**Notification Service**

How to build your own notification service with Node js andRabbitMQ?

This article will help you to grasp how to build a notification service with the help of message queue in node js.

Using a message queue is a no-brainer if you need to throttle the requests going to your API or workers, or you want things to happen asynchronously.

If you still want more reasons, please go through the following article.

<https://stackify.com/message-queues-12-reasons/>

Some popular message queues are:

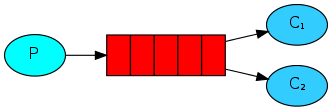
* RabbitMQ
* Kafka
* IBM MQ
* ActiveMQ
* RocketMQ
* Qpid

In this we are using RabbitMQ.

RabbitMQ is the most popular open source message broker. RabbitMQ is lightweight and easy to deploy. It supports multiple messaging protocols. RabbitMQ has decent performance and a strong community. If your requirement is to process thousands of messages per second, I would suggest you go for something like RabbitMQ.

To be honest, the RabbitMQ documentation is so well written (<https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html>) , you don’t need anything else. But we need a RabbitMQ tutorial as part of this series. So I will be summarizing the documentation to some extent and see how we can build a REST API to work with RabbitMQ drivers.

A normal work queue setup in RabbitMQ looks like:



**Work Queues setup in RabbitMQ**

There are 3 important parts in the RabbitMQ setup:

1. Producer. A program that sends messages is a producer. (Represented by P in the diagram).
2. Queue. Messages are stored inside a queue. It’s essentially a large message buffer. A queue is only bound by the host’s memory and disk limits. (Represented by the red rectangular boxes in the diagram).
3. Consumer. A consumer is a program that mostly waits to receive messages. (Represented by C1 & C2 in the diagram).

There can be multiple producers and consumers to a queue. But in normal use cases, we don’t need multiple producers, as putting messages into a queue is quite simple and fast. But consuming a message from the queue and performing a task based on the message normally takes time. Thus, we tend to implement multiple consumers for a queue, so that when a consumer is busy, the other consumer can read from the queue and perform the task.

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. In this case, if the worker crashes, we will lose the message it was just processing. In order to make sure a message is never lost, RabbitMQ supports message acknowledgments.

An (acknowledgement) is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed, and that RabbitMQ is free to delete it. RabbitMQ will understand that a message wasn’t processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer.

But what if the RabbitMQ server crashes? Will the messages in the queue get lost?

By default, yes. But there are steps to ensure your queue and messages are persistent to your disk. So that, in the case of a server crash, you can restart your server and process the messages.

First, we must ensure our queue is declared as durable. We can either do that in RabbitMQ admin console, or through code. Then we should ensure the messages in the queue are persistent by setting the persistent field as true

We can have a RabbitMQ setup on our local machine by installing RabbitMQ, or you can run a RabbitMQ as a container in a docker.

Once setup is done, we are good to use RabbitMQ admin console where you can get all information regarding Queues, Exchanges, Connections, Channels etc.

Messages are not published directly to a queue. Instead, the producer sends messages to an exchange.

**Exchanges** are message routing agents, defined by the virtual host within RabbitMQ.

An exchange is responsible for routing the messages to different queues with the help of header attributes, bindings, and routing keys.

A **binding** is a "link" that you set up to bind a queue to an exchange.

RabbitMQ documentation is so well written in detail regarding exchanges and its types. (<https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html>)

You can add or delete Queue/Exchange in the console itself.

Ideally we should have a producer and consumers which will have the relationship based on the Queues.

In simple words the producer will put the message to Queue in RabbitMQ, the consumer listening to the respective Queue will consume the message and later it will perform the respective tasks.

**In concept of Notification service:-**

Let me brief in simple words.

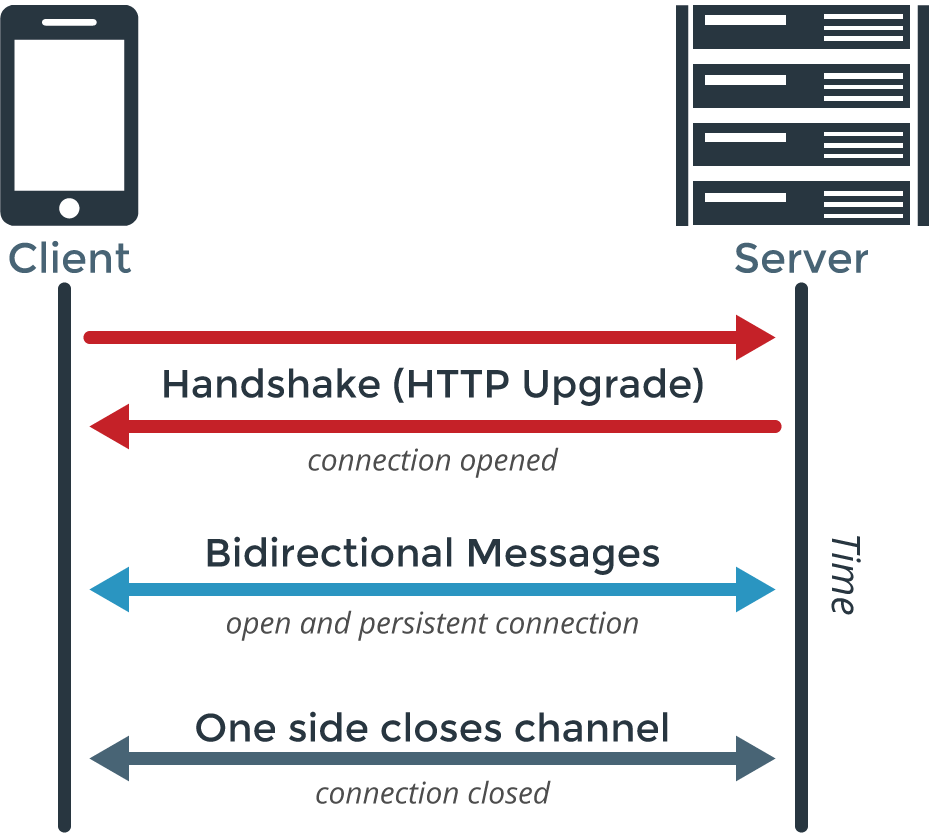
Client (reactjs app)  ----> Server(Nodejs)

Notification Server (Nodejs)

For understanding how the consumed message or notification will get to the client we have to first know about the concept of sockets and how it works.

**WebSocket?**

WebSocket’s can be regarded as the future of client-server communication. In WebSocket’s there would be one TCP connection between client & server at all times. There’s bi-directional data flow between client & server as well as real-time nature due to always open TCP connection.

****

**An Introduction to Socket.io:-**

**What is Socket.io?**

Socket.io is a cross-browser JavaScript library that provides event-driven communication capabilities for use in real-time applications. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have an identical API.

**Initialize Connection**

To begin with I need to install the npm package by npm install socket.io. Client can use client code from existing socket.io package, use CDN JavaScript file, or client based socket.io-client. The server is based on Express in this case but plain NodeJS code is there in the documentation.



In this instance when we simply connect to localhost:3000 in the browser it can be seen in the running server instance as ‘a user connected’. Initialization of connection is simple as that.

**Bidirectional Communication**

Communication between client and server is even simpler. The sender would have to socket.emit, while the client can listen on socket.on.



Here the channel is a custom\_event which either client or server can emit (send) or on (receive).

I hope from the above I was able to give a brief idea about WebSockets & SocketIO.

Now will come to our scenario where the client and servers will communicate through the socket connection in order to implement the notifications.

Assume that we have a client written in ReactJS, server in NodeJS(REST APIS) which will deal with the data for storing and retrieving and a notification server in NodeJS.

**OUR APP CLIENT**: - We have our UI which is developed using ReactJS. I am not going in depth on how we have components in the application and am focusing on how we will implement the notifications.

We must maintain a socket connection with the notification server in order to get the real time notifications.

**OUR APP SERVER:** - We have our server, which will take care of all application flow through the REST APIS. Also, the Publisher will be there which will in response push or publish the messages/notifications to the message queue.

**NOTIFICATION SERVER: -** We can say in other words notification server is a consumer which always listens to the binded queue of RabbitMQ. As i already explained in the client a socket connection must establish between client and notification server.

Soon when the publisher (Our App Server) pushes/publishes a message to queue then the consumer (Notification server) will consume and will transmit through the socket connection.

I hope from the above I was able to give a brief idea about how real time notifications work.

Let me give sample code for the publisher and consumer.

**The amqp.node client library**

We'll use the amqp.node client.

First, install amqp.node using npm: **npm install amqplib**

**var amqp = require('amqplib/callback\_api');**

then connect to RabbitMQ server

**Publisher :-**

amqp.connect('amqp://localhost', function(error0, connection) {});

amqp.connect('amqp://localhost', function(error0, connection) {

  if (error0) {

    throw error0;

  }

  connection.createChannel(function(error1, channel) {});

});

To send, we must declare a queue for us to send to; then we can publish a message to the queue:

**amqp.connect('amqp://localhost', function(error0, connection) {**

**if (error0) {**

**throw error0;**

**}**

**connection.createChannel(function(error1, channel) {**

**if (error1) {**

**throw error1;**

**}**

**var queue = 'hello';**

**var msg = 'Hello world';**

**channel.assertQueue(queue, {**

**durable: false**

**});**

**channel.sendToQueue(queue, Buffer.from(msg));**

**console.log(" [x] Sent %s", msg);**

**});**

**});**

**Consumer :-**

Setting up is the same as the publisher; we open a connection and a channel, and declare the queue from which we're going to consume. Note this matches up with the queue that **sendToQueue** publishes to.

**amqp.connect('amqp://localhost', function(error0, connection) {**

**if (error0) {**

**throw error0;**

**}**

**connection.createChannel(function(error1, channel) {**

**if (error1) {**

**throw error1;**

**}**

**var queue = 'hello';**

**channel.assertQueue(queue, {**

**durable: false**

**});**

**});**

**});**

Note that we declare the queue here, as well. Because we might start the consumer before the publisher, we want to make sure the queue exists before we try to consume messages from it.

We're about to tell the server to deliver us the messages from the queue. Since it will push us messages asynchronously, we provide a callback that will be executed when RabbitMQ pushes messages to our consumer. This is what **Channel.consume** does.

**console.log(" [\*] Waiting for messages in %s. To exit press CTRL+C", queue);**

**channel.consume(queue, function(msg) {**

**console.log(" [x] Received %s", msg.content.toString());**

**}, {**

**noAck: true**

**});**

The above is the sample code for the consumer and publisher. This is how we can build the notification service.

As this is my first article, I tried to explain how to build the notification service.