Web and Mobile Application Development

Server Side Intro & Intro to Node.js

Material created by:
David Augenblick, Bill Mongan, Dan Ziegler, Samantha Bewley, and Matt Burlick

Intro

- Technically, everything we've done thus far has been without a server!
  - The browser, instead of making an HTTP request to the server, reads in the HTML file
  - And reads in additional files as necessary (images, scripts, etc..)
- However to allow others to get/request files from our computer, we need a server!

Intro

- Recall when we talked about HTTP in week 1:
  - A client (i.e browser) makes an HTTP request to a server
  - The server processes the request, and if necessary, gets the requested data and returns it.
- The HTTP protocol allows servers to be in whatever language they like; they just have to return data in the requested format.
- For this course we will use Nodejs as our server.
  - Is both a web server and a server-side language
  - Built upon JavaScript so there's not much more to learn!

What is Nodejs?

- Node.js is a cross compatible server platform
- It's based on JavaScript, so it's easily accessible for developers who don't want to learn an entirely new language to get started in web app development (i.e. PHP, Java, etc.).
- It contains built in **modules** such as an HTTP server library and 3rd party modules (like `express`) to facilitate http based communication and easy to use development features respectively
- It is based on an asynchronous, event driven, non blocking I/O model architecture, it handles throughput in a very efficient manner.

Creating/Running a Node.js Server

- The basic idea of creating a Node server is as follows.
- Create a script that
    1. Creates a server object
    2. Defines behavior for the server (functions) for different events
    3. Starts the server listening to a port
- Then start up your server (run your script) by typing in the command line

```
node <myfile.js>
```

- Unfortunately if you make changes to your app, you must kill the server process and the run it again.

A Basic Node.js Server

- So let's make a script and create a server object in it.
- As mentioned, node has *modules* which are optional features we can load.
- NodeJS's native module to create a web server is the `http` module.
- But the popular `express` module is a wrapper for this that makes things a lot easier, so let's use it.

NPM Modules

- What's a module?
- Typically it's a collection of functions and/or classes
- There's a huge collection of existing ones (we'll see how to make our own later) located at a central repository!
- Most installations of node.js include the Node Package Manager (`npm`) that allows us to easily get and install modules from this central repository.
- To install a module using the node package manager all you need to do is type the following command!
  - `npm install <module_name>`
- For example, to install the express module, open a command line prompt and type:
  - `npm install express`

A Basic Node.js Server

- Now that we have express installed we can require it in our scripts.
- Once required, we can then set up an app/server by
  - Creating an express app

```
var app = express()
```

  - Then to start this app/server we just have it listen to a port

```
app.listen(8080);
```

- The last step in setting up a basic web server is to tell it where to serve files from.
- To do that we tell our application to "use" a feature, in particular a static path

```
app.use(express.static(_PATH));
```

  - Where `_PATH` is the path (some string, typically relative to the server script) that we want to serve files from

A Basic Node.js Example

- Now let's start up this server script: `node <file.js>`
- To have NodeJS serve up the content we must tell the browser to make an HTTP request to `<serverpath>:<port>/<filename>`
    - For our local application `<serverpath>` is localhost and `<port>` is whatever we told our express server to listen to
- Here's what happens….
    1. HTTP request made from browser to `<serverpath>:<port>/<filename>`
    2. Server sees request for a file, reads in file and places its content in its response.
    3. Browser gets response, parses to create DOM (requesting additional files as needed)

Warning!!! From now on request html from your server, DO NOT open the html file to your file browser

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

app.use(express.static("."));
app.listen(8080,function(){
      //whatever
});
```

Node Console

- For debugging purposes it might be nice to print stuff out to the command line.
- To do this you can just send strings to the console:

```
console.log('…..');
```

- We can bind a function to get run when the server starts listening:

```
server.listen(8080, function(){
    console.log('Server started...');
});
```

Dynamic Pages

- Ok so we can serve up static pages/files.
- But to make interesting web pages (or web services) we need to create content on-the-fly.
- Often this content will include information from other web services and/or a database.
- To differentiate these requests from static content requests, the paths will look like *actions* (i.e verbs)
  - http://localhost:8080/listusers
  - http://localhost:8080/login
  - http://localhost:8080/greet

Dynamic Pages

- One strength of NodeJS is that it's designed to be asynchronous and event-driven.
- In a later slide deck we'll look at how to have our code "emit" events and to catch those emissions.
- Many of the built-in classes/modules emit things that our server can catch.
- Some such events/emissions are the `get` and `post` events from our web server.

Dynamic Pages

- Recall from early in the course that there are typically to types of request made:
  - GET – Typically to get stuff from a server
  - POST – Typically to post stuff to a server
- Using the express module, we can specify how to react to each of these "dynamic action requests" for both types of requests. We'll call this *binding actions* to our app.
- We can then *write* stuff to the response
  - In particular, we'll use the `write` and `end` methods of the response object.

Dynamic Pages

- To bind an action to our app we just put:

```
<express_app_var>.<get|post>(<action_path>,function(req,res){
      //what to do
});
```

- Thinking emission/event wise we can say the `express` module can emit events `get` and `post` and our code binds functions to run when those emissions are caught
- Here's some examples!

```
app.get('/greet', function(req,res){
        res.write('Hello World');
        res.end();
});
app.get('/list_users', function(req,res){
        res.write('User List');
        res.end();
});
app.post('/login', function (req, res){
        res.write('You logged in.');
        res.edu();
});
```