Craig McKenna

Build a Simple Chat Application With RabbitMQ on Node.js

RabbitMQ is a message broker. You supply it with messages and it delivers these messages to other subscribed applications and services for processing. In it's most basic implementation RabbitMQ requires a producer to publish a message and a subscribed consumer to consume them.

For our simple chat app, we will hand-off chat messages (producer). RabbitMQ will deliver each message back to our application (consumer). A single app could handle all of this and in truth, RabbitMQ is probably overkill here given the simplicity of this application.

However, because of RabbitMQ, our application can be extensible. For example; messages can be consumed by other services written in different languages on entirely different platforms. These other consumers could be different chat platforms or they could be other data stores or data processing applications. In turn, these other applications can also produce messages.

I envision your head swimming with all sorts of RabbitMQ use case scenarios at this point. So, without further delay, let's do this.

User View

Initialize a node project in your usual way. Then, in the root of your project, create a `public` directory with 2 files: `index.html` and `scripts.js`

Create a basic chat form in index.html. I have included some Bootstrap here for styling.

/public/script.js

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="utf-8">

  <title>RabbitMQ Chat App</title>

  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
    integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
    crossorigin="anonymous"
  >

  <style>
    .top-margin {
      margin-top: 48px;
    }
   </style>
</head>
<body class="top-margin">

  <div class="container">
    <div class="row">

      <div class="col-lg-5 col-lg-offset-3">
        <div class="chat-feed">
          <!-- chat feed -->
        </div>

        <form id="chatForm" method="post">
          <div class="form-group top-margin">
            <label for="name">Name</label>
            <input type="text" class="form-control" id="name" placeholder="Your Chat
Name">
          </div>

          <div class="form-group">
            <label for="message">Message</label>
            <textarea class="form-control" id="message" rows="3"></textarea>
          </div>

          <button type="submit" class="btn btn-primary btn-block">Chat</button>
        </form>
      </div>
    </div>
  </div>
```

```
  <script src="https://code.jquery.com/jquery-2.2.1.min.js"
    integrity="sha256-gvQgAFzTH6trSrAWoH1iPo9Xc96QxSZ3feW6kem+O00="
    crossorigin="anonymous"></script>

  <script src="script.js"></script>
</body>
</html>
```

This looks good. Now we need a way to handle submitted chat messages. Notice I included jQuery for this in the example above but you can use any ajax library just the same.

/public/script.js

```
$('#chatForm').on('submit', function(e) {
  e.preventDefault()

  var data = $('#chatForm').serialize()

  $.ajax({
    type: "POST",
    url: '/api/chat',
    processData: false,
    data: data,
    success: function(response) {
      alert(JSON.stringify(response))
    }
  })
})
```

Node/Express Server

Our `.ajax` method posts to a nonexistent URL. Let's install **express** and **express/body-parser** , and build a server for our chat app.

```
$ npm install express body-parser --save
```

In the root of the project create a new file `app.js` for the chat API.

/app.js

```
var express = require('express')
var bodyParser = require('body-parser')
var router = express.Router()

app.use(bodyParser.urlencoded({ extended: true }))
app.use('/api', router)

router.route('/chat')
 .post(function(req, res) {
   // do something with the chat request

   res.send('{"success": true}')
 })
```

Note here that we are using `bodyParser.urlencoded()` because jQuery `.serialize()` returns a url encoded string.

Now to serve our app from the public directory using Express static middleware.

/app.js

```
...

app.use(express.static('public'))

app.listen(3030, '0.0.0.0',
  function() {
    console.log('Chat at localhost:3030')
  }
)
```

Very good, we have an app that posts a name and message to an express server the server responds with some JSON indicating that the message was received.

COMPOSE RAbbitMQ Setup

> Before you continue you may find it helpful to read **Getting Started with RabbitMQ**

To get started with RabbitMQ either sign into your COMPOSE account or get a free 30-day trial at **https://www.compose.io** and create a new RabbitMQ deployment.

Once your deployment is provisioned go to 'create a user' under **First Steps** on the **Overview** tab of your deployment and create a super user.

```
add_user super_user YourPassword
```

Then grant super_user full permissions

```
set_permissions -p your-deployment-name super_user .* .* .*
```

Now, back on the **Overview** tab, under **Recommended** go to 'IP Whitelisting' and add your public IP address to the list by clicking on 'Add IP'. If you deploy your app you will need to add your production server IP here as well.

That's it for our RabbitMQ configuration on COMPOSE. Easy!

SSL

COMPOSE only accepts secure connections so you will need a **SSL key** and **SSL Certificate**.

> If you are unsure how to do this, do an online search for 'self-signed SSL certificate'. You will find plenty of examples for your setup. A self-signed certificate is adequate for development.

You can place them anywhere in your project. I placed mine in a directory called **/certs**

Now go back to compose.io and get the SSL certificate from your RabbitMQ deployment by clicking on 'SSL Public key'. Copy and paste the contents into a file named **compose.crt** in your **/certs** directory.

/certs/compose.crt

```
-----BEGIN CERTIFICATE-----
EverythingInTheCertificate
FromYourComposeDeployment
-----END CERTIFICATE-----
```

Talking to RabbitMQ

We are going to use the AMQP 0-9-1 protocol to communicate with RabbitMQ with help from the amqp.node library (amqplib on npm).

```
npm install amqplib --save
```

> According to RabbitMQ AMQP 0-9-1 is it's "core" protocol. It's also the protocol that they recommend but it supports other protocols as well.

For secure connections, amqplib accepts an options object that contains:

1. Your certificate
2. Your private key
3. Your private key passphrase
4. An array of trusted certificates (compose.crt).

From the amqplib documentation on Using SSL/TLS:

> The certificates and keys are supplied as byte buffers or strings, and generally obtained from files; it's usually easiest, therefore, to just read the files into buffers synchronously (they're not big, and you only need to do it once):

Ok, now create a new file to connect to RabbitMQ:

/connection.js

```
var join = require('path').join
var rfs = require('fs').readFileSync
var amqp = require('amqplib/callback_api')

var opts = {
  cert: rfs(join(__dirname, '../certs/server.crt')),
  key: rfs(join(__dirname, '../certs/server.key')),
  passphrase: 'ThePassphraseForYourKey',
  ca: [rfs(join(__dirname, '../certs/compose.crt'))]
}
```

Notice that we are using amqplib's callback api. If you prefer, the default api is promise based. Also, we are using node's built-in 'fs' library to synchronously buffer the cert files.

Now, let's test our connection. You will need one of the connection strings from your COMPOSE deployment. They are located on the Overview tab of your COMPOSE deployment in the 'Connection info' section.

COMPOSE Deployment, 'Overview' Tab, under 'Connection info'

```
amqps://[username]:[password]@aws-us-east-1-portal.11.dblayer.com:27278/lovely-rabbitmq-00
amqps://[username]:[password]@aws-us-east-1-portal.10.dblayer.com:10857/lovely-rabbitmq-00
```

/connection.js

```
...

amqp.connect('amqps://[username]:[password]@aws-us-east-1-
portal.10.dblayer.com:2278/lovely-rabbitmq-00',
  opts, function(err, conn) {
    if (err) {
      throw new Error(err)
    }

    console.log(conn)
    conn.close()
})
```

Substitute `[username]` and `[password]` with **super_user and the password you set earlier for your super user.*

To test the connection run:

```
$ node connection
```

If you did everything correctly the script will log the connection object to the console, then close the connection and exit.

Sending Messages

In order to use the connection we created, you will need to export it as a function for use in our app. We'll attach the connection as an argument to a callback function.

connection.js

```
...

module.exports = function(cb) {
  amqp.connect('amqps://[username]:[password]@aws-us-east-1-
portal.10.dblayer.com:2278/lovely-rabbitmq-80',
  opts, function(err, conn) {
    if (err) {
```

```
      throw new Error(err)
    }

    cb(conn)
  })
}
```

Great! Now we can use the connection in our app...

First, import the connection. Then, 'use' it in our `.post()` method :

/app.js

```
var express = require('express')
var bodyParser = require('body-parser')
var connection = require('./connection') // <----
var app = express()
var router = express.Router()

...
```

Then, send a chat message to RabbitMQ when one posts to the server:

```
...

router.route('/chat')

 .post(function(req, res) {
   rabbitConn(function(conn) { // <----
    res._rabbitConn.createChannel(function(err, ch) {
      if (err) {
        throw new Error(err)
      }

      var q = 'chat'
      var msg = JSON.stringify(req.body)

      ch.assertQueue(q, {durable: false})
      ch.sendToQueue(q, new Buffer(msg))
      res.send({success: true, sent: req.body})
    })
```

```
    )}
  })
```

Ok, let's break this one down:

1. When a new chat message posts, we open a communication channel to RabbitMQ using the connection we imported; `rabbitConn()`
2. Then we pass the channel to a callback function `function(err, ch) {}`
3. Now we invoke 2 methods on the open channel. The first method; `ch.assertQueue(q, {durable: false})` basically tells RabbitMQ; "Create a queue named 'chat' unless one all ready esists".
4. `ch.sendToQueue(q, new Buffer(msg))` publishes the message to the 'chat' queue.

RabbitMQ may seem like a black box but I assure you, the message is on the queue. There is a simple way to see for yourself:

First, Run `node app` and submit a new message from `localhost:3030`

Then, navigate to the 'Admin UI' url from the overview tab of your COMPOSE deployment and login using your 'super_user' credentials. There, under the `Queues` tab you will see the 'chat' queue. Click it and you will see an option to 'Get messages'.

Now to receive messages...

Pub/Sub

By using the simple messaging pattern above, consumers can receive messages but this pattern has limitations. RabbitMQ will deliver the message to one consumer at a time in a round-robin fashion where one consumer receives a message, then, another consumer receives the next message and so on.

To solve this we will use the Publish/Subscribe or 'Pub/Sub' pattern in which each consumer will own a unique queue and receive messages from a central RabbitMQ exchange. In order for each user to have a continuous, real-time, open channel to the consumer, we will use Socket.io.

Since this tutorial is about RabbitMQ, I won't go into much detail regarding Socket.io here.

First, install **Socket.io**

```
npm install socket.io --save
```

Now, create a Node **HTTP** server and bind **Socket.io** and our **express** app to it.

/app.js

```
var express = require('express')
var bodyParser = require('body-parser')
var rabbitConn = require('./connection')
var app = express()
var router = express.Router()
var server = require('http').Server(app) // <----
var io = require('socket.io')(server) // <----
```

To implement the Pub/Sub pattern we'll use the RabbitMQ exchange type 'fanout'. With 'fanout', every consumer that is subscribed to the exchange will receive the messages.

To do this modify our Express `.post()` method like so:

```
...

  .post(function(req, res) {
    rabbitConn(function(conn) {
      conn.createChannel(function(err, ch) {
        if (err) {
          throw new Error(err)
        }
        var ex = 'chat_ex'
        var msg = JSON.stringify(req.body)

        ch.assertExchange(ex, 'fanout', {durable: false})
        ch.publish(ex, '', new Buffer(msg), {persistent: false})
        ch.close(function() {conn.close()})
      })
    })
  })

...
```

See how we publish to a named exchange but we didn't name the queue? Previously we published to a named queue on the default 'unnamed exchange'. The second argument to `.publish()`; `''` (empty string) tells RabbitMQ to publish the message only to the named exchange and not to any specific queue.

One other thing that's new here is the line; `ch.close(function() {conn.close()})`. If we don't close the connection RabbitMQ will create a new one every time a message is

posted and we end up with many open connections. As a result, we would receive multiple copies of messages when we consume them (one for each open connection).

Also, a note about `{durable: false}` and `{persistent: false}`:

- RabbitMQ 'holds on' to durable queues and exchanges, and persistent messages indefinitely. They are also written to disk to safeguard against server errors and restarts, etc. We don't need or even want durability or persistence here, more on this later...

Now, we just subscribe to the `chat_ex` exchange and emit messages to the client via Socket.io

/app.js

```
var express = require('express')
var bodyParser = require('body-parser')
var rabbitConn = require('./connection')
var app = express()
var router = express.Router()
var server = require('http').Server(app)
var io = require('socket.io')(server)

// Consume Messages and 'broadcast' over all open client connections
var chat = io.of('/chat')

rabbitConn(function(conn) {
  conn.createChannel(function(err, ch) {
    if (err) {
      throw new Error(err)
    }
    var ex = 'chat_ex'

    ch.assertExchange(ex, 'fanout', {durable: false})
    ch.assertQueue('', {exclusive: true}, function(err, q) {
      if (err) {
        throw new Error(err)
      }
      ch.bindQueue(q.queue, ex, '')
      ch.consume(q.que, function(msg) {
        chat.emit('chat', msg.content.toString())
      })
    }, {noAck: true})
  })
})
//-----------------------------------------------------------------
```

```
app.use(express.static('public'))
app.use(bodyParser.urlencoded({ extended: true }))
app.use('/api', router)
router.route('/chat')

  .post(function(req, res) {
    rabbitConn(function(conn) {
      conn.createChannel(function(err, ch) {
        if (err) {
          throw new Error(err)
        }
        var ex = 'chat_ex'
        var msg = JSON.stringify(req.body)

        ch.assertExchange(ex, 'fanout', {durable: false})
        ch.publish(ex, '', new Buffer(msg), {persistent: false})
        ch.close(function() {conn.close()})
      })
    })
  })

server.listen(3030, '0.0.0.0',
  function() {
    console.log('Chat at localhost:3030')
  }
)
```

Here we declared a channel of sorts for Socket.io to broadcast on. Then, we open a new connection and channel to RabbitMq, subscribe to the `chat_ex` exchange and instantiate a new unnamed queue (remember that RabbitMQ assigns a unique name for us). We then, pass the returned, uniquely named queue, to a callback function, where we bind the queue; `q.queue` to the exchange. Finally, we consume any messages.

Instead of closing the channel, we keep the channel open and continuously 'push' new messages to the client using `chat.emit()`

There are two more parameters of note here: `{exclusive: true}` and `{noAck: true}`.

1. RabbitMq deletes 'exclusive' queues when the connection associated with it closes.
2. You can configure RabbitMQ for message acknowledgements; `{noAck: false}`. In which case it will requeue delivered messages until it receives an acknowledgement from a consumer. Otherwise (as in our case here), messages are discarded as soon as they are delivered.

Rendering Chat Messages on the Client

Before you can fire up the app and give it a whirl, there are a few things to handle on the front-end. First, like I did, you may have found our chat form lacking, in that a user enters their name in the same form that is used to send a new chat message. Second, we need to receive the messages over Socket.io and render them to our **.chat-feed div**.

Let's create a modal that opens on page load where our users will enter a 'chat name'. We'll pass the given name to the chat form and assign it to the value of a hidden input where the name will be handled on 'submit' just as before.

Note, that you will need to include **bootstrap.js** in order to do this.

/public/index.html

```
...

<body class="top-margin">
  <div class="container">


    <!-- Modal -->
    <div class="modal fade" id="modal" tabindex="-1" role="dialog" aria-
labelledby="chatName">
      <div class="modal-dialog modal-sm" role="document">
        <div class="modal-content">

          <div class="modal-header">
            <h4 class="modal-title">Hello, Who Are You?</h4>
          </div>

          <form id="chatNameForm" method="post">
            <div class="modal-body">
              <div class="form-group">
                <label for="chatName">Name</label>
                <input type="text" class="form-control" id="chatName" name="chatName"
placeholder="Your Chat Name">
              </div>
            </div>

            <div class="modal-footer">
              <button type="submit" class="btn btn-primary btn-block">Lets Chat</button>
            </div>
          </form>
```

```
        </div>
      </div>
    </div>

    <div class="row">

      <div class="col-lg-5 col-lg-offset-3">
        <div class="chat-feed">
          <!-- chat feed -->
        </div>

        <form id="chatForm" method="post">
          <div class="form-group">
            <label for="message">New Message</label>
            <textarea class="form-control" id="message" name="message" rows="3">
</textarea>
          </div>


          <!-- Hidden Name Input -->
          <input type="hidden" id="name" name="name">
          <button type="submit" class="btn btn-primary btn-block">Chat</button>
        </form>
      </div>
    </div>
  </div>

  <script src="https://code.jquery.com/jquery-2.2.1.min.js"
    integrity="sha256-gvQgAFzTH6trSrAWoH1iPo9Xc96QxSZ3feW6kem+O00="
    crossorigin="anonymous"></script>


  <!-- More JS -->
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
    integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
    crossorigin="anonymous"></script>

  <script src="/socket.io/socket.io.js"></script>

  <script src="script.js"></script>
</body>
</html>
```

Notice that I included: **socket.io.js** and **bootstrap.min.js**.

Also, remember to update the chat form, specifically: `<input type="hidden" id="name" name="name">`

Now, with a few changes to **script.js**, we will have a working chat application!

/public/script.js

```
var chatSocket = io('/chat')

$(document).ready(function() {
  $('#modal').modal('show')
})

$('#chatNameForm').on('submit', function(e) {
  e.preventDefault()
  var name = $('#chatName').val()

  $('#name').val(name)
  $('#modal').modal('hide')
})

function updateFeed (message) {
  var newMessage = '<div><strong>' + message.name + '</strong><p>' + message.message +
'</p></div>'

  $('.chat-feed').append(newMessage)
}

chatSocket.on('chat', function(message) {
  message = JSON.parse(message)
  updateFeed(message, 'append')
})

...
```

The code above reads something like this:

1. Declare a new Socket.io socket.
2. When the document is ready, show the modal that contains our chat name form.
3. When `#chatNameForm` is submitted, set the value of the hidden `#name` input to the value the user entered in the modal form. Then, hide the modal.
4. Receive new messages over the **chatSocket** and append them to the chat feed.

We now have a working chat application using Node and RabbitMQ. Go ahead, give it a try: `$ node app`

New and Returning Users

There is an issue that right away. When new users arrive, they don't see any of the messages that were sent previous their arrival.

We can fix this issue with the simple messaging pattern that we used initially to publish a message to a named queue. Here's how.

1. Publish each posted message to a named, durable queue, using the `{persistent: true}` option.
2. Recreate the chat session by consuming the persisted messages each time a user returns or a new user arrives.
3. Retrieve the messages with AJAX HTTP GET via our Express chat API.

Let's start with the client. I have modified **script.js**. See if you can figure out how it works and then we'll go over the changes.

/public/script.js (complete)

```
var chatSocket = io('/chat')

$(document).ready(function() {
  $('#modal').modal('show')
})

$('#chatNameForm').on('submit', function(e) {
  e.preventDefault()
  var name = $('#chatName').val()

  $('#name').val(name)
  getMessages()
})

function updateFeed (message, method) {
  var newMessage = '<div><strong>' + message.name + '</strong><p>' + message.message +
'</p></div>'

  if (method === 'append') {
    $('.chat-feed').append(newMessage)
  } else if (method === 'prepend') {
```

```
    $('.chat-feed').prepend(newMessage)
  }
}

function getMessages() {
  $.ajax({
    type: 'GET',
    dataType: 'json',
    url: '/api/chat',
    success: function(response) {
      if (response.messages) {
        response.messages.reverse().map(function(message){
          updateFeed(message, 'prepend')
          $('#modal').modal('hide')
        })
      } else {
        $('#modal').modal('hide')
      }

      chatSocket.on('chat', function(message) {
        message = JSON.parse(message)
        updateFeed(message, 'append')
      })
    }
  })
}

$('#chatForm').on('submit', function(e) {
  e.preventDefault()
  var data = $('#chatForm').serialize()
  $.ajax({
    type: 'POST',
    url: '/api/chat',
    processData: false,
    data: data
  })
})
```

Ok, did you figure it out? Let's see.

Instead of closing the modal immediately after the user enters their name, we now call the new `getMessages()` function.

1. First, the function retrieves the messages from the server.
2. On successful retrieval:

- If there are messages it calls the now modified `updateFeed()` function and now instead of appending the feed it prepends the feed with all of the messages in reverse order. Then it hides the modal.
- If there are no messages the feed is not changed and the modal is then hidden.
- Notice that I moved the `chatSocket.on()` method to the success function as well. If I hadn't the user may see duplicate messages because any new messages posted during this process will be rendered once across Socket.io and then again when `getMessages()` is called.

Again, see if you can figure out the changes to **app.js**

/app.js (complete)

```
var express = require('express')
var bodyParser = require('body-parser')
var rabbitConn = require('./connection')
var app = express()
var router = express.Router()
var server = require('http').Server(app)
var io = require('socket.io')(server)


var chat = io.of('/chat')

rabbitConn(function(conn) {
  conn.createChannel(function(err, ch) {
    if (err) {
      throw new Error(err)
    }
    var ex = 'chat_ex'

    ch.assertExchange(ex, 'fanout', {durable: false})
    ch.assertQueue('', {exclusive: true}, function(err, q) {
      if (err) {
        throw new Error(err)
      }
      ch.bindQueue(q.queue, ex, '')
      ch.consume(q.que, function(msg) {
        chat.emit('chat', msg.content.toString())
      })
    }, {noAck: true})
  })
})
```

```
app.use(express.static('public'))
app.use(bodyParser.urlencoded({ extended: true }))
app.use('/api', router)
router.route('/chat')

  .post(function(req, res) {
    rabbitConn(function(conn) {
      conn.createChannel(function(err, ch) {
        if (err) {
          throw new Error(err)
        }
        var ex = 'chat_ex'
        var q = 'chat_q'
        var msg = JSON.stringify(req.body)

        ch.assertExchange(ex, 'fanout', {durable: false})
        ch.publish(ex, '', new Buffer(msg), {persistent: false})
        ch.assertQueue(q, {durable: true})
        ch.sendToQueue(q, new Buffer(msg), {persistent: true})
        ch.close(function() {conn.close()})
      })
    })
  })

  .get(function(req, res){
    rabbitConn(function(conn){
      conn.createChannel(function(err, ch) {
        if (err) {
          throw new Error(err)
        }

        var q = 'chat_q'

        ch.assertQueue(q, {durable: true}, function(err, status) {
          if (err) {
            throw new Error(err)
          }
          else if (status.messageCount === 0) {
            res.send('{"messages": 0}')
          } else {
            var numChunks = 0;

            res.writeHead(200, {"Content-Type": "application/json"})
            res.write('{"messages": [')

            ch.consume(q.que, function(msg) {
```

```
              var resChunk = msg.content.toString()

              res.write(resChunk)
              numChunks += 1
              numChunks < status.messageCount && res.write(',')

              if (numChunks === status.messageCount) {
                res.write(']}')
                res.end()
                ch.close(function() {conn.close()})
              }
            })
          }
        })
      }, {noAck: true})
    })
  })

server.listen(3030, '0.0.0.0',
  function() {
    console.log('Chat at localhost:3030')
  }
)
```

In addition to publishing to the `chat_ex` exchange, we are also publishing posted messages to the durable queue; `chat_q`.

The messages are consumed in the added `.get()` method. Because the messages are persistent they can be retrieved again, the next time they are needed.

Because each message is consumed independently, we stream the messages over a single response in individual chunks.

Notice that `.assertQueue()` returns a status object. Here we are using `status.messageCount`. We customize our response based on whether or not there are any messages in the queue. If there are no messages, we respond to the GET request with '{"messages": 0}'. If there are messages we increment `numChunks` by one for every message consumed. When `numChunks` equals `status.messageCount` we end the response and close the connection.

That's it. A fully functional chat application!

I hope you found this tutorial useful. I also hope that you extend it in ways that work well for you and your users.

Enjoy.

< previous next >