



Department of Computer Science

Dogfooding: Separating Roles by Consuming Internal APIs

Perry Donham

@perrydBUCS

perryd@bu.edu

Dogfooding? Really?

- The term comes from either Lorne Greene's Alpo ads ("I feed my own dogs Alpo), or Kalkin ... their president would famously eat the company's dog food at shareholder meetings



Source: (cc) <http://www.elezea.com/2015/02/eat-your-own-dog-food>

- In a general sense **dogfooding** means that we use our own product to do our work
 - If your firm makes accounting software, that's what the finance team uses
 - If you're Ford, your engineers drive Fords
- The goal is to give your developers a solid understanding of the customer experience

Dogfood and APIs

- For our discussion we'll narrow this down to a web-based application that consumes its own API
- API: Application Programming Interface
 - Used to refer just to libraries used internally
 - Now hijacked by devs to mean interface to access external data
 - Most web-based services have a public-facing API (Twitter, Amazon, Google, Spotify, etc)
 - ProgrammableWeb.com lists 15,000+ public APIs
- We're interested in how an **internal** API can be used to draw a strong line around the **model** component of an MVC (model-view-controller) architecture













Here's what we'll implement...



Department of Computer Science

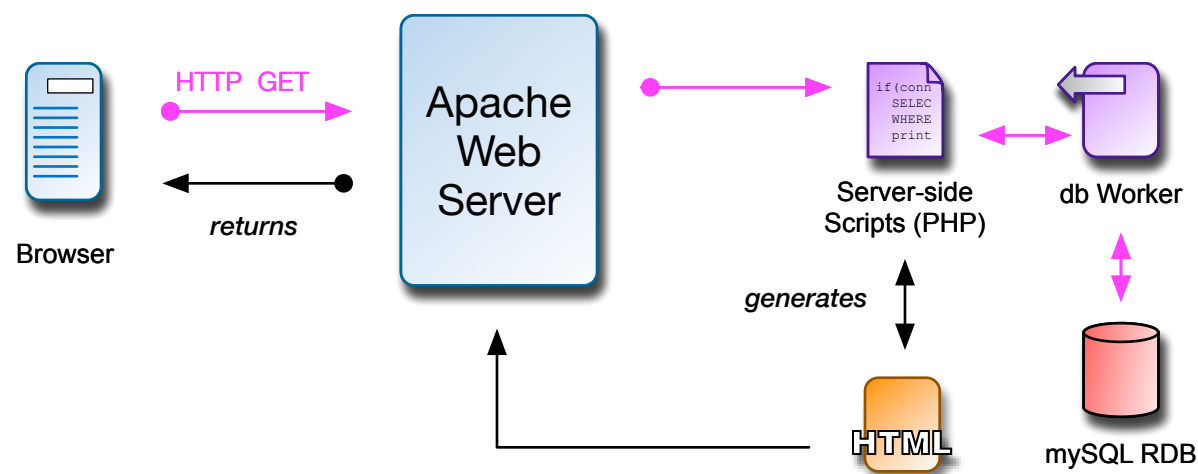
Add user

<input type="text" value="Enter name"/>	<input type="text" value="Enter UID"/>	<input type="text" value="Enter Department"/>	<input type="button" value="Add User"/>
-----------------------------------------	----------------------------------------	-----------------------------------------------	-----------------------------------------

PerryD	  
Wayne	  
Ernie	  
Bob	  

- We'll see that implementing MVC via dogfooding provides several important advantages over traditional client-server approaches
- Let's first take a quick look at the history of web app architecture... how have web apps typically been designed?

Traditional client-server on LAMP stack

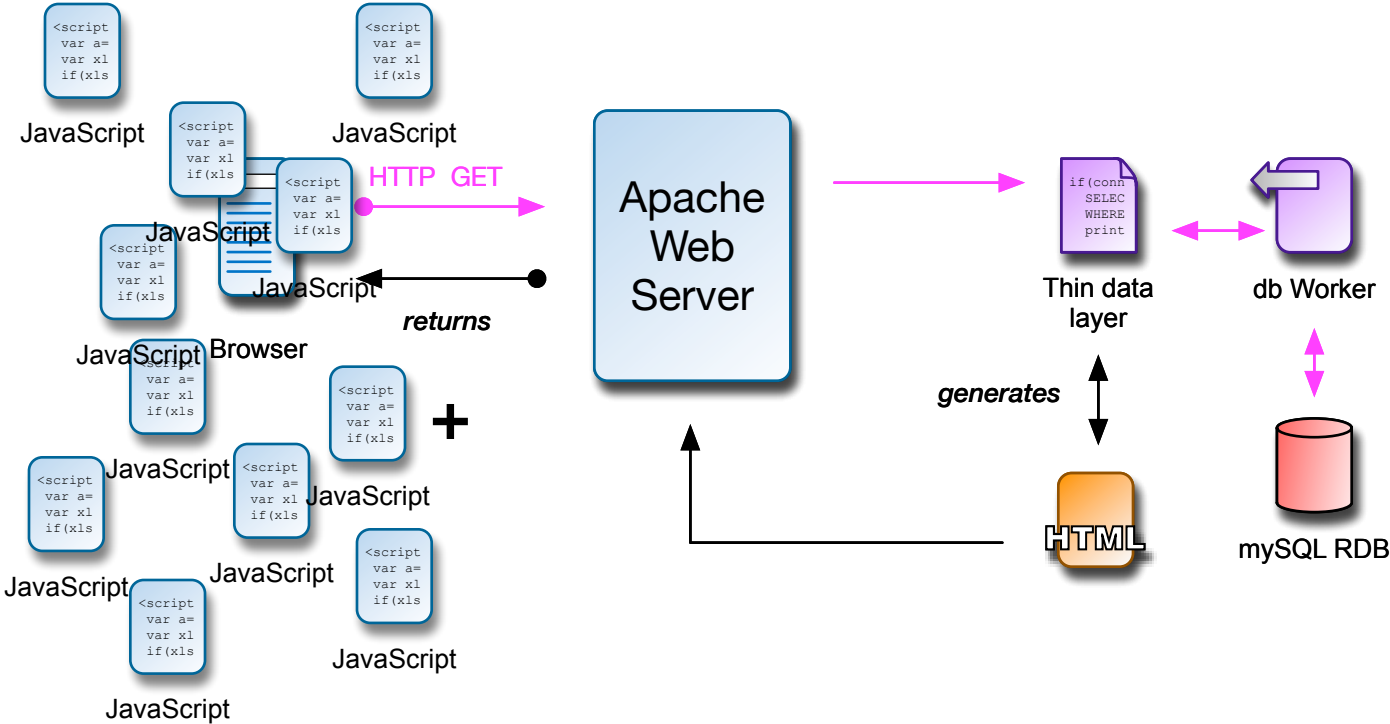


- This is **thin-client**
 - Work is done at the back end
 - Back-end data sources (BEDS) are typically relational
 - Scripting includes templating engines (Smarty, etc)
- LAMP-like architectures have been used since Day 1 on the web ... I wrote such a site in PERL in 1995
- There are several potential performance bottlenecks, including
 - RDB
 - Interpreter

- This is MVC (model-view-controller) for the web
 - The database is the model
 - Buttons and other actions on the web page are the controller
 - The web page itself is the view
- It's a fairly clean implementation of the pattern, and each component has well-defined roles

Javascript on the front end

- Javascript was released around 2000 to add functionality to what previously were static web pages
- Direct access to the document object model (DOM) allowed programmers to provide some interactivity to the page
- It wasn't really until 2004, when Google started deploying apps using asynchronous browser calls to the server via XML (AJAX) that we started getting serious about JS
- We got a little crazy with the AJAX stuff, and the pendulum swung...



- This is **thick-client** ... most of the work is being done on the front end
- Is that the right place?
- No one seemed to care!
- We were addicted to the sweet taste of AJAX interaction ... our web pages came to life!

The trouble with front-end Javascript

- Even though JS is compiled to machine language by the browser, it is compiled to machine language *in the browser*
- To accomplish this, we have to send the source *to the browser*
- We can obfuscate and minify all we want, but those are reversible functions
- Any actions taken by the front-end code are exposed
- What if your page is making calls to third-party services?
- Once the code hits the browser, it's pretty much out of your control

If you're doing this in the browser...

```
passport.use(new TwitterStrategy({
  consumerKey: 'xT0JrIMSWVc2XX4kZDXXXXDE3',
  consumerSecret: '1FXydshtl0xRF1P6vXXXXXXXXXUZdcW69P2kY0WFAVZJpn0Px1Q',
  callbackURL: "http://" + serverParams.server.host + ":"
    + serverParams.server.port + "/todo/auth/callback"
},
function (token, tokenSecret, profile, done) {
  console.log("Got Twitter user: " + profile.displayName);
  //create new user record
  var theUser = new models.Users({name: profile.displayName, isLoggedIn: true});
  //and add to db
  db.insert(theUser, function (err, theUser) {
    if (err) {
      return done(err);
    }
  });
  next();
});
```

...those secrets won't be secret for long!

Breaking MVC

- If we consider the page to be the **view** component, and its buttons, forms, and events to be the **controller**, where is the **model**?
- The short answer: It ends up being spread around
 - State is held in the page itself
 - Changes in state end up being initiated by view code
 - The lines between the components are quite blurry
- This divergence from a clean architectural model ends up being difficult to test, difficult to maintain, and difficult to secure

Node.js

- In 2010 a fascinating thing happened: A new framework was released for *server-side* execution of Javascript
- The capability had been around since at least 2000, but it wasn't widely used
- It worked with Google's V8 Javascript engine, the same engine deployed in browsers, and provided a built-in web server
- In that moment, developers could write Javascript on both the front end *and* the back end
- It was now possible to move much of the work from the front end to the back end

Serialization

- It gets better...
- Javascript uses a lightweight key-value string representation for serialization of objects called **JSON** (Javascript Object Notation)

```
{  
  "_id": "571d11662dc89227e6d982c0",  
  "name": "Perry",  
  "UID": "U123456",  
  "department": "BUCS",  
  "__v": 0  
}
```

- JSON quickly became the transport encoding of choice for moving data back and forth between the front end and the back end
- It wasn't long before someone asked...

...why can't we just store data in JSON?

- The answer to that was a flood of non-relational, document-based data stores that offer CRUD (create, read, update, delete) operations on objects
- The blanket term for this class of store is **noSQL**
- One of the more popular is MongoDB
- Another is an in-memory database, redis, that provides constant-time CRUD
- These are *fully denormalized* document stores ... all relational structures are held in the document

```

var Schema = mongoose.Schema;
var personSchema = new Schema ({
  name: String,
  UID: String,
  department: String
});
var people = mongoose.model('people', personSchema);

aPerson = new people(
  {
    name: 'Perry',
    UID: 'U123456789',
    department: 'BUCS'
  }
);
aPerson.save(function(err) {
  if (err) {res.send(err);}
  else {res.send ({message: 'success'})}
});

var foundPerson;
people.find({name: 'Perry'}, function(err, result) {
  if (!err) { foundPerson = result;}
})

```

← object prototype

← object instantiation

← store document

← retrieve document

About those lambdas...

```
people.find({name: 'Perry'}, function(err, result) { ... })
```

- The V8 Javascript engine main function is single-threaded and non-blocking (it's equivalent to a `listen()` method)
- I/O is handled asynchronously by worker threads that take a callback function as an argument
- When the I/O is complete, the callback function is passed back to the main thread in a call stack that includes results as parameters of the callback function
- This is similar to the continuation-passing style of programming in languages like Scheme.

- This makes sense when you recall that Javascript started life as a way to add functionality to things like buttons on a web page, which fire events asynchronously
- It does mean that Javascript programmers must take **asynchronicity** into account when designing applications
- **Scope** especially becomes important, since functions are being run asynchronously in different contexts
- We typically use closures — portable scope — to manage this

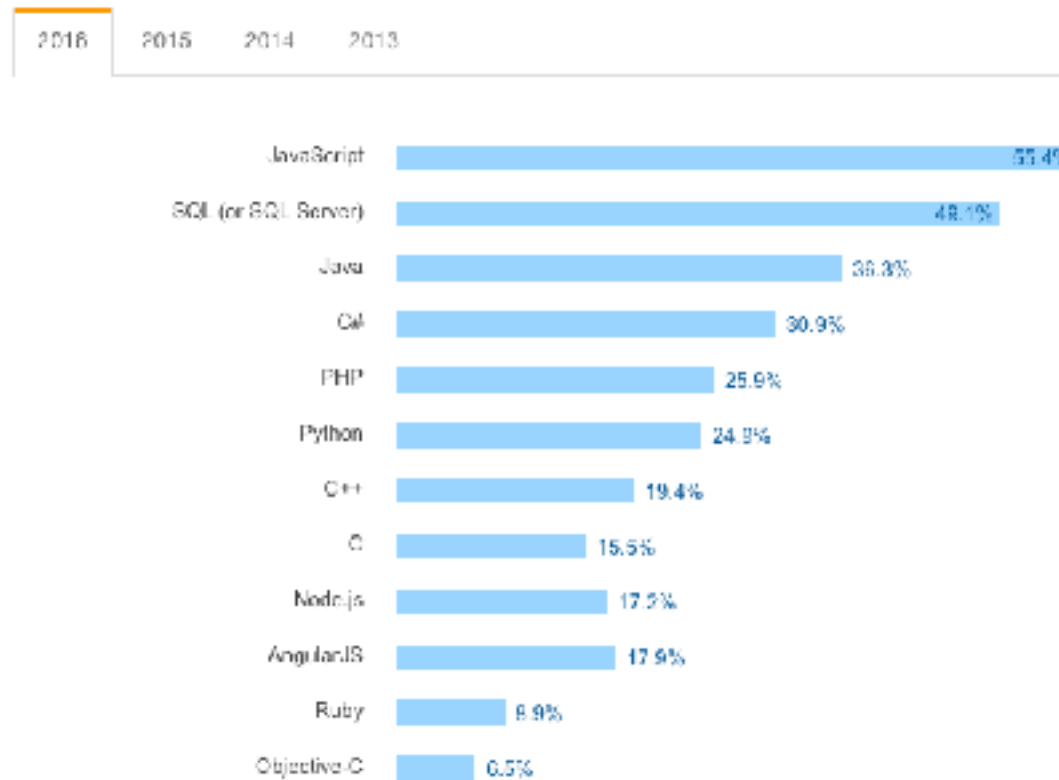
2010: JS transitions to mainstream

- This confluence of front- and back-end Javascript, coupled with databases that speak JSON, created a class of programmer that could move fluidly across the entire stack
- Further, it simplified the interfaces; for example, when an HTML form POSTs a JSON string, and the database uses JSON natively, there are no transformations needed

```
status = new people(httpRequest.body)
        .save(function(err, result) { ... }
        );
```

- This full-stack cohesion, along with a highly performant and horizontally scalable platform, has made Javascript extremely popular for web application development
- Large corporate deployments include
 - GoDaddy
 - Groupon
 - IBM
 - Netflix
 - PayPal
 - Walmart

I. Most Popular Technologies



Source: Stackoverflow 2016 survey n=50,000

Regroup!

- We're close to having all of the plumbing necessary to construct a fully decoupled, API based app
- We have:
 - Serialization using JSON
 - JSON based data stores
 - A back-end web server (Node.js)
- We still need:
 1. A way to call the API from the front end
 2. A way to implement an API on the back end

1. REST: Calling the API

- Representational State Transfer was described in 2000 by Roy Fielding in his PhD dissertation
- It provides a simple way to map HTTP semantics onto CRUD data operations
- **This is the decoupling mechanism**
 - The client interface is through a URL
 - The client doesn't know where requests are being satisfied, just that they are
- REST isn't a standard, just a style, but is in wide use
- While HTTP provides a dozen or so verbs, we'll only use four

URI	HTTP GET	HTTP PUT	HTTP POST	HTTP DELETE
Collection <u>//my.com/people</u>	List all people in db	Replace entire collection	Create a new person	Delete the entire collection
Record <u>//my.com/people/perryd</u>	Fetch details of specified person	Replace/update specified person	Not typically used	Delete the specified person

Create: POST
Read: GET
Update: PUT
Delete: DELETE

\$http in Angular

- Angular.js is a **front-end** framework that extends both HTML and Javascript
- We'll see a full example in a moment, but to make an HTTP request, we just need to instantiate the \$http object and fill in the method and any parameters
- If, for example, we had an HTML form to create a new person in our application, we'd bind this function to the 'Create User' button click event...

```
<button ng-click="createUser()">
```

```
angular.module('csdemo', [])  
  .controller('csdemoctrl', function($scope, $http){  
    $scope.createUser = function() {  
      var request = {  
        method: 'post',  
        url: 'http://localhost:3000/api/db',  
        data: {  
          name: $scope.name,  
          UID: $scope.UID,  
          department: $scope.department  
        }  
      };  
      $http(request)  
        .then(function(response){  
          $scope.showUser();  
        })  
    }  
  });
```

2. URL routing / dispatching

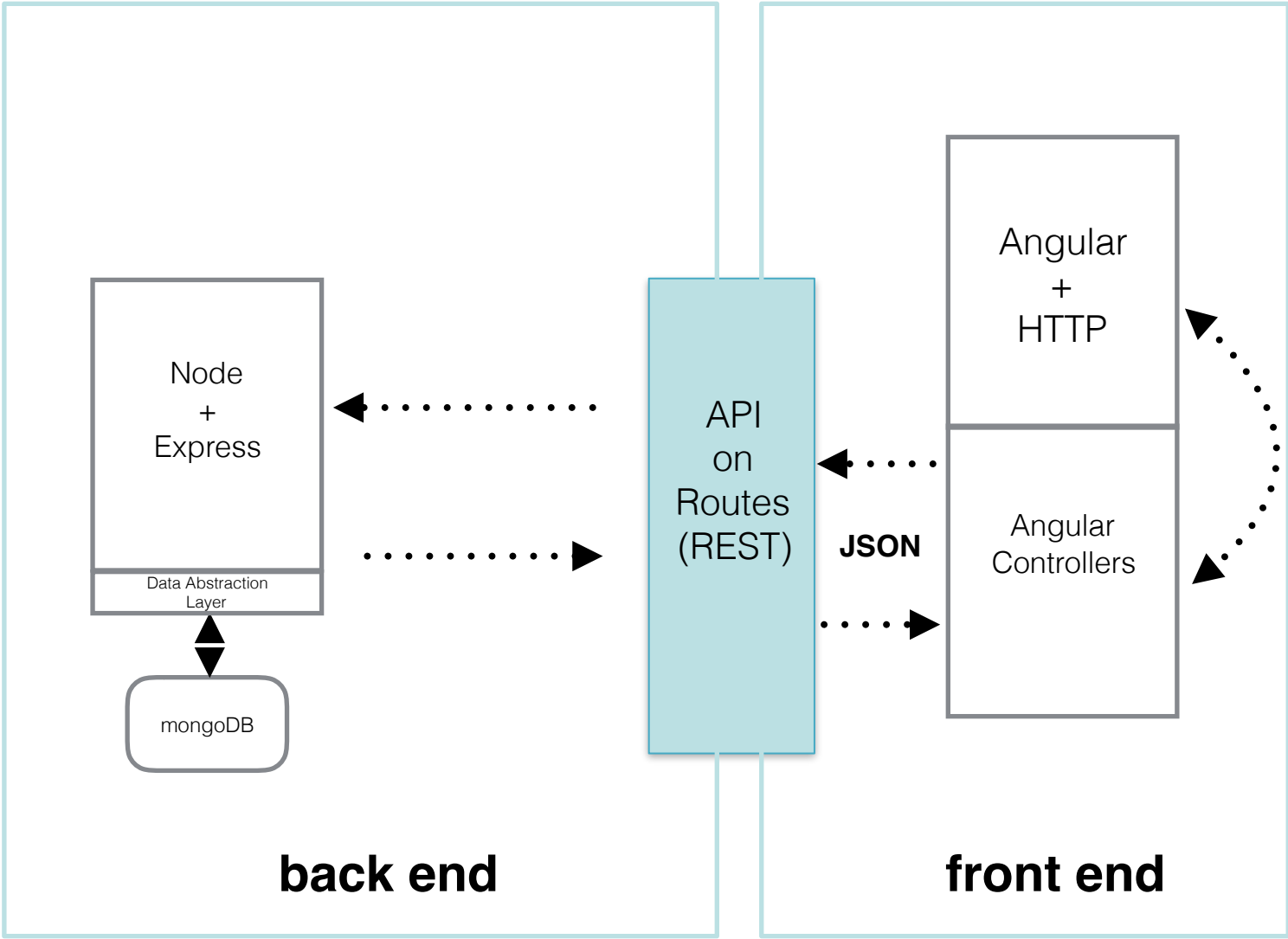
- Now that we've defined a RESTful API we need a way to map it to methods on the **back end**
- Routing must take both the URL and the HTTP verb into account
- Most languages with web frameworks provide this, either natively or as a library
 - Python: Flask, Bottle in Django framework
 - Javascript: Express.js in Node.js framework
 - Ruby: Rails
- Routing is done with regexp-like pattern matching

```
//Express routing in Node.js

var express = require('express');
var router = express.Router();

router.get('/db', function (req, res, next) {
  people.find({}, function (err, results) {
    res.json(results);
  })
});
```


- We now have all of the pieces to decouple the front end and back end
- State is held in the model
- Changes in state are initiated by the controller
- The view is strictly representational (with some decoration)



Consequences

- This sort of RESTful architecture isn't all puppies and rainbows
- We've severed the link between the view and model
 - In the implementation we've just seen the controller is responsible for initiating state changes in the model
 - However, the controller is also responsible for updating the view
 - In classical MVC the view can be independent of the controller
 - In fact, there might be many views, not necessarily all in the same application instance

- This might not matter, depending on the use case
- If it does matter, i.e. a distributed securities trading app, we can use a Observer pattern (publish-subscribe)
 - With pub-sub, the view would register with the model and be notified immediately when state changes
 - This might be a push of new model data or a notification so that the view's controller can decide what to do

- For web apps an appropriate implementation is through a web socket
 - These are full-duplex connections over port TCP:80 (or whatever port the HTTP server listens to)
 - They are handled as a URI in the form ws://localhost/...
- It's fair to argue that if you are using web sockets, REST isn't necessary
- However, using web socket connections to connect the view/controller directly to the model tightly **couples** them, which is what we're trying to avoid

What have we gained? *Security*

- Sensitive data, including keys and tokens, are held on the back end
- We have full control over third-party API calls
- We can fully validate inputs passed from the front end

What have we gained? *Reusability*

- The back end is just CRUD across one or more data sets
- We can reuse these capabilities over and over in new applications

What have we gained? *Abstraction*

- From the controller and client view, the model is abstracted
 - We can place whatever intermediate steps we want in between them
- Further, the model itself is abstracted into JSON
 - We can use any data store, either with native JSON or an adapter
- Since the model publishes in a standard form (JSON) it is agnostic of its clients

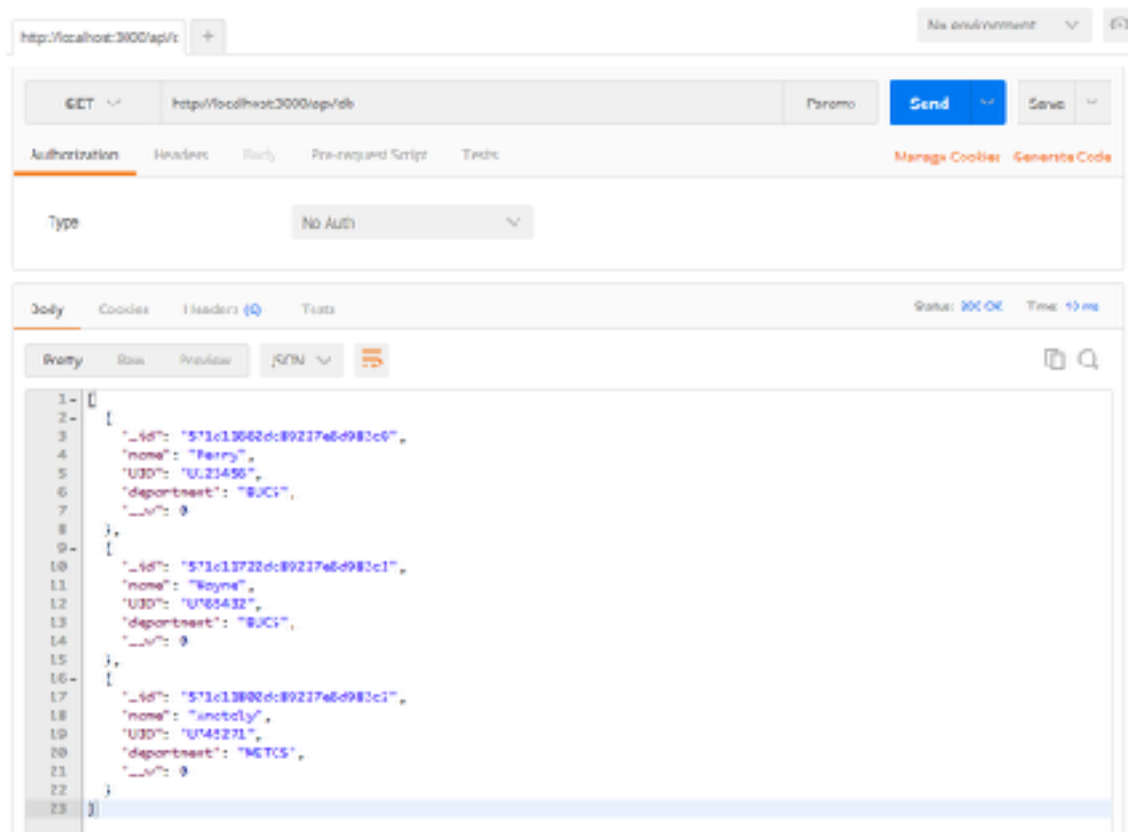
What have we gained? *Performance*

- Data operations are typically hotspots resulting in degraded performance as load increases
- By moving to denormalized non-relational data stores we remove much of the overhead required by more traditional RDBs
- Additionally, noSQL DBs such as mongoDB are designed to easily scale horizontally (via sharding and clustering)
 - We can use several low-cost data servers instead of a few high-cost vertically scaled ones

- The V8 Javascript engine is **optimized** to handle small requests at a high level of concurrency
- To further improve performance, the site can be configured to use a separate server, such as nginx, to serve static content such as images, leaving Node.js to handle dynamic requests

What have we gained? *Testability*

- The strong division of responsibility means that we readily test the internal API without use of a front end
- We reduce the universe of inputs by rigidly specifying the interface



Using Postman to perform API tests

What have we gained? *Concurrent development*

- Since the front end is completely decoupled from the back end, we can work on them simultaneously
- The front end can use stubbed API calls while the back end is being completed
- Note that this requires firm requirements

True MVC

- The most important advantage is that we've shifted from a muddled architecture with loosely defined responsibilities to a strict MVC framework in which roles are clearly delineated
 - All model operations take place on the back end
 - All view and controller operations take place on the front end
 - By using a RESTful API with JSON as the transport, we remove platform dependencies ... the application might have Python on the back end and JS on the front, for example
 - The model is also treated as an abstraction (using JSON) and so the data store is decoupled

Dogfooding really is good for you!

