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

- The job of a server is to, well, serve data
- 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.
- Requests to the server may include things like:
  - HTML
  - JSON data
  - Etc..
- 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.

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 `express` module (to name a few) 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.
- One of the most important one for web applications is the *http* module.
- So let's start by loading it:

```
// needed to be able to fulfill HTTP requests
var http = require('http');
```

A Basic Node.js Server

- Now we can create an HTTP server.
- To do this we call the `createServer` method on the http object
- This method takes a function as parameter that will get called when HTTP requests come in
  - This function has two parameters
    - `request` – The request that came in
    - `response` – The response we want to give
- Our HTTP response should include stuff like
  - Header information
    - HTTP status
    - Content type
  - Then whatever we want to say
  - Remember how we said this would be important in week 1!

A Basic Node.js Server

- Here we go

```
// needed to be able to fulfill HTTP requests
var http = require('http');
// create the instance of a server
var server = http.createServer(function(req, res) {
    // Write some header information
    // In particular state that the HTTP status was ok (200)
    res.writeHead(200);
    // now use the end method to add HTML to our response
    res.end('Hello Http');
});
```

A Basic Node.js Server

- Finally we tell our server to listen to a port.

```
server.listen(8080);
```

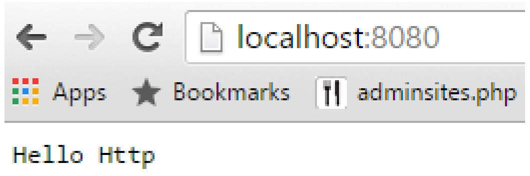
A Basic Node.js Example

- Now let's start up this server script: `node <file.js>`
- If your running this server locally than any requests to http://localhost:8080 will
    - Call the function defined when we created the http server
    - Pass in the request information
    - Populate the response information
    - Return the response information to the browser

```
// needed to be able to fulfill HTTP requests
var http = require('http');

// create the instance of a server
var server = http.createServer(function(req, res) {
  // return OK status (200)
  res.writeHead(200, {content-type: 'text/plain'});
  // message to provide to user (in browser)
  res.end('Hello Http\n');
});

// port upon which the server will listen for incoming traffic
// for example:  http://localhost:8080
server.listen(8080);
```



```
Hello Http
```

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...');
});
```

Serving Static Pages

- Well this isn’t a very interesting server!
    - It just sends the response “Hello HTTP” no matter what “page” we request.
- Ok well we could have a branching statement in there to return different things based on the request.
    - But that seems exhausting.
    - Not to mention we might have several static “.html” pages which link to images, css files, etc.. on the server
        - So we’d have to handle all these potentially recursive requests and different file types.
- Another idea would be to use NodeJS’s file system module to read in a file and write it out to the response

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.
- So to install a module all you need to do is type the following command!
  - `npm install <module_name> –save`

Serving Static Pages

- Back to reading in files….
- The file system module is called `fs` and has a function called `readFileSync`
- So lets first install it via the command line:

```
npm install fs –save
```

- Now lets use it!

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(req,res){
     var data = fs.readFileSync('./public/'+req.url,'utf8');
     res.end(data);
});
```

Serving Static Pages

- But the problem with this approach is all different types of requests may come in (text, html, css, images, video, etc..) and then we'd need to populate the correct response **header information** based on the request for this to all work well.
- Fortunately the extremely common NodeJS *express* module can make this (and a lot of other things!) easier for us ☺
- To install the express module

```
npm install express –save
```

Serving Static Pages with Express

- 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);
    ```
- So how do we serve up pages/content?
- We can tell the server what the path is to the pages to serve via:

  ```
  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

Serving Static Pages with Express

- Putting it all together we’re our static file server.

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

app.use(express.static("."));
app.listen(8080,function(){
    console.log(‘Server Running…’);
});
```

Dynamic Pages

- Ok so we can serve up static pages.
- But to make interesting web pages (or web services) we need to create *dynamic* content.
- 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 the next 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.

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