Using WebSockets

HTML5 and WebSocket

The WebSocket protocol was standardized in 2011 with the original goal of allowing browsers to create stable and bidirectional connections with a server. Before that, browsers used to only support HTTP requests, which aren't well-suited for bidirectional communication.

The protocol is message-based and a very powerful tool to send push notifications to browsers. It has been used to implement chats, turn-based games, and more. It still uses a TCP connection, which is good for reliability but not for latency, so it's not good for real-time applications like VoIP and fast-paced games (see WebRTC for those use cases).

Due to its simplicity, its wide compatibility, and being easier to use than a raw TCP connection, WebSocket started to spread outside the browsers, in native applications as a mean to communicate with network servers.

Godot supports WebSocket in both native and web exports.

Using WebSocket in Godot

WebSocket is implemented in Godot via WebSocketPeer . The WebSocket implementation is compatible with the High-Level Multiplayer. See section on high-level multiplayer for more details.

Warning

When exporting to Android, make sure to enable the `INTERNET` permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Minimal client example

en stable

This example will show you how to create a WebSocket connection to a remote server, and how to send and receive data.

```
extends Node

# The URL we will connect to.
# Use "ws://localhost:9080" if testing with the minimal server example
below.
# `wss://` is used for secure connections,
# while `ws://` is used for plain text (insecure) connections.
@export var websocket_url = "wss://echo.websocket.org"

# Our WebSocketClient instance.
var socket = WebSocketPeer.new()


func _ready():
    # Initiate connection to the given URL.
    var err = socket.connect_to_url(websocket_url)
    if err == OK:
        print("Connecting to %s..." % websocket_url)
        # Wait for the socket to connect.
        await get_tree().create_timer(2).timeout

        # Send data.
        print("> Sending test packet.")
        socket.send_text("Test packet")
    else:
        push_error("Unable to connect.")
        set_process(false)


func _process(_delta):
    # Call this in `_process()` or `_physics_process()`.
    # Data transfer and state updates will only happen when calling this
function.
    socket.poll()

    # get_ready_state() tells you what state the socket is in.
    var state = socket.get_ready_state()

    # `WebSocketPeer.STATE_OPEN` means the socket is connected and ready
    # to send and receive data.
    if state == WebSocketPeer.STATE_OPEN:
        while socket.get_available_packet_count():
            var packet = socket.get_packet()
            if socket.was_string_packet():
                var packet_text = packet.get_string_fro
                print("< Got text data from server: %s" % packet_text)
            else:
                print("< Got binary data from server: %d bytes" %
packet.size())

    # `WebSocketPeer.STATE_CLOSING` means the socket is closing.
    # It is important to keep polling for a clean close.
    elif state == WebSocketPeer.STATE_CLOSING:
        pass

    # `WebSocketPeer.STATE_CLOSED` means the connection has fully closed.
    # It is now safe to stop polling.
    elif state == WebSocketPeer.STATE_CLOSED:
        # The code will be `-1` if the disconnection was not properly
notified by the remote peer.
        var code = socket.get_close_code()
        print("WebSocket closed with code: %d. Clean: %s" % [code, code !=
-1])
        set_process(false) # Stop processing.
```

This will print something similar to:

```
Connecting to wss://echo.websocket.org...
< Got text data from server: Request served by 7811941c69e658
> Sending test packet.
< Got text data from server: Test packet
```

Minimal server example

This example will show you how to create a WebSocket server that listens for remote connections, and how to send and receive data.

```
extends Node

# The port we will listen to.
const PORT = 9080

# Our TCP Server instance.
var _tcp_server = TCPServer.new()

# Our connected peers list.
var _peers: Dictionary[int, WebSocketPeer] = {}

var last_peer_id := 1


func _ready():
    # Start listening on the given port.
    var err = _tcp_server.listen(PORT)
    if err == OK:
        print("Server started.")
    else:
        push_error("Unable to start server.")
        set_process(false)


func _process(_delta):
    while _tcp_server.is_connection_available():
        last_peer_id += 1
        print("+ Peer %d connected." % last_peer_id)
        var ws = WebSocketPeer.new()
        ws.accept_stream(_tcp_server.take_connection())
        _peers[last_peer_id] = ws

    # Iterate over all connected peers using "keys()" so we can erase in
the loop
    for peer_id in _peers.keys():
        var peer = _peers[peer_id]

        peer.poll()

        var peer_state = peer.get_ready_state()
        if peer_state == WebSocketPeer.STATE_OPEN:
            while peer.get_available_packet_count():
                var packet = peer.get_packet()
                if peer.was_string_packet():
                    var packet_text = packet.get_string
                    print("< Got text data from peer %d: %s ... echoing" %
[peer_id, packet_text])
```

en stable

```
                    # Echo the packet back.
                    peer.send_text(packet_text)
                else:
                    print("< Got binary data from peer %d: %d ... echoing"
% [peer_id, packet.size()])
                    # Echo the packet back.
                    peer.send(packet)
        elif peer_state == WebSocketPeer.STATE_CLOSED:
            # Remove the disconnected peer.
            _peers.erase(peer_id)
            var code = peer.get_close_code()
            var reason = peer.get_close_reason()
            print("- Peer %s closed with code: %d, reason %s. Clean: %s" %
[peer_id, code, reason, code != -1])
```

When a client connects, this will print something similar to this:

```
Server started.
+ Peer 2 connected.
< Got text data from peer 2: Test packet ... echoing
```

Advanced chat demo

A more advanced chat demo which optionally uses the multiplayer mid-level abstraction and a high-level multiplayer demo are available in the godot demo projects under *networking/websocket_chat* and *networking/websocket_multiplayer*.

User-contributed notes

Please read the User-contributed notes policy before submitting a comment.

Loading comments...

en stable