# [Document view definitions](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/" \l "view-document)

org.springframework.web.servlet.view.document

# 26 [WebSocket支持](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters/spring-boot-starter-websocket)

参考文档的这一部分涉及Spring框架在Web应用程序中支持WebSocket风格消息传递，包括使用STOMP作为应用层WebSocket子协议。

第1部分，“介绍”框架的建立思想，包括采用的要求,，设计的考虑，以及一些合适的想法。

第2部分“WebSocketAPI”查看服务器端的Spring WebSocketAPI，

第3部分“SockJS Fallback Options”解释了SOCKJS协议，并展示了如何配置和使用它。

第4.1部分“STOP概述”介绍了STOMP消息传递协议。

第4.2部分中，“Enable STOMP over WebSocke(在websocket上开启stomp)”演示如何在Spring添加和配置STOMP。

第4.4部分“Annotation Message Handling（注解消息的行为）”，下面的部分说明如何编写带注解的消息处理方法、发送消息、选择消息代理选项。

第4.17部分 “测试Controller上的注解方法”列出了测试STOMP/WebSocket应用程序的三种方法。

## 第1部分引言

WebSocket协议RFC 6455为Web应用程序定义了一项重要的新功能：客户端和服务器之间的全双工，双向通信。这是一个令人激动的新功能，它使得Web更具交互性，包括Java Applets，XMLHttpRequest，Adobe Flash，ActiveXObject，各种Comet技术，服务器发送的事件等。

对WebSoSt协议的正确介绍超出了本文的范围。然而，了解HTTP的初始握手过程是很重要的，它依赖于内置到HTTP中的机制来请求协议升级（或者在这种情况下是协议交换），如果服务器同意，将使用HTTP状态101（交换协议）响应。假设握手成功，HTTP升级请求下的TCP套接字保持打开状态，客户端和服务器都可以使用它来向彼此发送消息。

Spring Framework 4包含一个spring-websocket全面的WebSocket支持的新模块。它与Java WebSocket API标准（JSR-356）兼容，并且还提供了额外的增值功能，如引言的其余部分所述。

WebSocket RFC定义了子协议的使用 。在握手期间，客户端和服务器可以使用标头来 Sec-WebSocket-Protocol协商一个子协议，即使用更高的应用级协议。不需要使用子协议，但即使不使用，应用程序仍然需要选择客户端和服务器都可以理解的消息格式。该格式可以是定制的，框架特定的或标准的消息传递协议。

Spring框架为使用STOMP提供了支持  - STOMP是一种简单的消息传递协议，最初创建用于脚本语言，其框架受HTTP的启发。STOMP得到了广泛的支持，非常适合通过WebSocket和网络使用。

## 第2部分引言

## 第3部分引言

## 第4部分引言

## 26. WebSocket支持

参考文档的这一部分介绍Spring Framework对Web应用程序中WebSocket风格的消息传递的支持，包括将STOMP用作应用程序级WebSocket子协议。

[第26.1节“简介”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-intro)为思考WebSocket建立了一个思维框架，涵盖了采用挑战，设计考虑以及何时适合它的想法。

[第26.2节“WebSocket API”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-server)回顾了服务器端的Spring WebSocket API，而 [第26.3节“SockJS回退选项”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-fallback)解释了SockJS协议并展示了如何配置和使用它。

[第26.4.1节“STOMP概述”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-overview)介绍了STOMP消息传递协议。 [第26.4.2节“通过WebSocket启用STOMP”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-enable)演示了如何在Spring中配置STOMP支持。 [第26.4.4节“注释消息处理”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-handle-annotations)和以下各节介绍了如何编写带注释的消息处理方法，发送消息，选择消息代理选项以及如何处理特殊的“用户”目标。最后， [第26.4.17节“测试注释控制器方法”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-testing)列出了测试STOMP / WebSocket应用程序的三种方法。

## 26.1简介

WebSocket协议[RFC 6455](https://tools.ietf.org/html/rfc6455)为Web应用程序定义了一项重要的新功能：客户端和服务器之间的全双工，双向通信。这是一个令人激动的新功能，它使得Web更具交互性，包括Java Applets，XMLHttpRequest，Adobe Flash，ActiveXObject，各种Comet技术，服务器发送的事件等。

WebSocket协议的正确介绍超出了本文的范围。但至少要了解HTTP只用于初始握手，这依赖于内置于HTTP中的机制来请求协议升级（或者在此情况下为协议交换机），服务器可以使用HTTP状态101 （切换协议），如果它同意。假设握手成功，HTTP升级请求下的TCP套接字保持打开状态，并且客户端和服务器都可以使用它将消息发送给对方。

Spring Framework 4包含一个spring-websocket全面的WebSocket支持的新模块。它与Java WebSocket API标准（[JSR-356](https://jcp.org/en/jsr/detail?id=356)）兼容，并且还提供了额外的增值功能，如引言的其余部分所述。

### 26.1.1 WebSocket后备选项

采用的一个重要挑战是在某些浏览器中缺乏对WebSocket的支持。值得注意的是，支持WebSocket的第一个Internet Explorer版本是版本10（请参阅[http://caniuse.com/websockets](https://caniuse.com/websockets)获得浏览器版本的支持）。此外，一些限制性代理可能会被配置为阻止尝试进行HTTP升级，或者在一段时间后中断连接，因为它仍然打开太久。InfoQ文章[“HTML5 Web套接字如何与代理服务器互动”中](http://www.infoq.com/articles/Web-Sockets-Proxy-Servers)提供了Peter Lubbers关于此主题的良好概述 。

因此，为了今天构建一个WebSocket应用程序，需要使用回退选项，以便在必要时模拟WebSocket API。Spring框架提供了基于[SockJS协议的](https://github.com/sockjs/sockjs-protocol)透明回退选项。这些选项可以通过配置启用，否则不需要修改应用程序。

### 26.1.2消息体系结构

除了短期到中期的采用挑战之外，使用WebSocket提出重要的设计考虑因素，这对于早期认识很重要，特别是与我们现在了解的关于构建Web应用程序的知识相反。

今天，REST是一种广泛接受，理解和支持的构建Web应用程序的体系结构。它是一种依赖于具有许多URL（名词），一些HTTP方法（动词）以及诸如使用超媒体（链接），保持无状态等其他原理的体系结构。

相比之下，WebSocket应用程序可能仅使用单个URL来进行初始HTTP握手。之后的所有消息将共享并在同一个TCP连接上流动。这指向一个完全不同的，异步的，事件驱动的消息体系结构。一种与传统消息传递应用（例如JMS，AMQP）更接近的产品。

Spring框架4包括一个新的spring-messaging具有关键的抽象概念从模块 [Spring集成](https://projects.spring.io/spring-integration/)项目，例如Message，MessageChannel，MessageHandler，和其他人，可以作为一个基础，这样的消息架构。该模块还包含一组用于将消息映射到方法的注释，类似于基于Spring MVC注释的编程模型。

### 26.1.3 WebSocket中的子协议支持

WebSocket确实暗示了消息传递体系结构，但并未要求使用任何特定的消息传递协议。它是一个非常薄的TCP层，可将字节流转换为消息流（文本或二进制文件），而不是更多。应用程序需要解释消息的含义。

与HTTP（应用程序级协议）不同，在WebSocket协议中，传入消息中没有足够的信息可供框架或容器知道如何路由或处理它。因此WebSocket对于任何事物来说都是很低的水平，除了一个非常平凡的应用程序。它可以完成，但它可能会导致在顶部创建一个框架。这与现在大多数Web应用程序如何使用Web框架而不是单独使用Servlet API编写相当。

为此，WebSocket RFC定义了[子协议](https://tools.ietf.org/html/rfc6455#section-1.9)的使用 。在握手期间，客户端和服务器可以使用标头来 Sec-WebSocket-Protocol协商一个子协议，即使用更高的应用级协议。不需要使用子协议，但即使不使用，应用程序仍然需要选择客户端和服务器都可以理解的消息格式。该格式可以是定制的，框架特定的或标准的消息传递协议。

Spring框架为使用[STOMP](https://stomp.github.io/stomp-specification-1.2.html#Abstract)提供了支持  - [STOMP](https://stomp.github.io/stomp-specification-1.2.html#Abstract)是一种简单的消息传递协议，最初创建用于脚本语言，其框架受HTTP的启发。STOMP得到了广泛的支持，非常适合通过WebSocket和网络使用。

### 26.1.4我应该使用WebSocket吗？

考虑到围绕使用WebSocket的所有设计考虑因素，有必要问“什么时候适合使用？”。

最适合WebSocket的是Web应用程序，客户端和服务器需要以高频率和低延迟交换事件。总理候选人包括但不限于金融，游戏，协作等应用。这种应用对时间延迟都非常敏感，并且需要以高频交换各种各样的消息。

但对于其他应用程序类型，情况可能并非如此。例如，新闻或社交供稿在每几分钟进行一次简单的投票即可显示突发新闻。这里的延迟很重要，但如果新闻需要几分钟才能显示，这是可以接受的。

即使在延迟很重要的情况下，如果消息量相对较低（例如监测网络故障），[长轮询](https://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates)的使用 应被视为相对简单的替代方案，可以可靠地工作，并且在效率方面具有可比性（再次假设体积的消息相对较低）。

这是消息的低延迟和高频率的组合，可以使WebSocket协议的使用变得至关重要。即使在这样的应用程序中，选择仍然是否应该通过WebSocket消息完成所有的客户端 - 服务器通信，而不是使用HTTP和REST。答案会因应用而异; 然而，很可能某些功能可能暴露在WebSocket和REST API上，以便为客户提供替代方案。此外，REST API调用可能需要向通过WebSocket连接的感兴趣的客户端广播消息。

Spring框架允许@Controller和@RestController类具有HTTP请求处理和WebSocket消息处理方法。此外，Spring MVC请求处理方法或任何应用程序方法可以轻松地向所有感兴趣的WebSocket客户端或特定用户广播消息。

## 26.2 WebSocket API

Spring框架提供了一个WebSocket API，以适应各种WebSocket引擎。目前，该列表包括Tomcat 7.0.47 +，Jetty 9.1+，GlassFish 4.1+，WebLogic 12.1.3+和Undertow 1.0+（以及WildFly 8.0+）等WebSocket运行时。随着更多WebSocket运行时可用，可能会添加其他支持。

|  |
| --- |
| [注意] |
| 正如解释的[出台](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-intro-sub-protocol)，直接使用的WebSocket API的太低级的应用-直至假设有关消息的格式作出很少有一个框架可以做解释通过注释的消息或路由他们。这就是为什么应用程序应该考虑使用子协议和Spring的[STOMP over WebSocket](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp)支持。  当使用更高级别的协议时，WebSocket API的细节变得不那么相关，就像使用HTTP时TCP通信的细节未暴露给应用程序一样。尽管如此，本节将直接介绍使用WebSocket的细节。 |

### 26.2.1创建并配置WebSocketHandler

创建的WebSocket服务器是为实现简单WebSocketHandler或更可能要么延长TextWebSocketHandler或BinaryWebSocketHandler：

**import** org.springframework.web.socket.WebSocketHandler;

**import** org.springframework.web.socket.WebSocketSession;

**import** org.springframework.web.socket.TextMessage;

**公共** **类** MyHandler **扩展** TextWebSocketHandler {

*@Override*

**public**  **void** handleTextMessage（WebSocketSession session，TextMessage message）{

*// ...*

}

}

有专门的WebSocket Java-config和XML命名空间支持将上述WebSocket处理程序映射到特定的URL：

**import** org.springframework.web.socket.config.annotation.EnableWebSocket;

**import** org.springframework.web.socket.config.annotation.WebSocketConfigurer;

**import** org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

*@Configuration*

*@EnableWebSocket*

**公共** **类** WebSocketConfig**实现** WebSocketConfigurer {

*@Override*

**public**  **void** registerWebSocketHandlers（WebSocketHandlerRegistry registry）{

registry.addHandler（myHandler（），“/ myHandler”）;

}

*@Bean*

**public** WebSocketHandler myHandler（）{

**return**  **new** MyHandler（）;

}

}

等同的XML配置：

<beans xmlns = “http://www.springframework.org/schema/beans”

xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”

xmlns：websocket = “http：// www .springframework.org / schema / websocket“

xsi：schemaLocation = ”

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd“ >

<websocket：handlers>

<websocket：mapping path = “/ myHandler” handler = “myHandler” />

</ websocket：handlers>

<bean id = “myHandler” class = “org.springframework.samples.MyHandler” />

</豆>

以上内容适用于Spring MVC应用程序，并应包含在[DispatcherServlet](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#mvc-servlet)的配置中。但是，Spring的WebSocket支持不依赖于Spring MVC。WebSocketHandler 在[WebSocketHttpRequestHandler](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/web/socket/server/support/WebSocketHttpRequestHandler.html)的帮助下将其集成到其他HTTP服务环境中 相对简单。

### 26.2.2自定义WebSocket握手

定制初始HTTP WebSocket握手请求的最简单方法是通过a HandshakeInterceptor，它在握手方法的“之前”和“之后”公开。这种拦截器可以用来阻止握手或使任何属性可用WebSocketSession。例如，有一个用于将HTTP会话属性传递给WebSocket会话的内置拦截器：

*@Configuration*

*@EnableWebSocket*

**公共** **类** WebSocketConfig**实现** WebSocketConfigurer {

*@Override*

**public**  **void** registerWebSocketHandlers（WebSocketHandlerRegistry registry）{

registry.addHandler（**new** MyHandler（），“/ myHandler”）

.addInterceptors（**new** HttpSessionHandshakeInterceptor（））;

}

}

和XML配置等效：

<beans xmlns = “http://www.springframework.org/schema/beans”

xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”

xmlns：websocket = “http：// www .springframework.org / schema / websocket“

xsi：schemaLocation = ”

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd“ >

<websocket：handlers>

<websocket：mapping path = “/ myHandler” handler = “myHandler” />

<websocket：handshake-interceptors>

<bean class = “org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor” />

</ websocket：handshake-interceptors>

</ websocket：handlers>

<bean id = “myHandler” class = “org.springframework.samples.MyHandler” />

</豆>

更高级的选项是扩展DefaultHandshakeHandler执行WebSocket握手步骤，包括验证客户端来源，协商子协议等。如果需要配置自定义RequestUpgradeStrategy以适应尚未支持的WebSocket服务器引擎和版本，应用程序可能还需要使用此选项（有关此主题的更多信息[，](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-server-deployment)请参见[第26.2.4节“部署注意事项”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-server-deployment)） 。Java-config和XML命名空间都可以配置自定义 HandshakeHandler。

### 26.2.3 WebSocketHandler装饰

Spring提供了一个WebSocketHandlerDecorator基类，可以用来修饰WebSocketHandler其他行为。在使用WebSocket Java-config或XML命名空间时，默认情况下会提供并添加日志记录和异常处理实现。在ExceptionWebSocketHandlerDecorator捕获所有捕获的异常任何WebSocketHandler方法所产生的，并与状态关闭WebSocket的会议1011指示服务器错误。

### 26.2.4部署注意事项

Spring WebSocket API很容易集成到一个Spring MVC应用程序中，该应用程序DispatcherServlet既提供HTTP WebSocket握手，也提供其他HTTP请求。通过调用也可以轻松地集成到其他HTTP处理场景中WebSocketHttpRequestHandler。这很方便，易于理解。但是，JSR-356运行时需要特别注意。

Java WebSocket API（JSR-356）提供了两种部署机制。第一个涉及启动时的Servlet容器类路径扫描（Servlet 3功能）; 另一个是在Servlet容器初始化时使用的注册API。这两种机制都不能使用单个“前端控制器”来处理所有HTTP处理 - 包括WebSocket握手和所有其他HTTP请求 - 例如Spring MVC's DispatcherServlet。

这是JSR-356的一个重要限制，Spring的WebSocket支持RequestUpgradeStrategy即使在JSR-356运行时运行时也提供特定于服务器的地址。

|  |
| --- |
| [注意] |
| 已经创建了一个请求来克服Java WebSocket API中的上述限制，并且可以在[WEBSOCKET\_SPEC-211中](https://java.net/jira/browse/WEBSOCKET_SPEC-211)进行跟踪 。另请注意，Tomcat和Jetty已经提供了本地API替代方案，可以轻松克服这一限制。我们希望更多的服务器能够遵循他们的例子，而不管它在Java WebSocket API中的地址是什么时候。 |

第二个考虑因素是具有JSR-356支持的Servlet容器有望执行ServletContainerInitializer（SCI）扫描，可能会显着降低应用程序的启动速度。如果在升级到具有JSR-356支持的Servlet容器版本后观察到重大影响，则应该可以通过使用以下内容中的<absolute-ordering />元素来选择性地启用或禁用Web碎片（和SCI扫描）web.xml：

<web-app xmlns = “http://java.sun.com/xml/ns/javaee”

xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”

xsi：schemaLocation = “

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd”

版本 = “3.0” >

<绝对排序/>

</ web的应用>

然后，您可以选择性地按名称启用Web片段，例如Spring自己 SpringServletContainerInitializer提供对Servlet 3 Java初始化API的支持（如果需要）：

<web-app xmlns = “http://java.sun.com/xml/ns/javaee”

xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”

xsi：schemaLocation = “

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd”

版本 = “3.0” >

<absolute-ordering>

<name> spring\_web </ name>

</ absolute-ordering>

</ web的应用>

### 26.2.5配置WebSocket引擎

每个底层WebSocket引擎都公开可控制运行时特性的配置属性，例如消息缓冲区大小，空闲超时等等。

对于Tomcat，WildFly和GlassFish将添加ServletServerContainerFactoryBean到您的WebSocket Java配置中：

*@Configuration*

*@EnableWebSocket*

**公共** **类** WebSocketConfig**实现** WebSocketConfigurer {

*@Bean*

**public** ServletServerContainerFactoryBean createWebSocketContainer（）{

ServletServerContainerFactoryBean container = **new** ServletServerContainerFactoryBean（）;

container.setMaxTextMessageBufferSize（8192）;

container.setMaxBinaryMessageBufferSize（8192）;

**返回**容器;

}

}

或WebSocket XML命名空间：

<beans xmlns = “http://www.springframework.org/schema/beans”

xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”

xmlns：websocket = “http：// www .springframework.org / schema / websocket“

xsi：schemaLocation = ”

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd“ >

<bean class = “org.springframework ... ServletServerContainerFactoryBean” >

<property name = “maxTextMessageBufferSize” value = “8192” />

<property name = “maxBinaryMessageBufferSize” value = “8192” />

</ bean>

</豆>

|  |
| --- |
| [注意] |
| 对于客户端WebSocket配置，您应该使用WebSocketContainerFactoryBean （XML）或ContainerProvider.getWebSocketContainer()（Java配置）。 |

对于Jetty，您需要提供预配置的Jetty，WebSocketServerFactory并DefaultHandshakeHandler通过WebSocket Java配置将其插入到Spring中：

*@Configuration*

*@EnableWebSocket*

**public** **class** WebSocketConfig **implements** WebSocketConfigurer {

*@Override*

**public** **void** registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(echoWebSocketHandler(),

"/echo").setHandshakeHandler(handshakeHandler());

}

*@Bean*

**public** DefaultHandshakeHandler handshakeHandler() {

WebSocketPolicy policy = **new** WebSocketPolicy(WebSocketBehavior.SERVER);

policy.setInputBufferSize(8192);

policy.setIdleTimeout(600000);

**return** **new** DefaultHandshakeHandler(

**new** JettyRequestUpgradeStrategy(**new** WebSocketServerFactory(policy)));

}

}

or WebSocket XML namespace:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/echo" handler="echoHandler"/>

<websocket:handshake-handler ref="handshakeHandler"/>

</websocket:handlers>

<bean id="handshakeHandler" class="org.springframework...DefaultHandshakeHandler">

<constructor-arg ref="upgradeStrategy"/>

</bean>

<bean id="upgradeStrategy" class="org.springframework...JettyRequestUpgradeStrategy">

<constructor-arg ref="serverFactory"/>

</bean>

<bean id="serverFactory" class="org.eclipse.jetty...WebSocketServerFactory">

<constructor-arg>

<bean class="org.eclipse.jetty...WebSocketPolicy">

<constructor-arg value="SERVER"/>

<property name="inputBufferSize" value="8092"/>

<property name="idleTimeout" value="600000"/>

</bean>

</constructor-arg>

</bean>

</beans>

### 26.2.6 Configuring allowed origins

As of Spring Framework 4.1.5, the default behavior for WebSocket and SockJS is to accept only same origin requests. It is also possible to allow all or a specified list of origins. This check is mostly designed for browser clients. There is nothing preventing other types of clients from modifying the Origin header value (see [RFC 6454: The Web Origin Concept](https://tools.ietf.org/html/rfc6454) for more details).

The 3 possible behaviors are:

* Allow only same origin requests (default): in this mode, when SockJS is enabled, the Iframe HTTP response header X-Frame-Options is set to SAMEORIGIN, and JSONP transport is disabled since it does not allow to check the origin of a request. As a consequence, IE6 and IE7 are not supported when this mode is enabled.
* Allow a specified list of origins: each provided allowed origin must start with http:// or https://. In this mode, when SockJS is enabled, both IFrame and JSONP based transports are disabled. As a consequence, IE6 through IE9 are not supported when this mode is enabled.
* Allow all origins: to enable this mode, you should provide \* as the allowed origin value. In this mode, all transports are available.

WebSocket and SockJS allowed origins can be configured as shown bellow:

**import** org.springframework.web.socket.config.annotation.EnableWebSocket;

**import** org.springframework.web.socket.config.annotation.WebSocketConfigurer;

**import** org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

*@Configuration*

*@EnableWebSocket*

**public** **class** WebSocketConfig **implements** WebSocketConfigurer {

*@Override*

**public** **void** registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(myHandler(), "/myHandler").setAllowedOrigins("http://mydomain.com");

}

*@Bean*

**public** WebSocketHandler myHandler() {

**return** **new** MyHandler();

}

}

XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers allowed-origins="http://mydomain.com">

<websocket:mapping path="/myHandler" handler="myHandler" />

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

## 26.3 SockJS Fallback Options

As explained in the [introduction](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-into-fallback-options), WebSocket is not supported in all browsers yet and may be precluded by restrictive network proxies. This is why Spring provides fallback options that emulate the WebSocket API as close as possible based on the [SockJS protocol](https://github.com/sockjs/sockjs-protocol) (version 0.3.3).

### 26.3.1 Overview of SockJS

The goal of SockJS is to let applications use a WebSocket API but fall back to non-WebSocket alternatives when necessary at runtime, i.e. without the need to change application code.

SockJS consists of:

* The [SockJS protocol](https://github.com/sockjs/sockjs-protocol) defined in the form of executable [narrated tests](https://sockjs.github.io/sockjs-protocol/sockjs-protocol-0.3.3.html).
* The [SockJS JavaScript client](https://github.com/sockjs/sockjs-client/) - a client library for use in browsers.
* SockJS server implementations including one in the Spring Framework spring-websocket module.
* As of 4.1 spring-websocket also provides a SockJS Java client.

SockJS is designed for use in browsers. It goes to great lengths to support a wide range of browser versions using a variety of techniques. For the full list of SockJS transport types and browsers see the [SockJS client](https://github.com/sockjs/sockjs-client/) page. Transports fall in 3 general categories: WebSocket, HTTP Streaming, and HTTP Long Polling. For an overview of these categories see [this blog post](https://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/).

The SockJS client begins by sending "GET /info" to obtain basic information from the server. After that it must decide what transport to use. If possible WebSocket is used. If not, in most browsers there is at least one HTTP streaming option and if not then HTTP (long) polling is used.

All transport requests have the following URL structure:

http://host:port/myApp/myEndpoint/{server-id}/{session-id}/{transport}

* {server-id} - useful for routing requests in a cluster but not used otherwise.
* {session-id} - correlates HTTP requests belonging to a SockJS session.
* {transport} - indicates the transport type, e.g. "websocket", "xhr-streaming", etc.

The WebSocket transport needs only a single HTTP request to do the WebSocket handshake. All messages thereafter are exchanged on that socket.

HTTP transports require more requests. Ajax/XHR streaming for example relies on one long-running request for server-to-client messages and additional HTTP POST requests for client-to-server messages. Long polling is similar except it ends the current request after each server-to-client send.

SockJS adds minimal message framing. For example the server sends the letter o ("open" frame) initially, messages are sent as a["message1","message2"] (JSON-encoded array), the letter h ("heartbeat" frame) if no messages flow for 25 seconds by default, and the letter c ("close" frame) to close the session.

To learn more, run an example in a browser and watch the HTTP requests. The SockJS client allows fixing the list of transports so it is possible to see each transport one at a time. The SockJS client also provides a debug flag which enables helpful messages in the browser console. On the server side enable TRACE logging for org.springframework.web.socket. For even more detail refer to the SockJS protocol [narrated test](https://sockjs.github.io/sockjs-protocol/sockjs-protocol-0.3.3.html).

### 26.3.2 Enable SockJS

SockJS is easy to enable through Java configuration:

*@Configuration*

*@EnableWebSocket*

**public** **class** WebSocketConfig **implements** WebSocketConfigurer {

*@Override*

**public** **void** registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(myHandler(), "/myHandler").withSockJS();

}

*@Bean*

**public** WebSocketHandler myHandler() {

**return** **new** MyHandler();

}

}

and the XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/myHandler" handler="myHandler"/>

<websocket:sockjs/>

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

The above is for use in Spring MVC applications and should be included in the configuration of a [DispatcherServlet](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#mvc-servlet). However, Spring’s WebSocket and SockJS support does not depend on Spring MVC. It is relatively simple to integrate into other HTTP serving environments with the help of [SockJsHttpRequestHandler](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/web/socket/sockjs/support/SockJsHttpRequestHandler.html).

On the browser side, applications can use the [sockjs-client](https://github.com/sockjs/sockjs-client/) (version 1.0.x) that emulates the W3C WebSocket API and communicates with the server to select the best transport option depending on the browser it’s running in. Review the [sockjs-client](https://github.com/sockjs/sockjs-client/) page and the list of transport types supported by browser. The client also provides several configuration options, for example, to specify which transports to include.

### 26.3.3 HTTP Streaming in IE 8, 9: Ajax/XHR vs IFrame

Internet Explorer 8 and 9 are and will remain common for some time. They are a key reason for having SockJS. This section covers important considerations about running in those browsers.

The SockJS client supports Ajax/XHR streaming in IE 8 and 9 via Microsoft’s [XDomainRequest](https://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx). That works across domains but does not support sending cookies. Cookies are very often essential for Java applications. However since the SockJS client can be used with many server types (not just Java ones), it needs to know whether cookies matter. If so the SockJS client prefers Ajax/XHR for streaming or otherwise it relies on a iframe-based technique.

The very first "/info" request from the SockJS client is a request for information that can influence the client’s choice of transports. One of those details is whether the server application relies on cookies, e.g. for authentication purposes or clustering with sticky sessions. Spring’s SockJS support includes a property called sessionCookieNeeded. It is enabled by default since most Java applications rely on the JSESSIONID cookie. If your application does not need it, you can turn off this option and the SockJS client should choose xdr-streaming in IE 8 and 9.

If you do use an iframe-based transport, and in any case, it is good to know that browsers can be instructed to block the use of IFrames on a given page by setting the HTTP response header X-Frame-Options to DENY, SAMEORIGIN, or ALLOW-FROM <origin>. This is used to prevent [clickjacking](https://www.owasp.org/index.php/Clickjacking).

|  |
| --- |
| [注意] |
| Spring Security 3.2+ provides support for setting X-Frame-Options on every response. By default the Spring Security Java config sets it to DENY. In 3.2 the Spring Security XML namespace does not set that header by default but may be configured to do so, and in the future it may set it by default.  See [Section 7.1. "Default Security Headers"](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#headers) of the Spring Security documentation for details on how to configure the setting of the X-Frame-Optionsheader. You may also check or watch [SEC-2501](https://jira.spring.io/browse/SEC-2501) for additional background. |

If your application adds the X-Frame-Options response header (as it should!) and relies on an iframe-based transport, you will need to set the header value toSAMEORIGIN or ALLOW-FROM <origin>. Along with that the Spring SockJS support also needs to know the location of the SockJS client because it is loaded from the iframe. By default the iframe is set to download the SockJS client from a CDN location. It is a good idea to configure this option to a URL from the same origin as the application.

In Java config this can be done as shown below. The XML namespace provides a similar option via the <websocket:sockjs> element:

*@Configuration*

*@EnableWebSocket*

**public** **class** WebSocketConfig **implements** WebSocketConfigurer {

*@Override*

**public** **void** registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS()

.setClientLibraryUrl("http://localhost:8080/myapp/js/sockjs-client.js");

}

*// ...*

}

|  |
| --- |
| [注意] |
| During initial development, do enable the SockJS client devel mode that prevents the browser from caching SockJS requests (like the iframe) that would otherwise be cached. For details on how to enable it see the [SockJS client](https://github.com/sockjs/sockjs-client/) page. |

### 26.3.4 Heartbeat Messages

The SockJS protocol requires servers to send heartbeat messages to preclude proxies from concluding a connection is hung. The Spring SockJS configuration has a property called heartbeatTime that can be used to customize the frequency. By default a heartbeat is sent after 25 seconds assuming no other messages were sent on that connection. This 25 seconds value is in line with the following [IETF recommendation](https://tools.ietf.org/html/rfc6202) for public Internet applications.

|  |
| --- |
| [注意] |
| When using STOMP over WebSocket/SockJS, if the STOMP client and server negotiate heartbeats to be exchanged, the SockJS heartbeats are disabled. |

The Spring SockJS support also allows configuring the TaskScheduler to use for scheduling heartbeats tasks. The task scheduler is backed by a thread pool with default settings based on the number of available processors. Applications should consider customizing the settings according to their specific needs.

### 26.3.5 Servlet 3 Async Requests

HTTP streaming and HTTP long polling SockJS transports require a connection to remain open longer than usual. For an overview of these techniques see [this blog post](https://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/).

In Servlet containers this is done through Servlet 3 async support that allows exiting the Servlet container thread processing a request and continuing to write to the response from another thread.

A specific issue is that the Servlet API does not provide notifications for a client that has gone away, see [SERVLET\_SPEC-44](https://java.net/jira/browse/SERVLET_SPEC-44). However, Servlet containers raise an exception on subsequent attempts to write to the response. Since Spring’s SockJS Service supports sever-sent heartbeats (every 25 seconds by default), that means a client disconnect is usually detected within that time period or earlier if messages are sent more frequently.

|  |
| --- |
| [注意] |
| As a result network IO failures may occur simply because a client has disconnected, which can fill the log with unnecessary stack traces. Spring makes a best effort to identify such network failures that represent client disconnects (specific to each server) and log a minimal message using the dedicated log category DISCONNECTED\_CLIENT\_LOG\_CATEGORY defined in AbstractSockJsSession. If you need to see the stack traces, set that log category to TRACE. |

### 26.3.6 CORS Headers for SockJS

If you allow cross-origin requests (see [Section 26.2.6, “Configuring allowed origins”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-server-allowed-origins)), the SockJS protocol uses CORS for cross-domain support in the XHR streaming and polling transports. Therefore CORS headers are added automatically unless the presence of CORS headers in the response is detected. So if an application is already configured to provide CORS support, e.g. through a Servlet Filter, Spring’s SockJsService will skip this part.

It is also possible to disable the addition of these CORS headers via the suppressCors property in Spring’s SockJsService.

The following is the list of headers and values expected by SockJS:

* "Access-Control-Allow-Origin" - initialized from the value of the "Origin" request header.
* "Access-Control-Allow-Credentials" - always set to true.
* "Access-Control-Request-Headers" - initialized from values from the equivalent request header.
* "Access-Control-Allow-Methods" - the HTTP methods a transport supports (see TransportType enum).
* "Access-Control-Max-Age" - set to 31536000 (1 year).

For the exact implementation see addCorsHeaders in AbstractSockJsService as well as the TransportType enum in the source code.

Alternatively if the CORS configuration allows it consider excluding URLs with the SockJS endpoint prefix thus letting Spring’s SockJsService handle it.

### 26.3.7 SockJS Client

A SockJS Java client is provided in order to connect to remote SockJS endpoints without using a browser. This can be especially useful when there is a need for bidirectional communication between 2 servers over a public network, i.e. where network proxies may preclude the use of the WebSocket protocol. A SockJS Java client is also very useful for testing purposes, for example to simulate a large number of concurrent users.

The SockJS Java client supports the "websocket", "xhr-streaming", and "xhr-polling" transports. The remaining ones only make sense for use in a browser.

The WebSocketTransport can be configured with:

* StandardWebSocketClient in a JSR-356 runtime
* JettyWebSocketClient using the Jetty 9+ native WebSocket API
* Any implementation of Spring’s WebSocketClient

An XhrTransport by definition supports both "xhr-streaming" and "xhr-polling" since from a client perspective there is no difference other than in the URL used to connect to the server. At present there are two implementations:

* RestTemplateXhrTransport uses Spring’s RestTemplate for HTTP requests.
* JettyXhrTransport uses Jetty’s HttpClient for HTTP requests.

The example below shows how to create a SockJS client and connect to a SockJS endpoint:

List<Transport> transports = **new** ArrayList<>(2);

transports.add(**new** WebSocketTransport(**new** StandardWebSocketClient()));

transports.add(**new** RestTemplateXhrTransport());

SockJsClient sockJsClient = **new** SockJsClient(transports);

sockJsClient.doHandshake(**new** MyWebSocketHandler(), "ws://example.com:8080/sockjs");

|  |
| --- |
| [注意] |
| SockJS uses JSON formatted arrays for messages. By default Jackson 2 is used and needs to be on the classpath. Alternatively you can configure a custom implementation of SockJsMessageCodec and configure it on the SockJsClient. |

To use the SockJsClient for simulating a large number of concurrent users you will need to configure the underlying HTTP client (for XHR transports) to allow a sufficient number of connections and threads. For example with Jetty:

HttpClient jettyHttpClient = **new** HttpClient();

jettyHttpClient.setMaxConnectionsPerDestination(1000);

jettyHttpClient.setExecutor(**new** QueuedThreadPool(1000));

Consider also customizing these server-side SockJS related properties (see Javadoc for details):

*@Configuration*

**public** **class** WebSocketConfig **extends** WebSocketMessageBrokerConfigurationSupport {

*@Override*

**public** **void** registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/sockjs").withSockJS()

.setStreamBytesLimit(512 \* 1024)

.setHttpMessageCacheSize(1000)

.setDisconnectDelay(30 \* 1000);

}

*// ...*

}

## 26.4 STOMP Over WebSocket Messaging Architecture

The WebSocket protocol defines two types of messages, text and binary, but their content is undefined. It’s expected that the client and server may agree on using a sub-protocol (i.e. a higher-level protocol) to define message semantics. While the use of a sub-protocol with WebSocket is completely optional either way client and server will need to agree on some kind of protocol to help interpret messages.

### 26.4.1 Overview of STOMP

[STOMP](https://stomp.github.io/stomp-specification-1.2.html#Abstract) is a simple text-oriented messaging protocol that was originally created for scripting languages such as Ruby, Python, and Perl to connect to enterprise message brokers. It is designed to address a subset of commonly used messaging patterns. STOMP can be used over any reliable 2-way streaming network protocol such as TCP and WebSocket. Although STOMP is a text-oriented protocol, the payload of messages can be either text or binary.

STOMP is a frame based protocol whose frames are modeled on HTTP. The structure of a STOMP frame:

COMMAND

header1:value1

header2:value2

Body^@

Clients can use the SEND or SUBSCRIBE commands to send or subscribe for messages along with a "destination" header that describes what the message is about and who should receive it. This enables a simple publish-subscribe mechanism that can be used to send messages through the broker to other connected clients or to send messages to the server to request that some work be performed.

When using Spring’s STOMP support, the Spring WebSocket application acts as the STOMP broker to clients. Messages are routed to @Controller message-handling methods or to a simple, in-memory broker that keeps track of subscriptions and broadcasts messages to subscribed users. You can also configure Spring to work with a dedicated STOMP broker (e.g. RabbitMQ, ActiveMQ, etc) for the actual broadcasting of messages. In that case Spring maintains TCP connections to the broker, relays messages to it, and also passes messages from it down to connected WebSocket clients. Thus Spring web applications can rely on unified HTTP-based security, common validation, and a familiar programming model message-handling work.

Here is an example of a client subscribing to receive stock quotes which the server may emit periodically e.g. via a scheduled task sending messages through a SimpMessagingTemplate to the broker:

SUBSCRIBE

id:sub-1

destination:/topic/price.stock.\*

^@

Here is an example of a client sending a trade request, which the server may handle through an @MessageMapping method and later on, after the execution, broadcast a trade confirmation message and details down to the client:

SEND

destination:/queue/trade

content-type:application/json

content-length:44

{"action":"BUY","ticker":"MMM","shares",44}^@

The meaning of a destination is intentionally left opaque in the STOMP spec. It can be any string, and it’s entirely up to STOMP servers to define the semantics and the syntax of the destinations that they support. It is very common, however, for destinations to be path-like strings where "/topic/.." implies publish-subscribe (one-to-many) and "/queue/" implies point-to-point (one-to-one) message exchanges.

STOMP servers can use the MESSAGE command to broadcast messages to all subscribers. Here is an example of a server sending a stock quote to a subscribed client:

MESSAGE

message-id:nxahklf6-1

subscription:sub-1

destination:/topic/price.stock.MMM

{"ticker":"MMM","price":129.45}^@

It is important to know that a server cannot send unsolicited messages. All messages from a server must be in response to a specific client subscription, and the "subscription-id" header of the server message must match the "id" header of the client subscription.

The above overview is intended to provide the most basic understanding of the STOMP protocol. It is recommended to review the protocol [specification](https://stomp.github.io/stomp-specification-1.2.html) in full.

The benefits of using STOMP as a WebSocket sub-protocol:

* No need to invent a custom message format
* Use existing [stomp.js](https://github.com/jmesnil/stomp-websocket) client in the browser
* Ability to route messages to based on destination
* Option to use full-fledged message broker such as RabbitMQ, ActiveMQ, etc. for broadcasting

Most importantly the use of STOMP (vs plain WebSocket) enables the Spring Framework to provide a programming model for application-level use in the same way that Spring MVC provides a programming model based on HTTP.

### 26.4.2 Enable STOMP over WebSocket

The Spring Framework provides support for using STOMP over WebSocket through the spring-messaging and spring-websocket modules. Here is an example of exposing a STOMP WebSocket/SockJS endpoint at the URL path /portfolio where messages whose destination starts with "/app" are routed to message-handling methods (i.e. application work) and messages whose destinations start with "/topic" or "/queue" will be routed to the message broker (i.e. broadcasting to other connected clients):

**import** org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;

**import** org.springframework.web.socket.config.annotation.StompEndpointRegistry;

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **implements** WebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS();

}

*@Override*

**public** **void** configureMessageBroker(MessageBrokerRegistry config) {

config.setApplicationDestinationPrefixes("/app");

config.enableSimpleBroker("/topic", "/queue");

}

}

and in XML:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app">

<websocket:stomp-endpoint path="/portfolio">

<websocket:sockjs/>

</websocket:stomp-endpoint>

<websocket:simple-broker prefix="/topic, /queue"/>

</websocket:message-broker>

</beans>

|  |
| --- |
| [注意] |
| The "/app" prefix is arbitrary. You can pick any prefix. It’s simply meant to differentiate messages to be routed to message-handling methods to do application work vs messages to be routed to the broker to broadcast to subscribed clients.  The "/topic" and "/queue" prefixes depend on the broker in use. In the case of the simple, in-memory broker the prefixes do not have any special meaning; it’s merely a convention that indicates how the destination is used (pub-sub targetting many subscribers or point-to-point messages typically targeting an individual recipient). In the case of using a dedicated broker, most brokers use "/topic" as a prefix for destinations with pub-sub semantics and "/queue" for destinations with point-to-point semantics. Check the STOMP page of the broker to see the destination semantics it supports. |

On the browser side, a client might connect as follows using [stomp.js](https://github.com/jmesnil/stomp-websocket) and the [sockjs-client](https://github.com/sockjs/sockjs-client):

**var** socket = **new** SockJS("/spring-websocket-portfolio/portfolio");

**var** stompClient = Stomp.over(socket);

stompClient.connect({}, **function**(frame) {

}

Or if connecting via WebSocket (without SockJS):

**var** socket = **new** WebSocket("/spring-websocket-portfolio/portfolio");

**var** stompClient = Stomp.over(socket);

stompClient.connect({}, **function**(frame) {

}

Note that the stompClient above does not need to specify login and passcode headers. Even if it did, they would be ignored, or rather overridden, on the server side. See the sections [Section 26.4.8, “Connections To Full-Featured Broker”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-handle-broker-relay-configure) and [Section 26.4.10, “Authentication”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-authentication) for more information on authentication.

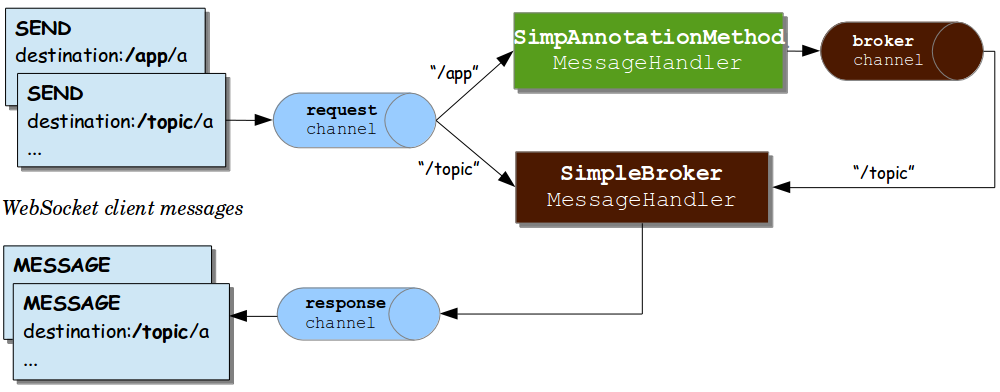
### 26.4.3 Flow of Messages

When a STOMP endpoint is configured, the Spring application acts as the STOMP broker to connected clients. This section provides a big picture overview of how messages flow within the application.

The spring-messaging module provides the foundation for asynchronous message processing. It contains a number of abstractions that originated in the [Spring Integration](https://spring.io/spring-integration) project and are intended for use as building blocks in messaging applications:

* [Message](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/Message.html) — a message with headers and a payload.
* [MessageHandler](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/MessageHandler.html) — a contract for handling a message.
* [MessageChannel](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/MessageChannel.html) — a contract for sending a message enabling loose coupling between senders and receivers.
* [SubscribableChannel](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/SubscribableChannel.html) — extends MessageChannel and sends messages to registered MessageHandler subscribers.
* [ExecutorSubscribableChannel](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/support/ExecutorSubscribableChannel.html) — a concrete implementation of SubscribableChannel that can deliver messages asynchronously via a thread pool.

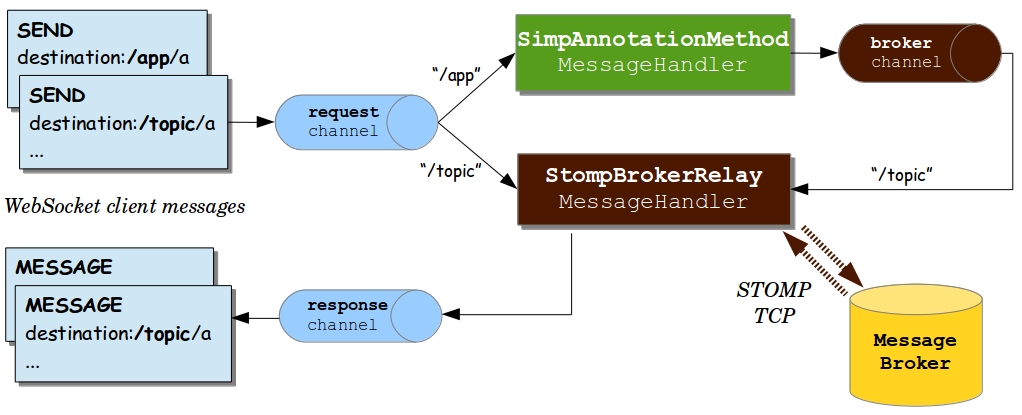
The @EnableWebSocketMessageBroker Java config and the <websocket:message-broker> XML config both assemble a concrete message flow. Below is a diagram of the part of the setup when using the simple, in-memory broker:



The above setup that includes 3 message channels:

* "clientInboundChannel" for messages from WebSocket clients.
* "clientOutboundChannel" for messages to WebSocket clients.
* "brokerChannel" for messages to the broker from within the application.

The same three channels are also used with a dedicated broker except here a "broker relay" takes the place of the simple broker:



Messages on the "clientInboundChannel" can flow to annotated methods for application handling (e.g. a stock trade execution request) or can be forwarded to the broker (e.g. client subscribing for stock quotes). The STOMP destination is used for simple prefix-based routing. For example the "/app" prefix could route messages to annotated methods while the "/topic" and "/queue" prefixes could route messages to the broker.

When a message-handling annotated method has a return type, its return value is sent as the payload of a Spring Message to the "brokerChannel". The broker in turn broadcasts the message to clients. Sending a message to a destination can also be done from anywhere in the application with the help of a messaging template. For example, an HTTP POST handling method can broadcast a message to connected clients, or a service component may periodically broadcast stock quotes.

Below is a simple example to illustrate the flow of messages:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **implements** WebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio");

}

*@Override*

**public** **void** configureMessageBroker(MessageBrokerRegistry registry) {

registry.setApplicationDestinationPrefixes("/app");

registry.enableSimpleBroker("/topic");

}

}

*@Controller*

**public** **class** GreetingController {

*@MessageMapping("/greeting")* {

**public** String handle(String greeting) {

**return** "[" + getTimestamp() + ": " + greeting;

}

}

The following explains the message flow for the above example:

* WebSocket clients connect to the WebSocket endpoint at "/portfolio".
* Subscriptions to "/topic/greeting" pass through the "clientInboundChannel" and are forwarded to the broker.
* Greetings sent to "/app/greeting" pass through the "clientInboundChannel" and are forwarded to the GreetingController. The controller adds the current time, and the return value is passed through the "brokerChannel" as a message to "/topic/greeting" (destination is selected based on a convention but can be overridden via @SendTo).
* The broker in turn broadcasts messages to subscribers, and they pass through the "clientOutboundChannel".

The next section provides more details on annotated methods including the kinds of arguments and return values supported.

### 26.4.4 Annotation Message Handling

The @MessageMapping annotation is supported on methods of @Controller classes. It can be used for mapping methods to message destinations and can also be combined with the type-level @MessageMapping for expressing shared mappings across all annotated methods within a controller.

By default destination mappings are treated as Ant-style, slash-separated, path patterns, e.g. "/foo\*", "/foo/\*\*". etc. They can also contain template variables, e.g. "/foo/{id}" that can then be referenced via @DestinationVariable-annotated method arguments.

|  |
| --- |
| [Note] |
| Applications can also use dot-separated destinations (vs slash). See [Section 26.4.9, “Using Dot as Separator in @MessageMapping Destinations”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-destination-separator). |

The following method arguments are supported for @MessageMapping methods:

* Message method argument to get access to the complete message being processed.
* @Payload-annotated argument for access to the payload of a message, converted with a org.springframework.messaging.converter.MessageConverter. The presence of the annotation is not required since it is assumed by default. Payload method arguments annotated with validation annotations (like @Validated) will be subject to JSR-303 validation.
* @Header-annotated arguments for access to a specific header value along with type conversion using an org.springframework.core.convert.converter.Converter if necessary.
* @Headers-annotated method argument that must also be assignable to java.util.Map for access to all headers in the message.
* MessageHeaders method argument for getting access to a map of all headers.
* MessageHeaderAccessor, SimpMessageHeaderAccessor, or StompHeaderAccessor for access to headers via typed accessor methods.
* @DestinationVariable-annotated arguments for access to template variables extracted from the message destination. Values will be converted to the declared method argument type as necessary.
* java.security.Principal method arguments reflecting the user logged in at the time of the WebSocket HTTP handshake.

The return value from an @MessageMapping method is converted with a org.springframework.messaging.converter.MessageConverter and used as the body of a new message that is then sent, by default, to the "brokerChannel" with the same destination as the client message but using the prefix "/topic" by default. An @SendTo message level annotation can be used to specify any other destination instead. It can also be set a class-level to share a common destination.

An @SubscribeMapping annotation can also be used to map subscription requests to @Controller methods. It is supported on the method level, but can also be combined with a type level @MessageMapping annotation that expresses shared mappings across all message handling methods within the same controller.

By default the return value from an @SubscribeMapping method is sent as a message directly back to the connected client and does not pass through the broker. This is useful for implementing request-reply message interactions; for example, to fetch application data when the application UI is being initialized. Or alternatively an @SubscribeMapping method can be annotated with @SendTo in which case the resulting message is sent to the "brokerChannel" using the specified target destination.

|  |
| --- |
| [Note] |
| In some cases a controller may need to be decorated with an AOP proxy at runtime. One example is if you choose to have @Transactional annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is typically the default choice with controllers. However if a controller must implement an interface that is not a Spring Context callback (e.g. InitializingBean, \*Aware, etc), you may need to explicitly configure class-based proxying. For example with <tx:annotation-driven />, change to <tx:annotation-driven proxy-target-class="true" />. |

### 26.4.5 Sending Messages

What if you want to send messages to connected clients from any part of the application? Any application component can send messages to the "brokerChannel". The easiest way to do that is to have a SimpMessagingTemplate injected, and use it to send messages. Typically it should be easy to have it injected by type, for example:

*@Controller*

**public** **class** GreetingController {

**private** SimpMessagingTemplate template;

*@Autowired*

**public** GreetingController(SimpMessagingTemplate template) {

**this**.template = template;

}

*@RequestMapping(path="/greetings", method=POST)*

**public** **void** greet(String greeting) {

String text = "[" + getTimestamp() + "]:" + greeting;

**this**.template.convertAndSend("/topic/greetings", text);

}

}

But it can also be qualified by its name "brokerMessagingTemplate" if another bean of the same type exists.

### 26.4.6 Simple Broker

The built-in, simple message broker handles subscription requests from clients, stores them in memory, and broadcasts messages to connected clients with matching destinations. The broker supports path-like destinations, including subscriptions to Ant-style destination patterns.

|  |
| --- |
| [Note] |
| Applications can also use dot-separated destinations (vs slash). See [Section 26.4.9, “Using Dot as Separator in @MessageMapping Destinations”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-destination-separator). |

### 26.4.7 Full-Featured Broker

The simple broker is great for getting started but supports only a subset of STOMP commands (e.g. no acks, receipts, etc.), relies on a simple message sending loop, and is not suitable for clustering. As an alternative, applications can upgrade to using a full-featured message broker.

Check the STOMP documentation for your message broker of choice (e.g. [RabbitMQ](https://www.rabbitmq.com/stomp.html), [ActiveMQ](http://activemq.apache.org/stomp.html), etc.), install the broker, and run it with STOMP support enabled. Then enable the STOMP broker relay in the Spring configuration instead of the simple broker.

Below is example configuration that enables a full-featured broker:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **implements** WebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS();

}

*@Override*

**public** **void** configureMessageBroker(MessageBrokerRegistry registry) {

registry.enableStompBrokerRelay("/topic", "/queue");

registry.setApplicationDestinationPrefixes("/app");

}

}

XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app">

<websocket:stomp-endpoint path="/portfolio" />

<websocket:sockjs/>

</websocket:stomp-endpoint>

<websocket:stomp-broker-relay prefix="/topic,/queue" />

</websocket:message-broker>

</beans>

The "STOMP broker relay" in the above configuration is a Spring [MessageHandler](https://docs.spring.io/spring-framework/docs/4.3.3.RELEASE/javadoc-api/org/springframework/messaging/MessageHandler.html) that handles messages by forwarding them to an external message broker. To do so it establishes TCP connections to the broker, forwards all messages to it, and then forwards all messages received from the broker to clients through their WebSocket sessions. Essentially it acts as a "relay" that forwards messages in both directions.

|  |
| --- |
| [Note] |
| Please add a dependency on org.projectreactor:reactor-net for TCP connection management. |

Furthermore, application components (e.g. HTTP request handling methods, business services, etc.) can also send messages to the broker relay, as described in [Section 26.4.5, “Sending Messages”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-handle-send), in order to broadcast messages to subscribed WebSocket clients.

In effect, the broker relay enables robust and scalable message broadcasting.

### 26.4.8 Connections To Full-Featured Broker

A STOMP broker relay maintains a single "system" TCP connection to the broker. This connection is used for messages originating from the server-side application only, not for receiving messages. You can configure the STOMP credentials for this connection, i.e. the STOMP frame login and passcode headers. This is exposed in both the XML namespace and the Java config as the systemLogin/systemPasscode properties with default values guest/guest.

The STOMP broker relay also creates a separate TCP connection for every connected WebSocket client. You can configure the STOMP credentials to use for all TCP connections created on behalf of clients. This is exposed in both the XML namespace and the Java config as the clientLogin/clientPasscode properties with default values guest/guest.

|  |
| --- |
| [Note] |
| The STOMP broker relay always sets the login and passcode headers on every CONNECT frame that it forwards to the broker on behalf of clients. Therefore WebSocket clients need not set those headers; they will be ignored. As the following section explains, instead WebSocket clients should rely on HTTP authentication to protect the WebSocket endpoint and establish the client identity. |

The STOMP broker relay also sends and receives heartbeats to and from the message broker over the "system" TCP connection. You can configure the intervals for sending and receiving heartbeats (10 seconds each by default). If connectivity to the broker is lost, the broker relay will continue to try to reconnect, every 5 seconds, until it succeeds.

|  |
| --- |
| [Note] |
| A Spring bean can implement ApplicationListener<BrokerAvailabilityEvent> in order to receive notifications when the "system" connection to the broker is lost and re-established. For example a Stock Quote service broadcasting stock quotes can stop trying to send messages when there is no active "system" connection. |

The STOMP broker relay can also be configured with a virtualHost property. The value of this property will be set as the host header of every CONNECT frame and may be useful for example in a cloud environment where the actual host to which the TCP connection is established is different from the host providing the cloud-based STOMP service.

### 26.4.9 Using Dot as Separator in @MessageMapping Destinations

Although slash-separated path patterns are familiar to web developers, in messaging it is common to use a "." as the separator, for example in the names of topics, queues, exchanges, etc. Applications can also switch to using "." (dot) instead of "/" (slash) as the separator in @MessageMapping mappings by configuring a custom AntPathMatcher.

In Java config:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **extends** AbstractWebSocketMessageBrokerConfigurer {

*// ...*

*@Override*

**public** **void** configureMessageBroker(MessageBrokerRegistry registry) {

registry.enableStompBrokerRelay("/queue/", "/topic/");

registry.setApplicationDestinationPrefixes("/app");

registry.setPathMatcher(**new** AntPathMatcher("."));

}

}

In XML config:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app" path-matcher="pathMatcher">

<websocket:stomp-endpoint path="/stomp" />

<websocket:simple-broker prefix="/topic, /queue"/>

</websocket:message-broker>

<bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">

<constructor-arg index="0" value="." />

</bean>

</beans>

And below is a simple example to illustrate a controller with "." separator:

*@Controller*

*@MessageMapping("foo")*

**public** **class** FooController {

*@MessageMapping("bar.{baz}")*

**public** **void** handleBaz(*@DestinationVariable* String baz) {

}

}

If the application prefix is set to "/app" then the foo method is effectively mapped to "/app/foo.bar.{baz}".

### 26.4.10 Authentication

In a WebSocket-style application it is often useful to know who sent a message. Therefore some form of authentication is needed to establish the user identity and associate it with the current session.

Existing Web applications already use HTTP based authentication. For example Spring Security can secure the HTTP URLs of the application as usual. Since a WebSocket session begins with an HTTP handshake, that means URLs mapped to STOMP/WebSocket are already automatically protected and require authentication. Moreover the page that opens the WebSocket connection is itself likely protected and so by the time of the actual handshake, the user should have been authenticated.

When a WebSocket handshake is made and a new WebSocket session is created, Spring’s WebSocket support automatically propagates the java.security.Principal from the HTTP request to the WebSocket session. After that every message flowing through the application on that WebSocket session is enriched with the user information. It’s present in the message as a header. Controller methods can access the current user by adding a method argument of type javax.security.Principal.

Note that even though the STOMP CONNECT frame has "login" and "passcode" headers that can be used for authentication, Spring’s STOMP WebSocket support ignores them and currently expects users to have been authenticated already via HTTP.

In some cases it may be useful to assign an identity to a WebSocket session even when the user has not been formally authenticated. For example, a mobile app might assign some identity to anonymous users, perhaps based on geographical location. The do that currently, an application can sub-class DefaultHandshakeHandler and override the determineUser method. The custom handshake handler can then be plugged in (see examples in [Section 26.2.4, “Deployment Considerations”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-server-deployment)).

### 26.4.11 User Destinations

An application can send messages targeting a specific user, and Spring’s STOMP support recognizes destinations prefixed with "/user/" for this purpose. For example, a client might subscribe to the destination "/user/queue/position-updates". This destination will be handled by the UserDestinationMessageHandlerand transformed into a destination unique to the user session, e.g. "/queue/position-updates-user123". This provides the convenience of subscribing to a generically named destination while at the same time ensuring no collisions with other users subscribing to the same destination so that each user can receive unique stock position updates.

On the sending side messages can be sent to a destination such as "/user/{username}/queue/position-updates", which in turn will be translated by the UserDestinationMessageHandler into one or more destinations, one for each session associated with the user. This allows any component within the application to send messages targeting a specific user without necessarily knowing anything more than their name and the generic destination. This is also supported through an annotation as well as a messaging template.

For example, a message-handling method can send messages to the user associated with the message being handled through the @SendToUser annotation (also supported on the class-level to share a common destination):

*@Controller*

**public** **class** PortfolioController {

*@MessageMapping("/trade")*

*@SendToUser("/queue/position-updates")*

**public** TradeResult executeTrade(Trade trade, Principal principal) {

*// ...*

**return** tradeResult;

}

}

If the user has more than one session, by default all of the sessions subscribed to the given destination are targeted. However sometimes, it may be necessary to target only the session that sent the message being handled. This can be done by setting the broadcast attribute to false, for example:

*@Controller*

**public** **class** MyController {

*@MessageMapping("/action")*

**public** **void** handleAction() **throws** Exception{

*// raise MyBusinessException here*

}

*@MessageExceptionHandler*

*@SendToUser(destinations="/queue/errors", broadcast=false)*

**public** ApplicationError handleException(MyBusinessException exception) {

*// ...*

**return** appError;

}

}

|  |
| --- |
| [Note] |
| While user destinations generally imply an authenticated user, it isn’t required strictly. A WebSocket session that is not associated with an authenticated user can subscribe to a user destination. In such cases the @SendToUser annotation will behave exactly the same as with broadcast=false, i.e. targeting only the session that sent the message being handled. |

It is also possible to send a message to user destinations from any application component by injecting the SimpMessagingTemplate created by the Java config or XML namespace, for example (the bean name is "brokerMessagingTemplate" if required for qualification with @Qualifier):

*@Service*

**public** **class** TradeServiceImpl **implements** TradeService {

**private** **final** SimpMessagingTemplate messagingTemplate;

*@Autowired*

**public** TradeServiceImpl(SimpMessagingTemplate messagingTemplate) {

**this**.messagingTemplate = messagingTemplate;

}

*// ...*

**public** **void** afterTradeExecuted(Trade trade) {

**this**.messagingTemplate.convertAndSendToUser(

trade.getUserName(), "/queue/position-updates", trade.getResult());

}

}

|  |
| --- |
| [Note] |
| When using user destinations with an external message broker, check the broker documentation on how to manage inactive queues, so that when the user session is over, all unique user queues are removed. For example, RabbitMQ creates auto-delete queues when destinations like /exchange/amq.direct/position-updates are used. So in that case the client could subscribe to /user/exchange/amq.direct/position-updates. Similarly, ActiveMQ has [configuration options](http://activemq.apache.org/delete-inactive-destinations.html) for purging inactive destinations. |

In a multi-application server scenario a user destination may remain unresolved because the user is connected to a different server. In such cases you can configure a destination to broadcast unresolved messages to so that other servers have a chance to try. This can be done through the userDestinationBroadcast property of theMessageBrokerRegistry in Java config and the user-destination-broadcast attribute of the message-broker element in XML.

### 26.4.12 Listening To ApplicationContext Events and Intercepting Messages

Several ApplicationContext events (listed below) are published and can be received by implementing Spring’s ApplicationListener interface.

* BrokerAvailabilityEvent — indicates when the broker becomes available/unavailable. While the "simple" broker becomes available immediately on startup and remains so while the application is running, the STOMP "broker relay" may lose its connection to the full featured broker, for example if the broker is restarted. The broker relay has reconnect logic and will re-establish the "system" connection to the broker when it comes back, hence this event is published whenever the state changes from connected to disconnected and vice versa. Components using the SimpMessagingTemplate should subscribe to this event and avoid sending messages at times when the broker is not available. In any case they should be prepared to handle MessageDeliveryException when sending a message.
* SessionConnectEvent — published when a new STOMP CONNECT is received indicating the start of a new client session. The event contains the message representing the connect including the session id, user information (if any), and any custom headers the client may have sent. This is useful for tracking client sessions. Components subscribed to this event can wrap the contained message using SimpMessageHeaderAccessor or StompMessageHeaderAccessor.
* SessionConnectedEvent — published shortly after a SessionConnectEvent when the broker has sent a STOMP CONNECTED frame in response to the CONNECT. At this point the STOMP session can be considered fully established.
* SessionSubscribeEvent — published when a new STOMP SUBSCRIBE is received.
* SessionUnsubscribeEvent — published when a new STOMP UNSUBSCRIBE is received.
* SessionDisconnectEvent — published when a STOMP session ends. The DISCONNECT may have been sent from the client, or it may also be automatically generated when the WebSocket session is closed. In some cases this event may be published more than once per session. Components should be idempotent with regard to multiple disconnect events.

|  |
| --- |
| [Note] |
| When using a full-featured broker, the STOMP "broker relay" automatically reconnects the "system" connection in case the broker becomes temporarily unavailable. Client connections however are not automatically reconnected. Assuming heartbeats are enabled, the client will typically notice the broker is not responding within 10 seconds. Clients need to implement their own reconnect logic. |

Furthermore, an application can directly intercept every incoming and outgoing message by registering a ChannelInterceptor on the respective message channel. For example to intercept inbound messages:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **extends** AbstractWebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** configureClientInboundChannel(ChannelRegistration registration) {

registration.setInterceptors(**new** MyChannelInterceptor());

}

}

A custom ChannelInterceptor can extend the empty method base class ChannelInterceptorAdapter and use StompHeaderAccessor or SimpMessageHeaderAccessor to access information about the message.

**public** **class** MyChannelInterceptor **extends** ChannelInterceptorAdapter {

*@Override*

**public** Message<?> preSend(Message<?> message, MessageChannel channel) {

StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);

StompCommand command = accessor.getStompCommand();

*// ...*

**return** message;

}

}

### 26.4.13 STOMP Client

Spring provides a STOMP over WebSocket client and a STOMP over TCP client.

To begin create and configure WebSocketStompClient:

WebSocketClient webSocketClient = **new** StandardWebSocketClient();

WebSocketStompClient stompClient = **new** WebSocketStompClient(webSocketClient);

stompClient.setMessageConverter(**new** StringMessageConverter());

stompClient.setTaskScheduler(taskScheduler); *// for heartbeats*

In the above example StandardWebSocketClient could be replaced with SockJsClient since that is also an implementation of WebSocketClient. The SockJsClient can use WebSocket or HTTP-based transport as a fallback. For more details see [Section 26.3.7, “SockJS Client”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-fallback-sockjs-client).

Next establish a connection and provide a handler for the STOMP session:

String url = "ws://127.0.0.1:8080/endpoint";

StompSessionHandler sessionHandler = **new** MyStompSessionHandler();

stompClient.connect(url, sessionHandler);

When the session is ready for use the handler is notified:

**public** **class** MyStompSessionHandler **extends** StompSessionHandlerAdapter {

*@Override*

**public** **void** afterConnected(StompSession session, StompHeaders connectedHeaders) {

*// ...*

}

}

Once the session is established any payload can be sent and that will be serialized with the configured MessageConverter:

session.send("/topic/foo", "payload");

You can also subscribe to destinations. The subscribe methods require a handler for messages on the subscription and return a Subscription handle that can be used to unsubscribe. For each received message the handler can specify the target Object type the payload should be deserialized to:

session.subscribe("/topic/foo", **new** StompFrameHandler() {

*@Override*

**public** Type getPayloadType(StompHeaders headers) {

**return** String.**class**;

}

*@Override*

**public** **void** handleFrame(StompHeaders headers, Object payload) {

*// ...*

}

});

To enable STOMP heartbeat configure WebSocketStompClient with a TaskScheduler and optionally customize the heartbeat intervals, 10 seconds for write inactivity which causes a heartbeat to be sent and 10 seconds for read inactivity which closes the connection.

|  |
| --- |
| [Note] |
| When using WebSocketStompClient for performance tests to simulate thousands of clients from the same machine consider turning off heartbeats since each connection schedules its own heartbeat tasks and that’s not optimized for a a large number of clients running on the same machine. |

The STOMP protocol also supports receipts where the client must add a "receipt" header to which the server responds with a RECEIPT frame after the send or subscribe are processed. To support this the StompSession offers setAutoReceipt(boolean) that causes a "receipt" header to be added on every subsequent send or subscribe. Alternatively you can also manually add a "receipt" header to the StompHeaders. Both send and subscribe return an instance of Receiptable that can be used to register for receipt success and failure callbacks. For this feature the client must be configured with a TaskScheduler and the amount of time before a receipt expires (15 seconds by default).

Note that StompSessionHandler itself is a StompFrameHandler which allows it to handle ERROR frames in addition to the handleException callback for exceptions from the handling of messages, and handleTransportError for transport-level errors including ConnectionLostException.

### 26.4.14 WebSocket Scope

Each WebSocket session has a map of attributes. The map is attached as a header to inbound client messages and may be accessed from a controller method, for example:

*@Controller*

**public** **class** MyController {

*@MessageMapping("/action")*

**public** **void** handle(SimpMessageHeaderAccessor headerAccessor) {

Map<String, Object> attrs = headerAccessor.getSessionAttributes();

*// ...*

}

}

It is also possible to declare a Spring-managed bean in the websocket scope. WebSocket-scoped beans can be injected into controllers and any channel interceptors registered on the "clientInboundChannel". Those are typically singletons and live longer than any individual WebSocket session. Therefore you will need to use a scope proxy mode for WebSocket-scoped beans:

*@Component*

*@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS)*

**public** **class** MyBean {

*@PostConstruct*

**public** **void** init() {

*// Invoked after dependencies injected*

}

*// ...*

*@PreDestroy*

**public** **void** destroy() {

*// Invoked when the WebSocket session ends*

}

}

*@Controller*

**public** **class** MyController {

**private** **final** MyBean myBean;

*@Autowired*

**public** MyController(MyBean myBean) {

**this**.myBean = myBean;

}

*@MessageMapping("/action")*

**public** **void** handle() {

*// this.myBean from the current WebSocket session*

}

}

As with any custom scope, Spring initializes a new MyBean instance the first time it is accessed from the controller and stores the instance in the WebSocket session attributes. The same instance is returned subsequently until the session ends. WebSocket-scoped beans will have all Spring lifecycle methods invoked as shown in the examples above.

### 26.4.15 Configuration and Performance

There is no silver bullet when it comes to performance. Many factors may affect it including the size of messages, the volume, whether application methods perform work that requires blocking, as well as external factors such as network speed and others. The goal of this section is to provide an overview of the available configuration options along with some thoughts on how to reason about scaling.

In a messaging application messages are passed through channels for asynchronous executions backed by thread pools. Configuring such an application requires good knowledge of the channels and the flow of messages. Therefore it is recommended to review [Section 26.4.3, “Flow of Messages”](https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#websocket-stomp-message-flow).

The obvious place to start is to configure the thread pools backing the "clientInboundChannel" and the "clientOutboundChannel". By default both are configured at twice the number of available processors.

If the handling of messages in annotated methods is mainly CPU bound then the number of threads for the "clientInboundChannel" should remain close to the number of processors. If the work they do is more IO bound and requires blocking or waiting on a database or other external system then the thread pool size will need to be increased.

|  |
| --- |
| [Note] |
| ThreadPoolExecutor has 3 important properties. Those are the core and the max thread pool size as well as the capacity for the queue to store tasks for which there are no available threads.  A common point of confusion is that configuring the core pool size (e.g. 10) and max pool size (e.g. 20) results in a thread pool with 10 to 20 threads. In fact if the capacity is left at its default value of Integer.MAX\_VALUE then the thread pool will never increase beyond the core pool size since all additional tasks will be queued.  Please review the Javadoc of ThreadPoolExecutor to learn how these properties work and understand the various queuing strategies. |

On the "clientOutboundChannel" side it is all about sending messages to WebSocket clients. If clients are on a fast network then the number of threads should remain close to the number of available processors. If they are slow or on low bandwidth they will take longer to consume messages and put a burden on the thread pool. Therefore increasing the thread pool size will be necessary.

While the workload for the "clientInboundChannel" is possible to predict — after all it is based on what the application does — how to configure the "clientOutboundChannel" is harder as it is based on factors beyond the control of the application. For this reason there are two additional properties related to the sending of messages. Those are the "sendTimeLimit" and the "sendBufferSizeLimit". Those are used to configure how long a send is allowed to take and how much data can be buffered when sending messages to a client.

The general idea is that at any given time only a single thread may be used to send to a client. All additional messages meanwhile get buffered and you can use these properties to decide how long sending a message is allowed to take and how much data can be buffered in the mean time. Please review the Javadoc and documentation of the XML schema for this configuration for important additional details.

Here is example configuration:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **implements** WebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setSendTimeLimit(15 \* 1000).setSendBufferSizeLimit(512 \* 1024);

}

*// ...*

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker>

<websocket:transport send-timeout="15000" send-buffer-size="524288" />

*<!-- ... -->*

</websocket:message-broker>

</beans>

The WebSocket transport configuration shown above can also be used to configure the maximum allowed size for incoming STOMP messages. Although in theory a WebSocket message can be almost unlimited in size, in practice WebSocket servers impose limits — for example, 8K on Tomcat and 64K on Jetty. For this reason STOMP clients such as stomp.js split larger STOMP messages at 16K boundaries and send them as multiple WebSocket messages thus requiring the server to buffer and re-assemble.

Spring’s STOMP over WebSocket support does this so applications can configure the maximum size for STOMP messages irrespective of WebSocket server specific message sizes. Do keep in mind that the WebSocket message size will be automatically adjusted if necessary to ensure they can carry 16K WebSocket messages at a minimum.

Here is example configuration:

*@Configuration*

*@EnableWebSocketMessageBroker*

**public** **class** WebSocketConfig **implements** WebSocketMessageBrokerConfigurer {

*@Override*

**public** **void** configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setMessageSizeLimit(128 \* 1024);

}

*// ...*

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker>

<websocket:transport message-size="131072" />

*<!-- ... -->*

</websocket:message-broker>

</beans>

An important point about scaling is using multiple application instances. Currently it is not possible to do that with the simple broker. However when using a full-featured broker such as RabbitMQ, each application instance connects to the broker and messages broadcast from one application instance can be broadcast through the broker to WebSocket clients connected through any other application instances.

### 26.4.16 Runtime Monitoring

When using @EnableWebSocketMessageBroker or <websocket:message-broker> key infrastructure components automatically gather stats and counters that provide important insight into the internal state of the application. The configuration also declares a bean of type WebSocketMessageBrokerStats that gathers all available information in one place and by default logs it at INFO level once every 30 minutes. This bean can be exported to JMX through Spring’s MBeanExporter for viewing at runtime, for example through JDK’s jconsole. Below is a summary of the available information.

Client WebSocket Sessions

Current

indicates how many client sessions there are currently with the count further broken down by WebSocket vs HTTP streaming and polling SockJS sessions.

Total

indicates how many total sessions have been established.

Abnormally Closed

Connect Failures

these are sessions that got established but were closed after not having received any messages within 60 seconds. This is usually an indication of proxy or network issues.

Send Limit Exceeded

sessions closed after exceeding the configured send timeout or the send buffer limits which can occur with slow clients (see previous section).

Transport Errors

sessions closed after a transport error such as failure to read or write to a WebSocket connection or HTTP request/response.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames processed indicating how many clients connected on the STOMP level. Note that the DISCONNECT count may be lower when sessions get closed abnormally or when clients close without sending a DISCONNECT frame.

STOMP Broker Relay

TCP Connections

indicates how many TCP connections on behalf of client WebSocket sessions are established to the broker. This should be equal to the number of client WebSocket sessions + 1 additional shared "system" connection for sending messages from within the application.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames forwarded to or received from the broker on behalf of clients. Note that a DISCONNECT frame is sent to the broker regardless of how the client WebSocket session was closed. Therefore a lower DISCONNECT frame count is an indication that the broker is pro-actively closing connections, may be because of a heartbeat that didn’t arrive in time, an invalid input frame, or other.

Client Inbound Channel

stats from thread pool backing the "clientInboundChannel" providing insight into the health of incoming message processing. Tasks queueing up here is an indication the application may be too slow to handle messages. If there I/O bound tasks (e.g. slow database query, HTTP request to 3rd party REST API, etc) consider increasing the thread pool size.

Client Outbound Channel

stats from the thread pool backing the "clientOutboundChannel" providing insight into the health of broadcasting messages to clients. Tasks queueing up here is an indication clients are too slow to consume messages. One way to address this is to increase the thread pool size to accommodate the number of concurrent slow clients expected. Another option is to reduce the send timeout and send buffer size limits (see the previous section).

SockJS Task Scheduler

stats from thread pool of the SockJS task scheduler which is used to send heartbeats. Note that when heartbeats are negotiated on the STOMP level the SockJS heartbeats are disabled.

### 26.4.17 Testing Annotated Controller Methods

There are two main approaches to testing applications using Spring’s STOMP over WebSocket support. The first is to write server-side tests verifying the functionality of controllers and their annotated message handling methods. The second is to write full end-to-end tests that involve running a client and a server.

The two approaches are not mutually exclusive. On the contrary each has a place in an overall test strategy. Server-side tests are more focused and easier to write and maintain. End-to-end integration tests on the other hand are more complete and test much more, but they’re also more involved to write and maintain.

The simplest form of server-side tests is to write controller unit tests. However this is not useful enough since much of what a controller does depends on its annotations. Pure unit tests simply can’t test that.

Ideally controllers under test should be invoked as they are at runtime, much like the approach to testing controllers handling HTTP requests using the Spring MVC Test framework. i.e. without running a Servlet container but relying on the Spring Framework to invoke the annotated controllers. Just like with Spring MVC Test here there are two two possible alternatives, either using a "context-based" or "standalone" setup:

1. Load the actual Spring configuration with the help of the Spring TestContext framework, inject "clientInboundChannel" as a test field, and use it to send messages to be handled by controller methods.
2. Manually set up the minimum Spring framework infrastructure required to invoke controllers (namely the SimpAnnotationMethodMessageHandler) and pass messages for controllers directly to it.

Both of these setup scenarios are demonstrated in the [tests for the stock portfolio](https://github.com/rstoyanchev/spring-websocket-portfolio/tree/master/src/test/java/org/springframework/samples/portfolio/web) sample application.

第二种方法是创建端到端的集成测试。为此，您需要以嵌入模式运行WebSocket服务器，并将其作为发送包含STOMP框架的WebSocket消息的WebSocket客户端连接到它。[股票投资组合](https://github.com/rstoyanchev/spring-websocket-portfolio/tree/master/src/test/java/org/springframework/samples/portfolio/web) 样本应用程序的[测试](https://github.com/rstoyanchev/spring-websocket-portfolio/tree/master/src/test/java/org/springframework/samples/portfolio/web)也演示了使用Tomcat作为嵌入式WebSocket服务器和用于测试目的的简单STOMP客户端的方法。