Documentation

Hummingbird Document… / WebSocket Server Upgrade

Article

# WebSocket Server Upgrade

Support for upgrading HTTP1 connections to WebSocket.

## Overview

Before a HTTP connection can be upgraded to a WebSocket connection a server must process an initial upgrade request and respond with a switching protocols response. HummingbirdWebSocket provides a server child channel setup that implements this for you with entry points to decide whether the upgrade should occur and then how to handle the upgraded WebSocket connection.

## Setup

You can access this by setting the `server` parameter in `Application.init()` to `http1WebSocketUpgrade(configuration:additionalChannelHandlers:shouldUpgrade:)`. This is initialized with a closure that returns either `ShouldUpgradeResult.dontUpgrade` to not perform the WebSocket upgrade or `ShouldUpgradeResult.upgrade(_:_:)` along with the closure handling the WebSocket connection.

```
let app = Application(
    router: router,
    server: .http1WebSocketUpgrade { request, channel, logger in
        // upgrade if request URI is "/ws"
        guard request.uri == "/ws" else { return .dontUpgrade }
        // The upgrade response includes the headers to include in the respo
        // the WebSocket handler
        return .upgrade([:]) { inbound, outbound, context in
```

```
            for try await frame in inbound {
                // send "Received" for every frame we receive
                try await outbound.write(.text("Received"))
            }
        }
    }
)
```

Alternatively you can provide a `Router` using a `RequestContext` that conforms to `WebSocketRequestContext`. The router can be the same router as you use for your HTTP requests, but it is preferable to use a separate router. Using a router means you can add middleware to process the initial upgrade request before it is handled eg for authenticating the request.

```
// Setup WebSocket router
let wsRouter = Router(context: BasicWebSocketRequestContext.self)
// add middleware
wsRouter.middlewares.add(LogRequestsMiddleware())
wsRouter.middlewares.add(BasicAuthenticator())
// An upgrade only occurs if a WebSocket path is matched
wsRouter.ws("/ws") { request, context in
    // allow upgrade
    .upgrade([:])
} onUpgrade: { inbound, outbound, context in
    for try await frame in inbound {
        // send "Received" for every frame we receive
        try await outbound.write(.text("Received"))
    }
}
let app = Application(
    router: router,
    server: .http1WebSocketUpgrade(webSocketRouter: wsRouter)
)
```

## WebSocket Handler

The WebSocket handle function has three parameters: an inbound sequence of WebSocket frames ( `WebSocketInboundStream`), an outbound WebSocket frame writer (`WebSocketOutboundWriter`) and a context parameter. The WebSocket is kept open as long as you don't leave this function. PING, PONG and CLOSE frames are managed internally. As soon as you leave this function it will perform the CLOSE handshake. If you want to send a regular PING keep-alive you can control that via the WebSocket configuration. By default servers send a PING every 30 seconds.

Below is a simple input and response style connection a frame is read from the inbound stream, processed and then a response is written back. If the connection is closed the inbound stream will end and we exit the function.

```
wsRouter.ws("/ws") { inbound, outbound, context in
    for try await frame in inbound {
        let response = await process(frame)
        try await outbound.write(response)
    }
}
```

If the reading and writing from your WebSocket connection are asynchronous then you can use a structured `TaskGroup`.

```
wsRouter.ws("/ws") { inbound, outbound, context in
    try await withThrowingTaskGroup(of: Void.self) { group in
        group.addTask {
            for try await frame in inbound {
                await process(frame)
            }
        }
        group.addTask {
            for await frame in outboundFrameSource {
                try await outbound.write(frame)
            }
        }
        try await group.next()
        // once one task has finished, cancel the other
        group.cancelAll()
    }
}
```

You should not use unstructured Tasks to manage your WebSockets. If you use an unstructured Task you increase the likelyhood of processing a WebSocket connection that has already been closed.

## Frames and messages

A WebSocket message can be split across multiple WebSocket frames. The last frame indicated by the `FIN` flag being set to true. If you want to work with messages instead of frames you can convert the inbound stream of frames to a stream of messages using `messages(maxSize:)`.

```
wsRouter.ws("/ws") { inbound, outbound, context in
    // We have set the maximum size of a message to be 1MB. If we don't set
    // a maximum size a client could keep sending us frames until we ran
    // out of memory.
    for try await message in inbound.messages(maxSize: 1024*1024) {
        let response = await process(message)
        try await outbound.write(response)
    }
}
```

## WebSocket Context

The context that is passed to the WebSocket handler along with the inbound stream and outbound writer is different depending on how you setup your WebSocket connection. In most cases the context only holds a `Logger` for logging output.

But if the WebSocket was setup with a router, then the context also includes the `Request` that initiated the WebSocket upgrade and the `RequestContext` from that same call. With this you can configure your WebSocket connection based on details from the initial request. Below we are using a query parameter to add a named WebSocket to a connection manager

```
wsRouter.ws("chat") { request, _ in
    // only allow upgrade if username query parameter exists
    guard request.uri.queryParameters["username"] != nil else {
        return .dontUpgrade
    }
    return .upgrade([:])
} onUpgrade: { inbound, outbound, context in
    // only allow upgrade to continue if username query parameter exists
    guard let name = context.request.uri.queryParameters["username"] else {
    await connectionManager.manageUser(name: String(name), inbound: inbound,
}
```

Alternatively you could use the `RequestContext` to extract authentication data to get the user's name.

## See Also

### Related Documentation

`class WebSocketInboundStream`

Inbound WebSocket data frame AsyncSequence

`struct WebSocketOutboundWriter`

Outbound websocket writer

### WebSockets

WebSocket Client

Connecting to WebSocket servers.