# Chapter 29: WebSockets

By Logan Wright

WebSockets与HTTP一样，定义了用于两个设备之间通信的协议。与HTTP不同，WebSocket协议专为实时通信而设计。对于需要实时行为的聊天或其他功能，WebSockets可以是一个很好的选择。 Vapor提供了一个简洁的API来创建WebSocket服务器或客户端。本章重点介绍如何构建基本服务器。

在本章中，您将构建一个简单的客户端 - 服务器应用程序，允许用户与其他人共享其当前位置，然后他们可以实时在地图上查看此位置。

## Tools

测试WebSockets可能有点棘手，因为您无法访问浏览器中的URL或使用简单的CURL请求。要解决这个问题，您将使用一个名为Simple WebSocket Client的Google Chrome扩展程序。它可以免费安装，来自 https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo.

安装该工具后，在Chrome中打开它。

## A basic server

现在您的工具已准备就绪，是时候设置一个非常基本的WebSocket服务器了。将本章的入门项目复制到您喜欢的位置，然后在该目录中打开一个终端窗口。

输入以下内容以构建和打开Xcode项目：

```
cd location-track-server
vapor xcode -y
```

## Echo server

打开**websockets.swift**并将以下内容添加到sockets(_:)的末尾以创建echo端点：

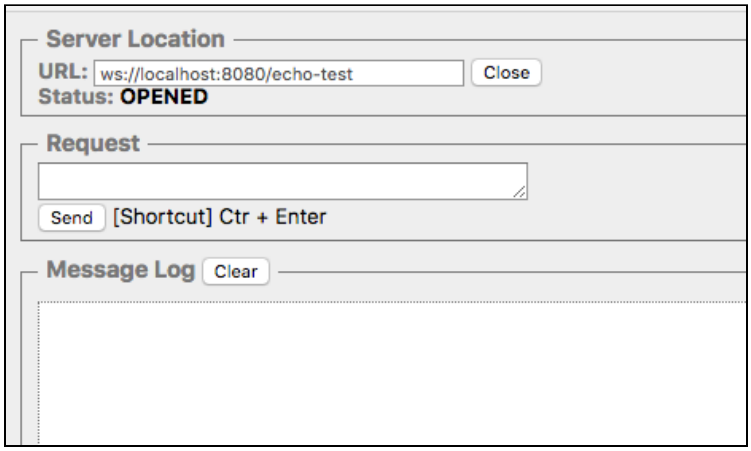
```
// 1
websockets.get("echo-test") { ws, req in
  print("ws connnected")

  // 2
  ws.onText { ws, text in
    print("ws received: \(text)")
    ws.send("echo - \(text)")
  }
}
```

这是它的作用：

1. 为**echo-test**端点创建路由处理程序。它会在每次连接时将消息记录到控制台。

2. 创建一个每次端点接收文本时触发的侦听器。它将收到的文本记录到控制台，然后在文本前添加**echo -** 之后将其回送给发送方。

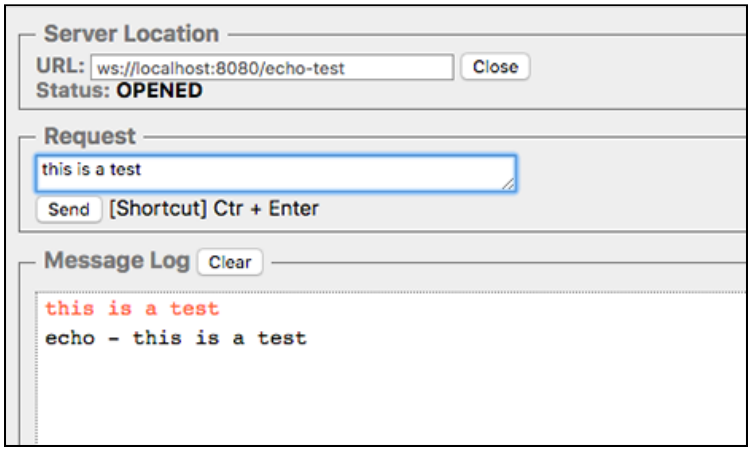
在Xcode的方案选择器中，选择**Run** scheme和**My Mac**作为目标。构建并运行。在Chrome中，打开Simple WebSocket Client并在URL字段中输入**ws://localhost:8080/echo-test**。单击“**Open**”，“**Status**”将更改为“**OPENED**”。



检查Xcode控制台，你会看到**ws connected**。

```
Server starting on http://localhost:8080
ws connnected
```

在Simple WebSocket Client中输入一条消息，您将看到您的服务器以适当的回应进行响应。




# iOS project

本章的资料包括一个几乎完整的iOS应用程序。稍后您将添加跟踪用户的功能。该应用程序包括由Josh Baker编写的WebSocket客户端实现。您可以在 https://github.com/tidwall/SwiftWebSocket 上找到更多信息及其原始源代码。

构建并运行您的服务器项目；让它继续运行。现在在模拟器中构建并运行iOS项目。点击主屏幕上的**Echo**按钮几次。您应该在Xcode控制台中看到类似于以下内容的输出。

```
sending sending echo 1521066998.62272
got message: echo - sending echo 1521066998.62272
sending sending echo 1521066999.33329
got message: echo - sending echo 1521066999.33329
sending sending echo 1521066999.90972
got message: echo - sending echo 1521066999.90972
```

真棒！您的服务器正在通过WebSocket与iOS应用程序通信哦！

> 注意：如果您尝试在真机设备上运行iOS应用程序，则需要在**WebServices.swift**中更改host的定义。

# Server word API

现在您已经验证了您的客户端和服务器可以进行通信，是时候向服务器添加更多功能了。服务器starter项目包括一个随机单词生成器，您将用它来创建跟踪session ID。

要演示此生成器，请打开**routes.swift**并将以下内容添加到routes(_:)末尾：

```
router.get("word-test") { request in
  return wordKey(with: request)
}
```

这为**word-test**端点定义了一个GET处理程序，它仅仅返回对wordKey(with:)的调用结果。

构建并运行。在浏览器中访问**http://localhost:8080/word-test**。您将看到类似于以下内容的结果：

```
exercise.green.power
```

现在您已经构建了服务器的基本结构，是时候添加位置共享端点了。

# Session管理

您的服务器应用支持两种类型的客户端用户：

- **Poster**: 共享位置供他人查看的客户端。
- **Observer**: 观察并绘制**Poster**位置的客户端。

**Posters**和**Observers**通过**TrackingSession**连接，使用您之前看到的随机生成的单词进行标识。

出于本教程的目的，您将创建一个TrackingSessionManager来协调所有这些内容。它将创建跟踪sessions，从Posters接收更新，并将这些更新通知Observers。

> 注意：您将创建的解决方案不可扩展，仅适用于单个服务器实例。为了使其更具可扩展性，您需要将TrackingSessionManager连接到大型实时数据库，例如Redis。

## 创建session

当Poster创建新的跟踪session时，您必须分配一个新ID并将其返回给用户。 starter项目包括一个线程安全的LockedDictionary实现，使存储session信息变得简单。

打开**SessionManager.swift**并在TrackingSessionManager中添加以下内容：

```
private(set) var sessions:
  LockedDictionary<TrackingSession, [WebSocket]> = [:]
```

每个TrackingSession都与一组WebSockets相关联，每个WebSocket都对应一个Observer。

Poster需要一种方法来创建跟踪session。将以下内容添加到TrackingSessionManager的末尾：

```
func createTrackingSession(for request: Request)
  -> Future<TrackingSession> {
  // 1
  return wordKey(with: request)
    .flatMap(to: TrackingSession.self) { [unowned self] key in
      // 2
      let session = TrackingSession(id: key)

      // 3
      guard self.sessions[session] == nil else {
        return self.createTrackingSession(for: request)
      }

      // 4
      self.sessions[session] = []

      // 5
      return Future.map(on: request) { session }
    }
}
```

这是它的作用：

1. 生成新的session ID。 wordKey(with:)返回Future <String>，因此必须将其解包以在后续步骤中使用。
2. 使用刚创建的ID为此新session创建TrackingSession。
3. 确保session ID是唯一的。如果不是，就递归调用自己再试一次。
4. 记录新的TrackingSession并为其提供一个空的Observers列表。
5. 将session包装成future并返回。

## 更新位置

starter项目包括一个遵循Content协议的Location模型。利用这一点并添加一些魔法，便可以轻松地将位置以JSON发送。关闭你的Xcode项目。在终端中，输入以下内容：

```
touch Sources/App/WebSocket+Extensions.swift
vapor xcode -y
```

这会将文件添加到项目结构中的正确位置，并生成更新的Xcode项目。在**WebSocket+Extensions.swift**中添加以下实现：

```
import Vapor
import WebSocket
import Foundation

extension WebSocket {
  func send(_ location: Location) {
    let encoder = JSONEncoder()
    guard let data = try? encoder.encode(location) else {
      return
    }

    send(data)
  }
}
```

此方法只是将Location模型转换为JSON以通过线路传输。

打开**SessionManager.swift**并将以下内容添加到类的末尾：

```
func update(_ location: Location,
            for session: TrackingSession) {
  guard let listeners = sessions[session] else {
    return
  }

  listeners.forEach { ws in
    ws.send(location)
  }
}
```

当Poster向服务器发送更新的位置时，会将该新位置发送给每个已注册的Observer。

## 关闭session

您已经构建了允许Poster创建和更新跟踪session的逻辑。Poster需要的最后功能是关闭session。

将以下内容添加到TrackingSessionManager类的末尾：

```
func close(_ session: TrackingSession) {
  guard let listeners = sessions[session] else {
    return
  }

  listeners.forEach { ws in
    ws.close()
  }

  sessions[session] = nil
}
```

这将关闭每个Observer的WebSocket，并从活动sessions列表中删除TrackingSession。

完成Poster所需的所有行为后，就可以实现Observer交互了。

## Observer行为

Tracking Session Manager必须为Observers提供两种交互：

- 为更新而注册。
- 断开与服务器的连接。

打开**SessionManager.swift**并将以下内容添加到TrackingSessionManager的末尾：

```
func add(listener: WebSocket, to session: TrackingSession) {
  // 1
  guard var listeners = sessions[session] else {
    return
  }

  listeners.append(listener)
  sessions[session] = listeners

  // 2
  listener.onClose.always { [weak self, weak listener] in
    guard let listener = listener else {
      return
    }

    self?.remove(listener: listener, from: session)
  }
}

func remove(listener: WebSocket,
            from session: TrackingSession) {
  // 3
```

```
    guard var listeners = sessions[session] else {
      return
    }

    listeners = listeners.filter { $0 !== listener }
    sessions[session] = listeners
  }
```

这是它的作用：

1. 验证session是否存在，并将Observer的WebSocket添加到监听列表中。

2. 注册一个onClose处理程序，该处理程序在Observer客户端关闭WebSocket时触发。这个处理程序会从监听列表中删除WebSocket。

3. 验证session是否存在，并从监听列表中删除Observer的WebSocket。

# 端点

既然TrackingSessionManager现在已完成，那么您必须创建一些端点，使客户端可以访问它的行为。支持Poster的端点都可以作为常规HTTP路由实现。它不需要使用WebSockets，因为它不需要实时更新。

## Create

打开**routes.swift**并将以下内容添加到routes(_:)末尾：

```
  router.post("create", use: sessionManager.createTrackingSession)
```

对/**create**的空POST请求将创建一个新的跟踪session，并将其返回给客户端。

构建并运行。通过在终端中输入以下内容来测试session创建：

```
  curl -X POST http://localhost:8080/create
```

服务器将返回一个如下所示的JSON对象：

```
  { "id": "pumped.arch.dime" }
```

## Close

接下来，是时候实现“close”支持了。为此，您要在**/close/:tracking-session-id**处创建一个端点。将以下内容添加到routes(_:)末尾：

```
router.post(
  "close",
  TrackingSession.parameter) { req -> HTTPStatus in
    let session = try req.parameters.next(TrackingSession.self)
    sessionManager.close(session)
    return .ok
}
```

此代码接收TrackingSession作为参数，用session管理器关闭session，然后返回空的HTTPResponse以表示成功。

构建并运行。像以前一样创建session。使用返回的跟踪session ID发送关闭请求，如下所示：

```
curl -w "%{response_code}\n" -X POST \
  http://localhost:8080/close/<tracking.session.id.goes.here>
```

您将在下一行看到显示**200**，显示服务器发送了**200 OK** HTTP状态。

## Update

最后，Poster需要一个端点来接收位置更新。您要在**/update/:tracking-session-id**处创建端点以实现此目的。将以下内容添加到routes(_:)末尾：

```
// 1
router.post(
  "update",
  TrackingSession.parameter) { req -> Future<HTTPStatus> in
    // 2
    let session = try req.parameters.next(TrackingSession.self)
    // 3
    return try Location.decode(from: req)
      .map(to: HTTPStatus.self) { location in
        // 4
        sessionManager.update(location, for: session)
        return .ok
  }
}
```

这是它的作用：

1. 为端点创建POST处理程序。

2. 从URL中提取跟踪session ID。

3. 从POST请求的正文中创建一个Location。

4. 调用session管理器以广播更新的位置，然后返回**200 OK** HTTP状态。

构建并运行。像之前一样创建session。使用返回的跟踪session ID发送更新请求，如下所示：

```
curl -w "%{response_code}\n" \
  -d '{"latitude": 37.331, "longitude": -122.031}' \
  -H "Content-Type: application/json" -X POST \
  http://localhost:8080/update/<tracking.session.id.goes.here>
```

就这样了！你已经实现了你的Posters所需要的一切！

# Observer端点

Observer只需要一个端点，用于连接WebSocket。为此，您必须定义新的WebSocket路由。打开**websockets.swift**并在sockets(_:)末尾添加以下内容：

```
// 1
websockets.get("listen", TrackingSession.parameter) { ws, req in
  // 2
  let session = try req.parameters.next(TrackingSession.self)
  // 3
  guard sessionManager.sessions[session] != nil else {
    ws.close()
    return
  }
  // 4
  sessionManager.add(listener: ws, to: session)
}
```

这是它的作用：

1. 为端点创建WebSocket处理程序 **/listen/:tracking-session-id**。

2. 从URL中提取跟踪session ID。

3. 确保session仍然有效。如果无效了，请关闭WebSocket。

4. 将WebSocket作为Observer添加到session中。

就这样！您的服务器已完成，准备运行新位置共享应用程序。构建并运行。让服务器在一个窗口中运行，然后在另一个窗口中打开iOS应用程序的项目。

# iOS follow location

如前所述，starter项目iOS应用程序即将完成。剩下的就是让你实现它的WebSocket能力。当用户希望观察Poster时，应用会提示输入跟踪session ID。然后调用startSocket()注册为Observer并处理位置更新。

打开**FollowViewController.swift**并用以下内容替换现有的startSocket()：

```
func startSocket() {
  // 1
  let ws = WebSocket("ws://\(host)/listen/\(session.id)")

  // 2
  ws.event.close = { [weak self] code, reason, clean in
    self?.navigationController?
      .popToRootViewController(animated: true)
  }

  // 3
  ws.event.message = { [weak self] message in
    guard let bytes = message as? [UInt8] else {
      fatalError("invalid data")
    }
    let data = Data(bytes: bytes)
    let decoder = JSONDecoder()
    do {
      // 4
      let location = try decoder.decode(
        Location.self,
        from: data
      )
      // 5
      self?.focusMapView(location: location)
    } catch {
      print("decoding error: \(error)")
    }
  }
}
```

这是它的作用：

1. 使用用户输入的跟踪session ID打开服务器的WebSocket。
2. 设置在WebSocket关闭时调用的事件处理程序。
3. 设置在WebSocket接收数据时调用的事件处理程序。
4. 将收到的消息解码为Location。
5. 在地图上标出收到的位置。

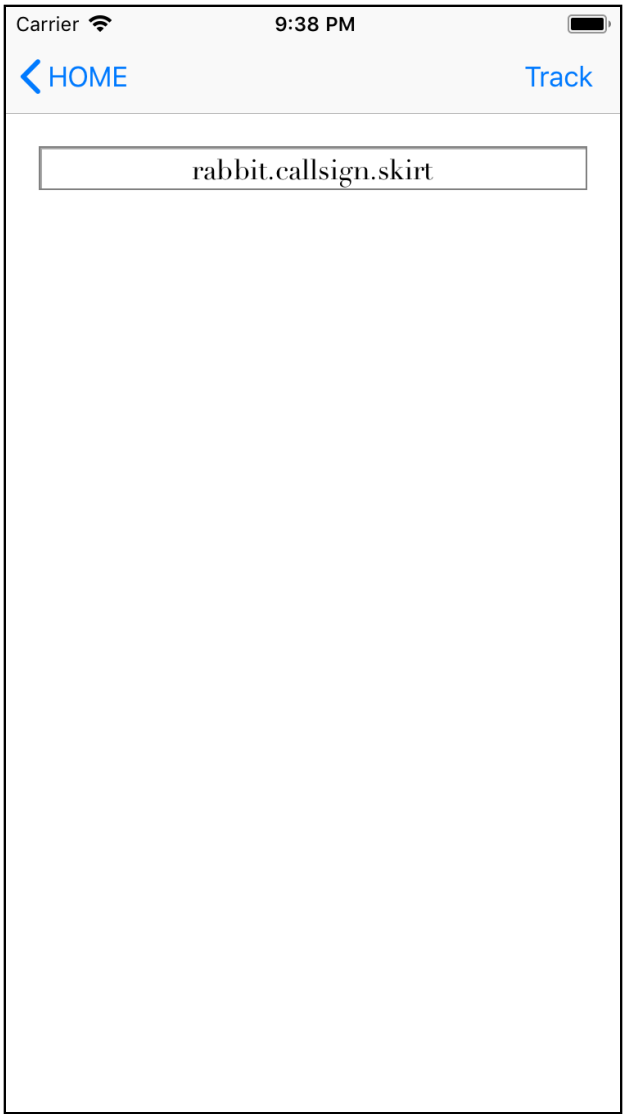
构建并运行。点击“**Echo**”按钮以验证应用和服务器是否正常通信。您将在终端中使用curl命令来模拟Poster。在终端中输入以下内容以创建新session：

```
curl -X POST http://localhost:8080/create
```

正如您所期望的那样，您将收到包含跟踪session ID的JSON响应。它看起来像这样：

```
{ "id": "rabbit.callsign.skirt" }
```

在iOS模拟器中，点按“**FOLLOW**”并输入跟踪session ID，然后点按“**Track**”。

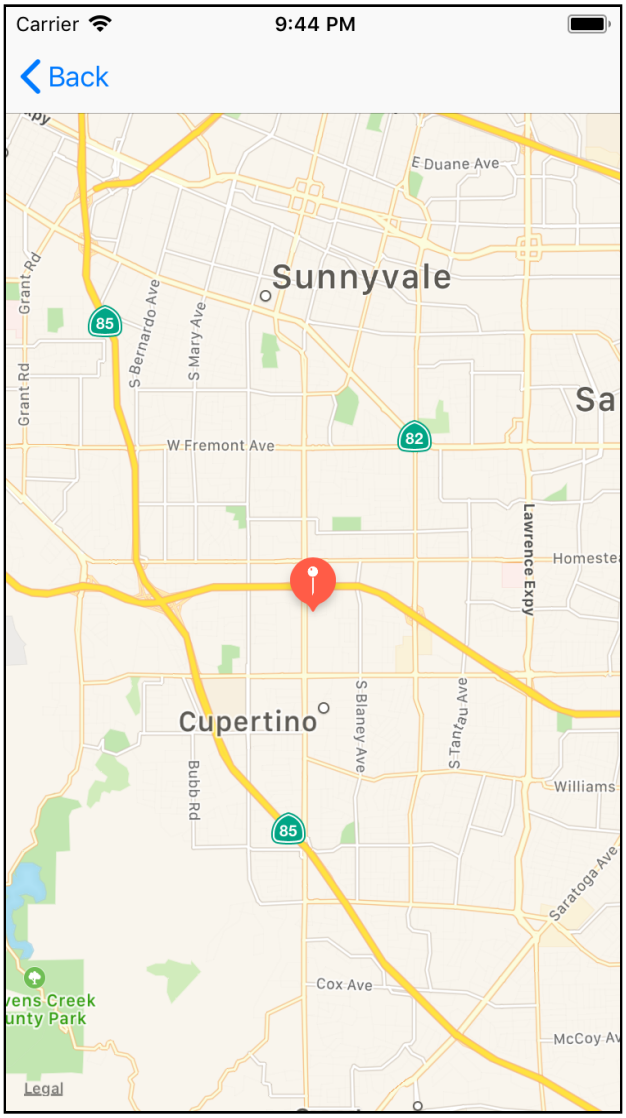



您现在需要发送位置更新。

在终端中，输入以下内容，根据需要插入跟踪session ID：

```
curl -w "%{response_code}\n" \
  -d '{"latitude": 37.331, "longitude": -122.031}' \
  -H "Content-Type: application/json" -X POST \
  http://localhost:8080/update/<tracking.session.id.goes.here>
```

模拟器上的应用程序将立即跳转到新收到的位置，恰好是Apple的总部。




稍微更改经纬度数字，验证地图是否随位置变化而更新。

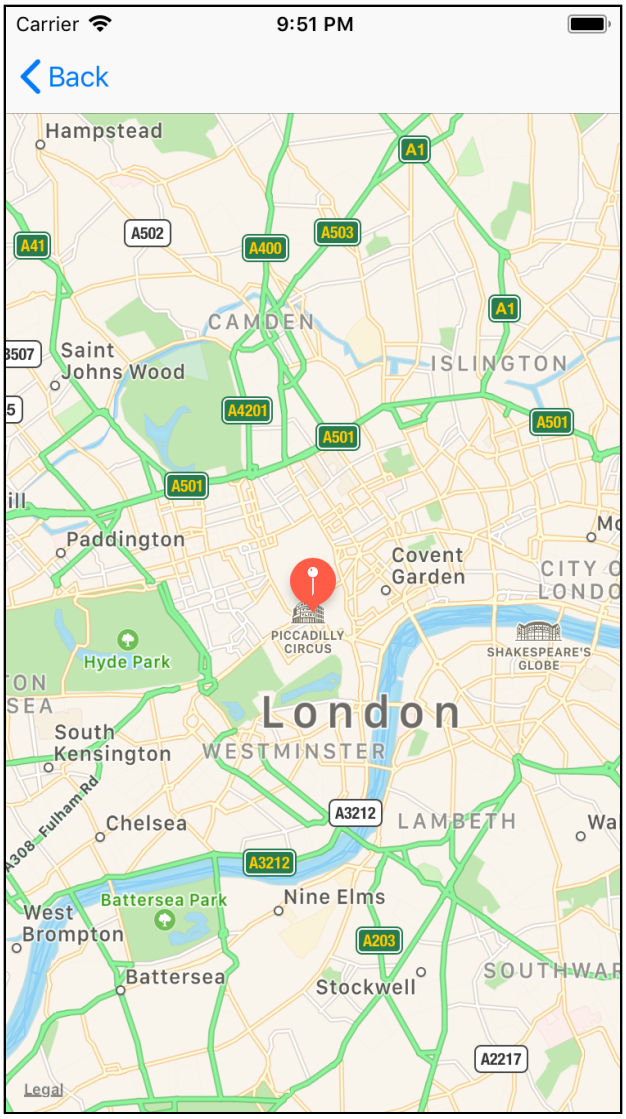
在终端中输入以下内容并观看地图更新：

```
curl -w "%{response_code}\n" \
  -d '{"latitude": 37.332, "longitude": -122.030}' \
  -H "Content-Type: application/json" -X POST \
  http://localhost:8080/update/<tracking.session.id.goes.here>
```

做最后一个测试。在终端中输入以下内容：

```
curl -w "%{response_code}\n" \
  -d '{"latitude": 51.510, "longitude": -0.134}' \
  -H "Content-Type: application/json" -X POST \
  http://localhost:8080/update/<tracking.session.id.goes.here>
```

地图将跳转到伦敦的皮卡迪利广场！



# 然后去哪儿?

你做到了。您的iOS应用程序通过WebSockets与您的Swift服务器实时通信了。许多不同类型的应用程序都可以从WebSockets实现的即时通信中受益，包括聊天应用程序，游戏，实时股票行情等等。如果您想象中的应用程序需要实时响应，WebSockets可能是您的答案！

## 挑战

有关WebSockets的更多练习，请尝试以下挑战：

- 向应用程序添加更多数据以对其进行个性化处理。也许Observer包含姓名或其他识别信息，以便Poster知道谁在观察。
- 为Poster提供Observers的实时列表。
- 尝试在远程服务器上托管您的基本应用程序。确保更新iOS应用程序中的host变量，看看是否可以使用几个iPhone运行它。您和朋友可以四处走动并测试您的位置更新。