webapplog [programming weblog]

Software engineering, Node.js, JavaScript and startups.

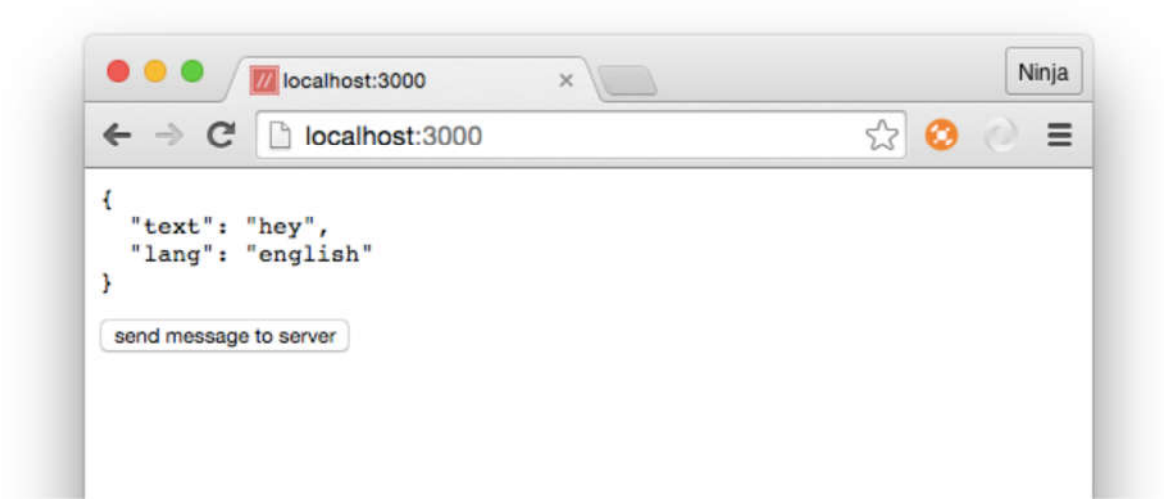
JSON is Not Cool Anymore: Implementing Protocol Buffers in Node.js

There's a better alternative to the ubiquitous JSON as the communication protocol of the web. It's Protocol Buffers (protobuf). In a nutshell, protobuf offers a more dense format (faster processing) and provides data schemas (enforcement of structure and better compatibility with

Protocol Buffers were introduced by Google. You can read, more about them at the official Protocol Buffers Developer Guide. For something shorter, read 5 Reasons to Use Protocol Buffers Instead of JSON For Your Next Service which will give you a quick overview of the protobuf benefits over JSON.

The purpose of this article is not to highlight why protobufs are better or sell you on the concept. There are many article online that'll do it for you. The purpose of this article is to show you how you can get started with this format in the Node.js environment.

This article will walk you through a RESTful API implementation of protocol buffers with Node.js, Express.js, Axios and Protobuf.js. The code in this post runs on Node v6.2.0 because it's written in the cutting-edge ES6/ES2015 version of the JavaScript language. We'll have a message consisting of two fields `text` and `lang` sent as protobuf from the server, decoded and shown on the browser. We'll also have a button which will send another protobuf message to the server. The source code's in the GitHub repository azat-co/proto-buffer-api.



This will be the structure of our project:

```
/proto-buffer-api
  /public
    axios.min.js
    bytebuffer.js
    index.html
    long.js
    message.proto
    protobuf.js
  /node_modules
  index.js
  package.json
```

The `public` folder is where all of our browser assets will reside. We have Axios to make HTTP requests from the browser to server. It's similar to Superagent or Request. You can also use jQuery to make HTTP requests. If you're going to use a library other than Axios, just make sure you are submitting data as ArrayBuffer and sending it as `application/octet-stream`.

The Protobuf.js is the library to work with Google's Protocol Buffers in JavaScript and Node.js so we'll need `protobuf.js` file on the browser. It requires support for long numbers (numbers in JavaScript are limited to 53 bits in size as you know) and there's a neat library to allow us to work with 64-bit integers called long.js.

which we'll send from server to browser and back. It looks like this:

```
message Message {
    required string text = 1;
    required string lang = 2;
}
```

Protobuf.js requires one more dependency— `bytebuffer.js` for the ArrayBuffer data type.

The format is relatively easy to understand. We have two fields `text` and `lang`. They re both required fields. The numbers next to the field names is something protocol buffers need for decoding/encoding.

The `index.html` has minimal HTML which contains libraries includes, `<pre>` container where we'll insert the response from the server, the button which triggers `sendMessage()` (we'll write it later), and the `<script>` tag with requests and protobuf code.

```
<html>
  <head>
    <script src="long.js"></script>
    <script src="bytebuffer.js"></script>
    <script src="protobuf.js"></script>
    <script src="axios.min.js"></script>
  </head>
  <body>
    <pre id="content"></pre>
    <button onClick="sendMessage()">send message to server</button>
```

```
    </script>
  </body>
</html>
```

Let's dive deeper browser JavaScript and implement two requests: a GET request to fetch a message from the server and a POST request to send a message to the server. They both will have to work with protocol buffers.

First of all, we create `Message` from our prototype file `message.proto`. In the callback of `loadProtoFile` we can invoke `loadMessage()` to make the GET request to the server.

[Sidenote]

Reading blog posts is good, but watching video courses is even better because they are more engaging.

A lot of developers complained that there is a lack of affordable quality video material on Node. It's distracting to watch to YouTube videos and insane to pay $500 for a Node video course!

Go check out Node University which has FREE video courses on Node: node.university.

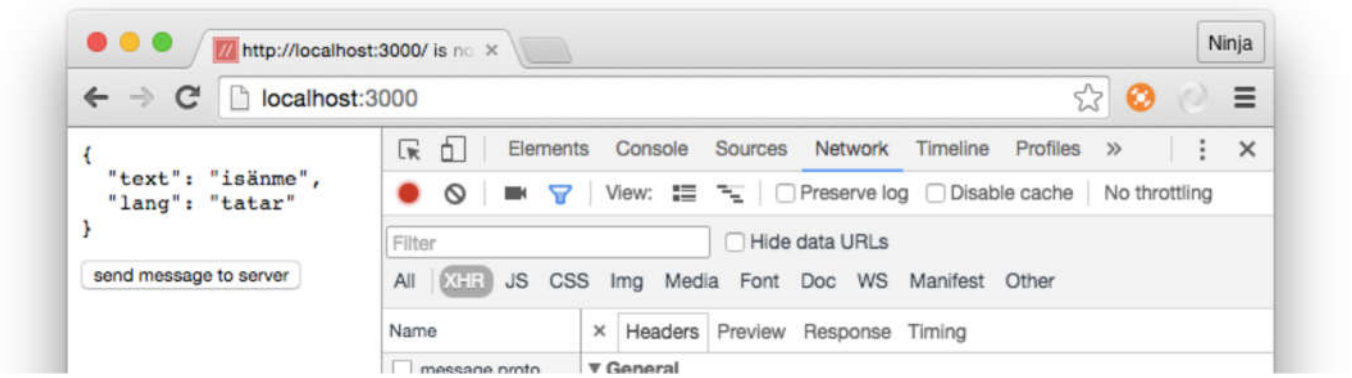
[End of sidenote]

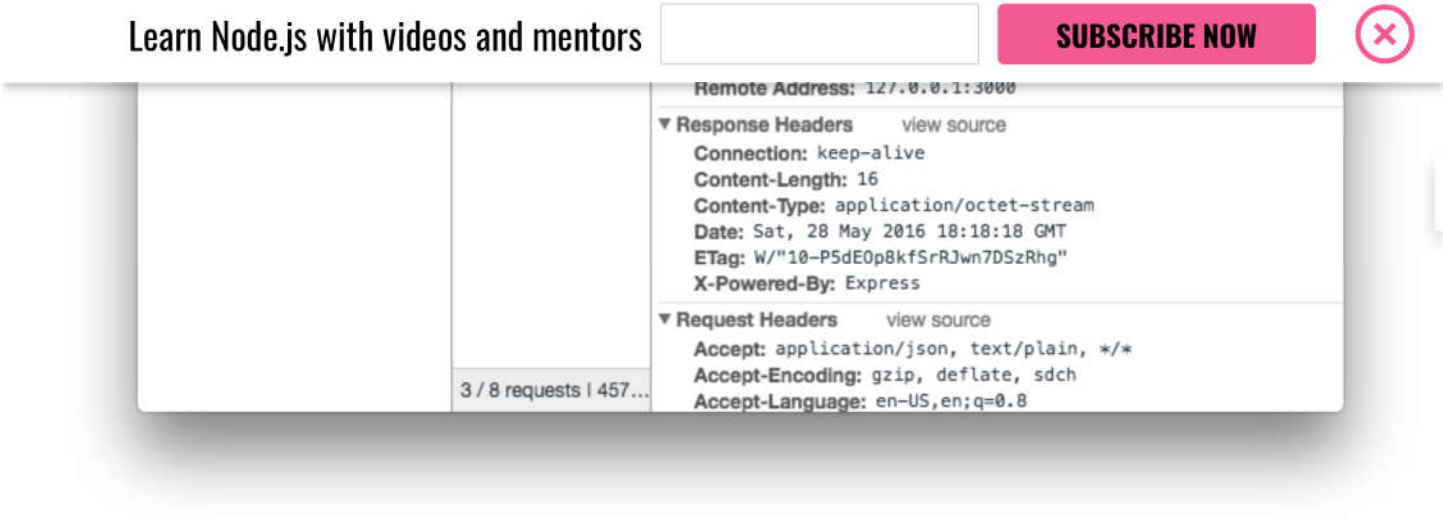
```
"use strict";
let ProtoBuf = dcodeIO.ProtoBuf
let Message = ProtoBuf
  .loadProtoFile('./message.proto', (err, builder)=>{
    Message = builder.build('Message')
    loadMessage()
```

Axios library take as the first argument the URL of the request and as a second the request options. One of the options we must provide is `arraybuffer`. This will tell the HTTP agent to give us the appropriate data type back. Axios works with promises, so the in `then` callback, we can get `response`, log it and decode using `Message.decode()`:

```
let loadMessage = ()=> {
  axios.get('/api/messages', {responseType: 'arraybuffer'})
    .then(function (response) {
      console.log('Response from the server: ', response)
      let msg = Message.decode(response.data)
      console.log('Decoded message', msg)
      document.getElementById('content').innerText = JSON.stringify(msg, null, 2)
    })
    .catch(function (response) {
      console.log(response)
    })
}
```

The result of the GET request is shown in DevTools in the screenshot below. You can observe that the response's in `application/octet-stream`:

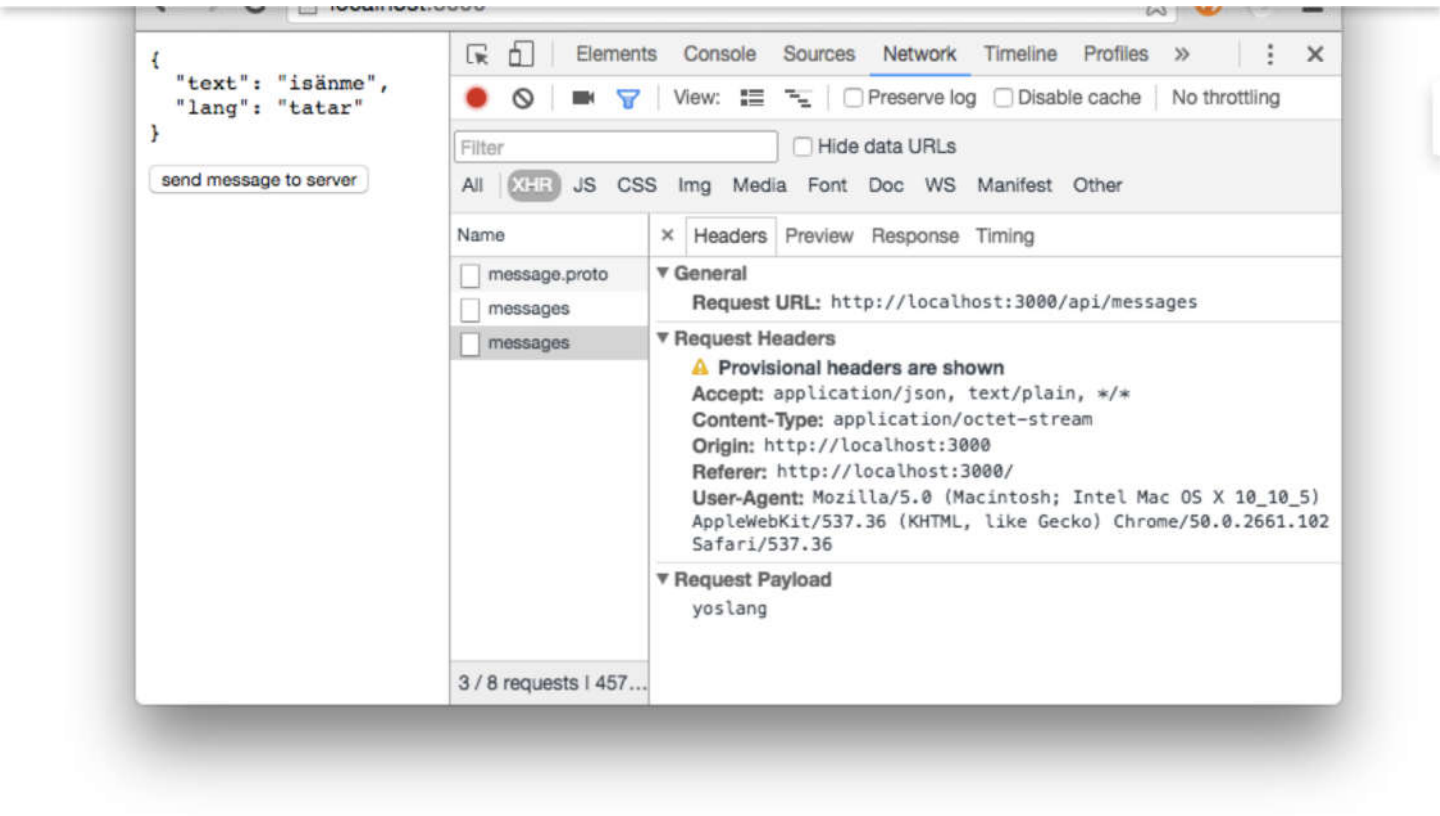




As for the sending of the protocol buffers to the server, make sure to create object with `new Message(data)` and then invoke `msg.toArrayBuffer()`. It's a good idea to set the `Content-Type` header to `application/octet-stream` so server knows the format of the incoming data:

```
let sendMessage = ()=>{
  let msg = new Message({text: 'yo', lang: 'slang'})
  axios.post('/api/messages', msg.toArrayBuffer(),
      { responseType: 'arraybuffer',
      headers: {'Content-Type': 'application/octet-stream'}}
    ).then(function (response) {
      console.log(response)
    })
    .catch(function (response) {
      console.log(response)
    })
}
```

The result of POST with the appropriate `Content-Type` and payload is shown in the screenshot below:



We have the front-end done, but it won't work with our the server code so let's implement Node/Express code next. First of all, you will want to create the `package.json`. Feel free to copy this file which has the dependencies:

```
{
  "name": "proto-buffer-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "dependencies": {
```

```
  }
}
```

Once you have `package.json`, you can install the dependencies with `npm i`. It will install express for building the HTTP server and protobufjs for working with Protocol Buffers on the server.

Let's implement the server code first. In `index.js`, we import dependencies, create the `express` object and apply the static middleware for the `public` folder:

```
let path = require('path')
let express = require('express')
let app = express()
let publicFolderName = 'public'
app.use(express.static(publicFolderName))
```

Next, we'll use an in-memory store to simplify this project. In other words, the data for the GET request will be coming from an array:

```
let messages = [
  {text: 'hey', lang: 'english'},
  {text: 'isänme', lang: 'tatar'},
  {text: 'hej', lang: 'swedish'}
]
```

Typically, you would use `body-parser` for parsing JSON requests. In order

of buffers. Let's implement our own custom middleware to parse protobufs and store them in `body.raw` (the decorator pattern). We need to create `body.raw` only when the header `Content-Type` is `application/octet-stream` and when there's data (`data.length>0`):

```
app.use (function(req, res, next) {
  if (!req.is('application/octet-stream')) return next()
  var data = [] // List of Buffer objects
  req.on('data', function(chunk) {
      data.push(chunk) // Append Buffer object
  })
  req.on('end', function() {
    if (data.length <= 0 ) return next()
    data = Buffer.concat(data) // Make one large Buffer of it
    console.log('Received buffer', data)
    req.raw = data
    next()
  })
})
```

Now we can create the builder object and "build" Message from our prototype file. We use the same prototype file `public/message.proto` as our front-end code:

```
let ProtoBuf = require('protobufjs')
let builder = ProtoBuf.loadProtoFile(
  path.join(__dirname,
  publicFolderName,
  'message.proto')
)
```

Now we can implement GET in which we create a new message, encode it and convert to the Buffer type before sending back to the front-end client. Express's `response.send()` is staking care of adding the proper 'Content-Type'. You can use `response.end()` as well:

```
app.get('/api/messages', (req, res, next)=>{
  let msg = new Message(messages[Math.round(Math.random()*2)])
  console.log('Encode and decode: ',
    Message.decode(msg.encode().toBuffer()))
  console.log('Buffer we are sending: ', msg.encode().toBuffer())
  // res.end(msg.encode().toBuffer(), 'binary') // alternative
  res.send(msg.encode().toBuffer())
  // res.end(Buffer.from(msg.toArrayBuffer()), 'binary') // alternative
})
```

In the POST request handler, we decode from `body.raw` (it's populated by the middleware which we defined earlier), and log in the terminal:

```
app.post('/api/messages', (req, res, next)=>{
  if (req.raw) {
    try {
        // Decode the Message
      var msg = Message.decode(req.raw)
      console.log('Received "%s" in %s', msg.text, msg.lang)
    } catch (err) {
      console.log('Processing failed:', err)
      next(err)
    }
  } else {
```

```
})

app.all('*', (req, res)=>{
  res.status(400).send('Not supported')
})

app.listen(3000)
```

If you typed all the code as I have, or copied from my GitHub repository azat-co/proto-buffer-api, you should see on a web page a random message from the server. Then, if you click on the button, you should see "yo" in the terminal /command prompt where your Node.js server is running.

That's it. We implemented GET and POST to communicate in protocol buffers between Node.js/Express.js and browser JavaScript with Protobuf.js. We used Axios to make HTTP requests from the browser. It allowed us to work with promises and abstract some of the low-level XMLHttpRequest interface for working with binary data.

Google's using Protocol Buffers for their API. Protobufs in many ways superior to JSON or XML, and with Protobuf.js and this quick tutorial you should be good to start using Protocol Buffers for your RESTful APIs!

--

Best Regards,
Azat Mardan
Microsoft MVP | Book and Course Author | Software Engineering

https://www.linkedin.com/in/azatm

To contact Azat, the main author of this blog, submit the contact form or schedule a call at clarity.fm/azat and we can go over your bugs, questions and career.

This entry was posted in Node.js, Tutorials and tagged node.js on June 3, 2016 [https://webapplog.com/json-is-not-cool-anymore/] .

11 thoughts on “JSON is Not Cool Anymore: Implementing Protocol Buffers in Node.js”

It's a good concise How To article... thanks for the blog post! Although, the title is a bit misleading because Protobufs and JSON serve different purposes. For instance, being able to have a human readable format is the most sensible approach for many use cases, especially if you don't need to validate against a schema. I don't want to be too picky, but I recommend you proof read the spelling and fix the typos because it distracts from the text, Azat.

Igor Ganapolsky
January 5, 2017 at 1:26 pm

Thank you for this informative writeup on ProtoBufs with Node.js. Like some others, I am very curious as to what is the use case of ProtoBufs over JSON? Is there a definitive scenario you found in some projects that would eliminate JSON as a candidate for message format, and would declare ProtoBufs as the undeniable choice?

Sebastien
December 23, 2016 at 1:09 am

JSON + Json schemas are fine for me. I don't see any valid reason to switch to protobuf or other.

Gan
November 17, 2016 at 7:57 pm

In a static typing environment protobuf makes some sense but we should not compare with json.

MartinHeidegger
July 3, 2016 at 5:56 pm

I do not enjoy the rigidness of Protocol buffers. MSGPack seems to be significantly more useful in daily development: http://msgpack.org/index.html

Eugenio

Have you try rjson?
https://github.com/dogada/RJSON

KK
June 10, 2016 at 6:51 pm

From my personal experience, I would suggest to stay away from protobuf unless you really have a good reason to. Everyone should first read “When Is JSON A Better Fit” section in the linked “5 Reasons to Use Protocol Buffers Instead of JSON For Your Next Service” post. Because there are no nice developer tools available, it is simply not possible to examine your http requests/responses in browsers’ developers tools. It was a huge pain for my team when debugging and troubleshooting applications that used services with protobuf responses.

Matt Baker
June 10, 2016 at 6:17 pm

I’d love to understand more about why you feel Protocol Buffers are better than JSON.

Chris Kimpton
June 10, 2016 at 2:32 pm

Protocol Buffers are so last year!

Flatbuffers are the future :P

https://google.github.io/flatbuffers/flatbuffers_benchmarks.html

In a real application, it would be better to use body-parser package to retrieve raw request body.

Also, you can write your protobuf parsing code as middleware as well. Then there will be no difference in your route handlers between json and proto-buf, you would work just with response.body whatever it is.

AJ Alger
June 8, 2016 at 4:13 am

You keep saying 'Asios', when it's 'Axios'. A confusing typo.