node.js

Why

- V8 is fast
    - So, you get the felxibility of a dynamic, interpreted language and runtime without sacrificing so much performance
- quick to get started and elaborate development
- seems to represent some leading-edge web development techniques (e.g. socket.io for WebSockets)
- a lot of activity and development around it an associated projects
- use one language for client- and server-side development

V8

Google's V8 is part of node.js, and makes it execute very quickly. In various benchmarks, it performs very well when compared to other "weakly-typed" languages.

Benchmark samples

- In this benchmark ("spectralnorm"):
    - FORTRAN and C took the top spots at 8 seconds
    - Java 7 took 17 seconds
    - V8 took 22 seconds (0.37 minutes)
    - JRuby took 4.22 minutes
    - Ruby 2.0 took 5 minutes
    - PHP 5.5.0 took 7 minutes
    - Python 3 took 13 minutes
    - Perl took 14 minutes
- Google has done a very good job optimizing the performance of regular expression operations, so V8 does very well in this benchmark ("regex-dna"):
    - V8 took the top spot, taking 3.72 seconds elpased time to execute
    - C (gcc compiler) is in second place at 5.16 seconds
    - Java 7 took 23.17 seconds
    - Ruby 2.0 took 29.76 seconds

Javascript

OOP

"Object-orientation" is achieved in node.js through the use of ECMAScript prototypical inheritance. You can go about this (or work around it) several different ways, but I'll just present here what seems to be preferred and standard:

```
var util = require("util");

/**
 * The constructor for the `MatrixAvatar` class.
 * @constructor
 */
function MatrixAvatar(joules) {
    // A generic name for a generic person.
    // Note that `this` here refers to the instance of
    // the class being constructed.
    this.avatarName = "Coppertop";

    // How much power will this man-pod give us?
    this.joules = joules;

    // Everyone is logged into the Matrix (<evil laugh>).
    this.status = "loggedIn";
}

/**
 * Admin note: NOBODY should be calling this.
 */
MatrixAvatar.prototype.logOut = function () {
    console.log(
        "WARNING: Logging out " +
            this.avatarName +
            "! Say goodbye to " +
            this.joules +
            " Joules!"
    );
    this.status = "loggedOut";
}

/**
 * Constructor for the Agent class.
 * @param name
 * @constructor
 */
function Agent(name) {
    // Call parent constructor.
    Agent.super_.apply(
        this,
```

```
        // Argument list to super-constructor. Agents provide
        // no power.
        [0]
    );

    // Allow callers to specify the name of the agent.
    this.avatarName = name;

    // Agents never disobey (most of the time).
    this.agentState = "obedient";
}

/**
 * As much as some `Agent`s might not like it, they're
 * avatars in the Matrix as well.
 *
 * Note that this uses the node.js utility method
 * `inherits` to achieve inheritance.
 */
util.inherits(Agent, MatrixAvatar);


/**
 * Overrides `MatrixAvatar`'s functionality.
 */
Agent.prototype.logOut = function () {
    if (this.agentState == "disobedient") {
        // You are done; you must be brought back to the Source.
        Agent.super_.prototype.logOut.apply(this);
    }
    else {
        console.log("Logout denied.");
    }
}

var generic = new MatrixAvatar(9500);
generic.logOut();

var smith = new Agent("Smith");
smith.agentState = "disobedient";
smith.logOut();

// Output:
// $ node matrix.js
// WARNING: Logging out Coppertop! Say goodbye to 9500 Joules!
// WARNING: Logging out Smith! Say goodbye to 0 Joules!
```

Web Development

Preface About node.js

- node.js has its own package manager called `npm`. npm is a lot like Ruby's `gem` and `bundler`: You can easily install packages from a central repository, and specify a project's dependencies in a file called package.json. After filling out your project's package.json, you can run `npm install` like you'd run `bundle install`.
- To my understanding, node.js is not multi-threaded. Instead, it has an "event loop" that executes closures that are added to its queue.
    - So for example, HTTP requests get put on the queue, as well as loading data from the database, network, or disk.
    - Properly utilizing this event queue is the key to keeping your node.js applications responsive. Always return from HTTP request handlers as quickly as possible.
- Node affords package functionality through the use of modules. They will be discussed later in this document.

Express

Express is one of ("the"?) most popular server-side web frameworks for node.js.

From what I can tell, Express seems a lot more similar to frameworks like Sinatra or PHP's Slim Framework, rather than being like Rails or CakePHP.

[Note that you can choose which packages to use for views, sessions, etc, but that I'll choose what I liked. Show example of configuring an Express app with these modules.]

app.js

Typically you'll declare a bootstrap file called something like `app.js` that initializes your web application and sets up request handlers, model objects, etc.

Express comes with an executable that creates a skeleton application for you. This is how a slightly-customized `app.js` generated by Express looks:

```
/**
 * Module dependencies.
 */

var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.cookieParser('your secret here'));
app.use(express.session());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

All of the `app#use` calls are to initialize/attach different middlewares to the Express instance. In this case, we enable session functionality, view template functionality through the Jade library, logging, and body parsing (e.g. functionality for handling URL-encoded and JSON-encoded request bodies) middleware, among others.

Routes and Actions

In the example above, you can see a couple routes defined this way:

```
app.get('/', routes.index);
app.get('/users', user.list);
```

However, you'll notice that nothing seems to be declaring `routes.index` or `user.list`. These functions are defined in modules called `routes` and `user` that are imported (`require`d) at the top of `app.js`.

Modules

node.js allows you to organize your code into modules that can be `require()`d by other modules for reuse.

The `routes` module mentioned above is defined in the file at `routes/index.js`, which looks like this:

```
/*
 * GET home page.
 */

exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

In node.js, you use the `exports` "magic" object to export functionality to callers. In this case, the `index` function is exported, and used by Express to handle an HTTP request. Express hands off request and response objects to the `index` method here through the `req` and `res` parameters. The response in this case is rendered as a Jade template with a local `title` variable given to it.

The `user` module is defined in the file at `routes/user.js` and looks like this:

```
/*
 * GET users listing.
 */

exports.list = function(req, res){
  res.send("respond with a resource");
};
```

Using the Event Queue

You'll notice that Express expects a Function reference for each HTTP handler. This is how node.js development works: Most (all?) things that involve responding to or listening for events, loading data, or performing long-running computations are executed in closures handed off to other functions. This allows calls to return as quickly as possible, which lets node.js (and Express) respond to other clients in a timely fashion.

Closure functions are added to the event queue and executed by node.js as it loops over all of the events. So when someone issues a request to your web application, typically you'll load some data from the database, but instead of waiting for the data to return you hand off a closure that waits for the data to arrive while Express is freed up to start responding to another request. When the data arrives, the closure finishes handling the response to the HTTP request. This looks like this:

```
app.post("/login", function(req, res, next) {
    // User.findOne "detaches" from the current flow of execution,
    // and when data arrives, it calls the provided callback closure.
    User.findOne({
        email: req.body.user.email,
        password: req.body.user.password
    }, function(err, doc) {
        // This is a typical pattern in node.js/Express development.
        // If an error occurs, the next handler is called with the
        // indication that an error has occurred. This is the only
        // way to let Express know that an error has occurred so that
        // it can report it to the client (or the logs, or whatever).
        if (err) return next(err);

        if (!doc) return res.send("<p>YOU DON'T EXIST.<p>");

        // Now that we have the data, the response can be delivered.
        req.session.loggedIn = doc._id.toString();
        res.redirect("/");
    });
});
```

Socket.IO

Socket.IO is a library that allows you to communicate events to the client browser - and receive events back from the browser - via a WebSocket (degrading to long-polling when necessary). Because node.js is event-centric, and because client and server code are both written in Javascript, this method of interaction is very natural and doesn't require a lot in the way of mental "context switching." It's also very cool and fast (immediate, actually, since WebSockets

are used).

Here's an example of how this looks:

Server Code

```
var io = require("socket.io");
// ...
io.listen(app);
io.sockets.on("connection", function(socket) {
    socket.emit("dataTime", {message: "fart"});
    socket.on("dataAcknowledged", function(otherData) {
        console.log(otherData);
    });
});
```

Client (Browser) Code

```
var socket = io.connect();
socket.on("dataTime", function(data) {
    if (data.message == "fart") {
        socket.emit("dataAcknowledged", {message: "fart yourself"});
    }
});
```

Mongoose

MongoDB is a very natural way to handle persistent data in node.js, since Mongo documents translate very easily into JSON. Mongoose seems to be the most popular interface to a MongoDB instance.

Sample usage:

```
var mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/my_crap");

var Person = mongoose.model("Person", {
    id: ObjectId,
    name: String,
    email: String,
    age: Number,
    address: {
        street1: String,
        street2: String,
        city: String,
        zip: Number
    }
});
```

Most likely, you would define your models in modules and export them (`module.exports = Person;` in the above example.)

Querying looks like this:

```
Person.find({email: "fudge@example.com"})
    .where("name", "Fudge Dude")
    .limit(1)
    .run(function(err, post) {
        // Do some stuff and then return to the client.
        // ...
    });
```

Other Notes

- Heroku supports node.js web applications, which makes development and public testing easy.