Node.js is a relatively young platform when it comes to frameworks (unlike Ruby or Java), but there's already a leader that has already become a de facto standard used in the majority of Node.js projects: Express.js.

Express is the most popular web application framework for Node. It is easy to work with since it ties into Node's functional paradigm. Some of the benefits of Express (for more features, see Express vs. http) include:

- Deliver static content
- Modularize business logic
- Construct an API
- Connect to various data sources (with additional plugins)
- Write less code (see Express vs. http)
- Validate data (with additional plugins)

Core http Module Server Example

Here's an example from my book *Full Stack JavaScript (Apress, 2015)* of a small RESTful API built with the core `http` module. This server has only two endpoints: POST /messages and GET /messages. The server connects to MongoDB. Just glance over the file:

```
var http = require('http')
var util = require('util')
var querystring = require('querystring')
var client = require('mongodb').MongoClient

var uri = process.env.MONGOLAB_URI || 'mongodb://@127.0.0.1:27017/me
//MONGOLAB_URI=mongodb://user:pass@server.mongohq.com:port/db_name

client.connect(uri, function(error, db) {
  if (error) return console.error(error)
  var collection = db.collection('messages')
  var app = http.createServer(function (request, response) {
    var origin = (request.headers.origin || '*')
    if (request.method == 'OPTIONS') {
      response.writeHead('204', 'No Content', {
        'Access-Control-Allow-Origin': origin,
        'Access-Control-Allow-Methods':
          'GET, POST, PUT, DELETE, OPTIONS',
        'Access-Control-Allow-Headers': 'content-type, accept',
        'Access-Control-Max-Age': 10, // In seconds
        'Content-Length': 0
      })
      response.end()
    } else if (request.method === 'GET' && request.url === '/message
      collection.find().toArray(function(error,results) {
        if (error) return console.error(error)
        var body = JSON.stringify(results)
        response.writeHead(200,{
          'Access-Control-Allow-Origin': origin,
          'Content-Type':'text/plain',
          'Content-Length':body.length
        })
        console.log('LIST OF OBJECTS: ')
        console.dir(results)
        response.end(body)
      })
    } else if (request.method === 'POST' && request.url === '/messag
      request.on('data', function(data) {
        console.log('RECEIVED DATA:')
        console.log(data.toString('utf-8'))
        collection.insert(JSON.parse(data.toString('utf-8')),
        {safe:true}, function(error, obj) {
```

```
            if (error) return console.error(error)
            console.log('OBJECT IS SAVED: ')
            console.log(JSON.stringify(obj))
            var body = JSON.stringify(obj)
            response.writeHead(200,{
              'Access-Control-Allow-Origin': origin,
              'Content-Type':'text/plain',
              'Content-Length':body.length
            })
            response.end(body)
          })
        })
      } else {
          response.end('Supported endpoints: \n/messages\n/messages')
      }
    })
    var port = process.env.PORT || 1337
    app.listen(port)
})
```

Can you see some problems with the code? There is a lot of extra work such as `JSON.parse()` and Content-Type/Content-Length headers. The code organization is not elegant due to the if/else conditions. The modularization of different routes is hard.

The bottom line is that `http` has all the functionality to build HTTP servers, but it's a very low-level implementation which will require you to code a lot of additional things. That's where Express comes to the rescue!

Express vs. http

If you write serious apps using only core Node.js modules (refer to the previous snippet for an example), you most likely find yourself reinventing the wheel by writing the same code continually for similar tasks, such as the following:

- Parsing of HTTP request bodies
- Parsing of cookies
- Managing sessions
- Organizing routes with a chain of `if` conditions based on URL paths and HTTP methods of the requests

- Determining proper response headers based on data types
- URL params and query strings parsing
- Automatic response headers
- Routes and better code organization
- Myriads of plugins (called middleware)
- Request body parsing (with a module)
- Authentication, validation, session and more! (with modules)

With Express you can develop APIs much faster!

Express.js is an amazing framework for Node.js projects, and it's used in the majority of web apps, which is why this second chapter is dedicated to getting started with this framework.