13장. Swift로 Socket.IO를 이용한 iOS 채팅 앱 만들기

iOS앱은 이미 수백만 개가 존재하며 대부분 서버와 통신하여 데이터를 교환한다. 대다수의 경우, 서버는 앱이 통신에 사용할 수있는 RESTful API를 구현하고 제공한다. 앱이 서버에 데이터를 보내거나 서버에서 데이터를 가져와야하는 경우 적절한 요청을 하고 잠시 후 데이터가 반환된다. 앱 런타임 기간 동안 이 일은 여러 번 발생한다.

위 내용은 대부분의 사용 사례를 다루지만 전부는 아니다. 예를 들어 실시간으로 업데이트 해야 하는 일종의 뉴스 피드가 앱에 표시되어야 한다면 어떻게 해야할까? 혹은 사용자간 실시간 대화가 앱 기능으로 지원되어야 한다면 어떻게 될까? 한 가지 해결책은 앱이 새로운 데이터를 서버에 자주 요청하도록 만들어서 가능한한 빨리 새로운 데이터를 수집하는 것이다. 그러나 이는 두 가지 기본적인 이유 때문에 좋은 방법은 아니다. 첫째, 끝없이 요청을 반복해서 수행하면 말할 것도 없이 불필요한 자원 낭비가 발생하고, 이는 모바일에서 특히나 중요하다. 둘째, 요청간 짧은 간격이 설정 되더라도 항상 데이터가 제때 전송된다는 건 보장할 수 없다.

고맙게도 서버에서 데이터를 받을 필요가 있을 때(이런 데이터가 사용가능해질 때마다) 앱이 서버에 요청을 보내지 않고도 이를 가능케하는 더 좋은 솔루션이 있다. 이 솔루션은 웹 소켓을 사용하는 것을 기반으로 하며, 앞서 언급한 두 가지 문제를 완전히 해결한다. 이곳과 이곳에서 웹소켓에 관한 몇 가지 흥미로운 주제를 찾을 수 있지만 추가적인 정보가 필요하면 구글링 해보도록 하자. 필자는 간단한 단어로 큰 그림을 그려줄 것이다. 그러면 나중에 나오는 것을 더 잘 이해할 수 있다.

웹소켓 통신은 서버와 클라간의 지속적인 연결이 항상 존재하는 클라-서버 로직을 사용한다. 정확하게 말하자면, 서버는 클라가 연결되는 전용 포트를 열어둔다. 연결된 모든 앱이 해당 포트로 메시지를 보내고 (보내는 메시지) 수신 메시지 또한 listen할 수 있다. 그러면 소켓에 연결할 때마다 서버에 연결된 모든 클라이언트는 추가 요청없이 메시지를 자동으로 받는다. 가장 중요한 점은 메시지가 서버에 의해 해당 포트로 전송되면 받는 사람의 클라이언트가 즉시 해당 메시지를 수신하므로 추가작업(예: 뉴스피드 업데이트)을 즉시 수행할 수 있다는 것이다.

주의사항 : 지금부터 간단히 하기 위해 "websocket (s)"대신 "socket (s)"이라는 용어를 주로 사용한다. 일반적으로 "클라이언트"라는 용어는 웹, 모바일 또는 데스크톱 응용 프로그램과 같이 서버에 연결할 수 있는 모든 응용 프로그램을 의미한다. 하지만 이 텍스트에서는 "클라이언트"이라고 말하면 구체적으로 iOS 응용 프로그램과 데모 응용 프로그램을 지칭한다.

소켓에 대한 연결을 구현하고 이를 통해 서버와 통신하는 것이 제일 쉬운 해결책은 아니다. 비록 전체적인 아이디어가 매혹적으로 들릴지라도, 적절한 의사소통을 달성할 수 있을 때까지는 몇 가지 문제가 존재할 수 있다. 고맙게도 (다시 한번 말하지만), 여기서는 배후의 모든 연결 문제를 담당하는 아주 편한 프레임 워크를 제공하고, 소켓 기반의 통신을 쉽게 만든다. 그리고 이를 Socket.IO라고 한다.

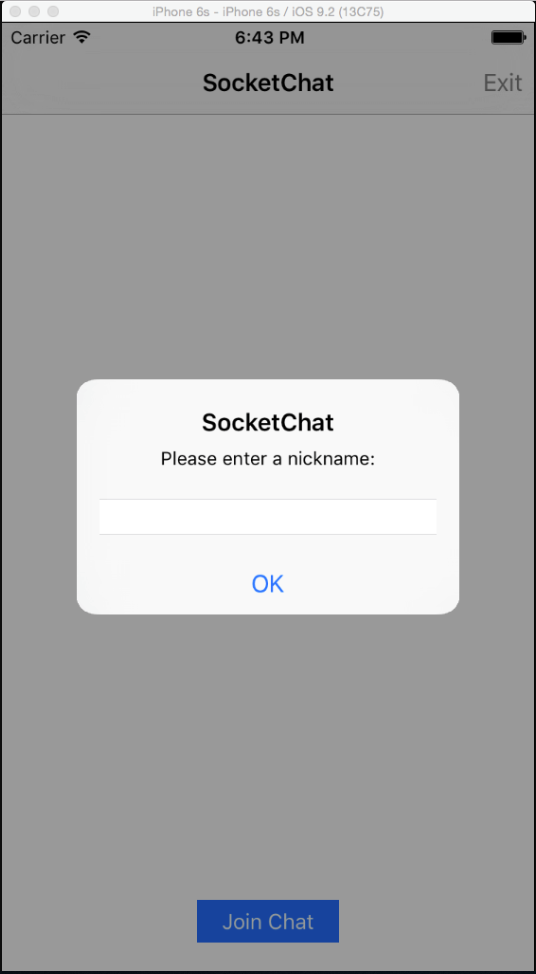
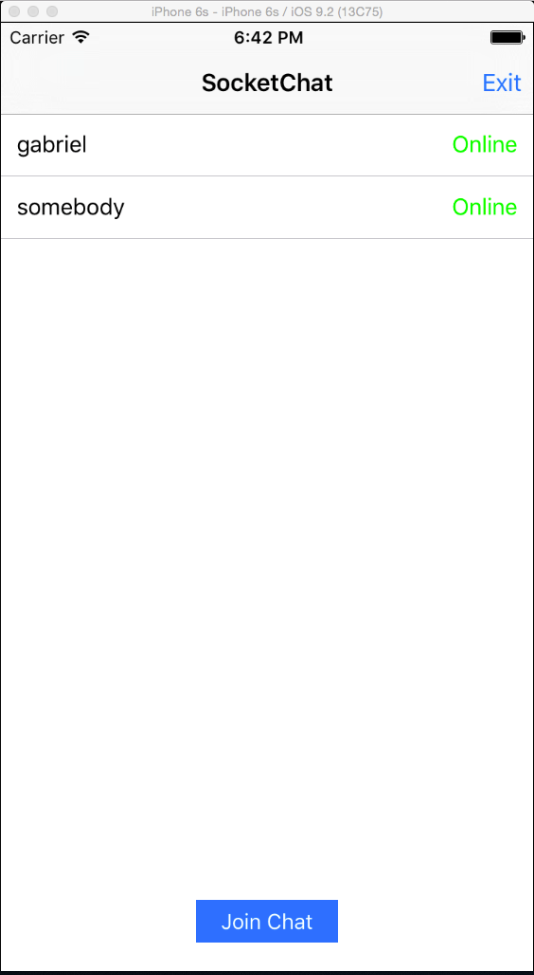
최근 큰 프로젝트에서 Socket.IO와 함께 일하게 된 것은 사실이다. 그리고 이 때 서버와 양방향 통신을 하고 앱에 즉시 메시지를 보내는 것이 얼마나 쉬운지를 볼 기회가 있었다. 여러분도 시간을 내서 공식 웹 사이트를 들어가서 무엇이 무엇이며 어떻게 작동하는지 이해하기를 바란다. Socket.IO는 주로 웹 어플리케이션용으로 설계 되었지만 이 프로젝트에 바로 사용될 수 있는 iOS용 라이브러리도 제공한다.

위 모든 내용을 염두에 두고 이 튜토리얼에서 Socket.IO의 맛보기를 소개하고자 한다. 서버와 메시지를 교환할 수 있도록 메시지를 통합하고 사용하는 방법을 보여 주고 예제를 통해 클라이언트-서버 통신의 작동 방식을 설명하고자 한다. 소개가 너무 길어졌으니 이 부분은 나중에 자세히 설명하겠다. 마지막으로 서버 측 부분은 localhost에서 작동한다. 그렇게 하기위해 필요한 모든 세부사항과 코드를 제공 할 것이다.

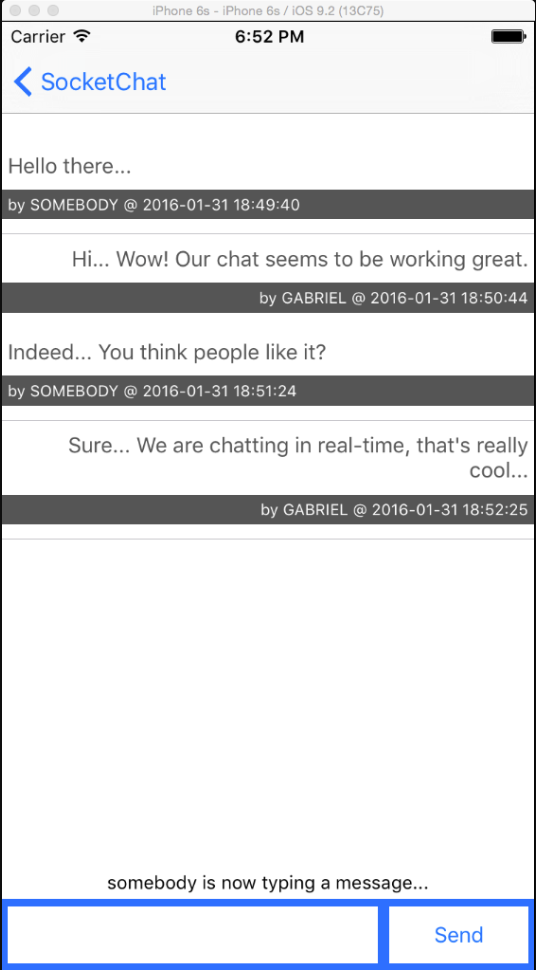
13-1. Demo App Overview

iOS 용 Socket.IO 라이브러리 (클라이언트)를 사용하는 방법과 서버와 실시간 통신을 수행하는 방법을 보여주기 위해 데모 앱으로 채팅 응용 프로그램을 만드는 것이 가장 좋았다. 물론 이 자습서에서는 중요한 부분에만 초점을 맞추고자 하므로 처음부터 모든 것을 만들지는 않을 것이다. 따라서 일단 작업 할 시작 프로젝트를 다운로드 하자.

뒤에서 모든 것을 자세하게 볼 수 있지만 일단 여기에서 우리의 목표에 대한 간략한 설명을 드리겠다. 첫째, 앱은 두 개의 ViewController로 분리된다. 첫 번째 경우 사용자는 닉네임만 제공하여 앱에 '로그인'할 수 있다(복잡한 로그인 메커니즘은 피할 것이다). 이렇게 되면 기존의 모든 사용자가 테이블 뷰에 나열되고 이름옆에 연결상태(온라인 또는 오프라인)가 표시된다. tableview 바로 아래에 채팅을 수행하기 위해 다음 View Controller로 이동하는 "Join Chat"버튼이 있다.

두 번째 ViewController는 채팅 기능 전용이다. 메시지를 보내고 받는것 외에도 앱에 몇 가지 흥미로운 기능을 추가할 것이다. 어떤 것이 있냐면, 새로운 사용자가 채팅에 연결되거나 기존 사용자가 채팅을 떠날 때 팝업 라벨이 표시되는 기능이 있다. 또한 한 명 이상의 유저가 메시지를 입력하면 다른 사용자에게 텍스트 뷰 바로 위에 나타나는 레이블로 다른 사용자에게 이를 알리는 기능도 추가한다.



이 튜토리얼의 서버 사이드 또한 흥미롭다. 작동시키기 위해서는 몇 가지 조치가 필요하기 때문이다. 실제로 로컬 호스트는 앱을 테스트 할 수 있는 가장 간단하고 안전한 방법으로 구성되며 원하는 경우 선택적으로 서버측을 변경할 수도 있다. 다운로드 해야하는 Starter Project외에 다른 .zip 파일도 여기에서 가져와야 한다. 일단 지금은 잔말말고 파일을 다운받자. 좀 이따 사용하는 방법을 알게 된다. 서버 사이드 구현에 전혀 관여하지 않고 앱을 테스트할 최종 프로젝트를 얻었더라도 서버 사이드는 앱을 실행하려는 경우 필수적으로 수행해야 하는 작업이다. 또한 이 자습서에서는 서버측 코드를 전혀 표시하지 않겠지만 서버 부분에 관해서는 상당히 자주 이야기 할 것이다. 모든 단계에서 우리가 서버와 주고받는 데이터의 종류를 명확히하는 것이 중요하기 때문이다.

이해했듯이, 이 채팅 앱은 iOS 개발에 대한 논의가 너무 깊지 않고 적절한 수준까지만 유지될 수 있도록 하나의 채팅방만 제공할 것이다. 그 이상으로 앱의 기능을 확장하는 것은 여러분의 몫으로 남겨두겠다. 또한 서버를 변경하거나 새로운 기능을 추가하거나하는 여기서 언급되지 않는 Socket.IO 기능을 알고싶으면 공식 웹 사이트에서 제공되는 모든 설명서를 참조하면 된다. 볼 거리가 완전 많을 것이다. 필자는 Getting Started 가이드를 따라 서버측 코드를 작성하고 그 이후에 몇 가지 새로운 사항을 추가했다. Socket.IO 웹 사이트에서 시간을 보낼수록 많이 성장할 것이다.

일단 필요한 모든 것을 다운로드 하고 준비가 되면, 그 다음으로 넘어가자.

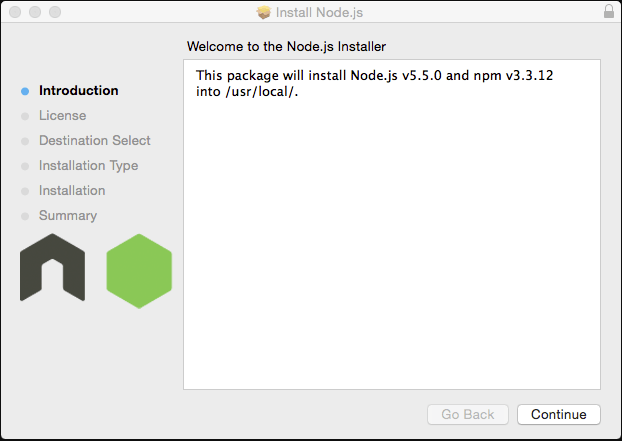
13-2. Getting Started: The Server-Side Part

정상적인 상황에서는 서버 구현과 관련하여 할 일이 없다. 물론 여러분이 iOS 앱 외에는 해야할 일이 없다면 말이다. 웹 개발자는 여러분을 위해 서버를 설정하고 코드를 짤 것이므로, 여러분이 할 일은 클라이언트 사이드의 통신 부분뿐이다. 사실, 여러분이 알아야 할 것은 딱 두 가지뿐이다.

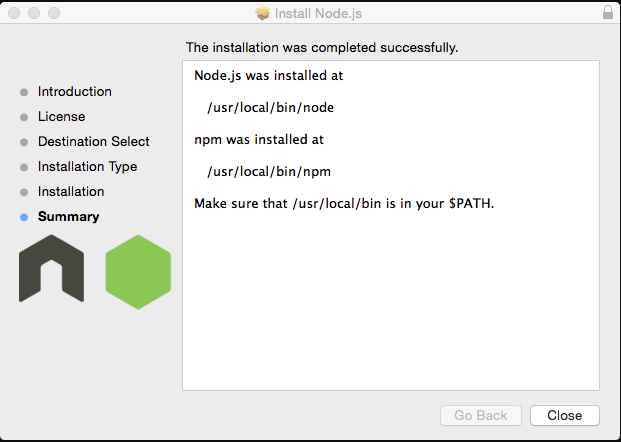
1. connection details이란 앱이 연결해야 하는 서버의 URL과 포트 번호를 의미한다는 점.
2. 앱이 서버와 교환할 메시지와 파라미터에 무엇이 있는지.

그러나 여기서는 그렇지 않다. 최소한의 노력만 들이긴 하겠지만 이 자습서에서도 서버 사이드를 처리하긴 할 것이다. 다만 이것은 iOS 튜토리얼이며 웹 내용에 관심이 없기 때문에, 가장 빨리 시작할 수 있는 방법으로 서버를 로컬로 설치하여 우리의 목적에 맞게만 작동시킬 것이다.

세부사항에 대해 자세히 알아보려면 시스템에 Node.js를 설치해야 한다. 그러면 서버 및 Socket.IO 라이브러리가 작동할 수 있다. 제공된 모든 서버측 코드는 Javascript로 작성되었다. Node.js를 다운로드하려면 https://nodejs.org/en/ 로 이동하여 stable 버전을 받으면 된다. 패키지가 다운로드되면 더블 클릭하여 패키지를 열고 설치 과정을 시작하자.

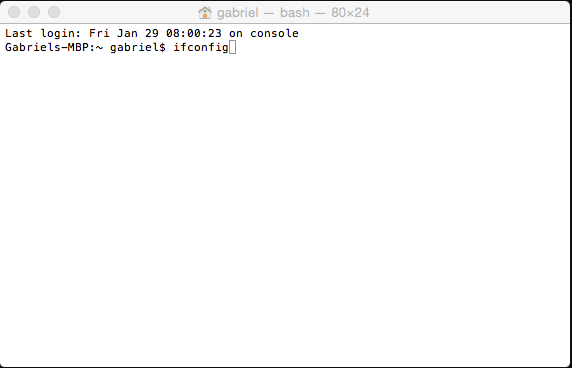


화면의 지시사항을 따라 끝내자. 설치가 끝나면 이 안내서는 Node.js와 패키지관리자인 npm이 설치된 곳을 다음처럼 알려주는데, 이 정보는 Node.js와 npm을 제거하고 싶을 때 유용하다. 근데 이거야 나중에 따로 찾더라도 쉽게 찾을 수 있으니까 굳이 어딘가에 적어둘 필요는 없다.

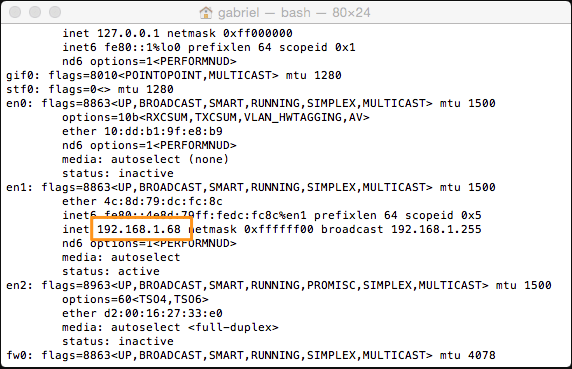


이제 Starter Project와 함께 다운로드한 srv-SocketChat.zip 파일의 압축을 풀고, 쉽게 접근할 수 있도록 압축풀기된 폴더를 바탕화면에 놓자. 샘플 서버 작업을 수행하는 데 필요한 모든 패키지와 코드는 폴더에 있으므로 궁금하면 간단히 훑어보자.

이제 터미널을 열어서 내부 네트워크에서 로컬의 IP 주소를 검색하기 위해 ifconfig 명령을 입력해보자.



터미널 창에서 네트워크에 관한 로그가 길게 나오는데, 다음 그림과 같이 와이파이에 접속되어 있다면 192.168.X.X 형태의 IP 주소를 찾으면 된다. 만약 와이파이에 접속되어 있지 않다면 다른 형식의 IP 주소를 찾으면 된다.



발견했으면 좀 있으면 iOS 프로젝트에서 필요하게 될테니 기억해두자. 물론 나중에 다시 ifconfig명령어를 이용해서 찾아봐도 된다. 이제 다음 명령어를 통해 바탕화면의 압축풀린 폴더로 들어가자.

cd Desktop/srv-SocketChat

폴더명이 다르면 당연히 위 명령도 폴더이름에 맞게 고쳐야 한다.

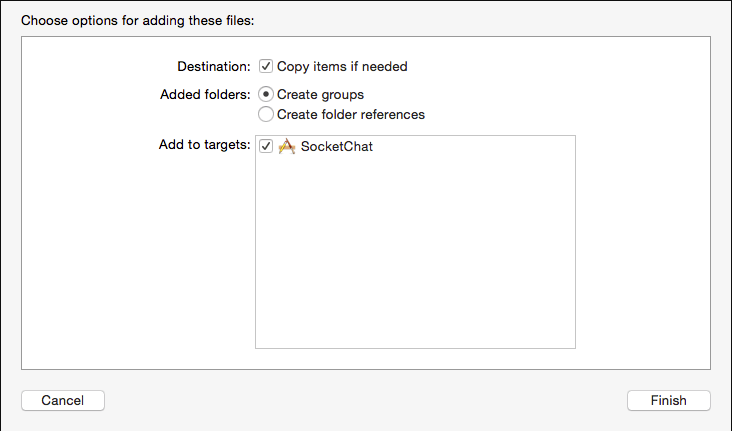
그리고 이제 서버를 켜보자.

node index.js

"listening on \* : 3000"이라는 메시지가 나타나면 서버가 실제로 실행 중인 것이다. 이는 서버가 이제 3000번 포트번호를 사용중이며 메시지를 주고받을 준비가 되었음을 의미한다. 실행을 중지하려면 Ctrl-C를 누르면 된다. 또한 어떤 이유든 3000말고 다른 번호를 사용하고 싶으면 index.js 파일을 열어서 원하는 포트번호로 바꾸면 된다. 서버코드에 대한 변경사항은 서버를 껐다 켜야 적용된다.



socket.IO를 다운 받아 Xcode의 Project Navigator로 바로 드래그 앤 드롭한다 (물론 그 전에 SocketChat 스타터 프로젝트를 열었어야 한다). Xcode에서 물어보면, 가져 오기 전에 대상에 파일을 추가하는 옵션을 확인하자.



끝으로 Project Navigator에서 Source라는 새 그룹을 볼 수 있을 것이고, 새 파일 목록도 거기에 추가될 것이다.

13-3. Getting Connected to the Server

이제는 Socket.IO 라이브러리와 서버 통신과 관련된 메소드 구현을 전용으로 하는 새로운 클래스 작성을 시작할 것이다. 더 명확히 말하자면, 서버에 메시지를 주고 받는 모든 메소드를 모으는 곳으로 생각하면 된다. 결국 이 클래스는 소켓 관련 액션을 수행하는 메커니즘으로 작동한다.

Xcode에서 File > New > File ... 메뉴로 이동하거나 Command-N를 눌러보자. 파일의 템플릿으로 \_\_Cocoa Touch Class\_\_를 선택하고 NSObject 클래스의 하위 클래스로 만들고 이름을 \_\_SocketIOManager\_\_로 지정하자. 새 클래스를 만들고 Project Navigator에서 새로 만들어진 파일을 클릭하여 연다.

처음에는 이 클래스에 싱글톤 패턴을 적용하는 것이 프로젝트의 다른 코드에서도 쉽게 액세스할 수 있어서 편리하다. 스위프트에서는 그게 쉽기도 하다. 코드 한 줄이면 된다.

class SocketIOManager: NSObject {

static let sharedInstance = SocketIOManager()

}

이렇게 해 두면 코드 어디서나 SocketIOManager.sharedInstance라고 써서 위의 클래스의 모든 public 속성 및 메서드에 액세스할 수 있다.

그 외에도, init 메소드가 필요하다.

override init() {

super.init()

}

이제 SocketIOClient 타입의 프로퍼티를 선언해야 한다. 이는 서버에서 메시지를 주고받을 수 있게 해주는 Socket.IO의 기본 클래스이다. 스위프트는 충분히 유연하기 때문에 명령어 한 줄로 초기화할 수 있다. 초기화의 인자로는 컴퓨터의 IP 주소와 지정된 포트를 써넣으면 된다.

var socket: SocketIOClient = SocketIOClient(socketURL: NSURL(string: "http://192.168.1.XXX:3000")!)

말할 필요도 없지만, 192.168.1.XXX에는 여러분의 IP주소를 넣으면 된다. 만약 서버의 index.js파일에서 포트번호도 다른 것으로 설정해 두었으면 3000 대신 그걸로 바꾸면 된다.

위 socket 프로퍼티를 사용하기 위해 두 가지 메소드를 정의하자. 한가지는 앱을 서버에 연결하기 위한 것이고 다른 한가지는 연결을 끊기 위한 것이다.

func establishConnection() {

socket.connect()

}

func closeConnection() {

socket.disconnect()

}

알다시피 socket.connect()와 socket.disconnect()만 쓰면 끝이다. 연결하는데 필요한 여러가지 복잡한 과정때문에 생기는 번거로움은 Socket.IO가 전부 처리해주니까 굉장히 편리할 것이다.

이제 위 두 메소드를 어떻게 사용해야 할까? 우리는 앱이 active될 때마다 서버와 연결할 것이고, 앱이 background로 들어갈때 연결을 끊을 것이다. 간단하게 AppDelegate.swift 파일을 열어서 delegate 메소드에서 위에서 정의한 첫 번째 메소드를 call하자.

func applicationDidBecomeActive(application: UIApplication) {

SocketIOManager.sharedInstance.establishConnection()

}

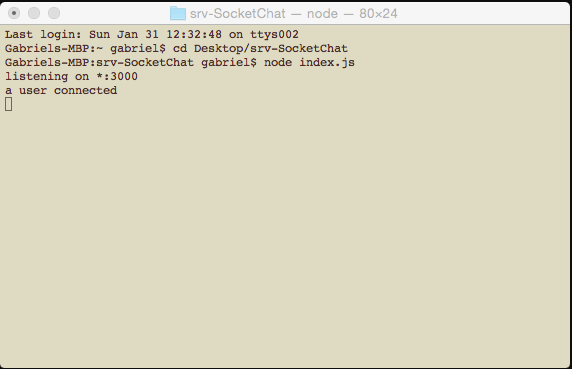
비슷하게 연결을 끊는 것도 만들자.

func applicationDidEnterBackground(application: UIApplication) {

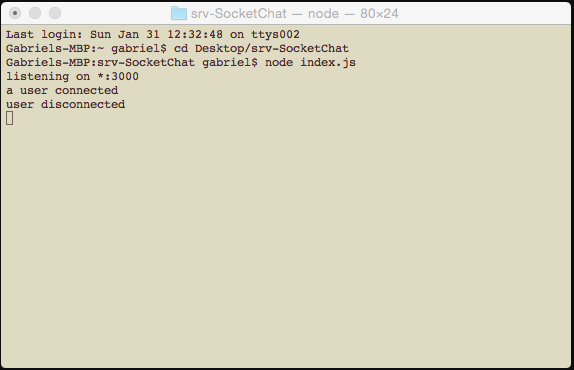
SocketIOManager.sharedInstance.closeConnection()

}

이것이 이 튜토리얼에서 Socket.IO 라이브러리를 접해본 첫 번째 사례다. 아직 앱 내에 내용은 없지만, 지금 (아이폰 혹은 시뮬레이터를 선택해서 프로젝트를 실행하고) 서버를 실행중인 터미널 창으로 이동하면 새 클라이언트가 연결되었다는 메시지가 표시된다. 왜냐하면 우리가 처음으로 성공적으로 연결을 구축한 거니까 기뻐해도 된다.



앱을 끄면 연결이 끊겼다는 메시지 또한 볼 수 있다.



13-4. Connecting a user to the chat room

내가 데모 앱의 오버뷰에서 이미 언급한 바에 따르면, 우리는 하나의 채팅방만을 만들 것이다. 이는 새로 들어오는 유저들이 전부 그 곳에 들어갈 것이라는 것을 의미하고, 우리가 작성하는 대화가 전부 모두에게 보여짐을 의미한다. 이를 위해서, 우리는 앱과 상호작용 하기 전에 각각의 새로운 유저를 채팅방에 연결부터 해야한다. (일련의 로그인 과정)

이를 쉽게 만들기 위해, 복잡한 로그인 시스템대신 단지 각 유저에게 닉네임을 물어볼 것이고 두 가지 중요한 가정을 할 것이다.

1. 각 유저들은 unique한 닉네임으로 로그인한다. 우리가 특별히 작업하지 않아도 데모를 위해 동일 닉네임을 사용하는 두 명이상의 유저가 없을 것이라고 가정할 것이다.
2. 기존의 닉네임은 기존 유저가 재로그인할 때만 사용된다.

이 튜토리얼의 다음 논의를 돕기 위해서, 그리고 위의 두 번째 가정을 명확하게 만들기 위해서 서버 사이드 코드가 로그인된 모든 유저들에 대한 하나의 배열을 유지한다고 말할 수 있다. 사실, 그 배열은 각 유저 데이터의 세가지 내용을 저장한다.

* 포트에 연결할 때 서버에 의해 자동으로 할당되는 각 클라이언트의 unique ID
* 유저의 닉네임
* 연결 상태 (유저가 연결되어 있는지 아닌지에 따라 true, false)

유저가 앱을 종료하고 서버의 소켓과의 연결이 없어진다고 해서 유저가 배열에서 사라지는 것이 아니다. 대신, 연결이 끊겼다고 마크될 뿐이다. 유저가 서버 리스트에서 완전히 삭제되게 하려면, 유저가 직접 앱에서 \_exit\_버튼을 눌러야 한다.

서두르지 말고, 질서정연하게 생각해 보자. 가장 먼저 우리가 해야할 액션은 당연히 socket.io 라이브러리를 이용해서 유저의 닉네임을 서버에 보내는 것이다. *SocketIOManager.swift* 파일을 열어서 다음 메소드를 정의하자.

func connectToServerWithNickname(nickname: String, completionHandler: (userList: [[String: AnyObject]]!) -> Void) {

}

닉네임을 서버에 보내는 것은 다음의 한줄이면 충분하다.

func connectToServerWithNickname(nickname: String, completionHandler: (userList: [[String: AnyObject]]!) -> Void) {

    socket.emit("connectUser", nickname)

}

socket object의 emit(...) 메소드는 우리가 Socket.IO 클라이언트 라이브러리를 사용해서 어떤 메시지든 서버에 보내기 위해 필요로 하는 메소드이다.

위 line은 단순히 서버에 \_\_connectUser\_\_라는 이름의 single 메시지를 보내는 것이고, nickname이라는 한 개의 파라미터를 같이 보낸다. 반면, 서버는 소켓으로 들어오는 메시지를 듣고 있다가, 메시지가 도착하면 당장 다음의 작업을 한다.

1. 유저가 새로운 유전지 아닌지 판단한다. 새로운 유저라면 내부의 배열(client ID, nickname, 연결상태)를 저장하고, 아니면 연결 상태를 true로 업데이트 한다.
2. 업데이트 된 유저 리스트를 return한다.

주의 : 이 부분의 서버 사이드 액션을 srv-SocketChat폴더의 index.js에서 볼 수 있다. clientSocket.on("connectUser" ...)함수를 찾아봐라.

내가 말한 것에 기반해서, 다음 할 일은 유저 리스트와 관련된 메시지든 소켓을 통해 듣고, 서버로부터 답이 오면 그를 잡는 것이다. 이를 위해서, 우리는 아래처럼 socket object의 on(...) 메소드를 사용해야 한다.

socket.on("SomeMessage") { ( dataArray, ack) -> Void in

}

첫 번째 인자는 우리가 관심을 가진 메시지의 literal이다. 여러분이 서버 개발자가 아니라도, 여러분이 대응해야 하는 메시지 리터럴은 모두 알고 있을 것이다. 여기에서 유저 리스트로 들어왔으면 하는 그 리터럴은 “userList”이다.

두 번째 인자는 두 개의 파라미터를 가진 closure이다. 첫 번째는 *NSArray* 객체이며, 이 객체의 크기는 서버에서 보낸 결과의 수에 따라 달라진다. 서버에서 single value를 리턴하더라도 클로져의 첫 번째 인자는 무조건 배열임을 기억하자(single value가 오면 배열의 첫 번째 인자에 존재한다). 두 번째 파라미터는 서버에게 메시지를 수신했다는 정보를 알리는데 쓰일 수 있는 *acknowledge* 메시지 이다. 물론 이 파라미터들의 네이밍은 여러분에게 달려 있고, dataArray, ack는 여기서 필자가 정한 것일 뿐이다.

예시로 돌아가서, 앱이 유저리스트 메시지를 listening하게 만들자.

func connectToServerWithNickname(nickname: String, completionHandler: (userList: [[String: AnyObject]]!) -> Void) {

...

socket.on("userList") { ( dataArray, ack) -> Void in

completionHandler(userList: dataArray[0] as! [[String: AnyObject]])

}

}

유저 리스트가 서버로부터 수신되면, 유저리스트를 인자로 가지고 있는 completion handler를 콜한다. 객체에 대한 dictionary들의 배열인데, 이것이 위의 변환이 필요한 이유다.

중요한 부분 : 위의 socket.on(...) 메소드는 서버가 유저리스트를 보낼때마다 Socket.IO에 의해 자동으로 콜 된다. 다른 방법으로는, 앱이 userList 메시지를 끊임없이 listening하고 있으면서, userList 하나가 도착하면 completion handler를 콜하게 만들 수도 있다. 물론, 이 모든 activity는 앱이 서버에서 연결이 끊어질 때 종료된다.

13-5. Displaying users

유저를 채팅방에 연결하는 메카니즘을 구현했고, 유저 리스트를 받아왔기 때문에 이제 이를 사용할 차례다. 먼저 UsersViewController.swift 파일을 열고, 새로운 custom method를 추가하자. 이 메소드는 앱이 켜졌을 때 유저가 nickname을 타입해넣을 수 있는 textfield와 함께 alert constroller를 보여주는 메소드이다.

아래가 해야 할 구현이다. 복붙하도록 하자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | func askForNickname() {  let alertController = UIAlertController(title: "SocketChat", message: "Please enter a nickname:", preferredStyle: UIAlertControllerStyle.Alert)  alertController.addTextFieldWithConfigurationHandler(nil)  let OKAction = UIAlertAction(title: "OK", style: UIAlertActionStyle.Default) { (action) -> Void in  }  alertController.addAction(OKAction)  presentViewController(alertController, animated: true, completion: nil)  } |

OKAction body 부분이 우리가 뭔가 로직을 추가할 곳이다. 먼저 alert controller의 textfield가 제대로된 값을 포함하고 있는지 아닌지 체크할 것이다. 만약 아니라면 우리는 recursion을 통해 같은 메소드를 한 번 더 call할 것이고 똑같은 alert controller가 다시 뜰 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | func askForNickname() {  ...    let OKAction = UIAlertAction(title: "OK", style: UIAlertActionStyle.Default) { (action) -> Void in  let textfield = alertController.textFields![0]  if textfield.text?.characters.count == 0 {  self.askForNickname()  }  else {    }  }    ...  } |

그러나 textfield가 값을 가지고 있다면, 우리는 먼저 nickname이라고 한 property에 먼저 저장할 것이다. 그리고 이전에 만들어둔 connectToServerWithNickname(\_:completionHandler:) 메소드를 call할 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | func askForNickname() {  ...    let OKAction = UIAlertAction(title: "OK", style: UIAlertActionStyle.Default) { (action) -> Void in  let textfield = alertController.textFields![0]  if textfield.text?.characters.count == 0 {  self.askForNickname()  }  else {  self.nickname = textfield.text    SocketIOManager.sharedInstance.connectToServerWithNickname(self.nickname, completionHandler: { (userList) -> Void in  dispatch\_async(dispatch\_get\_main\_queue(), { () -> Void in  if userList != nil {  self.users = userList  self.tblUserList.reloadData()  self.tblUserList.hidden = false  }  })  })  }  }    ...  } |

사용자 목록을 얻어서 비어 있지 않다는 걸 확인하면 UsersViewController class의 users property에 할당하고 tableview를 업데이트하고 볼 수 있게 만들자.

위의 코드를 통해 각 부분이 나머지 부분과 어떻게 조화롭게 상호작용 하는지 알 수 있다: 데이터를 주고 받는 이 클래스가 매개클래스로 SocketIOManager 클래스를 사용하고, 이를 통해 Socket.IO 라이브러리를 직접 처리한다.

어디선가는 위 메소드를 콜하도록 만들어야 한다는걸 잊지말자. 내가 보기에는, 가장 적절한 위치는 viewDidAppear(:\_) 메소드이다. UI가 적절하게 초기화 되었는지 확인할 수 있기 때문이다. 이렇게 만들자:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | override func viewDidAppear(animated: Bool) {  super.viewDidAppear(animated)    if nickname == nil {  askForNickname()  }  } |

nickname property가 nil일 때만 다시 묻는다는 점을 기억하자. 그 외에는 할 이유가 없다.

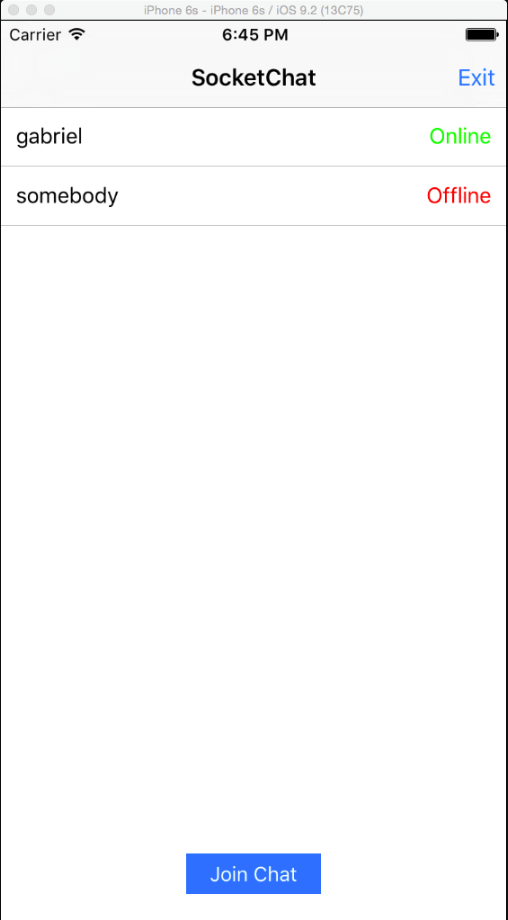
이제 유저의 정보를 tableview에 뿌릴 차례다. Starter Project에 관련된 메소드가 거의 완성되어 있는 tableview가 있지만 각 셀에 적절한 데이터를 보여줄 로직은 아직 빠져 있다. 아래를 보자:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCellWithIdentifier("idCellUser", forIndexPath: indexPath) as! UserCell  cell.textLabel?.text = users[indexPath.row]["nickname"] as? String  cell.detailTextLabel?.text = (users[indexPath.row]["isConnected"] as! Bool) ? "Online" : "Offline"  cell.detailTextLabel?.textColor = (users[indexPath.row]["isConnected"] as! Bool) ? UIColor.greenColor() : UIColor.redColor()  return cell  } |

**13-6. Leaving the chat room**

앞서 말한 것처럼, 유저가 앱 사용을 멈추거나 앱이 백그라운드로 들어갈 때 서버와의 연결이 끊어진다. 그러나 유저 정보는 서버의 레코드에 남아 있다(서버 프로그램이 아직 돌아가고 있는 한). 물론 해당 유저의 "isConnected" 플래그는 false일 것이다. 우리의 iOS앱에서 이는 Offline 표시로 유저의 tableview에 보여질 것이다.

필자가 말한걸 사실 지금 당장 테스트할 수 있다. 테스트를 위해 app을 디바이스든 시뮬레이터든 어디든 설치하고 실행해야 한다. 두 개의 디바이스(시뮬레이터와 진짜 디바이스)모두에 앱을 실행하여 각각 다른 닉네임을 입력하자. 연결되고 나면, 각 디바이스에 두 명의 유저가 보일 것이다. 둘 모두 Online 이라고 닉네임 옆에 표시되어 있다. 한 디바이스에서 앱을 종료하고 다른 디바이스에서는 그대로 두면, 종료된 디바이스의 유저가 목록에는 있지만 status가 Offline 으로 바뀐 것을 볼 수 있다.



서버가 유저들의 배열에서 한 유저를 완전히 삭제하게끔 하려면, 우리는 SocketIOManager.class 안에 새로운 custom 메소드를 만들어야 한다. 그 안에서 특정한 메시지를 우리가 지우고자 하는 유저의 닉네임 정보와 함께 서버에 emit 할 것이다. 우리한테 필요한건 그뿐이다.

그러니, SocketIOManager.swift 파일을 열고 아래의 새 메소드를 추가하자

|  |  |
| --- | --- |
| 1  2  3  4 | func exitChatWithNickname(nickname: String, completionHandler: () -> Void) {  socket.emit("exitUser", nickname)  completionHandler()  } |

서버는 "exitUser"라는 메시지를 받고나면 명시된 유저를 삭제해야 한다는 것을 즉시 이해한다.

UsersViewController.swift 파일로 돌아가서, 바로 exitChat(:\_) IBAction 메소드를 보자(역자 주 : exitUser(:\_)라고 해놓았는데 실수인 것 같다). 여기가 위의 코드가 불리는 곳이다(caller). 그 액션이 로그아웃 프로세스이기 때문에 그 메시지가 서버로 보내지고 나면 유저의 리스트에서 유저의 닉네임을 지우고 해당 유저에게 닉네임을 다시 입력하라고 요청해야 한다. 다음처럼 하자

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @IBAction func exitChat(sender: AnyObject) {  SocketIOManager.sharedInstance.exitChatWithNickname(nickname) { () -> Void in  dispatch\_async(dispatch\_get\_main\_queue(), { () -> Void in  self.nickname = nil  self.users.removeAll()  self.tblUserList.hidden = true  self.askForNickname()  })  }  } |

테스트를 해보면 한 디바이스에서 Exit 버튼을 눌렀을 때 그 유저가 즉시 다른 디바이스의 리스트에서 사라진다는 것을 볼 수 있다. 테스트 하기 전엔 각 디바이스에 우리가 작업한 모든 부분을 다시 설치해서 진행해야 된다는 것을 기억하자(역자 주 : 그냥 다시 재생하면 된다). 이전에 추가된 사용자들이 전부 제거되도록 서버를 재시작 (Ctrl-C 해서 정지, node index.js로 시작) 하는 것도 좋다.

13-7. Chatting

사용자 간의 실제 채팅을 구현할 때다. 예상 하다시피, 서버에 메시지를 보내고 새로운 것을 받아오기 위해 새로운 메소드를 SocketIOManager 클래스에 만들 것이다. 당연히 두 동작을 위해서는 새로운 메시지 리터럴을 사용할 것이고, 이를 서버도 어떻게 답해야 할지 안다. 그리고, 우리는 메시지를 보내고 앱에 띄우는데 부족한 코드를 모두 넣을 것이다.

급한게 먼저니까, SocketIOManager 클래스를 열자. 새로운 메소드를 정의하고 그 body는 한줄만으로 채울 것이다. 그 한줄은 채팅 메시지와 유저의 닉네임을 서버로 보내는 emit 명령이다.

|  |  |
| --- | --- |
| 1  2  3 | func sendMessage(message: String, withNickname nickname: String) {  socket.emit("chatMessage", nickname, message)  } |

서버가 “chatMessage”라는 메시지를 받으면, 이는 결국 연결된 모든 유저에게 표시된다. 이 모든 통신 과정은 한순간에 이루어진다. 그래서 새로운 어떤 메시지든 모든 유저에게 실시간으로 보여질 것이다. 이것이 바로 우리가 SocketIO 라이브러리를 이용해 성취하고자 하는 것이다.

실제 메시지 외에, 서버도 유저에게 다른 쓸모 있는 정보, 즉 메시지 발신자의 닉네임과 메시지의 날짜 및 시간을 보낸다. 그것을 염두에 두고 새롭게 들어오는 채팅 메시지를 listening하는 새로운 메소드를 구현하자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | func getChatMessage(completionHandler: (messageInfo: [String: AnyObject]) -> Void) {  socket.on("newChatMessage") { (dataArray, socketAck) -> Void in  var messageDictionary = [String: AnyObject]()  messageDictionary["nickname"] = dataArray[0] as! String  messageDictionary["message"] = dataArray[1] as! String  messageDictionary["date"] = dataArray[2] as! String    completionHandler(messageInfo: messageDictionary)  }  } |

socket.on(…) 클로져의 dataArray 배열은 세 개의 원소를 가지고 있다: 보낸이의 닉네임, 실제 메시지와 메시지의 시간이다. 이 모두는 string 값으로 보내질 것이다. 그리고 딕셔너리에 추가될 것이다. 이 딕셔너리는 completion handler를 통해 그 메소드의 콜러에게 리턴된다. 지금부터 계속해서 새로운 채팅 메시지가 올 때마다 on(...) 메소드를 자동으로 콜하게 됨을 다시 한 번 기억하자.

채팅 메시지를 보내고 받는 메카니즘이 이제 존재한다. 그러니 이걸 app flow에 전부 합치는 작업을 하자. 이 튜토리얼에서 처음으로 ChatViewController.swift 파일을 열어서 sendMessage(:\_) IBAction method를 찾자. 유저의 닉네임은 세그웨이를 따라 UsersViewController 에서 ChatViewController 로 전달됨을 유념하자. 그래서 우리가 여기서도 유저의 닉네임을 이용할 수 있다.

IBAction 메소드에서 보낼 텍스트가 있다는걸 확인할 것이다. 메시지를 보내고 나면 우리는 textview를 clear시키고 마지막으로는 keyboard를 아래로 숨길 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @IBAction func sendMessage(sender: AnyObject) {  if tvMessageEditor.text.characters.count > 0 {  SocketIOManager.sharedInstance.sendMessage(tvMessageEditor.text!, withNickname: nickname)  tvMessageEditor.text = ""  tvMessageEditor.resignFirstResponder()  }  } |

이제, 우리는 새로운 메시지를 받기 위한 로직을 구현해야 한다. viewDidAppear(\_:) 메소드에 다음을 추가하자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | override func viewDidAppear(animated: Bool) {  super.viewDidAppear(animated)    SocketIOManager.sharedInstance.getChatMessage { (messageInfo) -> Void in  dispatch\_async(dispatch\_get\_main\_queue(), { () -> Void in  self.chatMessages.append(messageInfo)  self.tblChat.reloadData()  // self.scrollToBottom()  })  }  } |

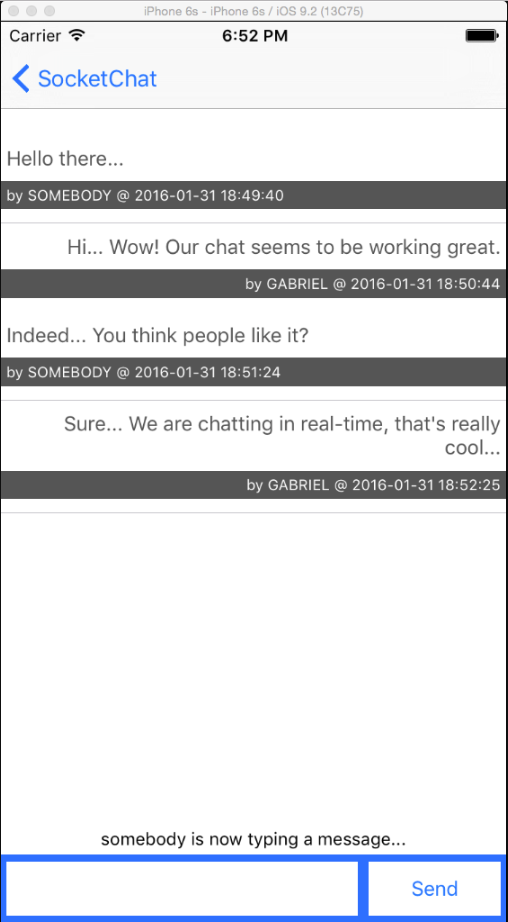
위와 같이, 자세한 새 채팅 메시지는 chatMessages 배열에 append될 것이다. 그리고 tableview는 새 메시지가 chat feed에 표시되게끔 리로드될 것이다.

tableview에 대해 말하자면, 셀 내용이 적절하게 표시되도록 만들자. 카톡처럼 내 메시지가 우리의 다른 유저에게 보내질 때 셀 내용은 오른쪽에 align 될 것이다. 반대로 다른 유저가 메시지를 보내면 셀 내용이 왼쪽으로 align될 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCellWithIdentifier("idCellChat", forIndexPath: indexPath) as! ChatCell    let currentChatMessage = chatMessages[indexPath.row]  let senderNickname = currentChatMessage["nickname"] as! String  let message = currentChatMessage["message"] as! String  let messageDate = currentChatMessage["date"] as! String    if senderNickname == nickname {  cell.lblChatMessage.textAlignment = NSTextAlignment.Right  cell.lblMessageDetails.textAlignment = NSTextAlignment.Right    cell.lblChatMessage.textColor = lblNewsBanner.backgroundColor  }    cell.lblChatMessage.text = message  cell.lblMessageDetails.text = "by \(senderNickname.uppercaseString) @ \(messageDate)"    cell.lblChatMessage.textColor = UIColor.darkGrayColor()    return cell  } |

축하한다! 챗 앱이 이제 거의 다 준비가 됐다. 그러나 더 흥미로운 것들이 많이 남아 있으니까 우리는 여기서 멈추지 않을 것이다.

앱을 다시 테스트하자. 두 장치 모두에 설치해서, Join Chat 버튼을 누르고, 유저들 사이에서 메시지를 보내보자.



**13-8. Being Notified When Users Join and Leave the Chat**

SocketIOManager.swift 파일으로 돌아가서, 두 개의 새로운 메시지(“ userConnectUpdate ”, “ userExitUpdate ”)를 듣기 위한 새로운 메소드를 구현할 것이다. 전자는 새로운 유저의 닉네임이 서버에 전달되고 나서 연결될 때 서버에 의해 보내지는 것이다. 반면 두 번째는 유저가 앱을 종료할 때나 유저가 Exit 버튼을 눌러서 유저리스트에서 완전히 삭제될 때 보내진다.

추가적으로 위 메시지가 들어올 때 우리는 Notification (NSNotifications)을 보낼 것이다. ChatViewController 클래스에 있는 알림들을 보게될 것이고, 최종적으로 적당한 메시지와 함께 팝업 라벨을 보여줄 것이다.

여기 여러분이 SocketIOManager.swift 파일에 만들어야 하는 새로운 메소드가 있다. 우리가 두 메시지를 listen할 메소드이다:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private func listenForOtherMessages() {  socket.on("userConnectUpdate") { (dataArray, socketAck) -> Void in  NSNotificationCenter.defaultCenter().postNotificationName(  "userWasConnectedNotification", object: dataArray[0] as! [String: AnyObject])  }    socket.on("userExitUpdate") { (dataArray, socketAck) -> Void in  NSNotificationCenter.defaultCenter().postNotificationName(  "userWasDisconnectedNotification", object: dataArray[0] as! String)  }  } |

첫 번째 케이스에서 서버는 모든 유저의 정보(ID, 닉네임, 연결상태)를 가지고 있는 딕셔너리를 리턴한다. 두 번째 케이스에서 서버는 채팅방을 나간 사람의 닉네임만 리턴한다. 그러나 두 경우 모두 알림의 object 속성을 이용하여 해당 정보를 보내준다.

위 메소드는 어디선가는 불릴 것이다. 그렇지 않으면 이 앱이 두 메시지를 받아들일 수 없다. 위 메소드를 유저가 서버에 연결되자마자 콜할 최적의 장소는 좀전에 우리가 만들었던 connectToServerwithNickname(\_:completionHandler:) 메소드이다. 이렇게 바꾸자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | func connectToServerWithNickname(nickname: String, completionHandler: (userList: [[String: AnyObject]]!) -> Void) {  ...    listenForOtherMessages()  } |

이제 ChatViewController.swift 를 열어서 viewDidLoad(\_:) 메소드를 보자. 우리가 여기서 해야할 것은 위의 두 알림을 관찰하는 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | override func viewDidLoad() {  ...  NSNotificationCenter.defaultCenter().addObserver(self, selector: "handleConnectedUserUpdateNotification:", name: "userWasConnectedNotification", object: nil)  NSNotificationCenter.defaultCenter().addObserver(self, selector: "handleDisconnectedUserUpdateNotification:", name: "userWasDisconnectedNotification", object: nil)  } |

위 코드는 두 개의 새 메소드를 알림 선택자로 지정한다. 즉, 우리의 다음 단계는 이 둘을 모두 정의하는 것이다.

첫 번째 것을 위해, 연결된 유저의 닉네임을 알림의 object 프로퍼티로부터 추출할 것이다. 그리고 우리는 라벨의 텍스트를 명시할 것이고 트리거를 만들 것이다.

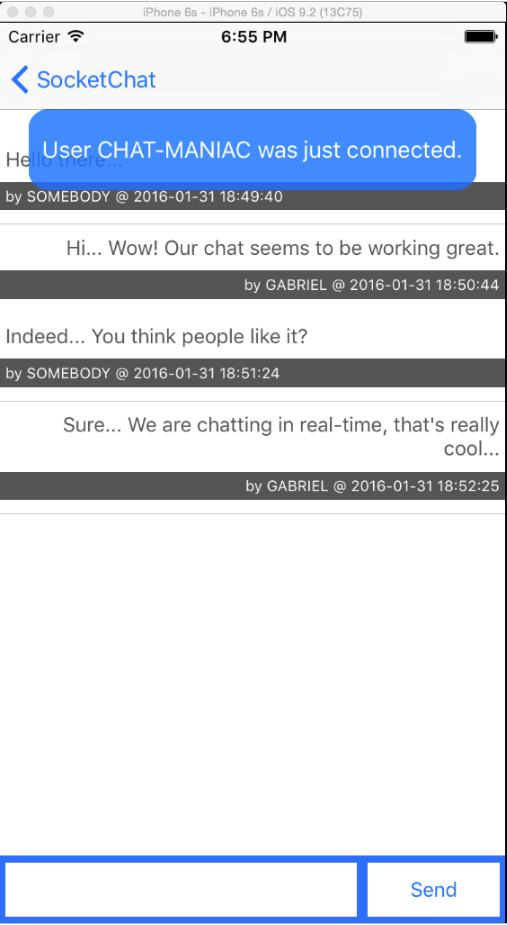
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | func handleConnectedUserUpdateNotification(notification: NSNotification) {  let connectedUserInfo = notification.object as! [String: AnyObject]  let connectedUserNickname = connectedUserInfo["nickname"] as? String  lblNewsBanner.text = "User \(connectedUserNickname!.uppercaseString) was just connected."  showBannerLabelAnimated()  } |

showBannerLabelAnimated 메소드 뿐만 아니라 lblNewsBanner IBOutlet 프라퍼티도 이미 프로젝트에 존재한다.

비슷한 방법으로 우리는 연결이 끊긴 유저의 닉네임을 표시하는 아래의 메소드를 구현할 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5 | func handleDisconnectedUserUpdateNotification(notification: NSNotification) {  let disconnectedUserNickname = notification.object as! String  lblNewsBanner.text = "User \(disconnectedUserNickname.uppercaseString) has left."  showBannerLabelAnimated()  } |

만약 여러분이 앱을 다시 킨다면, chat view controller로 가서 다른 유저를 연결하거나 연결을 끊어보라; 적절한 메시지의 라벨이 몇 초 만에 나타났다가 화면 위로 사라질 것이다.



13-9. Being Notified When Users Type Messages

마지막 흥미로운 주제는 다른 유저가 메시지를 치고 있을 때 알려주는 기능이다. 사실 우리의 목표는 현재 메시지를 치고 있는 유저의 닉네임을 라벨에 보여주는 것이고 아무도 아무것도 치고 있지 않으면 이를 숨기는 기능이다. 이를 가능하게 하기 위해서 우리는 한 유저가 타이핑을 시작하거나 멈출때 서버에 알릴 것이다. 그리고 그 결과로 우리는 타이핑하고 있는 모든 유저의 딕셔너리를 받게 된다.

이전까지 많이 해온 같은 로직을 따라서 SocketIOManager.swift 파일을 열고 메시지가 타입되고 있으면 새로운 메시지를 서버에 emit하는 새로운 메소드를 추가하자.

|  |  |
| --- | --- |
| 1  2  3 | func sendStartTypingMessage(nickname: String) {  socket.emit("startType", nickname)  } |

우리는 또한 유저가 타이핑을 멈출때에도 비슷한 메시지를 보낼 것이다. 예를 들어 swipe down 제스쳐가 만들어지고 키보드가 dismissed된다면 그런 상황이 일어날 수 있다.

|  |  |
| --- | --- |
| 1  2  3 | func sendStopTypingMessage(nickname: String) {  socket.emit("stopType", nickname)  } |

우리가 타이핑을 시작하거나 멈출 때마다 위의 것들 중 하나가 호출됨으로써 서버는 결과적으로 모든 타이핑을 하고 있는 유저들을 알려줄 수 있다. 그러므로 서버는 그 메시지를 보내줄 것이고 따라서 우리는 그것을 듣고 있어야 한다. 아래 코드가 listenForOtherMessages() 메소드에 추가될 것임을 주의하자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | private func listenForOtherMessages() {  ...      socket.on("userTypingUpdate") { (dataArray, socketAck) -> Void in  NSNotificationCenter.defaultCenter().postNotificationName("userTypingNotification", object: dataArray[0] as? [String: AnyObject])  }  } |

우리는 이제 아까 작업했었떤 ChatViewController.swift 파일로 전환할 준비가 되었다. viewDidLoad(\_:)메소드에서 위 알림을 계속 관찰할 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | override func viewDidLoad() {  ...    NSNotificationCenter.defaultCenter().addObserver(self, selector: "handleUserTypingNotification:", name: "userTypingNotification", object: nil)  } |

handleUserTypingNotification(\_:) 메소드를 구현할 때 타이핑중인 유저가 나라면 라벨을 보여주지 않도록 하자. 나머지는 쉽다. 서버에서 반환된 딕셔너리에 포함된 모든 사용자의 닉네임을 포함하는 문자열을 작성하고 출력 메시지를 올바르게 준비한다.

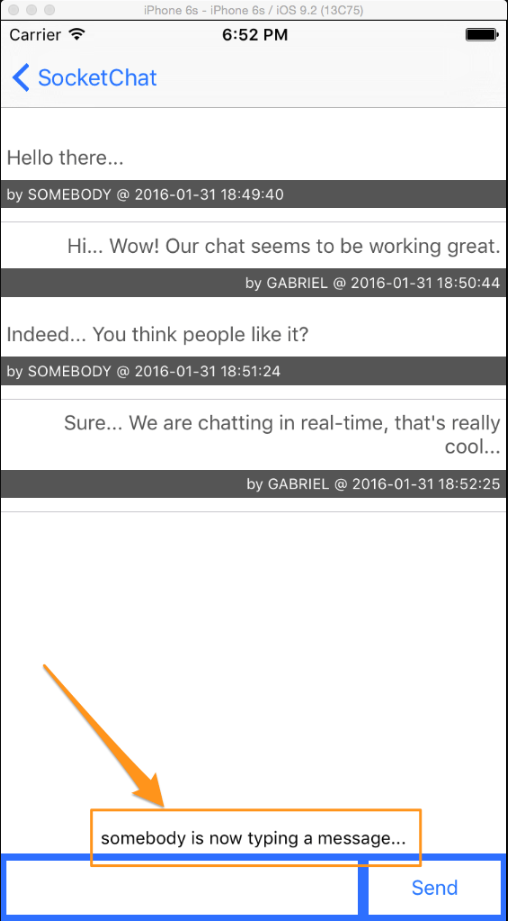
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | func handleUserTypingNotification(notification: NSNotification) {  if let typingUsersDictionary = notification.object as? [String: AnyObject] {  var names = ""  var totalTypingUsers = 0  for (typingUser, \_) in typingUsersDictionary {  if typingUser != nickname {  names = (names == "") ? typingUser : "\(names), \(typingUser)"  totalTypingUsers += 1  }  }    if totalTypingUsers > 0 {  let verb = (totalTypingUsers == 1) ? "is" : "are"    lblOtherUserActivityStatus.text = "\(names) \(verb) now typing a message..."  lblOtherUserActivityStatus.hidden = false  }  else {  lblOtherUserActivityStatus.hidden = true  }  }    } |

잘했다. 하지만 아직 유저가 타이핑 하고 있는지 아닌지를 서버에 알리지 않았다. 첫 번째 경우에는 textview의 textViewShouldBeginEditing(\_:)로 가서 delegate 메소드로 이동해서 다음처럼 업데이트 하자.

|  |  |
| --- | --- |
| 1  2  3  4  5 | func textViewShouldBeginEditing(textView: UITextView) -> Bool {  SocketIOManager.sharedInstance.sendStartTypingMessage(nickname)    return true  } |

유저가 키보드를 없애서 타이핑을 멈췄다는 것을 나타내려면, dismissKeyboard()라는 메소드를 찾아내서 라인을 한 줄 추가하자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | func dismissKeyboard() {  if tvMessageEditor.isFirstResponder() {  tvMessageEditor.resignFirstResponder()    SocketIOManager.sharedInstance.sendStopTypingMessage(nickname)  }  } |



바로 그거다! 위는 우리의 마지막 산고였다. 남은 일은 앱을 한 번 더 테스트하고 다른 유저가 타이핑할 때 알림을 받는 것이다.