

# Web Application Development

---

Produced  
by

David Drohan ([ddrohan@wit.ie](mailto:ddrohan@wit.ie))

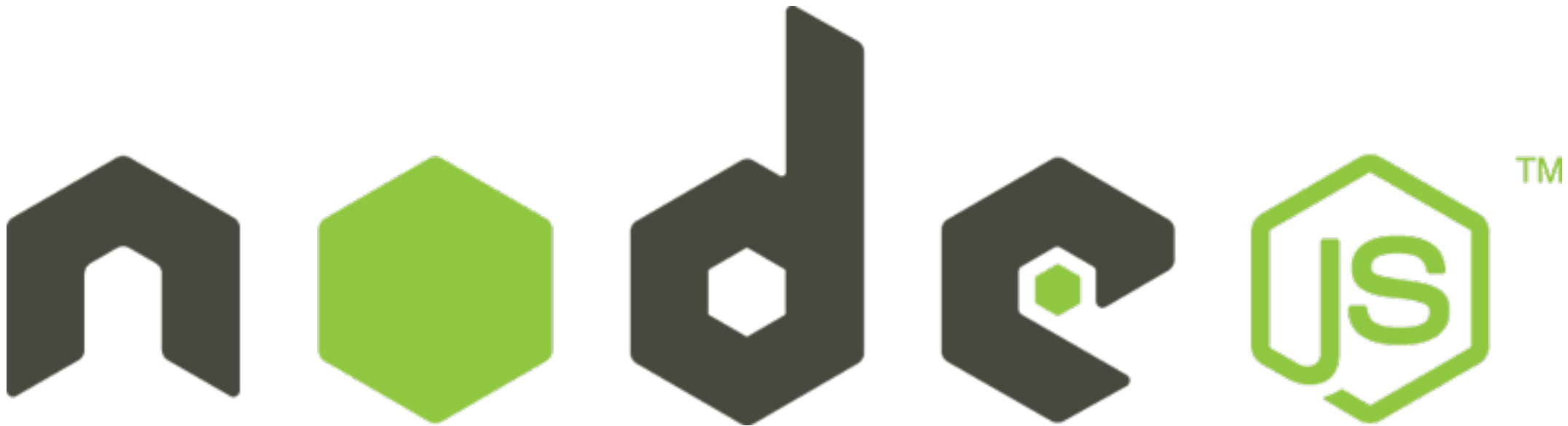
Department of Computing & Mathematics  
Waterford Institute of Technology

<http://www.wit.ie>



Waterford Institute of Technology  
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE





PART 2

---

REST AND EXPRESS



# Outline

---

1. Introduction – What Node is all about
2. Events – Nodes event-driven, non-blocking I/O model
3. Node Modules – The building blocks of Node
4. Express – A Framework for Node
5. REST – The architectural style of the Web
6. API Design – Exposing Application Functionality
7. REST in Express – Leveraging URLs, URI's and HTTP
8. Demo – Labs in action

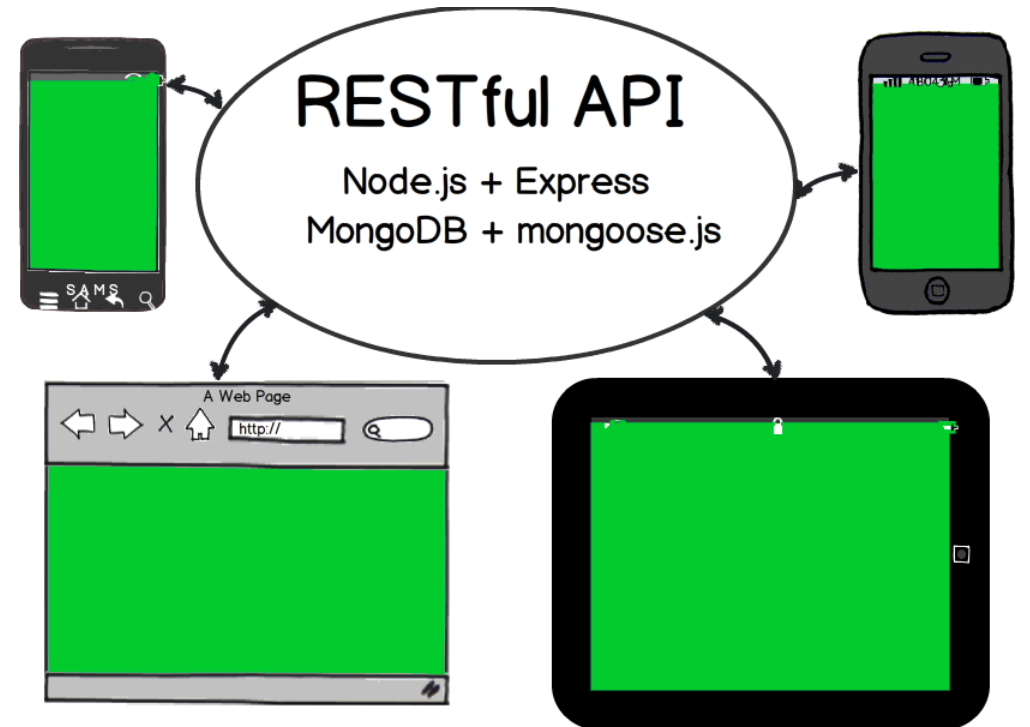
# Outline

---

1. Introduction – What Node is all about
2. Events – Nodes event-driven, non-blocking I/O model
3. Node Modules – The building blocks of Node
4. Express – A Framework for Node
5. REST – The architectural style of the Web
6. API Design – Exposing Application Functionality
7. REST in Express – Leveraging URLs, URI's and HTTP
8. Demo – Labs in action

# REST

THE ARCHITECTURAL STYLE OF THE WEB



# REST (REpresentational State Transfer)

---

*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

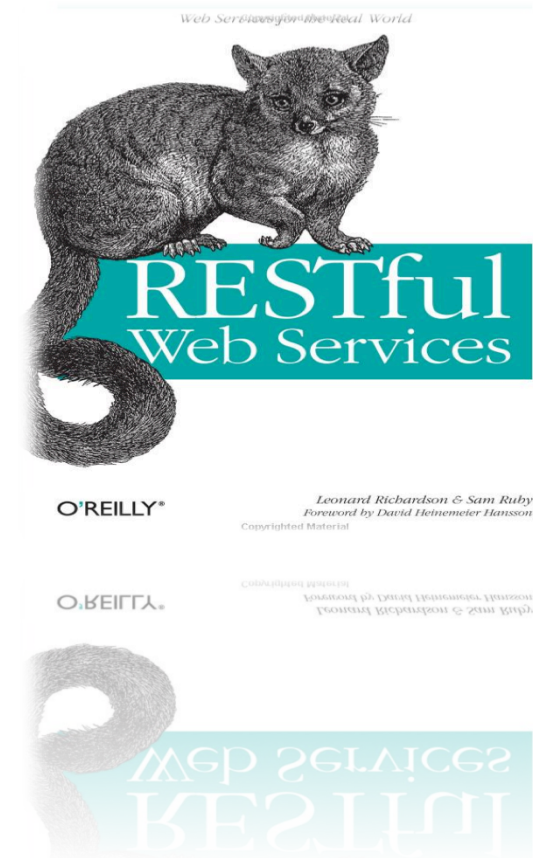
REST is not tied to the 'Web' i.e. doesn't depend on the mechanics of HTTP

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. 100<sup>th</sup> 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/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=123456789087654321234567876543234567865432345678876543&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](https://www.google.com/search?q=CS577+USC)
- Instead of
  - Go to [www.google.com](https://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

# 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 Error*)

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

# Some General Points to Note

---

Authentication/Authorization data 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

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 Routes  
app.get('/donations', donations.findAll);  
app.get('/donations/:id', donations.findOne);  
app.post('/donations', donations.addDonation);  
app.put('/donations/:id/votes', donations.incrementUpvotes);  
app.delete('/donations/:id', donations.deleteDonation);
```

# Donationweb

---

BEHIND THE SCENES <http://donationweb-4-0.herokuapp.com>

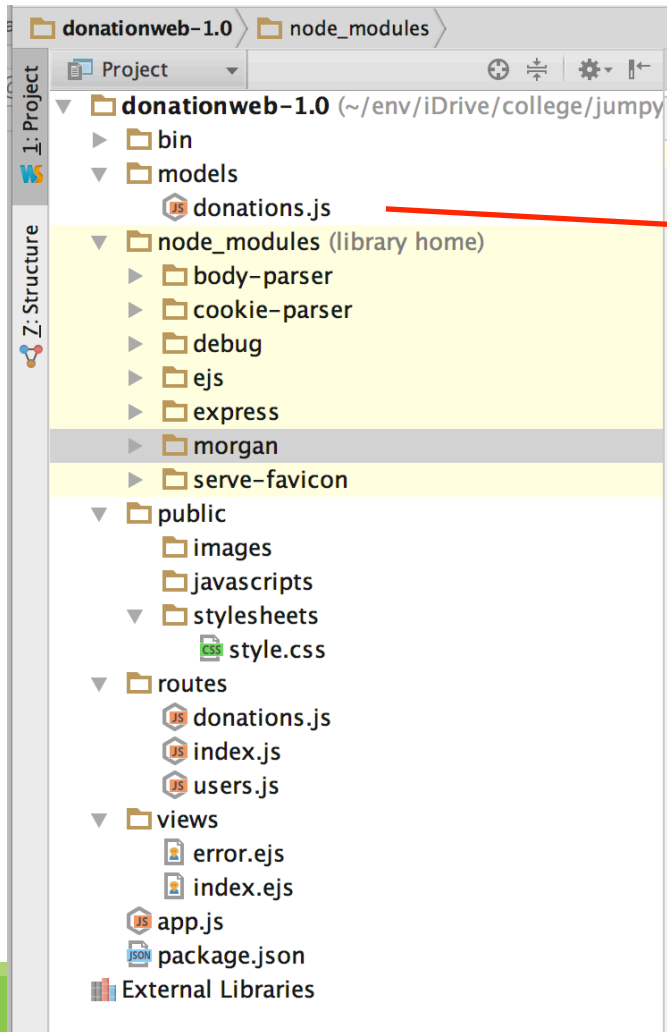
# Donation: Resource, URIs & Methods

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

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

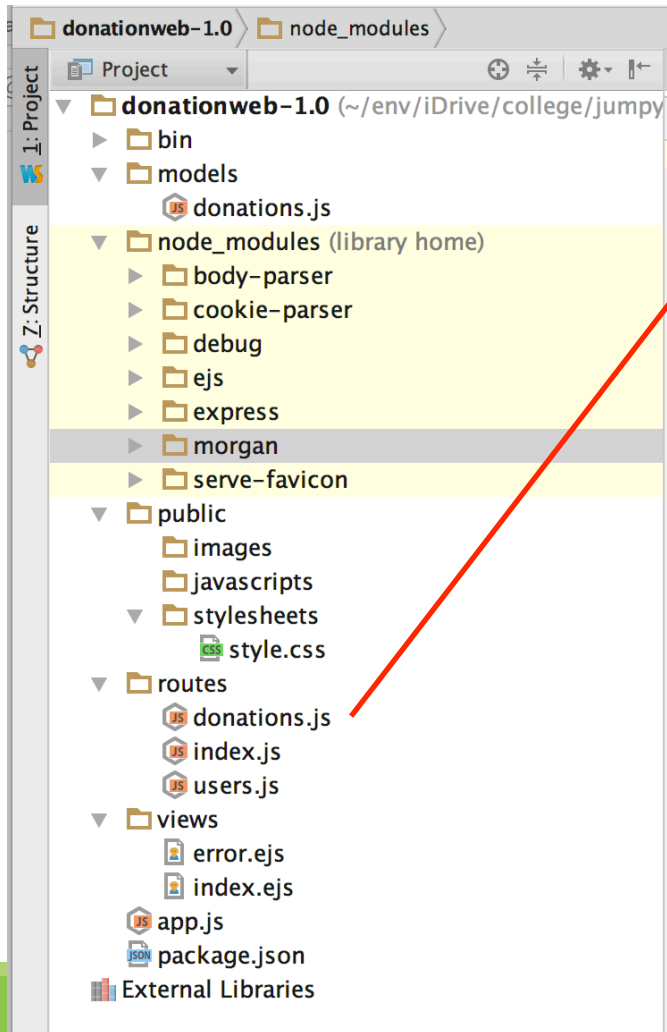
We'll look at this Use Case as an example...

# Creating the Model – *Server Side*



```
donations.js *
1 |var donations = [
2 |    {id: 1000000, paymenttype: 'PayPal', amount: 1600, upvotes: 1},
3 |    {id: 1000001, paymenttype: 'Direct', amount: 1100, upvotes: 2}
4 |    ];
5
6  module.exports = donations;
```

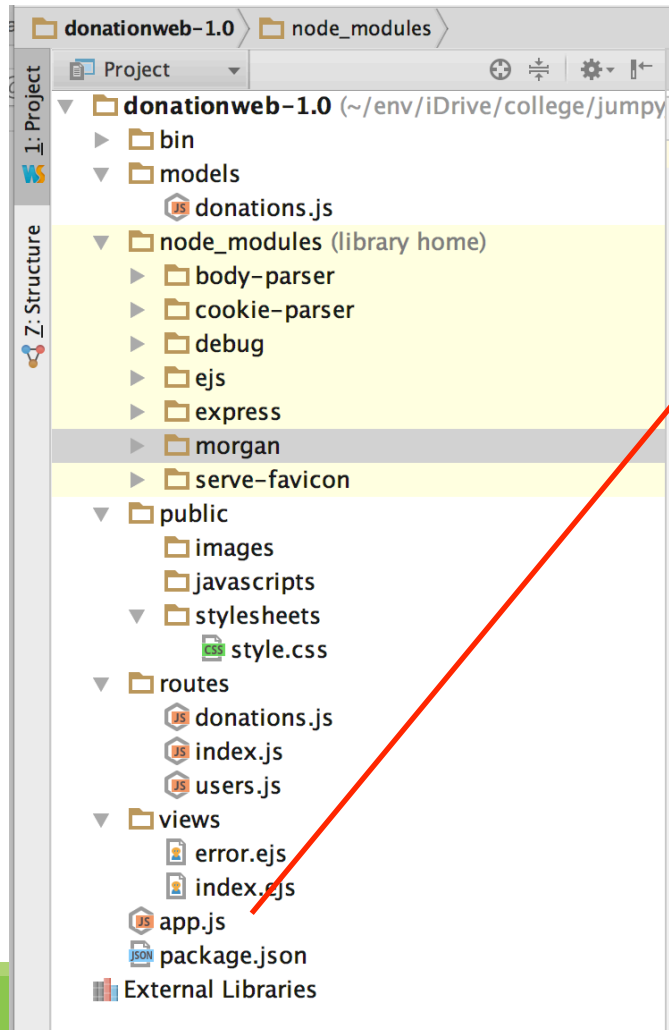
# Creating the Routes (1) – *Server Side*



```
routes/donations.js x
1  var donations = require('../models/donations');
2  var express = require('express');
3  var router = express.Router();
4
5  +function getByValue(arr, id) {...}
11
12  router.findAll = function(req, res) {
13    // Return a JSON representation of our list
14    res.json(donations);
15  }
16
17  +router.findOne = function(req, res) {...}
26
27  router.addDonation = function(req, res) {
28    //Add a new donation to our list
29    var id = Math.floor((Math.random() * 1000000) + 1); //Randomly generate an id
30    var currentSize = donations.length;
31    donations.push({"id":id,"paymenttype":req.body.paymenttype,"amount":req.body.amount,"upvotes":0});
32
33    if((currentSize + 1) == donations.length)
34      res.json({ message: 'Donation Added!'});
35    else
36      res.json({ message: 'Donation NOT Added!'});
37  }
38
39  +router.deleteDonation = function(req, res) {...}
53
54  +router.incrementUpvotes = function(req, res) {...}
59
60  module.exports = router;
```

N.B. on 'imports'

# Creating the Routes (2) – *Server Side*



```

app.js x
1  var express = require('express');
2  var path = require('path');
3  var favicon = require('serve-favicon');
4  var logger = require('morgan');
5  var cookieParser = require('cookie-parser');
6  var bodyParser = require('body-parser');
7
8  var routes = require('./routes/index');
9  var users = require('./routes/users');
10 var donations = require('./routes/donations.js');
11
12 var app = express();
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 //Our Custom Routes
30 app.get('/donations', donations.findAll);
31 app.get('/donations/:id', donations.findOne);
32 app.post('/donations', donations.addDonation);
33 app.put('/donations/:id/votes', donations.incrementUpvotes);
34 app.delete('/donations/:id', donations.deleteDonation);
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70 module.exports = app;
71

```

# 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

---

<code>req.param(name)</code>	Parameter 'name', if present
<code>req.query</code>	Parsed query string (from URL)
<code>req.body</code>	Parsed request body
<code>req.files</code>	Uploaded files
<code>req.cookies.foo</code>	Value of cookie 'foo', if present
<code>req.get(field)</code>	Value of request header 'field'
<code>req.ip</code>	Remote IP address
<code>req.path</code>	URL path name
<code>req.secure</code>	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

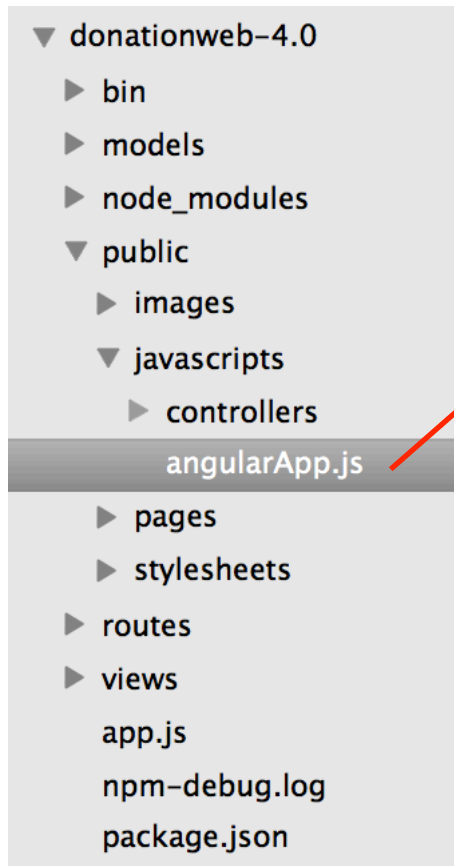
```
res.format({
  'text/plain': function() {
    res.send('hey');
  },
  'text/html': function() {
    res.send('<p>hey</p>');
  },
  'application/json': function() {
    res.send({ message: 'hey' });
  },
  'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not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

# Response Properties

---

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

# Creating the Routes (1) – *Client Side*



```
angularApp.js
1  var app = angular.module('DonationWebApp', ['ngRoute']);
2
3  app.config(function($routeProvider) {
4      $routeProvider
5
6          // route for the home page
7          .when('/', {
8              templateUrl : 'pages/home.ejs',
9              controller : 'mainController'
10         })
11         // route for the donate page
12         .when('/donate', {
13             templateUrl : 'pages/donate.ejs',
14             controller : 'donateController'
15         })
16         // route for the donations page
17         .when('/donations', {
18             templateUrl : 'pages/donations.ejs',
19             controller : 'donationsController'
20         })
21         // route for the about page
22         .when('/about', {
23             // ...
24         })
25     });
26     // route for the contact page
27     .when('/contact', {
28         // ...
29     });
30
31 });
```

# Creating the Controllers – *Client Side*

- ▼ donationweb-4.0
  - ▶ bin
  - ▶ models
  - ▶ node\_modules
  - ▼ public
    - ▶ images
    - ▼ javascripts
      - ▼ controllers
        - aboutcontroller.js
        - contactcontroller.js
        - donatecontroller.js
        - donationscontroller.js
        - maincontroller.js
      - angularApp.js
    - ▶ pages
    - ▶ stylesheets
    - ▶ routes
    - ▶ views
    - app.js
    - npm-debug.log
    - package.json

```

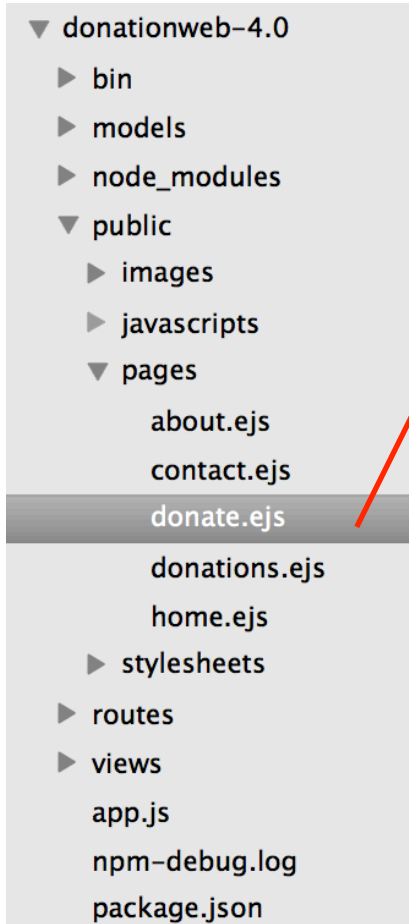
donatecontroller.js
1  var app = angular.module('DonationWebApp');
2
3  app.controller('donateController', ['$scope', '$location', '$http', function($scope, $location, $http) {
4
5      $scope.formData = {};
6      $scope.message = 'Donate Page!';
7      $scope.amount = 1000;
8      $scope.options = [{ name: "PayPal", id: 0 }, { name: "Direct", id: 1 }];
9      $scope.formData.paymentOptions = $scope.options[0];
10
11     //Reset our formData fields
12     $scope.formData.paymenttype = 'PayPal';
13     $scope.formData.amount = 1000;
14     $scope.formData.upvotes = 0;
15
16     $scope.addDonation = function(){
17         $scope.formData.paymenttype = $scope.formData.paymentOptions.name;
18         $http.post('/donations', $scope.formData)
19             .success(function(data) {
20                 $scope.donations = data;
21                 $location.path('/donations');
22                 console.log(data);
23             })
24             .error(function(data) {
25                 console.log('Error: ' + data);
26             });
27     };
28 }
29 ]);

```

addDonation()



# Creating the Views – *Client Side*



```

donate.ejs
1 <div class="jumbotron text-center">
2   <h1>Make a Donation</h1>
3
4   <p>{{ message }}</p>
5
6   <div ng-controller="donateController">
7     <div class="row">
8       <div class="col-md-6 col-md-offset-3">
9
10        <form ng-submit="addDonation()"
11          style="margin-top:30px;">
12          <h3>Add a new Donation</h3>
13
14          <div class="form-group" align="center">
15            <input type="text" class="form-control" />
16          </div>
17          <div class="form-group" align="center">
18            <input type="text" class="form-control" />
19          </div>
20          <button type="submit" class="btn btn-primary">Donate</button>
21        </form>
22      </div>
23    </div>
24  </div>
25 </div>
  
```

The .ejs pages are normal HTML pages but can have 'blanks' in them that we can fill with data at runtime


Need a new page? Just add a new controller!

# 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, client-side JavaScript, ...

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

Solution: `express.static`

# 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 😊

A solid green horizontal bar at the bottom of the slide.

# 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 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

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=FqMlyTH9wSg>

YouTube Channel – [https://www.youtube.com/channel/UCvhlsEIBlfWSn\\_Fod8FuuGg](https://www.youtube.com/channel/UCvhlsEIBlfWSn_Fod8FuuGg)

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

---

# Questions?