4

Building a Room With a View

In this chapter, we will cover the following recipes:

* Creating Chat Channels With Namespaces
* Joining Rooms
* Leaving Rooms
* Listing Rooms the Socket Is In
* Making Private Rooms
* Setting Up a Default Room
* Handling Messages from Outside the Room

# Introduction

Socket.IO uses namespaces to keep separate types of messages from colliding with each other. Using namespaces, we can be sure that our applications are listening for the correct events.

We can also define arbitrary rooms that our sockets can join or leave. These rooms restrict who receives messages to only the interested parties.

In this chapter, we will also learn how to harness namespaces and rooms to create richer real-time experiences.

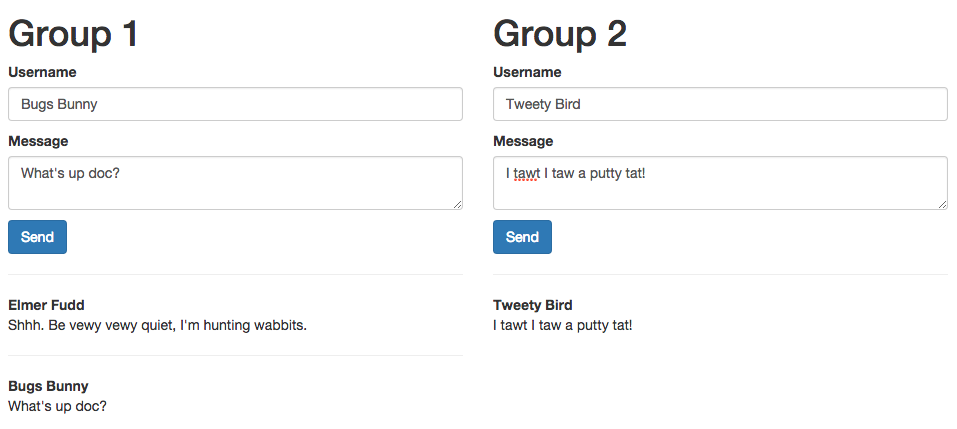
# Creating Chat Channels With Namespaces

Namespaces are a great way to make sure our Socket.IO events are not emitted globally to all of the sockets connected to the server.We can send messages to a namespace and only the sockets listening to that namespace will receive the event.

Many applications have multiple customers that should never be mixed together. In our URLs, we typically show use different domains to keep our customers separate so that customer1.website.com gets a different result than customer2.website.com. In the same way, our Socket.IO sockets can be namespaced to minimize concerns about intermingling data and messaging.

In this recipe, we are going to set up two separate groups of chat channels. We can post to either group and the message will be restricted to the namespace for that group.

Our interface will be a split page with two separate groups. We can post to either group and the messages will show up below that group.



Insert Image B04893\_04\_01.png

## Getting Ready...

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

## How To Do It...

To create chat channels using namespaces, follow these instructions:

1. First, add your server-side code. We will create a loop to call a function that will set up our two different namespaces:

function createNamespace (i) {

var group = io.of('/group-' + i);

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

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

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

});

});

}

for (var i = 0; i< 2; i++) {

createNamespace(i);

}

1. Now, create your client-side template. Note that most of the template is actually inside a script tag. The template inside the script tag will not be executed when the page loads. We will grab it with JavaScript and do some manipulation before it is rendered to the DOM:

<div class="container">

<div class="row"></div>

</div>

<script type="text/tpl" id="namespace-group-tpl">

<div class="col-md-6">

<h1>Group ${i}</h1>

<form class="message-form">

<p>

<label>Username</label>

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

</p>

<p>

<label>Message</label>

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

</p>

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

</form>

<div class="messages"></div>

</div>

</script>

1. Finally, we will create our client-side JavaScript. Our script will call the createNamespace() function twice in a loop to create two unique namespaces and render the interface to display those namespaces:

functioncreateNamespace (i, template) {

var socket = io(`http://localhost:5000/group-${i}`),

$el = $(template.replace(/\${i}/g, (i + 1)));

$('.row').append($el);

$el.find('.message-form').bind('submit', function (e) {

e.preventDefault();

socket.emit('message.send', {

message: $el.find('.input-message').val(),

username: $el.find('.input-username').val()

});

});

// Update the users count

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

$el.find('.messages').prepend(`

<div>

<hr />

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

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

</div>

`);

});

}

$(function () {

var template = $('#namespace-group-tpl').text();

for (var i= 0; i< 2; i++) {

createNamespace(i, template);

}

});

## How It Works...

On the server-side of our application, the io.of() method was used to create a namespace. It took a string with the name of the namespace as the first argument. The namespace name was important because we also used it on the client-side.

On the client-side, we just needed to add the namespace to the end of our first argument on the io() method. In our example, we instantiated the namespace using the port number like this:io.of(‘http://localhost:5000/my-namespace’). However, if we were listening on port 80 instead of port 5000, we could actually just pass in the namespace name and not worry about providing the port: io.of(‘/my-namespace’)

# Joining Rooms

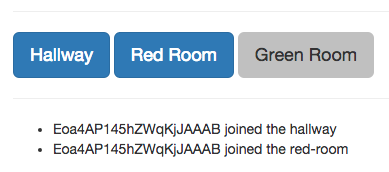
In addition to namespaces, we can also use rooms in Socket.IO to ensure that our messages are only being delivered to the correct sockets.

While every socket can only have a single namespace, every socket can belong to multiple rooms. You can think of rooms as channels that a socket subscribes to in order to get specific types of messages.

For example, if we built a programming application, the user might be interested in JavaScript and Node messages, but not Ruby or C#. Using rooms, we can allow users to send messages to specific channels so only the interested parties will receive it.

Since rooms can only be joined if we know the name of the room, rooms create a sort of sudo-security, but it is a bit of a hack because if someone can guess the room name and it is exposed to the client to join, they can join any arbitrary room.

In this recipe, we will demonstrate how a user can join a room.



Insert Image B04893\_04\_02.png

## Getting Ready...

We will use jQuery for some simple DOM manipulation in this recipe.

## How To Do It...

To enable joining of rooms in Socket.IO, follow these steps:

1. Add the server-side room implementation. This includes allowing a socket to join a room using the socket.join() method and then emitting a new messageto ensure that the socket correctly joined:

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

socket.emit('room.joined', socket.id + ' joined the hallway');

socket.on('room.join', function (room) {

socket.join(room);

io.to(room).emit('room.joined', socket.id + ' joined the ' + room);

});

});

1. Add the client-side template:

<button data-id="hallway" class="btn-primary">Hallway</button>

<button data-id="red-room">Red Room</button>

<button data-id="green-room">Green Room</button>

<ul id="messages"></ul>

1. Add the client-side JavaScript:

var socket = io('http://localhost:5000');

socket.on('room.joined', function (message) {

$('#messages').append(`<li>${message}</li>`);

});

$('button').on('click', function () {

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

if (!$(this).hasClass('btn-primary')) {

$(this).addClass('btn-primary');

socket.emit('room.join', id);

}

});

## How It Works...

Socket.IO can emit messages to any arbitrary room name by using the io.on(:room\_name).emit() method. It should be noted that you may emit to a room name whether or not the room already exists beforehand, so there is no need to do any checking to make sure the room is available.

When we click on a button in our interface to join one of our rooms, the socket will emit a message to the server asking to join that room. The server-side will be responsible for adding the socket to the requested group. We will also emit a message to the newly joined room announcing that the user has indeed joined it.

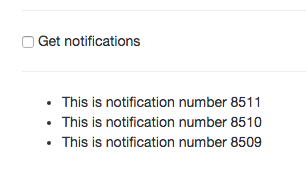
# Leaving Rooms

Not only can a socket join a room, it can also leave any room that it is a member of.

This important if you’re building a real-time application where users might want to disable certain notifications. By leaving a room entirely, the client-side socket will never get the events that are broadcasted to the room that it has left.

In this recipe, we will expose a single room to the client-side sockets. We will allow the room to be joined or left by simply toggling a checkbox.

The server-side will emit a message every two seconds with an ongoing count of how many times the notification has been sent. We will be able to turn the notifications on and see notification numbers logged one after another in sequence. Then we can toggle it off for a few seconds and finally turn it back on and see the notification number resume after skipping the numbers it would have emitted while it was off.



Insert Image B04893\_04\_03.png

## Getting Ready...

In this recipe, we will use a little jQuery for DOM manipulation.

## How To Do It...

To enable a socket to leave a room, follow these steps:

1. In our server-side code, we will create an event to join the “notifications” room. We will also create an event allowing the socket to leave the room.

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

socket.on('notifications.join', function () {

socket.join('notifications');

});

socket.on('notifications.leave', function () {

socket.leave('notifications');

});

});

// We will emit a message to every member   
// of our room every two seconds with a notification number.  
  
Var i= 0;

setInterval(function () {

io.to('notifications').emit('notify', 'This is notification number ' + i);

i++;

}, 2000);

1. On the client-side, we will create a template with a checkbox that allows us to toggle our membership of the “notifications” room on and off. We’ll also include a container unordered list to put our messages into when they are emitted to us.

<div class="container">

<hr />

<input type="checkbox" id="toggle" /> Get notifications

<hr />

<ul id="messages">

<!-- Messages will go here -->

</ul>

</div>

1. Finally, we will add some client-side JavaScript to toggle our group membership on and off and add messages as we receive them.

socket.on('notify', function (message) {

$('#messages').prepend(`<li>${message}</li>`);

});

$(function () {

$('#toggle').on('click', function () {

var checked = $(this).is(':checked'),

action = (checked) ? 'join' : 'leave';

socket.emit(`notifications.${action}`);

});

});

## How It Works...

We are emitting a message to our “notifications” room on an interval every two seconds. The portion of our code that is emitting the notification doesn’t care whether any sockets are subscribed to the room it is emitting to or not. It just blindly emits messages to the room.

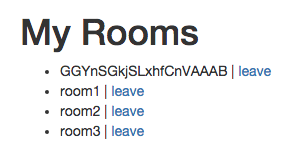
We are allowing our sockets to join and leave the room by listening for an event from the client.

When our client-side socket is a member of our room, it will receive notifications in the room, but when it is not a member of the room, no notifications will be received.

# Listing Rooms the Socket Is In

Socket.IO provides a dynamic list of the rooms that each socket is a member of. We can retrieve that list and use it as needed.

In this recipe, we will create a list of the rooms that our socket is in. The list will be dynamically updated if the socket leaves any of its rooms.



Insert Image B04893\_04\_04.png

## Getting Ready...

As usual, this recipe will use jQuery for DOM manipulation and event delegation.

## How To Do It...

To list the rooms that your socket is in, follow these steps:

1. On the server, we will add our socket to three distinct groups by default. We will add an event listener to request for us to list the rooms. When that event is sent, we will emit a response message containing the socket.rooms, which as an array that Socket.IO builds dynamically as you join and leave rooms.

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

socket.join('room1');

socket.join('room2');

socket.join('room3');

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

socket.emit('list.rooms.response', socket.rooms);

});

socket.on('leave.room', function (room) {

socket.leave(room);

});

});

1. We will need a simple template to display our room names in. We can create that on the client-side:

<div class="container">

<h1>My Rooms</h1>

<ul id="messages"></ul>

</div>

1. When the server emits a list.rooms.response event, we will need to display it in our template.

socket.on('list.rooms.response', function (rooms) {

$('#messages').html('');

rooms.forEach(function (room) {

$('#messages').append(`<li>${room} | <a href="#" data-id="${room}">leave</a></li>`);

});

});

1. Last of all, we will emit the initial event to list our rooms and add anonClick function to leave any of our rooms.

$(function () {

socket.emit('list.rooms');

$('#messages').delegate('a', 'click', function () {

socket.emit('leave.room', $(this).data('id'));  
  
 // Since we are already listing the rooms initially with  
 // this message, we can do reuse it to kick off the  
 // listing response message.

socket.emit('list.rooms');

});

});

## How It Works...

The socket.rooms variable is an array containing strings representing the rooms that our socket is a member of. Whenever we need to send the list of rooms to the client, we can just emit the socket.rooms and the client will have a fresh list.

## There’s More...

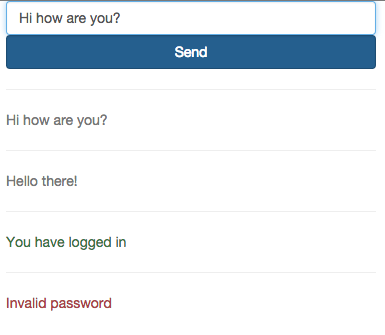
You may have noticed that there is a room with a random-looking name. It will look something like “GGYnSGkjSLxhfCnVAAAB”. This is the default room that the socket is associated with. Each socket gets its own room when it is connected. We will explore this in more details in the “Setting Up a Default Room” recipe later in this chapter.

# Making Private Rooms

It can often be useful to provide privacy for certain rooms. This allows us to send messages to a small group of sockets without worrying about the messages being received by sockets that should not be allowed to see them.

While Socket.IO doesn’t have any built-in way to consider a room “private” or “public”, we can add some logic around joining a room so that only sockets that validate against a password check are allowed to be members of our room.

In this recipe, we will create a simple login page. Sockets can log in with the static password “pass123”, but we could easily make it use a dynamic password that comes from our database or an environmental variable. When the socket joins the group, it will be able to see all of the messages that are emitted to that group as expected.



Insert Image B04893\_04\_05.png

## Getting Ready...

For this recipe, we will use jQuery for DOM manipulation.

We will also make use of the MD5 Node module to hash our password. While it may seem silly to hash a password that is hard-coded like this, typically our password would not be included in the code at all. As a best practice, it should either be in a database or an environmental variable. We can install the MD5 package by entering npm install md5 –save in the terminal.

## How To Do It...

To create a private room, follow these steps:

1. Add your server-side code. This will do some validation before the socket joins a room to ensure that they have entered the correct password.

// Include the md5 module

var md5 = require('MD5');

// This is the hashed password to join the private group

// It is the md5 hash of "pass123"

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

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

// Return and emit a message if the passwords don't match

if (md5(data.password) !== privatePassword) {

returnsocket.emit('message.posted', {

type: 'danger',

message: 'Invalid password'

});

}

// Join the group

socket.join('secret group');

socket.emit('join.group.success');

});

// Post a message to the secret group

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

io.to('secret group').emit('message.posted', {

type: 'muted',

message: data.message

});

});

});

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

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

// Return and emit a message if the passwords don't match

if (md5(data.password) !== privatePassword) {

return socket.emit('message.posted', {

type: 'danger',

message: 'Invalid password'

});

}

// Join the group

socket.join('secret group');

socket.emit('join.group.success');

});

// Post a message to the secret group

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

io.to('secret group').emit('message.posted', {

type: 'muted',

message: data.message

});

});

1. Create the client-side template. This will include a form to log in as well as a form to post messages. The messages form will only be displayed after the socket joins the room:

<div class="container">

<!-- Login Form -->

<form id="login">

<div class="row">

<div class="col-md-9">

<input class="form-control" placeholder="Password" type="password" />

</div>

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

<button class="btnbtn-primary form-control">Login</button>

</div>

</div>

</form>

<!-- Message -->

<form id="message" style="display: none">

<div class="row">

<div class="col-md-9">

<input class="form-control" placeholder="Message" />

</div>

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

<button class="btnbtn-primary form-control">Send</button>

</div>

</div>

</form>

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

</div>

1. Now we will add our client-side logic. This is mainly just listening for events that are triggered when the login form is submitted or a message is sent.

// Render messages from the server

function renderMessage (data) {

$('#messages').prepend(`<div class="text-${data.type}">

<hr />

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

</div>`);

}

socket.on('message.posted', renderMessage);

// Toggle the messages and login form when the socket logs in

socket.on('join.group.success', function () {

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

$('#login').hide();

renderMessage({

type: 'success',

message: 'You have logged in'

});

});

$(function () {

// Attempt to log in

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

e.preventDefault();

var password = $(this).find('input').val();

socket.emit('join.group', {

password: password

});

});

// Send a message

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

e.preventDefault();

var message = $(this).find('input').val();

socket.emit('message.post', {

message: message

});

});

});

## How It Works...

By doing a return before the socket joins a room, we are able to keep the code underneath the return from executing in cases where the password the user has entered doesn’t match the password we are expecting.

Since we are hashing our password, we will need to hash the value that the socket sends us so that we can compare the two.

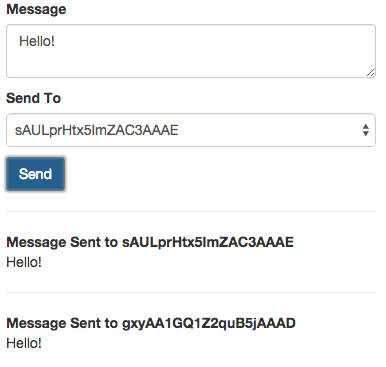
# Setting Up a Default Room

In Socket.IO, every socket that makes a connection is assigned a default room that messages can be emitted to. The default room could be used for a wide variety of purposes.

A practical use of the default room is for storing “friends” or “followers” of the socket. When another socket joins the default room of a socket, we can assume that the socket is interested in getting updates from the room that it has joined.

Other sockets are able to join the default room of any other socket.It isn’t safe to assume that the default room of a socket only has one member unless the server-side architecture is set up in that way.

In this recipe, we will let our socket emit messages to the default room of any other connected socket. We will build a dropdown list displaying all of our sockets.The client can select the socket it wants to emit messages to.



Insert Image B04893\_04\_06.png

## Getting Ready...

For this recipe, we will use jQuery for event delegation and DOM manipulation.

## How To Do It...

To emit messages to the default room for any connected socket, follow these steps:

1. In our server-side code, we will emit a socket.joined message with the socket id and the default room string. We will also create a listener to send a message to any room id that is passed in:

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

// When a socket connects, the default room will be

// the first item in the socket.rooms array

socket.broadcast.emit('socket.joined', {

userId: socket.id,

room: socket.rooms[0]

});

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

socket.broadcast.to(data.id).emit('message.sent', {

id: socket.id,

message: data.message

});

});

});

1. We will create a client-side template with a form to submit messages to any selected room:

<div class="container">

<form id="message-form">

<p>

<label>Message</label>

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

</p>

<p>

<label>Send To</label>

<select id="send-to" class="form-control">

<!-- This will be populated by JavaScript -->

</select>

</p>

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

</form>

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

</div>

1. Finally, we will write our client-side JavaScript. This will send messages to the server and listen to messages from the server which we will display in the UI of our application:

function messageSent (data) {

$('#messages').prepend(`

<p>

<hr />

<strong>${data.id}</strong><br />

${data.message}

</p>

`);

}

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

$('#send-to').append(`<option>${user.userId}</option>`);

});

socket.on('message.sent', messageSent);

$(function () {

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

e.preventDefault();

messageSent({

id: 'Message Sent to ' + $('#send-to').val(),

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

});

socket.emit('message.send', {

id: $('#send-to').val(),

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

});

});

});

## How It Works...

When each of our sockets created a connection, they were assigned a unique room with an un-guessable random name, that we could use emit messages to the socket as well as any other sockets that might join it.

We sent messages to our new room using the socket.broadcast.to() method. This method took the room name as its only argument and then exposed an emit() method to broadcast messages with. When we sent messages using the room specified in socket.broadcast.to(), they were only delivered to sockets that we members of the newly created room.