



자바 개발자를 위한

Vert.X

애플리케이션 개발

이연복 지음

자바 개발자를 위한
Vert.x
애플리케이션 개발

자바 개발자를 위한 Vert.x 애플리케이션 개발

초판발행 2015년 2월 5일

지은이 이연복 / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-6848-728-6 15000 / 정가 14,000원

총괄 배용석 / 책임편집 김창수 / 기획·편집 정지연

디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실

영업 김형진, 김진불, 조유미 / 마케팅 박상용

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2015 이연복 & HANBIT Media, Inc.

이 책의 저작권은 이연복과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_이연복

“들은 것은 잊어버리지만 본 것은 기억한다. 하지만 행동해야 진정으로 이해할 수 있다.”라는 말에 깊이 공감하고, 선택의 문제가 산재한 삶 속에 적용하기 위해 고민하는 개발자다. 그 결과, “하지 않고 후회하는 것보다 하고 나서 후회하는 것이 낫다.”라는 신념을 지니게 되었고 이 신념에 따라 2011년 SW 마에스트로 과정을 마친 직후 ‘helloworld’라는 벤처기업을 시작했다. 현재까지 ‘helloworld’에서 Java를 사용하여 다양한 서비스 개발에 참여하고 있다. 주로 서버 사이드 개발에 관심이 많다.

저자 서문

바야흐로 모바일 시대를 넘어 IoT(Internet of Things) 시대가 성큼 다가왔습니다. 많은 기기가 다양한 센서를 통해 외부 세계를 인지하고 정보를 수집하며, 수집된 정보는 기기가 연결된 네트워크를 따라 다른 기기 또는 인터넷을 통해 그것을 처리할 서버로 보내집니다. IoT 기기에 네트워크 기능이 없다면 이들 기기의 활용도는 매우 제한적일 수밖에 없습니다. 센서로 수집한 날씨, 속도, 사용자 생체 정보 등은 다른 기기 또는 인터넷과의 연동으로 좀 더 가치 있고 쓸모 있는 정보로 취급될 수 있기 때문입니다. 따라서 IoT 시대에서 빼놓을 수 없는 중요한 특징 중 하나가 바로 네트워크 기술입니다.

이에 따라 수많은 IoT 기기와 연결되고, 이들 기기가 보내오는 정보를 효과적으로 처리하기 위해서는 새로운 형태의 애플리케이션 개발 방법이 필요합니다. 전통적으로 이런 요구 사항을 반영하는 애플리케이션은 다중 스레드 방식으로 개발됐습니다. 그러나 다중 스레드 방식은 연결된 기기가 많아질수록 시스템 리소스가 과도하게 소모된다는 단점이 있는데, Vert.x는 이런 단점을 해결하고 효과적으로 다수의 클라이언트를 동시에 처리할 수 있는 강력한 방법을 제공합니다. Vert.x에서는 이 방법을 Multireactor라고 정의합니다.

최근 주목받는 또 다른 기술로 WebSocket을 꼽을 수 있습니다. HTML5에 포함된 WebSocket은 웹 브라우저와 웹 서버 간의 양방향 통신을 지원하는 차세대 표준으로, 웹 애플리케이션의 반응성을 극적으로 향상하여 기존의 Ajax에 기반을 둔 대화형 웹 애플리케이션을 뛰어넘는 훨씬 효율적이고 역동적인 웹 애플리케이션 개발을 가능하게 합니다. Vert.x에서는 WebSocket을 훌륭하게 지원하고 있으며, 이에 더해 WebSocket 에뮬레이터 중 하나인 SockJS로 좀 더 편리하게 실시간 웹 애플리케이션을 개발할 수 있도록 API를 제공하고 있습니다.

특히 SockJS Event Bus Bridge라는 확장 기술은 기존 웹 애플리케이션의 한계를 벗어나 좀 더 자유롭게 애플리케이션을 설계하는 데 도움을 주는 아주 강력한 도구입니다.

마지막으로, Vert.x는 요즘 주목받는 폴리글랏polyglot 패러다임을 지원합니다. 폴리글랏이란 여러 개의 언어를 동시에 사용할 수 있는 특징을 말하는데, 이 책에서 Vert.x 예제를 작성하기 위해 사용된 Java 외에도 JavaScript, Ruby, Python, Groovy, Clojure, Scala 등 다양한 언어를 사용할 수 있습니다. 물론 동일한 프로젝트에서 2개 이상의 언어를 혼용해 사용하는 것도 가능합니다.

Vert.x는 앞에서 언급한 내용 외에도 주목할 만한 몇 가지 특징과 함께 빠르게 성장하고 있는 오픈소스 플랫폼입니다. Vert.x가 시작된 지 이제 겨우 3년이 지났지만, 최근 JAX Innovation awards 2014에서 Most Innovative Java Technology 부문을 수상하며⁰¹ 더욱 그 가치를 높여가고 있는 만큼 Vert.x는 이제 전 세계 수많은 개발자가 주목하는 이벤트 기반 비동기 프로그래밍 모델의 대표 플랫폼이라 할 수 있습니다.

이 책은 이처럼 빠르게 성장하고 있는 Vert.x의 중요한 개념과 철학, 기본적인 응용 방법을 설명하고, 이를 통해 다양한 요구 조건이 있는 실무 프로젝트에서 Vert.x가 활용될 수 있기를 바라며 집필하였습니다. Java로 작성된 이 책의 예제들을 JavaScript나 Scala로 다시 작성해 보는 것도 좋은 학습 방법이 될 것입니다. 특히, 최근 발표된 Java 8과 함께 도입된 람다식을 적용해보는 것도 아주 흥미로운 주제가 될 것 같습니다.

01 <https://jax.de/awards2014/>

Vert.x를 사용하며 막히는 부분은 [Vert.x Google Groups⁰²](#)를 참고하거나 가능하다면 직접 Vert.x 구현 소스 코드를 살펴볼 것을 권장합니다. 개인적으로 필자에게 연락을 주신다면 필자의 능력 내에서 최대한 답변하겠습니다.

끝으로, 이 책을 집필할 기회를 제공해준 김창수 팀장님, 정지연 과장님을 비롯한 한빛미디어 관계자 여러분께 감사드리며, 항상 곁에서 응원해주는 사랑하는 가족과 여자친구 인희에게도 감사의 말을 전합니다.

02 <https://groups.google.com/forum/#!forum/vertx>

대상 독자 및 참고사항

초급

초중급

중급

중고급

고급

이 책은 Vert.x를 이용하여 실시간 채팅 서비스를 개발하는 과정을 담고 있습니다. Vert.x를 어느 정도 알고 있다면 좀 더 쉽게 이 책의 내용을 이해할 수 있지만, 꼭 Vert.x에 관한 사전 지식이 있을 필요는 없습니다. 그러나 이 책의 예제 소스는 대부분 Java로 작성되었기 때문에 Java의 기초적인 문법과 개념을 알고 있어야 합니다.

이 책의 예제 코드를 실행하려면 다음 환경이 갖춰져 있어야 합니다.

- JDK 7 이상 설치된 개발 환경
- Vert.x 2.1 이상 설치된 개발 환경

이 책의 예제 소스 코드는 다음 웹 사이트에서 내려받을 수 있습니다.

- <http://www.hanbit.co.kr/exam/2728/>

한빛 리얼타임

한빛 리얼타임은 IT 개발자를 위한 전자책입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook First - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 접할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 편리한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 내려받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

0 1	들어가며	1
	1.1 Vert.x의 소개.....	.1
	1.2 왜 Vert.x인가.....	.2
	1.4 간단한 Vert.x 애플리케이션18
	1.5 요약.....	.22
0 2	TCP 채팅 서버/클라이언트	23
	2.1 TCP 채팅 서버.....	.23
	2.2 TCP 채팅 클라이언트.....	.31
	2.3 첫 번째 결함의 분석과 해결방법.....	.43
	2.4 두 번째 결함의 분석과 해결방법.....	.49
	2.5 요약.....	.61
0 3	SockJS 채팅 서버/클라이언트	62
	3.1 HTTP 서버.....	.63
	3.2 SockJS.....	.67
	3.3 SockJS Event Bus Bridge.....	.78
	3.4 SockJS EBB 보안 설정.....	.92
	3.5 요약.....	.109

0 4	TCP/SockJS EBB 채팅 서비스 통합	111
	4.1 IntegratedTCPChatServerVerticle.....	113
	4.2 IntegratedSockJSEbbChatServerVerticle.....	117
	4.3 요약.....	125
0 5	Vert.x 모듈 시스템	127
	5.1 모듈 설치와 실행.....	127
	5.2 모듈 디스크립터.....	129
	5.3 Vert.x 모듈 만들기.....	131
	5.4 몇 가지 유용한 Vert.x 모듈.....	135
0 6	Advanced 채팅 서비스	156
	6.1 TCP 채팅 서버 로그인 기능.....	156
	6.2 채팅 메시지 구조화.....	181
	6.3 대회방 분리.....	189
	6.4 귓속말 처리.....	206
	6.5 채팅 메시지 저장.....	229
	맺음말	240

1 | 들어가며

Vert.x의 간단한 역사와 핵심원리, 특징을 살펴본 후 이클립스에서 메이븐 기반으로 Vert.x 애플리케이션을 개발하는 방법을 알아본다. 이 장을 마칠 때는 언제나 그렇듯 ‘hello, world’를 출력하는 아주 간단한 Vert.x 애플리케이션을 볼 수 있다.

1.1 Vert.x의 소개

Vert.x는 2011년 팀 폭스 Tim Fox에 의해 시작된 후로 VMware의 지원 아래 발전하다가 팀 폭스의 거취 문제로 우여곡절을 겪은 끝에 현재는 Eclipse Foundation에서 관리하고 있다.⁰¹ 최근 JAX Innovation awards 2014에서 Most Innovative Java Technology 부문을 수상하며⁰² 더욱 그 가치를 높여가고 있는 만큼 Vert.x는 이제 전 세계 수많은 개발자가 주목하는 이벤트 기반 비동기 프로그래밍 EAP⁰³ 모델의 대표 플랫폼이라 할 수 있다.

Vert.x와 유사한 개발 모델을 제공하는 또 다른 유명한 플랫폼으로 Node.js를 언급하지 않을 수 없다. Node.js는 라이언 달 Ryan Dahl에 의해 2009년부터 시작된 프로젝트로, Vert.x보다 앞선 시기에 이미 많은 이슈와 관심을 불러일으켰고, 그 결과 오늘날 Vert.x보다 훨씬 풍부한 에코 시스템을 보유하고 있다.

Vert.x는 Node.js에서 영감을 얻어 시작된 프로젝트인 만큼 Node.js를 접해본 개발자라면 아주 쉽고 빠르게 Vert.x에 익숙해질 수 있을 정도로 근본 원리는

01 <http://www.infoq.com/news/2013/01/vertx-eclipse>

02 <http://jax.de/awards2014>

03 편집자 주_ Event-based Asynchronous Pattern과 혼동될 수 있으나 이 책에서는 Event-based Asynchronous Programming의 약자로 사용한다.

Node.js의 그것과 크게 차이가 없지만, 한편으로는 Node.js의 단점을 극복하고 더욱 발전된 구조와 성능을 보여준다.⁰⁴ 또한, Vert.x가 JVM Java Virtual Machine이라는 오랜 시간 동안 충분히 검증된 훌륭한 공학적 산물 위에서 동작하는 만큼 JVM의 안정성과 성능 등의 이점을 그대로 안고 갈 수 있다. 그리고 Node.js와는 비교할 수 없을 정도로 방대한 Java 레퍼런스 및 서드 파티 라이브러리 Third Party Library를 그대로 이용할 수 있다는 점은 Java 개발자뿐만 아니라 EAP 플랫폼을 주목하는 모든 개발자에게 아주 매력적인 장점이다.

그런데 최근에 Vert.x나 Node.js와 같은 EAP 플랫폼이 주목받는 이유는 무엇일까? 어떤 대단한 알고리즘이나 신기술이 있어 개발자로 하여금 풀기 어려운 문제를 극복할 수 있게 하는가? EAP 플랫폼에서 제공하는 동시 처리 Concurrency 모델의 장단점과 주요 특징을 알아보며 이 질문에 대한 해답을 생각해 보자.

1.2 왜 Vert.x인가

결론부터 말하자면 여기에 그 어떤 새로운 알고리즘이나 신기술은 없다. EAP 플랫폼에서 제공하는 동시 처리 모델은 10년도 더 된 오래된 기술이다.⁰⁵ 네트워크 프로그래밍 경험이 있는 독자라면 잘 알고 있는 `select()`, `poll()`, `WaitForMultipleObjects()`와 같은 I/O 디멀티플렉서 Demultiplexer에 기초한 Reactor 모델이 바로 EAP 플랫폼의 핵심 원리라 할 수 있다.

04 <http://vertxproject.wordpress.com/2012/05/09/vert-x-vs-node-js-simple-http-benchmarks>

05 오래된 기술을 사용하고 있다고 해서 아무런 의미가 없는 것은 아니다. 본래 Reactor 모델은 프로그래밍 복잡도가 높고 디버깅하기가 어렵다. Vert.x에서는 이러한 Reactor 모델의 복잡도를 낮추고 일반 개발자들이 좀 더 쉽게 접근하고 사용할 수 있도록 간결하고 일관성 있는 API를 제공한다.

NOTE

리눅스 커널 2.6 이상의 시스템에서 Java 6 이후 버전을 사용한다면 기존의 `select()`, `poll()`보다 더 나은 I/O 처리 성능을 보여주는 `epoll()`이라는 고성능 디멀티플렉서를 기본으로 사용하게 된다.⁰⁶ Windows 시스템에서는 Reactor 모델의 효율적인 구현이 어렵다. 그러나 Windows 시스템에서 제공하는 IOCP는 `epoll()`을 기반으로 구현한 Reactor 모델에 절대 뒤떨어지지 않는 성능을 보여준다. IOCP는 Reactor 모델과 마찬가지로 최대한 적은 수의 스레드로 동시 처리를 가능하게 하는 또 다른 방식으로 Proactor 모델이라 한다.

1.2.1 Vert.x의 동시 처리 모델

Reactor 동시 처리 모델

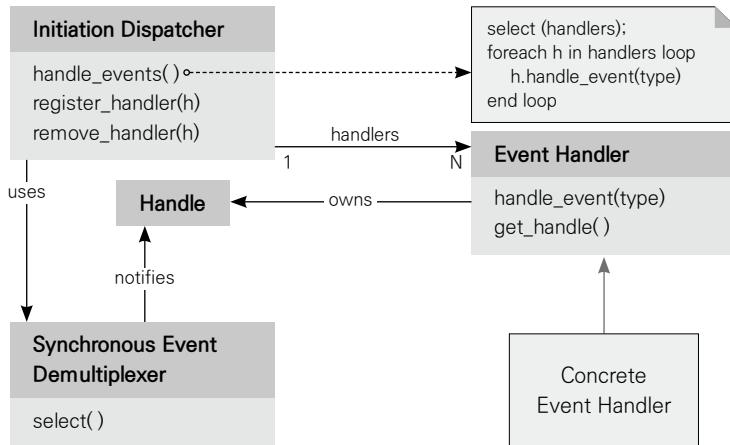
많은 작업을 동시에 수행해야 하는 애플리케이션은 다중 스레드(운영체제나 하드웨어에 따라 스레드가 아닌 프로세스를 사용할 수도 있음)를 사용해 작업을 처리하도록 디자인하는 것이 일반적인 방법이다. 이 방법은 요청된 작업을 처리하는 데 필요한 시간이 일정하지 않은, 특히 장시간 처리가 필요할 때 적합한 방법이다.

그러나 동시에 실행되는 2개 이상의 스레드 사이의 상호 작용과 공유 자원 동시 접근 문제 등을 다루려면 세마포어Semaphore나 뮤텍스Mutex 같은 동기화 장치를 반드시 사용해야 하는데, 이는 곧 데드락Deadlock 같은 잡재적인 문제를 내재하고 있음을 의미한다. 또한, 스레드의 생성과 관리는 증가하는 스레드에 비례하여 스레드 스택 메모리 유지와 문맥 전환Context Switching에 필요한 비용이 증가하고, 이는 곧 애플리케이션의 응답성이 나빠짐을 의미한다.

그렇다면 다중 스레드를 사용하지 않고 단일 스레드 상에서 많은 작업을 동시에 수행할 수 없을까? 이런 마법 같은 일을 수행해내는 것이 바로 Reactor 동시 처리 모델이다. Reactor의 구성 요소와 동작 방식을 [그림 1-1]에서 알아보자.

06 <http://docs.oracle.com/javase/7/docs/technotes/guides/io/enhancements.html>

[그림 1-1] Reactor OMT 다이어그램



출처: <http://blog.csdn.net/chexlong/article/details/22416775>

굵은 글씨 부분이 Reactor의 주요 구성 요소로, 이 그림에서 Synchronous Event Demultiplexer는 운영체제에서 제공하는 디멀티플렉서다(select() 외에도 poll(), WaitForMultipleObjects(), epoll() 등을 사용할 수 있다).

Initiation Dispatcher에는 이벤트 핸들러의 구체 클래스(Concrete Event Handler)를 등록, 해제, 호출하기 위한 인터페이스가 있어 블록된 상태를 유지하다가 디멀티플렉서를 통해 처리해야 할 한 개 이상의 이벤트를 감지하면 각 이벤트에 대응되는 이벤트 핸들러를 차례로 호출한다. Initiation Dispatcher는 단일 스레드에서 동작하므로 이벤트 핸들러의 동시 실행을 걱정할 필요는 없다.

이벤트 핸들러는 Initiation Dispatcher에서 필요한 추상 인터페이스를 정의하고 있어서 특정 애플리케이션에 비의존적인 메커니즘을 제공할 수 있다. 이와 같은 Initiation Dispatcher와 이벤트 핸들러의 느슨한 관계는 개발자가 집중해야 할 비즈니스 로직 부분을 이벤트 핸들링 부분과 분리해 주어 이벤트 주도 Event Driven 상황이 필요한 어떤 곳이라도 Reactor 동시 처리 모델을 쉽게 적용할 수 있다.

앞서 언급한 것처럼 Initiation Dispatcher는 단일 스레드에서 동작하므로 스레드 동기화 문제에서 벗어날 수 있지만, 본질적으로는 정교한 시분할 기법으로 각각의 작업을 순차적으로 처리하는 것에 지나지 않는다(순차적 처리가 매우 빨라서 사용자에게는 마치 동시에 작업이 처리되는 것처럼 보인다). 블록될 가능성이 있는 API를 호출하거나 CPU 연산 집약적인 처리로 이벤트 처리가 지연된다면 이는 곧 애플리케이션 전체에 심각한 병목 현상을 가져올 수 있음을 의미한다. 또한, 여러 개의 CPU를 가지고 이용할 수 있는 병렬 처리에 대한 이점을 얻을 수 없다는 단점이 있다.

실제로 이러한 단점은 Reactor 동시 처리 모델에 기반을 둔 Vert.x나 Node.js에서 그대로 나타나지만, Vert.x는 처음부터 이러한 단점을 보완하기 위한 구조를 갖추고 있어서 Node.js보다는 유리한 위치에 있다고 할 수 있다(Vert.x의 이런 독특한 구조를 Multireactor라 한다. Multireactor에 대한 내용은 뒤에서 자세히 설명하겠다). 그러나 Vert.x나 Node.js 관계없이 이벤트 기반 비동기 애플리케이션을 개발할 때 반드시 지켜야 할 규칙은 Initiation Dispatcher가 실행되는 스레드(Vert.x나 Node.js에서는 이벤트 루프 스레드 Event Loop Thread라고 함)를 블록시키거나 시간이 오래 걸리는 작업을 실행하느라 이벤트 처리가 지연되지 않게끔 해야 한다는 것이다. 이런 가지 규칙만 잘 지킨다면 스레드 동기화와 같은 어려운 문제에서 벗어나 단일 스레드만으로도 충분한 성능을 발휘하는 애플리케이션을 개발할 수 있다.⁰⁷

이와 같은 Reactor 동시 처리 모델에 기반한 애플리케이션 개발은 요즘처럼 웹 관련 기술이 빠르게 발전하고 모바일과 IoT 기기가 폭넓게 사용되고 있는 시점에 큰 의미를 지닌다. 다음 두 예를 살펴보자.

HTML5에 포함된 WebSocket은 웹 브라우저와 웹 서버 간의 양방향 통신을 지원하기 위

07 Vert.x가 실제로 단일 스레드로 동작하는 것은 아니다. 블록 API나 시간이 오래 걸리는 작업 처리하기 위해 별도의 스레드 풀(Thread Pool)을 제공한다.

한 차세대 표준으로, 웹 애플리케이션의 반응성을 극적으로 향상시켜 기존의 Ajax에 기반을 둔 대화형 웹 애플리케이션을 뛰어넘는 훨씬 효율적이고 역동적인 웹 애플리케이션 개발을 가능하게 한다.⁰⁸ Gmail, Google Docs, Facebook, 웹 채팅 서비스 등이 바로 그것으로, 이러한 종류의 웹 애플리케이션은 사용자에게 필요한 정보를 효과적이고 빠르게 전달해 준다는 공통점이 있다.

모바일 기기의 칠러 앱이 WhatsApp, KakaoTalk, Line 등과 같은 메신저 서비스라는 것은 의심할 여지가 없다. 모바일 기기의 메신저 서비스는 대부분 메시지를 전달하기 위해 푸시 알림PUSH Notification 방식을 사용한다. 푸시 알림 방식의 메시지 전달은 서버와 TCP 연결을 생성하고, 일정 주기로 Ping을 주고받으며 연결을 유지하다가 전달할 메시지가 발생하면 해당 TCP 연결을 통해 메시지를 전송받는 방식이다. 이 방식은 주기적으로 메시지를 확인하는 폴링Polling 방법보다 배터리와 네트워크 트래픽 소모량이 적다는 장점이 있다.

두 예에서 확인할 수 있는 사실은 HTML5의 WebSocket과 푸시 알림 방식 모두 높은 반응성과 모바일 기기의 효율성을 위해 TCP와 같은 연결 지향 기술을 사용한다는 점이다. 다중 스레드 동시 처리 모델은 연결되는 TCP 채널이 많아질수록 스레드 수도 많아지고 스레드 스택 메모리 유지와 문맥 전환에 필요한 비용도 함께 커져서 시스템 유지가 어려워지는 결과를 가져온다. 그러나 Reactor 동시 처리 모델은 연결되는 TCP 채널의 개수와 관계없이 항상 일정한 스레드 개수가 유지되어 불필요한 오버헤드가 없다.

또한, 일반적인 Reactor 동시 처리 모델은 단일 스레드에서 동작하여 병렬 처리의 이점을 얻을 수 없다는 단점이 있지만, Vert.x에서는 이러한 단점을 극복했다. Vert.x에서는 이것을 Multireactor이라 한다. Multireactor에 대해 좀 더 자세히 알아보자.

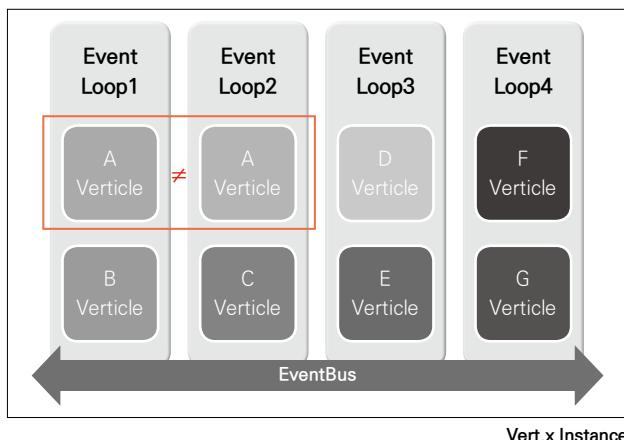
08 2005년 Ajax의 등장으로 전통적인 페이지 기반 웹 애플리케이션 개발의 한계를 벗어나 비동기 통신을 이용한 대화형 웹 애플리케이션 개발이 대중화되었다.

Multireactor

Multireactor는 Node.js와는 차별화되는 Vert.x만의 특징으로, 여러 개의 CPU를 이용할 수 있는 병렬 처리의 이점을 활용할 수 있게 한다. 이것은 시스템의 CPU 개수만큼 이벤트 루프 스레드를 생성해서 이벤트를 처리하는데, 각각의 이벤트 루프 스레드는 서로 다른 클래스로 더 ClassLoader로 만들어진 완전히 독립적인 코드를 실행하므로 스레드 동기화 문제는 걱정하지 않아도 된다. 또한, 이렇게 특정 이벤트 루프 스레드에서 한 번 실행된 코드는 절대 다른 이벤트 루프 스레드에서 실행되지 않음을 보장한다.

[그림 1-2]는 4개의 CPU를 사용할 수 있는 시스템에서 Vert.x를 실행했을 때의 모습이다.

[그림 1-2] Vert.x의 Multireactor



4개의 이벤트 루프 스레드가 생성되고 각 이벤트 루프 스레드는 서로 독립적인 Verticle⁰⁹을 실행한다. 각 Verticle은 서로 다른 클래스로 더로 만들어지므로

09 Vert.x의 이벤트 루프 스레드에서 실행되는 코드 단위를 Verticle이라고 한다.

A라는 Verticle이 이벤트 루프 스레드 1과 이벤트 루프 스레드 2에서 동시에 실행되고 있어도 서로 완전히 독립된 상태를 유지한다. 즉, Verticle 내에 static으로 선언한 변수를 포함한 그 어떤 값도 공유되지 않는다. 그리고 D~G Verticle은 이벤트 루프 스레드 3과 4에 할당되어 있는데, 이 Verticle은 최초 할당된 이벤트 루프 스레드에서만 실행됨을 보장하며 그 외 이벤트 루프 스레드에서는 절대 실행되지 않는다.

1.2.2 Vert.x의 주요 특징

일반적으로 Reactor 동시 처리 모델에 기반을 둔 Vert.x, Node.js, Meteror, EventMachine 같은 구현체들은¹⁰ 모두 비슷한 특징을 공유하므로 서로를 대체할 수 있다. 그러나 Vert.x는 Multireactor라는 고유의 독특한 구조 덕에 다른 구현체와 비교해 좀 더 나은 성능을 보여줄 수 있다.¹¹ 그럼 이번에는 다른 Reactor 동시 처리 모델 구현체와 차별화되는 Vert.x의 주요 특징을 알아보자.

단순함

Vert.x에서 제공하는 Core API는 단순하고 일관적이며 사용하기 쉽다. 또한, 앞서 설명한 것처럼 스레드 간 동기화 문제를 생각하지 않아도 되므로 개발자가 좀 더 생산적인 일에 집중할 수 있다.

다양한 언어 지원

Vert.x는 Java뿐만 아니라 다양한 언어를 지원한다. 현재 Vert.x에 지원하는 언어는 Java 외에도 JavaScript, Ruby, Python, Groovy, Clojure, Scala가 있으며 해당 언어의 샘플과 매뉴얼은 [Vert.x 공식 홈페이지](#)¹²에서 제공한다. 물론

10 http://en.wikipedia.org/wiki/Reactor_pattern#Implementations

11 그렇다고 무조건 Vert.x만을 고집하는 것은 바람직하지 않다. Reactor 동시 처리 모델에 기반을 둔 각 구현체의 장단점을 파악하고, 개발하려는 서비스의 성격에 맞는 것을 선택하는 것이 중요하다.

12 <http://vertx.io/docs.html>

2개 이상의 언어를 동시에 사용하여 애플리케이션을 개발할 수도 있다.

JVM

Vert.x는 JVM 위에서 동작한다. 이것은 오랜 시간 동안 연구되고 발전한 JVM의 안정성, 성능 등의 이점을 Vert.x에서도 그대로 안고 갈 수 있다는 것을 의미한다. 또한, 방대한 Java 라이브러리 Third Party Library를 Vert.x에서 그대로 이용할 수 있다는 것은 분명 큰 장점이다. 그러나 기존 Java 라이브러리를 가져와서 사용할 때 한 가지 주의해야 할 점이 있는데, 표준 JDBC API와 같이 스레드를 블록시킬 수 있는 API를 이벤트 루프 스레드 내에서 사용하면 안 된다는 것이다. 나중에 설명하겠지만, 이러한 API는 Worker Verticle이라는 이벤트 루프 스레드와는 별도로 관리되는 Worker 스레드 풀을 통해 실행해야 한다.

범용성

Vert.x는 웹 애플리케이션은 물론 P2P, 로드 밸런서, 프락시 서버 등 이벤트 주도가 필요한 어떠한 상황에도 적용할 수 있는 범용성이 있다. 심지어 임베디드 방식을 사용한다면 기존 Java 애플리케이션에서도 Vert.x의 기능들을 활용할 수 있다.¹³

확장성

Vert.x에서는 최소한의 노력으로 수평 확장이 가능한 애플리케이션을 구성할 수 있다. 네트워크상의 서로 다른 JVM에서 동작하는 Vert.x를 클러스터로 구성할 수 있으며, 동일한 클러스터에 묶인 Vert.x끼리 이벤트 버스 Event Bus를 통해 String, JSON, Byte Buffer 형태의 메시지를 주고받을 수 있다. 심지어 Vert.x에서 제공하는 SockJS Event Bus Bridge를 사용하면 클라이언트 사이드의 웹 브라우저까지도 대규모 클러스터에 참여시킬 수 있다.

¹³ 임베디드 방식을 사용하려면 Vert.x에서 제공하는 *.jar 형태의 라이브러리를 기존 애플리케이션에 추가하면 된다. 그러나 이 책에서는 임베디드 방식의 Vert.x 사용은 다루지 않으므로 자세한 정보는 Vert.x 홈페이지에서 제공하는 매뉴얼을 참고하기 바란다.

1.3 Vert.x 설치 및 개발 환경 설정

지금까지 Vert.x의 동시 처리 모델의 장점과 Vert.x의 특징을 알아보았다. 이어서 Vert.x를 실제 프로젝트에 사용하기 위한 이클립스와 메이븐 기반의 개발 환경 설정 방법을 알아보자.

1.3.1 JDK 설치

Vert.x는 JDK가 필요하므로 먼저 JDK를 설치해야 한다. Vert.x는 JDK7 이상 버전에서만 사용할 수 있으므로 JDK6 이하 버전을 사용 중이라면 JDK7을 새로 설치해야 한다.

- 1) 사용하는 운영체제에 맞게 JDK7 이상 버전을 내려받는다.
- 2) 환경 변수에 JAVA_HOME을 추가하고 JDK를 설치한 경로를 입력한다.
- 3) \$JAVA_HOME\bin을 \$PATH에 추가한다.
- 4) 콘솔에서 java -version 명령을 입력하고 결과가 정상적으로 출력되는지 확인한다.

다음은 Windows 8 PowerShell에서 java -version 명령을 실행한 결과다.

```
PS C:\Users\yeon> java -version
java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)
```

1.3.2 Vert.x 설치

JDK 설치를 마쳤으면 이제 Vert.x 배포본을 설치해 보자. Vert.x의 최신 배포본은 [Vert.x 홈페이지¹⁴](#)에서 받을 수 있다.

¹⁴ <http://vertx.io/downloads.html>

- 1) Vert.x 최신 배포본을 내려받는다(현재 최신 버전은 2.1.5다).
- 2) 적당한 위치에 내려받은 파일의 압축을 푼다.
- 3) 환경 변수에 VERTX_HOME을 추가하고, Vert.x의 압축을 푼 경로를 입력한다.
- 4) \$VERTX_HOME\bin을 \$PATH에 추가한다.
- 5) 콘솔에 vertx version 명령을 입력하고 결과가 정상적으로 출력되는지 확인한다.

다음은 Windows 8 PowerShell에서 vertx version 명령을 실행한 결과다.

```
PS C:\Users\yeon> vertx version
2.1.1 (built 2014-06-18 14:11:03)
```

1.3.3 환경 설정

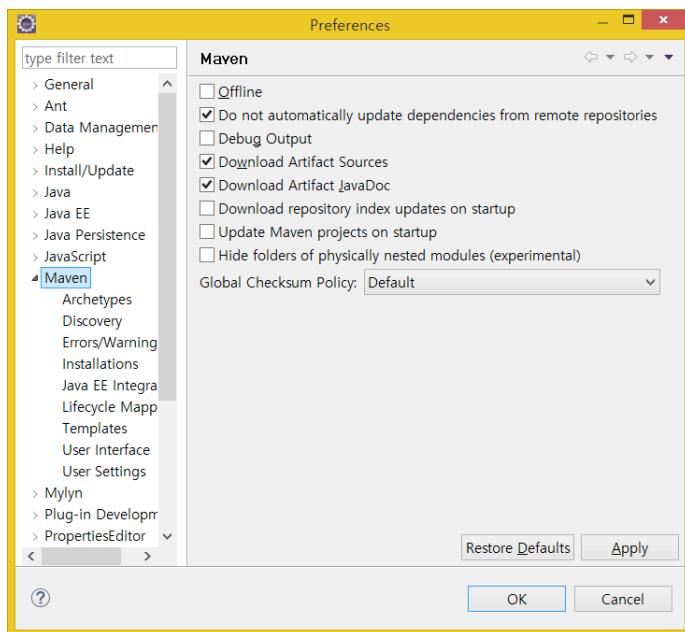
JDK와 Vert.x 설치가 모두 끝났다. 이제 이클립스에서 메이븐을 사용해 Vert.x 애플리케이션을 개발하기 위한 설정을 하면 된다.

먼저 이클립스 최신 배포본을 <http://www.eclipse.org/downloads>에서 내려 받는다. 메이븐을 사용하므로 가능하면 Maven Integration for Eclipse Plug-In이 포함된 Java EE Developers나 Java Developers 버전을 선택한다.¹⁵

이클립스에서 ‘Preferences → Maven’을 선택하여 [그림 1-3]과 같은 화면이 보이면 메이븐을 사용할 준비가 모두 끝난 것이다(체크된 항목은 다를 수 있다).

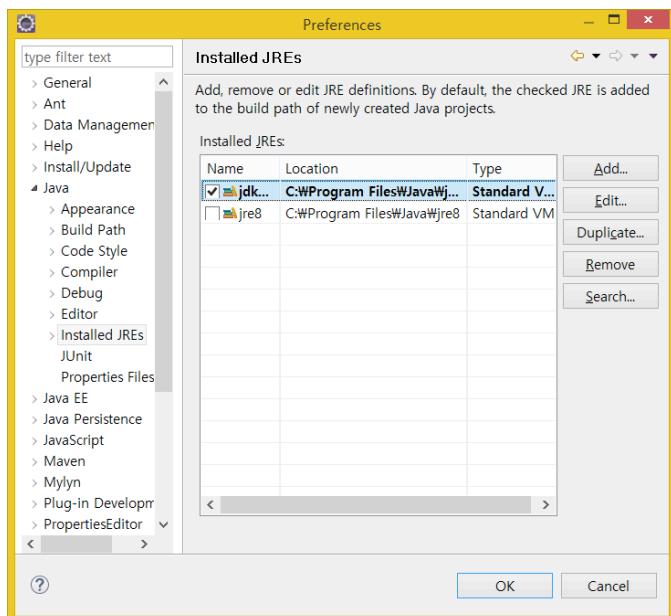
¹⁵ Maven Integration for Eclipse Plug-In이 포함되어 있지 않다면 Eclipse Marketplace에서 m2eclipse를 검색해서 사용 중인 이클립스에 맞는 최신 버전의 Plug-In을 설치한다.

[그림 1-3] Maven Integration for Eclipse Plug-In 확인



메이븐이 준비되었다면 이클립스에 JDK 설치 경로를 설정해 보자. ‘Preferences → Java → Installed JREs’를 선택하여 확인하면 기본적으로 시스템에 설치된 JRE가 설정되어 있다. Vert.x 애플리케이션을 개발하려면 JRE가 아닌 JDK가 필요하다. [그림 1-4]의 화면에서 우측의 Add 버튼을 클릭하고 JDK 설치 경로를 추가한 후 해당 JDK를 기본값으로 사용하도록 체크한다.

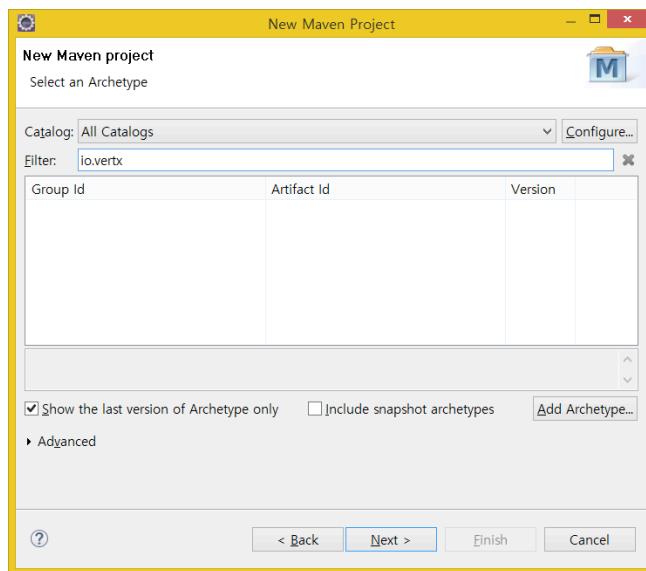
[그림 1-4] Installed JREs 설정



다음으로 메이븐의 Vert.x Archetype을 사용해 프로젝트를 생성해 보자. ‘File → New → Others’를 선택한 후, ‘Maven → Maven Project’를 선택한다. Select project name and location 부분에서 바로 Next를 선택하면 [그림 1-5]와 같이 Archetype을 지정하는 화면을 볼 수 있다. Filter에 io.vertx를 입력한다. 검색 결과가 아무것도 없다면 Vert.x Archetype을 설치해야 한다.¹⁶

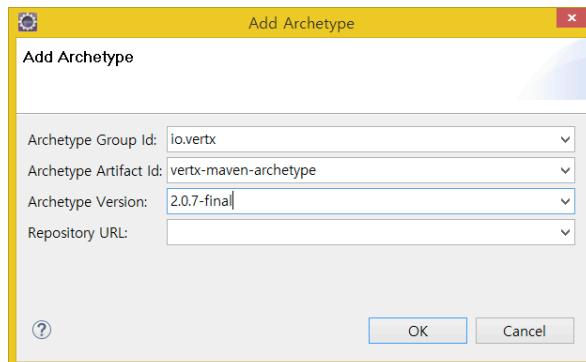
16 Vert.x Archetype이 이미 추가되어 있다면 다음 내용은 건너뛰어도 된다

[그림 1-5] 메이븐 Archetype 입력화면



<http://search.maven.org/>에서 vertx-maven-archetype를 검색한다. 가장 최신 버전의 GroupId, ArtifactId, Version을 확인한 후 [그림 1-5] 우측 하단의 Add Archetype 버튼을 클릭해 [그림 1-6]처럼 입력한다(현재 최신 버전은 2.0.11-final이다).

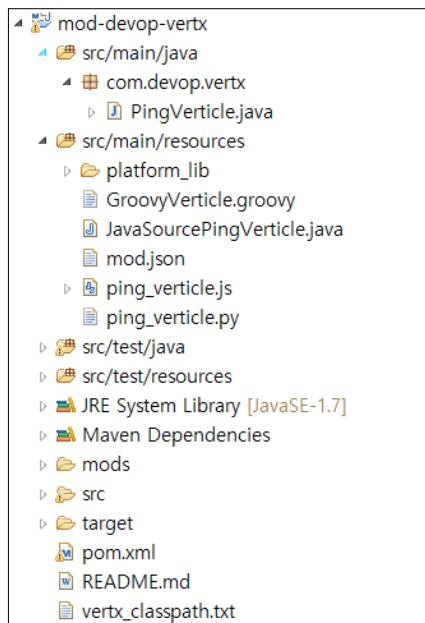
[그림 1-6] Vert.x Archetype 추가



완료되면 [그림 1-5]처럼 비어있는 검색 결과 화면이 아닌 Vert.x Archetype이 추가된 화면을 볼 수 있다. 해당 Archetype을 선택하고 Next를 클릭한다. 적절한 GroupId와 ArtifactId를 입력한 후 Finish를 클릭하면 프로젝트 생성이 완료된다(예시에서 프로젝트의 GroupId와 ArtifactId는 ‘com.devop.vertx’과 ‘mod-devop-vertx’를 입력했다).

Vert.x Archetype을 통해 자동 생성된 Vert.x 애플리케이션 프로젝트의 구조는 [그림 1-7]과 같다. 이 애플리케이션은 이벤트 버스로 메시지를 수신하면 ‘pong!’이라는 문자를 출력하는 PingVerticle을 다양한 언어(Java¹⁷, Groovy, JavaScript, Python)로 구현한 아주 간단한 프로그램이다.

[그림 1-7] Vert.x Archetype 샘플 프로젝트 구조



17 컴파일된 *.class와 원본 소스인 *.java 모두 Vert.x에서 사용할 수 있다.

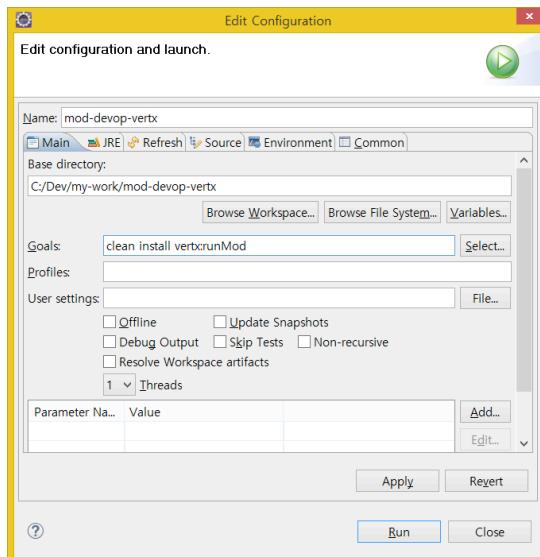
프로젝트를 구성하는 각 항목은 다음과 같다.

[표 1-1] Vert.x Archetype 샘플 프로젝트 구성

디렉터리	설명
src/main	java 해당 디렉터리를 포함한 모든 하위 디렉터리 내의 *.java 파일들은 컴파일된 후 *.class 형태로 Vert.x에서 사용된다. 즉, *.class 형태의 Verticle을 사용할 계획이라면 *.java 소스 코드를 이 디렉터리 내부에 두어야 한다.
	resources Vert.x에서 지원하는 다양한 언어로 구현된 Verticle들을 모아둔다. *.java 형태의 Verticle도 이 디렉터리 내부에 둘 수 있는데, 이런 경우 *.java는 컴파일되지 않는다. mod.json이라는 Vert.x 모듈 설정 파일도 여기에 둔다.
src/test	java *.class Verticle 테스트 케이스를 여기에 둔다.
	resources Vert.x에서 지원하는 다양한 언어로 구현된 Verticle 테스트 케이스를 여기에 둔다.

이제 Vert.x 애플리케이션을 빌드하고 실행해 보자. 이클립스에서 해당 프로젝트를 선택하고 ‘오른쪽 버튼 클릭 → Run As → Maven Build’(단축키는 Alt+Shift+X, M이다)를 선택하면 [그림 1-8]과 같은 화면이 나온다.

[그림 1-8] Vert.x 애플리케이션 실행



Main 탭의 Goals에 ‘clean install vertx:runMod’를 입력한 후¹⁸, 두 번째 JRE 탭에서 Runtime JRE가 JDK 7 이상 버전으로 지정되어 있는지 확인한다. JDK 6 이하 버전이라면 JDK 7 이상으로 변경해야 한다. 마지막으로 Common 탭에서 Encoding을 UTF-8로 설정한 후 실행한다. 필요한 모듈을 내려받고 src/test의 테스트 케이스가 성공적으로 실행되면 최종적으로 Vert.x 애플리케이션이 실행되는 것을 확인할 수 있다(‘PingVerticle started’라는 문자가 출력된다).

NOTE_ 오류 메시지 처리 방법

1. Vert.x 애플리케이션 실행이 실패하고, ‘Fatal error compiling: invalid target release: 1.7’이라는 오류 메시지가 나온다면 JDK 7 이상 버전을 사용하지 않아서 발생한 오류다. 이때는 프로젝트에서 ‘오른쪽 버튼 클릭 → Run As → Run Configurations’를 선택한 다음 [그림 1-8] 화면의 JRE 탭에서 JDK 7 이상 버전을 사용하도록 설정하면 정상적으로 실행된다.
2. Vert.x 애플리케이션 실행이 실패하고, ‘No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?’라는 오류 메시지가 나온다면 [그림 1-4]와 [그림 1-8]의 JRE 탭에서 JRE가 아닌 JDK로 수정해서 Vert.x 애플리케이션을 실행한다
3. Maven pom.xml 파일에서 ‘Plugin execution not covered by lifecycle configuration’이라는 오류가 발생할 수 있다. 해당 오류를 무시해도 Vert.x 애플리케이션을 빌드하고 실행하는 데에는 아무 문제 없다. 하지만 오류가 신경 쓰인다면 <https://www.eclipse.org/m2e/documentation/m2e-execution-not-covered.html> 내용을 참고하여 pom.xml 파일의 <build>/</build> 부분에 다음 내용을 추가한다.

```
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.eclipse.m2e</groupId>
      <artifactId>lifecycle-mapping</artifactId>
      <version>1.0.0</version>
```

18 <https://github.com/vert-x/vertx-maven>에서 더 많은 정보를 확인할 수 있다.

```
<configuration>
    <lifecycleMappingMetadata>
        <pluginExecutions>
            <pluginExecution>
                <pluginExecutionFilter>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-dependency-plugin</artifactId>
                    <versionRange>[1.0.0,)</versionRange>
                    <goals>
                        <goal>copy-dependencies</goal>
                    </goals>
                </pluginExecutionFilter>
                <action>
                    <ignore />
                </action>
            </pluginExecution>
        </pluginExecutions>
    </lifecycleMappingMetadata>
</configuration>
</plugin>
</plugins>
</pluginManagement>
```

1.4 간단한 Vert.x 애플리케이션

Vert.x가 설치된 것을 확인하기 위해 ‘hello, world’를 출력하는 아주 간단한 Vert.x 애플리케이션을 만들어 보면서 Vert.x의 주요 용어를 정리해 보자.

HelloVerticle.java 파일을 만들어서 [코드 1-1]의 내용을 에디터로 작성한다. 콘솔을 열고 ‘vertx run HelloVerticle.java’를 입력해 HelloVerticle을 실행하면 콘솔에 ‘hello, world’라는 문자가 출력된다.

NOTE

앞서 살펴본 이클립스와 메이븐 기반의 개발 환경은 기본적으로 Vert.x 모듈을 만드는 데 최적화되어 있다(Vert.x 모듈에 대해서는 4장에서 살펴본다). 물론 패키지 자동 import나 인텔리센스(IntelliSense) 같은 IDE 기능을 활용하면 좀 더 편리하게 HelloVerticle.java 같은 간단한 Verticle을 작성할 수 있다. 4장 이후부터는 Vert.x 모듈을 개발하는 데 이클립스와 메이븐 개발 환경을 적극적으로 활용한다.

[코드 1-1] Vert.x hello world 프로그램(HelloVerticle.java)

```
import org.vertx.java.core.logging.Logger;
import org.vertx.java.platform.Verticle;

public class HelloVerticle extends Verticle {
    private Logger logger;
    @Override
    public void start() {
        logger = container.logger();
        logger.info("hello, world");
    }
}
```

무슨 일이 벌어진 것일까? 하나씩 살펴보자. 먼저 Verticle이 의미하는 것이 무엇이지 확인할 필요가 있다. Verticle은 Vert.x의 가장 기본이 되는 구성 요소로, 실행과 배포 Deploy가 가능한 애플리케이션 코드라 할 수 있다. Vert.x 애플리케이션은 HelloVerticle처럼 1개의 Verticle로 구성될 수 있고, 2개 이상의 Verticle로 구성될 수도 있다. 각각의 Verticle은 서로 다른 클래스로 더로 만들어져 이벤트 루프 스레드에 할당되므로 Verticle 내부의 그 어떤 상태 변수도 공유되지 않으며, 최초 할당된 이벤트 루프 스레드에서만 실행된다는 특징이 있다. 이 때문에 Verticle을 작성할 때 스레드 동기화 문제를 걱정하지 않아도 된다.

org.vertx.java.platform.Verticle의 코드를 보자.

[코드 1-2] org.vertx.java.platform.Verticle

```
package org.vertx.java.platform;

import org.vertx.java.core.Future;
import org.vertx.java.core.Vertx;

public abstract class Verticle {
    protected Vertx vertx;
    protected Container container;

    public Container getContainer() {
        return container;
    }

    public void setContainer(Container container) {
        this.container = container;
    }

    public Vertx getVertx() {
        return vertx;
    }

    public void setVertx(Vertx vertx) {
        this.vertx = vertx;
    }

    public void start() {
    }

    public void start(Future<Void> startedResult) {
        start();
        startedResult.setResult(null);
    }

    public void stop() {
    }
}
```

`org.vertx.java.platform.Verticle`에서는 2개의 핵심 객체와 몇 개의 중요한 메서드를 제공하고 있으며, `Verticle`을 작성할 때 이 클래스를 상속받아서 필요한 메서드를 오버라이드 `Override`하면 된다.

[표 1-2] `Verticle`에서 제공하는 주요 객체 및 메서드

구분	설명	
객체	vertx	실행 중인 Vert.x 애플리케이션 Runtime 객체다.
	container	실행 중인 Vert.x 애플리케이션의 환경 정보와 설정 정보를 가지고 있다. Logging 객체도 여기에 포함되어 있다. 또한, 이 객체를 통해 <code>Verticle</code> 이나 모듈을 Deploy/Undeploy할 수 있다.
Method	start()	<code>Verticle</code> 이 Deploy되었을 때 실행되는 메서드다.
	stop()	<code>Verticle</code> 이 Undeploy되었을 때 실행되는 메서드다.

`Verticle`을 실행하려면 콘솔을 열고 '`vertx run YourVerticleFilePath`' 명령을 입력하면 되는데, 명령이 실행되면 새로운 Vert.x Instance가 만들어지고 Instance 내부에 `Verticle`이 배포된다.

Vert.x Instance는 JVM 위에서 실행되는 프로세스로, 하나의 Vert.x Instance에는 1개 이상의 `Verticle`이 포함될 수 있다. 동일한 Vert.x Instance 범위 안에 있는 `Verticle`들은 별도의 설정 없이도 이벤트 버스를 통해 서로 메시지를 주고받을 수 있다. 물론 네트워크의 서로 다른 JVM에서 동작하는 Vert.x Instance들을 클러스터로 구성할 수 있으며, 이때 클러스터에 참여한 Vert.x Instance의 `Verticle` 사이에서는 이벤트 버스를 통해 메시지를 주고받을 수 있다. `Verticle`은 서로 독립적이며 어떠한 상태 변수도 공유하지 않으므로 이 이벤트 버스가 `Verticle` 간 데이터를 공유할 수 있는 유일한 수단이다.¹⁹ [그림 1-2]를 다시 확인해 보면 4개의 이벤트 루프 스레드를 포함하는 Vert.x Instance에 총 8개의 `Verticle`이 배포된 형태임을 알 수 있다.

19 지금까지 따로 언급하지는 않았지만, 하나의 예외가 존재한다. Vert.x Instance 내부의 `Verticle` 사이에서는 `SharedData`를 통해 Immutable Type 데이터를 공유할 수 있다. `SharedData` 클러스터는 지원하지 않는다.

1.5 요약

지금까지 Vert.x의 주요 개념과 특징, 이클립스와 메이븐을 사용해 Vert.x 애플리케이션을 구현하는 방법까지 알아보았다. 시작이 반이라는 말처럼 개발자가 새로운 어떤 것을 배우는 데 가장 어렵고 시간을 들이는 부분이 바로 개발 환경을 설정하는 부분이다. 이제 힘든 부분은 지났으니 가벼운 마음으로 2장으로 넘어가 보자. 2장부터는 Vert.x로 채팅 서비스를 개발해 보고, Vert.x에서 제공하는 TCP, SockJS와 같은 기능을 자세히 알아본다.

2 | TCP 채팅 서버/클라이언트

Vert.x에서는 TCP 기반 서버와 클라이언트 개발이 편리하도록 다양한 기능을 제공한다. 지금부터 이러한 기능을 활용해 TCP 기반 채팅 서비스를 개발해 보자.

2.1 TCP 채팅 서버

Vert.x에서 TCP 기반 서버를 개발하기는 매우 쉽다. Verticle 내에서 NetServer 객체를 생성하고 여기에 몇 가지 이벤트 핸들러Event Handler를 등록하기만 하면 된다.

NetServer에 등록되는 이벤트 핸들러 역시 해당 NetServer 객체를 초기화하고, 소유하는 Verticle이 할당된 이벤트 루프 스레드 내에서만 실행되므로 마치 단일 스레드 애플리케이션을 개발한다고 생각하고 스레드 동기화 문제는 고민하지 않아도 된다. 또한, NetServer 객체 자체는 스레드 안전한 Thread-safe 특징을 지닌다.

가장 기본적인 NetServer 사용 방법은 NetServer에 TCP 클라이언트와의 연결이 수락되었을 때 호출되는 Connect Event Handler를 등록하는 것이다. 그 외 NetServer가 특정 IP 주소와 포트로 바인딩이 완료되었을 때, NetServer가 종료되었을 때 호출되는 이벤트 핸들러를 등록할 수 있다.

간단하게 Telnet 클라이언트로부터 문자열을 입력받아 출력하는 TCP 서버를 작성해 보자.

[코드 2-1] TCP 서버(TCPServerVerticle.java)

```
public class TCPServerVerticle extends Verticle {
```

```

private Logger logger;

@Override
public void start() {
    logger = container.logger();
    NetServer netServer = vertx.createNetServer();
    netServer.connectHandler(new Handler<NetSocket>() {
        @Override
        public void handle(NetSocket socket) {
            socket.dataHandler(new Handler<Buffer>() {
                @Override
                public void handle(Buffer buffer) {
                    logger.info("received data: "+buffer.toString());
                }
            });
        }
    });
    netServer.listen(8090, "localhost");
}
}

```

NOTE

이후 소스코드의 import 구문은 생략한다. 완전한 예제 소스 코드는 Github에서 확인할 수 있다.

전체적으로 매우 간결한 코드지만, Telnet 클라이언트에서 문자열을 입력받아 콘솔에 출력하는 간단한 기능을 수행하는 TCP 서버는 이것으로 충분하다. 이제 콘솔을 열고 ‘vertx run TCPServerVerticle.java’를 입력해 보자. Verticle이 정상적으로 배포되고 실행되었다면(Telnet 클라이언트에서 ‘connection refused’라는 오류가 발생하면 TCP 서버가 제대로 실행되지 않은 것이다) Telnet을 통해 TCP 서버에 연결하고 문자열을 전송할 수 있다. TCP 서버로 전송된 문자열은 TCP 서버

콘솔에 출력되는 것으로 확인할 수 있다.

그럼 [코드 2-1]을 참고하여 Vert.x에서 TCP 서버를 다루는 데 필요한 내용을 하나씩 알아보자.

먼저 Vert.x 애플리케이션 Runtime 객체인 `vertx`로부터 NetServer 객체를 생성한다. `vertx` 객체는 `TCPServerVerticle`이 상속하는 `org.vertx.java.platform.Verticle`에서 제공받는다(실행 중인 Vert.x 애플리케이션의 환경 정보와 설정 정보를 가진 `container` 객체도 `org.vertx.java.platform.Verticle`에서 제공받는다).

```
NetServer netServer = vertx.createNetServer();
```

다음으로 TCP 클라이언트와 연결이 수락되었을 때 호출되는 `connectHandler`를 `NetServer` 객체에 등록한다. `connectionHandler`에 전달되는 `NetSocket` 객체는 TCP 클라이언트와의 실제 연결점을 의미한다.

`NetSocket` 객체에는 [코드 2-1]처럼 메시지 수신을 처리하기 위해 `dataHandler`를 등록할 수 있다. `dataHandler`는 수신된 메시지의 바이트 버퍼`Byte Buffer`를 의미하는 `org.vertx.java.core.buffer.Buffer`를 전달받는다.

```
netServer.connectHandler(new Handler<NetSocket>() {  
    @Override  
    public void handle(NetSocket socket) {  
        socket.dataHandler(new Handler<Buffer>() {  
            @Override  
            public void handle(Buffer buffer) {  
                logger.info("received data: "+buffer.toString());  
            }  
        });  
    }  
});
```

```
    }  
});
```

또한, TCP 클라이언트의 연결이 종료되었을 때 호출되는 `closeHandler`와 예기치 못한 오류가 발생했을 때 호출되는 `exceptionHandler`를 `NetSocket` 객체에 등록할 수 있다.

```
socket.closeHandler(new VoidHandler() {  
    @Override  
    protected void handle() {  
        logger.info("connection closed: "+socket.remoteAddress());  
    }  
});  
  
socket.exceptionHandler(new Handler<Throwable>() {  
    @Override  
    public void handle(Throwable throwable) {  
        logger.error("unexpected exception: ", throwable);  
    }  
});
```

[코드 2-1]에서는 단순히 메시지 수신만을 처리하지만, `NetSocket`를 통해 TCP 클라이언트로 메시지를 전송하는 것도 가능하다. 메시지 전송을 위해 별도의 이벤트 핸들러를 등록할 필요는 없고, 단순히 `NetSocket`의 `write()` 메서드를 호출하면 된다. `write()` 메서드에는 `Buffer`나 `String`을 전달할 수 있다.

```
socket.write(new Buffer("hello, world"));  
socket.write("hello, world");
```

`NetServer`의 `connectHandler` 등록이 끝나면 `NetServer`가 TCP 클라이언트

연결을 대기할 특정 IP 주소와 포트를 바인드 bind 한다.

```
netServer.listen(8090, "localhost");
```

NetServer의 바인드 작업은 비동기로 처리되므로 listen() 메서드를 호출하고 시간이 조금 지난 후에 바인드 작업 결과를 확인할 수 있다. 바인드 작업이 항상 성공하는 것은 아니므로 필요하다면 바인드 작업의 결과를 확인하고 예외 처리를 해야 할 수도 있다. 이를 위해 listen() 메서드의 세 번째 파라미터로 이벤트 핸들러를 등록할 수 있다.

```
netServer.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {  
    @Override  
    public void handle(AsyncResult<NetServer> asyncResult) {  
        logger.info("bind result: "+asyncResult.succeeded());  
    }  
});
```

마지막으로 NetServer를 종료하려면 간단히 close() 메서드를 호출하면 된다. 그러나 close() 메서드 역시 비동기로 처리되므로 close() 메서드 호출이 완료되었다고 해서 NetServer가 완전히 종료된 것은 아니다. listen() 메서드와 마찬가지로 이벤트 핸들러를 등록해서 NetServer가 완전히 종료되었을 때 이벤트를 전달받을 수 있다.

```
netServer.close(new AsyncResultHandler<Void>() {  
    @Override  
    public void handle(AsyncResult<Void> asyncResult) {  
        logger.info("close result: "+asyncResult.succeeded());  
    }  
});
```

```
});
```

지금까지의 내용을 요약하면 다음과 같다. 신규 TCP 클라이언트 연결, 메시지 수신, 클라이언트 연결 종료, 오류 발생과 같은 이벤트 처리를 위해 NetServer와 NetSocket에 이벤트 핸들러를 등록하고, TCP 클라이언트와의 연결점을 의미하는 NetSocket을 통해 메시지를 전송할 수 있다.

앞의 내용을 바탕으로 간단한 채팅 서비스를 위한 TCP 서버를 만들어 보자.

[코드 2-2] TCP 채팅 서버(TCPChatServerVerticle.java)

```
public class TCPChatServerVerticle extends Verticle {
    private Logger logger;
    private NetServer server;
    private List<NetSocket> sockets;

    @Override
    public void start() {
        logger = container.logger();
        server = vertx.createNetServer();
        sockets = new ArrayList<NetSocket>();

        server.connectHandler(new Handler<NetSocket>() {
            @Override
            public void handle(final NetSocket socket) {
                sockets.add(socket);
                //-- this handler will be called every time data is received on the
                socket
                socket.dataHandler(new Handler<Buffer>() {
                    @Override
                    public void handle(Buffer buffer) {
                        for (NetSocket s : sockets) {
                            if (!socket.equals(s))
                                s.write(buffer);
                        }
                    }
                });
            }
        });
    }
}
```

```

        }
    }
});

//-- socket closed
socket.closeHandler(new VoidHandler() {
    @Override
    protected void handle() {
        logger.info("connection closed: "+socket.remoteAddress());
        Iterator<NetSocket> it = sockets.iterator();
        while (it.hasNext()) {
            if (socket.equals(it.next())) {
                it.remove();
                break;
            }
        }
    }
});

//-- something went wrong
socket.exceptionHandler(new Handler<Throwable>() {
    @Override
    public void handle(Throwable throwable) {
        logger.error("unexpected exception: ", throwable);
    }
});
}

server.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {
    @Override
    public void handle(AsyncResult<NetServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});
}

@Override

```

```
public void stop() {  
    if ( server != null )  
        server.close();  
}  
}
```

[코드 2-1]과 비교하면 몇 가지 이벤트 핸들러가 더 추가되었을 뿐 별로 달라진 것이 없다. 그러나 connectHandler, dataHandler, closeHandler의 달라진 부분에 주목해 보자. 먼저 connectHandler에는 신규 연결된 클라이언트의 NetSokcet을 저장하기 위해 ArrayList가 추가되었고, dataHandler에서 전달받은 메시지를 ArrayList에 저장된, 자신을 제외한 모든 NetSokcet에 전달하는 것을 확인할 수 있다. closeHandler에서는 연결이 종료된 클라이언트의 NetSokcet을 ArrayList에서 제거한다. 즉, TCP 채팅 서버가 전달받은 채팅 메시지를 자신을 제외한 모든 클라이언트에(좀 더 정확한 의미로는 채팅 메시지를 전송한 클라이언트를 제외한 다른 모든 클라이언트가 된다) 브로드캐스팅broadcasting한다.

기본적인 TCP 채팅 서버는 이것으로 끝이다. 생각보다 코드의 양이 많다고 생각할지도 모르겠다. Java 언어의 문법적 제약으로 비교적 장황한 코드가 만들어졌지만, 최근 도입된 Java 8의 람다식을 이용하면 훨씬 간결한 코드를 작성할 수 있다.⁰¹ 물론 Java 8까지 갈 것도 없이 [코드 2-2]와 완전히 동등한 코드를 훨씬 간결하게 JavaScript로 작성할 수도 있다.

이것으로 채팅 서비스를 위한 TCP 서버를 마무리하고, 다음으로 채팅 서비스 TCP 클라이언트를 만들어 보자. 그런 다음 TCP 채팅 서버와 연결해서 아무 문제가 없는지 테스트를 진행해 보자.⁰²

01 예제 소스가 올려져 있는 [Github](#)에는 이 책의 예제 코드를 람다식을 활용해 다시 만든 코드도 있다.

02 사실 [코드 2-2] TCP 채팅 서버에는 필자가 일부러 의도한 두 가지 문제점이 있다. 두 가지 문제점은 이후에 설명하겠다.

2.2 TCP 채팅 클라이언트

앞에서 TCP 기반 서버를 다루기 위해 사용했던 핵심 객체가 NetServer였다면, TCP 기반 클라이언트를 다루기 위해서는 NetClient 객체를 사용한다. NetClient는 NetServer와 마찬가지로 Vert.x 애플리케이션 Runtime 객체인 vertx로 생성할 수 있다.

```
NetClient client = vertx.createNetClient();
```

NetClient에 등록되는 이벤트 핸들러 역시 NetServer에 등록되는 이벤트 핸들러와 동일한 특징을 지닌다. 즉, NetClient 객체를 초기화하고 소유하는 Verticle이 할당된 이벤트 루프 스레드 내에서만 NetClient에 등록된 이벤트 핸들러가 실행된다. 마찬가지로 NetClient 객체 자체는 스레드 안전한 특징을 지닌다.

TCP 서버와의 연결은 NetClient의 connect() 메서드를 호출해서 생성할 수 있다. connect() 메서드는 비동기로 처리되므로 이벤트 핸들러를 등록해서 TCP 서버와의 연결 요청 처리 결과를 확인해야 한다. 연결이 성공적으로 이루어지면 이벤트 핸들러에 전달된 NetSocket을 통해 서버와 메시지를 주고받을 수 있다. 이는 NetSocket 객체가 TCP 서버와의 실제적인 연결점임을 의미하고, 앞서 NetServer의 connectHandler에서 확인한 NetSocket과 동일한 타입의 객체라는 것을 알 수 있다.

[코드 2-3] TCP 채팅 클라이언트(TCPChatOnlyRecvClientVerticle.java)

```
public class TCPChatOnlyRecvClientVerticle extends Verticle {  
    private Logger logger;  
    private NetClient client;
```

```

@Override
public void start() {
    logger = container.logger();
    client = vertx.createNetClient();

    client.connect(8090, "localhost", new Handler<AsyncResult<NetSocket>>() {
        @Override
        public void handle(AsyncResult<NetSocket> asyncResult) {
            logger.info("connect result: " + asyncResult.succeeded());
            if (asyncResult.succeeded()) {
                final NetSocket socket = asyncResult.result();
                // -- this handler will be called every time data is received on
the socket
                socket.dataHandler(new Handler<Buffer>() {
                    @Override
                    public void handle(Buffer buffer) {
                        logger.info("received data: "+buffer.toString());
                    }
                });
                // -- socket closed
                socket.closeHandler(new VoidHandler() {
                    @Override
                    protected void handle() {
                        logger.info("connection closed:
"+socket.remoteAddress());
                    }
                });
                // -- something went wrong
                socket.exceptionHandler(new Handler<Throwable>() {
                    @Override
                    public void handle(Throwable throwable) {
                        logger.error("unexpected exception: ", throwable);
                    }
                });
            }
        }
    });
}

```

```
        }
    });
}

@Override
public void stop() {
    if (client != null)
        client.close();
}
}
```

[코드 2-3]은 아직 완전한 TCP 채팅 클라이언트가 아니다. 단지 TCP 서버와 연결을 생성한 후 서버가 보내오는 메시지를 받아 출력하는 기능만 있을 뿐이고 사용자로부터 텍스트를 입력받아 TCP 서버로 전송하는 부분은 빠져있다.

사용자로부터 텍스트를 입력받는 부분은 `java.io.Console`로 처리할 수 있다. 그러나 여기서 한 가지 주의할 점이 있다. `java.io.Console`로 사용자 입력 텍스트를 읽어오는 API는 사용자가 입력을 완료할 때까지 스레드를 블록시킨다. 이처럼 스레드를 블록시키는 Java API를 Vert.x의 이벤트 루프 스레드 내에서 실행시키면 Vert.x 애플리케이션 전체에 심각한 성능 문제를 일으킨다. 이런 경우에는 이벤트 루프 스레드가 아닌, 별도로 관리되는 Worker 스레드 풀을 통해 해당 Verticle을 실행해야 한다. Vert.x에서는 이것을 Worker Verticle이라 한다.

사용자로부터 입력 텍스트를 읽어오는 Worker Verticle을 작성해 보자.

[코드 2-4] 사용자 입력 텍스트 처리 Worker Verticle(TCPChatClientWorkerVerticle.java)

```
public class TCPChatClientWorkerVerticle extends Verticle {
    private EventBus eb;
    private boolean readline;

    @Override
```

```
public void start() {
    eb = vertx.eventBus();
    readline = true;

    Console console = System.console();
    if ( console != null ) {
        while ( readline ) {
            String line = console.readLine();
            if ( line.equals("exit") )
                readline = false;
            else
                eb.send("com.devop.vertx.chat", line);
        }
    }
}

@Override
public void stop() {
    readline = false;
}
```

사용자 입력 텍스트를 읽기 위해 `Console` 객체의 `readLine()` 메서드가 사용됐다. `readLine()` 메서드는 사용자 입력이 완료될 때까지 스레드를 블록시키고, 읽어온 텍스트는 개행문자를 포함하지 않는다. 그리고 이벤트 버스가 어떻게 사용되고 있는지 주의 깊게 살펴보자.

```
EventBus eb = vertx.EventBus();
```

이벤트 버스는 동일한 Vert.x Instance 내의 Verticle은 물론 클러스터로 묶인 Vert.x Instance의 Verticle 사이에서 메시지를 공유할 수 있는 유일한 수단이다.

[코드 2-4]에서는 서버로 전송하는 역할을 하는 Verticle로 사용자 입력 텍스트를 전달하기 위해 이벤트 버스를 사용했다. 이벤트 버스의 send() 메서드의 첫 번째 파라미터 입력값인 ‘com.devop.vertx.chat’은 이벤트 버스의 주소를 의미하며, 다른 Verticle에서도 동일한 이벤트 버스 주소를 사용해서 사용자 입력 텍스트를 받을 수 있다.

이번에는 [코드 2-3]을 조금 수정해 [코드 2-5]를 작성한다. [코드 2-5]는 사용자 입력 텍스트를 이벤트 버스로 받고 서버로 전송하는 역할을 한다.

[코드 2-5] TCP 채팅 클라이언트(TCPChatClientVerticle.java)

```
public class TCPChatClientVerticle extends Verticle {
    private Logger logger;
    private EventBus eb;
    private NetClient client;
    private String writeHandlerID;

    @Override
    public void start() {
        logger = container.logger();
        eb = vertx.eventBus();
        client = vertx.createNetClient();

        client.connect(8090, "localhost", new Handler<AsyncResult<NetSocket>>() {
            @Override
            public void handle(AsyncResult<NetSocket> asyncResult) {
                logger.info("connect result: " + asyncResult.succeeded());
                if (asyncResult.succeeded()) {
                    final NetSocket socket = asyncResult.result();
                    writeHandlerID = socket.writeHandlerID();
                    //-- this handler will be called every time data is received on
                    the socket
                }
            }
        });
    }
}
```

```

        socket.dataHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                logger.info("received data: "+buffer.toString());
            }
        });
        //-- socket closed
        socket.closeHandler(new VoidHandler() {
            @Override
            protected void handle() {
                logger.info("connection closed:
"+socket.remoteAddress());
            }
        });
        //-- something went wrong
        socket.exceptionHandler(new Handler<Throwable>() {
            @Override
            public void handle(Throwable throwable) {
                logger.error("unexpected exception: ", throwable);
            }
        });
    }
});
eb.registerHandler("com.devop.vertx.chat", new
Handler<Message<String>>() {
    @Override
    public void handle(Message<String> message) {
        if( writeHandlerID != null )
            eb.send(writeHandlerID, new Buffer(message.body()));
    }
});

```

```
}

@Override
public void stop() {
    if (client != null)
        client.close();
}
}
```

[코드 2-3]과 비교하여 달라진 점을 살펴보자. 이벤트 버스로 사용자 입력 메시지를 수신하는 이벤트 핸들러가 추가되었고, 이벤트 핸들러에서 서버로 메시지를 전송하는 데 `NetSocket`을 직접 사용하지 않는 부분이 눈에 띈다.

이벤트 버스는 Vert.x 애플리케이션 개발에서 빼놓을 수 없는 필수 부분이다. 앞으로 이 책의 모든 예제에서 꾸준히 사용하는 만큼 이벤트 버스를 좀 더 자세히 확인하고 넘어가자.

이벤트 버스

이벤트 버스를 통해 메시지를 가져오는 것은 매우 간단하다. 이벤트 버스의 `registerHandler()` 메서드의 첫 번째 파라미터로 이벤트 버스 주소를 입력하면 해당 이벤트 버스 주소로 메시지가 수신되었을 때 `registerHandler()` 메서드의 두 번째 파라미터인 이벤트 핸들러가 실행된다.

```
eb.registerHandler("com.devop.vertx.chat", new Handler<Message<String>>() {
    @Override
    public void handle(Message<String> message) {
        logger.info("received data: "+message.body());
    }
});
```

즉, Verticle 사이에서 메시지를 공유하기 위해서는 오직 이벤트 버스 주소만 알면 된다. 이벤트 버스 주소는 단순 문자열로 어떤 제약도 없다. 외부 Vert.x 모듈이나 Verticle을 이용할 때 이벤트 버스 주소가 충돌되지 않도록 자신만의 고유한 값을 지정해주면 된다.⁰³

특정 이벤트 버스 주소로 더는 메시지를 수신하고 싶지 않다면 이벤트 버스의 `unregisterHandler()` 메서드를 호출한다. `unregisterHandler()` 메서드의 파라미터는 `registerHandler()` 메서드를 호출할 때 입력한 이벤트 버스 주소와 이벤트 핸들러를 그대로 입력하면 된다.

```
Handler<Message<String>> myHandler = new Handler<Message<String>>() {
    @Override
    public void handle(Message<String> message) {
        //-- TODO
    }
};

eb.registerHandler("com.devop.vertx.chat", myHandler);
eb.unregisterHandler("com.devop.vertx.chat", myHandler);
```

이벤트 버스로 메시지를 송/수신할 때 ‘publish/subscribe messaging’과 ‘Point to point and Request-Response messaging’이라는 두 가지 패턴을 사용한다.

‘publish/subscribe messaging’은 메시지를 브로드캐스팅하기 위한 용도로 사용하는데, 메시지를 특정 이벤트 버스 주소로 publish할 때 해당 이벤트 버스 주소에 등록된 모든 이벤트 핸들러가 메시지를 수신한다.

```
eb.publish("com.devop.vertx.chat", "hello, world");
```

03 Java 패키지명처럼 도메인을 역순으로 정렬하는 방식을 많이 쓴다.

'Point to point and Request-Response messaging'은 메시지를 특정 이벤트 핸들러 한 개에만 전달하기 위한 용도로 사용된다. 이벤트 버스 주소에 한 개 이상의 이벤트 핸들러가 등록되기도 하는데, 이런 경우 Round-Robin 알고리즘에 의해 순차적으로 한 개의 이벤트 핸들러가 선택되고 메시지가 전달된다. 때로는 메시지를 수신한 이벤트 핸들러 쪽에서 메시지를 전송한 쪽으로 응답 메시지를 보내야 하는 경우가 있다. 이런 경우 수신 측이 reply() 메서드로 응답 메시지를 보내면 발신 측에 등록해둔 Reply Event Handler가 실행된다.

```
//-- sender
eb.send("com.devop.vertx.chat", "this is a message", new
Handler<Message<String>>() {
    public void handle(Message<String> message) {
        logger.info("received reply:" + message.body());
    }
});
//-- recver
eb.registerHandler("com.devop.vertx.chat", new Handler<Message<String>>() {
    public void handle(Message<String> message) {
        logger.info("received message: " + message.body());
        //-- do some stuff
        //-- now reply to it
        message.reply("this is a reply");
    }
});
```

이벤트 버스로 처리할 수 있는 메시지 타입은 primitive type, String, JSON Object, JSON Array, Buffer가 있다(null 전송도 가능하다). 특히 JSON Object, JSON Array, Buffer 타입은 메시지가 전송되기 전 객체의 깊은 복사DeepCopy가 이루어진다. 이것은 메시지의 발신과 수신 모두가 동일한 JVM 내에서

이루어지더라도 서로 동일한 객체를 참조하지 않게 하기 위해서고, 또한 스레드 경합 상태를 만들어내는 것을 방지한다.

지금까지 알아본 내용처럼 이벤트 버스 관련 API는 단순하고 일괄적이며 사용하기 쉽다. 그러나 이벤트 버스로 메시지를 처리할 때 한 가지 주의해야 할 점이 있다. 이벤트 버스에서 처리되는 메시지는 휘발성 *transient*이라서 이벤트 버스 관련 API 내에서 오류가 발생하면 메시지를 유실할 수 있다. 따라서 개발하려는 Vert.x 애플리케이션에서 이벤트 버스로 메시지의 전송과 수신이 엄격하게 보장되어야 한다면 유실된 메시지를 복구할 수 있는 메커니즘을 자체적으로 구축해야 한다.

Event Bus Write Handler

[코드 2-5]로 돌아가서 내용을 살펴보면 `registerHandler()` 메서드에 익숙하지 않은 부분이 있을 것이다. 서버로 메시지를 전송하는 데 `NetSocket`을 직접 사용하지 않고 대신 이벤트 버스를 사용하는 부분이다.

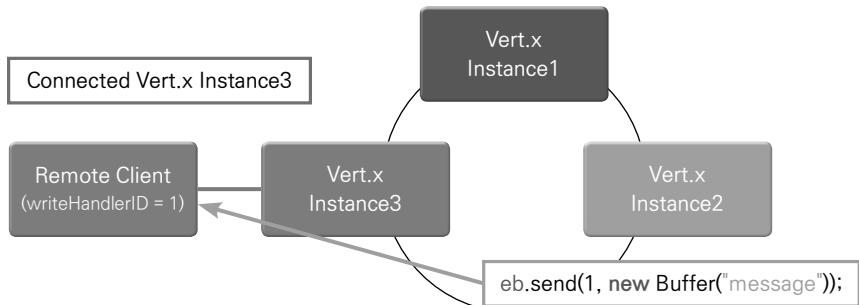
```
eb.send(writeHandlerID, new Buffer(message.body()));
```

모든 `NetSocket` 객체는 자동으로 하나의 이벤트 핸들러를 이벤트 버스에 등록한다. 그리고 이 이벤트 핸들러에 메시지가 수신되면 `NetSocket`을 통해 원격 호스트로 해당 이벤트 핸들러가 수신한 메시지를 그대로 전송하는 특징이 있다. `NetSocket`이 자동으로 등록한 이벤트 핸들러의 주소는 `NetSocket` 객체의 `writeHandlerID()` 메서드로 알 수 있다. [코드 2-5]에서 `connect()` 메서드의 이벤트 핸들러 부분을 보면 `writeHandlerID()` 메서드를 통해 서버와 연결된 `NetSocket`의 `writeHandlerID` 값을 가져오는 부분이 있는데, 이 `writeHandlerID` 값을 이용해서 서버로 사용자 입력 텍스트 메시지를 전송한다.

이와 같은 특징은 아무리 거대하고 복잡한 네트워크 연결을 하는 애플리케이션

일지라도 Vert.x를 사용하면 어렵지 않게 개발할 수 있음을 의미한다. 클라이언트와 실제 연결이 이루어진 `NetSocket`을 소유하고 있는 `Verticle`이 아니라도 해당 `NetSocket`의 `writeHandlerID`만 알고 있으면 어떤 위치의 `Verticle`에서도 메시지를 전송할 수 있기 때문이다. 심지어 네트워크상의 서로 다른 JVM에서 동작하는 다수의 Vert.x Instance가 클러스터로 묶인 상황이라도 이벤트 버스만 제대로 동작하고 있으면 이러한 특징은 여전히 유효하다.

[그림 2-1] Event Bus Write Handler



지금까지 TCP 채팅 클라이언트를 작성하는 데 필요한 몇 가지 핵심적인 내용을 살펴보았다. 이제 다시 TCP 채팅 클라이언트로 돌아와 [코드 2-4]의 사용자 입력 텍스트를 처리하기 위한 `TCPChatClientWorkerVerticle`과 [코드 2-5]의 사용자 입력 텍스트를 서버로 전송하는 `TCPChatClientVerticle`을 함께 실행하기 위한 방법을 알아보자.

`Verticle` 자체를 실행하는 것은 지금까지 해왔던 것처럼 콘솔에서 '`vertx run YourVerticleFilePath`' 명령을 실행하면 된다. 그러나 동시에 두 개의 `Verticle`을 실행해야 하고, 이 중 한 개는 `Worker Verticle`로 실행해야 한다는 조건이 있다.⁰⁴ 이러한 조건을 위해 [코드 2-6]을 작성한다.

04 앞서 언급한 것처럼 `TCPChatClientWorkerVerticle`은 스레드 블록 코드를 포함하기 때문에 반드시 이벤트 루프 스레드와는 별도로 관리되는 `Worker` 스레드 풀에서 실행되어야 한다.

[코드 2-6] TCP 채팅 클라이언트 실행 Verticle(TCPChatClientStarterVerticle.java)

```
public class TCPChatClientStarterVerticle extends Verticle {  
    @Override  
    public void start() {  
        container.deployVerticle("TCPChatClientVerticle.java");  
        container.deployWorkerVerticle("TCPChatClientWorkerVerticle.java");  
    }  
}
```

[코드 2-6]은 container 객체를 사용하는 프로그램으로, Verticle을 실행하는 예를 보여준다. `deployVerticle()` 메서드는 콘솔에서 '`vertx run YourVerticle FilePath`' 명령을 실행하는 것과 동일한 표준 Verticle을 실행하는 방법이다. `deployWorkerVerticle()` 메서드는 이름이 의미하는 것처럼 Worker 스레드풀에서 Verticle이 실행되도록 한다. 이것은 콘솔에서 '`- worker`' 옵션을 주고 '`vertx run`' 명령을 실행한 것(`vertx run - worker YourVerticleFilePath`)과 같다.

이제 콘솔에서 '`vertx run TCPChatClientStarterVerticle.java`' 명령을 입력해 보자. TCP 채팅 클라이언트가 실행되고 서버와 연결이 완료되면 콘솔로 입력한 텍스트를 전송할 수 있다(TCP 채팅 클라이언트 실행 전 TCP 채팅 서버를 먼저 실행해 놓자. TCP 채팅 서버를 실행하려면 콘솔에서 '`vertx run TCPChatServerVerticle`'을 입력하면 된다).

다음은 Windows 8의 PowerShell에서 한 개의 `TCPChatServerVerticle`과 두 개의 `TCPChatClientStarterVerticle`를 실행한 결과다. 모두 정상적으로 실행되면 한쪽 TCP 채팅 클라이언트에서 입력한 텍스트가 다른 쪽 TCP 채팅 클라이언트에서 출력되는 것을 확인할 수 있다.

```
PS C:\Users\yeon> ls
```

```
디렉터리: C:\Users\yeon
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	2014-07-20 오후 6:18	315	TCPChatClientStarterVerticle.java
-a---	2014-07-20 오후 3:23	2191	TCPChatClientVerticle.java
-a---	2014-07-17 오후 9:56	676	TCPChatClientWorkerVerticle.java
-a---	2014-07-17 오후 10:02	2177	TCPChatServerVerticle.java

```
PS C:\Users\yeon> vertx run TCPChatServerVerticle.java
```

```
Succeeded in deploying verticle  
bind result: true
```

```
PS C:\Users\yeon> vertx run TCPChatClientStarterVerticle.java
```

```
Succeeded in deploying verticle  
connect result: true  
hi  
received data: hello?
```

```
PS C:\Users\yeon> vertx run TCPChatClientStarterVerticle.java
```

```
Succeeded in deploying verticle  
connect result: true  
received data: hi  
hello?
```

2.3 첫 번째 결함의 분석과 해결방법

간단한 테스트 결과로 보면 TCP 채팅 서버와 클라이언트 모두 제대로 동작하는 것 같지만, 사실 앞의 예제는 두 가지 중대한 결함이 있다. 첫 번째 결함은 Vert.x의 특성에 기인한 것이고, 두 번째 결함은 TCP 프로토콜 처리가 올바르지 않아서 발생하는 것이다.

먼저 첫 번째 결함을 밝혀내기 위해 [코드 2-2]의 connectHandler를 다음과 같이 수정하자. 기능 수정은 아니고 스레드 아이디를 포함하는 몇 개의 로그를 출력하도록 수정한 것이다.

```
server.connectHandler(new Handler<NetSocket>() {
    @Override
    public void handle(final WebSocket socket) {
        logger.info("Thread ["+Thread.currentThread().getId()+"] client
connected: "+socket.remoteAddress());
        sockets.add(socket);
        //-- this handler will be called every time data is received on the socket
        socket.dataHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                logger.info("Thread ["+Thread.currentThread().getId()+"] received
data: "+buffer.toString());
                for (WebSocket s : sockets) {
                    if (!socket.equals(s)) {
                        logger.info("Thread ["+Thread.currentThread().getId()+"]
send data: "+buffer.toString());
                        s.write(buffer);
                    }
                }
            }
        });
        //-- 생략 --
    }
});
```

수정이 끝나면 다시 TCP 채팅 서버를 실행한다. 단, 이번에는 '-instances 2' 옵션을 줘서 2개의 TCPChatServerVerticle이 실행되도록 한다. 이어서 3개의

TCPChatClientStarterVerticle을 실행하고 메시지를 전송해 보자.

NOTE

TCPChatServerVerticle 실행 시 ‘-instances 2’ 옵션을 주면 ‘bind result: true’ 로그가 2개 생성되는데, 이것이 동일한 호스트와 포트(여기서는 8090)에 동시에 바인드되었다는 결과는 아니다. 한 개의 호스트와 포트는 당연히 한 번의 바인드만 가능하며, 두 번째 바인드부터는 이미 사용 중인 포트라는 바인드 오류가 발생한다. ‘-instances N’ 옵션으로 N개의 TCPChatServerVerticle을 실행해도 내부적으로 호스트와 포트에 바인드되는 것은 단 1개의 서버고, 클라이언트 연결 요청이 완료되었을 때 Round-Robin 알고리즘에 의해 순차적으로 1개의 connectHandler가 선택되고 실행된다. 이것은 별도의 처리가 없어도 자동으로 시스템의 CPU 개수만큼 TCPChatServerVerticle를 배포하고 실행하여 병렬 처리의 이점을 활용할 수 있게 한다. 물론 각각의 TCPChatServerVerticle은 여전히 스레드 동기화 문제를 신경 쓰지 않아도 된다.

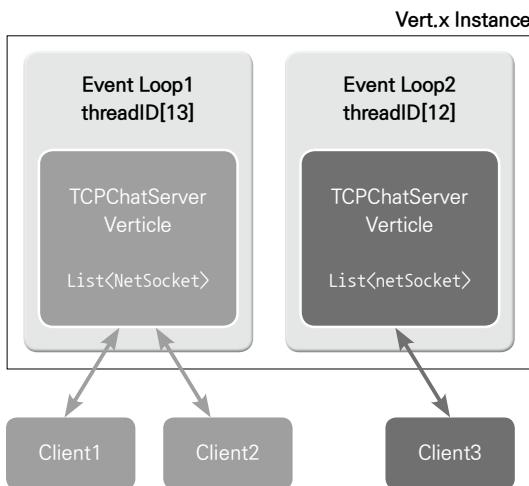
어떤 일이 벌어지는가? 3개 중 2개의 TCP 채팅 클라이언트만 메시지를 주고 받을 뿐 나머지 1개의 TCP 채팅 클라이언트는 전혀 메시지를 수신하지 못하는 현상이 생긴다. 다음은 TCPChatServerVerticle 실행 로그다. 로그를 자세히 살펴보자.

```
PS C:\Users\yeon> vertx run TCPChatServerVerticle.java -instances 2
Succeeded in deploying verticle
bind result: true
bind result: true
Thread [13] client connected: /127.0.0.1:4655
Thread [12] client connected: /127.0.0.1:4672
Thread [13] client connected: /127.0.0.1:4689
Thread [13] received data: hello?
Thread [13] send data: hello?
Thread [12] received data: hi?
```

‘-instances 2’ 옵션에 의해 2개의 TCPChatServerVerticle이 실행되고 있고

각각의 이벤트 루프 스레드 아이디는 12와 13이다. 3개의 채팅 클라이언트가 연결 요청했을 때 2개는 13번 이벤트 루프 스레드에 나머지 1개는 12번 이벤트 루프 스레드에 연결되었다. 13번 이벤트 루프 스레드에 연결된 2개의 TCP 채팅 클라이언트는 서로 메시지를 주고받을 수 있으나 12번 이벤트 루프 스레드 연결된 TCP 채팅 클라이언트는 다른 2개의 TCP 채팅 클라이언트와 메시지를 주고받을 수 없다. 이는 TCP 채팅 서버의 TCPChatServerVerticle이 실행되는 이벤트 루프 스레드 별로 TCP 채팅 클라이언트를 독립적으로 관리하기 때문에 발생하는 문제다.

[그림 2-2] TCP 채팅 서비스 첫 번째 결함의 원인



원인을 파악했으니 결함을 수정해 보자. 방법은 간단하다. TCPChatServer Verticle 별로 독립적으로 관리되는 `List<NetSocket>`을 모든 TCPChatServer Verticle이 공유할 수 있도록 만든다. 이를 위해 `vertx` 객체가 제공하는 `SharedData`를 사용한다. `SharedData`는 동일한 Vert.x Instance 내의 Verticle 이 `Map`과 `Set` 두 가지 자료구조⁰⁵를 이용해 Number, String, Boolean, Buffer

05 정확하게는 `org.vertx.java.core.shareddata.impl.SharedMap`, `org.vertx.java.core.shareddata.impl.SharedSet` 타입의 자료구조를 제공하며, 이들 자료구조는 `java.util.concurrent.ConcurrentHashMap`에 기반을 둔다.

같은 Immutable 객체 타입⁰⁶의 데이터를 공유하게 해주는 장치다. 특히 byte, Buffer 타입의 데이터는 자동으로 복사되어 Verticle로 전달되므로 각 Verticle은 절대 동일한 byte[], Buffer 타입의 데이터를 참조하지 않는다. 사용하는 방법은 SharedData의 getMap() 또는 getSet() 메서드에 접근 키를 입력하면 된다. Verticle에 관계없이 같은 값의 접근 키로 언제나 동일한 Map 또는 Set을 얻을 수 있다. 그러나 현재 SharedData는 클러스터는 지원하지 않아서 사용이 제한적이다. 차후 클러스터로 둑인 Vert.x Instance 사이에서도 SharedData를 사용해 데이터를 공유할 수 있도록 지원할 예정이다.

[코드 2-7]은 SharedData를 사용해 [코드 2-2] TCPChatServerVerticle의 결함을 수정한 내용이다. SharedData를 사용해 채팅 클라이언트의 writeHandlerID를 관리하도록 connectHandler와 closeHandler 부분이 수정되었다. 그리고 클라이언트에 채팅 메시지를 브로드캐스팅할 때 writeHandlerID를 사용하도록 dataHandler도 수정되었다. 앞서 설명한 것처럼 writeHandlerID를 사용하면 클라이언트와 실제 연결점인 NetSocket을 소유하고 있지 않은 Verticle에서도 해당 클라이언트에 메시지를 전송할 수 있다.

[코드 2-7] SharedData 기반 TCP 채팅 서버(TCPChatWithEbServerVerticle.java)

```
public class TCPChatWithEbServerVerticle extends Verticle {
    private Logger logger;
    private EventBus eb;
    private NetServer server;
    private Set<String> sockets;

    @Override
    public void start() {
        logger = container.logger();
```

06 org.vertx.java.core.shareddata.impl.Checker를 참고하면 지원하는 모든 메시지 타입과 메시지 복사 메서드를 확인할 수 있다.

```

eb = vertx.eventBus();
server = vertx.createNetServer();
sockets = vertx.sharedData().getSet("sockets");

server.connectHandler(new Handler<NetSocket>() {
    @Override
    public void handle(final NetSocket socket) {
        sockets.add(socket.writeHandlerID());
        //-- this handler will be called every time data is received on the
        socket
        socket.dataHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                for (String s : sockets) {
                    if (!socket.writeHandlerID().equals(s))
                        eb.send(s, buffer);
                }
            }
        });
        //-- socket closed
        socket.closeHandler(new VoidHandler() {
            @Override
            protected void handle() {
                sockets.remove(socket.writeHandlerID());
            }
        });
        //-- something went wrong
        socket.exceptionHandler(new Handler<Throwable>() {
            @Override
            public void handle(Throwable throwable) {
                logger.error("unexpected exception: ", throwable);
            }
        });
    }
});

```

```

        server.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {
            @Override
            public void handle(AsyncResult<NetServer> asyncResult) {
                logger.info("bind result: "+asyncResult.succeeded());
            }
        });
    }
}

```

다시 TCP 채팅 서버를 실행한다. 이번에도 ‘-instances 2’ 옵션을 줘서 2개의 TCPChatWithEbServerVerticle이 실행되도록 한다. 이어서 3개의 TCPChatClientStarterVerticle을 실행하고 메시지를 전송해 보자. 모든 TCP 채팅 클라이언트가 정상적으로 메시지를 주고받는 것을 확인할 수 있다.

NOTE

SharedData가 단일 Vert.x Instance 내의 Verteclle 사이에서 메시지를 공유하기 위한 장치인 만큼 SharedData를 활용한 NetSocket의 writeHandlerID 공유 방법도 사실 완전한 해결책은 아니다. 단일 Vert.x Instance에서만 채팅 서버를 구동하면 아무 문제 없지만, Vert.x Instance를 클러스터로 구성할 경우 일부 채팅 클라이언트끼리만 메시지를 주고받는 문제가 동일하게 재현될 수 있다. 이런 경우 NetSocket의 writeHandlerID를 MongoDB나 MySQL 같은 스토리지로 관리하는 방법을 사용하거나 NetSocket의 writeHandlerID를 동기화하기 위해 Vert.x Instance끼리 특정 이벤트 버스로 클라이언트 연결 상태를 브로드캐스팅하는 방법을 사용해야 한다. 이 두 가지 외에도 해결 방법은 다양하게 생각해볼 수 있다.

2.4 두 번째 결함의 분석과 해결방법

TCP 채팅 서비스의 두 번째 결함은 TCP 프로토콜에 대한 처리가 올바르지 않아서 발생한다. TCP는 스트림^{stream} 기반 프로토콜이므로 본질적으로 메시지가 어떤 경계에 의해 구분되지 않으며, 단지 바이트 스트림^{Byte Stream}으로 전달된다. 따라서

애플리케이션 레벨에서 스트림을 의미가 있는 프레임^{frame} 단위로 적절하게 재조합하는 작업이 필요하다. 그러나 앞서 작성한 TCP 기반 채팅 서비스는 TCP 프로토콜의 이러한 특징을 전혀 고려하지 않고 구현되었다.

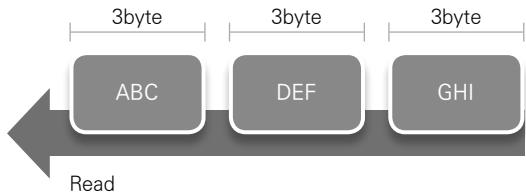
다음과 같은 상황을 생각해 보자. A 호스트와 B 호스트에서 구동되는 애플리케이션 사이에 TCP 연결이 있고, A 호스트의 애플리케이션이 B 호스트 애플리케이션으로 3개의 메시지를 전송하기 위해 `write()` 메서드를 세 번 호출했다. A 호스트에서 B 호스트로 전송되는 3개의 메시지는 서로 개별적인 개체라고 생각하기 쉽다. 그러나 TCP상에서 데이터 전송은 이 3개의 메시지를 개별적인 개체로 구분하지 않는다. 단지 연속된 바이트 스트림으로 처리할 뿐이다. 즉, A 호스트가 ‘ABC’, ‘DEF’, ‘GHI’로 구성된 메시지를 세 번 보냈어도 B 호스트는 ‘AB’, ‘CDE’, ‘FGH’, ‘I’처럼 네 번에 걸쳐 메시지를 수신할 수도 있다는 것이다. 이것은 단지 하나의 예시일 뿐이며, 무수히 많은 형태로 메시지가 수신될 수 있다.

지금까지 예제에서 채팅 메시지가 올바르게 처리되었다 하더라도 네트워크 상황에 따라서 언제든 잘못된 동작을 할 수 있다는 것을 알았다. 따라서 TCP 애플리케이션은 이처럼 예측할 수 없는 조합으로 수신된 메시지를 그대로 사용해서는 안 되고, 반드시 TCP 애플리케이션에서 올바르게 이해하고 처리할 수 있는 형태의 프레임으로 수신된 메시지를 가공한 후 처리해야 한다. 이제부터 TCP 애플리케이션에서 수신된 메시지를 프레임 단위로 재조합하는 방식을 알아보자.

2.4.1 고정길이 분할 방식

수신된 메시지를 항상 일정한 단위로 끊어 읽어서 처리하는 방식이다. 가장 간단한 방법이지만 대부분 TCP 애플리케이션에서는 가변길이 메시지를 사용하므로 활용도가 높지는 않다. 채팅 서비스 역시 사용자 입력 텍스트가 가변적이어서 이 방법을 적용하기에는 적합하지 않다.

[그림 2-3] 고정길이 분할 방식



[코드 2-8] 고정길이 분할 방식(TCPFixedParserServerVerticle.java)

```
public class TCPFixedParserServerVerticle extends Verticle {
    private Logger logger;

    @Override
    public void start() {
        logger = container.logger();
        NetServer netServer = vertx.createNetServer();

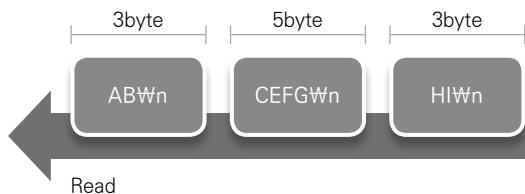
        netServer.connectHandler(new Handler<NetSocket>() {
            @Override
            public void handle(NetSocket socket) {
                socket.dataHandler(RecordParser.newFixed(3, new Handler<Buffer>() {
                    @Override
                    public void handle(Buffer buffer) {
                        logger.info("received data: " + buffer.toString());
                    }
                }));
            }
        });
        netServer.listen(8090, "localhost");
    }
}
```

RecordParser newFixed() 메서드의 첫 번째 파라미터는 메시지를 분할할 고정길이고, 두 번째 파라미터는 메시지를 수신했을 때 실행되는 이벤트 핸들러다.

2.4.2 Delimiter 분할 방식

수신된 메시지를 구분자Delimiter를 기준으로 끊어 읽어서 처리하는 방식이다. 따라서 메시지를 구성하는 데 잘 사용되지 않는 문자를 구분자로 선택하는 것이 중요하다. 구현은 상대적으로 간단한 편이다. 사용자 입력 텍스트의 끝에 개행문자를 추가하면 개행문자를 기준으로 메시지를 끊어 읽는 것이 가능하므로 채팅 서비스에도 적용할 수 있다.

[그림 2-4] Delimiter 분할 방식



[코드 2-9] Delimiter 분할 방식(TCPDelimiterParserServerVerticle.java)

```
public class TCPDelimiterParserServerVerticle extends Verticle {
    private Logger logger;

    @Override
    public void start() {
        logger = container.logger();
        NetServer netServer = vertx.createNetServer();

        netServer.connectHandler(new Handler<NetSocket>() {
            @Override
            public void handle(NetSocket socket) {
                socket.dataHandler(RecordParser.newDelimited("\n", new
                    Handler<Buffer>() {
                        @Override
                        public void handle(Buffer buffer) {
                            logger.info("received data: " + buffer.toString());
                        }
                    })
                );
            }
        });
    }
}
```

```

        }
    });
}
);
netServer.listen(8090, "localhost");
}
}

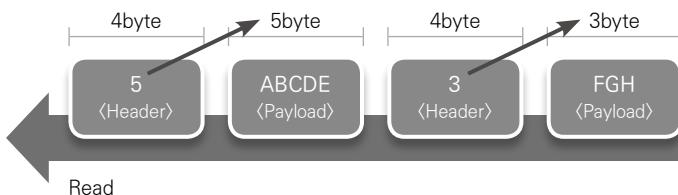
```

RecordParser newDelimited() 메서드의 첫 번째 파라미터는 메시지를 분할하기 위한 구분자고, 두 번째 파라미터는 메시지를 수신했을 때 실행되는 이벤트 핸들러다.

2.4.3 Header/Payload 분할 방식

앞의 두 가지 방법보다 상대적으로 구현이 복잡하지만, 가장 유연하고 폭넓게 사용될 수 있는 방법이다. 발신 측은 고정길이 Header와 가변길이 Payload로 구성된 메시지를 전송하는데, Header는 뒤따라 오는 Payload의 길이를 의미한다. 수신 측은 고정길이 Header를 먼저 읽고 뒤따라오는 Payload의 길이를 정확히 파악하여 해당 길이만큼 Payload를 읽어서 처리하면 된다.

[그림 2-5] Header, Payload 분할 방식



세 가지 해결 방법 중 Vert.x에서 Header/Payload 분할 방식으로 TCP에서 수신한 메시지를 처리하는 방법을 좀 더 자세히 살펴보자.

고정길이 분할 방식 예제에서 사용한 RecordParser의 newFixed() 메서드를

이용해 메시지를 Header와 Payload로 구분하여 처리할 수 있다. 먼저 4바이트(Header의 고정길이로, 반드시 4바이트일 필요는 없다) 고정길이로 메시지를 분할하도록 설정된 RecordParser 객체를 생성한다. 두 번째 파라미터(메시지를 수신했을 실행되는 이벤트 핸들러)는 null로 입력하고, 뒤의 setOutput() 메서드에서 설정한다.

```
final RecordParser framer = RecordParser.newFixed(4, null);
```

다음으로 메시지가 수신될 때 실행될 이벤트 핸들러를 작성하고, setOutput() 메서드로 RecordParser 객체를 설정한다. 눈여겨볼 곳은 fixedSizeMode() 메서드를 이용해 메시지를 분할하는 기준을 4바이트와 payloadLength 바이트로 교차해서 설정하는 부분이다. 이것은 첫 4바이트 Header를 수신하고, Header에 명시된 payloadLength만큼 Payload를 수신하게 한다. 마지막으로 Payload 수신을 완료한 후 다시 4바이트 Header를 수신할 수 있도록 재설정하면 다시 처음으로 돌아가 이 과정을 반복하게 된다.

```
framer.setOutput(new Handler<Buffer>() {
    int payloadLength = -1;
    @Override
    public void handle(Buffer buffer) {
        if ( payloadLength == -1 ) {
            payloadLength = buffer.getInt(0);
            framer.fixedSizeMode(payloadLength);
        }
        else {
            logger.info("received data: "+buffer.toString());
            framer.fixedSizeMode(4);
            payloadLength = -1;
        }
    }
});
```

```
    }
}
});
```

RecordParser 객체는 Handler<Buffer> 인터페이스의 구현체로 사용될 수 있다. 즉, setOutput() 메서드를 통해 설정한 이벤트 핸들러가 NetSocket의 dataHandler에 대응된다.

```
socket.dataHandler(framer);
```

이제 RecordParser를 이용한 Header/Payload 분할 방식을 TCP 채팅 서버와 클라이언트에 적용해 보자. TCP 채팅 서버/클라이언트는 기본으로 [코드 2-7]과 [코드 2-3]을 사용한다.

[코드 2-10] Header/Payload 분할 방식 채팅 서버(TCPChatWithFramingServerVerticle.java)

```
public class TCPChatWithFramingServerVerticle extends Verticle {
    private Logger logger;
    private EventBus eb;
    private NetServer server;
    private Set<String> sockets;

    @Override
    public void start() {
        logger = container.logger();
        eb = vertx.eventBus();
        server = vertx.createNetServer();
        sockets = vertx.sharedData().getSet("sockets");

        server.connectHandler(new Handler<NetSocket>() {
            @Override
            public void handle(final NetSocket socket) {
```

```

        sockets.add(socket.writeHandlerID());
        //-- tcp stream framing
        final RecordParser framer = RecordParser.newFixed(4, null);
        framer.setOutput(new Handler<Buffer>() {
            int payloadLength = -1;
            @Override
            public void handle(Buffer buffer) {
                if (payloadLength == -1) {
                    payloadLength = buffer.getInt(0);
                    framer.fixedSizeMode(payloadLength);
                }
                else {
                    Buffer newbuf = new Buffer(4+payloadLength);
                    newbuf.setInt(0, payloadLength);
                    newbuf.setBuffer(4, buffer);
                    for (String s : sockets) {
                        if (!socket.writeHandlerID().equals(s))
                            eb.send(s, newbuf);
                    }
                    framer.fixedSizeMode(4);
                    payloadLength = -1;
                }
            }
        });
        //-- this handler will be called every time data is received on the
        socket
        socket.dataHandler(framer);
        //-- socket closed
        socket.closeHandler(new VoidHandler() {
            @Override
            protected void handle() {
                sockets.remove(socket.writeHandlerID());
            }
        });
        //-- something went wrong
    }
}

```

```

        socket.exceptionHandler(new Handler<Throwable>() {
            @Override
            public void handle(Throwable throwable) {
                logger.error("unexpected exception: ", throwable);
            }
        });
    }

});

server.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {
    @Override
    public void handle(AsyncResult<NetServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});
}

@Override
public void stop() {
    if (server != null)
        server.close();
}
}

```

[코드 2-11] Header/Payload 분할 방식 채팅 클라이언트(TCPChatWithFramingClientVerticle.java)

```

public class TCPChatWithFramingClientVerticle extends Verticle {
    private Logger logger;
    private EventBus eb;
    private NetClient client;
    private String writeHandlerID;

    @Override
    public void start() {
        logger = container.logger();
        eb = vertx.eventBus();
    }
}

```

```

client = vertx.createNetClient();
client.connect(8090, "localhost", new Handler<AsyncResult<NetSocket>>() {
    @Override
    public void handle(AsyncResult<NetSocket> asyncResult) {
        logger.info("connect result: " + asyncResult.succeeded());
        if (asyncResult.succeeded()) {
            final NetSocket socket = asyncResult.result();
            writeHandlerID = socket.writeHandlerID();
            //-- tcp stream framing
            final RecordParser framer = RecordParser.newBuilder().setFramerType(RecordParser.FramerType.FIXED).build();
            framer.setOutputStream(new Handler<Buffer>() {
                int payloadLength = -1;
                @Override
                public void handle(Buffer buffer) {
                    if (payloadLength == -1) {
                        payloadLength = buffer.getInt(0);
                        framer.fixedSizeMode(payloadLength);
                    }
                    else {
                        logger.info("received data: " + buffer.toString());
                        framer.fixedSizeMode(4);
                        payloadLength = -1;
                    }
                }
            });
            //-- this handler will be called every time data is received on
            the socket
            socket.dataHandler(framer);
            //-- socket closed
            socket.closeHandler(new VoidHandler() {
                @Override
                protected void handle() {
                    logger.info("connection closed:
"+socket.remoteAddress());
                }
            });
        }
    }
});

```

```
        });
    //-- something went wrong
    socket.exceptionHandler(new Handler<Throwable>() {
        @Override
        public void handle(Throwable throwable) {
            logger.error("unexpected exception: ", throwable);
        }
    });
}
});

eb.registerHandler("com.devop.vertx.chat", new Handler<Message<String>>() {
    @Override
    public void handle(Message<String> message) {
        if( writeHandlerID != null ) {
            byte[] data;
            try {
                data = message.body().getBytes("UTF-8");
                Buffer buffer = new Buffer(4+data.length);
                buffer.setInt(0, data.length);
                buffer.setBytes(4, data);
                eb.send(writeHandlerID, buffer);
            } catch (UnsupportedEncodingException e) {
            }
        }
    });
}

@Override
public void stop() {
    if (client != null)
        client.close();
}
```

두 예제 모두 RecordParser 객체를 사용하여 TCP 스트림을 적절한 프레임으로 변환해서 처리하도록 메시지 수신 부분이 수정되었으며, 메시지 전송 부분에서는 메시지 길이를 담고 있는 4바이트 Header를 추가하도록 수정되었다.

```
//-- 채팅 서버의 전송 부분
Buffer newbuf = new Buffer(4+payloadLength);
newbuf.setInt(0, payloadLength);
newbuf.setBuffer(4, buffer);

//-- 채팅 클라이언트의 전송 부분. 메시지를 UTF8 인코딩처리 하는 것에 주의한다.
data = message.body().getBytes("UTF-8");
Buffer buffer = new Buffer(4+data.length);
buffer.setInt(0, data.length);
buffer.setBytes(4, data);
```

최종 확인을 위해 TCP 채팅 서버와 클라이언트를 실행해 보자. 서버 구동 시 ‘-instances 2’ 옵션을 줘서 2개의 TCPChatWithFramingServer Verticle이 실행되도록 한다. 채팅 클라이언트를 실행할 때는 TCPChatClient StarterVerticle에서 기존의 TCPChatClientVerticle 실행 부분은 주석 처리하고, 새로운 TCPChatWithFramingClientVerticle를 실행하도록 수정하는 것을 잊지 말자.

3개의 채팅 클라이언트를 실행해 보면 모든 채팅 클라이언트가 정상적으로 메시지를 주고받는 것을 확인할 수 있다. 앞의 예제 실행 결과와 차이는 없지만, 내부적으로 Header/Payload 분할 방식에 따라 TCP 스트림을 적절한 프레임으로 나누고 처리한다.

2.5 요약

지금까지 Vert.x에서 TCP 기반 서버/클라이언트 개발을 편리하게 할 수 있도록 제공하는 NetServer, NetClient, NetSocket에 대한 기본적인 응용 방법과 TCP 기반 서비스 개발 시 주의할 몇 가지 사항을 알아보았다. 특히 TCP 클라이언트에서 이벤트 버스에 관한 설명은 Vert.x 기반 애플리케이션 개발에 폭넓게 사용되는 내용이므로 반드시 이해하고 넘어가자.

3장에서는 SockJS와 SockJS Event Bus Bridge 기반 웹 채팅 서비스를 개발하는데 필요한 내용을 알아보겠다.

3 | SockJS 채팅 서버/클라이언트

Vert.x에서는 HTTP 기반 웹 서버와 클라이언트, SockJS 기반 애플리케이션을 편리하게 개발할 수 있도록 다양한 기능을 제공한다(SockJS 애플리케이션 개발은 WebSocket을 기반으로 하는 애플리케이션 개발과 매우 유사하다). 지금부터 이러한 기능을 활용해 SockJS 기반 채팅 서비스 개발 방법을 알아보자.⁰¹

NOTE_SockJS와 WebSocket

HTML5에 포함된 WebSocket은 웹 브라우저와 웹 서버 간의 양방향 통신을 지원하기 위한 차세대 표준으로, 웹 애플리케이션의 반응성을 극적으로 향상시켜 기존의 Ajax에 기반을 둔 대화형 웹 애플리케이션을 뛰어넘는 훨씬 효율적이고 역동적인 웹 애플리케이션 개발을 가능케 한다. Gmail, Google Docs, Facebook, 웹 채팅 서비스 등이 바로 이에 해당하는 예로, 이러한 종류의 웹 애플리케이션은 사용자가 필요한 정보를 효과적이고 빠르게 전달해 준다는 공통점이 있다.

그러나 WebSocket을 모든 웹 브라우저에서 사용할 수 있는 것은 아니다. 2009년 WebSocket API를 처음 공개한 후 2011년 표준화⁰²를 거쳐 지금에 이르렀는데, 표준 스펙이 비교적 최근에 제정된 만큼 IE10 이전 버전 등 노후 된 웹 브라우저에서는 WebSocket을 지원하지 않는다.⁰³

SockJS는 일종의 WebSocket 에뮬레이터^{Emulator}로, WebSocket 사용이 불가능한 웹 브라우저에서도 WebSocket 표준 API와 유사하면서 일관적인 API를 사용해 애플리케이션 개발을 가능하게 하는 JavaScript 라이브러리다. SockJS는 웹 브라우저에서 WebSocket 사용이 불가능하면 그것을 대체할 용도로 xhr-streaming, xhr-polling, jsonp-polling 등 구동 가능한 코멘^{Comet} 구현⁰⁴을 사용한다. SockJS에 관한 더 자세한 정보는 공식 웹 사이트⁰⁵에서 얻을 수 있다.

01 Vert.x에서 HTTP 클라이언트를 다루기 위한 HttpClient는 본문에서 다루지 않는다. 관련 내용은 공식 매뉴얼을 참고하기 바란다.

02 RFC6455 : <http://tools.ietf.org/html/rfc6455>

03 웹 브라우저 별 WebSocket 지원 현황 : <http://caniuse.com/websockets>

04 http://www.uengine.org:8088/wiki/index.php/Comet_구현_기법

05 <https://github.com/sockjs/sockjs-client>

3.1 HTTP 서버

먼저 SockJS의 기반이 되는 HTTP 서버의 구성 방법을 알아보자.

HTTP 서버를 구성하려면 Verticle 내에 HttpServer 객체를 생성하고, 여기에 몇 가지 이벤트 핸들러를 등록하기만 하면 된다. HttpServer 객체를 초기화하고, 소유하는 Verticle이 할당된 이벤트 루프 스레드 내에서만 HttpServer에 등록된 이벤트 핸들러가 실행되며, HttpServer 객체 자체는 스레드 안전한 특징을 지닌다.

간단한 HTTP 서버를 작성해 보자.

[코드 3-1] HTTP 서버(HTTPServerVerticle.java)

```
public class HTTPServerVerticle extends Verticle {
    private Logger logger;

    @Override
    public void start() {
        logger = container.logger();
        HttpServer httpServer = vertx.createHttpServer();

        //-- When a request arrives, the request handler is called passing in
        //an instance of HttpServerRequest.
        //-- The handler is called when the headers of the request have been
        //fully read.
        httpServer.requestHandler(new Handler<HttpServerRequest>() {
            @Override
            public void handle(HttpServerRequest request) {
                String method = request.method();
                String uri = request.uri();
                String path = request.path();
                String query = request.query();
                logger.info("received http request: {method=" + method + ",",

```

```

uri="+uri+", path="+path+", query="+query+"});

    //-- Http request params
    List<Map.Entry<String, String>> params =
request.params().entries();
    for(Map.Entry<String, String> param : params) {
        logger.info("param["+param.getKey()+"] = "+param.getValue());
    }
    //-- Http request headers
    List<Map.Entry<String, String>> headers =
request.headers().entries();
    for(Map.Entry<String, String> header : headers) {
        logger.info("header["+header.getKey()+"] =
"+header.getValue());
    }
    //-- The body handler is called only once when the entire
request body has been read.
    request.bodyHandler(new Handler<Buffer>() {
        @Override
        public void handle(Buffer buffer) {
            logger.info("received data: " + buffer.toString());
        }
    });

    request.response().setStatusCode(200).end("OK");
}
});

httpServer.listen(8080, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});
}
}

```

콘솔을 열고 `HTTPServerVerticle`을 실행한 다음 웹 브라우저에 `http://localhost:8080/a/b/c/index?p1=1&p2=2`를 입력해 보자. HTTP 요청에 대한 분석 결과가 콘솔에 출력될 것이다(웹 브라우저의 종류에 따라 콘솔에 출력되는 로그 형태는 다를 수 있다). [코드 3-1]을 참고하여 Vert.x에서 HTTP 서버를 다루는 데 필요한 내용을 하나씩 알아보자.

우선 2장의 `NetServer`와 마찬가지로 `Vert.x` 애플리케이션 Runtime 객체인 `vertx`로부터 `HttpServer` 객체를 생성한다.

```
HttpServer httpServer = vertx.createHttpServer();
```

그리고 HTTP 요청을 수신했을 때 호출되는 `requestHandler`를 `HttpServer` 객체에 등록한다. `requestHandler`에 전달되는 `HttpServerRequest` 객체는 HTTP 서버 측에서 HTTP 요청을 처리하는 데 필요한 정보를 가지고 있으며([표 3-1] 참고) HTTP 데이터를 처리하기 위한 추가 이벤트 핸들러를 등록할 수 있는 기능을 제공한다.

[표 3-1] `HttpServerRequest`의 주요 메서드

메서드	설명
<code>method()</code>	GET, POST, PUT, DELETE와 같은 HTTP 요청 메서드를 확인한다.
<code>uri()</code>	'/a/b/c/index?p1=1&p2=2'와 같은 HTTP 요청 URI를 얻는다.
<code>path()</code>	'/a/b/c/index'와 같이 HTTP 요청 URI에서 쿼리 부분을 제외한 나머지 부분을 얻는다.
<code>query()</code>	'p1=1&p2=2'와 같이 HTTP 요청 URI에서 쿼리 부분을 얻는다.
<code>params()</code>	HTTP 요청 URI의 쿼리 부분을 Key, Value 형태(MultiMap)로 얻는다.
<code>headers()</code>	HTTP 요청 헤더를 Key, Value 형태(MultiMap)로 얻는다.
<code>response()</code>	클라이언트로 응답을 처리하기 위한 <code>HttpServerResponse</code> 객체를 얻는다.

`requestHandler`를 등록할 때 `requestHandler`가 호출되는 시점은 HTTP 헤더를 완전히 수신했을 때라는 것을 유의해야 한다. 즉, HTTP 요청이 데이터 부분(HTTP

body)을 포함하고 있다면 requestHandler가 호출된 후 데이터를 처리해야 한다.

다음 코드에서 dataHandler, endHandler, bodyHandler는 HTTP 요청이 데이터를 포함하고 있을 때 데이터 부분을 처리하기 위해 HttpServerRequest 객체가 제공하는 이벤트 핸들러다.

```
final Buffer body = new Buffer(0);

//-- This will then get called every time a chunk of the request body arrives.
//-- The dataHandler may be called more than once depending on the size of the body.
request.dataHandler(new Handler<Buffer>() {
    @Override
    public void handle(Buffer buffer) {
        logger.info("received data: " + buffer.toString());
        body.appendBuffer(buffer);
    }
});

//-- The entire body has now been received
request.endHandler(new VoidHandler() {
    @Override
    protected void handle() {
        logger.info("total received data: " + body.toString());
    }
});
```

dataHandler는 HTTP 요청의 데이터 chunk를 수신할 때마다 호출되는 이벤트 핸들러인데, 데이터의 크기에 따라 호출 한 번에 모든 데이터를 수신할 수도 있고, 두 번 이상의 호출로 데이터를 수신할 수도 있다. 물론 대부분은 HTTP 데이터가 그리 크지는 않아서 한 번의 dataHandler 호출로 모든 데이터를 처리할 수 있다고 가정하지만, 정확하게 처리하려면 모든 HTTP 데이터를 수신했을 때 호출되는 endHandler를 등록하고 사용해야 한다.

그러나 `dataHandler`와 `endHandler`를 동시에 등록하기에는 번거로움이 있다. 그래서 [코드 3-1]처럼 `bodyHandler`를 사용한다. `bodyHandler`는 모든 HTTP 데이터가 수신되었을 때 호출되는 이벤트 핸들러로, `dataHandler`와 `endHandler`의 기능을 합친 것과 같다.

HTTP 서버에서 클라이언트로의 응답은 `response()` 메서드로 얻을 수 있는 `HttpServerResponse` 객체를 통해 처리한다. [코드 3-1]에서는 간단하게 HTTP 응답 코드와 본문 메시지를 전달하지만, HTTP 응답 헤더, 상태 메시지, 파일 전송 등 다양한 방법으로도 클라이언트로의 응답을 생성할 수 있다.

[표 3-2] `HttpServerResponse`의 주요 메서드

메서드	설명
<code>setStatuscode()</code>	HTTP 응답 코드를 설정한다.
<code>headers(), putHeader()</code>	HTTP 응답 헤더를 얻어오고 설정한다.
<code>write(), end()</code>	HTTP 응답 메시지를 설정한다.
<code>sendFile()</code>	하드 디스크에 존재하는 파일을 클라이언트로 전송한다.
<code>close()</code>	HTTP 연결을 종료한다.

3.2 SockJS

Vert.x에서는 HTML5의 WebSocket을 사용하기 위한 두 가지 방법을 제공하는데, 두 가지 방법 모두 앞서 살펴본 `HttpServer`를 확장하는 방식으로 사용한다.

첫 번째는 HTML5의 WebSocket을 직접 사용하는 방법으로, 간단히 `HttpServer`에 몇 개의 이벤트 핸들러만 추가하면 된다. 이는 전체적으로 TCP 기반 서버를 작성하기 위해 `NetServer`, `NetSocket`을 사용하는 방법과 매우 유사하다. 마찬가지로 SockJS 기반 개발 방법도 `NetServer`, `NetSocket`을 사용하는 방법과 매우 유사하다. 이것은 전체적으로 Vert.x API가 단순하고 일관적이며 사용하기 쉽다는 것을 의미한다.

```
HttpServer httpServer = vertx.createHttpServer();
httpServer.websocketHandler(new Handler<ServerWebSocket>() {
    @Override
    public void handle(ServerWebSocket webSocket) {
        logger.info("A WebSocket has connected!");
        webSocket.dataHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                logger.info("received data: " + buffer.toString());
            }
        });
    }
});
```

그러나 아직은 모든 웹 브라우저에서 WebSocket을 사용할 수 있는 것은 아니어서 WebSocket을 직접 사용하는 대신 WebSocket의 에뮬레이터 중 하나인 SockJS를 사용해 웹 브라우저의 WebSocket 지원 여부와 관계없이 모두 일관된 API로 클라이언트 코드를 작성할 수 있는 두 번째 방법을 추천한다.⁰⁶

SockJS는 WebSocket 사용이 불가능한 웹 브라우저에서도 WebSocket 표준 API와 유사하고 일관적인 API를 사용해 애플리케이션 개발을 가능하게 하는 JavaScript 라이브러리다. SockJS를 지원하는 서버 사이드 솔루션에는 최근 발표된 Spring 4와 Vert.x가 대표적이다. 현재 구글의 Go, JBOSS Netty, Perl, Ruby 등을 지원하기 위한 작업이 진행되고 있으므로⁰⁷ 앞으로 그 활용성이 더 다양해질 전망이다.

그리고 Vert.x에서 SockJS를 사용하면 SockJS Event Bus Bridge라는 특별한 기능을 활용해 클라이언트 사이드의 웹 브라우저를 이벤트 버스에 참여시킴으로써 대규모 클러스터를 구성할 수 있다는 장점이 있다.

06 잘 알려진 WebSocket 에뮬레이터로는 SockJS 외에 Node.js의 Socket.io가 있다.

07 <https://github.com/sockjs/sockjs-client>

3.2.1 SockJS 서버

간단한 SockJS 서버를 작성해 보자.

[코드 3-2] SockJS 애코 서버(SockJSServerVerticle.java)

```
public class SockJSServerVerticle extends Verticle {
    private Logger logger;

    @Override
    public void start() {
        logger = container.logger();
        HttpServer httpServer = vertx.createHttpServer();

        httpServer.requestHandler(new Handler<HttpServerRequest>() {
            @Override
            public void handle(HttpServerRequest request) {
                String method = request.method();
                String uri = request.uri();
                String path = request.path();
                String query = request.query();
                logger.info("received http request: {method=" + method +
                           ", uri=" + uri + ", path=" + path + ", query=" + query + "}");

                request.bodyHandler(new Handler<Buffer>() {
                    @Override
                    public void handle(Buffer buffer) {
                        logger.info("received data: " + buffer.toString());
                    }
                });
                request.response().setStatusCode(200).end("OK");
            }
        });
        SockJSServer sockJSServer = vertx.createSockJSServer(httpServer);

        JsonObject config = new JsonObject();
        config.putString("prefix", "/mySockJS");
    }
}
```

```

//-- this handler will be called when new SockJS sockets are created.
sockJSServer.installApp(config, new Handler<SockJSSocket>() {
    @Override
    public void handle(final SockJSSocket sockJSSocket) {
        //-- this handler will be called every time data is received on
        the sockJSSocket
        sockJSSocket.dataHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                logger.info("received data: " + buffer.toString());
                sockJSSocket.write(buffer);
            }
        });
        //-- something went wrong
        sockJSSocket.exceptionHandler(new Handler<Throwable>() {
            @Override
            public void handle(Throwable throwable) {
                logger.error("unexpected exception: ", throwable);
            }
        });
    }
});

httpServer.listen(8080, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});
}
}

```

콘솔을 열고 SockJSServerVerticle을 실행한 후, 웹 브라우저에 <http://localhost:8080/mySockJS>를 입력해 보자. 웹 브라우저에 ‘Welcome to

SockJS!'라는 메시지가 출력되는 것을 확인할 수 있을 것이다(그 외의 URL에서는 requestHandler에 의해 'OK' 문자가 출력된다).

[코드 3-2]에서 확인할 수 있듯이 SockJS 서버는 HttpServer를 확장해서 만든다. 물론 HttpServer도 이후에 계속 사용할 수 있지만, 한 가지 주의해야 할 점 있다. SockJSServer 객체를 생성하기 전에 반드시 HttpServer 객체의 requestHandler를 등록해야 한다. 그렇지 않으면 SockJS 클라이언트 접속 URL이 일반 HTTP 요청으로 처리되어 다음 같은 경고 메시지가 출력된다.

You have overwritten the Http server request handler AFTER the SockJSServer has been created which will stop the SockJSServer from functioning. **Make sure you set http request handler BEFORE you create the SockJSServer**

SockJSServer 객체를 생성한 다음 초기화를 위해 installApp() 메서드를 호출하는데, 첫 번째 파라미터는 SockJSServer의 설정 정보를 담고 있는 JSON 객체고([표 3-3] 참고), 두 번째 파라미터는 새로운 SockJSSocket이 생성될 때 호출되는 이벤트 핸들러다.

[표 3-3] SockJSServer 주요 설정 파라미터⁰⁸

파라미터	설명
prefix	SockJS 애플리케이션의 진입 URL을 정의한다. SockJS 클라이언트는 해당 URL로 연결 요청을 보낸다. 이는 필수 입력 항목이다.
insert_JSESSIONID	true면 JSESSIONID cookie set에 기반하여 Sticky Session을 활성화한다. 기본값은 true다.

08 모든 설정 파라미터와 그 기본값은 org.vertx.java.core.sockjs.impl.DefaultSockJSServer의 setDefaults() 메서드에서 확인할 수 있다.

09 <https://github.com/sockjs/sockjs-protocol/wiki/Heartbeats-and-SockJS>

파라미터	설명
heartbeat_period	SockJS의 서버/클라이언트 간의 Heartbeat 전송주기를 입력한다. 전송주기가 너무 길어지면 서버/클라이언트 연결의 중간에 위치하는 로드 밸런서 등의 네트워크 장비가 연결을 종료시킬 수 있다. 기본값은 2500(25초)이다. SockJS의 Heartbeat에 대한 자세한 설명은 SockJS Wiki 페이지⁹⁹ 에서 확인할 수 있다.
library_url	SockJS는 Cross Domain 문제를 해결하기 위해 iframe trick을 사용한다. 그 결과로 HTML DOM 내부에 invisible iframe이 생성되는데, invisible iframe에서도 SockJS JavaScript 라이브러리가 필요하다. 기본값은 http://cdn.sockjs.org/sockjs-0.3.4.min.js 이며, 최신의 SockJS JavaScript 라이브러리 경로를 입력하면 된다.

이벤트 핸들러에 전달되는 SockJSSocket은 SockJS 클라이언트와의 실제 연결점을 의미한다. 메시지 수신을 처리하기 위한 `dataHandler`, 예기치 못한 오류가 발생했을 때 호출되는 `exceptionHandler` 등 전체적으로 NetSocket과 거의 비슷한 방법으로 사용할 수 있다. 데이터를 전송하는 방법 역시 입력 파라미터로 스트링을 사용할 수 없다는 것을 제외하면 NetSocket과 차이가 없다.

```
sockJSSocket.write(new Buffer("hello, world"));
```

SockJSServer의 신규 클라이언트 연결과 종료 처리

SockJSServer를 처음 사용할 때 가장 난감한 부분은 신규 클라이언트 연결/종료 이벤트를 처리하는 인터페이스를 찾을 수 없다는 것이다. 이것은 SockJSServer와 SockJSSocket의 사용법이 앞서 살펴본 NetServer와 NetSocket의 사용법과 유사하므로 신규 클라이언트 연결을 처리하기 위한 NetServer의 `connectHandler`, 클라이언트 연결 종료 처리를 위한 NetSocket의 `closeHandler`와 비슷한 이벤트 핸들러를 SockJSServer와 SockJSSocket에서 찾기 때문이다. 하지만 안타깝게도 SockJSServer와 SockJSSocket에서는 `connectHandler`와 `closeHandler`를 발견할 수 없다.

물론 그렇다고 SockJSServer와 SockJSSocket에서 신규 클라이언트 연결/종료 이벤트를 처리할 수 없는 것은 아니다. 다만 이를 처리하기 위한 이벤트 핸들러가 NetServer와 NetSocket에서 제공하는 것처럼 명시적이고 직관적인 이름을 지니고 있지 않을 뿐이다.

신규 클라이언트 연결은 SockJSServer의 installApp() 메서드의 두 번째 파라미터로 입력하는 이벤트 핸들러에서 처리할 수 있다. 이 이벤트 핸들러는 새로운 SockJSSocket이 생성될 때 호출되며, connectHandler처럼 클라이언트 연결 정보 저장 등의 처리를 하기에 적합하다.

클라이언트 연결 종료는 SockJSSocket의 endHandler로 처리한다. 사실 endHandler는 더는 처리할 데이터가 없을 때 호출되는 이벤트 핸들러로, closeHandler와는 의미상 다소 차이가 있지만, SockJSSocket에서 endHandler가 NetSocket의 closeHandler와 같은 역할로 쓰이는 것은 분명하다. 이것은 SockJS Event Bus Bridge에서 사용되는 EventBusBridgeHook의 handleSocketClosed() 메서드를 구현하는 데 사용되기도 한다(handleSocketClosed() 메서드는 SockJS Event Bus Bridge에서 클라이언트의 연결이 종료되었을 때 호출되는 이벤트 핸들러로, 지금 설명하는 내용과 일치한다).

[코드 3-3]은 SockJS Event Bus Bridge를 사용할 때 내부에서 자동으로 설정되는 이벤트 핸들러 구현 코드로, 신규 클라이언트 연결을 처리하는 handleSoketCreated() 메서드와 클라이언트 연결 종료를 처리하는 handleSocketClosed() 메서드 호출을 확인할 수 있다.

[코드 3-3] org.vertx.java.core.sockjs.EventBusBridge의 handle() 메서드 구현

```
public void handle(final SockJSSocket sock) {
    if (!handleSocketCreated(sock)) {
        sock.close();
    }
}
```

```

    } else {
        final Map<String, Handler<Message>> handlers = new HashMap<>();
        sock.endHandler(new VoidHandler() {
            public void handle() {
                handleSocketClosed(sock, handlers);
            }
        });
        sock.dataHandler(new Handler<Buffer>() {
            public void handle(Buffer data) {
                handleSocketData(sock, data, handlers);
            }
        });
    });

    // Start a checker to check for pings
    final PingInfo pingInfo = new PingInfo();
    pingInfo.timerID = vertx.setPeriodic(pingTimeout, new Handler<Long>() {
        @Override
        public void handle(Long id) {
            if (System.currentTimeMillis() - pingInfo.lastPing >=
                pingTimeout) {
                // We didn't receive a ping in time so close the socket
                sock.close();
            }
        }
    });
    SockInfo sockInfo = new SockInfo();
    sockInfo.pingInfo = pingInfo;
    sockInfos.put(sock, sockInfo);
}
}

```

3.2.2 SockJS 클라이언트

[코드 3-2]의 SockJSServerVerticle은 SockJSSocket을 통해 클라이언트가 전송한 메시지를 그대로 돌려보내는 간단한 에코 Echo 서비스다. 이를 확인하기 위해 간단한 SockJS 클라이언트를 작성해 보자.¹⁰

[코드 3-4] SockJS 에코 클라이언트(SockJS.html)

```
<html>
<head>
<title>SockJS Test</title>
<script src="http://cdn.sockjs.org/sockjs-0.3.4.min.js"></script>
<script type="text/javascript">
    var sock = new SockJS('http://localhost:8080/mySockJS');

    sock.onopen = function() {
        console.log('open');
    };
    sock.onheartbeat = function() {
        console.log('heartbeat');
    };
    sock.onmessage = function(e) {
        console.log('message', e.data);
        alert('received message echoed from server: ' + e.data);
    };
    sock.onclose = function() {
        console.log('close');
    };

    function send(message) {
        if (sock.readyState === SockJS.OPEN) {
            console.log('sending message')
            sock.send(message);
        }
    }
</script>

```

¹⁰ 웹 브라우저에서 출력되는 'Welcome to SockJS!'라는 메시지는 단순히 SockJS 서버가 정상 동작하고 있음을 확인하기 위한 내용이고, SockJS 클라이언트를 작성해 해당 URL로 SockJS 연결을 생성해야 한다.

```

        }
    else {
        console.log('The socket is not open.');
    }
}

</script>
</head>
<body>
<form onsubmit="return false;">
    <input type="text" name="message" value="Hello, World!"/>
    <input type="button" value="Send SockJS data" onclick="send(this.form.message.value)"/>
</form>
</body>
</html>

```

SockJS JavaScript 라이브러리를 포함하고 연결 URL을 입력 파라미터로 SockJS 객체를 생성한다(앞서 SockJSServer의 설정 파라미터 prefix 값으로 '/mySockJS'를 입력했다). SockJS 객체를 생성하면 간단하게 몇 개의 핸들러를 등록하고 바로 사용할 수 있을 정도로 사용법이 간단하다.¹¹

[표 3-4] SockJS 자바스크립트 핸들러

핸들러	설명
onopen	서버와 SockJS 연결이 완료되었을 때 호출된다.
onheartbeat	서버로부터 SockJS Heartbeat 패킷을 수신했을 때 호출된다. Heartbeat 전송주기는 SockJSServer 설정 파라미터인 heartbeat_period다.
onmessage	서버로부터 메시지를 수신했을 때 호출된다.
onclose	서버와 SockJS 연결이 종료되었을 때 호출된다.

SockJSServerVerticle을 실행한 후 웹 브라우저에서 SockJS.html 파일을 열고

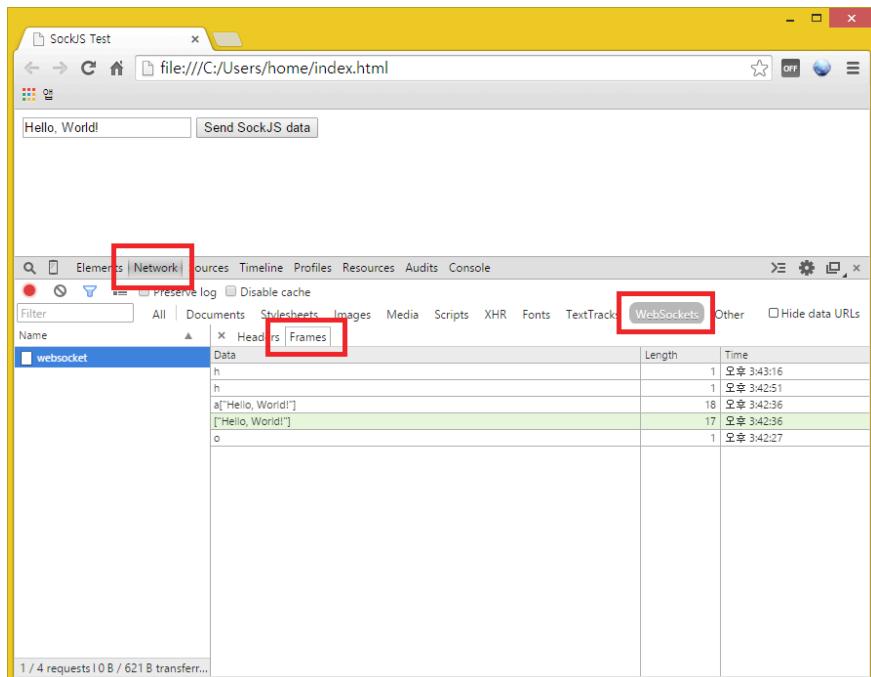
¹¹ SockJS API는 HTML5 WebSocket 표준 API와 가능한 유사하도록 구현되어 있다. HTML5 WebSocket 표준 API는 다음 링크에서 확인할 수 있다. <http://dev.w3.org/html5/websockets/#the-websocket-interface>

웹 브라우저 콘솔 로그를 확인해 보자. [그림 3-1]처럼 SockJS 연결이 정상적으로 처리되면 ‘open’ 메시지 이후 일정 주기로 ‘heartbeat’ 메시지가 출력되고, 웹 브라우저의 Send SockJS Data 버튼을 클릭하면 서버로 전송된 메시지가 그대로 에코되어 돌아와서 출력되는 것을 확인할 수 있다.

NOTE

웹 브라우저로 크롬을 사용한다면 WebSocket을 통한 프레임 흐름을 모니터링할 수 있다. 먼저 F12 버튼을 눌러 개발자 도구를 활성화하고, 상단의 Network 탭을 선택한 다음 하위 탭에서 WebSocket을 선택한다. 그다음 Headers 옆의 Frames을 선택하면 [그림 3-1]과 같은 화면을 볼 수 있다.

[그림 3-1] 크롬에서 WebSocket 프레임 흐름 모니터링하기



[그림 3-1]의 오른쪽 아랫부분을 보면 WebSocket을 통한 프레임 흐름을 확인할 수 있다. ‘o’는 SockJS Open, ‘h’는 SockJS Heartbeat, ‘[“문자열”]’은 클라이언트가 서버로 전송한 데이터, ‘a[“문자열”]’는 서버가 클라이언트로 전송한 데이터를 의미한다. 이와 같은 WebSocket 프레임 흐름을 통해 [코드 3-2] SockJS 에코 서버와 [코드 3-4] SockJS 에코 클라이언트 모두 정상적으로 동작하는 것을 확인할 수 있다.

3.3 SockJS Event Bus Bridge

3.2 SockJS에서 소개한 SockJS 서버/클라이언트를 응용하면 채팅 서비스를 포함한 웹만한 웹 기반 실시간 애플리케이션은 충분히 개발할 수 있다. 그러나 Vert.x에서는 여기서 더 나아가 SockJS Event Bus Bridge(이하 SockJS EBB라 한다)라는 특별한 기능을 제공한다. SockJS EBB는 SockJS를 기반으로 Vert.x에서 제공하는 이벤트 버스를 웹 브라우저에서도 사용할 수 있도록 확장하는 기술을 의미한다.¹²

NetSocket의 writeHandlerID를 사용해 실제 클라이언트와 연결된 WebSocket을 소유하지 않은 Verticle에서도 이벤트 버스를 통해 해당 클라이언트로 메시지를 전송할 수 있었던 것을 떠올려 보자. 이러한 특징은 웹 브라우저로 확장된 이벤트 버스에서도 여전히 유효한데, 이것을 잘만 응용하면 웹 브라우저를 포함하는 거대한 Vert.x 클러스터를 구성하고, 그 어떤 복잡한 메시지 송수신 처리도 효율적으로 구현할 수 있다.

그럼 이제부터 SockJS EBB 채팅 서버와 클라이언트를 작성해 보면서 그 사용법과 특징을 자세히 알아보자.

12 이벤트 버스는 클러스터로 묶인 Vert.x Instance 또는 단일 Vert.x Instance 내의 Verticle들이 서로 메시지를 주고받기 위해 사용한다

3.3.1 SockJS EBB 채팅 서버

SockJSServer에 몇 가지 설정을 추가하면 간단하게 SockJS EBB 기능을 사용할 수 있다. [코드 3-2]의 SockJS 기반 예코 서비스와 비교해 보면 setHook() 메서드로 EventBusBridgeHook 구현체¹³를 설정하고, installApp() 메서드 대신 bridge() 메서드를 호출하도록 수정하는 것이 전부다.

[코드 3-5] SockJS EBB 채팅 서버(SockJSEbbChatServerVerticle.java)

```
public class SockJSEbbChatServerVerticle extends Verticle {
    private Logger logger;
    private HttpServer httpServer;
    private int port;
    private String origin;

    @Override
    public void start() {
        JsonObject appConfig = container.config();

        logger = container.logger();
        httpServer = vertx.createHttpServer();
        port = appConfig.getInteger("port", 80);
        origin = appConfig.getString("origin", "http://localhost");

        httpServer.requestHandler(new Handler<HttpServerRequest>() {
            @Override
            public void handle(HttpServerRequest request) {
                if (request.path().equals("/")) request.response().sendFile("index.html");
                else if (request.path().endsWith("vertxbus-2.1.js")) request.response().sendFile("vertxbus-2.1.js");
                else request.response().setStatusCode(404).end("Not Found Page");
            }
        });
    }
}
```

13 EventBusBridgeHook 구현체는 handlerSocketCreated, handleSocketClosed와 같은 유용한 메서드를 포함한다.

```

SockJSServer sockJSserver = vertx.createSockJSserver(httpServer);

//-- bridge hook
sockJSserver.setHook(new EventBusBridgeHook() {
    @Override
    public boolean handleSocketCreated(SockJSocket sock) {
        if (origin!=null) {
            String originHeader = sock.headers().get("origin");
            if (originHeader == null || !originHeader.equals(origin)) {
                return false; //-- reject the socket
            }
        }
        return true; //-- true to accept the socket, false to reject it
    }

    @Override
    public void handleSocketClosed(SockJSocket sock) {
        logger.info("handleSocketClosed, sock = " + sock);
    }

    @Override
    public boolean handleSendOrPub(SockJSocket sock, boolean send,
JsonObject msg, String address) {
        logger.info("handleSendOrPub, sock = " + sock + ", send = " +
send + ", address = " + address);
        return true; //-- true To allow the send/publish to occur,
false otherwise
    }

    @Override
    public boolean handlePreRegister(SockJSocket sock, String address) {
        logger.info("handlePreRegister, sock = " + sock + ", address =
" + address);
        return true; //-- true to let the registration occur, false
otherwise
    }
}

```

```

    @Override
    public void handlePostRegister(SockJSocket sock, String address) {
        logger.info("handlePostRegister, sock = " + sock + ", address =
" + address);
    }

    @Override
    public boolean handleUnregister(SockJSocket sock, String address) {
        logger.info("handleUnregister, sock = " + sock + ", address =
" + address);
        return true;
    }

    @Override
    public boolean handleAuthorise(JsonObject message, String
sessionID, Handler<AsyncResult<Boolean>> handler) {
        return false; //-- true if you wish to override authorisation
    }
});

JsonObject sockJSconfig = new JsonObject();
sockJSconfig.putString("prefix", appConfig.getString("prefix",
"/mySockJS"));

//-- bridge perms(allow everytings)
JsonArray permitted = new JsonArray();
permitted.add(new JsonObject());

sockJSServer.bridge(sockJSconfig, permitted, permitted);

httpServer.listen(port, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});
}

```

```
    @Override
    public void stop() {
        if (httpServer != null)
            httpServer.close();
    }
}
```

먼저 SockJS EBB 기능을 활성화하는 SockJSServer의 bridge() 메서드를 살펴보자.

```
public SockJSServer bridge(JsonObject sjsConfig, JSONArray inboundPermitted,
JSONArray outboundPermitted) {
    EventBusBridge busBridge = new EventBusBridge(vertx, inboundPermitted,
outboundPermitted);
    (hook != null) {
        busBridge.setHook(hook);
    }
    installApp(sjsConfig, busBridge);
    return this;
}
```

bridge() 메서드는 내부적으로 SockJSServer 객체 초기화를 위해 installApp() 메서드를 호출하는데, 첫 번째 파라미터는 bridge() 메서드에서 입력받은 첫 번째 파라미터(SockJS 애플리케이션의 진입 URL을 정의하고 있다. [표 3-3] 참고)를 그대로 넘기고 두 번째 파라미터는 org.vertx.java.core.sockjs.EventBusBridge라는 특수한 이벤트 핸들러를 입력한다. 이 이벤트 핸들러는 setHook() 메서드를 통해 설정된 EventBusBridgeHook 구현체의 메서드를 상황에 맞게 적절히 호출하고, 추가로 클라이언트가 일정주기로 전송하는 ping을 통해 연결 상태를 확인하는 역할을 한다([코드 3-3]은 EventBusBridge 구현 코드 일부를 보여준다).

bridge() 메서드를 호출하기 위한 두 번째, 세 번째 파라미터는 클라이언트 보안 설정과 관련된 내용으로 [코드 3-5]에서는 보안 처리 관련 아무런 제약을 두지 않았다. 이는 3.4 SockJS EBB 보안 설정에서 자세히 알아본다.

SockJS EBB 채팅 서버는 이것이 전부다. 앞서 TCP 기반 채팅 서버와 SockJS 기반 애코 서버처럼 `NetSocket`과 `SockJSSocket` 같은 클라이언트 정보를 별도로 관리하지 않는다는 것에 주목하자. 대신 모든 클라이언트는 자체적으로 특정 주소를 통해 이벤트 버스에 연결할 수 있으며, 이 주소로 메시지를 전송하거나 수신할 수 있다.

`config.json`이라는 간단한 설정파일을 다음과 같이 작성하고, `SockJSEbb ChatServerVerticle`을 실행해 보자.¹⁴ 설정파일에는 HTTP 서버 포트, SockJS EBB 클라이언트를 출처를 검증하기 위한 `origin`, SockJS 애플리케이션의 진입 URL인 `prefix`가 정의되어 있다. 설정파일의 내용은 `container` 객체의 `config()` 메서드를 이용해 JSON 형태로 얻을 수 있다.

```
{ "port": 80, "origin": "http://localhost", "prefix": "/mySockJS"}
```

또한, HTTP 서버가 서비스할 `index.html` 파일과 `vertxbus-2.1.js` 파일이 `SockJSEbbChatServerVerticle` 파일과 동일한 경로에 존재하는지 확인하자. `index.html` 파일은 ‘3.3.2 SockJS EBB 채팅 클라이언트’에서 확인할 SockJS EBB 채팅 클라이언트의 소스 코드([코드 3-6])고, `vertxbus-2.1.js`¹⁵는 Vert.x에서 제공하는 SockJS EBB 클라이언트 라이브러리이다.

¹⁴ 설정파일을 포함하여 Verticle를 실행하려면 ‘`vertx run YourVerticleFilePath -conf YourConfigFilePath`’처럼 ‘`-conf`’ 옵션을 사용한다.

¹⁵ `vertxbus-2.1.js`파일은 Vert.x 배포본의 `client` 디렉터리에서 찾을 수 있다.

3.3.2 SockJS EBB 채팅 클라이언트

앞에서 SockJSEbbChatServerVerticle의 HTTP 서버로 서비스한 index.html 파일이 SockJS EBB 채팅 클라이언트다. 채팅 클라이언트는 Vert.x에서 제공하는 vertxbus-2.1.js와 SockJS 자바스크립트 라이브러리로 작성할 수 있다.¹⁶

[코드 3-6] SockJS EBB 채팅 클라이언트

```
<html>
<head>
<title>SockJS Chat Test</title>
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/
jquery.min.js"></script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/sockjs-client/0.3.4/
sockjs.min.js"></script>
<script src="http://localhost/vertxbus-2.1.js"></script>
<script type="text/javascript">
    var eb = null;
    var subscribed = false;
    var handler = function(msg, replyTo) {
        console.log('message', msg);
        $('#message-box').append(msg+'\n');
    };

    $(document).ready(function() {
        open();
        $('#subscribe-btn').click(function() {
            subscribe($('#adress').val());
        });
        $('#unsubscribe-btn').click(function() {
            unsubscribe($('#adress').val());
        });
        $('#send-btn').click(function() {

```

16 추가로 jQuery 자바스크립트 라이브러리가 사용되었다.

```

        publish($('#address').val(), $('#message').val());
    });
    $('#message').keypress(function(e) {
        if (e.which == 13) {
            publish($('#address').val(), $('#message').val());
        }
    });
});

function open() {
    if (!eb) {
        eb = new vertx.EventBus('http://localhost/mySockJS');
        eb.onopen = function() {
            console.log('open');
            $('#status-label').html('Status: connected');
        };
        eb.onclose = function() {
            console.log('close');
            $('#status-label').html('Status: Not connected');
        };
    }
}

function close() {
    if (eb) {
        eb.close();
    }
}

function subscribe(address) {
    if (eb && !subscribed) {
        eb.registerHandler(address, handler);
        subscribed = true;
        $('#status-label').html($('#status-label').html()+' , Subscribe:
'+address);
    }
}

```

```

}

function desubscribe(address) {
    if (eb && subscribed) {
        eb.unregisterHandler(address, handler);
        subscribed = false;
        $('#status-label').html('Status: connected');
    }
}

function publish(address, message) {
    $('#message').val('');
    $('#message').focus();

    if (!subscribed) {
        alert('Subscribe이 필요합니다.!');
        return;
    }
    if (eb && message.length>0) {
        eb.publish(address, message);
    }
}
</script>
</head>
<body>
    <div id="status-label">Status: Not connected</div>
    <hr/>
    <label>Address: </label><input type="text" id="address"
value="com.devop.vertx.ch3">
    <input type="button" id="subscribe-btn" value="Subscribe">
    <input type="button" id="desubscribe-btn" value="Desubscribe">
    <hr/>
    <textarea id="message-box" rows="20" cols="55"></textarea><br/>
    <label>Message: </label><input type="text" id="message" value="" size="40">
    <input type="button" id="send-btn" value="Send">

```

```
</body>  
</html>
```

SockJS EBB 채팅 클라이언트는 5개의 메서드와 이벤트 버스로 메시지를 수신할 때 실행되는 1개의 이벤트 핸들러로 구성된다.

[표 3-5] SockJS EBB 채팅 클라이언트 메서드와 이벤트 핸들러

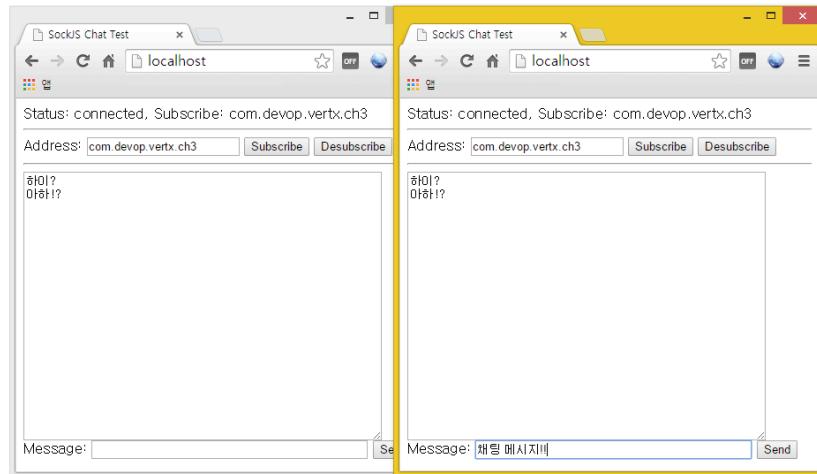
분류	종류	설명
메서드	open()	SockJS EBB 서버와 연결한다. 연결이 성공하면 화면 상단에 'Status: connected'라는 메시지를 출력하고, 연결이 종료되면 화면 상단에 'Status: Not connected'라는 메시지를 출력한다.
	close()	SockJS EBB 서버와 연결을 종료한다.
	subscribe()	Address InputBox에 정의된 주소로 이벤트 버스를 연결한다. 연결이 성공하면 화면 상단에 연결된 주소를 출력한다.
	desubscribe()	Address InputBox에 정의된 주소로 연결된 이벤트 버스를 종료한다. 연결이 종료되면 화면 상단에 출력된 주소를 삭제한다.
	publish()	Address InputBox에 정의된 주소로 연결된 이벤트 버스에 메시지를 브로드캐스팅한다.
이벤트 핸들러	function(msg, replyTo)	Address InputBox에 정의된 주소로 Subscribe를 성공할 경우, 해당 주소에 연결된 이벤트 버스가 메시지를 수신할 때 마다 실행된다. 수신된 메시지는 하단의 text area에 출력된다.

웹 브라우저에 <http://localhost/>를 입력해 보자. SockJSEbbChatServer Verticle이 정상적으로 실행되면 [그림 3-2]와 같은 화면이 뜨고, 상단에 'Status: connected'라는 메시지가 출력된다. 이것은 SockJS EBB 채팅 클라이언트가 SockJS EBB 채팅 서버와 연결된 것을 의미한다. 화면에서 Subscribe 버튼을 클릭하면 Address InputBox에 정의된 주소로 이벤트 버스가 연결된다.

이제 채팅을 위한 모든 준비가 끝났다. 메시지를 입력해 보자. 2개의 웹 브라우저에서 채팅 메시지가 전달되는 것을 볼 수 있다.¹⁷

17 Address InputBox의 주소가 다르면 채팅 메시지가 전달되지 않으므로 주의한다.

[그림 3-2] SockJS EBB 클라이언트 실행 결과 화면



두 개 이상의 웹 브라우저에서 채팅 메시지가 올바르게 전달되는 것을 확인했으니 이제 크롬 개발자 도구로 SockJS EBB 채팅 클라이언트를 좀 더 자세히 분석해보자.

[그림 3-3] SockJS EBB 채팅 클라이언트 프레임 흐름 모니터링

Data	Length	Time
["\ntype\nping"]	23	오후 4:22:30
a["\naddress\ncom.devop.vertx.ch3\n"\nbody\n"hi?"]]	59	오후 4:22:26
["\ntype\npublish\n"\naddress\n"com.devop.vertx.ch3"\nbody\n"hi?"]]	79	오후 4:22:26
["\ntype\nping"]	23	오후 4:22:25
["\ntype\nping"]	23	오후 4:22:20
["\ntype\nregister\n"\naddress\n"com.devop.vertx.ch3"]]	63	오후 4:22:15
["\ntype\nping"]	23	오후 4:22:15
["\ntype\nping"]	23	오후 4:22:10
h	1	오후 4:22:10
["\ntype\nping"]	23	오후 4:22:05
["\ntype\nping"]	23	오후 4:22:00
["\ntype\nping"]	23	오후 4:21:55
["\ntype\nping"]	23	오후 4:21:50
["\ntype\nping"]	23	오후 4:21:45
h	1	오후 4:21:45
["\ntype\nping"]	23	오후 4:21:40
["\ntype\nping"]	23	오후 4:21:35
["\ntype\nping"]	23	오후 4:21:30
["\ntype\nping"]	23	오후 4:21:25
["\ntype\nping"]	23	오후 4:21:20
o	1	오후 4:21:20

[그림 3-3]은 SockJS를 통해 송수신되는 프레임 흐름을 최근 시간순으로 배열한 것이다. 흰색 행은 서버로부터 전송받은 메시지, 초록색 행은 클라이언트가 서버로 전송한 메시지를 의미한다.¹⁸ 이 그림에서 25초 간격으로 서버로부터 ‘h’라는 메시지를 수신하는 부분이 있는데, 이것은 [표 3-3]에서 확인한 SockJS Heartbeat로 SockJSServer의 heartbeat_period라는 설정값에 의해 발생한다. 이는 서버/클라이언트 연결의 중간에 위치하는 로드 밸런서 등의 네트워크 장비가 연결을 강제로 종료시키는 것을 방지한다.

NOTE_SockJS Heartbeat 전송 구현

org.vertx.java.core.sockjs.impl.Session 구현을 보면 heartbeat_period 값에 따라 일정 주기로 org.vertx.java.core.sockjs.impl.TransportListener 인터페이스에 정의된 sendFrame() 메서드를 이용해 Heartbeat를 전송하는 것을 확인할 수 있다.

TransportListener 인터페이스의 구체 클래스는 SockJSServer가 지원하는 Transport 방식을 의미하며 XhrTransport, EventSourceTransport, HtmlFileTransport, JsonPTransport, WebSocketTransport 클래스가 정의되어 있다.

가장 하단의 ‘o’는 SockJS Open을 의미하며, 그 이후 5초 간격으로 ‘type: ping’이라는 메시지가 클라이언트에서 서버로 전송되는 것을 볼 수 있다. ‘type: ping’ 메시지는 SockJS EBB 클라이언트가 일정 주기로 SockJS EBB 서버에 전송하는 메시지인데, SockJS EBB 서버는 SockJS EBB 클라이언트가 일정 시간 안에 ‘type: ping’ 메시지를 보내지 않으면 해당 SockJS EBB 클라이언트와의 연결을 종료시킨다.¹⁹ [코드 3-3]에서 하단의 타이머 관련 코드가 SockJS EBB 클라이언트가 전송한 ‘type: ping’ 메시지의 최종 수신 시각을 현재 시각과 비교해 범위 안에 있지 않으면 SockJS EBB 클라이언트를 종료시키는 부분이다.

18 크롬에서는 WebSocket 사용이 가능해서 SockJS는 내부적으로 WebSocket을 사용한다.

19 SockJS EBB 서버의 ping_interval 기본값은 10초고, SockJS EBB 클라이언트의 vertxbus_ping_interval 기본값은 5초다.

다음 코드는 SockJS EBB 서버의 이러한 특징에 대응하도록 vertxbus-2.1.js에 정의된 부분으로, 일정 주기로 SockJS EBB 서버에 ‘type: ping’ 메시지를 전송하도록 설정한다.

```
sockJSConn.onopen = function() {
    // Send the first ping then send a ping every pingInterval milliseconds
    sendPing();
    pingTimerID = setInterval(sendPing, pingInterval);
    state = vertx.EventBus.OPEN;
    if (that.onopen) {
        that.onopen();
    }
};
```

SockJS EBB 클라이언트에서 ‘type: ping’ 메시지 전송 주기는 vertx bus_ping_interval에서 변경할 수 있다.

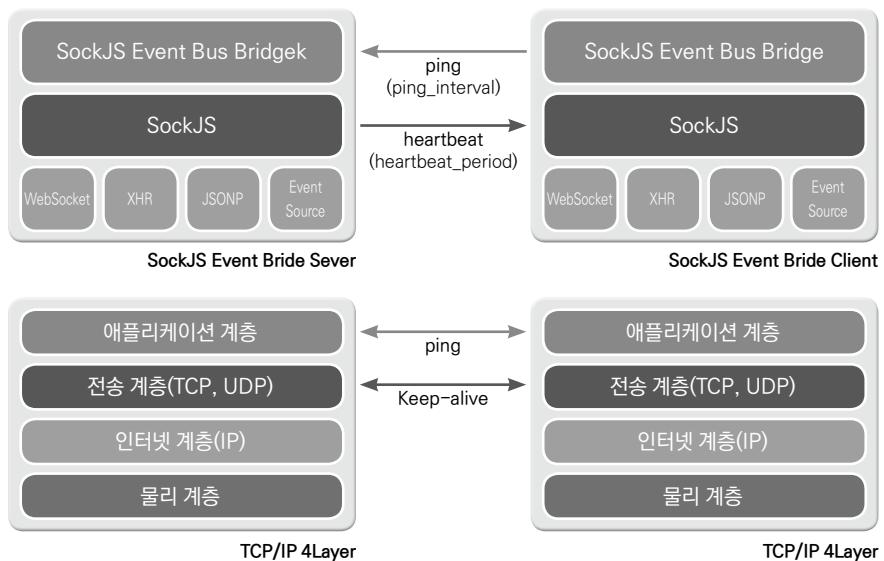
```
var options = {};
options.vertxbus_ping_interval = 10000;
eb = new vertx.EventBus('http://localhost/mySockJS', options);
```

SockJS EBB 클라이언트에서 ‘type: ping’ 메시지 전송 주기를 변경하면 SockJS EBB 서버에서도 적절한 ping_interval이라는 값을 설정해야 한다. 이는 SockJS EBB 클라이언트에 설정된 전송 주기보다 SockJS EBB 서버의 ping_interval 값이 작으면 SockJS EBB 클라이언트가 강제 종료되기 때문이다.

```
JsonObject bridgeConf = new JsonObject();
bridgeConf.putNumber("ping_interval", 20000);
sockJSServer.bridge(sockJSconfig, permitted, permitted, bridgeConf);
```

heartbeat_period와 ping_interval 값 둘 다 일정하게 SockJS EBB 서버와 클라이언트 사이에 전송되는 메시지의 간격을 설정하지만, 이 둘의 존재 목적과 전송 방식은 완전히 다르다. 집적적인 비교는 어렵지만 TCP의 keep-alive와 애플리케이션 레벨에서 정의된 ping을 각각 heartbeat_period와 ping_interval이라고 생각해 볼 수 있다. keep-alive와 heartbeat_period는 애플리케이션 레벨보다 낮은 수준의 레벨에서 정의된 프로토콜 규칙이고, 애플리케이션 레벨에서 정의된 ping과 ping_interval은 애플리케이션 특성에 적합하도록 고유하게 정의된 규칙이라 할 수 있다.²⁰

[그림 3-4] SockJS EBB의 heartbeat, ping과 TCP/IP 4Layer



²⁰ SockJS EBB는 SockJS 기반으로 작성된 하나의 애플리케이션으로 본다. 물론 SockJS 자체는 TCP를 기반으로 하는 애플리케이션 프로토콜이므로 이러한 비유가 반드시 올바른 것은 아니다. 단지 heartbeat_period와 ping_interval이 동작하는 계층이 서로 다르다는 것과 그 특성의 차이를 설명하기 위함이다.

[표 3-6] SockJS EBB의 heartbeat와 ping 비교

구분	heartbeat	ping
작동계층	SockJS	SockJS EBB(SockJS 응용 계층)
전송방향	서버 → 클라이언트	클라이언트 → 서버
목적	서버/클라이언트 연결의 중간에 위치하는 로드밸런서 등의 네트워크 장비가 연결을 강제로 종료하는 것을 방지한다.	애플리케이션이 정상적으로 동작하는지 확인한다.
처리	처리 없음	일정 시간 동안 서버가 ping을 수신하지 못하면 해당 클라이언트와의 연결을 종료한다
설정값	heartbeat_period(기본값 25초)	서버 ping_interval(기본값 10초) 클라이언트 vertxbus_ping_interval(기본값 5초)

[그림 3-3]에서 좀 더 살펴볼 부분은 ‘type: register’와 ‘type: publish’ 메시지다. ‘type: register’ 메시지는 Subscribe 버튼을 클릭했을 때 발생하고 Address InputBox에 정의된 주소로 이벤트 버스가 연결된다. ‘type: publish’ 메시지는 채팅 메시지를 전송할 때 발생하고 정의된 주소로 메시지를 브로드캐스팅한다. 브로드캐스팅된 메시지는 메시지를 발신한 자기 자신에게도 수신되어 출력되는데, ‘type: publish’ 메시지 바로 위 흰색 행이 브로드캐스팅된 메시지를 수신한 부분이다.

3.4 SockJS EBB 보안 설정

SockJS EBB를 사용할 때 보안상 인증되지 않은 SockJS EBB 클라이언트 애플리케이션에서 이벤트 버스를 통해 악의적 메시지를 서버 Verticle이나 다른 클라이언트 애플리케이션에 전달해 의도치 않은 동작을 일으키는 것을 주의해야 한다. 앞서 살펴본 [코드 3-5] SockJS EBB 채팅 서버 예제는 보안 처리와 관련하여 아무런 제약 없이 작성되었기 때문에 SockJS EBB 클라이언트 애플리케이션이 어떤 주소로 어떤 메시지를 전송하고 있는지 주의를 기울여야 한다.

기본적으로 SockJS EBB의 보안 처리 동작은 Whitelist를 기반으로 허가되지 않은

모든 요청을 거부하게 되어 있다. 이러한 Whitelist 기반 보안 정책은 개발자에게 사용 가능한 이벤트 버스 주소와 SockJS EBB 클라이언트 애플리케이션이 전달할 수 있는 메시지의 종류를 명시적으로 선언하게 하여 의도치 않은 특정 메시지에 반응하여 이벤트 핸들러 코드가 실행되는 것을 효과적으로 방지한다.

Whitelist 지정은 `SockJSserver`의 `bridge()` 메서드의 두 번째와 세 번째 파라미터를 통해 이루어지는데, 각각 inbound/outbound 퍼미션Permission을 의미한다. [코드 3-5]는 inbound/outbound 퍼미션 배열에 비어있는 JSON 객체를 입력하여(비어있는 퍼미션 배열이 아님에 주의한다) 모든 요청을 허가하도록 설정되었다.

NOTE_

그 어떤 JSON 객체도 포함하지 않은 비어있는 퍼미션 배열을 입력하면 모든 요청을 거부한다. Inbound/outbound 퍼미션을 기반으로 요청이 거부되는 것은 로그 레벨을 FINER로 설정하면 디버그 메시지 출력으로 확인할 수 있다. 이 로그 레벨 설정은 Vert.x 설치 디렉터리에서 `conf/logging.properties`를 수정하면 된다.

inbound/outbound 퍼미션에 관한 자세한 처리 과정은 `org.vertx.java.core.sockjs.EventBusBridge`의 `handleSocketData()` 메서드와 `checkMatches()` 메서드 구현으로 알아보고, 추가로 사용자 인증 기반 보안 설정 방법을 알아보자. 이 내용을 모두 이해하면 개발하려는 애플리케이션 성격에 맞게 사용자 인증 방식을 커스터마이징할 수 있다.

3.4.1 inbound/outbound 퍼미션

[코드 3-7]의 `handleSocketData()` 메서드는 SockJS EBB를 통해 서버가 메시지를 수신할 경우 호출된다. 이 메서드 구현을 살펴보면 수신한 메시지의 종류에 따라 `internalHandle***` 같은 명칭의 메서드를 호출하는데, 이 중에서

inbound/outbound 퍼미션에 관련된 처리가 이루어지는 메서드는 두 개다. 하나는 클라이언트 애플리케이션이 send 또는 publish를 요청했을 때 실행되는 internalHandleSendOrPub() 메서드고, 다른 하나는 클라이언트 애플리케이션이 registerHandler() 메서드를 요청했을 때 실행되는 internalHandleRegister() 메서드다.²¹

[코드 3-7] org.vertx.java.core.sockjs.EventBusBridge의 handleSocketData() 메서드 구현

```
private void handleSocketData(SockJSocket sock, Buffer data, Map<String, Handler<Message>> handlers) {
    JsonObject msg = new JsonObject(data.toString());

    String type = getMandatoryString(msg, "type");
    switch (type) {
        case "send":
            String address = getMandatoryString(msg, "address");
            internalHandleSendOrPub(sock, true, msg, address);
            break;
        case "publish":
            address = getMandatoryString(msg, "address");
            internalHandleSendOrPub(sock, false, msg, address);
            break;
        case "register":
            address = getMandatoryString(msg, "address");
            internalHandleRegister(sock, msg, address, handlers);
            break;
        case "unregister":
            address = getMandatoryString(msg, "address");
            internalHandleUnregister(sock, address, handlers);
            break;
        case "ping":
            internalHandlePing(sock);
```

21 [그림 3-3]에서 SockJS를 통해 'type: register', 'type: publish' 같은 메시지가 전송되는 것을 확인할 수 있다.

```

        break;
    default:
        throw new IllegalStateException("Invalid type: " + type);
    }
}

```

internalHandleSendOrPub() 메서드와 internalHandleRegister() 메서드는 다시 내부적으로 checkMatches() 메서드를 호출하는데, 이 메서드가 inbound/outbound 퍼미션을 기반으로 요청을 처리할지 거부할지 결정하는 역할을 한다.

[코드 3-8] org.vertx.java.core.sockjs.EventBusBridge의 checkMatches() 메서드 구현

```

private Match checkMatches(boolean inbound, String address, Object body) {
    if (inbound && acceptedReplyAddresses.remove(address)) {
        // This is an inbound reply, so we accept it
        return new Match(true, false);
    }

    List<JsonObject> matches = inbound ? inboundPermitted : outboundPermitted;

    for (JsonObject matchHolder: matches) {
        String matchAddress = matchHolder.getString("address");
        String matchRegex;
        if (matchAddress == null) {
            matchRegex = matchHolder.getString("address_re");
        } else {
            matchRegex = null;
        }

        boolean addressOK;
        if (matchAddress == null) {
            if (matchRegex == null) {
                addressOK = true;
            } else {
                addressOK = regexMatches(matchRegex, address);
            }
        } else {
            if (!matchAddress.equals(address))
                continue;
            addressOK = true;
        }
        if (addressOK)
            return new Match(true, false);
    }
}

```

```

        }
    } else {
        addressOK = matchAddress.equals(address);
    }

    if (addressOK) {
        boolean matched = structureMatches(matchHolder.getObject("match"),
body);
        if (matched) {
            Boolean b = matchHolder.getBoolean("requires_auth");
            return new Match(true, b != null && b);
        }
    }
}
return new Match(false, false);
}

```

`checkMatches()` 메서드의 첫 번째 파라미터는 어떤 퍼미션을 사용할지 결정하는 값으로, true면 inbound 퍼미션을 사용한다. inbound/outbound에 관한 구체적인 정의는 [표 3-7]을 참고한다.

[표 3-7] inbound/outbound 정의

구분	설명
inbound	클라이언트에서 서버로 전송되는 메시지에 대한 퍼미션을 정의한다. 즉, 클라이언트 애플리케이션에서 send 또는 publish를 요청했을 때 퍼미션 확인이 이루어진다.
outbound	서버에서 클라이언트로 전송되는 메시지에 대한 퍼미션을 정의한다. 클라이언트 애플리케이션에서 registerHandler를 요청했을 때, 그리고 서버가 수신한 메시지를 클라이언트로 전송하기 전에 퍼미션 확인이 이루어진다. ²²

사용할 퍼미션이 결정되면 해당 퍼미션 배열을 순차적으로 조회하여 요청을

²² `registerHandler`를 요청했을 때 outbound 퍼미션 확인이 이루어지는 것이 다소 어색할 수 있다. 그러나 클라이언트가 서버로부터 메시지를 수신할 경우 `registerHandler`에 등록한 이벤트 핸들러가 실행된다는 것을 떠올려보면 `registerHandler`를 요청했을 때 outbound 퍼미션을 확인하는 것이 그리 이상하지는 않다.

처리할지 거부할지 결정한다. 퍼미션 배열의 모든 조건을 만족시킬 필요는 없으며, 한 개라도 퍼미션 조건을 만족시키면 된다. 퍼미션 결정 방식은 세 가지가 있는데, 각각 address, address_re, match 모드라 한다.

address 모드

inbound/outbound 퍼미션의 address에 명시된 주소에 대해서만 요청을 처리한다. 다음 예시는 ‘com.devop.vertx.ch3’라는 주소에 대해서만 메시지를 수신하고 처리하도록 설정한 것이다.

```
JSONArray inbound = new JSONArray();
inbound.add(new JSONObject().putString("address", "com.devop.vertx.ch3"));
JSONArray outbound = new JSONArray();
outbound.add(new JSONObject().putString("address", "com.devop.vertx.ch3"));
sockJSServer.bridge(sockJSconfig, inbound, outbound);
```

address_re 모드

address 모드를 사용하면 사용 가능한 모든 주소를 명시적으로 선언해야 하는 불편함이 있다. 이런 불편함을 해결하기 위해 address_re 모드는 정규표현식으로 사용 가능한 모든 주소를 명시할 수 있다. 또한, address_re 모드가 사용되면 address 모드의 설정값은 모두 무시된다. 다음 예시는 ‘com.devop.vertx.ch3.*’라는 패턴의 주소에 대해서만 메시지를 수신하고 처리하도록 설정한 것이다.

```
JSONArray inbound = new JSONArray();
inbound.add(new JSONObject().putString("address_re",
"com\\.devop\\.vertx\\.ch3\\..+"));
JSONArray outbound = new JSONArray();
outbound.add(new JSONObject().putString("address_re",
"com\\.devop\\.vertx\\.ch3\\..+"));
```

```
sockJSServer.bridge(sockJSconfig, inbound, outbound);
```

match 모드

address 모드와 address_re 모드가 명시적 주소 지정과 관련한 설정이라면, match 모드는 이와 다르게 메시지 구조를 기반으로 요청을 처리할지 거부할지 결정한다. 따라서 address 모드나 address_re 모드와 함께 사용할 수 있다. 단, match 모드는 JSON 객체 메시지만 처리할 수 있으며 일반 string 메시지는 거부 처리한다.

모든 메시지는 inbound/outbound 퍼미션의 match에 명시된 JSON 객체와 동일한 필드를 포함해야 하고, 이 값이 모두 같아야 처리된다. 다음 예시는 'com.devop.vertx.ch3'이라는 주소에서 JSON 메시지가 필드 'app', 'type'를 포함하고, 이 값이 'my-chat', 'chat-msg'이어야만 처리되도록 설정한 것이다. address를 생략하고, match만 설정하면 주소와 관계없이 모든 메시지를 match에 명시된 JSON 객체와 비교하는 과정을 거치게 된다.

```
JSONArray inbound = new JSONArray();
inbound.add(new
JsonObject().putString("address", "com.devop.vertx.ch3").putObject("match",
new JsonObject().putString("app", "my-chat").putString("type", "chat-msg")));
JSONArray outbound = new JSONArray();
inbound.add(new JsonObject().putString
("address", "com.devop.vertx.ch3").putObject("match",
new JsonObject().putString("app", "my-chat").putString("type", "chat-msg")));
sockJSServer.bridge(sockJSconfig, inbound, outbound);
```

3.4.2 사용자 인증

앞서 살펴본 세 가지 방식에 따라 요청 처리가 결정되면, 최종으로 사용자 인증 여부를 확인한다. 사용자 인증 여부 확인은 ‘사용하지 않음’이 기본으로 설정되어 있는데, inbound/outbound 퍼미션에 `requires_auth`를 `true`로 설정하여 활성화한다.

사용자 인증 여부를 확인하도록 설정할 경우 ‘mod-auth-mgr’이라는 모듈이 필요하다.²³ mod-auth-mgr 모듈은 MongoDB를 사용해 사용자 아이디, 비밀번호를 확인하며 정보가 올바르면 만료시간이 정해진 세션 아이디를 생성해 돌려준다.²⁴ 이후 모든 요청은 이 세션 아이디를 포함하고 있어야만 정상적으로 처리된다.

그럼 SockJS EBB 채팅 서버에 사용자 인증을 확인하도록 설정하고, SockJS EBB 채팅 클라이언트에 로그인 기능을 추가해 보자.

SockJS EBB 채팅 서버를 설정하는 작업은 간단하다. [코드 3-5]에서 inbound/outbound 퍼미션을 다음과 같이 수정하고, `bridge()` 메서드의 파라미터로 입력하면 된다.

```
JSONArray inbound = new JSONArray();
inbound.add(new JSONObject().putString("address",
    "vertx.basicauthmanager.login"));
inbound.add(new JSONObject().putBoolean("requires_auth", true));
JSONArray outbound = new JSONArray();
outbound.add(new JSONObject());
JsonObject bridgeConfig = new JsonObject();
```

23 mod-auth-mgr은 [5.4.1 mod-auth-mgr](#)을 참고한다.

24 MongoDB를 다루기 위해 mod-auth-mgr 모듈은 내부적으로 mod-mongo-persistor 모듈을 사용하는데, mod-mongo-persistor 모듈은 [5.4.2 mod-mongo-persistor](#)를 참고한다. 따라서 사용자 인증 여부를 확인하려면 총 두 개의 모듈을 추가로 설치해야 한다.

```
bridgeConfig.putString("auth_address", "vertx.basicauthmanager.authorise");
bridgeConfig.putNumber("auth_timeout", 5 * 60 * 1000);

sockJSServer.bridge(sockJSconfig, inbound, outbound, bridgeConfig);
```

address로 입력한 ‘vertx.basicauthmanager.login’은 mod-auth-mgr 모듈에서 사용하는 이벤트 버스 주소로, 사용자 로그인 요청을 처리한다. 그 외의 모든 요청은 requires_auth를 true로 설정해 사용자 인증 여부를 확인한다(사용자 인증 여부 확인 설정이 추가된 Verticle은 SockJSEbbChatWithLoginServerVerticle이라 하자).

bridgeConfig의 auth_address는 mod-auth-mgr 모듈에서 사용하는 이벤트 버스 주소로, 세션 아이디의 유효성 확인 요청을 처리한다. auth_timeout은 SockJS EBB 서버에서 세션 아이디의 유효성을 캐시 Cache에 보관하는 시간을 설정한다(세션 아이디 유효성 확인을 위해 매번 mod-auth-mgr 모듈을 호출하는 것은 비효율적이므로 일정 시간 동안 SockJS EBB의 캐시에 유효한 세션 아이디를 저장해 둔다).

설정이 끝나면 추가로 필요한 두 개의 모듈과 함께 SockJSEbbChatWithLoginServerVerticle을 실행시키도록 SockJSEbbChatServerStarterVerticle을 작성한다.

[코드 3-9] 채팅 서버 시작 Verticle(SockJSEbbChatWithLoginServerStarterVerticle.java)

```
public class SockJSEbbChatWithLoginServerStarterVerticle extends Verticle {
    @Override
    public void start() {
        container.deployVerticle("SockJSEbbChatWithLoginServerVerticle.java");

        JsonObject authMgrConfig = new JsonObject();
        authMgrConfig.putString("address", "vertx.basicauthmanager");
        authMgrConfig.putString("user_collection", "users");
        authMgrConfig.putString("persistor_address", "vertx.mongodbpersistor");
        authMgrConfig.putNumber("session_timeout", 1800000); // --30min
```

```

        container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final",
authMgrConfig);

        JsonObject mongoPersistorConfig = new JsonObject();
        mongoPersistorConfig.putString("address", "vertx.mongopersistor");
        mongoPersistorConfig.putString("host", "localhost");
        mongoPersistorConfig.putNumber("port", 27017);
        mongoPersistorConfig.putString("db_name", "my-chat");
        mongoPersistorConfig.putNumber("pool_size", 10);
        container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",
mongoPersistorConfig);
    }
}

```

`authMgrConfig`과 `mongoPersistorConfig`는 각각 `mod-auth-mgr`과 `mod-mongo-persistor` 모듈의 기본 설정을 의미한다. `deployModule()` 메서드는 모듈을 실행시키기 위한 명령으로, 5장 [Vert.x 모듈 시스템](#)에서 자세히 살펴본다.

[표 3-8] `mod-auth-mgr`와 `mod-mongo-persistor` 모듈의 파라미터

모듈	파라미터	설명
mod-auth-mgr	address	로그인, 로그아웃, authorise 요청을 수신할 이벤트 버스 주소의 prefix다. 각 요청 주소는 prefix 값에 요청 이름이 결합되어 생성된다. 예를 들어, 로그인 요청의 주소는 'vertx.basicauthmanager.login'이 된다. 이 주소는 앞서 SockJS EBB 채팅 서버의 퍼미션 중 address와 그 값이 같아야 한다.
	user_collection	사용할 MongoDB 컬렉션을 지정한다.
	persistor_address	<code>mod-mongo-persistor</code> 의 이벤트 버스 주소다.
	session_timeout	로그인에 성공하면 세션 아이디를 발급하는데, 세션 아이디의 유효시간을 설정한다.
mod-mongo-persistor	address	<code>mod-mongo-persistor</code> 의 이벤트 버스 주소다.
	host, port	MongoDB의 호스트와 포트를 입력한다.
	db_name	사용할 데이터베이스를 지정한다.
	pool_size	커넥션 풀(Connection Pool)의 크기를 지정한다.

SockJS Ebb Chat Server Starter Verteicle을 실행하기 전에 MongoDB를 설치하고 앞에서 설명한 설정을 참고하여 my-chat이라는 데이터베이스를 생성한 다음 해당 데이터베이스에 users 컬렉션을 생성한다. 그리고 테스트 사용자 계정을 하나 만들어 로그인 테스트를 진행할 수 있게 하면 이것으로 서버 준비는 끝난다.

```
use my-chat;
db.createCollection("users");
db.users.insert({username:"myid", password:"1234"});
db.users.find({username:"myid"});
```

이제 [코드 3-6] 채팅 클라이언트를 기반으로 로그인/로그아웃 기능을 추가해보자. 로그인 기능은 운이 좋게도 vertxbus-2.1.js에 이미 구현되어 있다. 따라서 간단히 사용자 아이디와 비밀번호, 요청이 완료되었을 때 호출될 이벤트 핸들러를 파라미터로 입력하기만 하면 된다. 로그인 성공 시 해당 이벤트 핸들러는 '{"status": "ok"}'라는 응답 메시지를 수신한다. 로그인 성공 결과로 발급된 세션 아이디는 vertx.EventBus 객체가 내부적으로 관리하므로 여기서 따로 처리할 필요는 없다.

[코드 3-10] 로그인/로그아웃 기능이 추가된 SockJS EBB 채팅 클라이언트

```
<html>
<head>
<title>SockJS Chat Test</title>
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/
jquery.min.js"></script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/sockjs-client/0.3.4/
sockjs.min.js"></script>
<script src="http://localhost/vertxbus-2.1.js"></script>
<script type="text/javascript">
var eb = null;
```

```

var subscribed = false;
var handler = function(msg, replyTo) {
    console.log('message', msg);
    $('#message-box').append(msg+'\n');
};

$(document).ready(function() {
    open();
    $('#subscribe-btn').click(function() {
        subscribe($('#adress').val());
    });
    $('#unsubscribe-btn').click(function() {
        unsubscribe($('#adress').val());
    });
    $('#send-btn').click(function() {
        publish($('#adress').val(), $('#message').val());
    });
    $('#message').keypress(function(e) {
        if (e.which == 13) {
            publish($('#adress').val(), $('#message').val());
        }
    });
});

function open() {
    if (!eb) {
        eb = new vertx.EventBus('http://localhost/mySockJS');

        eb.onopen = function() {
            console.log('open');
            $('#status-label').html('Status: connected');
        };
        eb.onclose = function() {
            console.log('close');
            $('#status-label').html('Status: Not connected');
        };
    }
}

```

```

        }
    }

    function close() {
        if (eb) {
            eb.close();
        }
    }

    function login() {
        var id = $('#id').val()
        var passwd = $('#passwd').val();
        if (eb && id.length>0 && passwd.length>0) {
            eb.login(id, passwd, function(reply) {
                console.log(reply);
                if (reply.status == 'ok') {
                    loginOk(id);
                }
                else {
                    alert('로그인 실패!');
                }
            });
        }
    }

    function subscribe(address) {
        if (eb && !subscribed) {
            eb.registerHandler(address, handler);
            subscribed = true;
            $('#status-label').html($('#status-label').html()+'Subscribe:'+
            address);
        }
    }

    function desubscribe(address) {
        if (eb && subscribed) {

```

```

        eb.unregisterHandler(address, handler);
        subscribed = false;
        $('#status-label').html('Status: connected');
    }
}

function publish(address, message) {
    $('#message').val('');
    $('#message').focus();

    if (!eb.sessionID) {
        alert('로그인이 필요합니다.!');
        $('#id').focus();
        return;
    }
    if (!subscribed) {
        alert('Subscribe이 필요합니다.!');
        return;
    }
    if (eb && message.length>0) {
        eb.publish(address, message);
    }
}

function loginOk(id) {
    var html = id+'님 안녕하세요!! ';
    html += '<input type="button" id="logout-btn" value="Logout"';
    onclick="logout();">';
    $('#login-area').html(html);
}

function logout() {
    if (eb) {
        desubscribe($('#address'));
        eb.send('vertx.basicauthmanager.logout', { sessionID: eb.
sessionID }, function(reply) {

```

```

        if (reply.status === 'ok') {
            delete eb.sessionID;
            var html = '<label>ID: </label><input type="text" id="id" value="" size="15"><label>&nbsp;PASSWD: </label><input type="password" id="passwd" value="" size="15">';
            html += '<input type="button" id="login-btn" value="Login" onclick="login();">';
            $('#login-area').html(html);
        }
    });
}
}

</script>
</head>
<body>

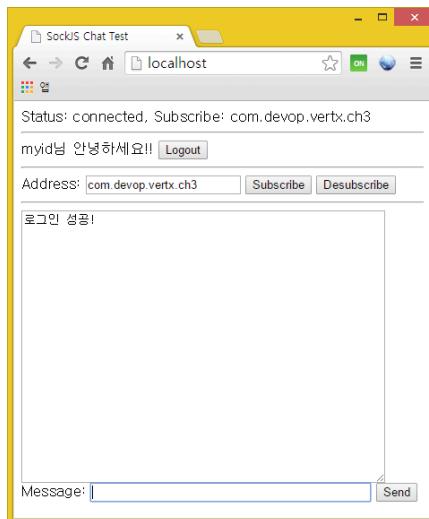
<div id="status-label">Status: Not connected</div>
<hr/>
<div id="login-area">
    <label>ID: </label><input type="text" id="id" value="" size="15"><label>&nbsp;PASSWD: </label><input type="password" id="passwd" value="" size="15">
    <input type="button" id="login-btn" value="Login" onclick="login();">
</div>
<hr/>
<label>Address: </label><input type="text" id="adress" value="com.devop.vertx.ch3">
<input type="button" id="subscribe-btn" value="Subscribe">
<input type="button" id="desubscribe-btn" value="Desubscribe">
<hr/>
<textarea id="message-box" rows="20" cols="55"></textarea><br/>
<label>Message: </label><input type="text" id="message" value="" size="40">
<input type="button" id="send-btn" value="Send">
</body>
</html>

```

로그아웃 기능은 따로 제공되지 않으므로 별도로 mod-auth-mgr에 로그아웃을 요청하는 기능을 추가해야 한다. 로그아웃 요청 주소는 기본 설정값을 따를 경우 ‘vertx.basicauthmanager.logout’이 된다. 요청이 완료되었을 때 호출될 이벤트 핸들러를 통해 로그아웃 정상 처리가 확인되면 vertx.EventBus 객체가 관리하는 세션 아이디를 명시적으로 삭제한다.²⁵

SockJSebbChatServerStarterVerticle을 실행하고, 웹 브라우저에 <http://localhost/>를 입력해 보자. [그림 3-5]와 같은 화면이 뜨고, 로그인을 해야만 메시지를 전송할 수 있는 것을 확인할 수 있다. 로그인 후 메시지를 전송하면 vertx.EventBus 객체는 내부적으로 채팅 메시지에 세션 아이디를 추가해 서버에 전송한다. 서버에서 requires_auth를 true로 설정했는데, 세션 아이디가 메시지에 포함되어 있지 않으면, 해당 메시지는 서버에서 무시된다.

[그림 3-5] 로그인 후 메시지 전송화면



25 세션 아이디를 명시적으로 삭제하지 않으면 mod-auth-mgr에서는 세션 아이디가 삭제되어도 SockJS EBB 서버가 세션 아이디를 캐시하는 동안은 여전히 유효하게 처리된다.

NOTE

세션 아이디가 메시지에 포함되어 있지 않으면 “Inbound message for address “ + address + “ rejected because it requires auth and sessionID is missing”이라는 디버그 메시지가 출력된다. 세션 아이디가 포함되어 있으나 유효하지 않으면 “Inbound message for address “ + address + “ rejected because sessionID is not authorized”이라는 디버그 메시지가 출력된다.

3.4.3 사용자 인증 커스터마이징

mod-auth-mgr을 이용한 사용자 인증 방식은 간단하고 사용하기 쉽지만, 개발하려는 애플리케이션 성격에 맞게 사용자 인증 방식을 커스터마이징해야 하는 경우도 있다.

클라이언트 애플리케이션은 기본 사용자 인증 방식으로 세션 아이디를 발급받고, 메시지에 세션 아이디를 추가해 서버에 전송한다. 여기까지는 기존 mod-auth-mgr을 사용한 사용자 인증 방식과 동일하다.

SockJS EBB 서버에서는 세션 아이디를 처리하기 위해 `authorize()` 메서드를 호출한다. `authorize()` 메서드는 먼저 자체 캐시로 세션 아이디의 유효성을 확인하고, 캐시에서 세션 아이디를 발견하지 못하면 mod-auth-mgr에 세션 아이디 유효성 검증을 위임한다. 세션 아이디 유효성 검증 요청 주소는 기본 설정값을 따를 경우 ‘`vertx.basicauthmanager.authorise`’가 된다. 사용자 인증 방식을 커스터마이징하는 것은 이 `authorize()` 메서드의 동작 방식을 변경하는 것이다.²⁶

`authorize()` 메서드는 EventBusBridgeHook의 `handleAuthorise()` 메서드를 호출하는데, 이 메서드가 `true`를 반환하면 `authorize()` 메서드는 더는 아무런

²⁶ 눈치가 빠른 독자는 [코드 3-5]에서 EventBusBridgeHook의 `handleAuthorise()` 메서드 오버라이드로 이러한 변경이 가능하다는 것을 알아챘을 것이다.

처리를 하지 않는다. 즉, EventBusBridgeHook의 handleAuthorise() 메서드가 사용자 인증 방식을 커스터마이징해야 할 때 코드를 작성하는 부분으로, 커스터마이징 코드를 작성한 후 단순히 반환값으로 true를 지정하기만 하면 된다.

[코드 3-11] org.vertx.java.core.sockjs.EventBusBridge의 authorise() 메서드 구현

```
private void authorise(final JsonObject message, final String sessionID, final Handler<AsyncResult<Boolean>> handler) {
    if (!handleAuthorise(message, sessionID, handler)) {
        // If session id is in local cache we'll consider them authorised
        final DefaultFutureResult<Boolean> res = new DefaultFutureResult<>();
        if (authCache.containsKey(sessionID)) {
            res.setResult(true).setHandler(handler);
        } else {
            eb.send(authAddress, message, new Handler<Message<JsonObject>>() {
                public void handle(Message<JsonObject> reply) {
                    boolean authed = reply.body().getString("status").equals("ok");
                    res.setResult(authed).setHandler(handler);
                }
            });
        }
    }
}
```

3.5 요약

지금까지 채팅 서비스 같은 실시간 반응형 웹 애플리케이션을 개발하기 위해 Vert.x에서 제공하는 SockJS Server와 이벤트 버스를 클라이언트 영역에서 사용할 수 있도록 확장한 SockJS EBB에 관해 알아보았다. 개인적으로는 SockJS와 SockJS EBB 기능이야말로 Vert.x에 관심 있는 개발자가 반드시 주목해야 하는 Vert.x의 장점이라 생각하기 때문에 3장의 내용을 비교적 자세히 설명하기 위해

노력했다. 그러나 필자가 놓친 부분이 있을 수 있고, 내용 전달이 어려웠던 부분도 있으므로 SockJS와 SockJS EBB를 올바르게 이해하고 효과적으로 사용하기 위해 다음에 명시한 Vert.x 관련 코드를 반드시 검토해 보기를 권한다.

SockJS 관련 코드

- org.vertx.java.core.sockjs.impl.DefaultSockJSServer
- org.vertx.java.core.sockjs.impl.WebSocketTransport
- org.vertx.java.core.sockjs.impl.JsonPTransport
- org.vertx.java.core.sockjs.impl.HtmlFileTransport
- org.vertx.java.core.sockjs.impl.EventSourceTransport
- org.vertx.java.core.sockjs.impl.XhrTransport

SockJS EBB 관련 코드

- org.vertx.java.core.sockjs.EventBusBridge
- vertxbus-2.1.js

4 | TCP/SockJS EBB 채팅 서비스 통합

지금까지 Vert.x에서 TCP와 SockJS EBB를 이용해 채팅 서비스를 개발하는 방법을 알아보았다. 이 채팅 서비스는 각각 TCP와 SockJS EBB 프로토콜(근본적으로는 WebSocket 프로토콜)에 기반을 둔 고유의 채팅 클라이언트와 연결될 수 있다. 하지만 이처럼 독립적으로 프로토콜을 처리하도록 개발된 채팅 서비스는 TCP 프로토콜 기반 채팅 클라이언트가 전송하는 메시지를 SockJS EBB 프로토콜 기반 채팅 클라이언트에서는 수신할 수는 없다는 한계가 있다. 물론 이런 한계점은 TCP 프로토콜을 처리하는 채팅 서버와 SockJS EBB 프로토콜을 처리하는 채팅 서버가 서로 메시지를 교환할 수 있는 메커니즘을 추가함으로써 해결할 수 있다.

TCP나 SockJS EBB 같은 프로토콜 규격은 오직 클라이언트와 서버 사이의 메시지 교환을 위해서만 사용되며, 일단 서버로 전달된 채팅 메시지는 그것이 어떤 프로토콜을 통해 서버로 전달되었는지 고려할 필요 없이 항상 일관된 데이터를 포함하고 해석된다(JSON 객체 같은 문자열 기반 메시지 포맷은 바이너리 데이터를 처리하는 방식에 독립적이므로 이러한 요구조건에 잘 들어 맞는다). 이것은 결국 서버에 전달된 메시지는 그 출처에 관계 없이 TCP나 SockJS EBB 또는 다른 프로토콜에 기반을 둔 그 어떤 채팅 클라이언트에도 전달될 수 있음을 의미한다.

채팅 서비스가 다양한 프로토콜과 매체를 지원하는 특징은 요즘처럼 PC와 웹, 모바일의 경계가 모호해지고 사용자가 언제 어디서든 다양한 매체를 통해 서비스에 연결될 수 있다는 것을 고려했을 때 아주 강력한 장점이 된다. PC에서 Windows 전용 애플리케이션으로 채팅 서비스를 사용하다가도 이동 중일 때처럼 PC를 사용할 수 없는 환경에서는 모바일 애플리케이션으로, 자신의 PC가 아닌 공용 PC에서 전용 애플리케이션이 설치되어 있지 않을 때는 웹 브라우저로 채팅

서비스에 연결될 수 있다면, 사용자를 끊임없이 서비스에 머무르게 할 수 있고 서비스 접근성을 극대화하며 이는 결국 사용자의 서비스 충성도를 한층 높여주는 계기가 된다.

일상생활에서 흔히 접할 수 있는 KakaoTalk 같은 메신저 서비스를 PC와 모바일에서 동시에 사용하는 경험을 떠올려 보면 이 내용을 피부로 느낄 수 있다. 물론 다양한 매체를 지원하는 서비스의 장점은 이것 외에도 마케팅, 기술 경쟁, 편의성 등 경쟁 관계에 놓여있는 서비스들과 차별화하기 위해 다양한 측면에서 활용될 수 있다.

이번 장에서는 이처럼 중요한 특징인 서비스 매체 통합을 실습해 보기 위해 앞서 살펴본 TCP 채팅 서버와 SockJS EBB 채팅 서버를 통합하고, TCP 채팅 클라이언트와 SockJS EBB 채팅 클라이언트가 서로 메시지를 주고받는 것을 확인해 보겠다.

채팅 서비스 통합을 실습하기 위해 [2.4.3 Header/Payload 분할 방식이 적용된 \[코드 2-10\] TCPChatWithFramingServerVerticle](#)과 [3.3 SockJS Event Bus Bridge](#)를 적용한 [\[코드 3-5\] SockJSEbbChatServerVerticle](#)를 사용한다. 통합을 마치면 TCP 채팅 클라이언트가 전송한 메시지를 SockJS EBB 채팅 클라이언트에서도 확인할 수 있다. 물론 반대로 SockJS EBB 채팅 클라이언트가 전송한 메시지를 TCP 채팅 클라이언트에서도 확인할 수 있다.

통합 과정은 TCP와 SockJS EBB 채팅 서버가 서로 메시지를 교환할 수 있도록 메커니즘을 추가하는 것이다. 메시지 교환 메커니즘은 다양한 방식으로 구현할 수 있는데, 가장 먼저 생각할 수 있는 방법은 Advanced Message Queuing Protocol(이하 AMQP)⁰¹을 적용하는 것이다. AMQP는 서로 독립적인 2개 이상의 프로세스가 효과적으로 메시지를 교환할 수 있도록 고안된 소프트웨어인 만큼

01 http://en.wikipedia.org/wiki/Advanced_Message_Queueing_Protocol

지금 요구사항과도 잘 들어 맞는다. 또한, 다양한 오픈소스 구현체가 존재하여 실행 바이너리와 관련 자료를 구하기 쉽다는 장점도 있다.

그러나 Vert.x의 이벤트 버스라는 더 사용하기 쉽고 편리한 도구가 이미 손안에 있다. Vert.x에서 이벤트 버스는 서로 독립적으로 실행되는 Verticle 사이의 메시지 교환을 목적으로 존재하는 도구임을 다시금 기억하자. TCPChatWithFramingServerVerticle과 SockJSChatServerVerticle 사이의 메시지 교환 매커니즘을 구현하는 것이 목적이므로 이벤트 버스를 사용하는 것이 목적지로 향하는 가장 빠른 지름길이라 할 수 있다.

그럼 TCPChatWithFramingServerVerticle과 SockJSChatServerVerticle 사이의 메시지 교환을 하는 데 이벤트 버스를 적용하는 방법을 알아보자.

4.1 IntegratedTCPChatServerVerticle

[코드 2-10] TCPChatWithFramingServerVerticle을 기본 토대로 일부 기능을 추가한다. 추가되는 기능은 두 가지다. 첫 번째 추가 기능은 TCP 채팅 클라이언트가 전송한 메시지를 이벤트 버스를 통해 IntegratedSockJSChatServerVerticle에 전달하는 것이고, 두 번째 기능은 IntegratedSockJSChatServerVerticle에서 이벤트 버스로 전송한 메시지를 수신해 TCP 채팅 클라이언트에 전달하는 것이다(이 두 가지 기능이 추가된 TCP 채팅 서버를 IntegratedTCPChatServerVerticle이라 하자).

[코드 4-1] 통합 TCP 채팅 서버(IntegratedTCPChatServerVerticle.java)

```
public class IntegratedTCPChatServerVerticle extends Verticle {
    public static final String INTERNAL_TCP_SERV_ADDR =
        "com.devop.vertx.chat.internal.tcp";
    public static final String INTERNAL_SOCKJS_SERV_ADDR =
        "com.devop.vertx.chat.internal.sockjs";
```

```

private Logger logger;
private EventBus eb;
private NetServer server;
private Set<String> sockets;

@Override
public void start() {
    logger = container.logger();
    eb = vertx.eventBus();
    server = vertx.createNetServer();
    sockets = vertx.sharedData().getSet("sockets");

    server.connectHandler(new Handler<NetSocket>() {
        @Override
        public void handle(final NetSocket socket) {
            sockets.add(socket.writeHandlerID());
            //-- tcp stream framing
            final RecordParser framer = RecordParser.newBuilder().setFramerType(RecordParser.FramerType.FIXED).build();
            framer.setOutput(new Handler<Buffer>() {
                int payloadLength = -1;
                @Override
                public void handle(Buffer buffer) {
                    if (payloadLength == -1) {
                        payloadLength = buffer.getInt(0);
                        framer.fixedSizeMode(payloadLength);
                    }
                    else {
                        //-- send to sockjs server
                        eb.send(INTERNAL_SOCKJS_SERV_ADDR, buffer);

                        Buffer newbuf = new Buffer(4+payloadLength);
                        newbuf.setInt(0, payloadLength);
                        newbuf.setBuffer(4, buffer);
                        for (String s : sockets) {
                            if (!socket.writeHandlerID().equals(s))
                                eb.send(s, newbuf);
                        }
                    }
                }
            });
            socket.handler(new Handler<NetSocket>() {
                @Override
                public void handle(NetSocket next) {
                    logger.info("New connection from {} to {} via SockJS", next.remoteAddress(), server.localAddress());
                    next.handler(new Handler<NetSocket>() {
                        @Override
                        public void handle(NetSocket next) {
                            logger.info("Connection closed by client");
                            next.close();
                        }
                    });
                }
            });
        }
    });
}

```

```

        }
        framer.fixedSizeMode(4);
        payloadLength = -1;
    }
}
});

//-- this handler will be called every time data is received on
the socket
socket.dataHandler(framer);
//-- socket closed
socket.closeHandler(new VoidHandler() {
    @Override
    protected void handle() {
        sockets.remove(socket.writeHandlerID());
    }
});
//-- something went wrong
socket.exceptionHandler(new Handler<Throwable>() {
    @Override
    public void handle(Throwable throwable) {
        logger.error("unexpected exception: ", throwable);
    }
});
});

server.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {
    @Override
    public void handle(AsyncResult<NetServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});

//-- event bus subscribe
eb.registerHandler(INTERNAL_TCP_SERV_ADDR, new Handler<Message<String>>() {
    @Override

```

```

    public void handle(final Message<String> message) {
        byte[] data;
        try {
            data = message.body().getBytes("UTF-8");
            Buffer buffer = new Buffer(4+data.length);
            buffer.setInt(0, data.length);
            buffer.setBytes(4, data);
            for (String s : sockets) {
                eb.send(s, buffer);
            }
        } catch (UnsupportedEncodingException e) {
        }
    });
}

@Override
public void stop() {
    if (server != null)
        server.close();
}
}

```

IntegratedTCPChatServerVerticle은 INTERNAL_TCP_SERV_ADDR과 INTERNAL_SOCKJS_SERV_ADDR이라는 두 개의 상수를 정의하는데, 이 두 상수는 이벤트 버스의 주소로 사용된다. INTERNAL_TCP_SERV_ADDR은 SockJS EBB 채팅 서버가 수신한 채팅 메시지를 TCP 채팅 서버로 전달하는 데, INTERNAL_SOCKJS_SERV_ADDR는 TCP 채팅 서버가 수신한 채팅 메시지를 SockJS EBB 채팅 서버로 전달하는 데 사용된다. 여기서 INTERNAL_TCP_SERV_ADDR로 정의된 이벤트 버스 주소를 통해 수신한 메시지를 TCP 채팅 클라이언트에 전달할 때는 반드시 Header/Payload 구조로 Buffer를 채워줘야 한다는 점을 주의해야 한다.

4.2 IntegratedSockJSEbbChatServerVerticle

이번에는 [코드 3-5] SockJSEbbChatServerVerticle을 기본으로 일부 기능을 추가해 IntegratedSockJSEbbChatServerVerticle를 작성해 보자.

추가되는 기능은 IntegratedTCPChatServerVerticle과 마찬가지로 두 가지다. 첫 번째는 SockJS EBB 채팅 클라이언트가 전송한 메시지를 이벤트 버스를 통해 IntegratedTCPChatServerVerticle에 전달하는 기능이고, 두 번째는 IntegratedTCPChatServerVerticle에서 이벤트 버스를 통해 전송한 메시지를 수신해 SockJS EBB 채팅 클라이언트에 전달하는 기능이다.

[코드 4-2] 통합 SockJS EBB 채팅 서버(IntegratedSockJSEbbChatServerVerticle.java)

```
public class IntegratedSockJSEbbChatServerVerticle extends Verticle {
    public static final String INTERNAL_TCP_SERV_ADDR      =
"com.devop.vertx.chat.internal.tcp";
    public static final String INTERNAL_SOCKJS_SERV_ADDR = "com.devop.vertx.chat.internal.sockjs";

    private Logger logger;
    private EventBus eb;
    private HttpServer httpServer;
    private Map<String, Integer> addresses;
    private int port;
    private String origin;

    private String getClasspathResourceFile(String filepath) {
        try {
            File file = new File(getClass().getResource(filepath).toURI());
            return file.toString();
        } catch (URISyntaxException e) {
        }
        return null;
    }
}
```

```

@Override
public void start() {
    JsonObject appConfig = container.config();

    logger = container.logger();
    eb = vertx.eventBus();
    httpServer = vertx.createHttpServer();
    addresses = vertx.sharedData().getMap("addresses");
    port = appConfig.getInteger("port", 80);
    origin = appConfig.getString("origin", "http://localhost");

    httpServer.requestHandler(new Handler<HttpServerRequest>() {
        final String indexHtml = getClasspathResourceFile("/com/devop/
vertx/ch4/index.html");
        final String vertxBusJs = getClasspathResourceFile("/com/devop/
vertx/ch4/vertxbus-2.1.js");

        @Override
        public void handle(HttpServerRequest request) {
            if (request.path().equals("/")) request.response().sendFile(in-
dexHtml);
            else if (request.path().endsWith("vertxbus-2.1.js")) request.r-
esponse().sendFile(vertxBusJs);
            else request.response().setStatusCode(404).end("Not Found Page");
        }
    });
}

SockJSServer sockJSServer = vertx.createSockJSServer(httpServer);

//-- bridge hook
sockJSServer.setHook(new EventBusBridgeHook() {
    @Override
    public boolean handleSocketCreated(SockJSocket sock) {
        if (origin!=null) {
            String originHeader = sock.headers().get("origin");
            if (originHeader == null || !originHeader.equals(origin)) {
                return false; //-- reject the socket
            }
        }
    }
});

```

```

        }
    }
    return true; //-- true to accept the socket, false to reject it
}

@Override
public void handleSocketClosed(SockJSocket sock) {
    logger.info("handleSocketClosed, sock = " + sock);
}

@Override
public boolean handleSendOrPub(SockJSocket sock, boolean send,
JsonObject msg, String address) {
    logger.info("handleSendOrPub, sock = " + sock + ", send = " +
send + ", address = " + address);
    //-- send to tcp server
    eb.send(INTERNAL_TCP_SERV_ADDR, msg.getString("body"));
    return true; //-- true To allow the send/publish to occur,
false otherwise
}

@Override
public boolean handlePreRegister(SockJSocket sock, String address) {
    logger.info("handlePreRegister, sock = " + sock + ", address =
" + address);
    return true; //-- true to let the registration occur, false
otherwise
}

@Override
public void handlePostRegister(SockJSocket sock, String address) {
    logger.info("handlePostRegister, sock = " + sock + ", address =
" + address);
    Integer counter = addresses.get(address);
    if (counter == null) {
        counter = 1;
}

```

```

        }
    else {
        ++counter;
    }
logger.info("address: "+address+, counter: "+counter);
addresses.put(address, counter);
}

@Override
public boolean handleUnregister(SockJSocket sock, String address) {
    logger.info("handleUnregister, sock = " + sock + ", address = "
+ address);
    Integer counter = addresses.get(address);
    if (counter != null) {
        if (--counter == 0) {
            addresses.remove(address);
        }
        else {
            addresses.put(address, counter);
        }
    }
    logger.info("address: "+address+, counter: "+counter);
    return true;
}

@Override
public boolean handleAuthorise(JSONObject message, String
sessionID, Handler<AsyncResult<Boolean>> handler) {
    return false; //-- true if you wish to override authorisation
}
});

JSONObject sockJSconfig = new JSONObject();
sockJSconfig.putString("prefix", appConfig.getString("prefix",
"/mySockJS"));

```

```

//-- bridge perms(allow everytings)
JSONArray permitted = new JSONArray();
permitted.add(new JSONObject());

sockJSServer.bridge(sockJSconfig, permitted, permitted);

httpServer.listen(port, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: "+asyncResult.succeeded());
    }
});

//-- event bus subscribe
eb.registerHandler(INTERNAL_SOCKJS_SERV_ADDR, new
Handler<Message<Buffer>>() {
    @Override
    public void handle(final Message<Buffer> message) {
        Iterator<String> keys = addresses.keySet().iterator();
        while (keys.hasNext()) {
            eb.publish(keys.next(), message.body().toString());
        }
    }
});

@Override
public void stop() {
    if (httpServer != null)
        httpServer.close();
}
}

```

IntegratedSockJSEbbChatServerVerticle에서도 IntegratedTCPChatServerVerticle과 마찬가지로 INTERNAL_TCP_SERV_ADDR과 INTERNAL_SOCKJS

_SERV_ADDR이라는 두 개의 상수를 정의한다. 이들 상수 값은 IntegratedTCP ChatServerVerticle에서 정의한 값과 같다.

IntegratedSockJSebbChatServerVerticle에서 주의 깊게 볼 부분은 SockJS EBB 채팅 클라이언트가 전송한 메시지를 IntegratedTCPChatServerVerticle로 전달하는 EventBusBridgeHook의 handleSendOrPub() 메서드 부분이다. 이 메서드는 SockJS EBB 채팅 클라이언트가 전송한 메시지를 다른 SockJS EBB 채팅 클라이언트에 전달하기 전에(send, publish 모두 해당) 실행되는 Hook 메서드다. 이 메서드가 false를 반환하면 해당 메시지는 서버에서 버려진다.

NOTE

[코드 3-7]의 handleSocketData() 메서드를 살펴보면, send와 publish를 처리할 때 internalHandleSendOrPub() 메서드를 호출하는 것을 확인할 수 있다. internalHandleSendOrPub() 메서드는 다시 handleSendOrPub() 메서드를 호출하여 Hook 메서드를 호출한다. 이 Hook 메서드의 반환값에 따라 메시지를 다른 SockJS EBB 채팅 클라이언트에 전달할지 결정하게 된다.

```
private void internalHandleSendOrPub(SockJSocket sock, boolean send,
    JSONObject msg, String address) {
    if (handleSendOrPub(sock, send, msg, address)) {
        doSendOrPub(send, sock, address, msg);
    }
}
protected boolean handleSendOrPub(SockJSocket sock, boolean send, JSONObject
    msg, String address) {
    if (hook != null) {
        return hook.handleSendOrPub(sock, send, msg, address);
    }
}
```

```
    return true;
}
```

IntegratedSockJSEbbChatServerVerticle은 이벤트 버스로 IntegratedTCPChatServerVerticle에서 전달하는 메시지를 수신하고, 해당 메시지를 SockJS EBB 채팅 클라이언트를 통해 활성화된 모든 이벤트 버스 주소에 브로드캐스팅한다. 활성화된 이벤트 버스 주소를 파악하는 데는 EventBusBridgeHook의 handlePostRegister() 메서드와 handleUnregister() 메서드를 사용한다. 이 메서드는 이름에서 의미하는 것처럼 SockJS EBB 채팅 클라이언트가 특정 이벤트 버스 주소를 Subscribe하거나 Desubscribe할 때 호출되는 메서드다. 이 두 메서드를 통해 이벤트 버스 주소별로 Subscribe된 SockJS EBB 채팅 클라이언트의 수를 파악하고, 클라이언트의 수가 0이라면 해당 이벤트 버스는 비활성화되었다고 판단한다.

통합 TCP/SockJS EBB 채팅 서비스가 제대로 동작하는지 확인하기 위해 [코드 4-3] IntegratedChatServerStarterVerticle을 작성하고 실행해 보자. 테스트를 위한 TCP 채팅 클라이언트는 2.4.3 Header/Payload 분할 방식이 적용된 [코드 2-10] TCPChatWithFramingClientVerticle을 사용한다.

[코드 4-3] 통합 TCP/SockJS EBB 채팅 서비스 동작 확인(IntegratedChatServerStarterVerticle.java)

```
public class IntegratedChatServerStarterVerticle extends Verticle {
    @Override
    public void start() {
        container.deployVerticle("IntegratedSockJSEbbChatServerVerticle.java");
        container.deployVerticle("IntegratedTCPChatServerVerticle.java");
    }
}
```

다음은 Windows 8의 PowerShell에서 IntegratedChatServerStarterVerticle을 실행시키고, TCP 채팅 클라이언트와 SockJS EBB 채팅 클라이언트를 실행한 다음 서로 메시지를 입력했을 때 IntegratedChatServerStarterVerticle의 출력 결과다. 출력 결과를 통해 ‘com.devop.vertx.ch4’라는 이벤트 버스 주소에 2개의 SockJS EBB 채팅 클라이언트가 연결된 것과 해당 주소로 브로드캐스팅(send=false라는 로그로 확인할 수 있다)하는 것을 확인할 수 있다.

```
PS C:\Users\yeon> vertx run IntegratedChatServerStarterVerticle.java
Succeeded in deploying verticle
bind result: true
bind result: true
handlePreRegister, sock = org.vertx.java.core.sockjs.impl.Session@23561566,
address = com.devop.vertx.ch4
handlePostRegister, sock = org.vertx.java.core.sockjs.impl.Session@23561566,
address = com.devop.vertx.ch4
address: com.devop.vertx.ch4, counter: 1
handlePreRegister, sock = org.vertx.java.core.sockjs.impl.Session@49ad8fcf,
address = com.devop.vertx.ch4
handlePostRegister, sock = org.vertx.java.core.sockjs.impl.Session@49ad8fcf,
address = com.devop.vertx.ch4
address: com.devop.vertx.ch4, counter: 2
handleSendOrPub, sock = org.vertx.java.core.sockjs.impl.Session@49ad8fcf, send
= false, address = com.devop.vertx.ch4
```

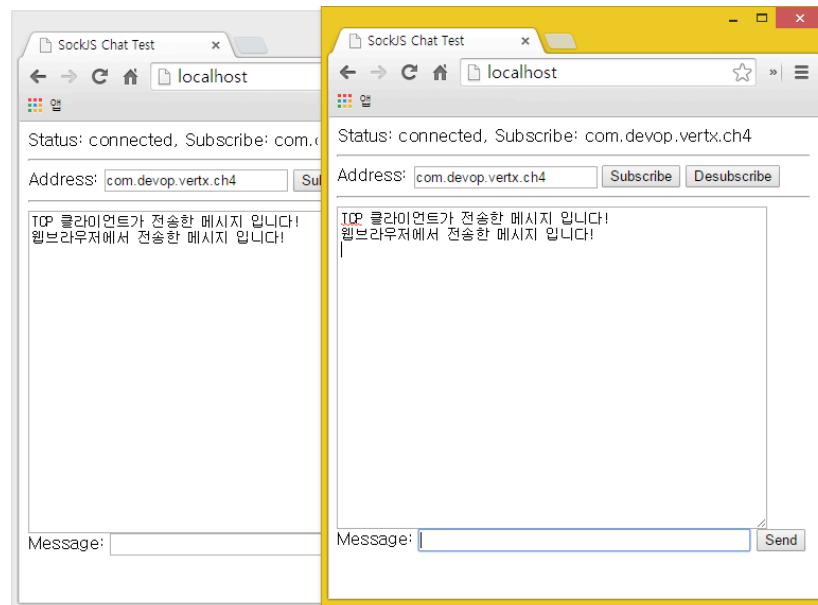
다음은 TCP 채팅 클라이언트를 실행한 다음 메시지를 전송하고 수신한 출력 결과다. SockJS EBB 채팅 클라이언트가 전송한 메시지를 수신하는 것을 확인할 수 있다.

```
PS C:\Users\yeon> vertx run TCPChatClientStarterVerticle.java
```

```
Succeeded in deploying verticle  
connect result: true  
TCP 클라이언트가 전송한 메시지입니다!  
received data: 웹브라우저에서 전송한 메시지입니다!
```

[그림 4-1]은 SockJS EBB 채팅 클라이언트의 출력 결과로, TCP 채팅 클라이언트가 전송한 메시지를 수신하는 것을 확인할 수 있다.

[그림 4-1] TCP/SockJS EBB 채팅 서버 통합 결과 화면



4.3 요약

지금까지 TCP/SockJS EBB 채팅 서비스를 통합하기 위한 내용을 알아보았다. 서비스 통합이라는 거창한 주제에 비해 간단하게 이벤트 버스로 TCP 채팅 서버와 SockJS EBB 채팅 서버가 서로 메시지를 주고받을 수 있도록 수정한 것이 전부다.

그러나 이것은 어디까지나 간단한 실습을 위한 예제며, 실 서비스 환경이라면 더
견고하고 효율적인 구현 방법을 고민해 봐야 한다.

다음 장에서는 Vert.x 모듈 시스템이 무엇인지 알아보고 TCP와 SockJS EBB 채팅
서비스 통합 결과를 모듈로 작성해 보자.

5 | Vert.x 모듈 시스템

지금까지 나온 예제에서 두 개 이상의 Verticle을 동시에 배포^{deploy}하기 위해 XXXStarterVerticle 같은 특수한 형태의 Verticle을 작성하고 실행하였다. 필요한 Verticle이나 기타 리소스 파일이 올바른 위치에 놓여 있지 않다면, 아마도 XXXStarterVerticle을 실행할 때 오류가 발생했을 것이다.

이런 불편을 해결하기 위해 실행에 필요한 한 개 이상의 Verticle과 관련 리소스를 한 곳으로 모으고 모듈화할 수 있다. 또한, 이렇게 작성된 모듈은 좀 더 편리하게 재사용할 수 있다는 장점이 있다.

특정 기능을 수행하는 모듈은 공개 모듈 저장소⁰¹를 찾아보면 가장 쉽게 얻을 수 있다. 공개 모듈 저장소는 자주 사용하는 모듈을 누구나 쉽게 찾을 수 있고, 재사용할 수 있도록 공개한 저장소다. 공개 모듈 저장소에서 필요한 모듈을 찾을 수 없다면, 직접 모듈을 작성해 사용할 수도 있다. 그리고 직접 작성한 모듈을 공개 저장소에 등록하고 공개하여 전 세계의 많은 개발자가 자신이 작성한 모듈을 이용하는 특별한 경험을 할지도 모른다([3.4.2 사용자 인증](#)에서 사용자 인증 정보를 처리하기 위해 mod-auth-mgr과 mod-mongo-persistor 모듈을 사용한 것을 떠올려 보자).

5.1 모듈 설치와 실행

모듈을 설치하려면 먼저 사용하려는 모듈의 정확한 이름을 알아야 한다. 모듈 공개 저장소의 모든 모듈은 고유한 이름이 있는데, 모듈의 이름은 Owner~Name~Version 순의 일정한 규칙에 따라 짓는다. 예를 들어, io.vertx~mod-mongo-persistor~2.1.0와 io.vertx~mod-web-server~2.0.0-final은 실제 Vert.x 공개

01 <http://modulereg.vertx.io/>

저장소에 있는 모듈로, 각각 MongoDB 접근과 간단한 HTTP 서버를 다루는 데 사용된다.

[표 5-1] 모듈 명명 규칙

구분	설명
소유자(Owner)	모듈의 소유자를 의미한다. 보통 Java 패키지명을 역순으로 나열한 문자열 형태로 사용한다.
모듈명(Name)	모듈의 이름이다. 따로 이름을 부여하는 규칙은 없다.
버전-Version)	모듈의 버전이다. 따로 버전명을 부여하는 규칙은 없다.

모듈 이름을 확인하면 시스템에 모듈을 설치할 수 있는데, 모듈을 설치하는 방법은 두 가지가 있다.

첫 번째 방법은 콘솔에 ‘vertx install 모듈명’ 명령을 입력하는 것이다. 다음은 Windows 8의 PowerShell에서 ‘vertx install’ 명령을 실행한 결과다. 현재 시스템에 모듈이 설치되어 있지 않으면 모듈을 자동으로 내려받아 설치하고, 모듈이 시스템에 이미 설치되어 있으면 아무런 작업도 하지 않는다.

```
PS C:\Users\yeon> vertx install io.vertx~mod-mongo-persistor~2.1.0
Attempting to install module io.vertx~mod-mongo-persistor~2.1.0
Downloading io.vertx~mod-mongo-persistor~2.1.0. Please wait...
Downloading 100%
Module io.vertx~mod-mongo-persistor~2.1.0 successfully installed
Succeeded in installing module
```

설치된 모듈은 명령을 실행한 경로의 mods 디렉터리에서 찾을 수 있다. 모듈을 시스템 전역에서 공통으로 사용하려면 Vert.x 설치 경로의 sys-mods 디렉터리에 복사하거나 VERTX_MODS라는 환경 변수에 지정된 디렉터리에 복사하면 된다.

모듈을 설치하는 다른 방법은 모듈을 배포/실행하면서 자동으로 설치하는 것인데, 콘솔에 ‘vertx runMod 모듈명’ 명령을 입력하거나 프로그램에서 ‘container

.deployModule("모듈명")' 코드로 모듈을 배포/실행할 수 있다. 모듈 배포/실행을 시도하면 시스템에서 사용 가능한 모듈을 찾기 위해 현재 실행 경로의 mods 디렉터리, VERTX_MODS 환경 변수 디렉터리, Vert.x 설치 경로의 sys-mods 디렉터리를 검색하고, 최종으로 사용 가능한 모듈이 현재 시스템에 없으면 모듈 공개 저장소에서 모듈을 자동으로 내려받아 설치한다. 이렇게 설치된 모듈은 앞의 방법과 마찬가지로 명령을 실행한 경로의 mods 디렉터리에서 찾을 수 있다.

다음은 Windows8의 PowerShell에서 'vertx runMod' 명령을 실행한 결과다.

```
PS C:\Users\yeon> vertx runMod io.vertx~mod-mongo-persistor~2.1.0
Downloading io.vertx~mod-mongo-persistor~2.1.0. Please wait...
Downloading 100%
Module io.vertx~mod-mongo-persistor~2.1.0 successfully installed
Succeeded in deploying module
```

5.2 모듈 디스크립터

Vert.x 모듈은 mod.json이라는 이름의 특수한 파일 하나를 반드시 포함해야 한다. 이 파일을 모듈 디스크립터Descriptor라고 하는데, 해당 모듈에 대한 각종 실행 설정, 의존성 정보, 작성자 정보 등을 포함하는 JSON 객체 파일이다.

모듈 디스크립터는 모듈의 루트 디렉터리에 있으며, 모듈 디스크립터의 위치를 기준으로 각종 리소스의 위치를 파악한다. 예를 들어, 모듈에서 사용하는 외부 라이브러리는 lib 디렉터리에 위치해야 한다. Verticle도 마찬가지로 모듈 디스크립터의 위치로부터 기술된다.

간단한 구성을 지닌 다음과 같은 모듈을 생각해 보자.

```
/mod.json
```

```
/com/devop/vertx/App.class  
/com/ devop / vertx /SomeOtherClass.class  
/lib/somelib.jar  
/lib/someotherlib.jar
```

모듈 디스크립터에서 App.class의 위치는 {"main": "com. devop. vertx.App"}로 표현된다(main은 실행 가능한 모듈의 최초 시작 Verticle을 의미한다. 모듈 디스크립터의 주요 설정 항목은 [표 5-2]를 참고하자). lib 디렉터리의 jar 라이브러리 파일들도 자동으로 해당 모듈의 classpath에 추가되며, 모듈 내부의 Verticle에서 해당 라이브러리를 사용할 수 있다.

[표 5-2] 모듈 디스크립터 주요 설정 항목⁰²

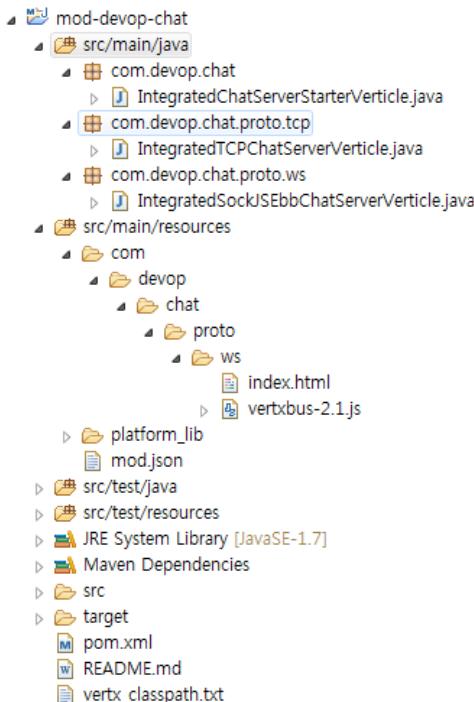
속성	설명
main	일반적으로 대부분 모듈은 실행 가능하며, 이는 곧 모듈 디스크립터에 main 속성이 정의되어 있다는 것을 뜻한다. main은 모듈이 배포될 때 최초 실행될 Verticle을 의미한다. 그러나 모든 모듈이 반드시 실행 가능한 것은 아니다. 이런 경우 모듈 디스크립터에 main 속성이 정의되지 않는다.
worker	이 값이 true면 해당 모듈이 Worker 스레드 풀에서 실행되어야 한다는 것을 의미한다. 이런 모듈은 스레드를 бл록시키는 API가 사용되고 있을 가능성성이 높다. 반대로 모듈이 이 벤트 루프 스레드 내에서 실행되어야 한다면 false를 입력한다. 기본값은 false다.
multi-threaded	이 값이 true면 모듈이 Worker 스레드 풀에서 실행될 때 2개 이상의 스레드에서 동시에 실행할 수 있다. 다중 스레드 문제를 야기할 수 있기 때문에 조심해서 다뤄야 한다. 기본값은 false다.
includes	모듈은 하나 이상의 다른 모듈을 포함할 수 있다. 특히 실행 불가능한 모듈을 포함하고, 해당 모듈이 포함하는 리소스에 접근하기 위해 이 속성을 유용하게 쓸 수 있다. 2개 이상의 모듈을 포함하려면 콤마로 구분한다.
auto-redeploy	이 값이 true면 모듈을 구성하는 Verticle이나 리소스가 변경되었을 때 해당 모듈을 자동으로 재배포(redeploy)한다. 개발 단계에서 매우 유용한 옵션으로, 기본값은 false다.
system	이 값이 true면 모듈을 설치할 때 해당 모듈이 Vert.x 설치 경로의 sys-mods 디렉터리에 설치되게 한다.

02 모듈 디스크립터의 모든 속성은 http://vertx.io/mods_manual.html#module-descriptor-file-modjson에서 확인할 수 있다.

5.3 Vert.x 모듈 만들기

Vert.x 모듈을 만들기 위해 1.3 Vert.x 설치 및 이클립스 환경 설정에서 소개한대로 이클립스 기반 Vert.x 프로젝트를 생성한 후(불필요한 파일은 삭제한다) 4장 TCP/SockJS EBB 채팅 서비스 통합의 Verticle 관련 리소스 파일을 프로젝트로 복사한다. 여기까지 완료되면 프로젝트 구조는 [그림 5-1]과 같다.

[그림 5-1] TCP와 SockJS 통합 채팅 서비스 프로젝트 구조



프로젝트를 구성하는 항목은 다음과 같다.

IntegratedChatServerStarterVerticle

채팅 서비스를 구동하기 위한 StarterVerticle로 [코드 5-1]과 같다. 단, 모듈내의

각 Verticle을 배포하기 위해 `container.deployVerticle()` 메서드 입력
파라미터 문자열 값이 다음과 같이 수정되었다.

```
container.deployVerticle("com.devop.chat.proto.ws.IntegratedSockJSEbbChat  
ServerVerticle");  
container.deployVerticle("com.devop.chat.proto.tcp.IntegratedTCPChatServer  
Verticle");
```

IntegratedTCPChatServerVerticle

TCP 채팅 클라이언트를 처리하기 위한 TCP 채팅 서버 Verticle로 [코드 4-1]과
동일하다.

IntegratedSockJSEbbChatServerVerticle

SockJS EBB 채팅 클라이언트를 처리하기 위한 SockJS EBB 채팅 서버 Verticle로
[코드 4-2]와 같다. 단, 모듈 안의 정적 웹 리소스(*.html, *.js)를 가리키기 위해
다음과 같이 `IntegratedSockJSEbbChatServerVerticle`안에 `getClasspath
ResourceFile()` 메서드가 추가되었고, `httpServer.requestHandler()` 메서드의
입력 파라미터 이벤트 핸들러 구현이 수정되었다.

```
private String getClasspathResourceFile(String filepath) {  
    try {  
        File file = new File(getClass().getResource(filepath).toURI());  
        return file.toString();  
    } catch (URISyntaxException e) {  
    }  
    return null;  
}  
// 중략  
httpServer.requestHandler(new Handler<HttpServerRequest>() {
```

```
final String indexHtml = getClasspathResourceFile("/com/devop/chat/
proto/ws/index.html");
final String vertxBusJs = getClasspathResourceFile("/com/devop/chat/
proto/ws//vertxbus-2.1.js");

@Override
public void handle(HttpServerRequest request) {
    if (request.path().equals("/")) request.response().sendFile(indexHtml);
    else if (request.path().endsWith("vertxbus-2.1.js")) request.response()
        .sendFile(vertxBusJs);
    else request.response().setStatusCode(404).end("Not Found Page");
}
});
```

index.html, vertxbus-2.1.js

SockJS EBB 채팅 서비스에 사용되는 정적 웹 리소스 파일이다. 이 파일은 src\main\java 디렉터리가 아닌 src\main\resources에 두어야 한다는 점을 주의해야 한다(*.java 파일은 제외한 모든 리소스 파일은 src\main\resources에 두어야 한다). 이 파일은 IntegratedSockJSEbbChatServerVerticle에서 참조한다.

mod.json

통합 채팅 서비스 모듈의 디스크립터다. 모듈이 배포될 때 최초 실행될 Verticle로 IntegratedChatServerStarterVerticle을 지정한다.

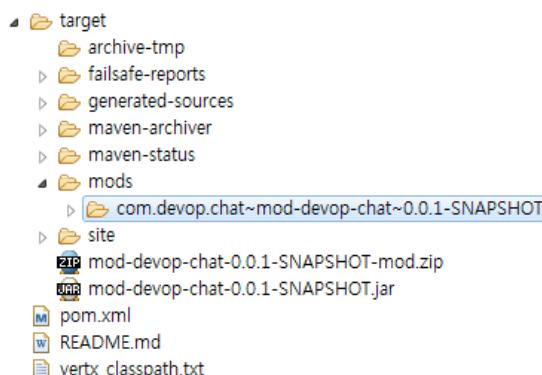
```
{
    "main": "com.devop.chat.IntegratedChatServerStarterVerticle",
    "description": "tcp and sockjs integrated chat service",
    "author": "Yeob-Bok Lee",
    "auto-redeploy": true
}
```

이제 이클립스에서 모듈을 빌드하고 실행해 보자. 프로젝트에서 ‘오른쪽 버튼 클릭 → Run As → Maven Build’ 또는 단축키 Alt+Shift+X, M를 선택하면 된다([그림 1-8] 참고). 프로젝트가 정상적으로 빌드되고 실행되면 이클립스 콘솔에 ‘bind result: true’라는 문자열이 2번 출력된다.

이클립스에서 빌드가 정상적으로 완료되면 [그림 5-2]처럼 target\mods 디렉터리에 모듈이 생성된 것을 확인할 수 있다. 생성된 모듈을 테스트하기 위해 콘솔을 열고 target 디렉터리로 이동한 다음 ‘vertx runMod’ 명령으로 모듈을 실행해 보자. 모듈이 정상적으로 실행되면 TCP 채팅 클라이언트와 웹 브라우저에서 채팅 메시지를 서로 전송할 수 있다.

```
PS C:\Users\yeon> cd C:\Dev\Projects\Java\my-work\mod-devop-chat\target
PS C:\Dev\Projects\Java\my-work\mod-devop-chat\target> vertx runMod
com.devop.chat~mod-devop-chat~0.0.1-SNAPSHOT
Succeeded in deploying module
bind result: true
bind result: true
```

[그림 5-2] TCP/SockJS 통합 채팅 서비스 모듈



5.4 몇 가지 유용한 Vert.x 모듈

Vert.x 공개 모듈 저장소에는 3.4.2 사용자 인증에서 간단히 살펴본 mod-auth-mgr과 mod-mongo-persistor 같은 유용한 여러 가지 모듈이 등록되어 있다. 모든 모듈을 살펴보기에는 무리가 있으므로 실제 프로젝트에 유용하게 활용할 수 있는 몇 가지 모듈을 간략히 살펴보겠다.

5.4.1 mod-auth-mgr

mod-auth-mgr⁰³은 사용자 인증 처리를 위한 모듈로, 아이디와 비밀번호로 사용자를 인증하고, 인증 결과로 만료시간이 정해진 세션 아이디를 생성한다. 이렇게 생성된 세션 아이디는 이벤트 버스를 통해 필요한 곳에 어디든 전달할 수 있다(SockJS EBB의 사용자 인증 보안 방식을 설정하면 기본 동작으로 mod-auth-mgr을 사용하는 것을 3.4.2 사용자 인증에서 확인했다). 참고로 이 모듈은 사용자 아이디와 비밀번호 저장소로 MongoDB를 사용하므로 이어서 소개할 mod-mongo-persistor에 의존성이 있다.

mod-auth-mgr의 설정 항목은 [표 3-8]에서 확인할 수 있다. 여기서는 이 모듈의 간단한 사용법을 알아보자.

[코드 5-1] mod-auth-mgr 예제(HTTPAuthMgrClientVerticle.java)

```
public class HTTPAuthMgrClientVerticle extends Verticle {
    private final static String AUTH_MGR_ADDRESS_PREFIX =
        "vertx.basicauthmanager";

    private Logger logger;
    private EventBus eb;
    HttpServer httpServer;
```

03 <https://github.com/vert-x/mod-auth-mgr>

```

private void deployAuthMgr() {
    JsonObject authMgrConfig = new JsonObject();
    authMgrConfig.putString("address", AUTH_MGR_ADDRESS_PREFIX);
    authMgrConfig.putString("user_collection", "users");
    authMgrConfig.putString("persistor_address", "vertx.mongopersistor");
    authMgrConfig.putNumber("session_timeout", 1800000); // --30min
    container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final",
    authMgrConfig);

    JsonObject mongoPersistorConfig = new JsonObject();
    mongoPersistorConfig.putString("address", "vertx.mongopersistor");
    mongoPersistorConfig.putString("host", "localhost");
    mongoPersistorConfig.putNumber("port", 27017);
    mongoPersistorConfig.putString("db_name", "my-chat");
    mongoPersistorConfig.putNumber("pool_size", 10);
    container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",
    mongoPersistorConfig);
}

final Handler<HttpServerRequest> requestHandler = new
Handler<HttpServerRequest>() {
    @Override
    public void handle(final HttpServerRequest request) {
        if (!"POST".equals(request.method())) {
            request.response().setStatusCode(405).end("Method Not
Allowed");
            return;
        }
        request.bodyHandler(new Handler<Buffer>() {
            @Override
            public void handle(Buffer buffer) {
                logger.info("received data: " + buffer.toString());
                String address = null;
                //-- login request
                //-- {"username": "myid", "password": "1234"}
                if ("/login".equals(request.path())) {

```

```

        address = AUTH_MGR_ADDRESS_PREFIX+".login";
    }
    //-- logout request
    //-- {"sessionID":"28488cf5-c909-4e99-a134c-7a25d3362986"}
    else if ("/logout".equals(request.path())) {
        address = AUTH_MGR_ADDRESS_PREFIX+".logout";
    }
    //-- authorise request
    //-- {"sessionID":"28488cf5-c909-4e99-a134c-7a25d3362986"}
    else if ("/authorise".equals(request.path())) {
        address = AUTH_MGR_ADDRESS_PREFIX+".authorise";
    }
    else {
        request.response().setStatusCode(404).end("Not Found Page");
    }
    if (address != null) {
        eb.send(address, new JsonObject(buffer.toString()), new
Handler<Message<JsonObject>>() {
            @Override
            public void handle(Message<JsonObject> reply) {
                request.response().setStatusCode(200).end(reply.
body().toString());
            }
        });
    }
}
});
}
};

@Override
public void start() {
    logger = container.logger();
    eb = vertx.eventBus();
    httpServer = vertx.createHttpServer();
}

```

```
deployAuthMgr();
httpServer.requestHandler(requestHandler);

httpServer.listen(80, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: " +asyncResult.succeeded());
    }
});
}
```

mod-auth-mgr은 로그인, 로그아웃, 인증(authorise)의 세 가지 기능을 제공한다. 각 기능은 mod-auth-mgr의 address 설정 문자열에 각 기능의 고유 문자열을 더한 결과를 이벤트 버스 주소로 사용하여 호출할 수 있다. 예를 들어, 로그인은 'vertx.basicauthmanager.login'이 된다. 로그인 요청은 사용자 아이디와 비밀번호를 다음의 JSON 객체로 만들고 호출한다.

```
{"username": <username>, "password": <password>}
```

로그인 결과는 성공과 실패로 나누어지는데, 성공하면 생성된 세션 아이디가 결과에 포함된다. 세션 아이디의 유효시간은 session_timeout 설정값을 따르며, 기본값은 30분이다.

```
Login 성공 : {"status": "ok", "sessionID": <session_id>}
Login 실패 : {"status": "denied"}
```

로그아웃과 인증 요청은 세션 아이디를 파라미터로 입력한다. 로그아웃은 의미 그대로 세션 아이디를 폐기하고 더는 사용할 수 없게 만든다. 인증 요청은 해당

세션 아이디를 검증하여 세션 아이디가 유효하다면 해당 세션 아이디를 소유한 사용자 아이디와 함께 출력한다.

Authorise 성공 : {"username":"myid","status":"ok"}

Authorise 실패 : {"status": "denied"}

mod-auth-mgr을 테스트하기 전에 MongoDB를 실행하는 것을 잊지 말자.

5.4.2 mod-mongo-persistor

mod-mongo-persistor⁰⁴는 Vert.x에서 MongoDB를 다루기 위한 모듈이다. SAVE, UPDATE, FIND, DELETE 등 기본적인 CRUD 연산은 물론, MongoDB의 고유 명령도 대부분 지원한다.

mod-mongo-persistor의 설정 항목은 [표 3-8]에서 확인할 수 있다. 여기서는 이 모듈의 간단한 사용법을 알아보자. [코드 5-2]는 find, save, update, delete의 기본 네 가지 명령을 다루고 있다.

[코드 5-2] mod-mongo-persistor 예제(HTTPMongoPersistorClientVerticle.java)

```
public class HTTPMongoPersistorClientVerticle extends Verticle {  
    private final static String MONGO_PERSISTOR_ADDRESS = "vertx.mongopersistor";  
  
    private Logger logger;  
    private EventBus eb;  
    HttpServer httpServer;  
  
    private void deployMongoPersistor() {  
        JsonObject mongoPersistorConfig = new JsonObject();  
        mongoPersistorConfig.putString("address", MONGO_PERSISTOR_ADDRESS);  
        mongoPersistorConfig.putString("host", "localhost");  
    }  
}
```

04 <https://github.com/vert-x/mod-mongo-persistor>

```

        mongoPersistorConfig.putNumber("port", 27017);
        mongoPersistorConfig.putString("db_name", "my-chat");
        mongoPersistorConfig.putNumber("pool_size", 10);
        container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",
mongoPersistorConfig);
    }

    private String getParamValue(MultiMap params, String name) {
        for(Map.Entry<String, String> entry : params.entries()) {
            if (entry.getKey().equals(name))
                return entry.getValue();
        }
        return null;
    }

    private JsonObject find(String username) {
        JsonObject q = new JsonObject();
        q.putString("action", "find");
        q.putString("collection", "users");
        q.putObject("matcher", new JsonObject().putString("username",
username));
        return q;
    }

    private JsonObject save(String username, String password) {
        JsonObject data = new JsonObject();
        data.putString("action", "save");
        data.putString("collection", "users");
        data.putObject("document", new JsonObject()
.putString("username", username).putString("password", password));
        return data;
    }

    private JsonObject update(String username, String password) {
        JsonObject data = new JsonObject();
        data.putString("action", "update");

```

```

        data.putString("collection", "users");
        data.putObject("criteria", new JsonObject().putString("username",
username));
        data.putObject("objNew", new JsonObject()
.putObject("$set", new JsonObject().putString("password",
password)));
        data.putBoolean("upsert", false);
        data.putBoolean("upsert", false);
        return data;
    }

    private JsonObject delete(String username) {
        JsonObject q = new JsonObject();
        q.putString("action", "delete");
        q.putString("collection", "users");
        q.putObject("matcher", new JsonObject().putString("username",
username));
        return q;
    }

    final Handler<HttpServerRequest> requestHandler = new
Handler<HttpServerRequest>() {
    @Override
    public void handle(final HttpServerRequest request) {
        String path = request.path();
        String method = request.method();
        MultiMap params = request.params();

        if ( !"/users".equals(path) )
            request.response().setStatusCode(404).end("Not Found Page");
        return;
    }
    JsonObject mongoOp = null;

    if ("GET".equals(method)) {
        String username = getParamValue(params, "username");

```

```

        if (username != null)
            mongoOp = find(username);
    }
    else if ("POST".equals(method)) {
        String username = getParamValue(params, "username");
        String password = getParamValue(params, "password");
        if (username!=null &&password!=null)
            mongoOp = save(username, password);
    }
    else if ("PUT".equals(method)) {
        String username = getParamValue(params, "username");
        String password = getParamValue(params, "password");
        if (username!=null &&password!=null)
            mongoOp = update(username, password);
    }
    else if ("DELETE".equals(method)) {
        String username = getParamValue(params, "username");
        if (username != null)
            mongoOp = delete(username);
    }
    else {
        request.response().setStatuscode(405).end("Method Not
Allowed");
    }
    if (mongoOp !=null) {
        eb.send(MONGO_PERSISTOR_ADDRESS, mongoOp, new
Handler<Message<JsonObject>>() {
            @Override
            public void handle(Message<JsonObject> reply) {
                request.response().setStatuscode(200).end(reply.body()
.toString());
            }
        });
    }
}

```

```

};

@Override
public void start() {
    logger = container.logger();
    eb = vertx.eventBus();
    httpServer = vertx.createHttpServer();

    deployMongoPersistor();
    httpServer.requestHandler(requestHandler);

    httpServer.listen(80, new Handler<AsyncResult<HttpServer>>() {
        @Override
        public void handle(AsyncResult<HttpServer> asyncResult) {
            logger.info("bind result: " + asyncResult.succeeded());
        }
    });
}
}

```

먼저 테스트를 위해 MongoDB을 실행한다. MongoDB에 미리 users 컬렉션을 생성하고, [그림 5-3]처럼 username와 password 항목이 있는 간단한 document를 입력해 두었다.

[그림 5-3] MongoDB users 컬렉션

		Document
0 { .. }	_id 53f5c7bcd88335a4d1bfa67 username myid password 1234	ObjectId String String
(1) { .. }	_id 3af03561-469e-4faf-8377-ab91c87e1b08 username myid1 password 1234	Document String String String

find 명령 테스트를 위해 웹 브라우저 주소에 ‘<http://localhost/users?username=myid>’를 입력해 보자. 다음 결과가 출력된다.

```
{"results": [
  {"_id": {"$oid": "53f5c7bcd88335a4d1bfa67"}, "username": "myid", "password": "1234"}
],
"status": "ok", "number": 1}
```

username=myid에 해당하는 document가 존재하지 않으면 number 값이 0이 되고 results는 빈 배열이 된다.

save 명령 테스트를 위해 URL '<http://localhost/users?username=myid2&password=1234>'로 POST 요청을 생성해 보자. 다음 결과가 출력된다.

```
{ "_id": "47ac2ea8-68a3-4c0e-88cd-617720ee4c0f", "status": "ok" }
```

users 컬렉션을 조회해 보면 username=myid2에 해당하는 document가 생성되어 있다. 실패하면 status는 error가 되고 message에 오류 메시지가 담겨 출력된다.

update 명령 테스트를 위해 URL '<http://localhost/users?username=myid2&password=123>'으로 PUT 요청을 생성해 보자. 다음 결과가 출력된다.

```
{"number": 1, "status": "ok" }
```

여기서 number는 변경에 영향을 받은 document의 개수를 의미한다. 따라서 변경된 document가 없다면 number 값은 0이 된다.

마지막으로 delete 명령 테스트를 위해 URL '<http://localhost/users?username=myid2>'로 DELETE 요청을 생성해 보자. 출력 결과는 update 명령과 같다. 여기서 number는 삭제된 document의 개수를 의미한다.

mod-mongo-persistor는 이 예제 외에도 다양한 명령을 지원하고 있다. Vert.x와 MongoDB를 함께 사용할 일이 있다면 공식 홈페이지를 꼭 확인해 보기 바란다.

5.4.3 mod-mysql-postgresql

mod-mysql-postgresql⁰⁵은 Vert.x에서 MySQL 또는 PostgreSQL을 다루기 위한 모듈이다. 이와 유사한 모듈로 mod-jdbc-persistor가 있으나 이 모듈은 표준 JDBC API를 사용해서 필연적으로 스레드 블록을 유발한다. 따라서 반드시 Worker 스레드 풀을 통해 실행해야 하는 제약이 있다. 이에 비해 mod-mysql-postgresql은 완전한 비동기 API로 처리되어서 Worker 스레드 풀에서 실행할 필요가 없다.

mod-mysql-postgresql은 Scala로 구현되어 추가로 Scala Language 모듈이 필요하고 Vert.x는 버전 2.1 이상이어야 한다.

NOTE

mod-mysql-postgresql을 배포하는 도중 ‘java.lang.ClassNotFoundException: org.vertx.scala.core.VertxAccess\$class’라는 오류가 발생하면 Vert.x의 설치 경로에서 conf/langs.properties 파일의 scala 항목을 다음과 같이 수정한다.

```
scala=io.vertx~lang-scala_2.10~1.1.0-M1:org.vertx.scala.platform.impl.Scala
VerticleFactory
```

Scala Language 모듈은 sys-mods에 설치된다. mod-mysql-postgresql을 배포하는 도중 ‘not found’ 오류가 발생할 수 있는데, 이 경우 conf/repos.txt 파일의 Sonatype snapshots 경로 프로토콜을 http에서 https로 수정한다.⁰⁶

05 <https://github.com/vert-x/mod-mysql-postgresql>

mod-mysql-postgresql의 설정 항목은 다음과 같다.

[표 5-3] mod-mysql-postgresql 모듈 파라미터

파라미터	설명
address	mod-mysql-postgresql의 이벤트 버스 주소다.
connection	데이터베이스의 종류를 입력한다. PostgreSQL 또는 MySQL을 입력한다. 기본값은 PostgreSQL이다.
host, port	데이터베이스의 호스트와 포트를 입력한다.
maxPoolSize	데이터베이스 커넥션 풀의 최대 크기를 입력한다. 기본값은 100이다.
username	데이터베이스의 사용자 계정 아이디를 입력한다.
password	데이터베이스의 사용자 계정 비밀번호를 입력한다.
database	사용할 데이터베이스를 지정한다. 기본값은 test다.

이 모듈의 간단한 사용법을 알아보자. [코드 5-3]은 select, insert, update, delete의 기본 4가지 명령을 다루고 있다.

[코드 5-3] mod-mysql-postgresql 예제(HTTPMySQLPersistorClientVerticle.java)

```
public class HTTPMySQLPersistorClientVerticle extends Verticle {  
    private final static String MYSQL_PERSISTOR_ADDRESS =  
    "vertx.mysql.persistor";  
  
    private Logger logger;  
    private EventBus eb;  
    HttpServer httpServer;  
  
    private void deployMySQLPersistor() {  
        JsonObject mysqlConf = new JsonObject();  
        mysqlConf.putString("address", MYSQL_PERSISTOR_ADDRESS);  
        mysqlConf.putString("connection", "MySQL");  
        mysqlConf.putString("host", "localhost");  
        mysqlConf.putNumber("port", 3306);  
        mysqlConf.putString("username", "root");  
    }  
}
```

06 <https://github.com/vert-x/mod-mysql-postgresql/issues/35>

```

        mysqlConf.putString("password", "mypassword");
        mysqlConf.putString("database", "mydb");
        container.deployModule("io.vertx~mod-mysql-postgresql_2.10~0.3.1",
mysqlConf);
    }

    private String getParamValue(MultiMap params, String name) {
        for(Map.Entry<String, String> entry : params.entries()) {
            if (entry.getKey().equals(name))
                return entry.getValue();
        }
        return null;
    }

    private JsonObject select(String username) {
        JsonObject sql = new JsonObject();
        sql.putString("action", "prepared");
        sql.putString("statement", "select * from users where username=?");
        JSONArray values = new JSONArray();
        values.addString(username);
        sql.putArray("values", values);
        return sql;
    }

    private JsonObject insert(String username, String password) {
        JsonObject sql = new JsonObject();
        sql.putString("action", "prepared");
        sql.putString("statement", "insert into users values(?,?)");
        JSONArray values = new JSONArray();
        values.addString(username);
        values.addString(password);
        sql.putArray("values", values);
        return sql;
    }

    private JsonObject update(String username, String password) {

```

```

        JsonObject sql = new JsonObject();
        sql.putString("action", "raw");
        sql.putString("command", "update users set password='"+password+"'
where username='"+username+"'");
        return sql;
    }

    private JsonObject delete(String username) {
        JsonObject sql = new JsonObject();
        sql.putString("action", "raw");
        sql.putString("command", "delete from users where
username='"+username+"'");
        return sql;
    }

    final Handler<HttpServerRequest> requestHandler = new
Handler<HttpServerRequest>() {
    @Override
    public void handle(final HttpServerRequest request) {
        String path = request.path();
        String method = request.method();
        MultiMap params = request.params();

        if ( !"/users".equals(path) )
            request.response().setStatusCode(404).end("Not Found Page");
        return;
    }
    JsonObject mysqlOp = null;

    if ("GET".equals(method)) {
        String username = getParamValue(params, "username");
        if (username != null)
            mysqlOp = select(username);
    }
    else if ("POST".equals(method)) {
        String username = getParamValue(params, "username");

```

```

        String password = getParamValue(params, "password");
        if (username!=null &&password!=null)
            mysqlOp = insert(username, password);
    }
    else if ("PUT".equals(method)) {
        String username = getParamValue(params, "username");
        String password = getParamValue(params, "password");
        if (username!=null &&password!=null)
            mysqlOp = update(username, password);
    }
    else if ("DELETE".equals(method)) {
        String username = getParamValue(params, "username");
        if (username != null)
            mysqlOp = delete(username);
    }
    else {
        request.response().setStatusCode(405).end("Method Not Allowed");
    }
    if (mysqlOp !=null) {
        eb.send(MYSQL_PERSISTOR_ADDRESS, mysqlOp, new
Handler<Message<JsonObject>>() {
            @Override
            public void handle(Message<JsonObject> reply) {
                request.response().setStatusCode(200).end(reply.body().
toString());
            }
        });
    }
};

@Override
public void start() {
    logger = container.logger();
    eb = vertx.eventBus();
}

```

```
httpServer = vertx.createHttpServer();

deployMySQLPersistor();
httpServer.requestHandler(requestHandler);

httpServer.listen(80, new Handler<AsyncResult<HttpServer>>() {
    @Override
    public void handle(AsyncResult<HttpServer> asyncResult) {
        logger.info("bind result: " + asyncResult.succeeded());
    }
});
}
```

기본 구조는 [코드 5-2] mod-mongo-persistor 예제와 동일하다.

Select 명령을 테스트하기 위해 웹 브라우저 주소에 '<http://localhost/users?username=myid>'를 입력해 보자. 다음 결과가 출력된다.

```
{"message":null,"rows":1,"fields":["username","password"],"results":[{"myid": "1234"}],"status":"ok"}
```

username=myid에 해당하는 row가 존재하지 않으면 rows 값은 0이 되고, results는 빈 배열이 된다.

insert 명령을 테스트하기 위해 URL '<http://localhost/users?username=myid2&password=1234>'로 POST 요청을 생성해 보자. 다음 결과가 출력된다.

```
{"message":"","rows":1,"status":"ok"}
```

users 테이블을 조회해 보면 username=myid2에 해당하는 row가 생성되어 있다.

실패하면 status는 error가 되고, message에 오류 메시지가 담겨 출력된다.

```
{"status":"error",
 "message":"Error 1062 - #23000 - Duplicate entry 'myid2' for key 'PRIMARY'", "e
rror":"MODULE_EXCEPTION"}
```

update 명령을 테스트하기 위해 URL '<http://localhost/users?username=myid2&password=123>'로 PUT 요청을 생성해 보자. 다음 결과가 출력된다.

```
{"message":"(Rows matched: 1  Changed: 1  Warnings: 0)", "rows":1, "status":"ok"}
```

Changed는 변경에 영향을 받은 row의 개수를 의미한다. 따라서 변경된 row가 없다면 Changed 값은 0이 된다.

마지막으로 delete 명령을 테스트하기 위해 URL '<http://localhost/users?username=myid2>'로 DELETE 요청을 생성해 보자. 출력은 insert 명령의 출력 결과와 같다. rows는 삭제된 row의 개수를 의미한다.

mod-mysql-postgresql는 이 예제 외에도 raw action을 통해 MySQL(또는 PostgreSQL)의 고유 명령을 지원하며, 트랜잭션 처리를 위한 commit과 rollback 등의 명령도 지원한다. Vert.x와 MySQL(또는 PostgreSQL)을 함께 사용할 일이 있으면 공식 홈페이지에서 꼭 확인해 보기 바란다.

5.4.4 mod-mailer

mod-mailer⁰⁷는 SMTP 프로토콜을 다루기 위한 간단한 메일 발송 클라이언트 모듈로, 이 모듈로 Gmail 등의 계정을 사용해 메일 발송을 처리할 수 있다. mod-

07 <https://github.com/vert-x/mod-mailer>

mailer의 설정 항목은 다음과 같다.

[표 5-4] mod-mailer 모듈 파라미터

파라미터	설명
address	mod-mailer의 이벤트 버스 주소다.
host, port	SMTP 서버의 호스트와 포트를 입력한다.
ssl	SSL 사용 여부를 지정한다. Gmail은 true를 입력한다.
auth	사용자 인증 여부를 지정한다. Gmail은 true를 입력한다.
username	SMTP 사용자 계정 아이디를 입력한다.
password	SMTP 사용자 계정 비밀번호를 입력한다.
content_type	메일 본문의 Content Type을 입력한다. 기본값은 text/plain이다. HTML 형식으로 메일 본문을 발송하려면 text/html을 입력한다.

이 모듈의 간단한 사용법을 알아보자. [코드 5-4]은 Gmail 계정으로 메일을 발송하는 예다.

[코드 5-4] mod-mailer 예제 (HTTPMailerClientVerticle.java)

```
public class HTTPMailerClientVerticle extends Verticle {  
    private final static String MAILER_ADDRESS = "vertx.mailer";  
  
    private Logger logger;  
    private EventBus eb;  
    HttpServer httpServer;  
  
    private void deployMailer() {  
        JsonObject mailerConf = new JsonObject();  
        mailerConf.putString("address", MAILER_ADDRESS);  
        mailerConf.putString("host", "smtp.googlemail.com");  
        mailerConf.putNumber("port", 465);  
        mailerConf.putBoolean("ssl", true);  
        mailerConf.putBoolean("auth", true);  
        mailerConf.putString("username", "yourmail@gmail.com");  
        mailerConf.putString("password", "yourpassword");  
        mailerConf.putString("content_type", "text/plain");  
    }  
}
```

```

        container.deployModule("io.vertx~mod-mailer~2.0.0-final", mailerConf);
    }

    final Handler<HttpServerRequest> requestHandler = new
    Handler<HttpServerRequest>() {
        @Override
        public void handle(final HttpServerRequest request) {
            if (!"POST".equals(request.method())) {
                request.response().setStatusCode(405).end("Method Not Allowed");
                return;
            }
            request.bodyHandler(new Handler<Buffer>() {
                @Override
                public void handle(Buffer buffer) {
                    logger.info("received data: " + buffer.toString());
                    JsonObject params = new JsonObject(buffer.toString());

                    JsonObject mail = new JsonObject();
                    mail.putString("from", "yourmail@gmail.com");
                    mail.putString("to", params.getString("to"));
                    mail.putString("subject", params.getString("subject"));
                    mail.putString("body", params.getString("body"));

                    eb.send(MAILER_ADDRESS, mail, new Handler<Message<JsonObject>>() {
                        @Override
                        public void handle(Message<JsonObject> reply) {
                            request.response().setStatusCode(200).end(reply.body
                                .toString());
                        }
                    });
                }
            });
        }
    };

    @Override

```

```
public void start() {
    logger = container.logger();
    eb = vertx.eventBus();
    httpServer = vertx.createHttpServer();

    deployMailer();
    httpServer.requestHandler(requestHandler);

    httpServer.listen(80, new Handler<AsyncResult<HttpServer>>() {
        @Override
        public void handle(AsyncResult<HttpServer> asyncResult) {
            logger.info("bind result: " + asyncResult.succeeded());
        }
    });
}
```

Gmail 계정 설정이 올바르지 않으면 다음의 오류 메시지가 출력된다.

```
Failed to setup mail transport
javax.mail.AuthenticationFailedException: 535-5.7.8 Username and Password not
accepted. Learn more at
535 5.7.8 http://support.google.com/mail/bin/answer.py?answer=14257
j11sm9279345pdk.76 - gsmtp
```

메일 발송을 테스트하기 위해 URL '<http://localhost>'로 POST 요청을 생성해보자. POST 요청 body에는 다음 JSON 파라미터를 추가한다.

```
{"to": "recvermail@gmail.com", "subject": "test mail", "body": "it body!"}
```

to는 메일 수신자(수신자 메일이 반드시 Gmail일 필요는 없다), subject는 메일 제목, body는 메일 본문이다. 메일 발송이 성공하면 다음 결과가 출력된다.

```
{"status": "ok"}
```

5.4.5 mod-web-server

지금까지 HTTP 서버를 다루기 위해 Verticle 내에 직접 `HttpServer` 객체를 생성하고 코드를 작성했다. 그러나 사실 간단한 HTTP 서버를 다루기 위한 훌륭한 모듈이 이미 있는데, 바로 `mod-web-server`⁰⁸라는 모듈이다.

이 모듈은 HTTP 처리는 물론 SSL 설정 처리, SockJS EBB 기능까지 통합하고 있어서 좀 더 편리하게 사용할 수 있다. HTTP 서버가 필요하다면 해당 모듈의 사용을 적극 고려해 보길 바란다.

08 <https://github.com/vert-x/mod-web-server>

6 | Advanced 채팅 서비스

지금까지 Vert.x의 여러 가지 특징과 이를 활용한 채팅 서비스 개발을 알아보았다. 그러나 앞서 작성한 TCP/SockJS EBB 채팅 서비스는 예제를 위한 수준에 머물러 있으며, 완전한 형태의 서비스라 할 수 없다. 사용자 인증 기능, 제어 메시지와 채팅 메시지 정의, 채팅 대화방 구분, 권속말 처리, 채팅 메시지 저장 등 실제 사용 가능한 수준의 채팅 서비스를 만들려면 더 많은 기능을 추가해야 한다.

필요한 모든 기능을 추가하는 것은 오랜 시간과 노력이 필요하다. 5장만으로 이 모든 내용을 다루는 것은 무리가 있으므로 실제 사용 가능한 수준에 가깝도록 채팅 서비스에 몇 가지 핵심 기능을 추가해 보겠다.

6.1 TCP 채팅 서버 로그인 기능

[3.4.2 사용자 인증](#)에서 SockJS EBB 채팅 서버에 사용자 인증 기능을 추가하는 방법을 알아보았다. 사용자 인증은 mod-auth-mgr로 처리되고(mod-auth-mgr은 [5.4.1 mod-auth-mgr](#) 참고) 이후 인증된 사용자는 세션 아이디로 신원을 증명하게 된다.

그러나 아직까지 TCP 채팅 서버에 사용자 인증 기능을 추가하는 방법은 알아보지 않았다. 그래서 [4장 TCP/SockJS EBB 채팅 서비스 통합](#)에서 사용자 인증 기능이 추가된 SockJS EBB 채팅 서버를 사용하지 않았다.

여기서는 SockJS EBB 채팅 서버와 동일하게 TCP 채팅 서버에도 mod-auth-mgr을 사용하여 사용자 인증 기능을 추가하고, 사용자 인증 기능이 추가된 TCP와 SockJS EBB 채팅 서버를 다시 통합해 본다.

6.1.1 TCP 채팅 서버 사용자 인증

SockJS EBB 채팅 서버에서의 사용자 인증 방법을 참고하여 TCP 채팅 서버에서도 사용자 인증 기능을 추가해 보자.

SockJS EBB 채팅 클라이언트는 mod-auth-mgr의 로그인 기능을 호출해 사용자 아이디와 비밀번호를 확인하고, 이 정보가 올바르면 응답 메시지에 포함되는 세션 아이디를 따로 저장해둔다. 이후 SockJS EBB 채팅 서버로 전송하는 모든 메시지는 세션 아이디를 포함한다.

SockJS EBB 채팅 서버에서는 메시지에 세션 아이디가 포함되어 있지 않거나 세션 아이디가 mod-auth-mgr의 인증 기능으로 확인되지 않을 때 해당 메시지를 다른 클라이언트로 전달하지 않고 즉시 폐기한다.

사용자의 메시지가 도착할 때마다 mod-auth-mgr의 인증 기능을 호출하는 것은 매우 비효율적이므로 SockJS EBB 채팅 서버는 일정 시간 동안 mod-auth-mgr의 인증 기능으로 인증된 세션 아이디를 캐시에 저장해 둔다.⁰¹ 인증된 세션 아이디가 캐시에 유지되는 최대 시간은 설정을 변경하지 않을 경우 5분이고 이 시간은 일반적으로 mod-auth-mgr에서 세션 아이디가 최대로 유지되는 30분을 넘지 않아야 한다.⁰² 물론 mod-auth-mgr에서 세션 아이디가 최대로 유지되는 시간은 설정을 변경해서 조절할 수 있다.

[코드 2-10] `TCPChatWithFramingServerVerticle`을 기본으로 일부 기능을 추가해 TCP 채팅 서버에서도 SockJS EBB 채팅 서버와 유사한 사용자 인증 기능을 구현해 보자. 사용자 인증 기능이 추가된 TCP 채팅 서버는 `TCPChatWith`

⁰¹ 좀 더 정확하게는 SockJS EBB 채팅 서버가 자체적으로 인증된 세션 아이디를 캐시하는 메커니즘을 구현하고 있는 것은 아니다. 해당 기능은 `org.vertx.java.core.sockjs.EventBusBridge`라는 특수한 이벤트 핸들러를 통해 이미 구현되어 있다.

⁰² mod-auth-mgr 모듈 또한 인증된 세션 아이디를 캐시하는 메커니즘을 사용하고 있다. 결국 인증된 세션 아이디는 1차로 mod-auth-mgr 모듈에서, 2차로는 SockJS EBB 채팅 서버에서 캐시된다

LoginServerVerticle이라 하자.

먼저 메시지의 Header와 Payload를 분할하기 위해 작성한 RecordParser 객체를 다음과 같이 수정한다. 기존의 RecordParser 객체는 메시지의 Payload 부분을 일반 문자열로 처리하지만, 수정된 RecordParser 객체는 메시지의 Payload 부분을 JSON 객체로 처리하는 것이 가장 큰 차이다.

```
final RecordParser framer = RecordParser.newFixed(4, null);
framer.setOutput(new Handler<Buffer>() {
    int payloadLength = -1;
    @Override
    public void handle(Buffer buffer) {
        if (payloadLength == -1) {
            payloadLength = buffer.getInt(0);
            framer.fixedSizeMode(payloadLength);
        }
        else {
            final JsonObject message = new JsonObject(buffer.toString());
            final String sessionID = message.getString("sessionID");
            //-- session id check
            if (sessionID != null) {
                authorise(message, sessionID, new AsyncResultHandler<Boolean>() {
                    @Override
                    public void handle(AsyncResult<Boolean> result) {
                        if (result.succeeded()) {
                            if (result.result()) {
                                cacheAuthorisation(sessionID, socket);
                                writeAll(message.getObject("body"), socket);
                            }
                        }
                    }
                });
            }
        }
    }
});
```

```

is not authorised");
        }
    }
}
else {
    logger.error("error in performing authorisation",
result.cause());
}
}
});
}
//-- session id null
else {
    eb.send(AUTH_MGR_ADDRESS_PREFIX+.login", message, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(Message<JsonObject> reply) {
        boolean authed = reply.body().getString("status").equal
s("ok");
        if (authed) {
            sockets.add(socket.writeHandlerID());
            setUsername(message.getString("username"), socket);
        }
        write(reply.body(), socket);
    }
});
}
framer.fixedSizeMode(4);
payloadLength = -1;
}
}
});

```

메시지의 Payload에서 세션 아이디의 존재를 확인하고, 세션 아이디가 없으면 해당 메시지를 그대로 mod-auth-mgr의 로그인 기능을 호출하는 데 사용한다. 올바른 로그인 요청이면 해당 사용자의 writeHandlerID를 sharedData를 통해 공유한다. 이것은 '- instances' 옵션으로 2개 이상의 TCPChatWithLoginServerVerticle이 배포되었을 때 임의의 TCPChatWithLoginServerVerticle에서 로그인된 모든 사용자에게 메시지를 브로드캐스팅하기 위해 필요하다. 그리고 로그인 결과로 출력된 세션 아이디는 TCP 채팅 클라이언트로 전달된다.

메시지의 Payload에 세션 아이디가 있으면 해당 세션 아이디가 올바른지 확인해야 한다. 세션 아이디는 authorize() 메서드로 확인한다. authorize() 메서드는 먼저 캐시로 세션 아이디가 올바른지 확인하고, 캐시에서 세션 아이디를 찾을 수 없으면 mod-auth-mgr의 인증 기능을 호출해 세션 아이디를 확인한다.

```
private void authorise(JsonObject message, String sessionID, final Handler<AsyncResult<Boolean>> handler) {
    final DefaultFutureResult<Boolean> res = new DefaultFutureResult<>();
    if (authCache.containsKey(sessionID)) {
        logger.debug("cache hit: "+sessionID);
        res.setResult(true).setHandler(handler);
    } else {
        logger.debug("cache miss: "+sessionID);
        eb.send(AUTH_MGR_ADDRESS_PREFIX+".authorise", message, new Handler<Message<JsonObject>>() {
            public void handle(Message<JsonObject> reply) {
                boolean authed = reply.body().getString("status").equals("ok");
                res.setResult(authed).setHandler(handler);
            }
        });
    }
}
```

authorize() 메서드 호출 결과로 세션 아이디에 문제가 없는지 확인하고, 캐시에 해당 세션 아이디가 없으면 cacheAuthorisation() 메서드는 세션 아이디를 캐시한다. 캐시된 세션 아이디는 일정 시간(authCacheTimeout 설정값을 따른다)이 경과한 후 자동으로 캐시에서 삭제되도록 uncacheAuthorisation() 메서드를 호출하는 타이머를 설정한다.

```
private class SocketInfo {
    private Set<String> sessions;
    private String writeHandlerID;
    private String username;
}

private class Auth {
    private final long timerID;
    Auth(final String sessionID, final NetSocket socket) {
        timerID = vertx.setTimer(authCacheTimeout, new Handler<Long>() {
            public void handle(Long id) {
                uncacheAuthorisation(sessionID, socket);
            }
        });
    }

    void cancel() {
        vertx.cancelTimer(timerID);
    }
}

private void cacheAuthorisation(String sessionID, NetSocket socket) {
    if (!authCache.containsKey(sessionID)) {
        logger.debug("cache: "+sessionID);
        authCache.put(sessionID, new Auth(sessionID, socket));
    }
    SocketInfo socketInfo = sockInfos.get(socket);
    Set<String> sess = socketInfo.sessions;
```

```
    if (sess == null) {
        sess = new HashSet<>();
        socketInfo.sessions = sess;
    }
    sess.add(sessionID);
}

private void uncacheAuthorisation(String sessionID, WebSocket socket) {
    logger.debug("uncache: "+sessionID);
    authCache.remove(sessionID);
    SocketInfo socketInfo = sockInfos.get(socket);
    Set<String> sess = socketInfo.sessions;
    if (sess != null) {
        sess.remove(sessionID);
        if (sess.isEmpty()) {
            socketInfo.sessions = null;
        }
    }
}
```

세션 아이디의 캐시 과정을 마치면 로그인된 모든 사용자에게 `writeAll()` 메서드로 메시지를 브로드캐스팅한다. 여기서 메시지 자체는 JSON 객체이고, 이를 `WebSocket`을 통해 다른 클라이언트로 전달할 때는 반드시 Header와 Payload 구조로 Buffer를 채워줘야 한다는 점을 주의해야 한다.

```
private void write(JSONObject message, WebSocket socket) {
    if (message == null) return;

    byte[] data;
    try {
        data = message.toString().getBytes("UTF-8");
        Buffer buffer = new Buffer(4+data.length);
        buffer.setInt(0, data.length);
```

```

        buffer.setBytes(4, data);
        socket.write(buffer);
    } catch (UnsupportedEncodingException e) {
    }
}

private void writeAll(JSONObject message, NetSocket exclude) {
    if (message == null) return;

    byte[] data;
    try {
        data = message.toString().getBytes("UTF-8");
        Buffer buffer = new Buffer(4+data.length);
        buffer.setInt(0, data.length);
        buffer.setBytes(4, data);
        for (String s : sockets) {
            if (exclude == null || !exclude.writeHandlerID().equals(s)) {
                eb.send(s, buffer);
            }
        }
    } catch (UnsupportedEncodingException e) {
    }
}

```

SocketInfo 객체는 클라이언트가 연결된 NetSocket의 부가정보를 저장하기 위해 사용된다. 부가정보는 해당 NetSocket에서 인증된 모든 세션 아이디와 writeHandlerID, 인증된 사용자 아이디를 포함한다. writeHandlerID와 사용자 아이디는 이후 굿속말 기능 구현에 사용되고, 인증된 모든 세션 아이디는 해당 NetSocket의 연결이 종료될 때 연관된 모든 타이머를 적절하게 취소시키는 데 사용된다.

```
socket.closeHandler(new VoidHandler() {
```

```
    @Override
    protected void handle() {
        sockets.remove(socket.writeHandlerID());
        //-- close any cached authorisations for this connection
        SocketInfo socketInfo = sockInfos.remove(socket);
        if (socketInfo != null) {
            Set<String> sses = socketInfo.sessions;
            if (sses != null) {
                for (String sessionID: sses) {
                    Auth auth = authCache.remove(sessionID);
                    if (auth != null)
                        auth.cancel();
                }
            }
        }
    });
}
```

마지막으로 사용자 인증 처리를 위해 mod-auth-mgr과 mod-mongopersistor 모듈을 적절한 설정을 적용하여 함께 배포한다(전체 코드는 이어서 설명할 [코드 6-1]과 별다른 차이가 없으므로 [코드 6-1]을 참고한다).

```
private void deployAuthMgr() {
    JsonObject authMgrConfig = new JsonObject();
    authMgrConfig.putString("address", AUTH_MGR_ADDRESS_PREFIX);
    authMgrConfig.putString("user_collection", "users");
    authMgrConfig.putString("persistor_address", "vertx.mongopersistor");
    authMgrConfig.putNumber("session_timeout", 1800000); //--30min
    container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final",
    authMgrConfig);

    JsonObject mongoPersistorConfig = new JsonObject();
    mongoPersistorConfig.putString("address", "vertx.mongopersistor");
```

```
        mongoPersistorConfig.putString("host", "localhost");
        mongoPersistorConfig.putNumber("port", 27017);
        mongoPersistorConfig.putString("db_name", "my-chat");
        mongoPersistorConfig.putNumber("pool_size", 10);
        container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",
        mongoPersistorConfig);
    }
```

6.1.2 TCP 채팅 클라이언트 사용자 인증

사용자 인증 기능을 추가한 TCP 채팅 서버에 대응해 [코드 2-4] TCPChatClientWorkerVerticle과 [코드 2-11] TCPChatWithFramingClientVerticle을 기본으로 일부 기능을 추가해 TCP 채팅 클라이언트에서도 SockJS EBB 채팅 클라이언트에서와 유사한 사용자 인증을 구현해 보자(사용자 인증 기능이 추가된 TCP 채팅 클라이언트는 TCPChatWithLoginClientVerticle과 TCPChatWithLoginClientWorkerVerticle이다).

먼저 TCPChatWithLoginClientWorkerVerticle의 반복 구문을 다음과 같이 변경한다.

```
@Override
public void start() {
    eb = vertx.eventBus();
    readline = true;

    Console console = System.console();
    if (console != null) {
        inputDelay(500);
        while (readline) {
            JsonObject input = null;
            if (isLoggedIn) {
                String line = console.readLine();
```

```

        if (line.equals("exit")) {
            readline = false;
            input = new JSONObject();
            input.putBoolean("exit", true);
        }
        else if (line.length() > 0) {
            input = new JSONObject();
            input.putObject("body", new JSONObject().putString("messag
e", line));
        }
    }
    else {
        isLoggedIn = true;
        System.out.print("input username: ");
        username = console.readLine();
        System.out.print("input password: ");
        password = console.readLine();

        input = new JSONObject();
        input.putString("username", username);
        input.putString("password", password);
    }

    if (input != null)
        eb.send("com.devop.vertx.chat", input);
}
}
}

```

변경된 반복 구문은 첫째, 최초 반복 구문 진입 시 사용자 아이디와 패스워드를 입력받아 mod-auth-mgr의 로그인 기능을 호출하는 데 사용될 수 있게 JSON 객체를 구성하고 둘째, 일반 채팅 메시지도 일정한 규칙으로 JSON 객체로 구성한 후 이벤트 버스에 전달되는 특징이 있다.

사용자 입력 메시지는 이벤트 버스를 통해 TCPChatWithLoginClientVerticle에 전달되는데, 사용자 인증 여부에 따라 세션 아이디와 사용자 아이디를 사용자 입력 메시지에 자동으로 포함한 후 TCP 채팅 서버로 해당 메시지를 전송한다.

```
eb.registerHandler("com.devop.vertx.chat", new Handler<Message<JsonObject>>() {
    @Override
    public void handle(Message<JsonObject> input) {
        if (writeHandlerID != null) {
            JsonObject body = input.body();
            if (body.getBoolean("exit", false)) {
                container.exit();
            }
            else {
                if (sessionId != null) {
                    body.getJsonObject("body").putString("username", username);
                    body.putString("sessionId", sessionId);
                }
                else {
                    if (username == null)
                        username = body.getString("username");
                }
                byte[] data;
                try {
                    data = body.toString().getBytes("UTF-8");
                    Buffer buffer = new Buffer(4+data.length);
                    buffer.setInt(0, data.length);
                    buffer.setBytes(4, data);
                    eb.send(writeHandlerID, buffer);
                } catch (UnsupportedEncodingException e) {
                }
            }
        }
    });
});
```

다음은 세션 아이디와 사용자 아이디가 포함된 일반 채팅 메시지의 최종 JSON 객체다.

```
{"body": {"username": <username>, "message": <message>}, "sessionID":  
<sessionID>}
```

TCP 채팅 서버는 클라이언트가 전달한 메시지가 로그인 요청인지 일반 채팅 메시지인지 세션 아이디 유무에 따라 구분하게 되고, 로그인 요청이라면 앞서 살펴본 것처럼 mod-auth-mgr의 로그인 기능을 호출하고, 그 결과를 TCP 채팅 클라이언트로 그대로 전달한다. 일반 채팅 메시지라면 세션 아이디를 확인하고, 문제가 없으면 메시지를 다른 TCP 채팅 클라이언트 모두에게 브로드캐스팅한다.

따라서 TCP 채팅 클라이언트가 서버로부터 수신할 수 있는 메시지는 사용자 로그인 요청의 결과와 다른 사용자가 전달한 일반 채팅 메시지로 구분되는데, 이 두 종류의 메시지를 처리하기 위한 RecordParser 객체를 다음과 같이 작성한다.

```
final RecordParser framer = RecordParser.newBuilder().setDecoder(new Decoder() {  
    @Override  
    public void decode(ByteBuf byteBuf) {  
        if (byteBuf.readableBytes() <= 0) {  
            return;  
        }  
        int payloadLength = byteBuf.readInt();  
        if (payloadLength > 0) {  
            byteBuf.skipBytes(payloadLength);  
        }  
        String sessionID = byteBuf.readText();  
        String message = byteBuf.readText();  
        if (sessionID != null) {  
            Map<String, String> map = new HashMap<>();  
            map.put("sessionID", sessionID);  
            map.put("username", message);  
            map.put("message", msg);  
            framer.setObject(map);  
        }  
    }  
});
```

```

        System.out.println(from+": "+msg);
    }
    else {
        boolean authed = message.getString("status").equals("ok");
        //-- login success
        if (authed) {
            sessionID = message.getString("sessionID");
            System.out.println("user '"+username+"' login success at
"+new Date());
        }
        //-- login failure
        else {
            System.out.println("user '"+username+"' login failure.
please enter exit.");
        }
    }
    framer.fixedSizeMode(4);
    payloadLength = -1;
}
}
});

```

로그인 요청이 정상적으로 처리되면 응답에는 세션 아이디가 포함되어 있다. 이 세션 아이디를 저장하고 이후 모든 채팅 메시지에 세션 아이디를 포함해서 TCP 채팅 서버로 전송한다.

다음은 Windows 8의 PowerShell에서 TCPChatWithLoginServerVerticle을 실행하고 2개의 TCP 채팅 클라이언트를 실행한 다음 로그인 요청이 정상적으로 처리되었을 때 출력된 결과다. 로그인된 다른 사용자가 채팅 메시지를 전달하면 해당 메시지를 작성한 사용자 로그인 아이디와 함께 출력되는 것도 확인할 수 있다.

```
PS C:\Users\yeon> vertx run TCPChatWithLoginClientVerticle.java
Succeeded in deploying verticle
connect result: true
input username: myid
input password: 1234
user 'myid' login success at Tue Nov 04 19:03:03 KST 2014
하이?
myid2: hello?
```

다음은 로그인에 실패했을 때 출력 결과다. 더는 진행이 불가능하며, ‘exit’를 입력하여 TCP 채팅 클라이언트를 종료시켜야 한다.

```
PS C:\Users\yeon> vertx run TCPChatWithLoginClientVerticle.java
connect result: true
Succeeded in deploying verticle
input username: myid
input password: 1345
user 'myid' login failure. please enter exit.
```

다음은 세션 아이디가 올바르지 않을 때 TCP 채팅 서버의 출력 결과다. 해당 채팅 메시지는 다른 클라이언트로 전달되지 않는다.

```
PS C:\Users\yeon> vertx run TCPChatWithLoginServerVerticle.java
bind result: true
Succeeded in deploying verticle
message rejected because sessionID is not authorised
```

6.1.3 사용자 인증 TCP/SockJS EBB 채팅 서비스 통합

TCP 채팅 서버에 사용자 인증 기능을 추가했으니 사용자 인증 기능이 추가된 SockJS EBB 채팅 서버([3.4.2 사용자 인증](#)의 SockJSEbbChatWithLoginServerVerticle)와 통합해 보자. 통합을 마치면 이전처럼 TCP/SockJS EBB 양쪽 채팅 클라이언트 모두가 채팅 메시지를 확인할 수 있고, 해당 채팅 메시지를 작성한 사용자 아이디를 함께 확인할 수 있다.

먼저 [6.1.1 TCP 채팅 서버 사용자 인증](#)에서 작성한 TCPChatWithLoginServer Verticle을 기본으로 일부 기능을 추가한다. 이는 TCP 채팅 클라이언트가 전송한 메시지를 이벤트 버스로 SockJS EBB 채팅 서버에 전달하는 기능과 SockJS EBB 채팅 서버에서 이벤트 버스를 통해 전달한 메시지를 수신해 TCP 채팅 클라이언트에 전송하는 기능이다. 이 기능은 [4장 TCP/SockJS EBB 채팅 서비스 통합](#)에서 추가한 것과 같은 역할을 한다(즉, TCP와 SockJS EBB 채팅 서버가 서로 메시지를 교환할 수 있도록 메커니즘을 추가하는 것이다). 단, TCP/SockJS EBB 양쪽 채팅 클라이언트에 전달되는 메시지에 사용자 아이디가 포함되도록 이벤트 버스를 통한 메시지 교환은 JSON 객체를 사용하도록 변경되었다.

[코드 6-1] 통합 로그인 TCP 채팅 서버(IntegratedTCPChatWithLoginServerVerticle.java)

```
public class IntegratedTCPChatWithLoginServerVerticle extends Verticle {
    public static final String INTERNAL_TCP_SERV_ADDR      =
"com.devop.vertx.chat.internal.tcp";
    public static final String INTERNAL_SOCKJS_SERV_ADDR = "com.devop.vertx.chat.internal.sockjs";

    private final static String AUTH_MGR_ADDRESS_PREFIX =
"vertx.basicauthmanager";
    private final static int LOCAL_AUTH_CACHE_TIMEOUT = 900000; //-- 15min

    private Logger logger;
    private EventBus eb;
```

```

private NetServer server;
private Set<String> sockets;

private long authCacheTimeout;
private final Map<String, Auth> authCache = new HashMap<>();
private final Map<NetSocket, SocketInfo> sockInfos = new HashMap<>();

private class SocketInfo {
    private Set<String> sessions;
    private String writeHandlerID;
    private String username;
}

private class Auth {
    private final long timerID;

    Auth(final String sessionID, final NetSocket socket) {
        timerID = vertx.setTimer(authCacheTimeout, new Handler<Long>() {
            public void handle(Long id) {
                uncacheAuthorisation(sessionID, socket);
            }
        });
    }

    void cancel() {
        vertx.cancelTimer(timerID);
    }
}

private void cacheAuthorisation(String sessionID, NetSocket socket) {
    if (!authCache.containsKey(sessionID)) {
        authCache.put(sessionID, new Auth(sessionID, socket));
    }
    SocketInfo socketInfo = sockInfos.get(socket);
    Set<String> sess = socketInfo.sessions;
    if (sess == null) {
        sess = new HashSet<>();
        socketInfo.sessions = sess;
    }
}

```

```

        }
        sess.add(sessionID);
    }

    private void uncacheAuthorisation(String sessionID, WebSocket socket) {
        authCache.remove(sessionID);
        SocketInfo socketInfo = sockInfos.get(socket);
        Set<String> sess = socketInfo.sessions;
        if (sess != null) {
            sess.remove(sessionID);
            if (sess.isEmpty()) {
                socketInfo.sessions = null;
            }
        }
    }

    private void authorise(JsonObject message, String sessionID,
        final Handler<AsyncResult<Boolean>> handler) {
        final DefaultFutureResult<Boolean> res = new DefaultFutureResult<>();
        if (authCache.containsKey(sessionID)) {
            res.setResult(true).setHandler(handler);
        }
        else {
            eb.send(AUTH_MGR_ADDRESS_PREFIX+".authorise", message, new
Handler<Message<JsonObject>>() {
                public void handle(Message<JsonObject> reply) {
                    boolean authed = reply.body().getString("status").equals("o
k");
                    res.setResult(authed).setHandler(handler);
                }
            });
        }
    }

    private void setUsername(String username, WebSocket socket) {
        SocketInfo socketInfo = sockInfos.get(socket);
        if (socketInfo == null)

```

```

        socketInfo.username = username;
    }

private void write(JsonObject message, NetSocket socket) {
    if (message == null) return;

    byte[] data;
    try {
        data = message.toString().getBytes("UTF-8");
        Buffer buffer = new Buffer(4+data.length);
        buffer.setInt(0, data.length);
        buffer.setBytes(4, data);
        socket.write(buffer);
    } catch (UnsupportedEncodingException e) {
    }
}

private void writeAll(JsonObject message, NetSocket exclude) {
    if (message == null) return;

    byte[] data;
    try {
        data = message.toString().getBytes("UTF-8");
        Buffer buffer = new Buffer(4+data.length);
        buffer.setInt(0, data.length);
        buffer.setBytes(4, data);
        for (String s : sockets) {
            if (exclude == null || !exclude.writeHandlerID().equals(s)) {
                eb.send(s, buffer);
            }
        }
    } catch (UnsupportedEncodingException e) {
    }
}

final Handler<NetSocket> connectHandler = new Handler<NetSocket>() {
    @Override

```

```

public void handle(final WebSocket socket) {
    SocketInfo socketInfo = new SocketInfo();
    socketInfo.writeHandlerID = socket.writeHandlerID();
    sockInfos.put(socket, socketInfo);

    //-- tcp stream framing
    final RecordParser framer = RecordParser.newFixed(4, null);
    framer.setOutput(new Handler<Buffer>() {
        int payloadLength = -1;
        @Override
        public void handle(Buffer buffer) {
            if (payloadLength == -1) {
                payloadLength = buffer.getInt(0);
                framer.fixedSizeMode(payloadLength);
            }
            else {
                final JsonObject message = new JsonObject(buffer.toString());
                final String sessionID = message.getString("sessionID");
                //-- session id check
                if (sessionID != null) {
                    authorise(message, sessionID, new
AsyncResultHandler<Boolean>() {
                        @Override
                        public void handle(AsyncResult<Boolean> result) {
                            if (result.succeeded()) {
                                if (result.result()) {
                                    cacheAuthorisation(sessionID, socket);
                                    writeAll(message.getObject("body"), socket);

                                    //-- send to sockjs server
                                    eb.send(INTERNAL_SOCKJS_SERV_ADDR,
message.getObject("body"));
                                }
                            }
                            else {
                                logger.warn("message rejected because
sessionID is not authorised");
                            }
                        }
                    });
                }
            }
        }
    });
}

```

```

        }
        else {
            logger.error("error in performing
authorisation", result.cause());
        }
    }
    //-- session id null
    else {
        eb.send(AUTH_MGR_ADDRESS_PREFIX+".login", message, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(Message<JsonObject> reply) {
        boolean authed = reply.body().getString("status"
).equals("ok");
        if (authed) {
            sockets.add(socket.writeHandlerID());
            setUsername(message.getString("username"),
socket);
        }
        write(reply.body(), socket);
    }
});
}
framer.fixedSizeMode(4);
payloadLength = -1;
}
}
});
//-- this handler will be called every time data is received on the socket
socket.dataHandler(framer);
//-- socket closed
socket.closeHandler(new VoidHandler() {
    @Override
    protected void handle() {

```

```

        sockets.remove(socket.writeHandlerID());
        //-- close any cached authorisations for this connection
        SocketInfo socketInfo = sockInfos.remove(socket);
        if (socketInfo != null) {
            Set<String> sses = socketInfo.sessions;
            if (sses != null) {
                for (String sessionID: sses) {
                    Auth auth = authCache.remove(sessionID);
                    if (auth != null)
                        auth.cancel();
                }
            }
        }
    });
    //-- something went wrong
    socket.exceptionHandler(new Handler<Throwable>() {
        @Override
        public void handle(Throwable throwable) {
            logger.error("unexpected exception: ", throwable);
        }
    });
}
};

@Override
public void start() {
    JsonObject appConfig = container.config();

    logger = container.logger();
    eb = vertx.eventBus();
    server = vertx.createNetServer();
    sockets = vertx.sharedData().getSet("sockets");

    authCacheTimeout = appConfig.getInteger("authCacheTimeout",
        LOCAL_AUTH_CACHE_TIMEOUT);
}

```

```
server.connectHandler(connectHandler);
server.listen(8090, "localhost", new AsyncResultHandler<NetServer>() {
    @Override
    public void handle(AsyncResult<NetServer> asyncResult) {
        logger.info("bind result: " + asyncResult.succeeded());
    }
});

//-- event bus subscribe
eb.registerHandler(INTERNAL_TCP_SERV_ADDR, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(final Message<JsonObject> message) {
        writeAll(message.body(), null);
    }
});
}
```

이번에는 3.4.2 사용자 인증의 SockJSEbbChatWithLoginServerVerticle을 수정해 새로운 IntegratedSockJSEbbChatWithLoginServerVerticle을 만들어 보자. (이는 [코드 4-2] IntegratedSockJSEbbChatServerVerticle에 사용자 인증을 요구하는 설정을 추가한 것과 유사하다)

먼저 TCP 채팅 서버가 이벤트 버스를 통해 전달하는 메시지를 수신하고, 해당 메시지를 SockJS EBB 채팅 클라이언트를 통해 활성화된 모든 이벤트 버스 주소에 브로드캐스팅하는 이벤트 핸들러를 추가한다.

```
eb.registerHandler(INTERNAL_SOCKJS_SERV_ADDR, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(final Message<JsonObject> message) {
        Iterator<String> keys = addresses.keySet().iterator();
```

```
        while (keys.hasNext()) {
            eb.publish(keys.next(), message.body());
        }
    });
}
```

그리고 mod-auth-mgr에서 사용되는 이벤트 버스 주소 외의 모든 주소에서 처리되는 메시지를 TCP 채팅 서버로 전달하도록 handleSendOrPub() 메서드를 작성한다.

```
@Override
public boolean handleSendOrPub(SockJSocket sock, boolean send, JSONObject
msg, String address) {
    logger.info("handleSendOrPub, sock = " + sock + ", send = " + send +",
address = " + address);
    //-- send to tcp server
    if (!address.startsWith(AUTH_MGR_ADDRESS_PREFIX))
        eb.send(INTERNAL_TCP_SERV_ADDR, msg.getObject("body"));
    return true; //-- true To allow the send/publish to occur, false otherwise
}
```

마지막으로 SockJS EBB 채팅 클라이언트를 통해 활성화되는 이벤트 버스 주소를 파악하기 위해 handlePostRegister() 메서드와 handleUnregister() 메서드를 작성한다.

```
@Override
public void handlePostRegister(SockJSocket sock, String address) {
    logger.info("handlePostRegister, sock = " + sock + ", address = " + address);
    Integer counter = addresses.get(address);
    if (counter == null) {
        counter = 1;
```

```

        }
    else {
        ++counter;
    }
    logger.info("address: "+address+", counter: "+counter);
    addresses.put(address, counter);
}

@Override
public boolean handleUnregister(SockJSocket sock, String address) {
    logger.info("handleUnregister, sock = " + sock + ", address = " +
address);
    Integer counter = addresses.get(address);
    if (counter != null) {
        if (--counter == 0) {
            addresses.remove(address);
        }
        else {
            addresses.put(address, counter);
        }
    }
    logger.info("address: "+address+", counter: "+counter);
    return true;
}

```

로그인 기능을 추가한 통합 TCP/SockJS EBB 채팅 서비스가 제대로 동작하는지 확인하기 위해 IntegratedChatWithLoginServerStarterVerticle을 작성하고 실행해 보자.

[코드 6-2] IntegratedChatWithLoginServerStarterVerticle

```

public class IntegratedChatWithLoginServerStarterVerticle extends Verticle {
    @Override
    public void start() {

```

```

JsonObject authMgrConfig = new JsonObject();
authMgrConfig.putString("address", "vertx.basicauthmanager");
authMgrConfig.putString("user_collection", "users");
authMgrConfig.putString("persistor_address", "vertx.mongodb");
authMgrConfig.putNumber("session_timeout", 1800000); // --30min
container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final", authMgrConfig);

JsonObject mongoPersistorConfig = new JsonObject();
mongoPersistorConfig.putString("address", "vertx.mongodb");
mongoPersistorConfig.putString("host", "localhost");
mongoPersistorConfig.putNumber("port", 27017);
mongoPersistorConfig.putString("db_name", "my-chat");
mongoPersistorConfig.putNumber("pool_size", 10);
container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",
mongoPersistorConfig);
container.deployVerticle("com.devop.vertx.ch5.IntegratedSockJSEbbChat
WithLoginServerVerticle");
container.deployVerticle("com.devop.vertx.ch5.IntegratedTCPChatWithLo
ginServerVerticle");
}
}

```

TCP 채팅 클라이언트와 SockJS EBB 채팅 클라이언트 모두 정상 로그인한 후 채팅 메시지와 함께 채팅 메시지를 발신한 사용자 아이디를 수신하는 것을 확인할 수 있다.

6.2 채팅 메시지 구조화

여기까지의 TCP/SockJS EBB 통합 채팅 서비스는 단순히 로그인과 로그인 성공 이후 채팅 메시지를 처리하는 수준에 머물러 있다. 채팅 대화방 구분, 귓속말 처리 등 좀 더 다양한 기능의 채팅 서비스를 구현하려면 채팅 서버와 클라이언트 사이에 오가는 메시지를 엄격하게 구조화할 필요가 있다. 메시지 구조화가 끝나면

일반 채팅 메시지와 대화방 입장/퇴장 같은 제어 메시지를 구분하고 상황에 맞는 동작을 좀 더 편리하게 구현할 수 있다.

6.2.1 일반 채팅 메시지와 제어 메시지

여기에서 일반 채팅 메시지는 오직 사용자의 입력에 의해 발생하며 채팅 서비스 내에서 특별한 의미를 지니지 않고 그대로 다른 사용자들에게 전달되는 문자열 데이터라 정의하자. 제어 메시지는 시스템에서 필요에 의해 자동으로 발생하거나 사용자의 입력으로 발생하지만 일반 채팅 메시지가 아닌 대화방 입장/퇴장, 귓속말 같은 특별한 의미를 지니는 문자열 데이터로 정의하자.

[표 6-1] 일반 채팅 메시지와 제어 메시지의 종류

분류	코드	값	의미
일반 채팅 메시지	CHAT	0	일반 채팅 메시지
제어 메시지	ENTER	1	대화방 입장
	LEAVE	2	대화방 퇴장
	NOTICE	3	시스템 공지 메시지
	WHISPER	4	귓속말 메시지

사용자 입력 데이터를 [표 6-1]에 대응되도록 해석하고, [코드 6-3]과 같이 처리하는 데 사용할 `MessageParser` 클래스를 선언한다. 제어 메시지는 이후 필요에 따라 언제든 추가할 수 있다. 단, 제어 메시지 정의가 추가되면 `MessageParser.MessageType`도 함께 수정하는 것을 잊지 말아야 한다.

[코드 6-3] `MessageParser.java`

```
public class MessageParser {  
    public enum MessageType {  
        CHAT, ENTER, LEAVE, NOTICE, WHISPER;  
    }  
    public static MessageType getMessageType(int code) {  
        if (code == 0) return CHAT;
```

```

        if (code == 1) return ENTER;
        if (code == 2) return LEAVE;
        if (code == 3) return NOTICE;
        if (code == 4) return WHISPER;
        return null;
    }
}

private JsonObject origin;
private MessageType messageType;
private String status;
private String sessionID;
private String rmID;
private JsonObject body;
private String username;
private String bodymsg;

public MessageParser(Buffer buffer) {
    this(buffer.toString());
}

public MessageParser(String jsonString) {
    this(new JsonObject(jsonString));
}

public MessageParser(JsonObject message) {
    this.origin = message;
    messageType = MessageType.getMessageType(message.getInteger("messageType", -1));
    status = message.getString("status");
    sessionID = message.getString("sessionID");
    rmID = message.getString("rmID");
    body = message.getObject("body");
    if (body != null) {
        username = body.getString("username");
        bodymsg = body.getString("message");
    }
}

```

```

    }

}

public JsonObject getInternalChatBroadcastData(WebSocket writer) {
    JsonObject internalBroadcastData = new JsonObject();
    internalBroadcastData.putNumber("messageType",
        messageType.ordinal());
    internalBroadcastData.putObject("body", body);
    internalBroadcastData.putString("rmID", rmID);
    if (writer != null)
        internalBroadcastData.putString("writerHandlerID",
            writer.writeHandlerID());
    return internalBroadcastData;
}

public boolean statusOK() {
    return (status!=null && "ok".equalsIgnoreCase(status));
}

//getter/setter 생략
}

```

[표 6-2] MessageParser 속성

속성	설명
origin	채팅 클라이언트에서 전송한 원본 JSON 객체다.
messageType	일반 채팅 메시지와 제어 메시지의 구분 코드다. 채팅 서버와 클라이언트는 이 구분 코드에 따라 적절한 처리를 수행해야 한다.
status	채팅 클라이언트의 로그인, 대화방 입장/퇴장 같은 요청 처리 결과를 의미한다. 이 값이 'ok'면 정상 처리된 것을 의미한다.
sessionID	로그인된 사용자의 세션 아이디이다. 로그인 요청을 제외한 모든 메시지에는 세션 아이디가 포함되어 있어야 한다(반대로 메시지에 세션 아이디가 없으면 해당 메시지를 mod-auth-mgr의 로그인 기능을 호출하는데 사용한다).
rmID	사용자가 현재 입장해 있는 대화방의 아이디다.
body	사용자 아이디와 사용자 입력 문자열을 포함하는 JSON 객체다.
username	사용자 아이디로, body로부터 추출된다.
bodymsg	사용자 입력 문자열로, body로부터 추출된다.

6.2.2 주요 제어 메시지 처리 흐름

SockJS EBB 채팅 클라이언트는 자신이 직접 서버 애플리케이션 이벤트 버스에 메시지를 보내거나 받을 수 있다. 이러한 특징 때문에 SockJS EBB 채팅 클라이언트는 메시지 전송, 로그인, 대화방 입장/퇴장 등 대부분 기능을 쉽고 편리하게 구현할 수 있다. 그 예로 실제 로그인 처리를 위해 SockJS EBB 채팅 클라이언트의 index.html 파일에 구현된 login() 메서드를 보면 그 구현이 매우 간단하고 직관적인 것을 알 수 있다. 물론 SockJS EBB 채팅 클라이언트가 직접 서버 애플리케이션 이벤트 버스에 접근할 수 있는 만큼 더욱 보안에 신경 써야 한다(이 내용은 [3.4 SockJS EBB 보안 설정](#)에서 다루었다).

```
function login() {
    var id = $('#id').val()
    var passwd = $('#passwd').val();
    if (eb && id.length>0 && passwd.length>0) {
        eb.login(id, passwd, function(reply) {
            console.log(reply);
            if (reply.status == 'ok') {
                loginOk(id);
            }
            else {
                alert('로그인 실패!');
            }
        });
    }
}
```

하지만 TCP 채팅 클라이언트는 직접 서버 애플리케이션 이벤트 버스에 메시지를 보내거나 받을 수 없다. 대신 TCP 채팅 클라이언트는 모든 요청을 TCP 채팅 서버로 전송해야 하고, TCP 채팅 서버가 해당 요청을 처리한 다음 그 결과를 다시

TCP 채팅 클라이언트에 돌려주는 방식을 사용한다.

따라서 TCP 채팅 서버는 TCP 채팅 클라이언트가 보내오는 메시지를 올바르게 해석하고 처리하기 위해 해당 메시지가 일반 채팅 메시지인지 아니면 어떤 특정한 동작을 요구하는 제어 메시지인지를 구분할 필요가 있다.

MessageParser는 이러한 처리가 쉽도록 작성된 것이다. 그럼 TCP 채팅 서버, 클라이언트에서 MessageParser를 사용해 몇 가지 주요 제어 메시지와 일반 채팅 메시지를 어떻게 처리하는지 자세히 알아보자.

로그인 요청 흐름

TCP 채팅 클라이언트는 최초 구동 시 콘솔로 사용자 아이디와 비밀번호를 입력받고, 이 사용자 아이디, 비밀번호는 다음의 JSON 객체로 만들어져 TCP 서버로 전송된다.

```
{"username": <username>, "password": <password>}
```

TCP 채팅 서버는 메시지를 수신하면 먼저 메시지에 세션 아이디가 포함되어 있는지 확인한다. 로그인 요청 메시지는 세션 아이디가 포함되어 있지 않으므로 TCP 채팅 서버는 해당 메시지를 그대로 mod-auth-mgr의 로그인 기능을 호출하는 데 사용한다. 그리고 로그인 결과는 그대로 TCP 채팅 클라이언트에 다시 전달한다. 이때 TCP 채팅 서버는 일종의 프록시Proxy 역할을 하게 된다.

```
Login 성공 : {"status": "ok", "sessionID": <session_id>}
```

```
Login 실패 : {"status": "denied"}
```

로그인 요청 결과를 수신한 TCP 채팅 클라이언트는 로그인 처리 결과를 확인하고,

성공이면 응답 메시지에 포함된 세션 아이디를 저장한다. 이후 TCP 채팅 클라이언트가 TCP 채팅 서버로 전송하는 모든 메시지에는 이 세션 아이디가 포함된다.

대화방 입장/퇴장 처리 흐름

로그인 처리 후 TCP 채팅 클라이언트는 대화방을 선택해야 한다. 대화방은 콘솔에 '/enter 대화방아이디' 명령을 입력하면 선택할 수 있다. 대화방 입장 명령은 다음의 JSON 객체로 만들어져 TCP 채팅 서버로 전송된다.

```
{"messageType":1,"rmID": <rmID>,"sessionID": <sessionID>,
"body": {"username": <username>,"message": "/enter <rmID>"}}
```

여기서 messageType 1(ENTER)은 대화방 입장 제어 메시지를 의미하며, rmID는 사용자가 콘솔로 입력한 대화방 아이디다.

TCP 채팅 서버는 대화방 입장 제어 메시지를 수신하고, rmID에 해당하는 대화방 사용자 목록에 TCP 채팅 클라이언트를 추가한다. 정상적으로 처리되면 TCP 채팅 서버는 대화방 입장 제어 메시지 원본에 "status": "ok" 값을 추가해 TCP 채팅 클라이언트에 응답한다.

TCP 채팅 클라이언트는 대화방 입장 처리 결과를 확인하고, 성공이면 응답 메시지에 포함된 대화방 아이디를 저장한다. 이후 TCP 채팅 클라이언트가 TCP 채팅 서버로 전송하는 모든 메시지에는 이 대화방 아이디가 포함된다. 대화방 아이디가 없는 상태에서는 일반 채팅 메시지를 전송할 수 없다.

TCP 채팅 클라이언트는 대화방 입장 후 '/leave 현재대화방아이디' 명령을 입력해 대화방에서 퇴장할 수 있다. 대화방 퇴장 명령은 다음의 같은 JSON 객체로 만들어져 서버로 전송된다.

```
{"messageType":2,"rmID": <rmID>,"sessionID": <sessionID>,
"body": {"username": <username>,"message": "/leave <rmID>"}}
```

여기서 messageType 2(LEAVE)는 대화방 퇴장 제어 메시지를 의미하며, rmID는 사용자가 콘솔로 입력한 현재 대화방 아이디다.

TCP 채팅 서버는 대화방 퇴장 제어 메시지를 수신하고, rmID에 해당하는 대화방 사용자 목록에서 TCP 채팅 클라이언트를 제거한다. 정상적으로 처리되면 TCP 채팅 서버는 대화방 퇴장 제어 메시지 원본에 “status”: “ok” 값을 추가해 TCP 채팅 클라이언트에 응답한다.

TCP 채팅 클라이언트는 대화방 퇴장 처리 결과를 확인하고, 성공이면 현재 저장되어 있는 대화방 아이디를 null로 설정한다. TCP 채팅 클라이언트는 다시 '/enter 대화방아이디' 명령을 입력하여 대화방에 입장한 다음 채팅 메시지를 전송할 수 있다.

일반 채팅 메시지 처리 흐름

대화방 입장 후 사용자는 같은 대화방에 참여한 다른 사용자에게 일반 채팅 메시지를 전송하거나 수신할 수 있다. 물론 서로 다른 대화방에 참여한 사용자끼리는 일반 채팅 메시지를 전송하거나 수신할 수 없다.

```
{"messageType":0,"rmID": <rmID>,"sessionID": <sessionID>,
"body": {"username": <username>,"message": "채팅메시지"}}
```

여기서 messageType 0(CHAT)은 일반 채팅 메시지를 의미한다.

6.3 대화방 분리

일반적인 채팅 서비스에서 사용자는 서로 독립적인 대화방에 자유롭게 입장하거나 퇴장하며 채팅 서비스를 이용한다. 그러나 이제까지 작성한 채팅 서비스는 대화방에 대한 명확한 정의 없이 대부분의 채팅 클라이언트가 서로 메시지를 주고 받을 수 있다.

NOTE

SockJS EBB 채팅 클라이언트는 동일한 이벤트 버스 주소에 대해 Subscribe가 이루어진 사용자 사이에서만 채팅 메시지 공유가 가능하므로 ‘대부분의 채팅 클라이언트’라고 표현했다. 즉, SockJS EBB 채팅 서버는 서로 구분되는 이벤트 버스 주소가 의도치 않게 각각의 대화방을 구성하는 형태가 된다.

이번 장에서는 [6.2.2 주요 제어 메시지 처리 흐름](#)에서 설명한 내용에 따라 채팅 대화방을 독립적으로 분리하고 동일한 대화방 내의 사용자 사이에서만 채팅 메시지가 공유되도록 TCP/SockJS EBB 통합 채팅 서비스에 기능을 추가해 보자.

6.3.1 TCP 채팅 서버 대화방 분리

[[코드 6-1](#)] [IntegratedTCPChatWithLoginServerVerticle](#)을 기본으로 일부 기능을 추가해 대화방을 분리해 보자(대화방 분리 기능이 추가된 TCP 채팅 서버는 [IntegratedTCPChatWithRoomServerVerticle](#)이다).

먼저 대화방 아이디별로 대화방에 참여 중인 사용자 목록을 저장할 자료구조를 선언한다.

```
private final Map<String, Set<NetSocket>> rooms = new HashMap<>();
```

클라이언트가 연결된 `NetSocket`의 부가정보를 저장하기 위한 용도로 사용되는 `SocketInfo` 클래스에 클라이언트가 참여 중인 대화방 아이디를 의미하는 `rmID`를 추가한다. `rmID` 값이 `null`이면 해당 클라이언트가 참여 중인 대화방이 없다는 것을 의미한다.

```
private class SocketInfo {  
    private Set<String> sessions;  
    private String writeHandlerID;  
    private String username;  
    private String rmID;  
}
```

그리고 `NetSocket` 별로 `SocketInfo`를 저장할 자료구조를 선언한다.

```
private final Map<NetSocket, SocketInfo> sockInfos = new HashMap<>();
```

앞서 선언한 2개의 자료구조 `rooms`와 `sockInfos`는 각 `Verticle`이 독립적으로 관리하는 고유의 데이터이며, `Verticle` 사이에서 공유되지 않는다는 것을 염두에 두자.

TCP 채팅 클라이언트의 대화방 입장/퇴장 제어 메시지를 처리하는 `enter()`, `leave()` 메서드를 추가한다. 이 두 메서드는 `rmID`에 해당하는 대화방 사용자 목록에 TCP 채팅 클라이언트를 추가하거나 제거하고, `SocketInfo`의 `rmID` 값에 참여 중인 대화방 아이디를 설정하거나 삭제하며 모든 처리가 정상적으로 끝나면 `true`를 반환한다.

```
private boolean enter(String rmID, NetSocket socket) {  
    SocketInfo socketInfo = sockInfos.get(socket);
```

```

        if (socketInfo != null && socketInfo.rmID == null) {
            Set<NetSocket> users = rooms.get(rmID);
            if( users == null ) {
                users = new HashSet<>();
                rooms.put(rmID, users);
            }
            users.add(socket);
            socketInfo.rmID = rmID;
            return true;
        }
        return false;
    }

private boolean leave(String rmID, NetSocket socket) {
    SocketInfo socketInfo = sockInfos.get(socket);
    if (socketInfo != null && socketInfo.rmID != null &&
    socketInfo.rmID.equals(rmID)) {
        Set<NetSocket> users = rooms.get(rmID);
        if (users != null) {
            users.remove(socket);
        }
        socketInfo.rmID = null;
        return true;
    }
    return false;
}

```

rmID에 해당하는 대화방에 참여 중인 모든 사용자에게 메시지를 브로드캐스팅 하는 `writeAll()` 메서드도 추가한다.

```

private void writeAll(JSONObject message, String rmID, String exclude) {
    if (message == null || rmID == null) return;

    Set<NetSocket> users = rooms.get(rmID);

```

```
if (users != null) {
    byte[] data;
    try {
        data = message.toString().getBytes("UTF-8");
        Buffer buffer = new Buffer(4+data.length);
        buffer.setInt(0, data.length);
        buffer.setBytes(4, data);
        for (NetSocket socket : users) {
            if (exclude == null || !exclude.equals(socket.writeHandlerID())) {
                socket.write(buffer);
            }
        }
    } catch (UnsupportedEncodingException e) {
    }
}
```

TCP 채팅 클라이언트가 대화방에 참여 중인 상태에서 TCP 채팅 서버와 연결이 종료되었을 때 대화방에서 해당 TCP 채팅 클라이언트를 제거할 수 있도록 `closeHandler`를 추가한다.

```
socket.closeHandler(new VoidHandler() {
    @Override
    protected void handle() {
        sockets.remove(socket.writeHandlerID());
        SocketInfo socketInfo = sockInfos.remove(socket);
        if (socketInfo != null) {
            //-- leave a room
            if (socketInfo.rmID != null) {
                Set<NetSocket> users = rooms.get(socketInfo.rmID);
                if (users != null ) {
                    users.remove(socket);
                }
            }
        }
    }
});
```

```
    }
    //-- close any cached authorisations for this connection
    Set<String> sses = socketInfo.sessions;
    if (sses != null) {
        for (String sessionID: sses) {
            Auth auth = authCache.remove(sessionID);
            if (auth != null)
                auth.cancel();
        }
    }
}
});
```

그리고 TCP 채팅 클라이언트가 보내오는 메시지를 MessageParser를 활용해 구분하고 적절히 처리할 수 있도록 RecordParser 객체를 다음과 같이 작성한다. 특히 세션 아이디 확인을 위한 authorize() 메서드 호출 이후 처리 흐름을 주의 깊게 살펴보자.

```
final RecordParser framer = RecordParser.newFixed(4, null);
framer.setOutput(new Handler<Buffer>() {
    int payloadLength = -1;
    @Override
    public void handle(Buffer buffer) {
        if (payloadLength == -1) {
            payloadLength = buffer.getInt(0);
            framer.fixedSizeMode(payloadLength);
        }
        else {
            final MessageParser message = new MessageParser(buffer);
            //-- session id check
            if (message.getSessionID() != null) {
                authorise(message.origin(), message.getSessionID(), new
AsyncResultHandler<Boolean>() {
```

```

@Override
public void handle(AsyncResult<Boolean> result) {
    if (result.succeeded()) {
        if (result.result()) {
            cacheAuthorisation(message.getSessionID(), socket);
            if (message.getMessageType() == MessageType.CHAT) {
                if (message.getBody() != null &&
message.getRmID() != null ) {
                    SocketInfo socketInfo = sockInfos.get(socket);
                    if (socketInfo.rmID != null && socketInfo.
rmID.equals(message.getRmID())) {
                        eb.publish(INTERNAL_TCP_SERV_ADDR, mes
sage.getInteralChatBroadcastData(socket));
                        //-- send to sockjs server
                        eb.send(INTERNAL SOCKJS SERV ADDR, mes
sage.getInteralChatBroadcastData(null));
                    }
                } else {
                    logger.warn("message rejected because
invalid rmID");
                }
            } else {
                logger.warn("message rejected because body
or rmID is missing");
            }
        }
    } else if (message.getMessageType() ==
MessageType.ENTER) {
        if (message.getBody() != null &&
message.getRmID() != null ) {
            boolean isOK = enter(message.getRmID(),
socket);
            JSONObject response = message.origin();
            response.putString("status",
(isOK)?"ok":"denied");
            write(response, socket);
        }
    }
}

```

```
        else {
            logger.warn("message rejected because body
or rmID is missing");
        }
    }
    else if (message.getMessageType() ==
MessageType.LEAVE) {
        if (message.getBody() != null &&
message.getRmID() != null ) {
            boolean isOK = leave(message.getRmID(),
socket);
            JSONObject response = message.origin();
            response.putString("status",
(isOK)?"ok":"denied");
            write(response, socket);
        }
        else {
            logger.warn("message rejected because body
or rmID is missing");
        }
    }
    else {
        logger.warn("message rejected because not
supported messageType");
    }
}
else {
    logger.warn("message rejected because sessionID
is not authorised");
}
}
else {
    logger.error("error in performing authorisation",
result.cause());
}
}
});
```

```

    //-- session id null
    else {
        eb.send(AUTH_MGR_ADDRESS_PREFIX+".login", message.origin(),
new Handler<Message<JsonObject>>() {
    @Override
    public void handle(Message<JsonObject> reply) {
        boolean authed = reply.body().getString("status").equal
s("ok");
        if (authed) {
            sockets.add(socket.writeHandlerID());
            setUsername(message.origin().getString("username"),
socket);
        }
        write(reply.body(), socket);
    }
});
    }
    framer.fixedSizeMode(4);
    payloadLength = -1;
}
}
});

```

메시지에 포함된 messageType 구분 코드에 따라 해당 메시지가 일반 채팅 메시지인지 대화방 입장/퇴장을 위한 제어 메시지인지 구분하는데, 대화방 입장이면 enter() 메서드를, 대화방 퇴장이면 leave() 메서드를 호출한다. 일반 채팅 메시지면 메시지를 보내온 TCP 채팅 클라이언트가 참여한 대화방 아이디를 확인하고, 해당 대화방에 참여 중인 모든 사용자에게 일반 채팅 메시지를 전달하게 작성된 이벤트 핸들러(상수 INTERNAL_TCP_SERV_ADDR로 정의되어 있다)에 publish를 수행한다.

NOTE

바로 `writeAll()` 메서드를 호출해 대화방에 입장한 모든 사용자에게 일반 채팅 메시지를 전달하지 않는 이유를 생각해 보자. `writeAll()` 메서드를 살펴보면 대화방 아이디별로 참여 중인 사용자 목록을 저장하는 자료구조 `rooms`를 이용하는 것을 확인할 수 있다. 여기서 `rooms`의 선언 타입은 `Map<String, Set<NetSocket>>`이며, 이는 `sharedData`를 통해 다른 Verticle과 공유되지 않는다. 즉, `rooms`는 서로 독립적인 Verticle의 고유 데이터라는 뜻이다.

따라서 2개 이상의 `IntegratedTCPChatWithRoomServerVerticle`이 배포되었을 때 `rooms`는 서로 독립적이며 특정 `IntegratedTCPChatWithRoomServerVerticle`에서 호출되는 `writeAll()` 메서드는 해당 `IntegratedTCPChatWithRoomServerVerticle`의 `rooms`에 포함된 TCP 채팅 클라이언트에게만 일반 채팅 메시지를 전달한다.

이와 같은 문제를 해결하기 위해 모든 `IntegratedTCPChatWithRoomServerVerticle`은 상수 `INTERNAL_TCP_SERV_ADDR`로 정의된 주소를 처리하는 이벤트 핸들러가 필요하다. 이 이벤트 핸들러를 통해 배포된 모든 `IntegratedTCPChatWithRoomServerVerticle`에 일반 채팅 메시지를 전달하고, 그 이후 각각의 이벤트 핸들러는 `writeAll()` 메서드를 호출해 최종으로 일반 채팅 메시지를 사용자에게 전달한다.

SockJS EBB 채팅 클라이언트도 메시지를 받을 수 있게 SockJS EBB 채팅 서버에도 메시지를 전달하는 것을 잊지 말자.

마지막으로 특정 대화방에 참여 중인 모든 사용자에게 일반 채팅 메시지를 전달하기 위한 이벤트 핸들러를 작성한다.

```
eb.registerHandler(INTERNAL_TCP_SERV_ADDR, new Handler<Message<JsonObject>>() {
{
    @Override
    public void handle(final Message<JsonObject> message) {
        JsonObject internalBroadcastData = message.body();
        String rmID = (String) internalBroadcastData.removeField("rmID");
    }
}
```

```
        String writerHandlerID = (String) internalBroadcastData.removeField("writerHandlerID");
        writeAll(internalBroadcastData, rmID, writerHandlerID);
    }
});
```

일반 채팅 메시지가 최종으로 사용자에게 전달될 때는 rmID, writeHandlerID처럼 불필요한 값은 제거된다.

6.3.2 TCP 채팅 클라이언트 대화방 분리

6.1.2 TCP 채팅 클라이언트 사용자 인증의 TCPChatWithLoginClient Verticle을 기본으로 일부 기능을 추가해 대화방 분리를 구현해 보자(대화방 분리 기능이 추가된 TCP 채팅 클라이언트는 TCPChatWithRoomClientVerticle이다).

먼저 사용자 입력 메시지를 이벤트 버스로 받아 처리하는 부분을 수정한다. 대화방 입장/퇴장 제어 메시지는 각각 '/enter'와 '/leaver'로 시작되며, 모든 제어 메시지는 콘솔로 입력받은 대화방명을 rmID라는 파라미터명으로 TCP 채팅 서버로 전송해야 한다.

```
/enter 대화방명  
/leaver 대화방명
```

특히 대화방에 입장하지 않은 상태에서는 채팅 메시지를 전송할 수 없게 한다.

```
eb.registerHandler("com.devop.vertx.chat", new Handler<Message<JsonObject>>()
{
    @Override
    public void handle(Message<JsonObject> input) {
```

```

if (writeHandlerID != null) {
    JSONObject message = input.body();
    JSONObject body = message.getJSONObject("body");
    if (message.getBoolean("exit", false)) {
        container.exit();
    }
    else {
        if (sessionId != null) {
            message.putString("sessionId", sessionId);
            body.putString("username", username);
            String line = body.getString("message");
            String[] tokens = line.split(" ");

            if ("/enter".startsWith(tokens[0]) && tokens.length == 2 &&
rmID == null) {
                message.putNumber("messageType",
MessageType.ENTER.ordinal());
                message.putString("rmID", tokens[1]);
            }
            else if ("/leave".startsWith(tokens[0]) && tokens.length ==
2 && rmID != null) {
                message.putNumber("messageType",
MessageType.LEAVE.ordinal());
                message.putString("rmID", tokens[1]);
            }
            else if (rmID != null) {
                message.putNumber("messageType",
MessageType.CHAT.ordinal());
                message.putString("rmID", rmID);
            }
            else {
                System.out.println("you must enter roomID");
            }
        }
    }
}

```

```

        return;
    }
}

else {
    if (username == null)
        username = message.getString("username");
}

byte[] data;
try {
    data = message.toString().getBytes("UTF-8");
    Buffer buffer = new Buffer(4+data.length);
    buffer.setInt(0, data.length);
    buffer.setBytes(4, data);
    eb.send(writeHandlerID, buffer);
} catch (UnsupportedEncodingException e) {
}
}

});


```

대화방 입장/퇴장 제어 메시지에 대한 TCP 채팅 서버의 응답 메시지 같은 대화방의 다른 사용자가 전달한 일반 채팅 메시지, 로그인 요청 결과를 처리하기 위해 RecordParser 객체를 다음과 같이 작성한다.

```

final RecordParser framer = RecordParser.newBuilder()
    .setOutput(new Handler<Buffer>() {
        int payloadLength = -1;
        @Override
        public void handle(Buffer buffer) {

```

```

if (payloadLength == -1) {
    payloadLength = buffer.getInt(0);
    framer.fixedSizeMode(payloadLength);
}
else {
    MessageParser message = new MessageParser(buffer);
    if (sessionId != null) {
        if (message.getMessageType() == MessageType.CHAT) {
            String from = message.getUsername();
            String msg = message.getBodymsg();
            System.out.println(from+": "+msg);
        }
        else if (message.getMessageType() == MessageType.ENTER) {
            if (message.statusOK()) {
                rmID = message.getRmID();
                System.out.println("enter the room '"+rmID+"'");
            }
        }
        else if (message.getMessageType() == MessageType.LEAVE) {
            if (message.statusOK()) {
                System.out.println("leave the room '"+rmID+"'");
                rmID = null;
            }
        }
    }
    else {
        boolean authed = message.statusOK();
        //-- login success
        if (authed) {
            sessionId = message.getSessionID();
            System.out.println("user '"+username+"' login success at "
"+new Date());
        }
        //-- login failure
    else {

```

```
        System.out.println("user '" +username+ "' login failure.  
please enter exit.");
    }
}

framer.fixedSizeMode(4);
payloadLength = -1;
}
}
});
```

대화방 입장 제어 메시지가 정상적으로 처리되면 대화방 입장 알림을 출력하고, 현재 입장해 있는 대화방을 의미하는 rmID에 대화방 아이디를 저장한다. 대화방 퇴장 제어 메시지가 정상적으로 처리되면 대화방 퇴장 알림을 출력하고, 현재 입장해 있는 대화방이 없다는 의미로 rmID를 null로 설정한다.

TCP 채팅 클라이언트에서 대화방을 분리하기 위해 추가해야 하는 기능은 이것이 전부다. TCPChatWithLoginClientWorkerVerticle은 따로 변경할 필요 없이 그대로 사용할 수 있다. 그러나 클래스 이름의 규칙성을 위해 대화방 분리 클라이언트 Worker Verticle은 TCPChatWithRoomClientWorkerVerticle이라고 정한다.

6.3.3 SockJS EBB 채팅 서버 대화방 분리

사실 SockJS EBB 채팅 서버에는 대화방을 분리하기 위한 별도의 기능을 추가할 필요가 없다. 단, 대화방 분리 기능이 추가된 TCP 채팅 서버와 통합하려면 일부 메서드는 수정되어야 한다. 이는 [6.1.3 사용자 인증 TCP/SockJS EBB 채팅 서비스 통합](#)의 IntegratedSockJSEbbChatWithLoginServerVerticle을 기본으로 일부 메서드를 수정한다(수정된 새로운 Verticle은 IntegratedSockJSEbbChatWithRoomServerVerticle이라 하자).

먼저 handleSendOrPub() 메서드를 다음과 같이 작성한다. 일반 채팅 메시지를 의미하는 messageType과 대화방 아이디를 의미하는 rmID가 추가되었다.

```
@Override
public boolean handleSendOrPub(SockJSocket sock, boolean send, JSONObject
msg, String address) {
    logger.info("handleSendOrPub, sock = " + sock + ", send = " + send + ", "
address = " + address);
    //-- send to tcp server
    if (!address.startsWith(AUTH_MGR_ADDRESS_PREFIX)) {
        JSONObject internalBroadcastData = new JSONObject();
        internalBroadcastData.putNumber("messageType",
MessageType.CHAT.ordinal());
        internalBroadcastData.putString("rmID", address);
        internalBroadcastData.putObject("body", msg.getObject("body"));
        eb.publish(INTERNAL_TCP_SERV_ADDR, internalBroadcastData);
    }
    return true; //-- true To allow the send/publish to occur, false otherwise
}
```

이 코드를 보면 rmID 값으로 address가 입력되는데, 그 이유를 생각해 보자.

SockJS EBB 채팅 클라이언트는 동일한 이벤트 버스 주소에 대한 Subscribe가 이루어진 사용자 사이에서만 채팅 메시지 공유가 가능한데, 이는 이벤트 버스 주소가 대화방 아이디처럼 사용되기 때문이다. 즉, SockJS EBB 채팅 서버에서 이벤트 버스 주소는 TCP 채팅 서버에서의 rmID와 동일하며 TCP 채팅 서버에서의 rmID는 SockJS EBB 채팅 서버의 이벤트 버스 주소가 된다는 뜻이다.

NOTE

4장 TCP/SockJS EBB 채팅 서비스 통합과 6.1 TCP 채팅 서버 로그인 기능에서 TCP/SockJS EBB 채팅 서비스 통합은 SockJS EBB 채팅 서버에서 TCP 채팅 서버로 이벤트 버스를 통해 채팅 메시지를 전달하기 위해 모두 `handleSendOrPub()` 메서드 내에서 `send()` 메서드를 사용했다.

그러나 앞의 코드에서는 이벤트 버스를 통해 TCP 채팅 서버로 채팅 메시지를 전달하기 위해 `send()` 메서드가 아닌 `publish()` 메서드가 사용되었다. 이것은 매우 중요한 의미가 있는데, 앞서 6.3.1의 NOTE에서 설명한 것처럼 대화방 분리 기능을 적용한 TCP 채팅 서버 Verticle은 서로 독립적인 자료구조 `rooms`가 있고, 이에 따라 `send()` 메서드를 통해 한 개의 TCP 채팅 서버 Verticle로만 채팅 메시지를 전달하면 동일한 대화방에 입장한 모든 TCP 채팅 클라이언트가 채팅 메시지를 받아볼 수 없을지도 모른다. 따라서 모든 TCP 채팅 서버 Verticle이 채팅 메시지를 전달받고 처리할 수 있도록 `send()` 메서드가 아닌 `publish()` 메서드를 사용해야 한다. 이처럼 Verticle 간의 메시지 전달을 위해 이벤트 버스를 사용할 때는 2개 이상의 Verticle이 실행될 수 있다는 가정하에 `send()` 메서드를 호출해야 할지 `publish()` 메서드를 호출해야 할지 신중히 생각하고 결정해야 한다. 그렇지 않으면 특정 클라이언트에만 메시지가 전달되지 않는 버그가 발생할 수 있다.

TCP 채팅 서버의 `rmID`와 SockJS EBB 서버의 `address`가 상호 교환할 수 있다는 특징은 TCP 채팅 서버에서 전달받은 메시지를 SockJS EBB 채팅 클라이언트로 전달하기 위해 작성한 이벤트 핸들러에서도 그대로 활용된다.

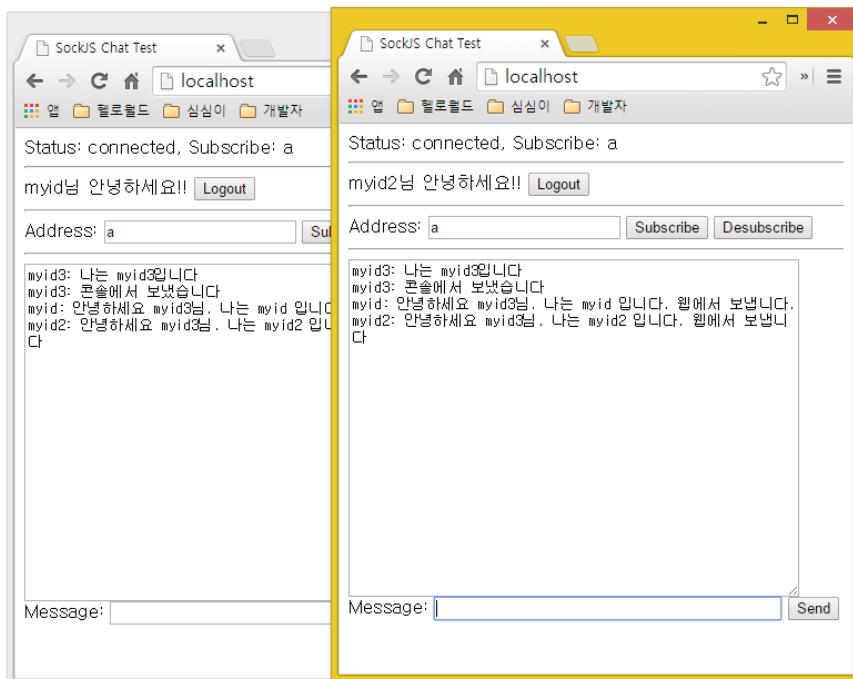
```
eb.registerHandler(INTERNAL_SOCKJS_SERV_ADDR, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(final Message<JsonObject> message) {
        JsonObject internalBroadcastData = message.body();
        String rmID = internalBroadcastData.getString("rmID");
        Integer count = addresses.get(rmID);
        if (count != null && count > 0 ) {
            eb.publish(rmID, internalBroadcastData.getObject("body"));
        }
    }
}
```

```
});
```

SockJS EBB 채팅 서버에서 대화방 분리를 위해 수정해야 하는 내용은 이것이 전부다. 심지어 SockJS EBB 채팅 클라이언트의 index.html은 수정할 필요도 없다.

간단한 테스트를 위해 TCP/SockJS 통합 채팅 서비스를 시작하고, 각 채팅 클라이언트를 실행해 보자. [그림 6-1]은 SockJS EBB 채팅 클라이언트에서 'a'라는 대화방의 사용자들이 채팅 메시지를 주고받는 모습이다.

[그림 6-1] TCP/SockJS 통합 채팅 서비스에서 대화방 분리 결과 화면



다음은 Windows 8의 PowerShell에서 TCPChatWithRoomClientVerticle을 실행하고 로그인해서 대화방에 입장한 후 채팅 메시지를 전송하거나 수신했을 때 출력된 결과다. [그림 6-1]과 마찬가지로 ‘a’라는 대화방에서 채팅 메시지를 주고받는 것을 확인할 수 있다.

```
PS C:\Users\yeon> vertx run TCPChatWithRoomClientVerticle.java
connect result: true
Succeeded in deploying verticle
input username: myid3
input password: 1234
user 'myid3' login success at Fri Nov 14 00:42:33 KST 2014
/enter a
enter the room 'a'
나는 myid3입니다
콘솔에서 보냈습니다
myid: 안녕하세요 myid3님. 나는 myid 입니다. 웹에서 보냅니다.
myid2: 안녕하세요 myid3님. 나는 myid2 입니다. 웹에서 보냅니다
```

6.4 귓속말 처리

이번에는 같은 대화방에 입장한 사용자끼리 귓속말을 주고받는 기능을 추가해 보자. 귓속말 기능 구현할 때 귓속말 메시지는 오직 수신자로 지정된 사용자와 귓속말 입력 사용자만 확인할 수 있으며, 그 외 사용자는 확인할 수 없어야 한다는 점을 유의해야 한다

6.4.1 whisperTarget 속성

TCP/SockJS EBB 채팅 서버에 귓속말 기능을 추가하기에 앞서 MessageParser에 귓속말 수신자를 의미하는 `whisperTarget`이라는 속성을 추가한다. 그리고 `getInternalChatBroadcastData()` 메서드를 다음과 같이 작성한다. 수정된

`getInternalChatBroadcastData()` 메서드가 반환하는 JSON 객체는 메시지가
귓속말인지 일반 채팅 메시지인지 구분하는 boolean 값 `whisper`와 메시지가
귓속말일 때 귓속말 수신자를 의미하는 `whisperTarget`을 포함한다.

```
private String whisperTarget;

public MessageParser(JSONObject message) {
    this.origin = message;
    messageType = MessageType.getMessageType(message.getInteger("messageType", -1));
    status = message.getString("status");
    sessionID = message.getString("sessionID");
    rmID = message.getString("rmID");
    whisperTarget = message.getString("whisperTarget");
    body = message.getJSONObject("body");
    if (body != null) {
        username = body.getString("username");
        bodymsg = body.getString("message");
    }
}

public JSONObject getInternalChatBroadcastData(WebSocket writer) {
    JSONObject internalBroadcastData = new JSONObject();
    internalBroadcastData.putNumber("messageType", messageType.ordinal());
    internalBroadcastData.putObject("body", body);
    internalBroadcastData.putString("rmID", rmID);
    body.putBoolean("whisper", (whisperTarget != null));
    if (whisperTarget != null)
        internalBroadcastData.putString("whisperTarget", whisperTarget);

    if (writer != null)
        internalBroadcastData.putString("writerHandlerID",
writer.writeHandlerID());
```

```
    return internalBroadcastData;
}
```

그리고 문자열 배열을 하나의 문자열로 연결해 반환하는 유틸리티 메서드를 추가한다. 이 메서드는 '/whisper 수신자아이디 귓속말메시지'와 같은 입력 값에서 귓속말 메시지만을 추출하기 위해 사용된다.

```
public static String arrayToString(String[] array, int offset, int length) {
    StringBuffer sb = new StringBuffer();
    for (int i=offset; i<length; i++) {
        sb.append(array[i]);
        if (i < length-1)
            sb.append(" ");
    }
    return sb.toString();
}
```

6.4.2 TCP 채팅 서버 귓속말

6.3.1 TCP 채팅 서버 대화방 분리의 IntegratedTCPChatWithRoomServerVerticle을 기본으로 일부 기능을 추가해 귓속말 기능을 구현해 보자(귓속말 기능이 추가된 TCP 채팅 서버는 IntegratedTCPChatWithWhisperServerVerticle이다).

TCP 채팅 서버의 특정 rmID에 해당하는 대화방 사용자 목록에서 귓속말 수신자를 찾는 메서드를 추가한다. 귓속말 수신자를 찾을 수 없으면 null을 반환한다.

```
private SocketInfo findByUsername(String username, String rmID) {
    Set<NetSocket> sockets = rooms.get(rmID);
    if (sockets != null) {
```

```
        for(NetSocket socket : sockets) {
            SocketInfo socketInfo = sockInfos.get(socket);
            if (username.equals(socketInfo.username) &&
                rmID.equals(socketInfo.rmID)) {
                return socketInfo;
            }
        }
        return null;
    }
}
```

writeAll() 메서드를 다음과 같이 작성한다. writeAll() 메서드는 메시지가
귓속말 메시지인지 일반 채팅 메시지인지 구분해 해당 메시지를 다른 사용자에게
전달한다. 귓속말이면 findByUsername() 메서드로 수신자를 찾을 수 있을 때
해당 수신자 한 명에게만 메시지가 전달된다.

```
private void writeAll(JSONObject message, String rmID, String whisperTarget,
String exclude) {
    if (message == null || rmID == null) return;

    Set<NetSocket> users = rooms.get(rmID);
    if (users != null) {
        byte[] data;
        try {
            boolean isWhisper = (whisperTarget != null);
            data = message.toString().getBytes("UTF-8");
            Buffer buffer = new Buffer(4+data.length);
            buffer.setInt(0, data.length);
            buffer.setBytes(4, data);
            if (!isWhisper) {
                for (NetSocket socket : users) {
```

```

        if (exclude == null || !exclude.equals(socket.writeHandlerID)) {
            socket.write(buffer);
        }
    }
}
else {
    SocketInfo socketInfo = findByUsername(whisperTarget, rmID);
    if (socketInfo != null) {
        eb.send(socketInfo.writeHandlerID, buffer);
    }
}
} catch (UnsupportedEncodingException e) {
}
}
}

```

세션 아이디 확인을 위한 authorize() 메서드를 호출한 다음 메시지가 일반 채팅 메시지일 때 실행되는 부분을 다음과 같이 수정한다. messageType 코드 구분에 궂속말인 경우를 처리하기 위한 부분이 추가되었다.

```

if (message.getMessageType() == MessageType.CHAT || message.getMessageType()
== MessageType.WHISPER) {
    if (message.getBody() != null && message.getRmID() != null ) {
        SocketInfo socketInfo = sockInfos.get(socket);
        if (socketInfo.rmID != null && socketInfo.rmID.equals(message.getRmID())) {
            if (message.getMessageType() == MessageType.CHAT) {
                eb.publish(INTERNAL_TCP_SERV_ADDR, message.getInteralChatBroadcastData(socket));
                //-- send to sockjs server
            }
        }
    }
}

```

```

        eb.send(INTERNAL_SOCKJS_SERV_ADDR, message.getInteralChatBroad
castData(null));
    }
    else if (message.getWhisperTarget() != null) {
        eb.publish(INTERNAL_TCP_SERV_ADDR, message.getInteralChatBroad
castData(socket));
        //-- publish to sockjs server
        eb.publish(INTERNAL_SOCKJS_SERV_ADDR, message.getInteralChatBr
oadcastData(null));
    }
    else {
        logger.warn("message rejected because whisperTarget is
missing");
    }
}
else {
    logger.warn("message rejected because invalid rmID");
}
}
else {
    logger.warn("message rejected because body or rmID is missing");
}
}

```

여기서 궁속말 메시지를 상수 INTERNAL_TCP_SERV_ADDR와 INTERNAL_SOCKJS _SERV_ADDR로 정의된 2개의 이벤트 버스 주소로 publish하는 부분을 주의 깊게 살펴봐야 한다.

앞서 일반 채팅 메시지를 대화방에 입장해 있는 모든 사용자에게 전달하기 위해 writeAll() 메서드를 바로 호출하는 대신 상수 INTERNAL_TCP_SERV_ADDR로 정의된 이벤트 버스에 publish하는 것을 확인했다. 이것은 2개 이상의 TCP 채팅

서버 Verticle이 배포되었을 때 각 서버 Verticle이 대화방 아이디별로 참여 중인 사용자 목록을 저장하는 자료구조 rooms를 독립적으로 관리하기 때문이었다.

귓속말 메시지 처리도 이와 동일한 이유로 상수 INTERNAL_TCP_SERV_ADDR로 정의된 이벤트 버스에 publish해야 한다. 상수 INTERNAL_TCP_SERV_ADDR에 의해 실행되는 이벤트 핸들러는 각 TCP 채팅 서버 Verticle이 관리하는 자료구조 rooms와 sockInfos로 귓속말 메시지 수신자를 찾고 수신자를 찾으면 귓속말 메시지를 전달한다.

그렇다면 상수 INTERNAL_SOCKJS_SERV_ADDR로 정의된 이벤트 버스에서 일반 채팅 메시지와 귓속말 메시지를 처리하는 방법이 다른 이유는 무엇일까?(일반 채팅 메시지는 send, 귓속말 메시지는 publish로 처리된다)

일반 채팅 메시지는 2개 이상의 SockJS EBB 채팅 서버 Verticle이 배포되었을 때 그중 1개의 Verticle에서만 일반 채팅 메시지를 수신해도 대화방에 입장한 모든 사용자에게 메시지를 전달할 수 있다. 이는 SockJS EBB의 특징 때문인데, SockJS EBB 채팅 클라이언트는 대화방 아이디에 해당하는 이벤트 버스 주소를 subscribe하고 있고, 특정 SockJS EBB 채팅 서버 Verticle 한 개에서만 해당 이벤트 버스 주소로 메시지를 publish하면 된다.

그러나 SockJS EBB 채팅 서버에서 귓속말 메시지는 TCP 채팅 서버와 마찬가지로 독립적으로 관리하는 자료구조 rooms가 있어서(SockJS EBB 채팅 서버에서 귓속말 구현은 [6.4.4 SockJS EBB 채팅 서버 귓속말 참조](#)) 한 개의 SockJS EBB 채팅 서버 Verticle이 아닌 모든 SockJS EBB 채팅 서버 Verticle이 메시지를 수신하고 처리할 수 있도록 publish해야 한다.

마지막으로 귓속말 수신자를 찾아 귓속말 메시지를 전달하도록 이벤트 핸들러를 다음과 같이 작성한다. 해당 메시지가 귓속말 메시지가 아니라면 whisperTarget

값은 null이 되고, 이 메시지는 일반 채팅 메시지이므로 대화방에 참여한 모든 사용자에게 전달된다.

```
eb.registerHandler(INTERNAL_TCP_SERV_ADDR, new Handler<Message<JsonObject>>() {
    @Override
    public void handle(final Message<JsonObject> message) {
        JsonObject internalBroadcastData = message.body();
        String rmID = (String) internalBroadcastData.removeField("rmID");
        String whisperTarget = (String) internalBroadcastData.removeField("whisperTarget");
        String writerHandlerID = (String) internalBroadcastData.removeField("writerHandlerID");
        writeAll(internalBroadcastData, rmID, whisperTarget,
writerHandlerID);
    }
});
```

6.4.3 TCP 채팅 클라이언트 귓속말

[6.3.2 TCP 채팅 클라이언트 대화방 분리](#)의 TCPChatWithRoomClientVerticle을 기본으로 일부 기능을 수정해 귓속말 기능을 구현해 보자(귓속말 기능이 추가된 TCP 채팅 클라이언트는 TCPChatWithWhisperClientVerticle이다).

사용자 입력 메시지를 이벤트 버스로 받아 처리하는 부분을 다음과 같이 수정한다.
기존 TCPChatWithRoomClientVerticle에 귓속말 처리를 위한 WHISPER라는 messageType 구분 코드를 추가한 것 외에는 달라진 점이 없다

귓속말 메시지는 '/whisper'로 시작되며 콘솔로 입력받은 수신자 아이디를 whisperTarget이라는 파라미터명으로 TCP 채팅 서버에 전송해야 한다.

'/whisper 수신자아이디 굿속말메시지'

최종으로 굿속말 명령은 다음 JSON 객체로 만들어져 TCP 채팅 서버로 전송된다.

```
{"messageType":4,"rmID": <rmID>,"sessionID": <sessionID>,"whisperTarget":  
<whisperTarget>,"body": {"username": <username>,"message": <whisper  
message>}}
```

또한, 사용자 입력 메시지 부분을(body.message 부분) '/whisper' 명령이 포함된 전체 메시지로 처리할 필요가 없으므로 MessageParser의 arrayToString() 메서드를 이용해 전체 메시지에서 굿속말 메시지 부분만 추출해 사용자 입력 메시지를 다시 설정하도록 한다.

```
eb.registerHandler("com.devop.vertx.chat", new Handler<Message<JsonObject>>() {  
    @Override  
    public void handle(Message<JsonObject> input) {  
        if (writeHandlerID != null) {  
            JsonObject message = input.body();  
            JsonObject body = message.getObject("body");  
            if (message.getBoolean("exit", false)) {  
                container.exit();  
            }  
            else {  
                if (sessionId != null) {  
                    message.putString("sessionId", sessionId);  
                    body.putString("username", username);  
                    String line = body.getString("message");  
                    String[] tokens = line.split(" ");
```

```

        if ("/enter".startsWith(tokens[0]) && tokens.length == 2 &&
rmID == null) {
            message.putNumber("messageType",
MessageType.ENTER.ordinal());
            message.putString("rmID", tokens[1]);
}
else if ("/leave".startsWith(tokens[0]) && tokens.length ==
2 && rmID != null) {
            message.putNumber("messageType",
MessageType.LEAVE.ordinal());
            message.putString("rmID", tokens[1]);
}
else if (rmID != null) {
            message.putString("rmID", rmID);
if ("/whisper".startsWith(tokens[0]) && tokens.length
>= 3 ) {
            message.putNumber("messageType",
MessageType.WHISPER.ordinal());
            message.putString("whisperTarget", tokens[1]);
            message.getObject("body").putString("message", Mess
ageParser.arrayToString(tokens, 2, tokens.length));
}
else {
            message.putNumber("messageType",
MessageType.CHAT.ordinal());
}
}
else {
        System.out.println("you must enter roomID");
        return;
}
}

```

```

        else {
            if (username == null)
                username = message.getString("username");
        }

        byte[] data;
        try {
            data = message.toString().getBytes("UTF-8");
            Buffer buffer = new Buffer(4+data.length);
            buffer.setInt(0, data.length);
            buffer.setBytes(4, data);
            eb.send(writeHandlerID, buffer);
        } catch (UnsupportedEncodingException e) {
        }
    }
}
});
```

TCP 채팅 서버가 전송한 메시지를 받아 처리하는 RecordParser 객체에서 Payload를 처리하는 부분이 굿속말 메시지를 수신할 경우 굿속말 작성자와 함께 해당 메시지가 출력되도록 WHISPER messageType 구분 코드를 처리하는 내용을 추가한다.

…생략…

```

if (message.getMessageType() == MessageType.CHAT) {
    String from = message.getUsername();
    String msg = message.getBodymsg();
    System.out.println(from+": "+msg);
}

if (message.getMessageType() == MessageType.WHISPER) {
```

```
String from = message.getUsername();
String msg = message.getBodymsg();
System.out.println("whisper from "+from+": "+msg);
}
else if (message.getMessageType() == MessageType.ENTER) {
    if (message.statusOK()) {
        rmID = message.getRmID();
        System.out.println("enter the room '"+rmID+"'");
    }
}
...생략...
```

TCPChatWithLoginClientWorkerVerticle은 바로 변경할 필요가 없이 그대로 사용할 수 있다(그러나 클래스 이름의 규칙성을 위해 귓속말 클라이언트 Worker Verticle은 TCPChatWithWhisperClientWorkerVerticle라 하자).

6.4.4 SockJS EBB 채팅 서버 귓속말

6.3.3 SockJS EBB 채팅 서버 대화방 분리의 IntegratedSockJSEbbChat WithRoomServerVerticle을 기본으로 일부 기능을 추가해 귓속말 기능을 구현해 보자(귓속말 기능이 추가된 SockJS EBB 채팅 서버는 IntegratedSockJSEbbChat WithWhisperServerVerticle이다).

먼저 대화방 아이디별로 참여 중인 사용자 목록을 저장할 자료구조를 선언한다.

```
private final Map<String, Set<SockJSSocket>> rooms = new HashMap<>();
```

클라이언트가 연결된 SockJSSocket의 부가정보를 저장하기 위한 용도로 사용되는 SocketInfo 클래스를 추가한다.

```
private class SocketInfo {  
    private String username;  
    private String rmID;  
}
```

SockJSocket 별로 SocketInfo를 저장할 자료구조도 선언한다.

```
private final Map<SockJSocket, SocketInfo> sockInfos = new HashMap<>();
```

선언한 두 자료구조 rooms와 sockInfos는 각 Verticle이 독립적으로 관리하는 고유의 데이터이며 Verticle 사이에 공유되지 않는다.

SockJS EBB 채팅 서버의 특정 rmID에 해당하는 대화방 사용자 목록에서 궂속말 수신자를 찾는 메서드를 추가한다. 궂속말 수신자를 찾을 수 없으면 null을 반환한다.

```
private SocketInfo findByUsername(String username, String rmID) {  
    Set<SockJSocket> sockets = rooms.get(rmID);  
    if (sockets != null) {  
        for(SockJSocket socket : sockets) {  
            SocketInfo socketInfo = sockInfos.get(socket);  
            if (username.equals(socketInfo.username) &&  
                rmID.equals(socketInfo.rmID)) {  
                return socketInfo;  
            }  
        }  
    }  
    return null;  
}
```

다음으로 SockJSSocket의 상태 변화 이벤트에 대응하여 호출될 수 있는 몇 가지 유용한 메서드를 포함하는 EventBusBridgeHook 구현체를 작성해 보자. 이 메서드들은 TCP 채팅 서버에서의 처리 흐름과 최대한 비슷하게 구현한다.

TCP 채팅 서버에서의 처리 흐름과 비슷하게 구현하려면 첫 번째로 새로운 클라이언트가 연결되고 이를 위해 새로운 SockJSSocket이 생성되었을 때 호출되는 handleSocketCreated() 메서드를 작성한다. TCP 채팅 서버에서는 클라이언트가 연결된 NetSocket의 부가정보를 저장하기 위해 SocketInfo 객체를 생성하는 것이 주요 처리 흐름이었으므로 handleSocketCreated() 메서드에서도 새로운 SockJSSocket의 부가정보를 저장하기 위해 앞서 선언한 SocketInfo 객체를 생성한다.

```
public boolean handleSocketCreated(SockJSSocket sock) {
    if (origin!=null) {
        String originHeader = sock.headers().get("origin");
        if (originHeader == null || !originHeader.equals(origin)) {
            return false; //-- reject the socket
        }
    }
    SocketInfo socketInfo = new SocketInfo();
    sockInfos.put(sock, socketInfo);
    return true; //-- true to accept the socket, false to reject it
}
```

TCP 채팅 서버에서 NetSocket 연결이 종료되었을 때 대화방에서 종료된 NetSocket에 해당하는 TCP 채팅 클라이언트를 대화방에서 제거했다. 이에 대응되는 handleSocketClosed() 메서드를 다음과 같이 작성한다.

```
public void handleSocketClosed(SockJSocket sock) {
    logger.info("handleSocketClosed, sock = " + sock);
    SocketInfo socketInfo = sockInfos.remove(sock);
    if (socketInfo != null) {
        if (socketInfo.rmID != null) {
            Set<SockJSocket> users = rooms.get(socketInfo.rmID);
            if (users != null) {
                users.remove(sock);
            }
        }
    }
}
```

대화방에 입장/퇴장할 때는 대화방 아이디에 해당하는 대화방 사용자 목록에 새로운 사용자를 추가하거나 삭제하고, `SocketInfo`의 `rmID`를 설정했다. 이에 대응되도록 `handlePostRegister()` 메서드와 `handleUnregister()` 메서드도 다음과 같이 작성한다. `handlePostRegister()` 메서드는 SockJS EBB 채팅 클라이언트가 새로운 이벤트 버스 주소를 `Subscribe`할 때 호출되며, `handleUnregister()` 메서드는 `Subscribe`한 주소를 `Desubscribe`할 때 호출된다(각각 대화방 입장과 퇴장에 해당된다). `handlePostRegister()` 메서드와 `handleUnregister()` 메서드 구현 시 SockJS 채팅 클라이언트를 통해 활성화되는 이벤트 버스 주소를 파악하기 위한 고유 기능도 잊지 않도록 한다.

```
public void handlePostRegister(SockJSocket sock, String address) {
    logger.info("handlePostRegister, sock = " + sock + ", address = " + address);

    if (address.startsWith("whisper:")) return;

    SocketInfo socketInfo = sockInfos.get(sock);
    if (socketInfo != null) {
```

```

        Set<SockJSSocket> users = rooms.get(address);
        if( users == null ) {
            users = new HashSet<>();
            rooms.put(address, users);
        }
        users.add(sock);
        socketInfo.rmID = address;
    }

    Integer counter = addresses.get(address);
    if (counter == null) {
        counter = 1;
    }
    else {
        ++counter;
    }
    logger.info("address: " + address + ", counter: " + counter);
    addresses.put(address, counter);
}

public boolean handleUnregister(SockJSSocket sock, String address) {
    logger.info("handleUnregister, sock = " + sock + ", address = " + address);

    if (address.startsWith("whisper:")) return true;

    SocketInfo socketInfo = sockInfos.get(sock);
    if (socketInfo != null) {
        Set<SockJSSocket> users = rooms.get(address);
        if (users != null) {
            users.remove(sock);
        }
        socketInfo.rmID = null;
    }

    Integer counter = addresses.get(address);
    if (counter != null) {
        if (--counter == 0) {

```

```
        addresses.remove(address);
    }
    else {
        addresses.put(address, counter);
    }
}
logger.info("address: "+address+", counter: "+counter);
return true;
}
```

이 코드를 보면 ‘whisper:’ 문자열로 시작되는 이벤트 핸들러 주소는 아무런 처리도 하지 않는 것을 알 수 있다. 이 주소는 SockJS EBB 채팅 클라이언트별로 귀속말 메시지를 전달하기 위한 특별한 이벤트 버스 주소로 나중에 다시 설명하겠다.

EventBusBridgeHook 구현체의 마지막 메서드로 handleSendOrPub() 메서드를 다음과 같이 작성한다. handleSendOrPub() 메서드는 mod-auth-mgr의 로그인 기능 호출에서 사용자 아이디를 캡쳐하고 이를 SocketInfo의 username에 설정한다. 또한, 일반 채팅 메시지를 이벤트 버스를 통해 TCP 채팅 서버로 전달하고, 귀속말 메시지를 TCP 채팅 서버는 물론 자신을 포함한 모든 배포된 모든 SockJS EBB 채팅 서버 Verticle로 publish하는 역할을 한다.

```
public boolean handleSendOrPub(final SockJSocket sock, boolean send,
JsonObject msg, String address) {
    logger.info("handleSendOrPub, sock = " + sock + ", send = " + send +",
address = " + address);
    if (address.equals(AUTH_MGR_ADDRESS_PREFIX+".login")) {
        SocketInfo socketInfo = sockInfos.get(sock);
        if (socketInfo != null && socketInfo.username == null)
            socketInfo.username = msg.getObject("body").getString("username");
    }
}
```

```

else if (!address.startsWith(AUTH_MGR_ADDRESS_PREFIX)) {
    JSONObject body = msg.getJSONObject("body");
    String line = body.getString("message");
    String[] tokens = line.split(" ");
    if ("/whisper".startsWith(tokens[0]) && tokens.length >= 3 ) {
        body.putBoolean("whisper", true);
        body.putString("message", MessageParser.arrayToString(tokens, 2,
tokens.length));
        JSONObject internalBroadcastData = new JSONObject();
        internalBroadcastData.putNumber("messageType",
MessageType.WHISPER.ordinal());
        internalBroadcastData.putString("rmID", address);
        internalBroadcastData.putString("whisperTarget", tokens[1]);
        internalBroadcastData.putObject("body", body);
        eb.publish(INTERNAL_TCP_SERV_ADDR, internalBroadcastData);
        eb.publish(INTERNAL SOCKJS_SERV_ADDR, internalBroadcastData);
        return false;
    }
    body.putBoolean("whisper", false);
    JSONObject internalBroadcastData = new JSONObject();
    internalBroadcastData.putNumber("messageType",
MessageType.CHAT.ordinal());
    internalBroadcastData.putString("rmID", address);
    internalBroadcastData.putObject("body", body);
    eb.publish(INTERNAL_TCP_SERV_ADDR, internalBroadcastData);
}
return true; //-- true To allow the send/publish to occur, false otherwise
}

```

NOTE

mod-auth-mgr의 로그인 기능 호출에서 사용자 아이디를 캡쳐하는 방법은 사실 그다지 좋은 방법은 아니다. 이는 이 시점에서 사용자 아이디가 아직 인증이 이루어지지 않았기 때문이다. 따라서 좀 더 엄격한 사용자 인증을 통한 귓속말 기능을 구현하기 위해서는 인증된 사용자 아이디를 `SocketInfo`의 `username`에 저장하는 방법이 필요하다.

TCP 채팅 서버에서는 TCP 채팅 클라이언트의 로그인 요청을 수신하고, mod-auth-mgr의 로그인 기능을 대신 호출한다. 그리고 그 결과를 TCP 채팅 클라이언트로 전달하는데, 여기서 TCP 채팅 서버는 일종의 프록시로 동작하므로 자연스럽게 인증된 사용자 아이디를 `SocketInfo`의 `username`에 설정할 수 있다.

그러나 SockJS EBB 채팅 서버에서는 `handleAuthorise()` 메서드를 오버라이드하고 인증 방식을 변경하지 않는 한 SockJS EBB 채팅 클라이언트의 mod-auth-mgr의 로그인 기능 호출 결과를 확인할 방법이 없다.

앞의 코드에서 메시지가 귓속말일 때 `false`를 반환하여 이후 `org.vertx.java.core.sockj.EventBusBridge`에 의한 처리를 방지하는 부분을 주의 깊게 살펴봐야 한다([4.1 NOTE](#) 참고). 기본적으로 `org.vertx.java.core.sockj.EventBusBridge`는 입력 파라미터 `address`에 해당하는 이벤트 버스 주소를 `Subscribe`하는 모든 SockJS EBB 채팅 클라이언트로 메시지를 전달하는데, 귓속말 메시지는 이런 처리를 원하는 것은 아니므로 `false`를 반환해 처리가 되지 않게 한다. SockJS EBB 채팅 서버에서 귓속말 메시지 처리는 오직 상수 `INTERNAL_SOCKJS_SERV_ADDR`을 통해 실행되는 이벤트 핸들러에서 이루어지게 해야 한다.

마지막으로 상수 `INTERNAL_SOCKJS_SERV_ADDR`을 통해 실행되는 이벤트 핸들러를 다음과 같이 작성한다. 일반 채팅 메시지라면 그대로 `rmID`에 해당하는 대화방에 입장한 모든 사용자에게 메시지를 전달한다. 귓속말 메시지라면 귓속말 수신자를 찾고, 수신자를 찾으면 귓속말 메시지를 전달한다.

```
eb.registerHandler(INTERNAL_SOCKJS_SERV_ADDR, new
Handler<Message<JsonObject>>() {
    @Override
    public void handle(final Message<JsonObject> message) {
        JsonObject internalBroadcastData = message.body();
        MessageType messageType = MessageType.getMessageType(internalBroadcastData.getInteger("messageType", -1));
        String rmID = internalBroadcastData.getString("rmID");
        String whisperTarget = internalBroadcastData.getString("whisperTarget");
        JsonObject body = internalBroadcastData.getObject("body");

        if (messageType == MessageType.CHAT) {
            Integer count = addresses.get(rmID);
            if (count != null && count > 0 ) {
                eb.publish(rmID, body);
            }
        }
        else if (messageType == MessageType.WHISPER && whisperTarget != null) {
            String whisperer = body.getString("username");
            if (!whisperer.equals(whisperTarget)) {
                SocketInfo socketInfo = findByUsername(whisperTarget, rmID);
                if (socketInfo != null) {
                    eb.send("whisper:"+socketInfo.username+"://"+rmID, body);
                }
            }
            SocketInfo socketInfo = findByUsername(whisperer, rmID);
            if (socketInfo != null) {
                body.putString("whisperTarget", whisperTarget);
                eb.send("whisper:"+socketInfo.username+"://"+rmID, body);
            }
        }
    }
})
```

귓속말 메시지를 전달할 때 특이한 점은 귓속말 메시지를 전달하는 주소로 다음과 같은 포맷의 문자열을 사용한다는 것이다.

```
"whisper:"+socketInfo.username+"://"+rmID
```

예를 들어, 수신자 아이디가 ‘A’고 대화방 아이디가 ‘B’라면 귓속말 수신 주소는 ‘whisper:A://B’가 된다. 즉, 이벤트 버스 주소 중 ‘whisper’로 시작되는 주소로 귓속말을 전달하기 위해 예약된 주소라 할 수 있다.

NOTE_

사실 SockJSocket의 `writeHandlerID`를 통해 귓속말 메시지 전송이 가능하면 귓속말은 전송자와 수신자만 볼 수 있어야 한다는 보안 조건을 더 확실하게 만족할 수 있다. 또한, `handleSocketCreated()` 메서드에서 SockJSocket의 `writeHandlerID`를 얻어 `SocketInfo`에 저장하는 것도 어렵지 않다. 이 경우 TCP 채팅 서버에서 귓속말을 전달하기 위해 `findByUsername()` 메서드를 호출해 귓속말 수신자를 찾고, 해당 수신자의 `writeHandlerID`로 귓속말을 전달하는 것과 거의 똑같은 처리 흐름이 될 것이다.

그럼 왜 귓속말 메시지를 전달하기 위해 SockJSocket의 `writeHandlerID`를 사용하지 않는 것일까? 이는 `writeHandlerID`를 통한 메시지 전송은 이벤트 핸들러를 호출하지 않기 때문이다. 그 대신 `vertxbus-2.1.js`의 내부에 정의된 `sockJSConn.onmessage` 함수가 호출된다. 이 함수는 수신한 메시지를 처리할 이벤트 핸들러를 찾아 호출하는 방식으로 동작한다. SockJS EBB 채팅 서버에서 SockJSocket의 `writeHandlerID`를 통해 메시지를 전송하면 이에 해당하는 이벤트 핸들러는 정의되지 않은 상태인데, 이 이벤트 핸들러를 정의하려면 SockJSocket의 `writeHandlerID`를 알아야 한다. 그러나 SockJSocket의 `writeHandlerID`는 오직 SockJS EBB 채팅 서버에서만 관리되며, `vertxbus-2.1.js`에서 확인할 수 있는 방법이 없다.

따라서 `sockJSConn.onmessage` 함수의 기본 동작 방식을 변경하지 않는 한 SockJS EBB 채팅 서버에서 SockJSocket의 `writeHandlerID`를 통해 전송한 메시지는 SockJS EBB 채팅 클라이언트가 수신은 하지만 이를 처리할 방법이 없다.

6.4.5 SockJS EBB 채팅 클라이언트 굿속말

6.4.4 SockJS EBB 채팅 서버 굿속말에서 설명한 굿속말 수신 이벤트 버스 주소를 처리하려면 SockJS EBB 채팅 클라이언트의 index.html의 구현 일부도 수정되어야 한다. 먼저, Address InputBox에 정의된 주소로 이벤트 버스를 연결하기 위해 호출되는 subscribe() 메서드를 다음과 같이 수정한다. Address InputBox에 정의된 주소 외에 굿속말을 수신하기 위한 별도의 이벤트 버스를 연결하는 부분이 추가되었다.

```
function subscribe(address) {
    if (eb && eb.sessionID && !subscribed) {
        eb.registerHandler(address, handler);
        eb.registerHandler('whisper:' + username + '://' + address, handler);
        subscribed = true;
        $('#status-label').html($('#status-label').html() + 'Subscribe:' +
            address);
    }
}
```

그리고 일반 채팅 메시지와 굿속말 메시지를 구분해 출력하도록 이벤트 핸들러를 수정한다.

```
var handler = function(msg, replyTo) {
    console.log('message', msg);
    if (!msg.whisper) {
        $('#message-box').append(msg.username + ': ' + msg.message + '\n');
    }
    else {
        if (msg.whisperTarget != null) {
            $('#message-box').append('whisper to ' + msg.whisperTarget + ': ' +
                msg.message);
        }
    }
}
```

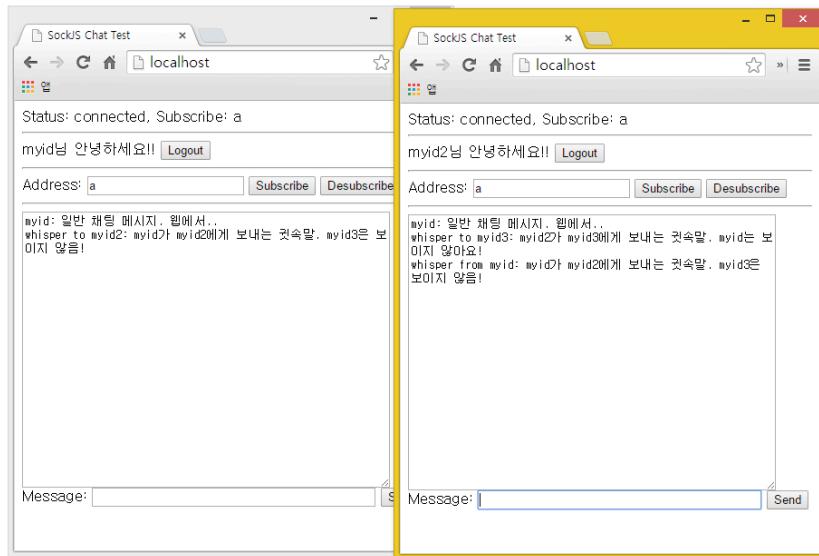
```

        '+msg.message+'\n');
    }
    else {
        $('#message-box').append('whisper from '+msg.username+':
        '+msg.message+'\n');
    }
}
};


```

간단한 테스트를 위해 TCP/SockJS 통합 채팅 서비스를 시작하고, 각 채팅 클라이언트를 실행해 보자. [그림 6-2]는 SockJS EBB 채팅 클라이언트에서 ‘a’라는 대화방의 사용자들이 일반 채팅 메시지와귓속말 메시지를 주고받는 모습이다.

[그림 6-2] TCP/SockJS 통합 채팅 서비스 귀속말 결과 화면



다음은 Windows 8의 PowerShell에서 TCPChatWithWhisperClientVerticle을 실행하고 로그인한 다음 대화방에 입장하여 일반 채팅 메시지와 귓속말 메시지를 전송하거나 수신했을 때 출력된 결과다. [그림 6-2]와 마찬가지로 ‘a’라는 대화방에서 일반 채팅 메시지와 귓속말 메시지를 주고받는 것을 확인할 수 있다.

```
PS C:\Users\yeon> vertx run TCPChatWithWhisperClientVerticle.java
Succeeded in deploying verticle
connect result: true
input username: myid3
input password: 1234
user 'myid3' login success at Fri Nov 14 01:31:43 KST 2014
/enter a
enter the room 'a'
myid: 일반 채팅 메시지. 웹에서..
whisper from myid2: myid2가 myid3에게 보내는 귓속말. myid는 보이지 않아요!
```

6.5 채팅 메시지 저장

이번에는 채팅 메시지를 저장하는 기능을 추가해 보자. 채팅 메시지 저장소는 MongoDB 또는 MySQL을 사용한다. 필요에 따라 더 다양한 저장소를 지원해야 할지도 모른다. 따라서 저장소를 쉽게 추가할 수 있는 구조가 되는 것이 좋다.

채팅 메시지의 저장은 대화방 아이디별로 이루어지고 귓속말 메시지는 저장하지 않는다. 채팅 메시지를 저장하는 기능을 추가하면 나중에 이전 대화 내용 보기 같은 기능을 추가하는 데 응용할 수 있다.

6.5.1 채팅 메시지 저장 모듈

가장 먼저 채팅 메시지를 저장할 구조를 결정해야 한다. 저장소로 MongoDB와 MySQL을 사용하므로 각 저장소에 다음 컬렉션과 테이블을 추가한다.

먼저 MongoDB에 `chat_message`라는 컬렉션을 추가한다. 이 컬렉션은 [그림 6-3]과 같은 document를 저장한다. 이 document는 사용자 아이디(username), 채팅 메시지(message), 대화방 아이디(rmID), 등록일시(regdate) 항목으로 구성된 간단한 구조다.

[그림 6-3] 채팅 메시지 저장을 위한 MongoDB 컬렉션

		Document
0 (...)	<code>_id</code>	dc8ea822-24c7-4713-85e8-d423c6f89111
	<code>username</code>	myid
	<code>message</code>	메시지!!
	<code>rmID</code>	com.devop.vertx.ch5.whisper
	<code>regdate</code>	2014-11-14 13:41:05
1 (...)	<code>_id</code>	28bd144f-bd65-4ef0-aab0-b5612e1e7b75
	<code>username</code>	myid
	<code>message</code>	채팅 메시지입니다.
	<code>rmID</code>	com.devop.vertx.ch5.whisper
	<code>regdate</code>	2014-11-14 13:41:21

이를 JSON 객체로 표현하면 다음과 같다.

```
{ "_id" : "dc8ea822-24c7-4713-85e8-d423c6f89111",
  "username" : "myid",
  "message" : "메시지!!",
  "rmID" : "com.devop.vertx.ch5.whisper",
  "regdate" : "2014-11-14 13:41:05"}
```

다음으로 MySQL에 `chat_message`라는 테이블을 추가한다. 이 테이블의 구조는 [그림 6-4]와 같다. `seq`는 자동 증가가 적용된 주키 Primary Key다. 그 외 항목은 MongoDB와 같다.

[그림 6-4] 채팅 메시지 저장을 위한 MySQL 테이블

Name	Type	Length	Decimals	Not null	
seq	bigint	20	0	<input checked="" type="checkbox"/>	 1
username	varchar	50	0	<input checked="" type="checkbox"/>	
rmiD	varchar	50	0	<input checked="" type="checkbox"/>	
message	varchar	255	0	<input checked="" type="checkbox"/>	
regdate	timestamp	0	0	<input checked="" type="checkbox"/>	

채팅 메시지를 저장할 데이터 구조 생성을 완료하면 채팅 메시지 저장을 위한 모듈을 작성한다. 이 모듈은 설정 파라미터를 통해 채팅 메시지 저장소로 MongoDB를 사용할지 MySQL을 사용할지 결정하게 된다. 따라서 이 모듈은 채팅 메시지 저장소에 따라 mod-mongo-persistor 모듈 또는 mod-mysql-postgresql 모듈에 의존성을 지닌다.

[코드 6-4] 채팅 메시지 저장 모듈(ChatMessagePersistorVerticle.java)

```

public class ChatMessagePersistorVerticle extends Verticle {
    public static final String INTERNAL_CHAT_MESSAGE_PERSISTOR_ADDR =
"com.devop.vertx.chat.internal.persistor";

    public static final String MONGODB_REPOSITORY_TYPE = "mongodb";
    public static final String MONGODB_REPOSITORY_ADDR =
"vertx.mongopersistor";
    public static final String MYSQL_REPOSITORY_TYPE = "mysql";
    public static final String MYSQL_REPOSITORY_ADDR =
"vertx.mysqlpersistor";

    public static final int REPOSITORY_OP_TIMEOUT = 1000;

    private Logger logger;
    private EventBus eb;

    private String address;
    private String repositoryType;
    private String repositoryAddress;

```

```

private int repositoryOpTimeout;

private Handler<Message<JsonObject>> persistorHandler;

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

private JsonObject saveOp(JsonObject param) {
    JsonObject body = param.getObject("body");
    String rmID = param.getString("rmID");
    String username = body.getString("username");
    String message = body.getString("message");
    if (rmID != null && username != null && message != null) {
        JsonObject op = new JsonObject();
        if (MONGODB_REPOSITORY_TYPE.equalsIgnoreCase(repositoryType)) {
            op.putString("action", "save");
            op.putString("collection", "chat_messages");
            op.putObject("document", new JsonObject()
                .putString("username", username)
                .putString("message", message)
                .putString("rmID", rmID)
                .putString("regdate", sdf.format(new Date())));
        }
        return op;
    }
    else if (MYSQL_REPOSITORY_TYPE.equalsIgnoreCase(repositoryType)) {
        op.putString("action", "prepared");
        op.putString("statement", "insert into chat_messages
(username,rmID,message,regdate) values(?, ?, ?, CURRENT_TIMESTAMP())");
        JSONArray values = new JSONArray();
        values.addString(username);
        values.addString(rmID);
        values.addString(message);
        op.putArray("values", values);
    }
    return op;
}
}

return null;

```

```

    }

    private void persist(Message<JsonObject> message) {
        JsonObject op = saveOp(message.body());
        if (op != null) {
            eb.sendWithTimeout(repositoryAddress, op, repositoryOpTimeout, new
Handler<AsyncResult<Message<JsonObject>>>() {
            @Override
            public void handle(AsyncResult<Message<JsonObject>> result) {
                if (result.succeeded()) {
                    JsonObject reply = result.result().body();
                    if (reply.getString("status").equals("error"))
                        logger.error(reply.toString());
                }
                else {
                    ReplyException cause = (ReplyException) result.cause();
                    JsonObject error = new JsonObject().putString("address",
s", address)
                        .putNumber("failureCode", cause.failureCode())
                        .putString("failureType", cause.failureType().name())
                        .putString("message", cause.getMessage());
                    logger.error(error);
                }
            }
        });
    }

    @Override
    public void start() {
        JsonObject appConfig = container.config();
        logger = container.logger();

        eb = vertx.eventBus();

        address = appConfig.getString("address", INTERNAL_CHAT_MESSAGE_PERSIS

```

```

    TOR_ADDR);

    repositoryType = appConfig.getString("repository_type",
MONGODB_REPOSITORY_TYPE);

    repositoryAddress = appConfig.getString("repository_address",
MONGODB_REPOSITORY_ADDR);

    repositoryOpTimeout = appConfig.getInteger("repository_op_timeout",
REPOSITORY_OP_TIMEOUT);

    persistorHandler = new Handler<Message<JsonObject>>() {
        @Override
        public void handle(Message<JsonObject> message) {
            persist(message);
        }
    };
    eb.registerHandler(address, persistorHandler);
}
}

```

채팅 메시지 저장 모듈의 설정 항목은 다음과 같다.

[표 6-3] 채팅 메시지 저장 모듈 파라미터

파라미터	설명
address	채팅 메시지 저장 모듈의 이벤트 버스 주소다. 기본값은 상수 INTERNAL_CHAT_MESSAGE_PERSISTOR_ADDR로 정의되어 있다.
repository_type	채팅 메시지를 저장하기 위해 사용할 저장소를 지정한다. mongodb 또는 mysql을 지정할 수 있다. 기본값은 mongodb다.
repository_address	채팅 메시지를 저장하기 위해 사용하는 저장소 모듈의 이벤트 버스 주소다. 예를 들어, MongoDB를 사용한다면 'vertx.mongodb'가 기본값이 된다.
repository_op_timeout	채팅 메시지 저장 처리 타임아웃 값이다. 기본값은 1000(1초)이다.

채팅 메시지 저장 모듈 구현의 핵심은 `saveOp()` 메서드다. 이 메서드는 채팅 메시지 저장에 필요한 필수 파라미터를 입력받고, 데이터 저장소에 따라 채팅 메시지 저장 명령을 담고 있는 JSON 객체를 생성해 반환한다. 이후 데이터 저장소가 추가되면 추가된 저장소에 채팅 메시지를 저장할 수 있는 명령을 담은

JSON 객체를 반환하도록 이 메서드를 수정하면 된다. 다른 부분은 수정할 필요가 없다.

6.5.2 TCP/SockJS EBB 채팅 서비스 적용

6.4.2 TCP 채팅 서버 구속말의 IntegratedTCPChatWithWhisperServerVerticle과 6.4.4 SockJS EBB 채팅 서버 구속말의 IntegratedSockJSEbbChatWithWhisperServerVerticle에 다음 코드 한 줄을 추가하면 채팅 메시지 저장 기능이 구현된다.

```
eb.send("com.devop.vertx.chat.internal.persistor", data);
```

이 코드는 채팅 메시지 저장 모듈에 채팅 메시지를 전달하기 위한 이벤트 버스 호출 코드다. 첫 번째 파라미터로 입력된 문자열은 채팅 메시지 저장 모듈의 address 설정값이며, 두 번째 파라미터는 채팅 메시지 JSON 객체다. 채팅 메시지 JSON 객체는 반드시 대화방 아이디인 rmID와 사용자 아이디, 채팅 메시지를 담고 있는 body가 필요하다.

```
{"rmID": <rmID>, "body": {"username": <username>, "message": "chat message"}}
```

IntegratedTCPChatWithWhisperServerVerticle에 채팅 메시지 저장 코드를 다음과 같이 추가한다. SockJS EBB 채팅 서버로 채팅 메시지를 전송한 후 채팅 메시지가 저장된다(채팅 메시지 저장 기능이 추가된 TCP 채팅 서버는 IntegratedTCPChatWithPsersistorServerVerticle이다).

```
if (message.getMessageType() == MessageType.CHAT || message.getMessageType() == MessageType.WHISPER) {
```

```

    if (message.getBody() != null && message.getRmID() != null) {
        SocketInfo socketInfo = sockInfos.get(socket);
        if (socketInfo.rmID != null && socketInfo.rmID.equals(message.getRmID())) {
            if (message.getMessageType() == MessageType.CHAT) {
                eb.publish(INTERNAL_TCP_SERV_ADDR, message.getInteralChatBroadcastData(socket));
                //-- send to sockjs server
                eb.send(INTERNAL_SOCKJS_SERV_ADDR, message.getInteralChatBroadcastData(null));
                //-- save chat message
                eb.send("com.devop.vertx.chat.internal.persistor",
                        message.origin());
            }
        }
    }
}

```

IntegratedSockJSEbbChatWithWhisperServerVerticle에 채팅 메시지 저장 코드를 다음과 같이 추가한다. TCP 채팅 서버로 채팅 메시지를 전송한 후 채팅 메시지가 저장된다(채팅 메시지 저장 기능이 추가된 SockJS EBB 채팅 서버는 IntegratedSockJSEbbChatWithPersistorServerVerticle이다).

```

public boolean handleSendOrPub(final SockJSocket sock, boolean send,
JsonObject msg, String address) {
    logger.info("handleSendOrPub, sock = " + sock + ", send = " + send + ", "
            + address);
    if (address.equals(AUTH_MGR_ADDRESS_PREFIX + ".login")) {
        SocketInfo socketInfo = sockInfos.get(sock);
        if (socketInfo != null && socketInfo.username == null)
            socketInfo.username = msg.getObject("body").getString("username");
    }
}

```

```

else if (!address.startsWith(AUTH_MGR_ADDRESS_PREFIX)) {
    JsonObject body = msg.getJsonObject("body");
    String line = body.getString("message");
    String[] tokens = line.split(" ");
    if ("/whisper".startsWith(tokens[0]) && tokens.length >= 3 ) {
        // 중략
    }

    body.putBoolean("whisper", false);
    JsonObject internalBroadcastData = new JsonObject();
    internalBroadcastData.putNumber("messageType",
MessageType.CHAT.ordinal());
    internalBroadcastData.putString("rmID", address);
    internalBroadcastData.putObject("body", body);
    eb.publish(INTERNAL_TCP_SERV_ADDR, internalBroadcastData);
    //-- save chat message
    eb.send("com.devop.vertx.chat.internal.persistor",
internalBroadcastData);
}
return true; //-- true To allow the send/publish to occur, false otherwise
}

```

마지막으로 채팅 메시지 저장 기능을 추가한 통합 TCP/SockJS EBB 채팅 서비스를 실행하기 위한 IntegratedChatWithPersistorServerStarterVerticle 을 작성한다. 채팅 메시지 저장소로 MySQL을 사용한다면 주석으로 처리된 mysqlConf 부분을 풀면 된다. 그러나 mod-auth-mgr 모듈을 사용하고 이미 MongoDB를 사용하고 있으므로 MySQL보다는 MongoDB를 사용할 것을 권장한다.

NOTE

[코드 6-5]에서는 ChatMessagePersistorVerticle을 하나의 Verteicle로 프로젝트에 직접 포함해서 실행하고 있다. ChatMessagePersistorVerticle를 별도의 mod-chatmsg-persistor라는 모듈로 작성하여 모듈로 실행하는 것도 가능하다.

[코드 6-5] IntegratedChatWithPersistorServerStarterVerticle

```
public class IntegratedChatWithPersistorServerStarterVerticle extends Verteicle {  
    @Override  
    public void start() {  
        JsonObject authMgrConfig = new JsonObject();  
        authMgrConfig.putString("address", "vertx.basicauthmanager");  
        authMgrConfig.putString("user_collection", "users");  
        authMgrConfig.putString("persistor_address", "vertx.mongopersistor");  
        authMgrConfig.putNumber("session_timeout", 1800000); // --30min  
        container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final",  
                               authMgrConfig);  
  
        JsonObject mongoPersistorConfig = new JsonObject();  
        mongoPersistorConfig.putString("address", "vertx.mongopersistor");  
        mongoPersistorConfig.putString("host", "localhost");  
        mongoPersistorConfig.putNumber("port", 27017);  
        mongoPersistorConfig.putString("db_name", "my-chat");  
        mongoPersistorConfig.putNumber("pool_size", 10);  
        container.deployModule("io.vertx~mod-mongo-persistor~2.1.0",  
                               mongoPersistorConfig);  
  
        /*JsonObject mysqlConf = new JsonObject();  
        mysqlConf.putString("address", "vertx.mysqlpersistor");  
        mysqlConf.putString("connection", "MySQL");  
        mysqlConf.putString("host", "localhost");  
        mysqlConf.putNumber("port", 3306);*/  
    }  
}
```

```
    mysqlConf.putString("username", "youname");
    mysqlConf.putString("password", "yourpassword");
    mysqlConf.putString("database", "yourdb");
    container.deployModule("io.vertx~mod-mysql-postgresql_2.10~0.3.1",
mysqlConf);*/
    JsonObject chatMsgPersistor = new JsonObject();
    chatMsgPersistor.putString("address",
"com.devop.vertx.chat.internal.persistor");
    chatMsgPersistor.putString("repository_type", "mongodb");
    chatMsgPersistor.putString("repository_address",
"vertx.mongopersistor");
    chatMsgPersistor.putNumber("repository_op_timeout", 1000);
    container.deployVerticle("com.devop.vertx.ch5.persistor.ChatMessagePe
rsistorVerticle", chatMsgPersistor);
    //container.deployModule("com.devop.vertx~mod-chatmsg-
persistor~0.0.1-SNAPSHOT", chatMsgPersistor)
    container.deployVerticle("com.devop.vertx.ch5.persistor.IntegratedSoc
kJSEbbChatWithPersistorServerVerticle");
    container.deployVerticle("com.devop.vertx.ch5.persistor.IntegratedTCP
ChatWithPersistorServerVerticle");
}
}
```

맺음말

지금까지 Vert.x의 소개부터 주요 개념과 철학, 채팅 예제 프로젝트를 통해 이를 응용하는 방법까지 알아보았다. Vert.x 그 자체의 러닝 커브 Learning Curve는 높은 편이 아니어서 Vert.x를 새롭게 시작하는 분들이 이 책을 통해 어려움 없이 기초를 다지고, 다양한 요구 조건이 있는 실무 프로젝트에서 Vert.x를 활용하며, 문제 해결을 위한 선택의 폭을 보다 넓힐 수 있는 계기가 되었으면 좋겠다.

그러나 Vert.x를 좀 더 심도 있게 이해하고 사용하기 위해서는 이 책의 내용만으로 만족해서는 안 된다. Vert.x의 내부 동작 원리와 분산 환경에서의 Vert.x 운영 방법 등 이 책에서 다루지 않은 내용이 많다. 또한, Vert.x는 지금도 빠르게 발전하고 있고, 3.0 버전에서는 여러 가지 새로운 기능이 추가될 예정이므로 Vert.x에 관심이 많은 독자는 [Github⁰¹](#)에 공개된 소스 코드를 분석하거나 [Vert.x Google Groups⁰²](#)를 구독하는 것도 좋은 연구 방법이 될 것이다.

마지막으로, 우리나라에서도 Vert.x의 사용층이 넓어져 활발하게 정보 교류가 일어나고, 멋진 Vert.x 애플리케이션을 만나볼 수 있기를 바라며 이 책을 마친다.

01 <https://github.com/eclipse/vert.x>

02 <https://groups.google.com/forum/#!forum/vertx>