Prerequisites

Goals for this recitation

Introduction: Express.js

Model + View + Controller

Closer look at Express

MongoDB

Openshift

Other Resources

Prerequisites

- Installation instructions completed for Node.js, Express, and MongoDB

Goals for this recitation

1. Understand how to write a web application using Express
2. Understand how Express fits into the MVC design pattern
3. Understand how to interact with MongoDB as a datastore
4. Introduce Openshift as a deployment solution

Express.js

Introduction: Express.js

Express.js is a web application framework for Node.js which provides a robust set of features for building single and multi-page, and hybrid web applications. Node.js as a standalone language, as you've already explored in the Node.js server programming tutorial, is not particularly easy to build web-applications with. Writing complete web applications without using a framework on top of Node would not necessarily be easier than using standalone Python or Ruby to accomplish the same task.

Express was inspired by frameworks like Sinatra for Ruby or Flask for Python -- relatively light. The core functionality of Express is contained in < 1000 lines of code.

Express.js Hello World

Follow the Express.js ‘Hello World’ instructions located at the install guide.
Express is not ...

Now that we’ve seen some very simple Express code and understand what this framework is meant to do, it’s just as important to understand what Express is not meant to do/be.

Rails, Meteor, or Django -- If you are familiar with any of these frameworks you will notice immediately that Express gives you much more freedom to design the structure of your application, and much less hand-holding as well. Be more careful with Express than you would be with these frameworks: with more power comes more responsibility.

Particularly stable -- Express is a framework which is continually being worked on, and as a result, if you find a link explaining a feature in Express or a Stack Overflow answer more than a year old, there’s a possibility that it doesn’t apply to the most recent version of Express (4.0.0)

As widely used as most languages/frameworks you’ve used before -- It can be hard to find debugging help on Google. However, because Express is so lightweight, the hope is that by doing some reading and working with Express for one or two projects, you’ll gain enough understanding of the framework to determine where your bugs and errors are coming from.

Model + View + Controller

The MVC paradigm is an important concept in web programming. Simply put, we seek to separate the presentation (HTML/CSS) from the data (Database, Node.js backend), and only have a small amount of code known as the controller (Node.js) to interface between these two.

This concept is covered pretty thoroughly in 6.005 for a variety of systems, but here is how this paradigm relates to web programming:
Closer look at Express

Setup

Now that we've visited the MVC design pattern, let's take another look at Express. This time, we'll generate a project from scratch:

1) Install and run express-generator to generate a basic project with EJS templating:

   ```
   $ npm install -g express-generator
   $ express -e myapp
   ```

   Note the ‘-e’, which indicates that we want to use EJS (embedded javascript) for our templating language. Express by default uses jade, which is less verbose than HTML. You're free to use Jade if you want, but the course staff hasn't been trained to help you debug Jade.

   Note that this generator will create a bunch of directories and give you a package.json file. A lot of the packages in this file are unneeded, so don't worry if you don't know what everything in there means just yet.

2) Install dependencies:

   ```
   $ cd myapp & & npm install
   ```

   We've now generated a base for an Express application. Now our focus will shift to implementing MVC in our Express application.

Routing

We're trying to build a web application, so we need to have a mapping from URLs and paths to content in our web application. What we will show you is only one way out of many to perform routing in Express, but it gives you a nice separation of modules and is easy to grasp.

To begin let's focus in on these lines, which map paths to content:

```javascript
// Use module routes/users, a router() instance
var users = require('./routes/users');
app.use('/:users', users);
```
// Use module routes/index, a router() instance
var index = require('./routes/index');
app.use('/', index);

As a reminder, a URL can be split into the domain and the path: http://example.com/hello. The routing code above only matches against the path of a URL. Matching and ordering are both important in routing.

In the example above, the path will first be checked against ‘/users’. If there is a match there, it will use the first router. If not, it will be checked against ‘/’. Every path matches against ‘/’, so Express will try to use this second router if the first one did not match. This is known as a catch all and should be placed below more specific routes.

Now, open routes/index up and take a look.

You’ll notice the comment “GET home page” followed by some code. The presence of an HTTP verb indicates that This code adds a route for ‘/’ to the application. This is second-level routing -- whatever prefix is matched in app.js is removed from the path. Open up routes/users. Under the comment “GET users listing.”, notice there is another route for ‘/’, but because this is the next phase of routing the path it matches is ‘/users’.

Further reading:
https://www.packtpub.com/books/content/understanding-express-routes
http://expressjs.com/api.html#router

Controllers

Although routes/index and routes/users contain routing, in Express, they are also the controllers for index and users, respectively.

Going back to routes/index, when this route is visited, the code inside this method is called. In this case, we will render ‘index’ (don’t worry about `` title: ‘Express’ ``` yet).

This line is a call from the controller to a view. More often than not, this is how a controller method will end. The controller performs whatever logic is necessary (none in this case), and renders a view to the user.

Every route in your application will correspond to a controller method. Therefore, in order to add a new page to your application, add a controller method and give it a route:

    /* GET signup page */
router.get('/signup', function(req, res) {
    res.render('signup', {title:"Signup"});
});

Notice that so far we've been using 'router.get' in our examples. You can place any HTTP verb in place of 'get':

    /* POST to signup and create new user */
    router.post('/signup', function(req, res) {
        ...
        ...
    });

You should always use the correct HTTP verb when writing controller routes; if you're ever unsure of what to use, you can refer to this reference here.

Views

Now go to views/index.ejs to see what a view looks like. It should look like standard HTML with one exception that appears three times:

    <%= title %>

This is embedded javascript, or javascript that is evaluated and turned into HTML. Notice the special tags around 'title'. When you navigate to this page in the browser, any EJS tags are evaluated using a javascript interpreter, and the result is displayed as HTML. Read more about how EJS works here.

Now, recall that the controller actually passed the title to this page via the code that I asked you to ignore earlier. This is how dynamic pages are generated: the controller passes information to the view that is different depending on the user and the state of the database. Any javascript object can be passed from controller to view, although it's highly recommended that you only pass data you actually intend to display.

The last piece of our MVC design pattern is the Model. We won’t be covering models in this recitation (so we won’t expect you to separate your models from your controllers in your code) but will be covering them next week.

MongoDB
Introduction to Mongo & NoSQL

MongoDB is a high performance document database. It stands in contrast to traditional databases such as MySQL due to its dynamic schema which means that documents in the same collection do not need to have the same set of fields or structure.

Fig 1. This is an overview of MongoDB structure.

source: http://docs.mongodb.org/manual/_images/crud-insert-stages.png

We’ll be using MongoDB to store persistent state in our application. As we went over in lecture, HTTP is a stateless protocol, so in order to generate different content under different circumstances using HTTP, we’ll be storing and accessing documents in MongoDB.

There is tons of literature on whether SQL or databases similar to MongoDB (known as NoSQL) are “better”. However, the decision to use Mongo has been made for us because the Node.js and Express community have defaulted to using Mongo.

One major way this affects your projects is that you must be careful to preserve consistent representations of your models. Let’s say you have a collection named ‘usercollection’ in MongoDB, where you store all users of your application. Once you’ve been working on your app for a while, you decide that you should store users’ email addresses. However, all users who are currently in ‘usercollection’ don’t have email addresses stored for them. Mongo lets you insert email addresses for users you create in the future, but ‘usercollection’ will then be in a state where some users have emails and other don’t. This can be dangerous.

Interfacing between Express & MongoDB
There are a number of Node.js modules built to interface with MongoDB. The most popular ones are Node-mongodb-native, Monk, and Mongoose. The native driver is a direct interface with Mongo while Mongoose is a full-fledged ORM (object relational mapper). Monk falls somewhere in the middle and is the one we’ll be using in this recitation, although you are free to use any one you wish to.

Before you can interface between Express & Mongo, you’ll need to make sure MongoDB is fully set up according to the installation instructions.

In order to connect to MongoDB using Monk:

```javascript
var db = require('monk')('localhost:27017/mydb')
```

Where ‘mydb’ is the name of your database. Once you have a connection to the database, you can get a collection with the following:

```javascript
var users = db.get('usercollection');
```

You now have a connection to the collection ‘usercollection’ under the variable name ‘users’. Note that it’s not the collection itself but an object that allows you to connect to the collection.

You will hear the term “asynchronous callbacks” mentioned in relation to Node.js often when working with Mongo. What this means is that when you interface with the database, a query is sent to the database, and then your application will do something else until the database responds. This is what a query looks like:

```javascript
users.find({ 'name': 'Evan' }, function (err, users){
  // do something with users here
});
```

You pass in a callback function as an argument to your query and once the database responds, the callback function is executed on the response.

In order to perform more complicated queries, you can chain together additional terms. Here is an example:

```javascript
user.find({
  "email": {$in : emails},
  "sort": [['_id', -1]]
}, function(e, docs) {
  ...
});
```

In general, reading through the [Mongo docs](https://mongodb.com) will give you a good idea of how to write queries.
For a complete overview of how to add Mongo to an application, follow the Mongo instructions in part 3 of this tutorial: [http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/](http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/)

**Openshift**

Follow the [OpenShift instructions](http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/) to deploy the sample application you’ve made.

**Other Resources**

Additional useful resources for Node.js and Express can be found on [Stellar](http). [Nodebeginner](http) is particularly useful for understanding many of the quirks of Node.js.