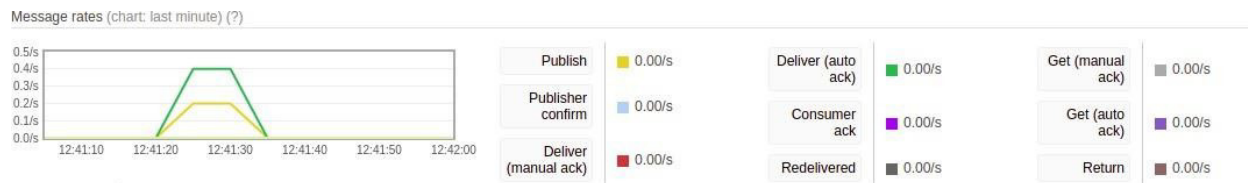


Figure 71: Message Rate

As we can see from figure 72, Global counts in the management UI shows the total number of connections to the RabbitMQ, total number of channels, total number of exchanges, total number of queues, total number of consumers.

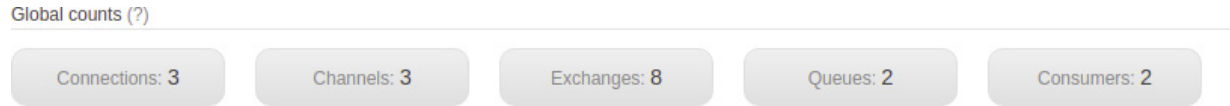


Figure 72: Global Counts

Each user when is connected to the server of the application to be a member of the chat application group, specific queue will be created. In this case, we have two users joined in the group which implies two queues to be generated as shown in the output figure 73.

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
stomp-subscription-a76EaiVirTTUXPFOUMe0Gw	AD	idle	0	0	0	0.00/s	0.00/s	0.00/s
stomp-subscription-pVKB1BLxSDbts_nN5e1SDg	AD	idle	0	0	0	0.00/s	0.00/s	0.00/s

Figure 73: Each User Queues

When the maintainers of the application or developers working for the application want to know all the metrics and other details of the queues and their status etc. They can get them through the end points provided by the documentation of the RabbitMQ or they can download through the export definitions as shown in the figure 74 which will be a json file with the details.

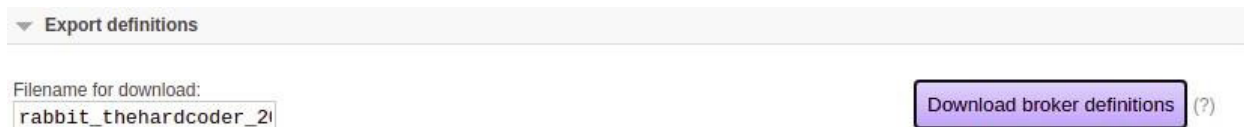


Figure 74: Download details JSON file

8.3 POM.XML

Finally, as shown in figure 75, there are two important dependencies to be added to successfully start the project.

```
<!-- RabbitMQ Starter Dependency -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>

<!-- Following additional dependency is required for Full Featured STOMP Broker Relay -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-reactor-netty</artifactId>
</dependency>
```

Figure 75: POM.xml

Conclusion

Among the research that has been done and the implementations of few protocols, there are several advantages of using different protocols based on the requirements of the chat applications.

WebRTC enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary.

XMPP is the Extensible Messaging and Presence Protocol, a set of open technologies for instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.

The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.

The Advanced Message Queuing Protocol (AMQP) is an open standard for passing business messages between applications or organizations. It connects systems, feeds business processes with the information they need and reliably transmits onward the instructions that achieve their goals.

We have also developed and tested the chat applications in Java using Spring Boot with WebSockets alone, having STOMP over WebSockets and also using RabbitMQ as message broker while using STOMP over WebSockets.

We can clearly understand that whichever protocol is being used based on the requirements having a broker (ex: RabbitMQ) really helps the applications a lot. As a message broker is an architectural pattern for message validation, transformation, and routing. It mediates communication among applications, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling.

Bibliography

- Baeldung Contributors. (2022, April 29). *Learn Spring Boot*. Retrieved from baeldung.com: <https://www.baeldung.com/spring-boot>
- Cosette Cressler. (2021, July 20). *Everything About XMPP - Extensible Messaging & Presence Protocol*. Retrieved from cometchat.com: <https://www.cometchat.com/blog/xmpp-extensible-messaging-presence-protocol>
- Cullen, J., Henrik, B., & Jan-Ivar, B. (2021, January 26). *WebRTC 1.0: Real-Time Communication Between Browsers*. Retrieved from w3.org: <https://www.w3.org/TR/webrtc/>
- Dejan, S., Matija, H., & Sinisa, S. (2014). Performance evaluation of WebSocket protocol for implementation of full-duplex web streams. *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. researchgate.
- Enyinnaya, C. (2021, March 30). *What are WebSockets*. Retrieved from cometchat.com: <https://www.cometchat.com/tutorials/what-is-websockets#:~:text=A%20WebSocket%20is%20a%20persistent,between%20a%20client%20and%20server.>
- Esegece admin. (2020, May 16). *WebSockets .NET*. Retrieved from esegece.com: <https://www.esegece.com/websockets/main-components/net-components>
- I, F., & A, M. (2011, December). *The WebSocket Protocol*. Retrieved from rfc-editor.org: <https://www.rfc-editor.org/rfc/rfc6455.html>
- IETF Trust. (2011). *The WebSocket Protocol*. Retrieved from tools.ietf.org: <https://tools.ietf.org/id/draft-ietf-hybi-thewebsocketprotocol-09.html>
- Implementing WebSocket Protocol in Go*. (2017, May 07). Retrieved from hassansin.github.io: <https://hassansin.github.io/implementing-websocket-protocol-in-go>
- Jabber hot chilli. (2022). *Jabber/XMPP*. Retrieved from jabber.hot-chilli.net: <https://jabber.hot-chilli.net/>
- Java Dev Journal Contributors. (n.d.). *Spring Boot Tutorials*. Retrieved from javadevjournal.com: <https://www.javadevjournal.com/spring-boot/>
- Javatpoint Contributors. (n.d.). *Spring Boot Tutorial*. Retrieved from javatpoint.com: <https://www.javatpoint.com/spring-boot-tutorial>
- LEDBROOK, P. (2010, June 14). *Understanding AMQP, the protocol used by RabbitMQ*. Retrieved from spring.io: <https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq>

- Liakh, A. (2020, November 08). *WebSockets With Spring, Part 3: STOMP Over WebSocket*. Retrieved from medium.com: <https://medium.com/swlh/websockets-with-spring-part-3-stomp-over-websocket-3dab4a21f397>
- LX Pty Ltd. (2014, March 27). *XMPP – an extensible messaging protocol for the IoT*. Retrieved from lx-group.com.au: <https://lx-group.com.au/xmpp-extensible-messaging-protocol-iot/>
- M. Jones. (2009, September 14). *Meet the Extensible Messaging and Presence Protocol (XMPP)*. Retrieved from developer.ibm.com: <https://developer.ibm.com/tutorials/xmppintro/>
- MDN Contributors. (2022, June 03). *The WebSocket API (WebSockets)*. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
- MDN Contributors. (2022, June 03). *WebRTC API*. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
- OASIS Members. (n.d.). *AMQP is the Internet Protocol for Business Messaging*. Retrieved from amqp.org: <https://www.amqp.org/about/what>
- OpenDomain.org. (n.d.). *An Overview of XMPP*. Retrieved from xmpp.org: <https://xmpp.org/about/technology-overview/>
- Programmer Sought Contributors. (n.d.). *Spring uses WebSocket and STOMP to realize the message function*. Retrieved from programmersought.com: <https://www.programmersought.com/article/34333859023/>
- RabbitMQ Community. (n.d.). *Management Plugin*. Retrieved from rabbitmq.com: <https://www.rabbitmq.com/management.html#:~:text=Management%20UI%20and%20External%20Monitoring%20Systems>
- Redhat Contributors. (n.d.). *AMQP Exchange Types*. Retrieved from access.redhat.com: https://access.redhat.com/documentation/en-us/red_hat_enterprise_mrg/3/html/messaging_programming_reference/amqp_exchange_types
- Reock, J. (2020, June 04). *What is Apache ActiveMQ?* Retrieved from openlogic.com: <https://www.openlogic.com/blog/what-apache-activemq>
- RF & Wireless Vendors and Resources. (n.d.). *What is XMPP Protocol in IoT | XMPP Server | XMPP Client*. Retrieved from rfwireless-world.com: <https://www.rfwireless-world.com/IoT/XMPP-protocol.html>
- Stomp*. (n.d.). Retrieved from stomp.github.io: <https://stomp.github.io/>
- Stomp Protocol Specification, Version 1.0*. (n.d.). Retrieved from stomp.github.io: <https://stomp.github.io/stomp-specification-1.0.html>

STOMP Protocol Specification, Version 1.1. (n.d.). Retrieved from [stomp.github.io](https://stomp.github.io/stomp-specification-1.1.html):
<https://stomp.github.io/stomp-specification-1.1.html>

STOMP Protocol Specification, Version 1.2. (2012, October 22). Retrieved from [stomp.github.io](https://stomp.github.io/stomp-specification-1.2.html): <https://stomp.github.io/stomp-specification-1.2.html>

The Developer Blog Contributors. (n.d.). *Spring Boot Tutorial*. Retrieved from [thedeveloperblog.com](https://thedeveloperblog.com/spring/spring-boot-tutorial): <https://thedeveloperblog.com/spring/spring-boot-tutorial>

W3CSchool Contributors. (n.d.). *Spring Boot Tutorial*. Retrieved from [w3school.com](https://w3school.com/spring-boot-tutorial):
<https://w3school.com/spring-boot-tutorial>