Web Application Development

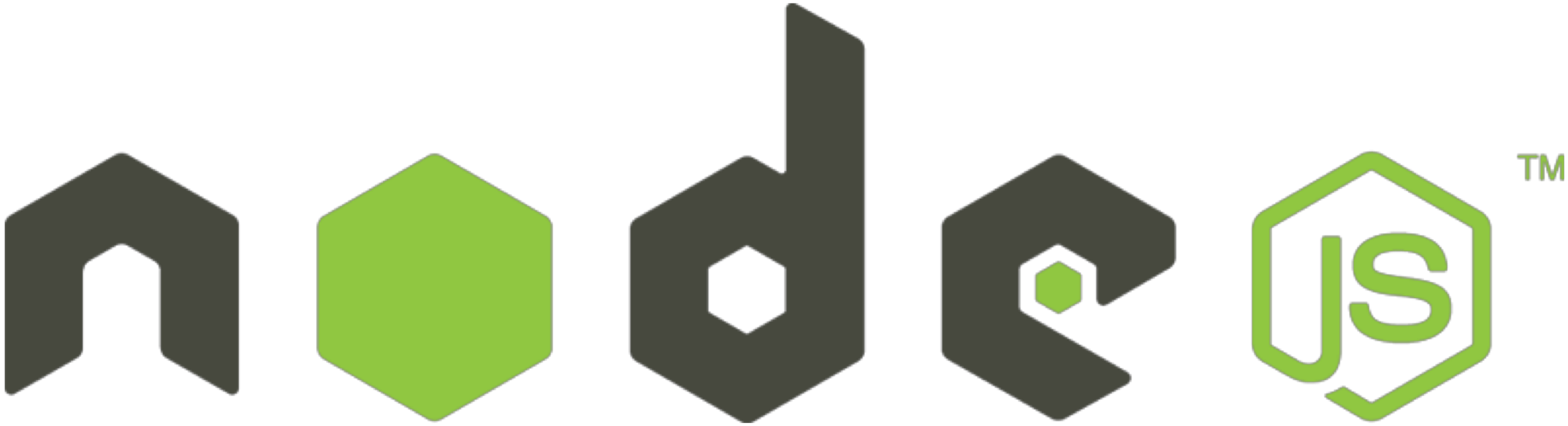
Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology

http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE



PART 2

EXPRESS AND RESTFUL API'S

Outline

1. **Introduction** – What Node is all about
2. **Node Execution Model** – Nodes Event-Driven, Non-Blocking I/O model
3. **Asynchrony in Node** – Events, Callbacks, Promises & Async/Awaits
4. **Node Modules** – The Building Blocks of Node
5. **Express** – A Framework for Node
6. **REST** – The Architectural Style of the Web
7. **API Design** – Exposing Application Functionality
8. **REST in Express** – Leveraging URLs, URI's and HTTP

Outline

1. **Introduction** – What Node is all about
2. **Node Execution Model** – Nodes Event-Driven, Non-Blocking I/O model
3. **Asynchrony in Node** – Events, Callbacks, Promises & Async/Awaits
4. **Node Modules** – The Building Blocks of Node
5. Express – A Framework for Node
6. REST – The Architectural Style of the Web
7. API Design – Exposing Application Functionality
8. REST in Express – Leveraging URLs, URI's and HTTP

Express

A FRAMEWORK FOR NODE

RECAP - Using NodeJS

NodeJS is *just* a JavaScript interpreter.

It comes with a package manager called npm

- Install packages like this: **`npm install <package_name>`**
- This will install it in the current folder.
- To install globally, do **`npm install -g <package_name>`**

To use Node as a webserver, you must write an application that responds to web requests.

Node has a library (HTTP) for doing this, but it's much easier to use a framework, like *Express*

What is Express?

It's a minimalist, "unopinionated" and extensible web framework built for the Node.js ecosystem.

It enables you to create a web server that is more readable, flexible, and maintainable than you would be able to create using only the Node HTTP library, which can get verbose and complicated for even the most basic web servers.

Express makes creating a web server much easier - As a matter of fact, it's difficult to even find examples of real-world web applications that use only the Node HTTP library because you'd have to be sadistic to do it.

```
npm init --yes
npm install --save express
```

Express Basics

The classic “Hello World” example (to run [**`node server.js`**])

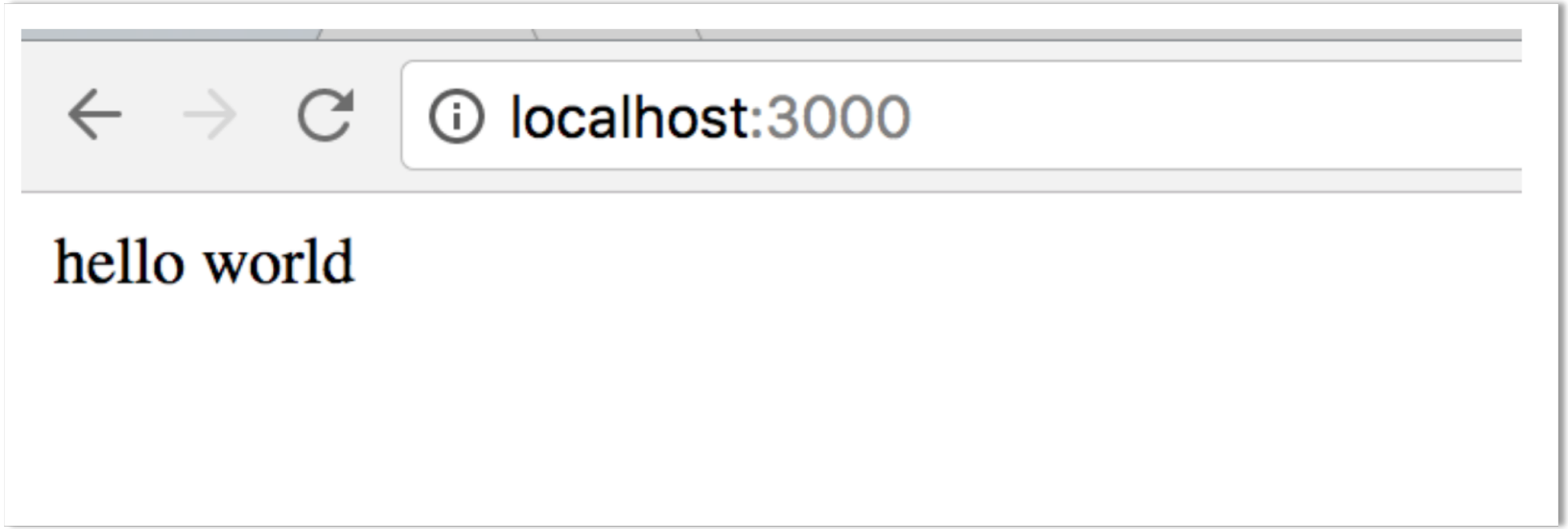
```
const express = require('express');
const app = express();

app.get('/', (request, response) => {
  response.send('hello world');
});

app.listen(3000, () => {
  console.log('Express intro running on localhost:3000');
});
```

Express Basics

And if you head over to http://localhost:3000, you get :



Express Basics

Let's have a look at the first two lines :

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

The first line **`requires`** the Express module that was installed via **`NPM`**, and the second line sets up our Express application. With this application (**`app`**), you can configure and add functionality to your server.

Express Basics

```
app.listen(3000, () => {
  console.log('Express Intro running on localhost:3000');
});
```

The **`app.listen()`** function tells the server to start listening for connections on a particular port, in this case port 3000.

This is why we went to *localhost:3000* to look at our hello world example.

When the server is ready to listen for connections, the callback is called and logs *'Express Intro running on localhost:3000'* in the terminal.

Express Basics

```
app.get('/', (request, response) => {
  response.send('hello world');
});
```

This part, although small, is fairly dense because Express is able to give a lot of functionality with very little code. **`app.get()`** creates a route handler to listen for **`GET`** requests from a client. The first argument in this function is the *route path*. In this case, we're listening for **`GET`** requests on **`localhost:3000/`**. If we wanted to listen for a **`POST`** request, then we would use **`app.post()`** for a **`PUT`** request, **`app.put()`**, and so on for any other **`HTTP`** method.

Express Basics

```
app.get('/', (request, response) => {
  response.send('hello world');
});
```

The second argument is a callback function that takes a ***request*** object and a ***response*** object. The ***request*** object contains information about the request that came from the client (request headers, query parameters, request body, etc.). The ***response*** object contains information that we want to send as a response back to the client. The ***response*** object also has functions that enable us to actually send a response.

Inside ***app.get()***, the ***response.send('hello world');*** function sends a response with content in the body of the response. In this case, the body contains the plain text hello world. Now we know how to set up a route!

Express Highlights

Express is just a package for Node (but a very useful one!)

- Create a new web application with **`var app = express();`**
- Respond to requests like **`app.get('/user', function(req, res){`**
- Look at parameters through the **`req`** object
  - **`req.params`** for query parameters
  - **`req.body`** for post fields
  - **`req.files`** for files
- Send responses through the `res` object
  - **`res.send`**`("Hi mom!")`
- Start the application with
  - **`app.listen`**`(<port>)`

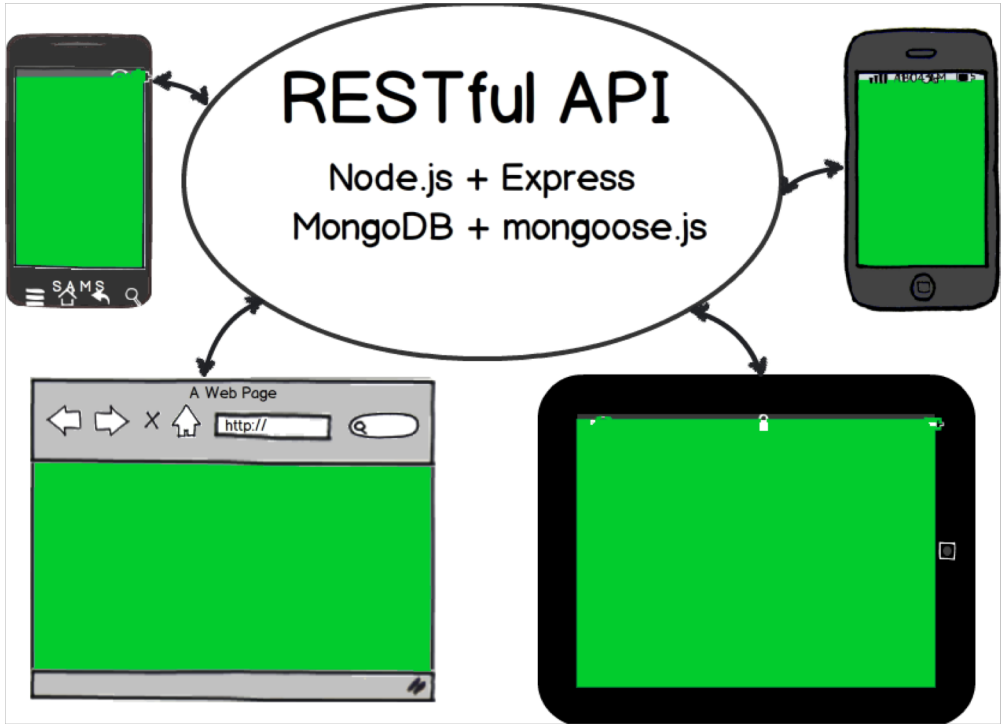
Express & Beyond

- How to serve Static Assets
- How to serve JSON
- Middleware

We'll cover this (and more) in detail in later Sections

API reference: http://expressjs.com/api.html

REST



THE ARCHITECTURAL STYLE OF THE WEB

REST (**RE**presentational **S**tate **T**ransfer)

The architectural style of the web

So what the %&*@#$ does that even mean?? ☺

REST is a set of design criteria and not the physical structure (architecture) of the system

REST is not tied to the 'Web' i.e. doesn't depend on the mechanics of HTTP

However, 'Web' applications are the most prevalent – hence RESTful architectures run off of it

Coined by **Roy Fielding** in his PhD thesis

Understanding REST

Based on content from chapter #3 of:

RESTful Web Services (O'Reilly)

-Richardson & Ruby

Understanding REST – *Resources*

Anything that's important enough to be referenced as a thing in itself

Something that can be stored on a computer and represented as a stream of bits:

- A document (e.g. information about WIT)
- A Row in a DB (e.g. a 'Donation')
- Output of executing an algorithm (e.g. 100th Prime number or Google Search ☺)

URIs and Resources

URI is an ‘address’ of a resource

A resource must have *at least one* URI

No URI → Not a resource (i.e. it’s really not on the web, so to speak ☺)

URIs should be descriptive (human parseable) and have structure. For Example:

- http://www.ex.com/software/releases/latest.tar.gz
- http://www.ex.com/map/roads/USA/CA/17_mile_drive
- http://www.ex.com/search/cs578
- http://www.ex.com/sales/2012/Q1
- http://www.ex.com/relationships/Alice;Bob

URIs and Resources (Cont'd)

Not so good URIs (everything as query parameters):

- http://www.ex.com?**software**=VisualParadigm&**release**=latest &**filetype**=tar&**method**=fetch
- http://www.ex.com?**sessionId**=12345678908765432123456787654323456786543234567 8876543&**itemId**=9AXFE5&**method**=addToCart

URIs need not have structure/predictability but are valuable (and easier) for the (human) clients to navigate through the application

May have multiple URIs to refer to same resource – convenient but confusing

Each URI must refer to a unique resource – although they may point to the 'same one' at some point in time (Ex.: …/latest.tar.gz and …/v1.5.6.tar.gz)

Understanding REST - *Addressability*

An application is addressable if it exposes *interesting aspects* of its data set as resources

An addressable application exposes a URI for every piece of information it might conceivably serve (usually infinitely many ☺)

Most important from end-user perspective

Addressability allows one to *bookmark URIs* or embed them in presentations/books etc. Ex.:

- google.com/search?q=CS577+USC
- Instead of
  - Go to www.google.com
  - Enter 'CS577 USC' (without quotes in search box)
  - Click 'Search' or hit the 'Enter key'

REST Principle #1

THE KEY ABSTRACTION OF INFORMATION IS A RESOURCE, NAMED BY A URI. ANY INFORMATION THAT CAN BE NAMED CAN BE A RESOURCE

Understanding REST - *Statelessness*

Every HTTP request happens in complete isolation

- Server NEVER relies on information from prior requests
- There is no specific 'ordering' of client requests (i.e. page 2 may be requested before page 1)
- If the server restarts a client can resend the request and continue from where it left off

Possible states of a server are also resources and should be given their own URIs!

REST Principle #2

ALL INTERACTIONS ARE CONTEXT-FREE: EACH INTERACTION CONTAINS ALL OF THE INFORMATION NECESSARY TO UNDERSTAND THE REQUEST, INDEPENDENT OF ANY OTHER REQUESTS THAT MAY HAVE PRECEDED IT.

Understanding REST - *Representations*

Resources are NOT data – they are an abstraction of how the information/data is split up for presentation/consumption

The web server must respond to a request by sending a series of bytes in a specific file format, in a specific language – i.e. a *representation* of the resource

- Formats: **XML/JSON**, HTML, PDF, PPT, DOCX...
- Languages: English, Spanish, Hindi, Portuguese…

Which Representation to Request?

Style 1: Distinct URI for each representation:

- ex.com/press-release/2012-11.en (English)
- ex.com/press-release/2012.11.fr (French)
- …and so on

Style 2: Content Negotiation

- Expose Platonic form URI:
  - ex.com/press-release/2012-11
- Client sets specific HTTP request headers to signal what representations it's willing to accept
  - **Accept:** Acceptable file formats
  - **Accept-Language:** Preferred language

REST Principle #3

THE REPRESENTATION OF A RESOURCE IS A SEQUENCE OF BYTES, PLUS REPRESENTATION METADATA TO DESCRIBE THOSE BYTES. THE PARTICULAR FORM OF THE REPRESENTATION CAN BE NEGOTIATED BETWEEN REST COMPONENTS

Understanding REST – *Uniform Interface*

HTTP Provides 4 basic methods for CRUD (create, read, update, delete) operations:

- **GET**: Retrieve representation of resource
- **PUT**: Update/modify existing resource (or create a new resource)
- **POST**: Create a new resource
- **DELETE**: Delete an existing resource

Another 2 less commonly used methods:

- **HEAD**: Fetch meta-data of representation only (i.e. a metadata representation)
- **OPTIONS**: Check which HTTP methods a particular resource supports

Be clear of the difference between PUT vs. POST

HTTP Request/Response

Method	Request Entity-Body/Representation	Response Entity-Body/Representation
GET	(Usually) Empty Representation/entity-body sent by client	Server returns representation of resource in HTTP Response
DELETE	(Usually) Empty Representation/entity-body sent by client	Server may return entity-body with status message or nothing at all
PUT	Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all
POST	Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all

(entity-body == http term for content)

REST Principle #4

COMPONENTS PERFORM ONLY A SMALL SET OF WELL-DEFINED METHODS ON A RESOURCE PRODUCING A REPRESENTATION TO CAPTURE THE CURRENT OR INTENDED STATE OF THAT RESOURCE AND TRANSFER THAT REPRESENTATION BETWEEN COMPONENTS.

THESE METHODS ARE GLOBAL TO THE SPECIFIC ARCHITECTURAL INSTANTIATION OF REST; FOR INSTANCE, ALL RESOURCES EXPOSED VIA HTTP ARE EXPECTED TO SUPPORT EACH OPERATION IDENTICALLY

Understanding REST – *Safety & Idempotence*

Idempotence: Executing the same operation multiple times is the same as executing it once

- Deleting an already DELETE-ed resource is still deleted
- Updating an already updated resource with PUT has no effect

Safety: The request doesn't change server state i.e. no side effects → no changing state of resource

- Making 10 requests is same as making one or none at all

When *correctly used* GET and HEAD requests are ***safe*** and GET, HEAD, PUT, DELETE are ***idempotent***. POST is neither safe nor idempotent

Safety and Idempotence

Why do they matter?

Lets a client make reliable HTTP requests over an unreliable connection

If no response then just reissue the request

Some common mistakes/misuses:

- GET https://some.api.com/item/delete
- (Overloaded)POST https://some.api.com/item
  - Entity-body: Method=fetch
  - Or setting different query parameters
  - Basically using POST for everything ☺

REST Principle #5

IDEMPOTENT OPERATIONS AND REPRESENTATION METADATA ARE ENCOURAGED IN SUPPORT OF CACHING AND REPRESENTATION REUSE.

Steps to a RESTful Architecture

Read the Requirements and turn them into resources ☺

1. Figure out the data set
2. Split the data set into resources

   For each kind of resource:

3. Name resources with URIs
4. Expose a subset of uniform interface
5. Design representation(s) accepted from client (Form-data, JSON, XML to be sent to server)
6. Design representation(s) served to client (file-format, language and/or (which) status message to be sent)
7. Consider typical course of events: sunny-day scenarios
8. Consider alternative/error conditions: rainy-day scenarios

A Bit on HTTP Status/Response Codes

HTTP is built in with a set of status codes for various types of scenarios:

- 2xx Success (*200 OK*, *201 Created*…)
- 3xx Redirection (*303 See other*)
- 4xx Client error (*404 Not Found*)
- 5xx Server error (*500 Internal Server Errror*)

Leverage existing status codes to handle sunny/rainy-day scenarios in your application!

Some General Points to Note

Authentication/Authorization data should be sent with every request

Sessions are NOT RESTful (i.e. sessions = state)

Cookies, if used appropriately (for storing client state) are RESTful

100% RESTful architecture is not practical and not valuable either

You need to be unRESTful at times (Eg.: Login/Logout)

- These are actions and not a resource per se
- Usually POST requests sent to some URI for logging in/out
- Advantages: Gives login page, provides ability of “Forgot your password” type functionalities etc.
- Benefits of UnRESTful-ness outweigh adherence to style

Some server frameworks only support GET/POST forcing one to overload POST requests for PUT/DELETE

Benefits of RESTful Design

Simpler and intuitive design – easier navigability

Server doesn't have to worry about client timeout

Clients can easily survive a server restart (state controlled by client instead of server)

Easy distribution – since requests are independent – handled by different servers

Scalability: As simple as connecting more servers ☺

Stateless applications are easier to cache – applications can decide which response to cache without worrying about 'state' of a previous request

Bookmark-able URIs/Application States

HTTP is stateless by default – developing applications with it gets above benefits (unless you wish to break them on purpose ☺)

API Design

EXPOSING APPLICATION FUNCTIONALITY

API Design

APIs expose functionality of an application or service

Designer must:

- Understanding enough of the important details of the application for which an API is to be created,
- Model the functionality in an API that addresses all use cases that come up in the real world, following the RESTful principles as closely as possible.

Nouns are good, verbs are bad

Keep your base URL simple and intuitive

2 base URLs per resource

The first URL is for a collection; the second is for a specific element in the collection.

Example

- /contacts
- /contacts/1234

Keep verbs out of your URLs

REST in Express

LEVERAGING URL'S, URI'S AND HTTP

RESTful Frameworks

Almost all frameworks allow you to:

1. Specify URI Patterns for routing HTTP requests
2. Set allowable HTTP Methods on resources
3. Return various different representations (JSON, XML, HTML most popular)
4. Support content negotiation
5. Implement/follow the studied REST principles

Express is just ONE of the many frameworks…

List of REST Frameworks

Rails Framework for Ruby (Ruby on Rails)

Django (Python)

Jersey /JAX-RS (Java)

Restlet (Java)

Sinatra (Ruby)

Express.js (JavaScript/Node.js)

…and many others: View complete list at:
http://code.google.com/p/implementing-rest/wiki/RESTFrameworks

REST in Express

We can easily implement REST APIS using express routing functionality

Functionality usually implemented in api routing script

```
// Our Custom Donation Web App Routes
app.get('/donations', donations.findAll);
app.get('/donations/votes', donations.findTotalVotes);
app.get('/donations/:id', donations.findOne);
app.post('/donations',donations.addDonation);
app.put('/donations/:id/vote', donations.incrementUpvotes);
app.delete('/donations/:id', donations.deleteDonation);
```

Donationweb

BEHIND THE SCENES http://donationweb-server.herokuapp.com

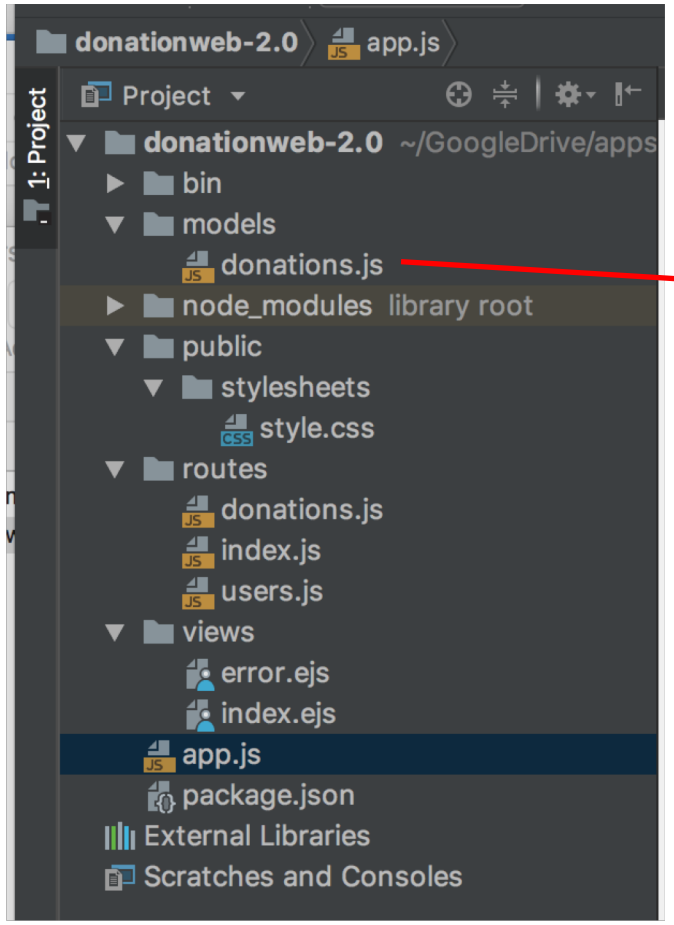
Donation: Resource, URIs & Methods

Resource	URI (structure)	HTTP Request
List of Donations	/donations	GET
Get a Single Donation	/donations/{id}	GET
Upvote a Donation	/donations/{id}/vote	PUT
Delete a Donation	/donations/{id}	DELETE
Update a Donation	/donations/{id}	PUT
Add a Donation	/donations/{id}	POST
Total of Donation Votes	/donations/votes	GET

{…} = variable value; changeable by user/application to refer to specific resource

We'll look at this Use Case as an example…

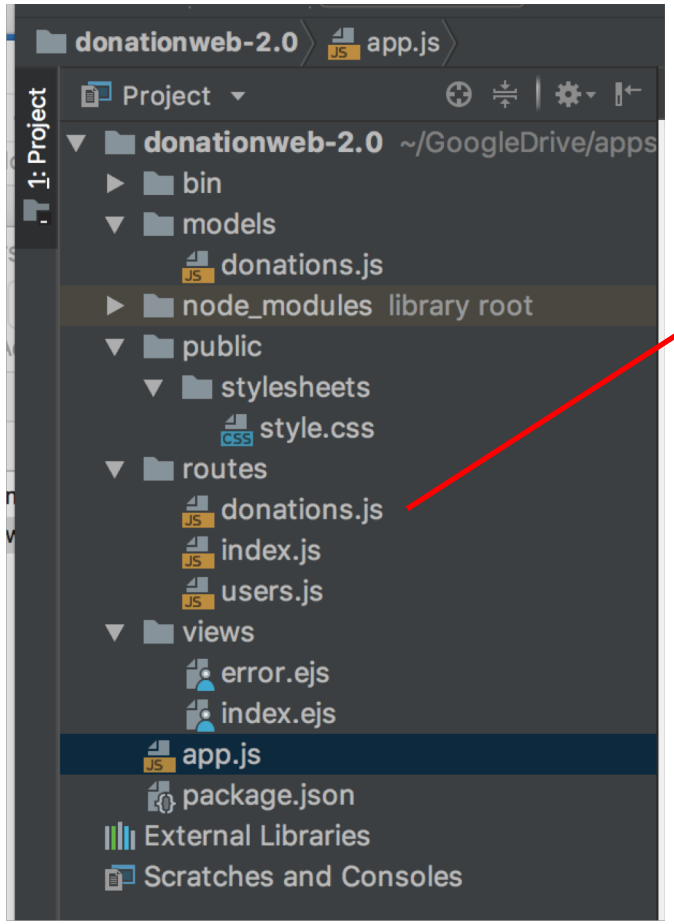
Creating the Model – *Server Side*



donations.js

```
1  const donations = [
2      {id: 1000000, paymenttype: 'PayPal', amount: 1600, upvotes: 1},
3      {id: 1000001, paymenttype: 'Direct', amount: 1100, upvotes: 2},
4      {id: 1000002, paymenttype: 'Visa', amount: 1000, upvotes: 1}
5  ];
6
7  module.exports = donations;
```

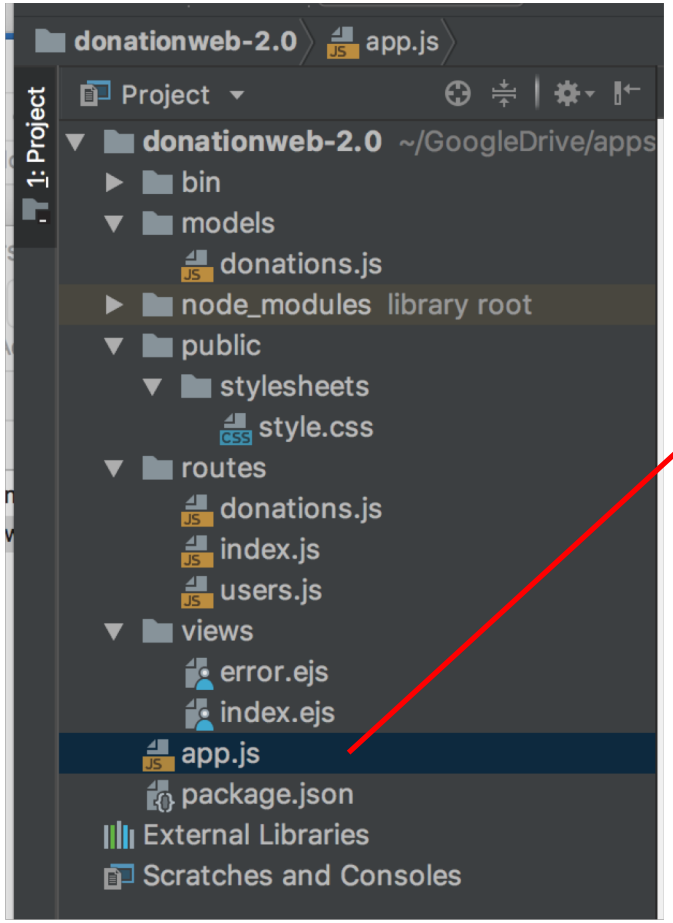
Creating the Routes (1) – *Server Side*




```
donations.js
1   let donations = require('../models/donations');
2   let express = require('express');
3   let router = express.Router();
4
5   router.findAll = (req, res) => {...}
10
11  router.findOne = (req, res) => {...}
23
24  function getByValue(array, id) {...}
28
29  function getTotalVotes(array) {...}
34
35  router.addDonation = (req, res) => {
36      //Add a new donation to our list
37
38      //Randomly generate an id
39      var id = Math.floor((Math.random() * 1000000) + 1);
40      var currentSize = donations.length;
41
42      donations.push({"id" : id, "paymenttype" : req.body.paymenttype,
43                      "amount" : req.body.amount, "upvotes" : 0});
44
45      if((currentSize + 1) == donations.length)
46          res.json({ message: 'Donation Added Successfully!'});
47      else
48          res.json({ message: 'Donation NOT Added!'});
49  }
50
51  router.incrementUpvotes = (req, res) => {...}
64
65  router.deleteDonation = (req, res) => {...}
78
79  router.findTotalVotes = (req, res) => {...}
84
85  module.exports = router;
```

N.B. on ‘imports’

Creating the Routes (2) – *Server Side*



app.js

```
1  var createError = require('http-errors');
2  var express = require('express');
3  var path = require('path');
4  var cookieParser = require('cookie-parser');
5  var logger = require('morgan');
6
7  var indexRouter = require('./routes/index');
8  var usersRouter = require('./routes/users');
9
10 const donations = require("./routes/donations");
11
12 var app = express();
```

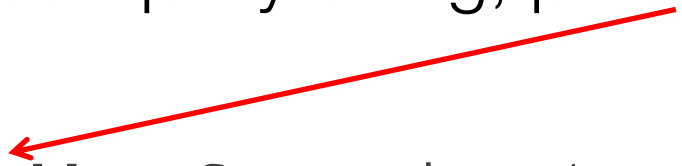
```
27 // Our Custom Donation Web App Routes
28 app.get('/donations', donations.findAll);
29 app.get('/donations/votes', donations.findTotalVotes);
30 app.get('/donations/:id', donations.findOne);
31 app.post('/donations',donations.addDonation);
32 app.put('/donations/:id/vote', donations.incrementUpvotes);
33 app.delete('/donations/:id', donations.deleteDonation);
34
```

```
55
56 module.exports = app;
57
```

The Request object

The req object represents the HTTP request - by convention, the object is always referred to as 'req', Response is 'res'

Can use it to access the request query string, parameters, body, HTTP headers.



```
app.get('/user/:id', function(req, res){
    res.send('user ' + req.params.id);

});
```

Request Properties

req.param(name)	Parameter 'name', if present
req.query	Parsed query string (from URL)
req.body	Parsed request body
req.files	Uploaded files
req.cookies.foo	Value of cookie 'foo', if present
req.get(field)	Value of request header 'field'
req.ip	Remote IP address
req.path	URL path name
req.secure	Is HTTPS being used?

Response Object

The res object represents the HTTP response that an Express app sends back when it gets an HTTP request.

```
app.get('/user/:id', function(req, res){
    res.send('user ' + req.params.id);
});
```

Response Properties

`res.json([body])`

- Sends a JSON response. This method is identical to res.send() with an object or array as the parameter.

```
res.json({ user: 'tobi' });
res.status(500).json({ error: 'message' });
```

Response Properties

`res.send([body])`

- Sends the HTTP response.
- The body parameter can be a String, an object, or an Array.

For example:

```
res.send({ some: 'json' });

res.send('<p>some html</p>');
res.status(404).send('Sorry, we cannot find that!');

res.status(500).send({ error: 'something blew up'
});
```

Response Properties

`res.format(object)`

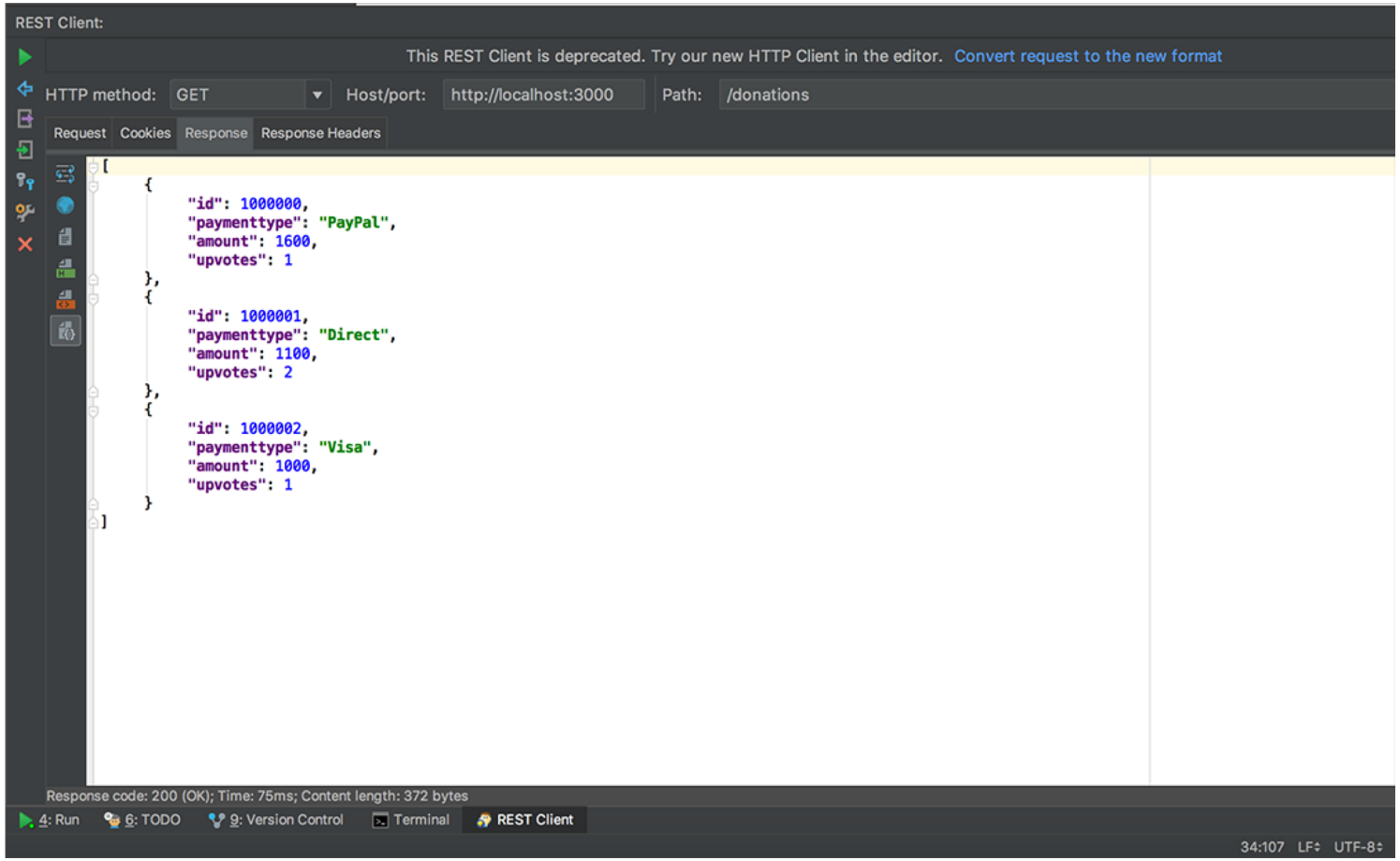
- Performs content-negotiation on the Accept HTTP header on the request object

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },
  'text/html': function(){
    res.send('<p>hey</p>');
  },
  'application/json': function(){
    res.send({ message: 'hey' });
  },
  'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable');
  }
});
```

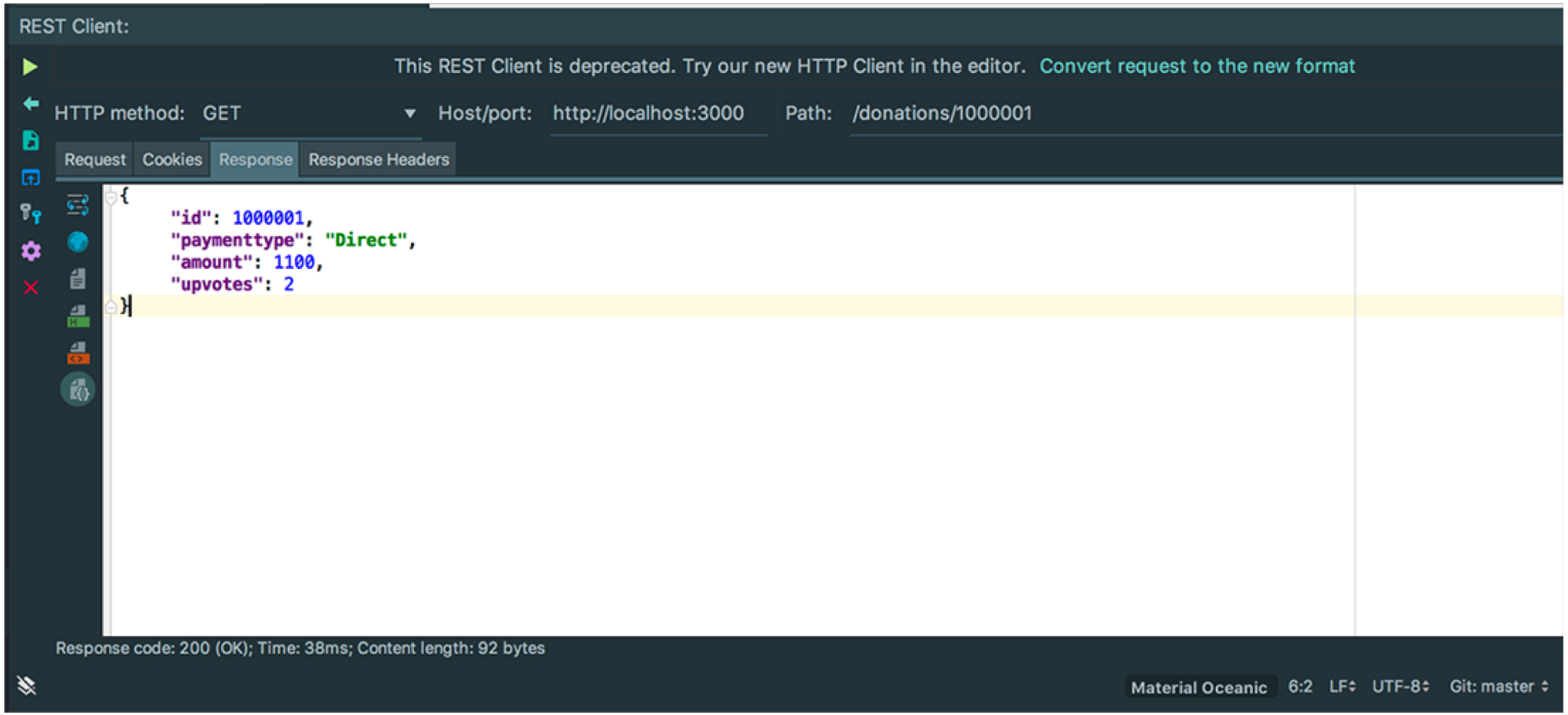
Response Properties

res.status(code)	Sets status 'code' (e.g., 200)
res.set(n,v)	Sets header 'n' to value 'v'
res.cookie(n,v)	Sets cookie 'n' to value 'v'
res.clearCookie(n)	Clears cookie 'n'
res.redirect(url)	Redirects browser to new URL
res.send(body)	Sends response (HTML, JSON...)
res.type(t)	Sets Content-type to t
res.sendfile(path)	Sends a file

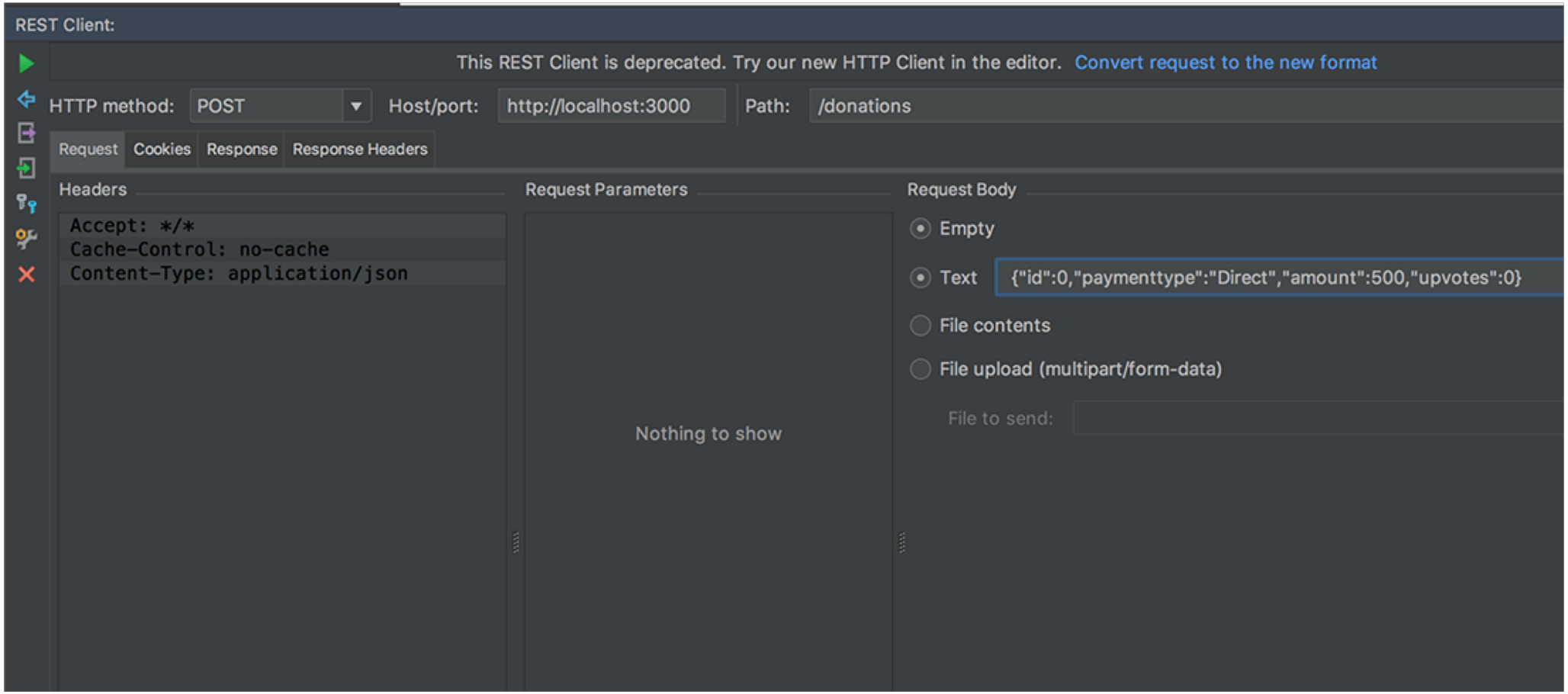
Testing the Routes (1) – GET all



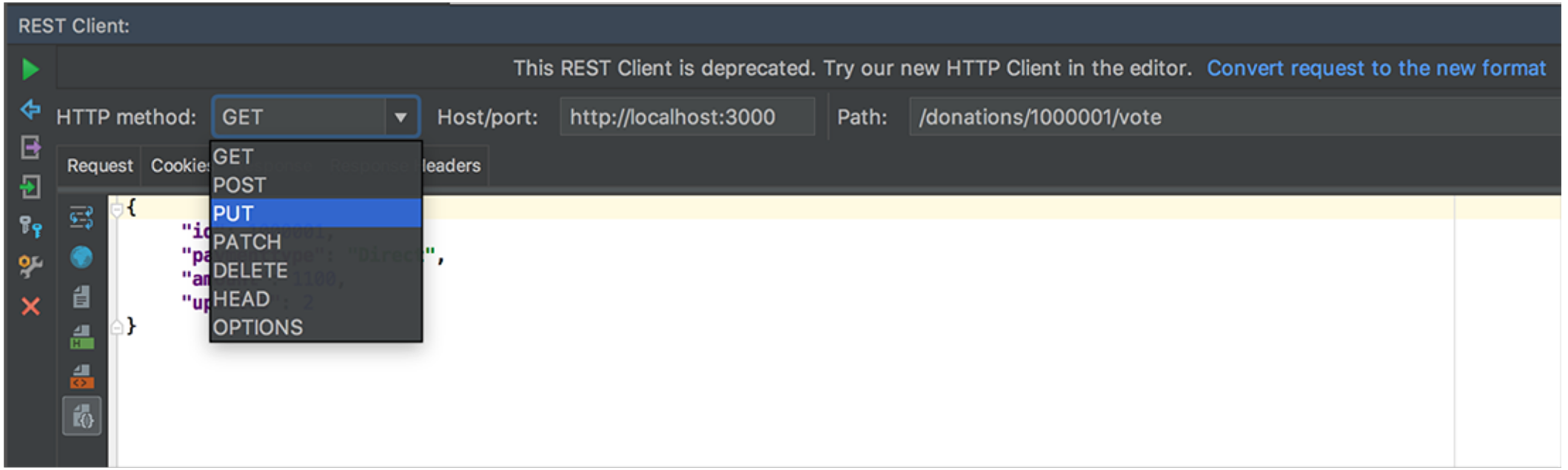
Testing the Routes (2) – GET one



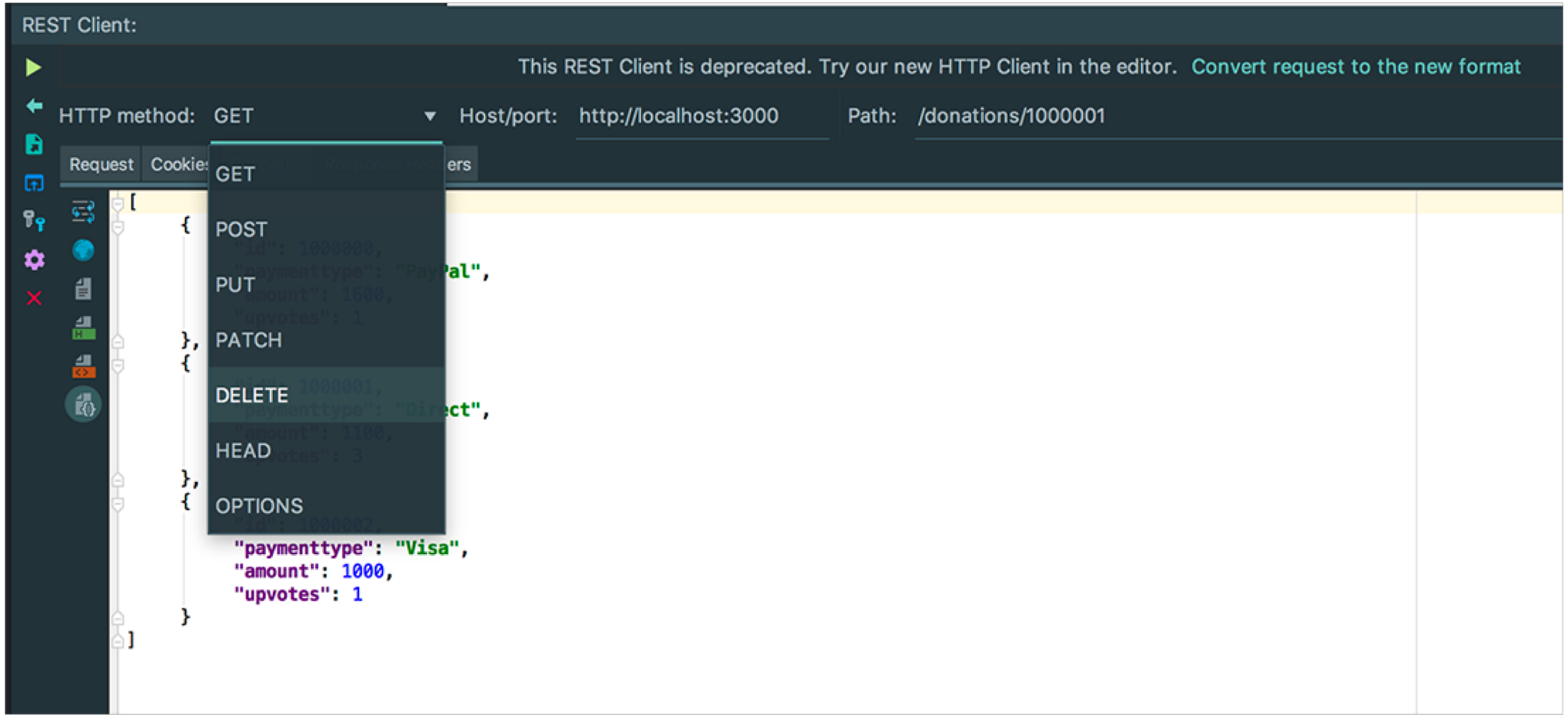
Testing the Routes (3) – POST



Testing the Routes (4) – PUT



Testing the Routes (5) – DELETE



Serving static content

```
app.use(express.static(path.join(__dirname, 'public')));
```

Where content lives in the file system on the server

Your web app will probably have static files

- Examples: **Images**, CSS, client-side JavaScript, ...

Writing an app.get(...) route every time would be too cumbersome

Solution: express.static

Serving static content

If you have different folders for different types of resources then you can set express.static middleware as shown below.

Example: Serve resources from different folders

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

app.use(express.static('public'));

//Serves all the request which includes /images in the url from Images folder
app.use('/images', express.static(__dirname + '/Images'));

var server = app.listen(5000);
```

Serving static content

In the above example, app.use() method mounts the express.static middleware for every request that starts with "/images". It will serve images from images folder for every HTTP requests that starts with "/images". For example, HTTP request http://localhost:5000/images/myImage.png will get myImage.png as a response. All other resources will be served from public folder.

Now, run the above code using `node server.js` and point your browser to *http://localhost:5000/images/myImage.jpg* and it will display myImage.jpg from the **images** folder, whereas *http://localhost:5000/myJSFile.js* request will be served from public folder. (images folder must include myImage.png and public folder must include myJSFile.js)

Serving static content

You can also create a virtual path in case you don't want to show actual folder name in the url.

Example: Setting virtual path

```
app.use('/resources',express.static(__dirname + '/images'));
```

So now, you can use *http://localhost:5000/resources/myImage.jpg* to serve all the images instead of *http://localhost:5000/images/myImage.jpg*.

In this way, you can use Express.js to server static resources such as images, CSS, JavaScript or other files.

How to structure the app

Your web app will have several pieces:

- Main application logic
- 'Routes' for displaying specific pages (/login, /main, ...)
- Database model (get/set functions, queries, ...)
- Views (HTML or EJS files)

Suggestion: Keep them in different directories

- **routes/** for the route functions
- **model/** for the database functions
- **views/** for the HTML pages and EJS templates
- Keep only app.js/package.json/config... in main directory

Architectural Styles Encountered With REST

REST ISN'T ALONE ☺

Model-View-Controller (MVC)

Most commonly employed style with frameworks:

- **Model**: Classes responsible for talking to the DB and fetching/populating objects for the application
- **Controller**: Acts as URI Router i.e. routes calls to specific resources and invokes actions based on the corresponding HTTP Method
- **View**: Usually the resource itself that returns the content/representation as requested by the client

May/may-not be true MVC but parts of application usually split as such – leading to clean code organization/separation of concerns

Client-Side MVC

JS heavy pages lead to spaghetti code

Frameworks like Backbone.js, Ember.js implement MVC paradigm on web page itself making code easier to manage/maintain

- **Models**: Data that is fetched/saved from/to the server
- **Views**: HTML elements that display the data and change if the data is updated
- **Controller**: Intercepts user-events and sends appropriate messages to model/views

JS Models communicate with server (controller) to update themselves

Client-side MVC becoming very popular and critical for 'front-heavy'/smart-client web-apps based on Ajax

Event-Based Architectures

Exclusively client-side:

- Required for communicating between various parts of the JS application/elements
- Based on the Observer pattern – an event bus is used for sending/receiving messages across components

Exclusively server-side:

- For implementing asynchronous communications between different process (e.g.: sending email after a particular action)
- Communicating with other processes on the network via a Message oriented Middleware (MoM) (e.g.: RabbitMQ, WebSphereMQ etc.)
- Communicating with client-side apps – using Node.js or Pub/Sub (Publish/Subscribe) web services like PubNub.com or Pusher.com

Conclusion

Just REST isn't enough

100% REST isn't the goal either

Various architectural styles work together in tandem for creating distributed web-based systems

MVC on client-side is gaining high momentum (we'll use this..)

Event-based communication exceedingly important for near-real-time/asynchronous applications (reason for Node.js popularity)

You can learn the REST by reading a few books and designing/implementing a few systems ☺

Great Resources

Official Tutorial – https://nodejs.org/documentation/tutorials/

Official API – https://nodejs.org/api/

Developer Guide – https://nodejs.org/documentation

Video Tutorials – http://nodetuts.com

Video Introduction – https://www.youtube.com/watch?v=FqMIyTH9wSg

YouTube Channel – https://www.youtube.com/channel/UCvhIsEIBIfWSn_Fod8FuuGg

Articles, explanations, tutorials – https://nodejs.org/community/

https://medium.com/@jaeger.rob/introduction-to-nodes-express-js-db5617047150

Questions?