Setting up a Web Server

- Up to this point in the course, we've been relying on the Apache Web server installed on mathlab to serve the files that make up our Web-apps
- Now it's time to take the next step, by setting up a personal server, that will implement your app's server-side logic
- Place the following JavaScript code in file **app.js**:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200,{'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(port_#, 'mathlab.utsc.utoronto.ca');
console.log('Server running at
      http://mathlab.utsc.utoronto.ca:port_#/');
```

Basic Web Server Operation

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200,{'Content-Type':'text/plain'});
  res.end('Hello World\n');
}).listen(port_#, 'mathlab.utsc.utoronto.ca');
console.log('Server running at
      http://mathlab.utsc.utoronto.ca:port_#/');
```

- Let's look at what this code does …
- First it loads the HTTP library, then creates a Web (HTTP) server with **`createServer()`**
- Parameter to **`createServer()`** is a callback function, that is invoked whenever a client connects to this server
- The server is then instructed to listen for incoming connections on mathlab at the designated port_# (which should be set from one of the port #'s reserved for you and listed on the Assignment Web page (do not use other #'s!)

Basic Web Server Operation

- You should be getting familiar with callback functions for client-side events (e.g. backbone destroy callback navigates to browse), but what does a callback mean on the server?
- Whenever the server receives a request on the designated port #, it invokes the callback function and passes it 2 objects as parameters: request-data "req" and response-data "res".
- "req" is an object that packages request information, such as URL-path requested, HTTP method, and parameter values.
- The "res" object is created using methods **`writeHead`** and **`end`**. **`writeHead`** sends an HTTP response header, and **`end`** sends an HTTP message body, which for the example above is just the text "Hello World".

Basic Web Server Limitations

- The problem with the HTTP module is that operates at a very low level of abstraction
- It enables you to interact directly with HTTP request and response objects, but it doesn't give you any higher-level abstractions to streamline build Web apps
- Among the problems when trying to build the server-side of your app using just the HTTP library are the following:
  - only a single callback function can be specified
  - request URL has to be extracted from the from the req object and then pattern-matched against a list of available URL's supported by your server
  - response headers have to be created manually

Web Frameworks

- A Web framework is a collection of tools and libraries designed to support building Web applications.
- Example server-side frameworks are Java Spring/JSF/Struts, Ruby on Rails, PHP Zend, and Python Django
- A Web framework simplifies app development by providing briefer and/or simplified ways to implement common functionality; for example: routing, templating, sessions, etc.
- A Web micro-framework provides a more minimalistic set of tools. Examples include Ruby Sinatra and Python Flask.
- Node.js has its own popular micro-framework called Express, which will be using.

Express Micro-Framework

- Let's revisit our Hello-World example to see how we would implement it using Express:

```
var express = require("express");
var app = express();  // define a new app
// register a route and its handler (function)
app.get("/", function(req, res) {
    res.send("Hello world!");
})
var server = app.listen(port_#);
console.log('Express server listening on port %d',
                server.address().port);
```

- Is this better than our first version?

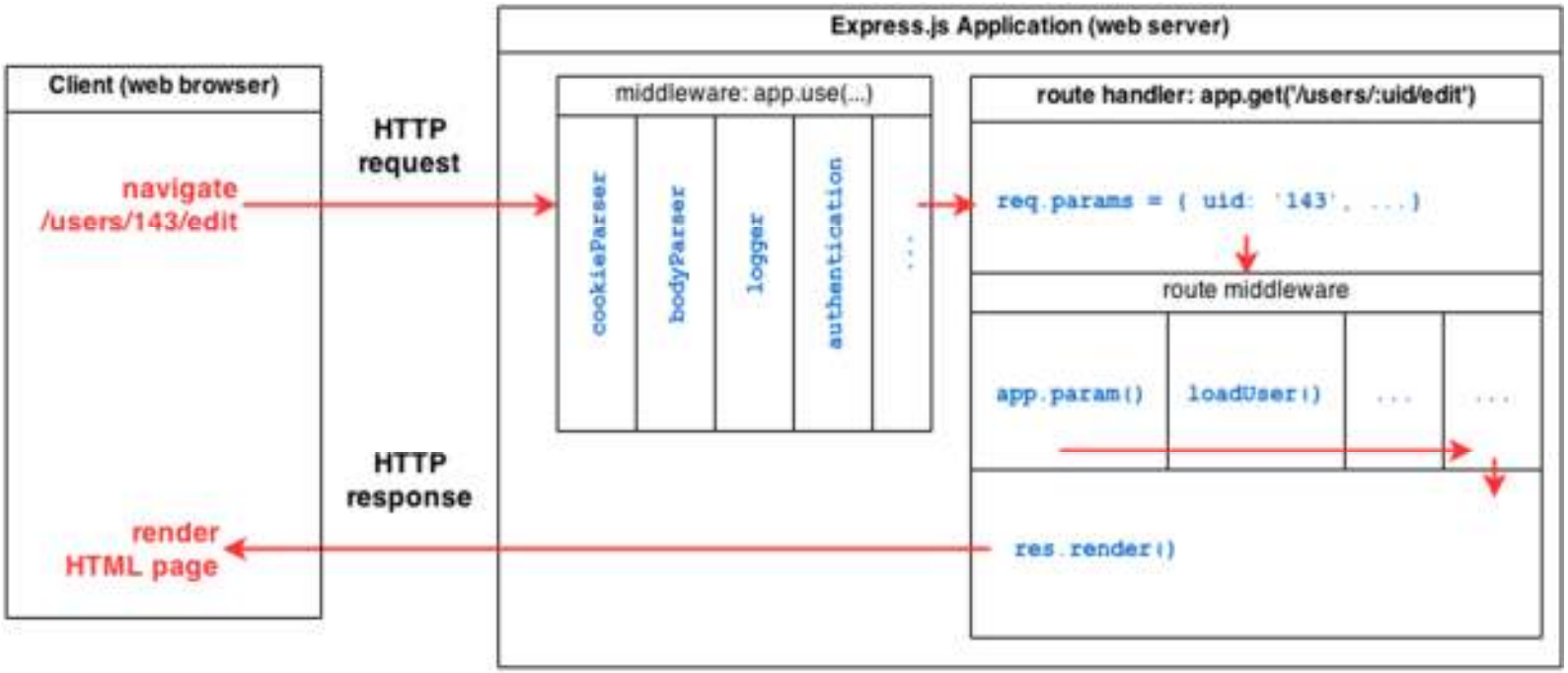
Express Micro-Framework

```
var express = require("express");
var app = express();  // define a new app
app.get("/", function(req, res) { // "route"
        res.send("Hello world!");
        res.end();
}).listen(port_#);
console.log('Server running at
   http://mathlab.utsc.utoronto.ca:port_#/');
```

- What's going on here?
  - first we load the Express module and create an application.
  - next we defined a "route" (URL "/" here), and tell the server to listen for incoming connections
- Is this better than our first version?
- It isn't any shorter, but it does have some advantages:
  - we can tell our handler to only accept GET-method requests on the "/" route
  - we eliminate the need to output HTTP response headers

Express Middleware

- Some of the code in the **`app.js`** starter is classified as "middleware", e.g. **`app.use(…)`**



Express Middleware

- Route middleware behaves like a chained sequence of actions on a route
- Middleware addresses "cross-cutting" issues – that is tasks that are relevant to many/most requests and don't belong in any specific request
- What kinds of tasks does middleware handle?
  - logging
  - parsing the body of an http request (e.g. to extract uploaded data)
  - serving static files
  - session handling
  - error handling

Middleware Processing

- Middleware plugins are processed in the order of their declaration; effectively earlier plugins have “precedence”
- When a request arrives at the server, it is handed off to the 1st middleware function, which can call methods on the response object and/or hand off the request to the next middleware or request handler
- You can specify a path prefix when adding middleware, and the middleware will only be asked to handle requests that match the prefix path, e.g.

```
app.use('/prefix', middleware)
```

Middleware: Static Files

- Although our main interest in Express is to develop the dynamic server-side of Web apps, it is also capable of serving "static" content such as HTML, CSS, and JS files.

```
app.use(express.static(__dirname + "/public"));
```

- This tells Express to use **`__dirname/public`** as the directory from which to serve files. **`__dirname`** is the location of the server's executing script (e.g. **`app.js`**)

- If the server receives a request that does not match any of its routes, but matches a file in that directory, Express will respond by sending that file's content

Starter-Code Middleware

```
app.configure(function() {   // global configuration
    // use PORT enviro variable, or local config file value
    app.set('port', process.env.PORT || config.port);
    // change param value to control level of logging
    app.use(express.logger('...'));
    // use compression to reduce size of HTTP responses
    app.use(express.compress());
    // parses HTTP request-body and populates req.body
    app.use(express.bodyParser({
        uploadDir: __dirname + '/public/img/uploads',
        keepExtensions: true
    }));
    app.use(app.router);
    app.use(express.static(__dirname + "/public"));
    app.use(express.errorHandler({dumpExceptions:true, … }));
});
```

Express: Route Handlers

- Route handlers (sometimes called request handlers) are where the app's server-side "business logic" is implemented
- In the assignment, we're parceling these off into a separate file, **`routes/eatz.js`** to keep this app logic segregated from the basic HTTP server with its routing-middleware-server functionality
- By default, a route handler is passed "req" (request) and "res" (response) object parameters, which provide access to the client request and the to-be generated response

Express Route Handlers

- Real Express servers usually define multiple routes, that select between different possible behaviors/outputs. For example, we could refine the HelloWorld example as follows:

```
app.get("/sp", function(req, res) {
    res.send("Hola Mundo!");
    res.end();
});
app.get("/fr", function(req, res) {
    res.send("Bonjour tout le Monde!");
    res.end();
});
```

- HTTP methods other than GET are also used, and the same route-URL can be associated with different methods

Express Route Handlers

- You can access request query-parameters in callback functions using the **`req.query`** object. For example:

```
app.get("/", function(req, res) {
    res.send("Hello, " + req.query.name + "!");
    res.end();
```

If you load the above using Node (with the necessary requires declarations, e.g. for Express), and then visit URL:

`http://mathlab.utsc.utoronto.ca:port_#/?name=Elson`

the response page will include the name parameter value

Express: Route-Handler API

- **`req.send()`** is the workhorse than handles sending a prepared response to the client
- **`req.get()/set()`** are used to retrieve/set HTTP header fields, e.g. **`req.get('content-type')`**
- **`req.params()`** is a hash of the parsed URL parameters along the entire path, e.g. req.params.id (id param value)
- **`req.query()`** is a hash of the parsed query-string
- **`req.body()`** is a hash of the request body
- **`req.route()`** provides detailed info about the route (same object as **`app.routes`**)
- **`req.accepts(type), req.accepted()`** are used to check the media types accepted by the requesting client

Starter-Code Route Handlers (app.js)

```
// App routes (API)
// route-handlers implemented in routes/eatz.js

// Heartbeat test of server API
app.get('/', eatz.api);

// Retrieve single dish by its id attribute
app.get('/dishes/:id', eatz.getDish);

// Upload and process image file
app.post('/dishes/image', eatz.uploadImage);

// other routes listed on assignment handout
```

Starter-Code Route Handlers (routes/eatz.js)

```
// heartbeat response for server API
exports.api = function(req, res){
  res.send(200, '<h3>Eatz API is running!</h3>');
};
// retrieve individual dish model, using id as DB key
exports.getDish = function(req, res){
  DishModel.findById(req.params.id, function(err, dish)
    {
      if (err) {
         res.send(500, "Sorry, can't retrieve");
      } else if (!dish) {
         res.send(404, "Sorry, dish doesn't exist");
      } else { res.send(200, dish); }  // JSON result
  });
};
```

Asynchronous Functions

- Remember these from JavaScript and jQuery? (hopefully you're getting practice with them on the assignments)
- Callbacks are the last argument passed to an asynchronous function, and are themselves functions
- When an asynchronous function completes its task, it hands off control to the callback function
- By convention the first parameter passed to callback functions is an error object, that details what went wrong in the case when the asynchronous function failed

Asynchronous Example

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

// synchronous version ============================
var data = fs.readFileSync('example.file','utf8');
console.log(data);
// do other stuff

// asynchronous version ============================
fs.readFile('example.file', 'utf8',
   function (err, data){
      if (err) {
         return console.log(err);
      }
      console.log(data);
});
// do other stuff
```

Which version is better? Why?

Where's the callback?

Asynchronous Callbacks

- When we looked at an example of jQuery's Ajax callback, I mentioned that any code that needs to reference the Ajax result must be inside the callback, not after the callback.
- Why is that the case?

- Here's another tricky case:

```
for (var i = 0; i < 5; i++) {
  setTimeout(function () {
    console.log(i);
  }, i);
}
```

Example: **async_loop_closure.js**

Asynchronous Callbacks

- What's different here?

```
for (var i = 0; i < 5; i++) {
   (function(i) {
      setTimeout(function () {
         console.log(i);
      }, i);
   })(i);  // note immediate call of func
};
```

Example: **async_loop_closure.js**

Express Sessions: Tracking State

- We looked at 3 mechanisms that support state-tracking between the client and server parts of an app (URL-rewriting, hidden form fields, HTTP cookie headers)
- On their own, these are not very convenient for developers: low-level, too much code needed to track of all the details
- Express provides a session middleware capability that streamlines creation of sessions, e.g.:

```
// in middleware section of router app.js
app.use(express.cookieParser());
app.use(express.session());

// in route-handler function(req, res)
var session = req.session;
```

Express Sessions: Tracking State

- You can store data in a session object by simply "dotting it":

```
// set the session username/auth attributes
req.session.username = req.param('username')
req.session.auth = true;  // login

// check the session auth attribute
if (req.session.auth) { … user logged in … }
```

- Sessions are server-side objects – not accessible to client-side code
- Client queries server if it needs information from session

Express Sessions: Tracking State

- Express sessions can be customized by configuring them with:
  - a “key” that will name the cookies (makes them easier to locate in developer tools),
  - a “secret” that protects the session cookie against tampering
  - a cookie timeout, that will cause the session to expire
  - other parameters for added security … maybe later

```
app.use(express.session({
      key: 'eatz.sess',
      secret:'xyz',
      cookie: {
          maxAge: … // in milliseconds
      }
}));
```

Authentication with Express Sessions

```
exports.signup = function(req, res) {
  var user = new User(… where's the data? …);
  // generate salt value for new user
  bcrypt.genSalt(10, function(err, salt) {
    // hash user's plaintext with salt
    bcrypt.hash(user.password, salt,
                              function(err, hash) {
      user.password = … need hash+salt …
      // … create record for user in DB
      user.save(function (err, result) {
      // … return {username, userid} to client
      // … handle assorted error conditions …
    });
  });
};
```

Authentication with Express Sessions

```
exports.auth = function(req, res) {
  if (… is this a login request…) {
    var username = … get username …;
    var password = … get password …;
    if (!username || !password) {
      … oops, that's an error …
    };
    User.findOne({… search field …},
                              function(err, user){
      if (… we found the user …) {
        bcrypt.compare(password,… model field…,
                            function(err, result) {
          if (result) { // username-password OK
            … continued on next slide …
```

Authentication with Express Sessions

```
… continued from previous slide …
    if (result) {  // valid username-password
      req.session.auth = … ;  // logged in
      req.session.username = …;
      req.session.userid = …;
      // extend session life if remember-me set
      if (… user said remember-me …) {
         req.session.cookie.maxAge = … } else …
       res.send({ … userid, username …});
    } else … handle various error conditions …

 } else { // logout request
    req.session.auth = … ;  // and other fields
    res.send({ … userid, username … });
};
```

Example:
async_loop_closure.js

Sessions: a visit counter

```
var express = require('express');
var app = express();
app.use(express.cookieParser());
// cookie with secret value, and session timeout in ms
app.use(express.session({secret: "xyz",
                         cookie: {maxAge: 10 * 1000}}));
app.use(app.router);
app.get('/', function(req, res) {
    var sess = req.session;
    sess.visits = sess.visits || 0;  // session data
    sess.visits++;
    res.json({visits: sess.visits});
});
app.listen(8080);
```