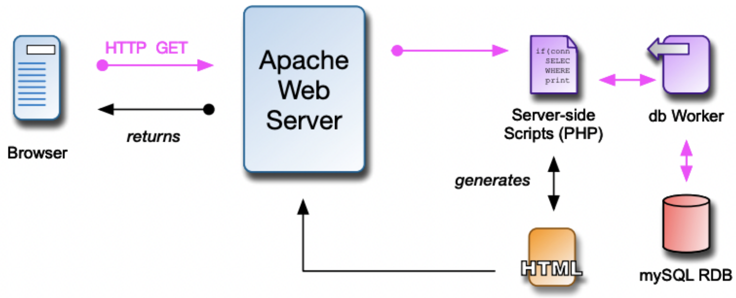
CAS CS 411
Lec 5

Dogfooding: Separating Roles by Consuming Internal APIs

1. Dogfooding? Really?
    a. The term comes from either Lorne Greene’s Alpo ads (“I feed my own dogs Alpo), or KalKan … their president would famously eat the company’s dog food at shareholder meetings
    b. In a general sense dogfooding means that we use our own product to do our work
        i. If your firm makes accounting software, that’s what the finance team uses
        ii. If you’re Ford, your engineers drive Fords
    c. The goal is to give your developers a solid understanding of the customer experience
2. Dogfood and APIs
    a. For our discussion we’ll narrow this down to a web-based application that consumes its own API
    b. API: Application Programming Interface
        i. Used to refer just to libraries used internally
        ii. Now hijacked by devs to mean interface to access external data
        iii. Most web-based services have a public-facing API (Twitter, Amazon, Google, Spotify, etc)
        iv. ProgrammableWeb.com lists 15,000+ public APIs
    c. 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
3. Traditional client-server on LAMP stack



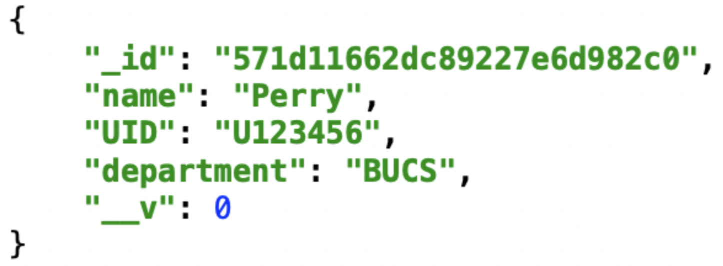
    a.
    b. This is thin-client
        i. Work is done at the back end
        ii. Back-end data sources (BEDS) are typically relational
        iii. Scripting includes templating engines (Smarty, etc)
    c. LAMP-like architectures have been used since Day 1 on the web
    d. There are several potential performance bottlenecks, including RDB and Interpreter

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

4. Javascript on the front end
   a. Javascript was released around 2000 to add functionality to what previously were static web pages
   b. Direct access to the document object model (DOM) allowed programmers to provide some interactivity to the page
   c. 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
   d. We were addicted to the sweet taste of AJAX interaction … our web pages came to life!
5. The trouble with front-end Javascript
   a. Even though JS is compiled to machine language by the browser, it is compiled to machine language in the browser
   b. To accomplish this, we have to send the source to the browser
   c. We can obfuscate and minify all we want, but those are reversible functions
   d. Any actions taken by the front-end code are exposed
   e. What if your page is making calls to third-party services?
   f. Once the code hits the browser, it’s pretty much out of your control
6. Breaking MVC
   a. If we consider the page to be the view component, and its buttons, forms, and events to be the controller, where is the model?
   b. The short answer: It ends up being spread around
      i. State is held in the page itself
      ii. Changes in state end up being initiated by view code
      iii. The lines between the components are quite blurry
   c. This divergence from a clean architectural model ends up being difficult to test, difficult to maintain, and difficult to secure
7. Node.js
   a. In 2010 a fascinating thing happened: A new framework was released for server-side execution of Javascript
   b. The capability had been around since at least 2000, but it wasn’t widely used
   c. It worked with Google’s V8 Javascript engine, the same engine deployed in browsers, and provided a built-in web server

    d. In that moment, developers could write Javascript on both the front end and the back end
    e. It was now possible to move much of the work from the front end to the back end

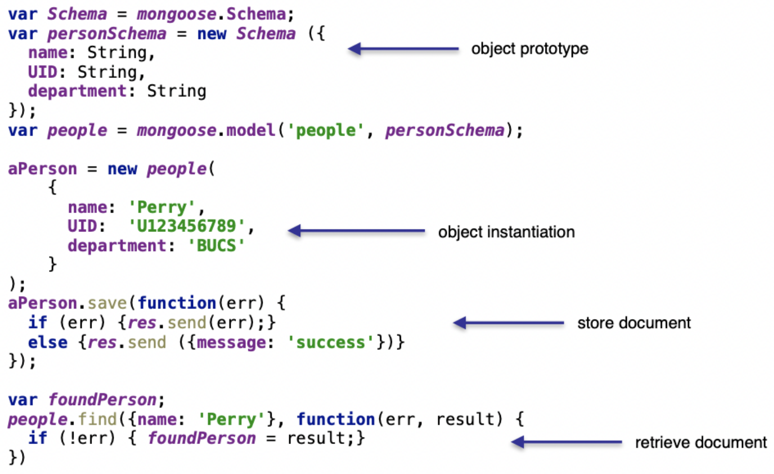
8. Serialization
    a. It gets better…
    b. Javascript uses a lightweight key-value string representation for serialization of objects called JSON (Javascript Object Notation)
    c.



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

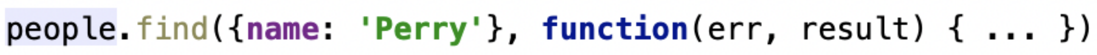
    d. JSON quickly became the transport encoding of choice for moving data back and forth between the front end and the back end

9. Why can't we just store data in JSON?
    a. The answer to that was a flood of non-relational, documentbased data stores that offer CRUD (create, read, update, delete) operations on objects
    b. The blanket term for this class of store is noSQL
    c. One of the more popular is mongoDB
    d. Another is an in-memory database, redis, that provides constant-time CRUD
    e. These are fully denormalized document stores … all relational structures are held in the document
    f.



```
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;}
})
```

10. About those lambdas
    a.

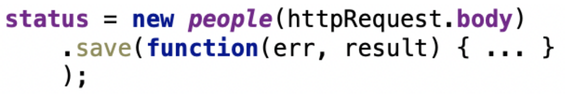


```
people.find({name: 'Perry'}, function(err, result) { ... })
```

    b. The V8 Javascript engine's main function is single-threaded and nonblocking (it's equivalent to a listen() method)
    c. I/O is handled asynchronously by worker threads that take a callback function as an argument
    d. 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
    e. This is similar to the continuation-passing style of programming in languages like Scheme.

11. (A1 stopped here)
    a. 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
    b. It does mean that Javascript programmers must take asynchronicity into account when designing applications
    c. Scope especially becomes important, since functions are being run asynchronously in different contexts

12. 2010: JS transitions to mainstream
    a. 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
    b. 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) { ... }
    );
```

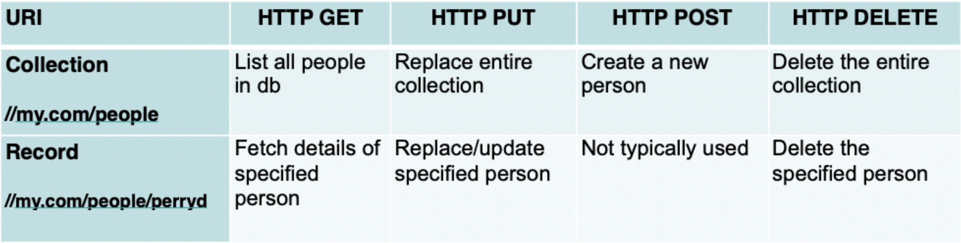
    c.
    d. This full-stack cohesion, along with a highly performant and horizontally scalable platform, has made Javascript extremely popular for web application development
    e. Large corporate deployments include
        i. GoDaddy
        ii. Groupon
        iii. IBM
        iv. Netflix
        v. PayPal
        vi. Walmart

13. REST: Calling the API
    a. Representational State Transfer was described in 2000 by Roy Fielding in his PhD dissertation
    b. It provides a simple way to map HTTP semantics onto CRUD data operations
    c. This is the decoupling mechanism
        i. The client interface is through a URL

    ii. The client doesn't know where requests are being satisfied, just that they are

  d. 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

  e.



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

  f. Create: POST
     Read: GET
     Update: Put
     Delete: DELETE

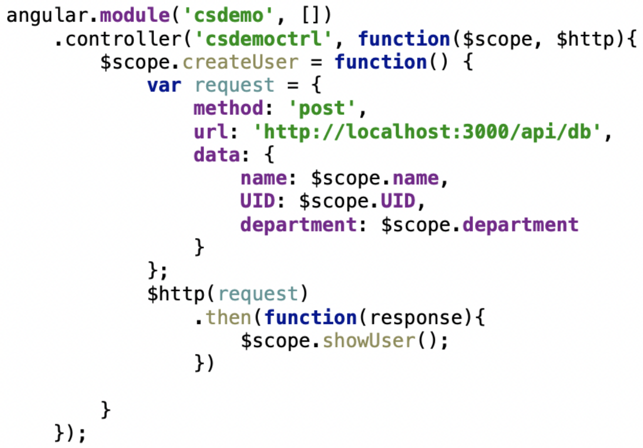
14. $http in Angular

  a. Angular.js is a front-end framework that extends both HTML and Javascript

  b. Just need to instantiate the $http object and fill in the method and any parameters

  c. 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…

  d. `<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();
                })

        }
    });
```

15. URL routing/dispatching

  a. Now that we've defined a RESTful API we need a way to map it to methods on the back end

  b. Routing must take both the URL and the HTTP verb into account

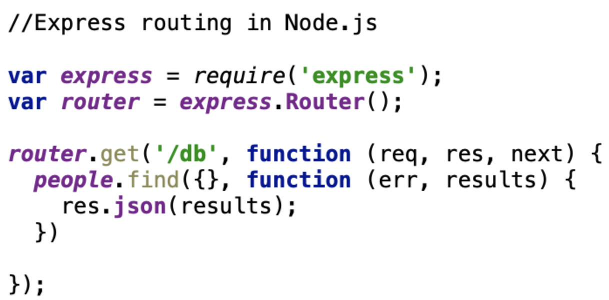
  c. Most languages with web frameworks provide this, either natively or as a library

    i. Python: Flask, Bottle in Django framework

    ii. Javascript: Express.js in Node.js framework

    iii. Ruby: Rails

  d. Routing is done with regexp-like pattern matching

e. 

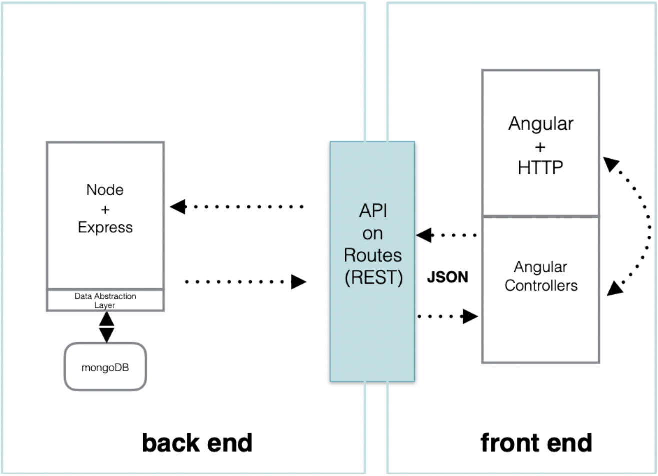
```
//Express routing in Node.js

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

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

});
```

f. We now have all of the pieces to decouple the front end and back end

g. State is held in the model

h. Changes in state are initiated by the controller

i. The view is strictly representational (with some decoration)

j. 

16. Consequences

    a. This sort of RESTful architecture isn’t all puppies and rainbows

    b. We’ve severed the link between the view and model

        i. In the implementation we’ve just seen the controller is responsible for initiating state changes in the model

        ii. However, the controller is also responsible for updating the view

        iii. In classical MVC the view can be independent of the controller

        iv. In fact, there might be many views, not necessarily all in the same application instance

    c. This might not matter, depending on the use case

    d. If it does matter, i.e. a distributed securities trading app, we can use a Observer pattern (publish-subscribe)

        i. With pub-sub, the view would register with the model and be notified immediately when state changes

        ii. This might be a push of new model data or a notification so that the view’s controller can decide what to do

e. For web apps an appropriate implementation is through a web socket
   i. These are full-duplex connections over port TCP:80 (or whatever port the HTTP server listens to
   ii. They are handled as a URI in the form ws://localhost/…

f. It’s fair to argue that if you are using web sockets, REST isn’t necessary

g. 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

17. What have we gained? Security
    a. Sensitive data, including keys and tokens, are held on the back end
    b. We have full control over third-party API calls
    c. We can fully validate inputs passed from the front end
18. What have we gained? Reusability
    a. The back end is just CRUD across one or more data sets
    b. We can reuse these capabilities over and over in new applications
19. What have we gained? Abstraction
    a. From the controller and client view, the model is abstracted
        i. We can place whatever intermediate steps we want in between them
    b. Further, the model itself is abstracted into JSON
        i. We can use any data store, either with native JSON or an adapter
    c. Since the model publishes in a standard form (JSON) it is agnostic of its clients
20. What have we gained? Performance
    a. Data operations are typically hotspots resulting in degraded performance as load increases
    b. By moving to denormalized non-relational data stores we remove much of the overhead required by more traditional RDBs
    c. Additionally, noSQL DBs such as mongoDB are designed to easily scale horizontally (via sharding and clustering)
        i. We can use several low-cost data servers instead of a few high-cost vertically scaled ones
    d. The V8 Javascript engine is optimized to handle small requests at a high level of concurrency
    e. 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
21. What have we gained? Testability
    a. The strong division of responsibility means that we readily test the internal API without use of a front end
    b. We reduce the universe of inputs by rigidly specifying the interface

22. What have we gained? Concurrent development
    a. Since the front end is completely decoupled from the back end, we can work on them simultaneously
    b. The front end can use stubbed API calls while the back end is being completed ▪ Note that this requires firm requirements
23. True MVC
    a. 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
        i. All model operations take place on the back end
        ii. All view and controller operations take place on the front end
        iii. 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
        iv. The model is also treated as an abstraction (using JSON) and so the data store is decoupled