3

Having Two-Way Conversations

In this chapter, we will cover the following recipes:

* Creating A Simple Chat Room
* Managing the Socket Lifecycle
* Emitting A Private Message To Another Socket
* Sending Messages To All Sockets Except the Sender
* Building A Multiplayer Tic-Tac-Toe Game

# Introduction

While we can do some interesting things with one-way communication, the real power of Socket.IO begins to show through when the client and server are both participants in a constant dialogue.

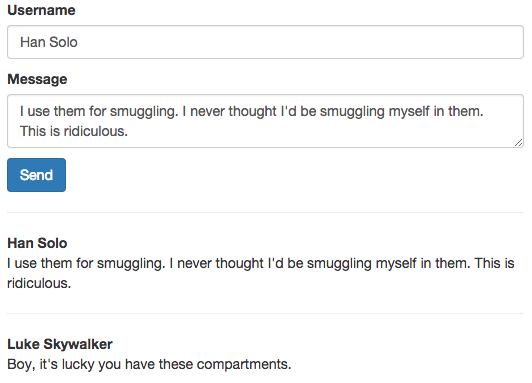
In this chapter, we will work through various examples of two-way Socket.IO communication. We will implement various tactics to make bidirectional communication work for our specific needs.

Creating a Simple Chat Room

A basic chat room application is one of the most widely used demos showing off Socket.IO or even web sockets in a more general sense. The reason for this is that it immediately gives the business case for these technologies in a way that is easy to follow and digest.

Building a basic chat room using Socket.IO is not terribly difficult or complex. This is the sort of application that Socket.IO was designed for.

When we have completed our chat application, it will look something like this:



**Insert Image B04893\_03\_01.png**

## Getting Ready...

For this recipe, we will be using jQuery for some simple DOM manipulation and Bootstrap for styling. None of these libraries are required to make a chat app with Socket.IO, but they all provide useful utilities that we can easily tap into.

## How To Do It...

To create a simple chat application with Socket.IO, follow these steps:

1. Create a **server.js** file. This file will start up your server and emit Socket.IO events whenever a new message is posted to the chat application. In addition to the typical Socket.IO server setup, we will need to add this:

io.on('connection', function (socket) {

socket.on('message.send', function (data) {

io.emit('message.sent', data);

});

});

1. Create your **index.html** template. This will include a form at the top of the page to post new messages. It also contains a container div to hold our chat messages.

<div class="container">

<form id="message-form">

<p>

<label>Username</label>

<input class="form-control" id="username" />

</p>

<p>

<label>Message</label>

<textarea class="form-control" id="message"></textarea>

</p>

<button class="btnbtn-primary" type="submit">Send</button>

</form>

<div id="messages"></div>

</div>

1. Add the client-side JavaScript. It will submit messages to the server and render messages when they are emitted from the server:

// Update the users count

socket.on('message.sent', function (data) {

$('#messages').prepend(`

<div>

<hr />

<div><strong>${data.username}</strong></div>

<p>${data.message}</p>

</div>

`);

});

$(function () {

$('#message-form').on('submit', function (e) {

e.preventDefault();

socket.emit('message.send', {

message: $('#message').val(),

username: $('#username').val()

});

});

});

## How It Works...

The server-side code is acting as a hub for incoming messages. When new messages come in, it emits them to all of the connected sockets.

We are submitting the messages from our form on the client-side. We are also rendering the new messages when they are emitted from the server. In this way, the client who is emitting the message will listen for the same message.sent event as all of the other clients.

## There’s More...

The messages in this simple example will not be persisted. That means that when the client first loads the page, there will not be any messages in the interface. They will only get messages when new messages are posted after the page is loaded.

To show all of the messages that occurred before the page load, we would need to maintain them somehow and emit them when the socket connection event is fired on the server.

We could hold an array of posted messages in memory, for example, and emit them when the page is loaded. The downside to the in-memory approach is that when the server is restarted, all of the messages that we previously in memory wouldbe lost.

A better approach would be to keep the messages in a database and fetch the previously posted messages when the connection is made.

# Managing the Socket Lifecycle

If our server is maintaining a list of our connected sockets, it should always be mindful of when a socket disconnects.

A socket can disconnect for any number of reasons. The user may navigate away from the webpage that the WebSocket connection is on. The user’s Internet may go down.

When these things happen, we can tap into the disconnect event to notify the client-side that the socket is no longer available.

## How To Do It...

To remove references to disconnected sockets, follow these steps:

1. Listen for the socket disconnect event on the server-side. When it occurs,we will emit an event back to the clients with the ID of the socket that was disconnected. Socket.IO associates every socket with a unique ID, which we can use to manage our sockets.

io.on('connection', function (socket) {

io.emit('user.add', socket.id);

socket.on('disconnect', function () {

io.emit('user.remove', socket.id)

});

});

1. Add an element to your view that contains the list of users:

<div id=”users”></div>

1. Finally, in the client, we will listen to theuser.add and user.removeevents to add or remove users as they are connected or disconnected:

socket.on('user.add', function (id) {

$('#users').prepend(`<p id="${id}">${id}</p>`);

});

socket.on('user.remove', function (id) {

$(`#${id}`).remove();

});

## How It Works...

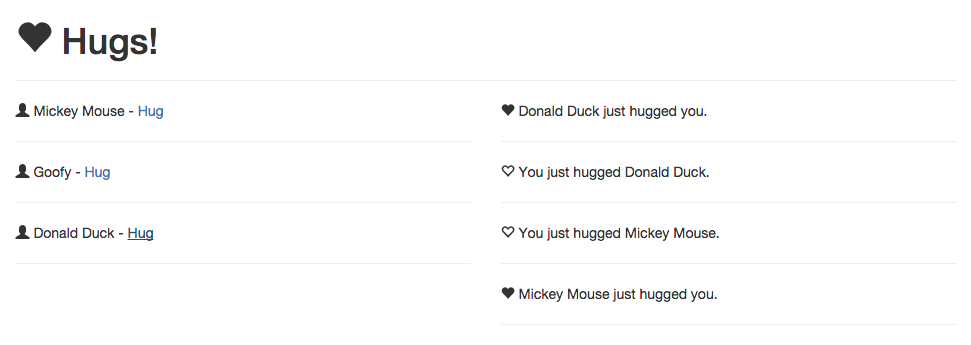
While the ID of a socket is primarily available for internal use, when we are managing a list of connected users, it can be beneficial to have a record of the socket IDs to associate with the rendered list in our interface.

In our case, we are using the socket ID as the actual id attribute for our DOM elements. The ID will look like a random assortment of numbers and letters such as “AL8r1DvmiQVT50trAAAC”. Using the socket ID in tandem with the socket lifecycle events, we are able to show the list of currently active users on the page.

# Emitting A Private Message To Another Socket

Sometimes, you need to send a private message to just one other socket and not every socket that might be listening in. Since the server-side is in charge of managing all of the connected sockets, we can very granularly specify which sockets our events are emitted to.

In this recipe, we will use that ability to create a simple app where the user can specify another user to give a hug to. No other users but the giver and the receiver will be aware that the hug was initiated. Our application will look something like this:



**Insert Image B04893\_03\_02.png**

## Getting Ready...

For this recipe, we will be using jQuery for DOM manipulation and Bootstrap for styling.

## How To Do It...

To send private messages with Socket.IO, follow these steps:

1. Add the relevant events to your server. These will be in charge of managing a list of connected users and emitting private messages to users as needed. In addition to the typical Socket.IO server-side setup, you will need this code:

// We will keep a record of all connected sockets

var sockets = {};

io.on('connection', function (socket) {

// Emit the connected users when a new socket connects

for (vari in sockets) {

socket.emit('user.add', {

username: sockets[i].username,

id: sockets[i].id

});

}

// Add a new user

socket.on('username.create', function (data) {

socket.username = data;

sockets[socket.id] = socket;

io.emit('user.add', {

username: socket.username,

id: socket.id

});

});

// Send the hug event to only the socket specified

socket.on('user.hug', function (id) {

sockets[id].emit('user.hugged', socket.username);

});

// Remove disconnected users

socket.on('disconnect', function () {

delete sockets[socket.id];

io.emit('user.remove', socket.id);

});

});

1. Create an **index.html** template to display the interface for your application:

<div class="container">

<h1><span class="glyphiconglyphicon-heart"></span> Hugs!</h1>

<hr />

<form id="add-username" class="row">

<div class="col-md-3">

<input class="form-control" id="username" placeholder="Username" />

</div>

<div class="col-md-3">

<button class="btnbtn-primary">Join</button>

</div>

</form>

<div class="row">

<div class="col-md-6" id="sockets" style="display: none"></div>

<div class="col-md-6" id="hugs"></div>

</div>

</div>

1. Add listeners to the client-side to display the users and keep a log of which private messages have been emitted:

function addUser (user) {

$('#sockets').append(`

<div id="${user.id}" class="socket">

<span class="glyphicon-user"></span>

${user.username} -

<a href="#" class="hug" data-username="${user.username}" data-id="${user.id}">Hug</a>

<hr />

</div>

`);

}

function addUsername (e) {

e.preventDefault();

socket.emit('username.create', $('#username').val());

$('#sockets').show();

$(this).hide();

}

function giveHug (e) {

var id = $(this).data('id'),

username = $(this).data('username');

e.preventDefault();

socket.emit('user.hug', id);

$('#hugs').prepend(`

<p>

<span class="glyphiconglyphicon-heart-empty"></span>

You just hugged ${username}.

<hr />

</p>

`);

}

socket.on('users.list', function (list) {

list.forEach(addUser);

});

socket.on('user.hugged', function (username) {

$('#hugs').prepend(`

<p>

<span class="glyphiconglyphicon-heart"></span>

${username} just hugged you.

<hr />

</p>

`);

});

socket.on('user.remove', function (id) {

$('#' + id).remove();

});

socket.on('user.add', addUser);

$(function () {

$('#sockets').delegate('.hug', 'click', giveHug);

$('#add-username').on('submit', addUsername);

});

## How It Works...

By maintaining a list of available sockets in our server-side object, we are able to look up any socket by its ID. When we get the socket that we want to send a private message to, we can emit an event to only that socket and no others.

# Sending Messages To All Sockets Except the Sender

When a socket sends a message, we don’t necessarily want it to receive the message that it sent. We might want to display a different message to the sender than to the receivers. This can be accomplished by using the socket.broadcast.emit()syntax.

## How To Do It...

To send messages to every socket except for the sender, follow these steps:

1. Create a template for sockets to announce their presence. This will also include a “messages” container where we will render incoming messages:

<div class="container">

<hr />

<form id="form" class="row">

<div class="col-md-10">

<input class="form-control" id="name" placeholder="Who are you?" />

</div>

<div class="col-md-2">

<button class="btn-primary form-control" type="submit">Send</button>

</div>

</form>

<div id="messages"></div>

</div>

1. Add some client-side JavaScript to render messages:

socket.on('user.joined', function (data) {

$('#messages').prepend(`

<p>

<hr />

${data.name} is finally here.

</p>

`);

});

$(function () {

$('#form').on('submit', function (e) {

e.preventDefault();

var name = $('#name').val();

socket.emit('user.join', {

name: name

});

$('#messages').prepend(`<p>

<hr />

Hi ${name}!

</p>`);

});

});

1. Add a server-side event that broadcasts theuser.joinedevent off of the socket:

io.on('connection', function (socket) {

socket.on('user.join', function (data) {

socket.broadcast.emit('user.joined', data);

});

});

## How It Works...

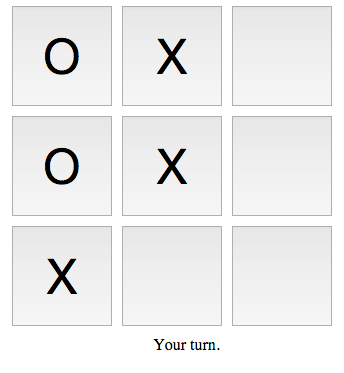
The socket.broadcast.emit() method sends the message to every socket except for the socket that the method is called from. By using this method, we can exclude the initiating socket from receiving the event.

# Building A Multiplayer Tic-Tac-Toe Game

We can observer the real power of Socket.IO by using it in a multiplayer game. While there are many examples of multiplayer games we could implement, Tic-Tac-Toe is one of the more simple games.

The game board is always three tiles long and three tiles tall. The goal is for either user to select any three tiles in a row.

Our final user interface will look something like this:



**Insert Image B04893\_03\_03.png**

Our game will allow any number of players to join and be paired with another player. The paired players will be able to interact on the board together in real-time.

## Getting Ready...

In this recipe, we will be using jQuery for DOM manipulation and event delegation.

## How To Do It...

To create a real-time multiplayer Tic-Tac-Toe game, follow these instructions:

1. Create a **server.js** file to handle the server-side portion of our game. Make sure that you create an instantiated Socket.IO connection on your server before you add this code.

var players = {},

unmatched;

function joinGame (socket) {

// Add the player to our object of players

players[socket.id] = {

// The opponent will either be the socket that is

// currently unmatched, or it will be null if no

// players are unmatched

opponent: unmatched,

// The symbol will become 'O' if the player is unmatched

symbol: 'X',

// The socket that is associated with this player

socket: socket

};

// Every other player is marked as 'unmatched', which means

// there is not another player to pair them with yet. As soon

// as the next socket joins, the unmatched player is paired with

// the new socket and the unmatched variable is set back to null

if (unmatched) {

players[socket.id].symbol = 'O';

players[unmatched].opponent = socket.id;

unmatched = null;

} else {

unmatched = socket.id;

}

}

// Returns the opponent socket

function getOpponent (socket) {

if (!players[socket.id].opponent) {

return;

}

return players[

players[socket.id].opponent

].socket;

}

io.on('connection', function (socket) {

joinGame(socket);

// Once the socket has an opponent, we can begin the game

if (getOpponent(socket)) {

socket.emit('game.begin', {

symbol: players[socket.id].symbol

});

getOpponent(socket).emit('game.begin', {

symbol: players[getOpponent(socket).id].symbol

});

}

// Listens for a move to be made and emits an event to both

// players after the move is completed

socket.on('make.move', function (data) {

if (!getOpponent(socket)) {

return;

}

socket.emit('move.made', data);

getOpponent(socket).emit('move.made', data);

});

// Emit an event to the opponent when the player leaves

socket.on('disconnect', function () {

if (getOpponent(socket)) {

getOpponent(socket).emit('opponent.left');

}

});

});

1. Create an **index.html** template for the view of our game.

<div class="board">

<button id="a0"></button>

<button id="a1"></button>

<button id="a2"></button>

<button id="b0"></button>

<button id="b1"></button>

<button id="b2"></button>

<button id="c0"></button>

<button id="c1"></button>

<button id="c2"></button>

<div id="messages">Waiting for opponent to join...</div>

</div>

1. Write the client-side JavaScript to handle the gameplay.

VarmyTurn = true, symbol;

function getBoardState () {

varobj = {};

// We will compose an object of all of the Xs and Ox

// that are on the board

$('.board button').each(function () {

obj[$(this).attr('id')] = $(this).text() || '';

});

return obj;

}

function isGameOver () {

var state = getBoardState(),

// One of the rows must be equal to either of these

// value for

// the game to be over

matches = ['XXX', 'OOO'],

// These are all of the possible combinations

// that would win the game

rows = [

state.a0 + state.a1 + state.a2,

state.b0 + state.b1 + state.b2,

state.c0 + state.c1 + state.c2,

state.a0 + state.b1 + state.c2,

state.a2 + state.b1 + state.c0,

state.a0 + state.b0 + state.c0,

state.a1 + state.b1 + state.c1,

state.a2 + state.b2 + state.c2

];

// Loop over all of the rows and check if any of them compare

// to either 'XXX' or 'OOO'

for (vari = 0; i<rows.length; i++) {

if (rows[i] === matches[0] || rows[i] === matches[1]) {

return true;

}

}

}

function renderTurnMessage () {

// Disable the board if it is the opponents turn

if (!myTurn) {

$('#messages').text('Your opponent\'s turn');

$('.board button').attr('disabled', true);

// Enable the board if it is your turn

} else {

$('#messages').text('Your turn.');

$('.board button').removeAttr('disabled');

}

}

function makeMove (e) {

e.preventDefault();

// It's not your turn

if (!myTurn) {

return;

}

// The space is already checked

if ($(this).text().length) {

return;

}

// Emit the move to the server

socket.emit('make.move', {

symbol: symbol,

position: $(this).attr('id')

});

}

// Event is called when either player makes a move

socket.on('move.made', function (data) {

// Render the move

$('#' + data.position).text(data.symbol);

// If the symbol is the same as the player's symbol,

// we can assume it is their turn

myTurn = (data.symbol !== symbol);

// If the game is still going, show who's turn it is

if (!isGameOver()) {

renderTurnMessage();

// If the game is over

} else {

// Show the message for the loser

if (myTurn) {

$('#messages').text('Game over. You lost.');

// Show the message for the winner

} else {

$('#messages').text('Game over. You won!');

}

// Disable the board

$('.board button').attr('disabled', true);

}

});

// Set up the initial state when the game begins

socket.on('game.begin', function (data) {

// The server will asign X or O to the player

symbol = data.symbol;

// Give X the first turn

myTurn = (symbol === 'X');

renderTurnMessage();

});

// Disable the board if the opponent leaves

socket.on('opponent.left', function () {

$('#messages').text('Your opponent left the game.');

$('.board button').attr('disabled', true);

});

$(function () {

$('.board button').attr('disabled', true);

$('.board> button').on('click', makeMove);

});

1. Add some CSS to make the board look nice.

body {

text-align: center;

}

.board {

margin: auto;

width: 350px;

}

.board button {

height: 100px;

width: 100px;

margin: 0px;

padding: 0px;

float: left;

margin-right: 10px;

margin-bottom: 10px;

font-size: 3em

}

## How It Works...

Our server is responsible for pairing up sockets as opponents and emitting events to the paired sockets whenever a move is made or the state of the game changes in any way. Most of the actual game logic is happening on the client-side where we keep track of which squares are selected, who selected them and who’s turn it currently is.

It is important to manage the state of multiplayer games on the server so that all of the players will be using the same state at all times. When the first player makes a move, the state on the server should update and then broadcast the change to all of the sockets that are involved in the game.