Build Killer RESTful APIs with

@djidja8

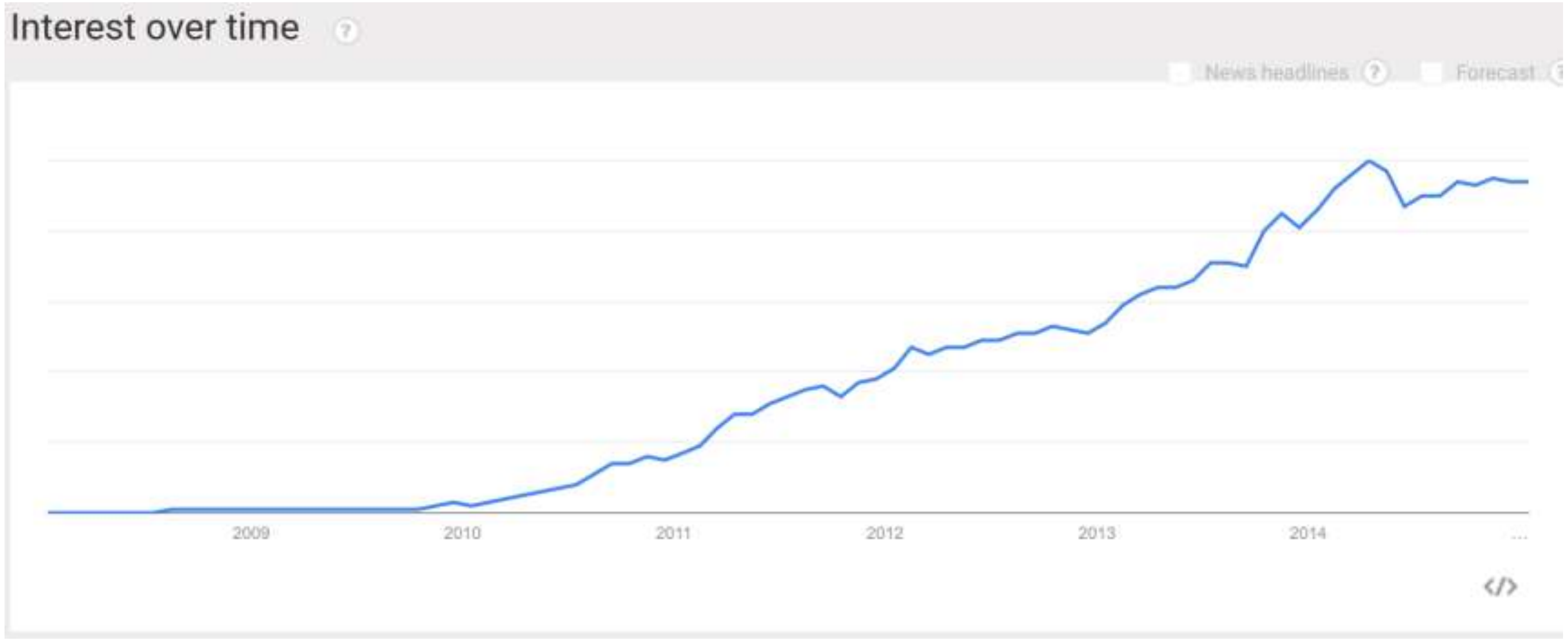
ThoughtWorks

This deck on

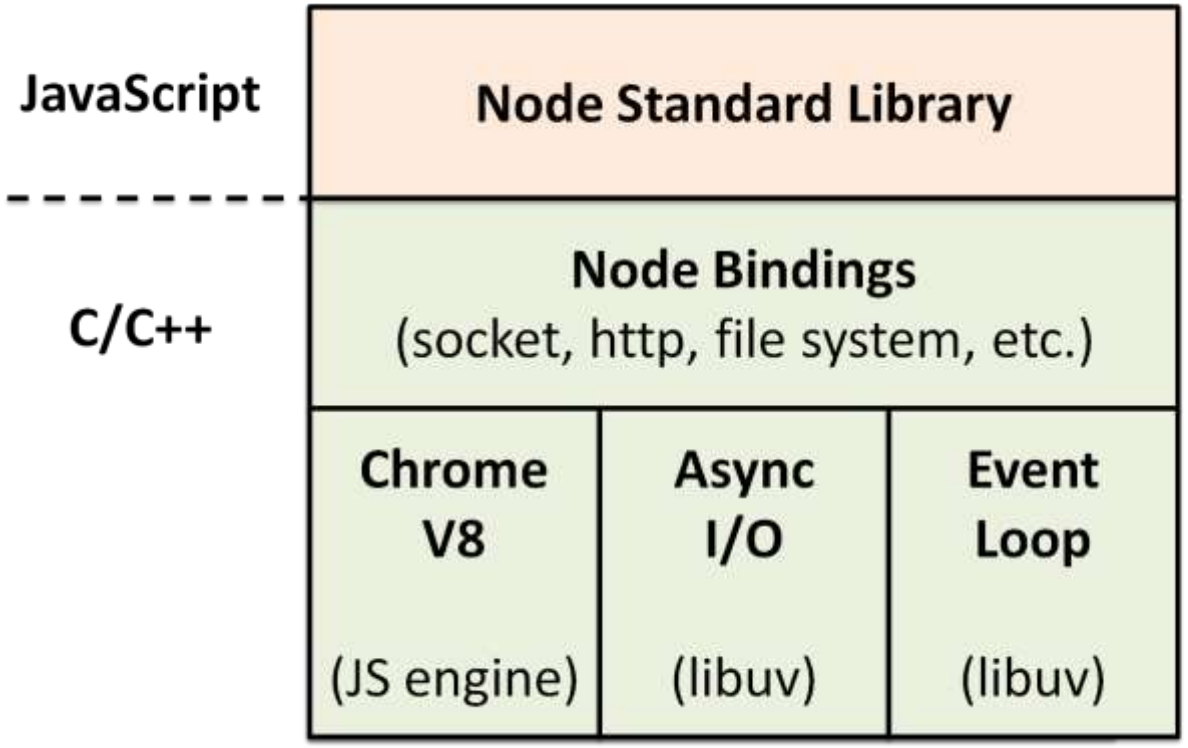
What is node JS ?

Runtime + Libraries + REPL + Package Manager

Node.js is a cross-platform runtime environment and a set of libraries for server side applications written in JavaScript



NODE.JS Overview – RUNTIME



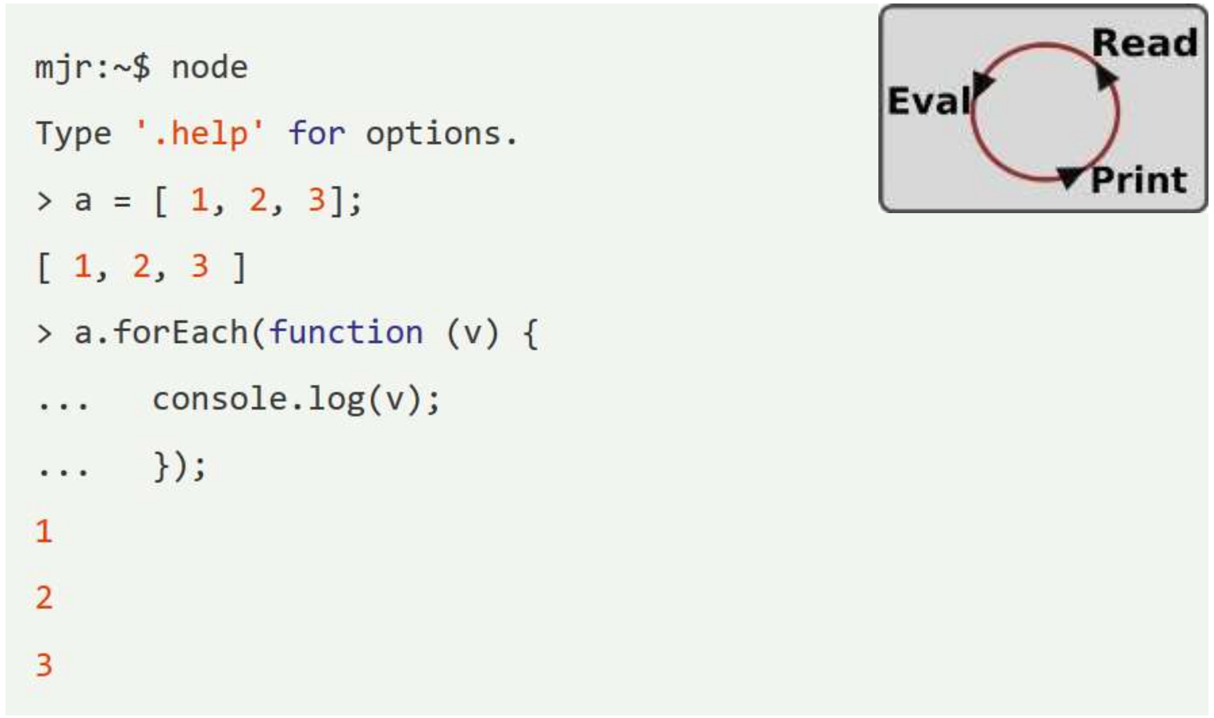
- **V8 JavaScript runtime:**
  V8 is Google's open source JavaScript engine written in C++ and used in Google's open source browser Chrome

- **libuv:**
  A cross-platform library that abstracts OS host platform system and: provides an event loop with callback based notifications for Async I/O and other activities, timers, non-blocking networking support, asynchronous file system access, child processes and more.

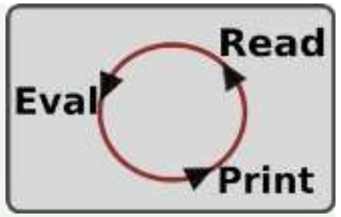
- **Non-blocking standard libraries:**
  Async + streams = highly concurrent, and it works well for IO-bound workloads (but it is not parallelism).

NODE.JS Overview – REPL

The REPL provides a way to interactively run JavaScript and see the results. It can be used for debugging, testing, or just trying things out. It is available both as a standalone program and easily embeddable in other programs.



```
mjr:~$ node
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach(function (v) {
...   console.log(v);
...   });
1
2
3
```



NODE.JS – Package manager

NPM is the official package manager for Node.js and is written entirely in JavaScript. It is bundled and installed automatically with the environment. It has the command line interface and manages packages (modules) dependencies for an application.

NPM enables you to install, publish, manage, and distribute JavaScript code easily

189,881
total packages

99,803,133
downloads in the last day

518,514,812
downloads in the last week

2,434,752,064
downloads in the last month

NODE.JS – Package manager

NPM install and uninstall commands

```
// Package will be installed globaly.
npm (un)install <name> -g

// Package will be installed localy.
npm (un)install <name>

// Package will appear in your dependencies.
npm install <name> --save

// Package will appear in your devDependencies.
npm (un)install <name> --save-dev
```

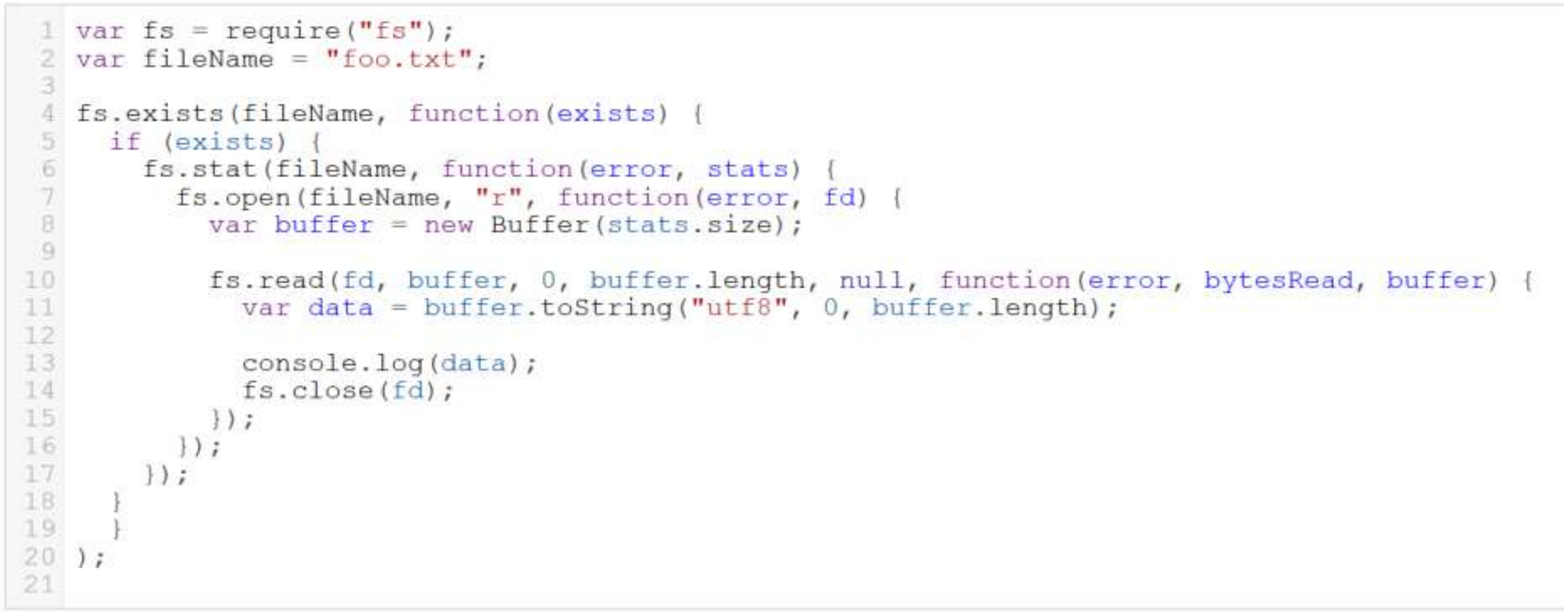
NODE.JS – Package manager

Other useful commands

```
// Sets up package.json file
npm init [--force|-f|--yes|-y]

// Runs a package's "test" script, if one was provided
npm test [-- <args>]

// Publishes a package to the registry so that it can be installed by name
npm publish [<tarball>|<folder>] [--tag <tag>] [--access <public|restricted>]

// Updates all the packages listed to the latest version
npm update [-g] [<pkg>...]

// many more CLI commands available...
```

NODE.JS Overview – Async, no streaming

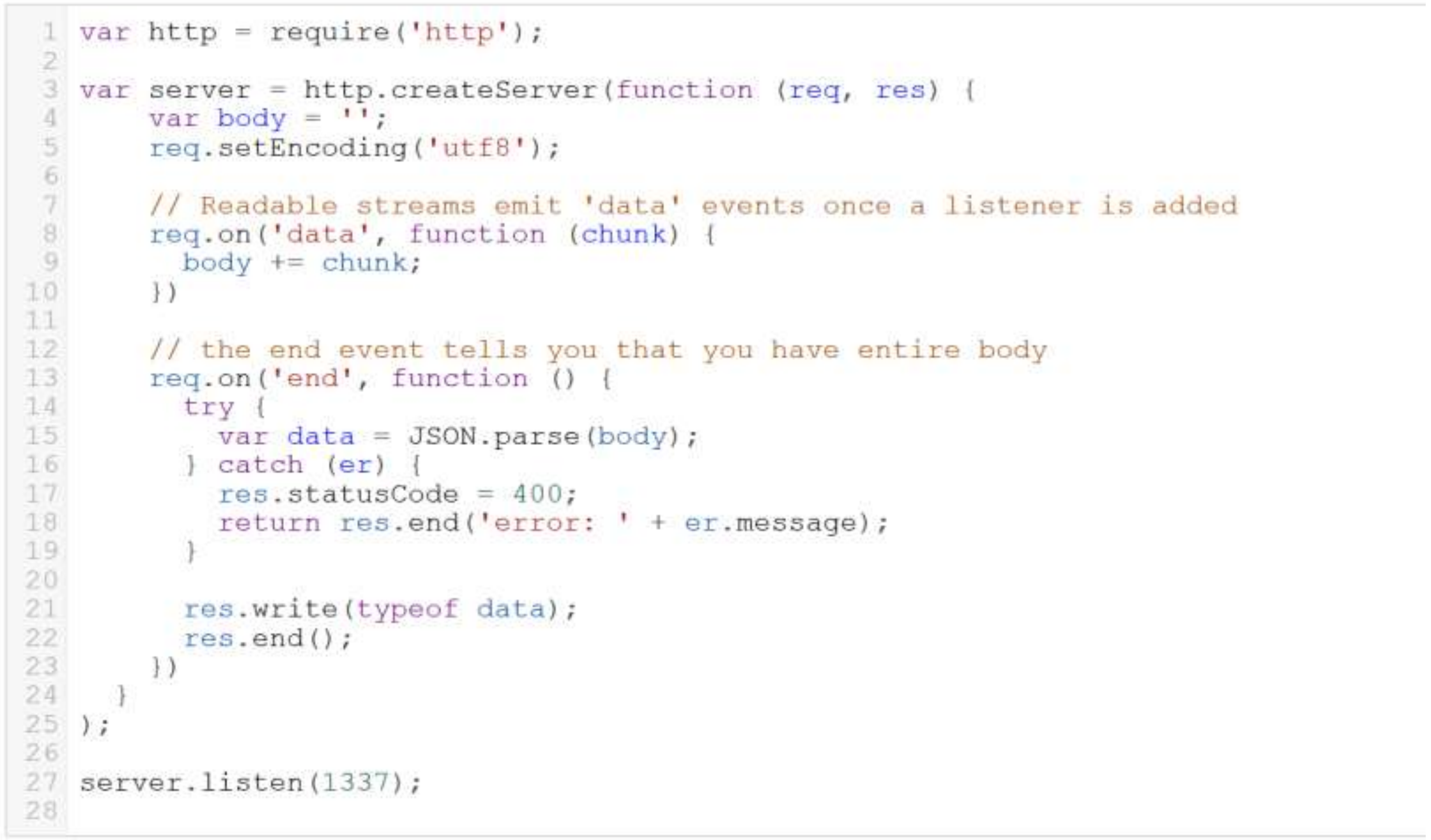
Node.js provides the file system module, fs. For the most part, fs simply provides a wrapper for the standard file operations. The following example uses the fs module to read contents of a file into memory



```
var fs = require("fs");
var fileName = "foo.txt";

fs.exists(fileName, function(exists) {
  if (exists) {
    fs.stat(fileName, function(error, stats) {
      fs.open(fileName, "r", function(error, fd) {
        var buffer = new Buffer(stats.size);

        fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
          var data = buffer.toString("utf8", 0, buffer.length);

          console.log(data);
          fs.close(fd);
        });
      });
    });
  }
  }
);

```

NODE.JS Overview – Async with streaming

The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. http module never buffers entire requests or responses



```
var http = require('http');

var server = http.createServer(function (req, res) {
    var body = '';
    req.setEncoding('utf8');

    // Readable streams emit 'data' events once a listener is added
    req.on('data', function (chunk) {
      body += chunk;
    })

    // the end event tells you that you have entire body
    req.on('end', function () {
      try {
        var data = JSON.parse(body);
      } catch (er) {
        res.statusCode = 400;
        return res.end('error: ' + er.message);
      }

      res.write(typeof data);
      res.end();
    })
  }
);

server.listen(1337);

```

NODE.JS Overview – summary

- **Single threaded / Async**

  I/O is provided by simple wrappers around standard POSIX functions. All the methods have asynchronous and synchronous forms, but Async is default.

- **Most APIs speak Streams:**

  The built-in stream module is used by the core libraries and can also be used by user-space modules, while providing a backpressure mechanism to throttle writes for slow consumers

- **Extensible via C/C++ add-ons:**

  Node.js is extensible and modules can include add-ons written in native code

- **Local scope:**

  Browser JavaScript lifts everything into its global scope, Node.js was designed to have everything being local by default. Exporting is done explicitly, and in case we need to access globals, there is a global object.

NODE.JS Overview – summary

- **Development efficiency**
  Web development in a dynamic language (JavaScript) and great package manager (NPM)

- **Performant**
  Ability to handle thousands of concurrent connections with minimal overhead on a single process.

- **Familiar**
  People already know how to use JavaScript, having used it in the browser.

- **Full stack development**
  Using JavaScript on a web server as well as the browser reduces the impedance mismatch between the two programming environments (+ JSON)

EXPRESS.JS Overview

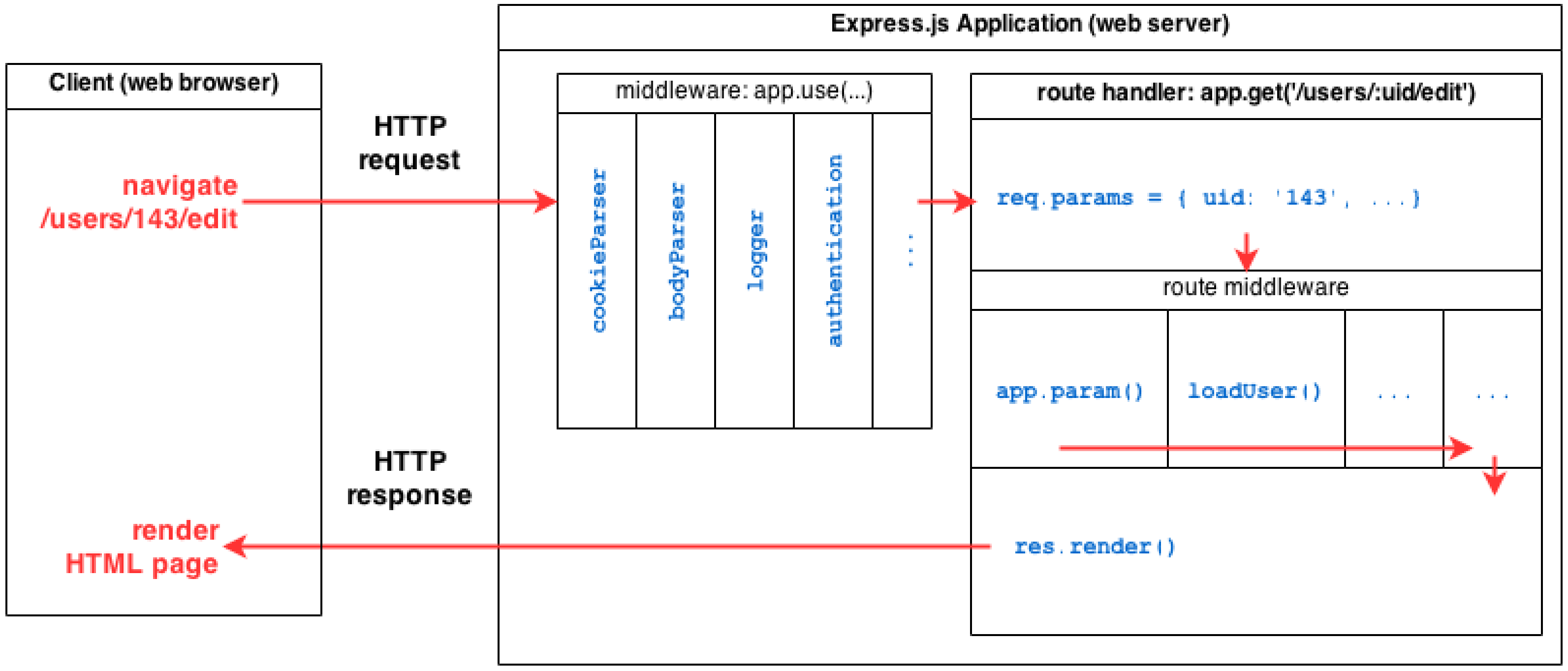
- What is Express.js?
- Hello World
- Request and Response
- Router
- Middleware

express

EXPRESS.JS Overview – What is EXPRESS.JS?

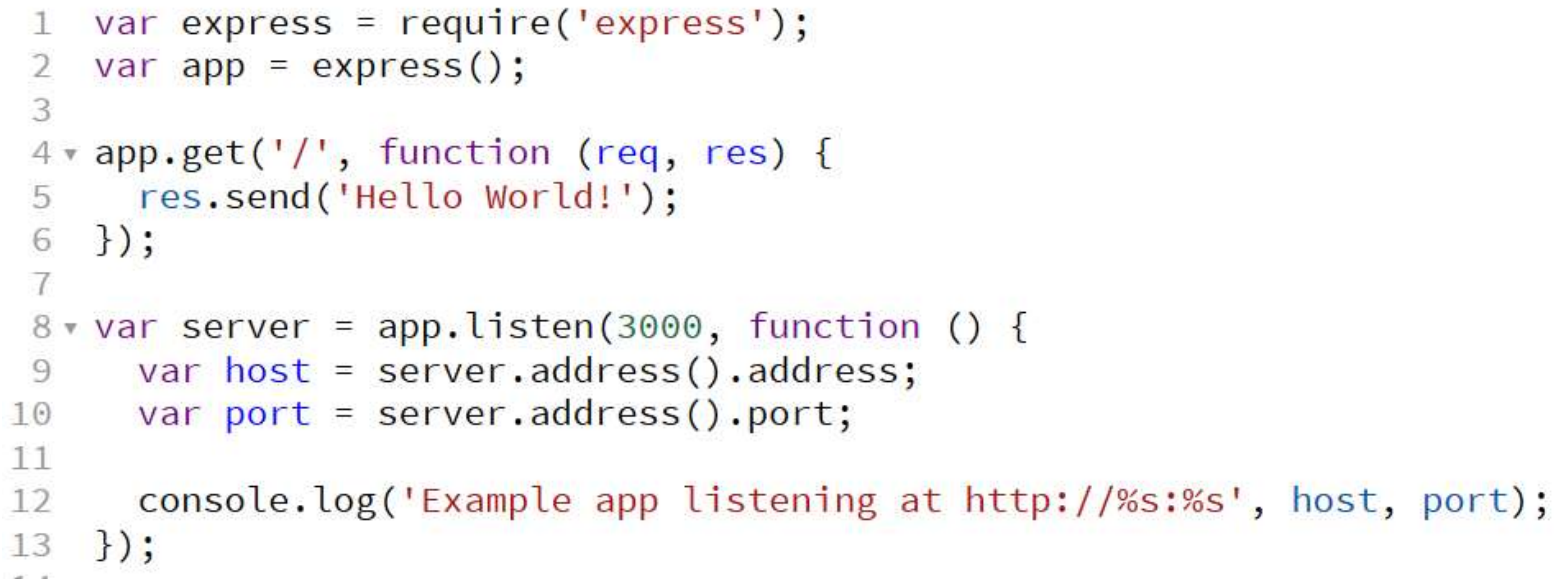
A minimal and flexible node.js web application framework, providing a robust set of features for building full web applications and/or APIs.

- Middleware
- Routing
- Templating engines
- Layouts and partial views
- Environment-based configuration



http://[adrianmejia.com/blog/2014/10/01/creating-a-restful-api-tutorial-with-nodejs-and-mongodb]/

EXPRESS.JS Overview – Hello World

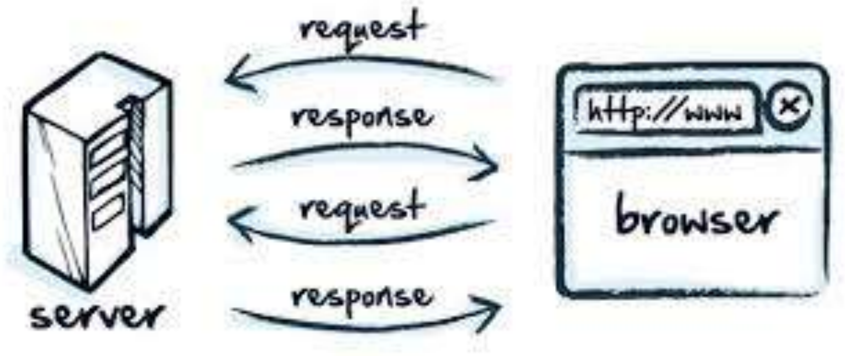


```javascript
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

EXPRESS.JS Overview – Request and Response

The Request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.

The Response object represents the HTTP response that an Express app sends back

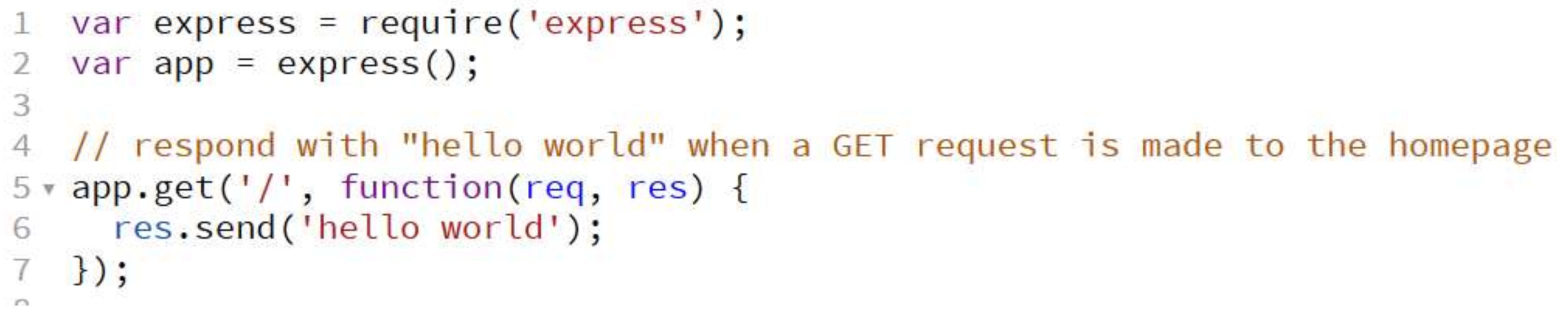


EXPRESS.JS Overview – Router

A router object is an isolated instance of middleware and routes, capable only of performing middleware and routing functions.

A router behaves like middleware itself, so you can use it as an argument to app.use() or as the argument to another router’s use() method.

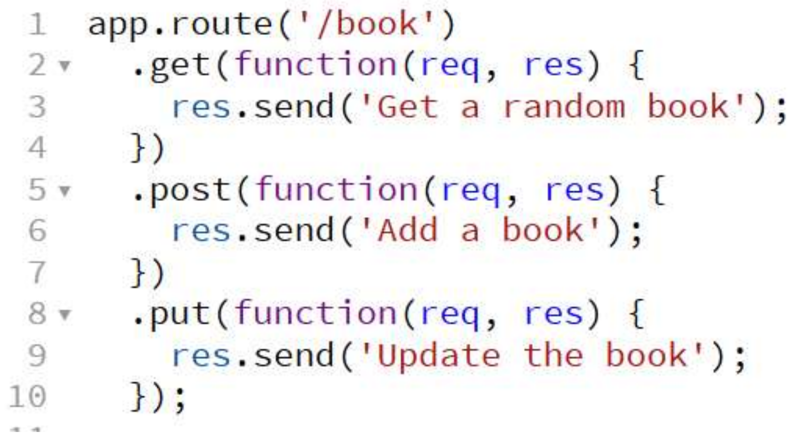
Every Express application has a built-in app router:



```
var express = require('express');
var app = express();

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function(req, res) {
  res.send('hello world');
});
```

EXPRESS.JS Overview – Router

Chainable route handlers for a route path can be created using app.route().
Since the path is specified at a single location, it helps to create modular routes and reduce redundancy and typos.



```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

EXPRESS.JS Overview – Router

The express.Router class can be used to create modular mountable route handlers. A Router instance is a complete middleware and routing system; for this reason it is often referred to as a "mini-app"

```
var db = imort("some-db")
var apples = express.Router();

apples.param('user_id', function(req, res, next, id) {
  req.user = db.get(id);
  next();
});

apples.route('/users/:user_id')
.all(function(req, res, next) {
  next(); // runs for all HTTP verbs first, a route specific middleware
})
.get(function(req, res, next) {
  res.json(req.user);
})
.put(function(req, res, next) {
  req.user.name = db.Save(req.params.name);
  res.send(req.user);
})
module.exports = apples;
```

And then

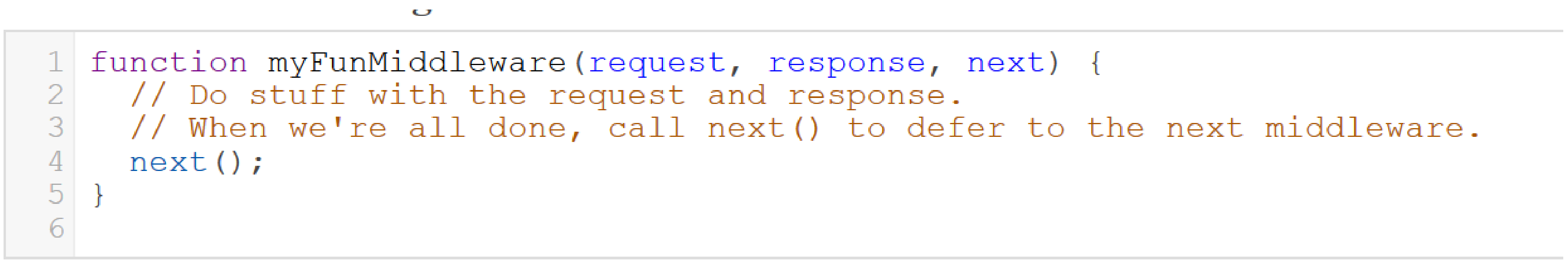
```
var apples = require('./apples');
//...
app.use('/apples', apples);
```

EXPRESS.JS Overview – Middleware

Middleware is the core concept behind Express.js request processing and routing and is composed from any number of functions that are invoked by the Express.js routing layer before final application request handler is invoked.

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Middleware function signature is simple:

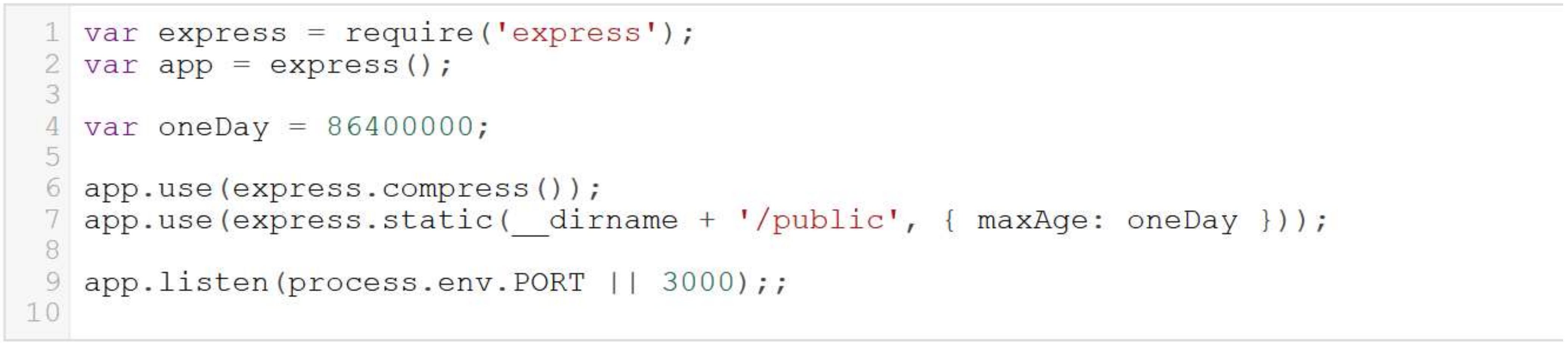


```
function myFunMiddleware(request, response, next) {
  // Do stuff with the request and response.
  // When we're all done, call next() to defer to the next middleware.
  next();
}

```

EXPRESS.JS Overview – Middleware: serving static files

Express.js has great built in capabilities to serve static content. The module is able to also gzip compress and/or cache served files, too…

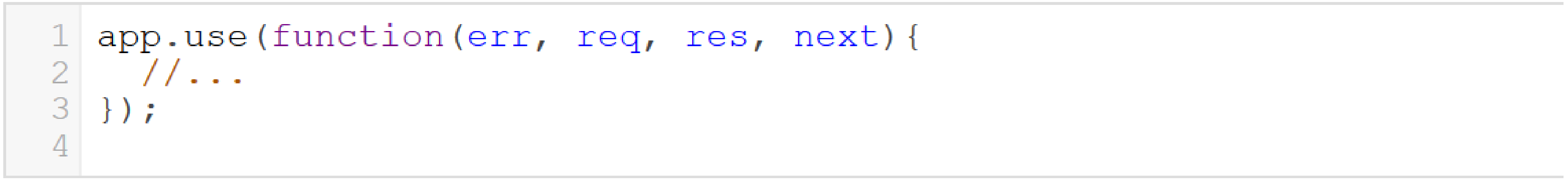
***As of 4.x, all of Express' previously included middleware are now in separate repositories. The only included middleware is: express.static(), used to server static files.**



```javascript
var express = require('express');
var app = express();

var oneDay = 86400000;

app.use(express.compress());
app.use(express.static(__dirname + '/public', { maxAge: oneDay }));

app.listen(process.env.PORT || 3000);;

```

EXPRESS.JS Overview – Middleware: error handling

Error-handling middleware are defined just like regular middleware, however must be defined with an arity of 4 signature: (error, request, response, next)



```
app.use(function(err, req, res, next){
  //...
});
```

EXPRESS.JS Overview – Middleware: error handling

error-handling middleware are typically defined very last, below any other app.use() calls...

```
app.get('/' ... function() {} ) // same as before, omitted for brevity
app.get('/catpic', function(req, res, next) {
  db.load('my-catpic', function(err, pic) {
      if (err) return next(err);

      if (!pic) {
        var notFound = new Error('no such catpic');
        notFound.status = 404;
        return next(notFound);
      }
      res.send(pic);
    });
  }
);
app.get('*', function(req, res, next) {
    var err = new Error();
    err.status = 404;
    next(err);
  }
);
app.use(function(err, req, res, next)  {  // handling 404 errors
    if(err.status !== 404) return next();
    res.send(err.message || '** no unicorns here **');
  }
);
app.use(function(err, req, res, next) {   //handle errors passed with: next(error);
    log.error(err, req);  //log the error
    res.status(500);  //send back a 500 with a generic message
    res.send('oops! something broke');
  }
);
app.listen(3000);
```

EXPRESS.JS Overview – Middleware, examples

All of Express' previously (before v4.0) included middleware are now in separate repos. Here are some example middleware modules:

- body-parser
- compression
- connect-timeout
- csurf
- errorhandler
- method-override
- morgan - previously logger
- response-time
- serve-favicon

EXPRESS.JS – but there are other options…

- **Koa**

  Koa is designed by the team behind Express, which aims to be a smaller, more expressive, and more robust foundation for web applications and APIs. Through leveraging generators Koa allows you to ditch callbacks and greatly increase error-handling

- **Restify**

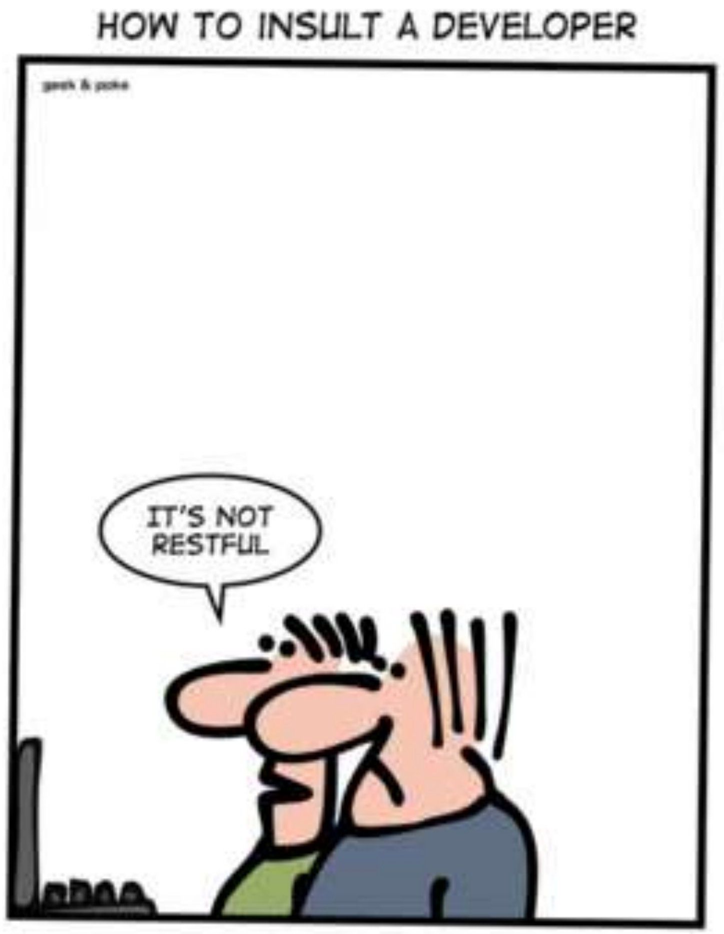
  restify is a node.js module built specifically to enable you to build correct REST web services. It intentionally borrows heavily from express as that is more or less the de facto API for writing web applications on top of node.js.

- **Hapi**

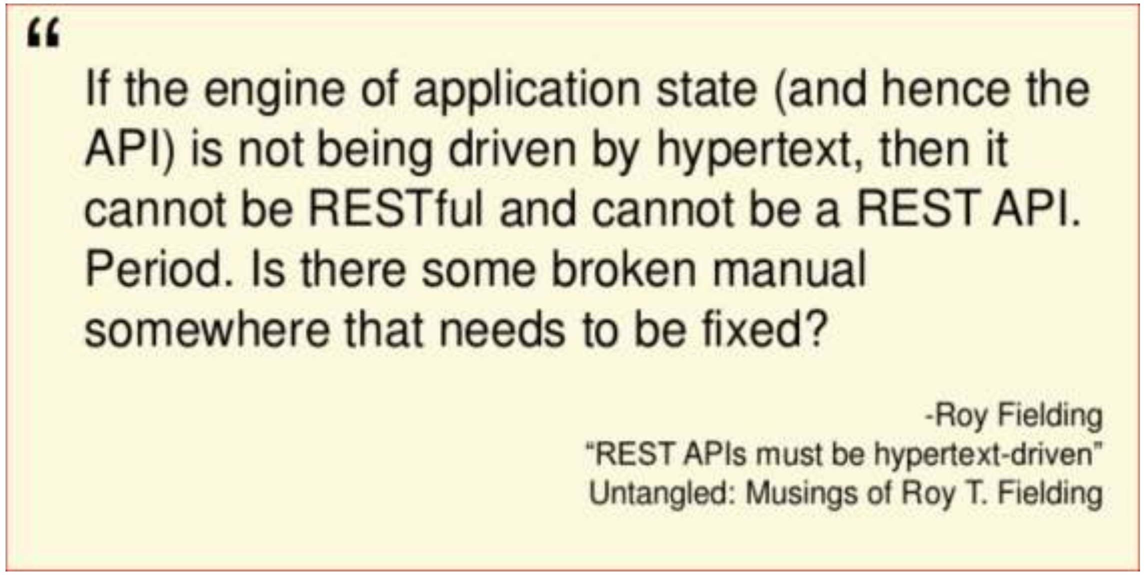
  hapi is a simple to use configuration-centric framework with built-in support for input validation, caching, authentication, and other essential facilities for building web and services applications.

Building RESTful APIs

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand (optional)
- Example



RESTful APIs – before the demo: HATEOS


“

If the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?

-Roy Fielding
“REST APIs must be hypertext-driven”
Untangled: Musings of Roy T. Fielding

roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

RESTful APIs – Uniform Interface

The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture and enables each part to evolve independently. The four guiding principles of the uniform interface are:

- **Resource-Based**
  Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client

- **Manipulation of Resources Through Representations**
  When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so

- **Self-descriptive Messages**
  Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type)

- **Hypermedia as the Engine of Application State (HATEOAS)**
  HATEOS is the key constrain that makes web browsing possible. Applicable to APIs but not widely used. It is simple to understand: each response message includes the links to next possible request message.

RESTful APIs – Stateless

As REST is an acronym for REpresentational State Transfer, server statelessness is key.

Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers

Clients handle application state, servers resource state

Application state is information about where you are in the interaction that is used during your session with an application.

Resource state is the kind of (semi-)permanent data a server stores, and lasts beyond the duration of a single session of interactions.

RESTful APIs – Cashable

The goal of caching is never having to generate the same response twice. The benefit of doing this is that we gain speed and reduce server load.

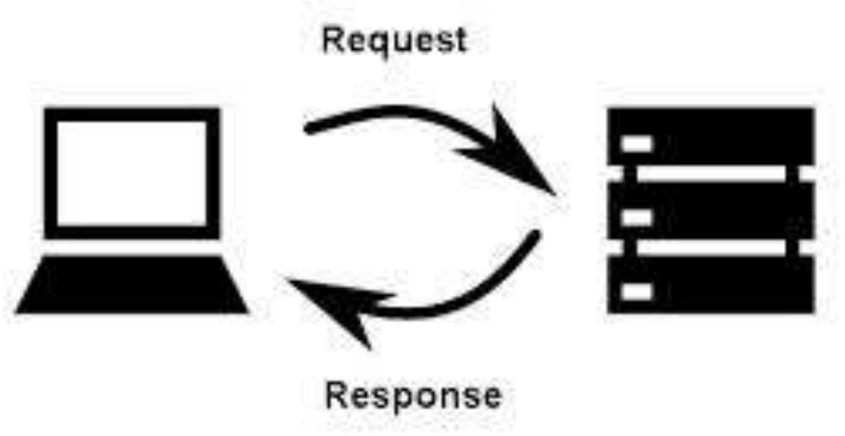
- **Expiration**
  - Expires header
  - Cache-Control header

- **Validation**
  - Last-Modified header
  - Etag header

RESTful APIs – Client-Server

The client-server constraint is based on a principle known as the separation of concerns. It simply requires the existence of a client component that sends requests and a server component that receives requests

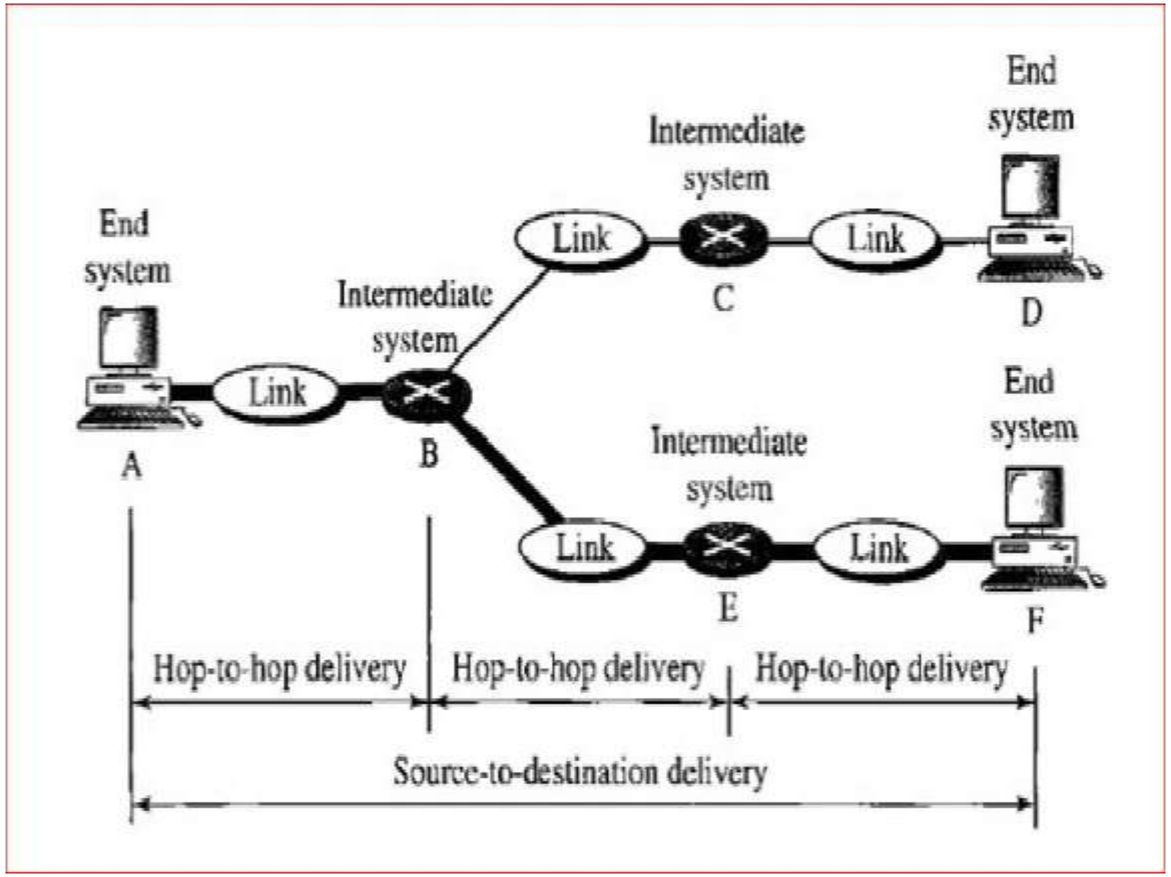
Servers and clients may also be replaced and developed independently, as long as the interface is not altered



RESTful APIs – Layered System

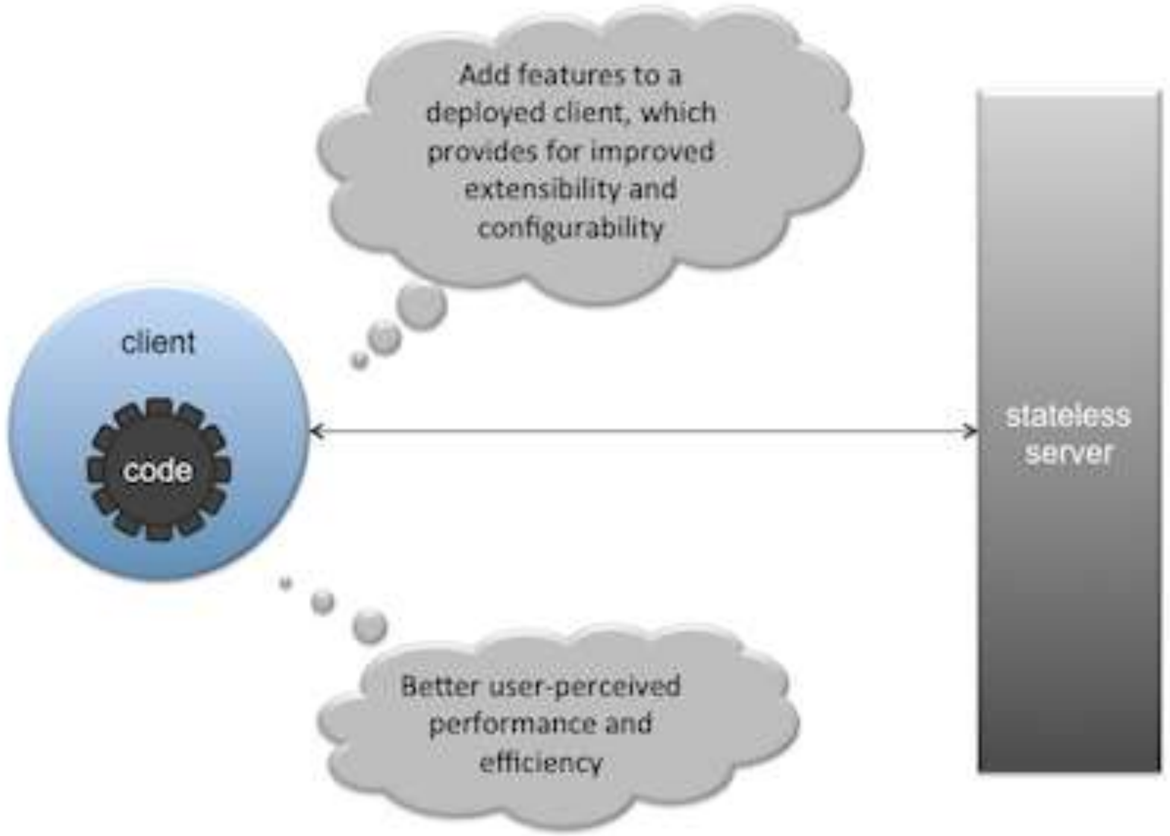
REST architecture is conceived as hierarchical layers of components, limited to communication with their immediate neighbors.

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.



RESTful APIs – Code on Demand (*optional)

The optional code-on-demand constraint allows clients to request and execute code from servers. This, in turn, allows the server to deploy new features to clients. The result is improved extensibility and configurability for servers, and improved performance and efficiency for clients


Add features to a deployed client, which provides for improved extensibility and configurability
client
code
stateless server
Better user-perceived performance and efficiency

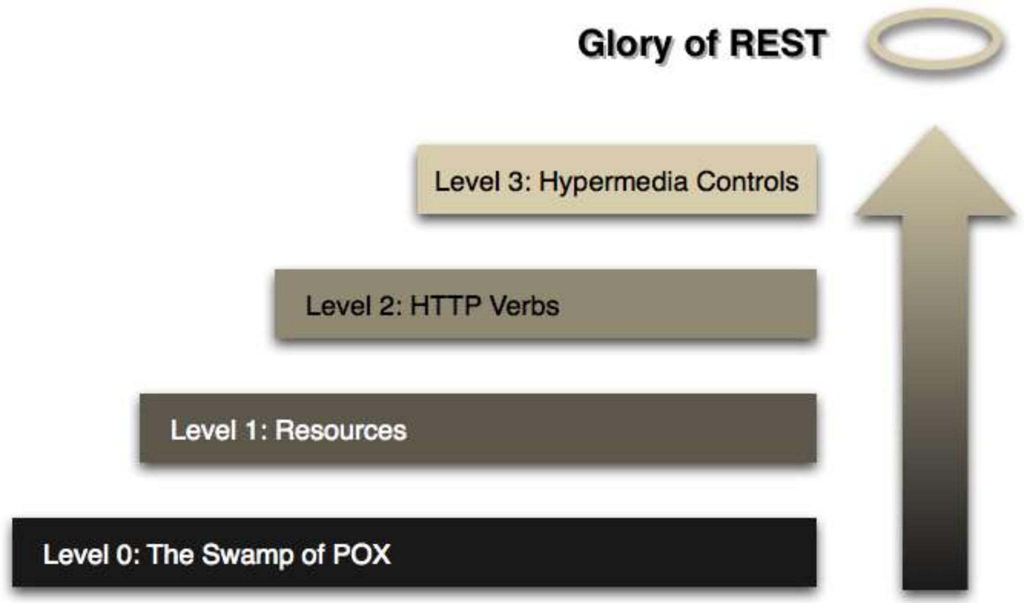
RESTful APIs – Richardson Maturity Model

Level 0: Swamp of POX
tunnels requests and responses through its transport protocol, using only one entry point (URI) and one kind of method (in HTTP, this normally is the POST method). Examples of these are SOAP and XML-RPC/JSON/RPC.

Level 1: Resources
This level uses multiple URIs, where every URI is the entry point to a specific resource. Still, this level uses only one single method like POST.

Level 2: (HTTP) verbs
This level suggests that in order to be truly RESTful, your API MUST use limited set of verbs.

Level 3: Hypermedia controls
Level 3, the highest level, uses HATEOAS to deal with discovering the possibilities of your API towards the clients.



RESTful APIs – Resource modeling

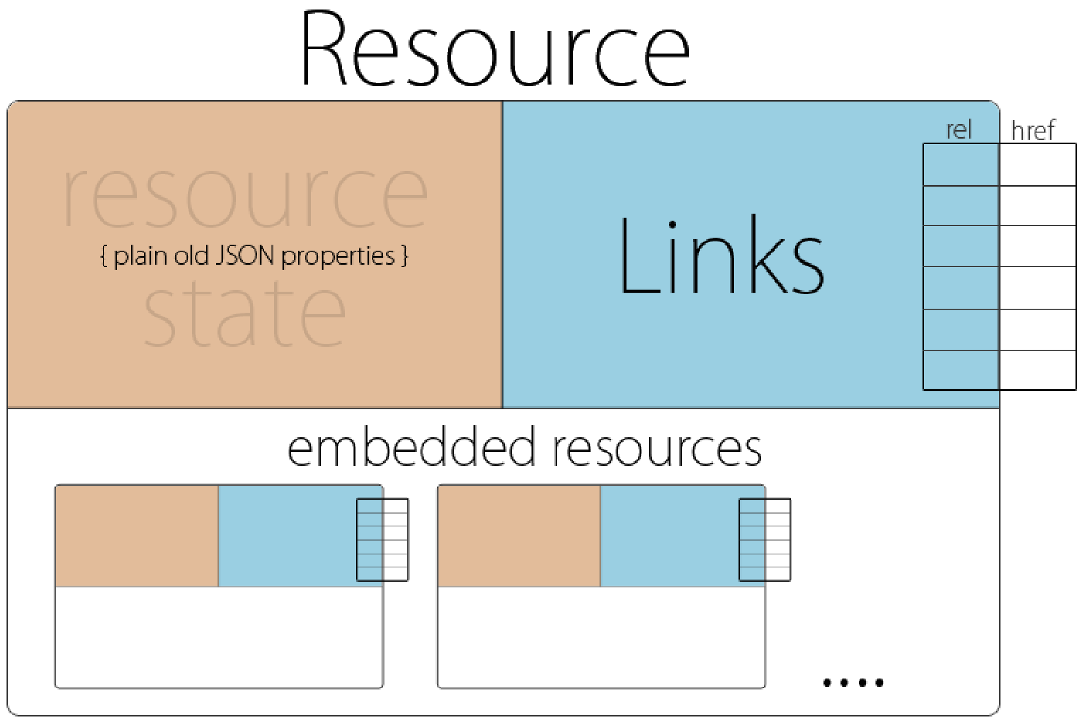
- **Granular (leads to CRUD)**
  - Not scalable
  - Low level
  - Chatty
  - Business logic moves to consumer

- **Coarse grained**
  - Focus on business entities vs activities
  - Decoupled from internal domain implementation
  - Business logic contained inside the service

***Using only GET and POST (CQRS)**

RESTful APIs – before the demo: HAL

HAL is designed for building APIs in which clients navigate around the resources by following links. Links are identified by link relations, the lifeblood of a hypermedia API: they are how you tell client developers about what resources are available and how they can be interacted with, and they are how the code they write will select which link to traverse



HAL - Specification

RESTful APIs – Show Me The Code…

Code repository: github.com/Srdjan/resto

Resources…

This presentation on GitHub: Building Killer RESTful APIs

Node.Js

- Mixu Online book
- Why Asynchronous?
- Mastering Node

Express

- Express Home Page
- JUnderstanding Express.js
- A short guide to Connect Middleware

REST APIs

- REST API Tutorial
- Restful Exploration
- HAL - Hypertext Application Language

Build Killer RESTful APIs with NODE.JS

THANK YOU!

@djidja8