Emerging Technologies

COMP-308 Winter 2018

Lesson 3 Review

- **JavaScript Closures**
  - The function has access to **parent function scope** after the parent function has closed.
  - Used as callback functions in JavaScript event-driven model
- **JavaScript event loop**
  - **Call stack** to handle function calls
  - **Message queue** to handle events
  - **Callback functions**
- JavaScript is **non-blocking**, **single-threaded**
  - I/O operations called **asynchronously**
  - JavaScript is **non-blocking**, **single-threaded**
- **Node.js**
  - Uses "**single-threaded event loop model**" architecture to handle multiple concurrent clients
  - Uses JavaScript **asynchronous** behavior for I/O operations
- **CommonJS**
  - **require** method
  - **exports** object
  - **module.exports** object

Lesson 3 Review

- **Connect module**
  - It wraps the **Server**, **ServerRequest**, and **ServerResponse** objects of node.js' standard **http** module
  - Connect **middleware** are **callback functions**, which get executed when an HTTP request occurs
  - Perform some logic, **return a response**, or **call the next registered middleware**
  - Take three arguments:
    - req
    - res
    - next

- **Mounting Connect middleware**
  - determine which request path is required for the middleware function to get executed
  - done by adding the path argument to the **app.use**() method

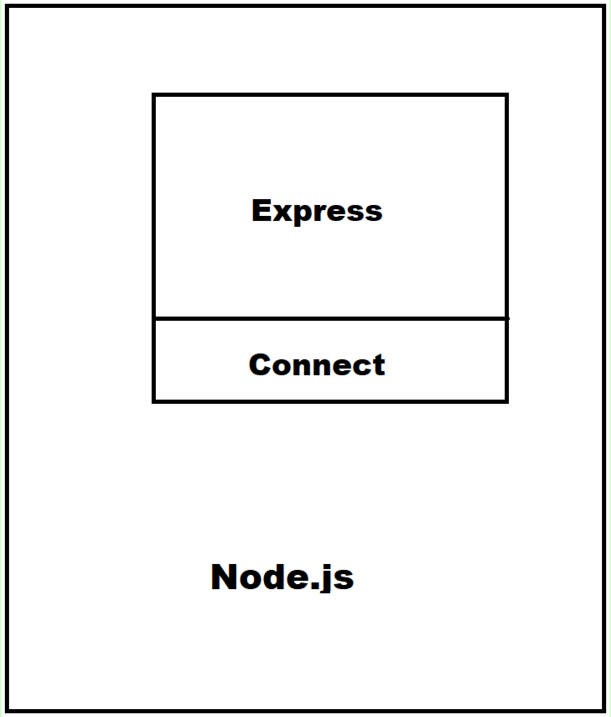
Building an Express Web Application

Objectives:

- ❑ Explain Express
- ❑ Create a new Express Application
- ❑ Configure the Express application
- ❑ Implement MVC pattern

Intro to Express

- TJ Holowaychuk created **Express Framework**.
- Express is a small set of common web application features.
- It is **built on top of Connect** and makes use of its middleware architecture.
- Extends **Connect**:
  - includes **modular HTML template engines**
  - extends the **response object** to support various **data format outputs**, a **routing system**, and much more.



Installing Express

- Create a new **working folder** and a new **package.json file** inside it, which contains the following code snippet:

```
{
"name" : "MEAN",
"version" : "0.0.3",
    "dependencies" : {
    "express" : "4.14.0"
    }
}
```

- *package.json* file, has **three properties**:
  - **name** of your application
  - **version** of your application
  - **dependencies** property that defines what modules should be installed before your application can run

Installing Express

- You can install your application dependencies, by navigating to your application folder, and then issue the following command:

```
npm install
```

- In **Visual Studio 2017**, right click on the project and open **Node.js Interactive Window**.
- Use the following command:

```
.npm install express
```

- You can now create your first Express application by adding your already familiar **server.js** file.

First Express Application

- ❑ Put the following code in *server.js* file:

```
//require and create a new express app
const express = require('express');
const app = express();
//mount a middleware function with a specific path
app.use('/', function(req, res) {
        res.send('Hello World'); //send the response back
});
app.listen(3000); //this app listens to port 3000
console.log('Server running at http://localhost:3000/');
module.exports = app; //returns the application object
```

- ❑ Run your application, by executing **node server**
- ❑ Test it by visiting http://localhost:3000 in your browser.

The application object

- Contains the following methods to help you configure your application:
  - **app.set(name, value)** - used to **set environment variables** that Express will use in its configuration.
  - **app.get(name)** - used to **get environment variables** that Express is using in its configuration.
  - **app.engine**(ext, callback) - used to **define a given template engine** to render certain file types, for example, you can tell the EJS template engine to use HTML files as templates like this:

    app.engine('html', require('ejs').renderFile)
  - **app.locals** - used to **send application-level variables** to all rendered templates.

EJS
embeded JavaScript

The application object

- **app.use**([path], callback) - used to **create an Express middleware to handle HTTP requests sent to the server**.
  - Optionally, you'll be able to mount middleware to respond to certain paths.
- **app.VERB**(path, [callback...], callback) - used to **define one or more middleware functions to respond to HTTP requests** made to a certain path in conjunction with the HTTP verb declared.
  - For instance, when you want to respond to requests that are using the GET verb, then you can just assign the middleware using the **app.get()** method.
  - For POST requests you'll use **app.post()**, and so on.

The application object

- **app.route**(path).VERB([callback...], callback) - used to **define one or more middleware functions to respond to HTTP requests** made to a certain unified path in conjunction with multiple HTTP verbs.
  - For instance, when you want to respond to requests that are using the GET and POST verbs, you can just assign the appropriate middleware functions using

    app.**route**(path).**get**(callback).**post**(callback)
- **app.param**([name], callback) - used to **attach a certain functionality to any request** made to a path that includes a certain routing parameter.
  - For instance, you can map logic to any request that includes the userId parameter using:

    app.**param**('userId', callback)

The request object

- Contains methods and properties that provide **information about the current HTTP request**:
  - **req.query** - an object containing the **parsed query-string parameters**.
  - **req.params** - an object containing the **parsed routing parameters**.
  - **req.body** - an object used to retrieve the **parsed request body**. This property is included in the bodyParser() middleware.
  - **req.param**(name) - used to **retrieve a value of a request parameter**.
    - Note that the parameter can be a **query-string parameter**, a **routing parameter**, or a **property** from a JSON request body.

The request object

- **req.path** - used to retrieve the **current request path**
- **req.host** - used to retrieve the **current host name**
- **req.ip** - used to retrieve the **current remote IP**.
- **req.cookies** - used in conjunction with the **cookieParser()** middleware to retrieve the **cookies sent by the user-agent**.

The response object

- The response object is frequently used when developing an Express application because **any request sent to the server will be handled and responded using the response object methods:**
  - **res.status**(code): used to **set the response HTTP status code**.
  - **res.set**(field, [value]): used to **set the response HTTP header**.
  - **res.cookie**(name, value, [options]): used to **set a response cookie**. The options argument is used to pass an object defining common cookie configuration, such as the **maxAge** property.
  - **res.redirect**([status], url): used to **redirect the request to a given URL**. Note that you can add an HTTP status code to the response. When not passing a status code, it will be defaulted to 302 Found.
  - **res.send**([body|status], [body]): **used for non-streaming responses**. This method does a lot of background work, such as **setting the Content-Type and Content-Length** headers, and responding with the proper cache headers.
  - **res.json**([status|body], [body]): identical to the res.send() method when **sending an object or array**. Most of the times, it is used as syntactic sugar, but sometimes you may need to use it to force a JSON response to non-objects, such as null or undefined.
  - **res.render**(view, [locals], callback): used to **render a view and send an HTML response**.

External middleware

- Extend Express to provide a better framework support.
- The popular Express middleware are as follows:
  - **morgan**: an HTTP request **logger** middleware.
  - **body-parser**: a body-parsing middleware that is used to **parse the request body**, and it supports various request types.
  - **method-override**: This is a middleware that provides **HTTP verb support** such as PUT or DELETE in places where the client doesn't support it.
  - **Compression**: a compression middleware that is used to **compress the response data** using gzip/deflate.
  - **express.static**: used to **serve static files**.
  - **cookie-parser**: a cookie-parsing middleware that **populates the req.cookies object**.
  - **Session**: a session middleware used to **support persistent sessions**.

Implementing the MVC pattern

- Applying the MVC pattern to your Express application means that you can **create specific folders where you place your JavaScript files** in a certain logical order.
- All those files are basically CommonJS modules that **function as logical units**. For instance:
  - **models** will be **CommonJS modules** containing a definition of Mongoose models placed in the **models folder**.
  - **views** will be **HTML or other template files** placed in the **views folder**.
  - **controllers** will be CommonJS modules with functional methods placed in the **controllers folder**.

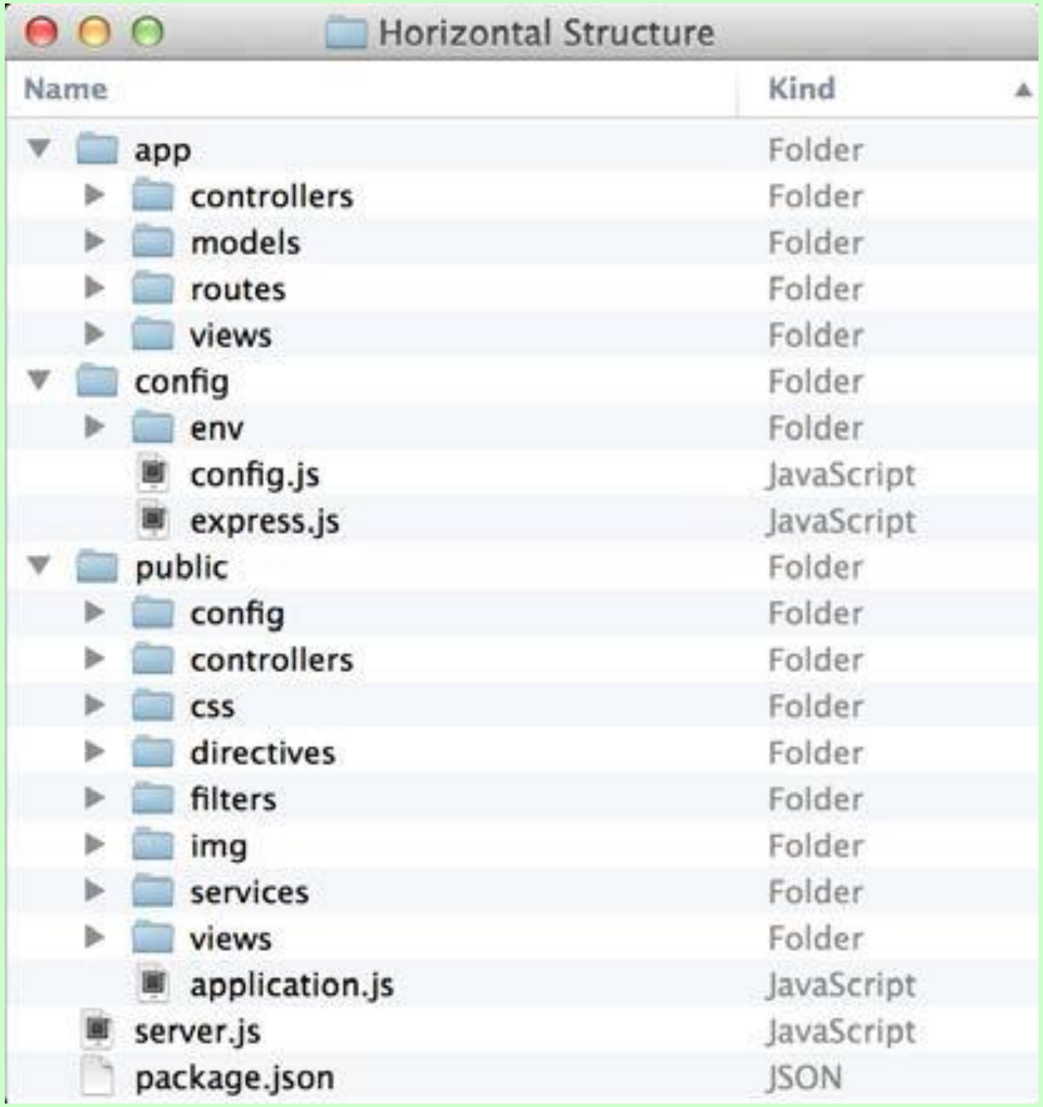
Application Structure

- MEAN stack can be used to build all sorts of applications that vary in size and complexity.
- This allows to handle the project structure in various ways.
  - **Simple projects** may require a leaner folder structure – **horizontal structure**.
  - **Complex projects** will often require a more complex structure and a better breakdown of the logic since it will include many features and a bigger team working on the project – **vertical structure**.

Horizontal folder structure

- Is based on the **division of folders and files by their functional role** rather than by the feature they implement.
  - All the application files are placed inside a main application folder that contains an MVC folder structure. This also means that there is:
    - a single **controllers** folder that contains all of the application controllers
    - a single **models** folder that contains all of the application models.
    - A single **views** folder that contain all of the application views, and so on.

Horizontal folder structure


Horizontal Structure

Name	Kind
▼ app	Folder
▶ controllers	Folder
▶ models	Folder
▶ routes	Folder
▶ views	Folder
▼ config	Folder
▶ env	Folder
config.js	JavaScript
express.js	JavaScript
▼ public	Folder
▶ config	Folder
▶ controllers	Folder
▶ css	Folder
▶ directives	Folder
▶ filters	Folder
▶ img	Folder
▶ services	Folder
▶ views	Folder
application.js	JavaScript
server.js	JavaScript
package.json	JSON

Horizontal folder structure

- The **app** folder is where you keep your **Express application logic** and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
  - The **controllers** folder is where you keep your Express application controllers
  - The **models** folder is where you keep your Express application models
  - The **routes** folder is where you keep your Express application **routing middleware**
  - The **views** folder is where you keep your Express application views

Horizontal folder structure

- The **config** folder is where you keep your Express **application configuration files**.
  - **each application module will be configured in a dedicated JavaScript file**, which is placed inside this folder.
- Currently, it contains several files and folders, which are as follows:
  - The **env** folder is where you'll keep your Express application **environment configuration files**.
  - The **config.js** file is where you'll **configure your Express application**.
  - The **express.js** file is where you'll **initialize your Express application**.

Horizontal folder structure

- The **public** folder is where you keep your **static client-side files** and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
  - The **config folder** – keeps your Angular application configuration files.
  - The **controllers** folder - keeps your Angular application controllers.
  - The **css** folder - keeps your CSS files
  - The **directives** folder - keeps your Angular application directives
  - The **filters** folder - keeps your Angular application filters
  - The **img** folder is where you keep your image files
  - The **views** folder is where you keep your Angular application views
  - The **application.js** file is where you **initialize** your AngularJS application

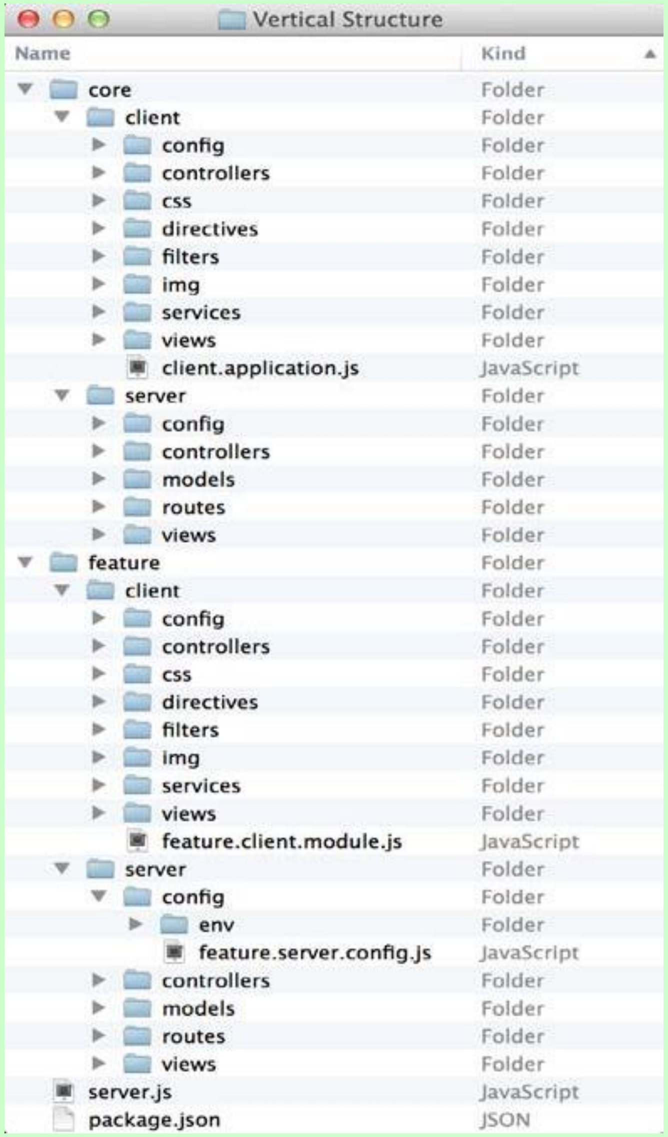
Horizontal folder structure

- In application root folder:
  - The **package.json** file is the metadata file that helps you to **organize your application dependencies**.
  - The **server.js** file is the **main file of your Node.js** application, and it will load the **express.js** file as a module **to bootstrap your Express application**.

Vertical folder structure

- Is based on the **division of folders and files by the feature they implement**.
- This means **each feature has its own autonomous folder that contains an MVC folder structure.**
  - An example feature would be a **user management feature** that includes the **authentication** and **authorization** logic.

Vertical folder structure



Vertical Structure

```
Name                                   Kind
▼ core                                 Folder
    ▼ client                           Folder
        ▶ config                       Folder
        ▶ controllers                  Folder
        ▶ css                          Folder
        ▶ directives                   Folder
        ▶ filters                      Folder
        ▶ img                          Folder
        ▶ services                     Folder
        ▶ views                        Folder
          client.application.js        JavaScript
    ▼ server                           Folder
        ▶ config                       Folder
        ▶ controllers                  Folder
        ▶ models                       Folder
        ▶ routes                       Folder
        ▶ views                        Folder
▼ feature                              Folder
    ▼ client                           Folder
        ▶ config                       Folder
        ▶ controllers                  Folder
        ▶ css                          Folder
        ▶ directives                   Folder
        ▶ filters                      Folder
        ▶ img                          Folder
        ▶ services                     Folder
        ▶ views                        Folder
          feature.client.module.js     JavaScript
    ▼ server                           Folder
        ▼ config                       Folder
            ▶ env                      Folder
              feature.server.config.js JavaScript
        ▶ controllers                  Folder
        ▶ models                       Folder
        ▶ routes                       Folder
        ▶ views                        Folder
  server.js                            JavaScript
  package.json                         JSON
```

Vertical folder structure

- The **server** folder - keeps your **feature's server logic** and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
  - The **controllers** folder - keeps your feature's Express controllers
  - The **models** folder - keeps your feature's Express models
  - The **routes** folder - keeps your feature's Express routing middleware
  - The **views** folder - keeps your feature's Express views
  - The **config** folder - keeps your feature's server configuration files
  - The **env** folder - keeps your feature's environment server configuration files
  - The *feature.server.config.js* file – to configure your feature

Vertical folder structure

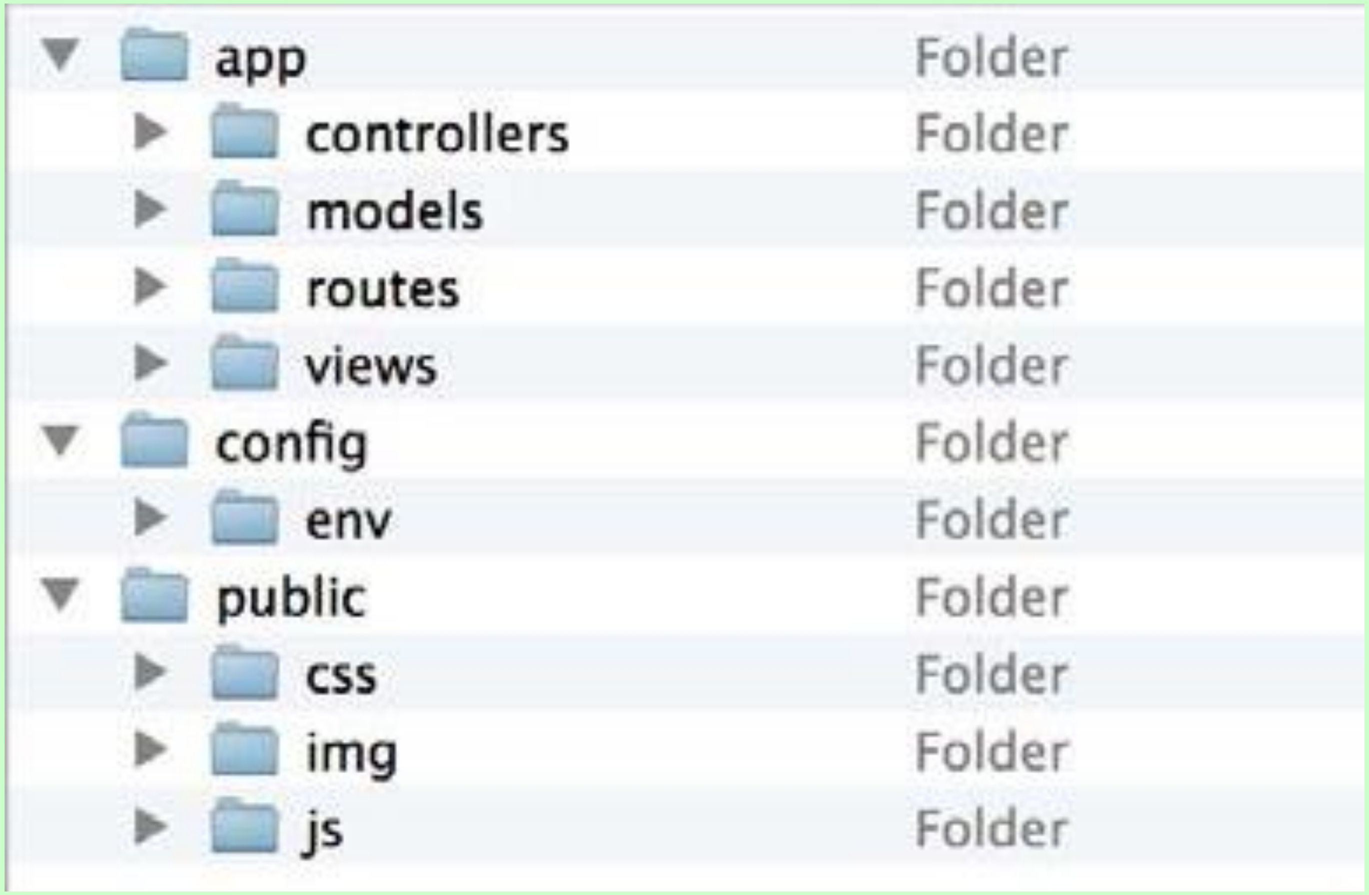
- The **client** folder is where you keep your **feature client-side files** and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
  - The **config** folder - keeps your feature's Angular configuration files
  - The **controllers** folder - keeps your feature's Angular controllers
  - The **css** folder - keeps your feature's CSS files
  - The **directives** folder - keeps your feature's Angular directives
  - The **filters** folder - keeps your feature's Angular filters
  - The **img** folder - keeps your feature's image files
  - The **views** folder - keeps your feature's Angular views
  - The ***feature1.client.module.js*** file - initialize your feature's Angular module

File-naming conventions

- MEAN applications use JavaScript MVC files for both the Express and Angular applications.
- This means that you'll often have two files with the same name; for instance, a **feature.controller.js** file might be an Express controller or an Angular controller.
- To solve this issue, it is also recommended that you **extend files names with their execution destination**.
- A simple approach would be to name our Express controller ***feature.server.controller.js*** and our Angular controller ***feature.client.controller.js***.
  - Helps to quickly identify the role and execution destination of your application files.

Implementing the horizontal folder structure

- In **Visual Studio 2017 or Code**, create the following folder structure under your new project folder:



```
▼ app            Folder
  ▶ controllers  Folder
  ▶ models       Folder
  ▶ routes       Folder
  ▶ views        Folder
▼ config         Folder
  ▶ env          Folder
▼ public         Folder
  ▶ css          Folder
  ▶ img          Folder
  ▶ js           Folder
```

Developing Express app - Steps

- Create the *package.json* file in application's root folder
- Create **controller** file *index.server.controller.js* in the *app/controllers* folder.
- Create your first **routing file** *index.server.routes.js* in the *app/routes* folder.
- **Configure** the Express app by creating *express.js file in the config* folder.
- Create ***server.js* file** in the root folder of your app
- Install your app **dependencies** using *npm*
- Start your application using Node's command-line tool in application's root folder:

node server

Developing Express app

- Create the following *package.json* file in application's root folder:

```
{
"name" : "MEAN",
"version" : "0.0.3",
"dependencies" : {
"express" : "~4.8.8"
}
}
```

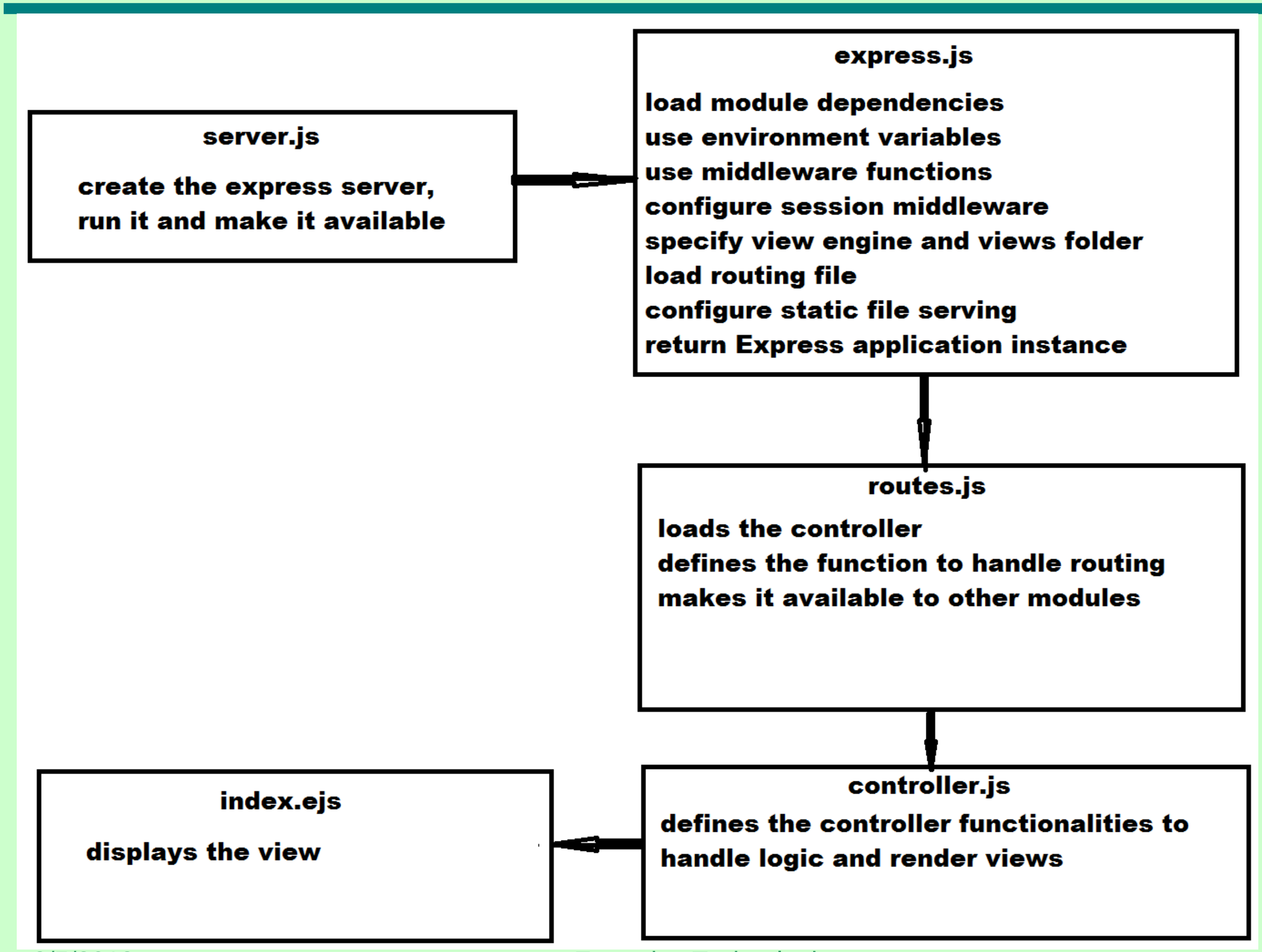
Developing Express app

- In the *app/controllers* folder, create a file named *index.server.controller.js* with the following lines of code:

```
exports.render = function(req, res) {
    res.send('Hello World from COMP308 class');
};
```

- This code uses the CommonJS module pattern to define a function named **render()**.
  - You can require this module and use this function.
- You'll need to use **Express routing functionality** to utilize the controller.

Express App Flowchart


server.js
create the express server,
run it and make it available
express.js
load module dependencies
use environment variables
use middleware functions
configure session middleware
specify view engine and views folder
load routing file
configure static file serving
return Express application instance
routes.js
loads the controller
defines the function to handle routing
makes it available to other modules
controller.js
defines the controller functionalities to
handle logic and render views
index.ejs
displays the view

Handling request routing

- *Routing* is the mechanism by which **requests** (as specified by a URL and HTTP method) **are routed to the code that handles them**.
- Express supports the **routing of requests** using either the **app.route(path).VERB(callback)** method or the **app.VERB(path, callback)** method, where VERB should be replaced with a lowercase HTTP verb (**get** or **post**).
- Example that tells Express to execute the middleware function for any HTTP **GET** request directed to the root path :

```
app.get('/', function(req, res) {
        res.send('This is a GET request');
});
```

- Example using **POST**:

```
app.post('/', function(req, res) {
        res.send('This is a POST request');
});
```

Handling request routing

- Express also enables you to define **a single route and then chain several middleware to handle different HTTP requests**:

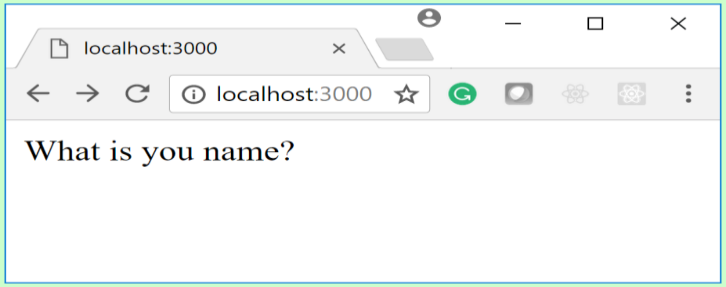
```
app.route('/').get(function(req, res) {
      res.send('This is a GET request');
     }).post(function(req, res) {
      res.send('This is a POST request');
});
```

Handling request routing

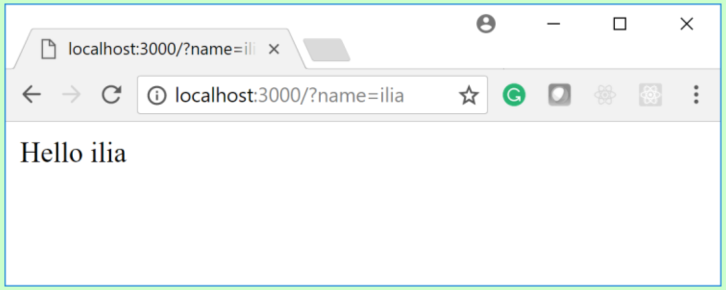
- Another cool feature of Express is the **ability to chain several middleware in a single routing definition**.
- This means **middleware functions will be called in order**, passing them to the next middleware so you could determine how to proceed with middleware execution.
- This is usually **used to validate requests** before executing the response logic.
  - See the example on next slide:

Handling request routing

```
var express = require('express');
var hasName = function(req, res, next) {
    if (req.param('name')) {
    next();
    } else {
    res.send('What is your name?');
    }
};
var sayHello = function(req, res, next) {
    res.send('Hello ' + req.param('name'));
};
var app = express();
//add the middleware function in a row to specify the order in which is called
app.get('/', hasName, sayHello); //hasName is called first, then sayHello
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```



What is you name?



Hello ilia

Handling request routing

- In the preceding code, there are two middleware functions named **hasName()** and **sayHello()**.
  - The **hasName()** middleware is looking for the *name* parameter:
    - if it finds a defined *name* parameter, it will call the next middleware function using the *next* argument.
      - In this case, the next middleware function would be the **sayHello()** middleware function.
    - Otherwise, the **hasName()** middleware will handle the response by itself.
- This is possible because we've **added the middleware function in a row** using the **app.get()** method.
  - the **order of the middleware functions** determines which middleware function is executed first.

Handling request routing

- ❑ Create your first **routing file**:
- ❑ In the *app/routes* folder, create a file named **index.server.routes.js** with the following code snippet:

```
module.exports = function(app) {
    var index =
    require('../controllers/index.server.controller');
    app.get('/', index.render);
};
```

- ❑ This uses CommonJS module pattern to **export** a single module function.
- ❑ Then it requires the index controller and uses its render() method as a middleware to GET requests made to the root path.

Configure Express application

- Create the Express application object and bootstrap it using the controller and routing modules you just created.
  - go to the *config* folder and create a file named *express.js* with the following code snippet:

```
var express = require('express');
module.exports = function() {
var app = express();
require('../app/routes/index.server.routes.js')(app);
return app;
};
```

- The *express.js* file is where we **configure our Express application**.

Run Express application

- Create a file named *server.js* **in the root folder** and copy the following code:

```
var express = require('./config/express');
var app = express();
app.listen(3000);
module.exports = app;
```

- Navigate to your **application's root folder** using your command-line tool, and install your application dependencies using npm, as follows:

npm install

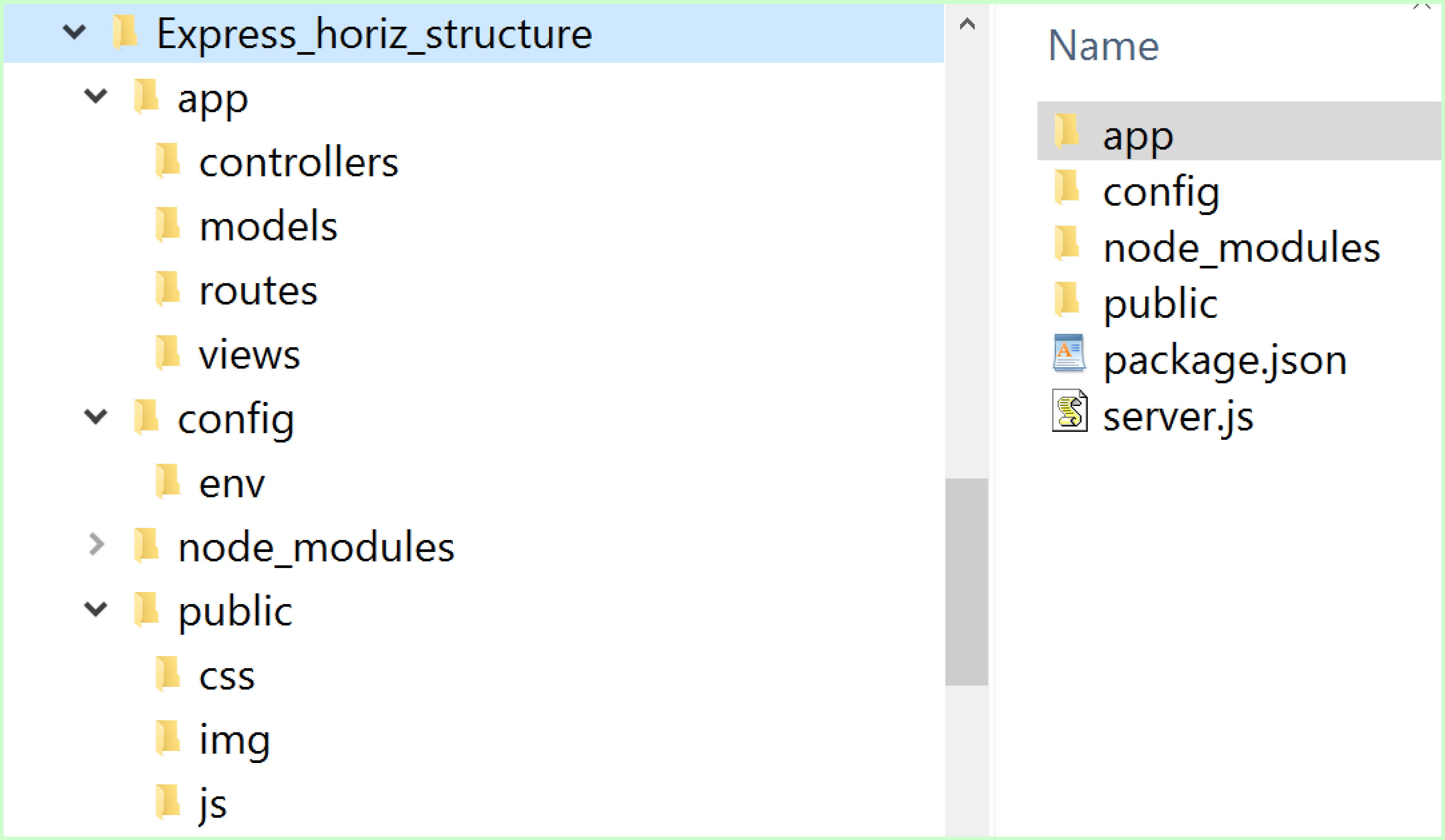
- Once the installation process is over, all you have to do is start your application using Node's command-line tool:

node server

- Test the app by navigating to http://localhost:3000.

App Folder Structure

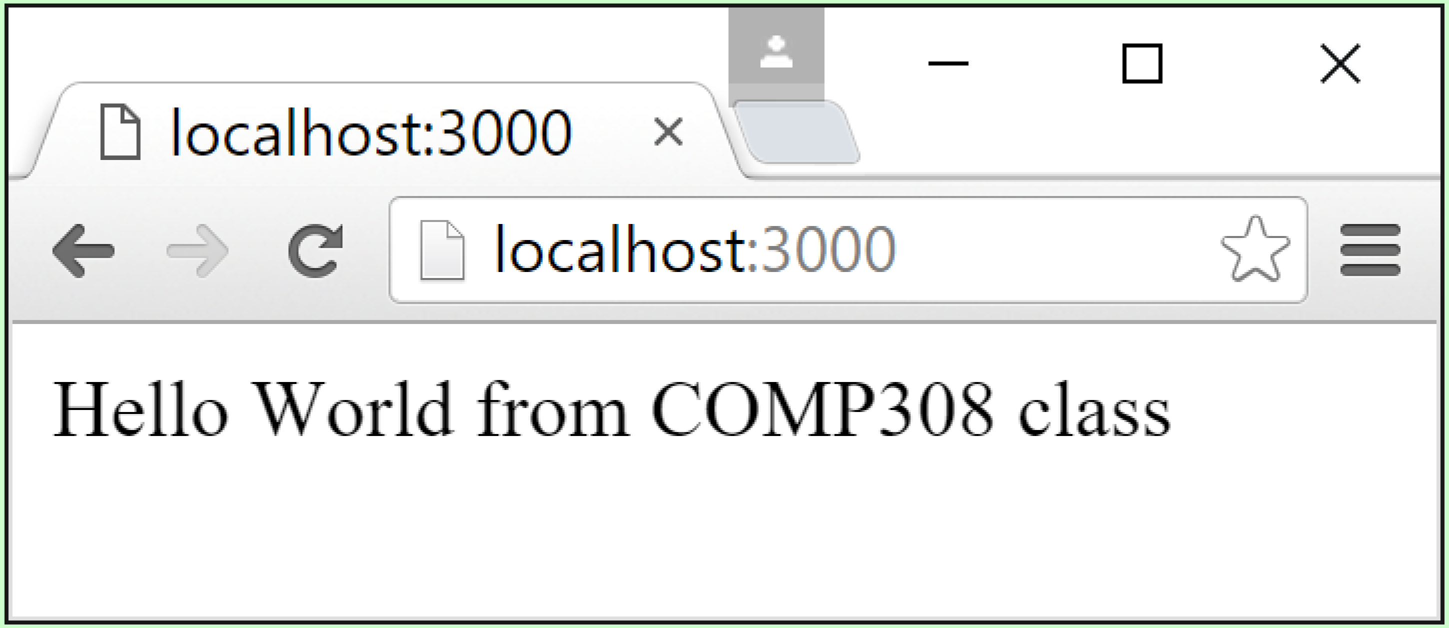
- **Node_modules** is created after running *npm*:



```
Express_horiz_structure
    app
        controllers
        models
        routes
        views
    config
        env
    node_modules
    public
        css
        img
        js
```

Name
- app
- config
- node_modules
- public
- package.json
- server.js

Run Express application



Hello World from COMP308 class

Configuring an Express application

- Another robust feature of Express is the ability to configure your application **based on the environment it's running on**.
- Let’s demonstrate the use the Express logger in your development environment and not in production.
  - the **process.env** property allows you to **access predefined environment variables** such as **NODE_ENV**.
  - Some external middleware is needed – update package.json file: {

```
"name": "MEAN",
"version": "0.0.3",
"dependencies": { "express": "~4.8.8", "morgan": "~1.3.0",
"compression": "~1.0.11", "body-parser": "~1.8.0",
"method-override": "~2.2.0“ }
}
```

Configuring an Express application

- The **morgan** module provides a simple logger middleware.
- The **compression** module provides response compression.
- The **body-parser** module provides several middleware to handle request data.
- The **method-override** module provides DELETE and PUT HTTP verbs legacy support.
- **Modify your config/express.js** to use these modules

Configuring an Express application – express.js

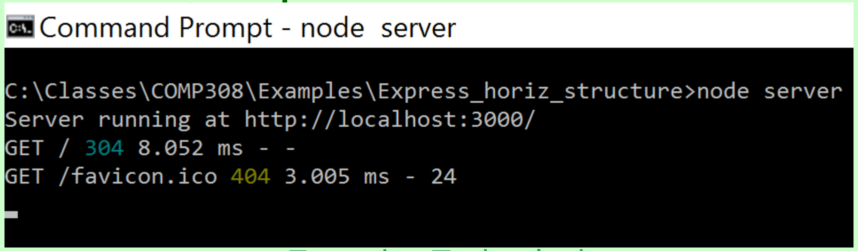
```
var express = require('express'),
morgan = require('morgan'),
compress = require('compression'),
bodyParser = require('body-parser'),
methodOverride = require('method-override');
module.exports = function() {
var app = express();
if (process.env.NODE_ENV === 'development') {
app.use(morgan('dev'));
} else if (process.env.NODE_ENV === 'production') {
app.use(compress());
}
app.use(bodyParser.urlencoded({
extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
require('../app/routes/index.server.routes.js')(app);
return app;
};
```

Configuring an Express application

- Finalize your configuration, by changing *server.js* file to look like the following code snippet:

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';
var express = require('./config/express');
var app = express();
app.listen(3000);
module.exports = app;
console.log('Server running at http://localhost:3000/');
```

- The process.env.NODE_ENV variable is set to the default 'development' value if it doesn't exist.
- Install your application dependencies: **npm install**
- Start your application: **node server**
- Test by navigating to http://localhost:3000 - see the logger in action in your command-line output:



Command Prompt - node server

```
C:\Classes\COMP308\Examples\Express_horiz_structure>node server
Server running at http://localhost:3000/
GET / 304 8.052 ms - -
GET /favicon.ico 404 3.005 ms - 24
```

Environment configuration files

- Often you need **to configure third party modules** to run differently in various environments.
  - For instance, when you connect to your MongoDB server, you'll probably use **different connection strings** in your development and production environments.
  - Use a set of environment configuration files to hold these properties, rather than if statements in your code.
  - Use the **process.env.NODE_ENV** environment variable to determine which configuration file to load, thus keeping your code shorter and easier to maintain

Environment configuration files

- ❑ Create a new file inside your **config/env folder** and call it *development.js*:

```
module.exports = {
// Development configuration options
};
```

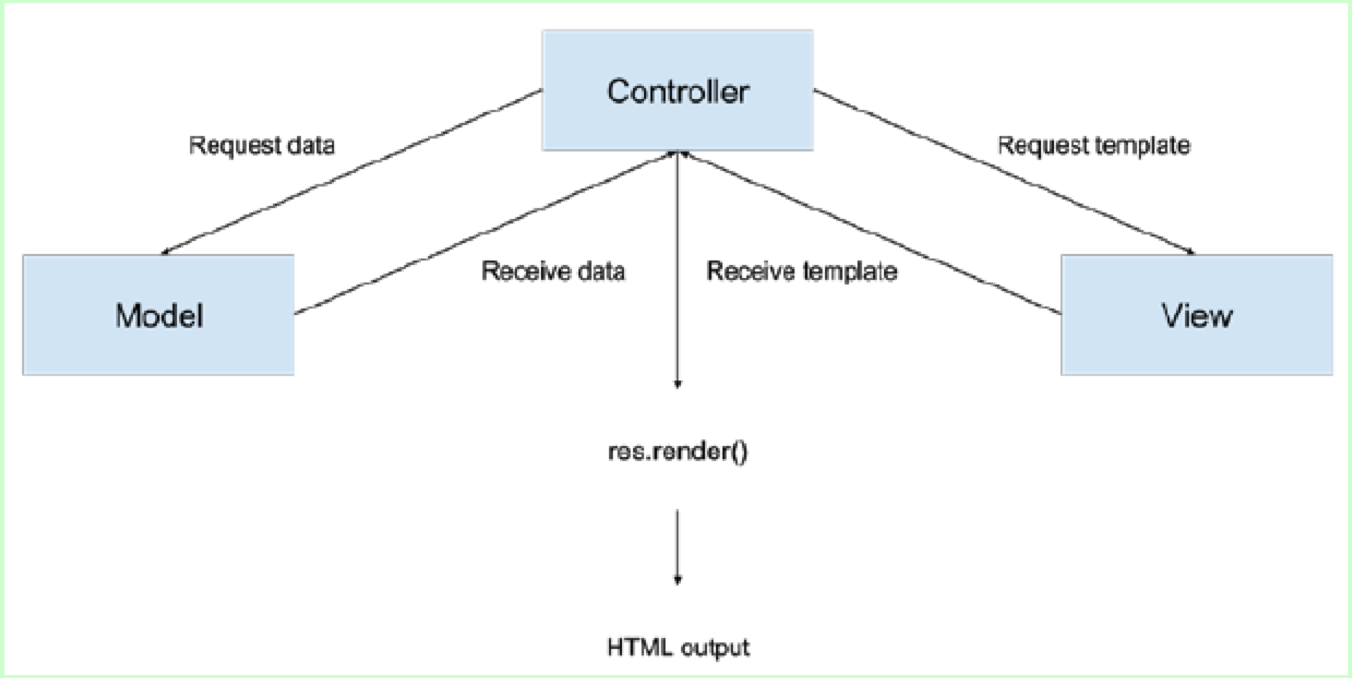
- ❑ Go to your **application config folder** and create a new file named *config.js*:

```
module.exports = require('./env/' + process.env.NODE_ENV + '.js');
```

- ❑ This file simply loads the correct configuration file according to the *process.env.NODE_ENV* environment variable.

Rendering views

- In the MVC pattern, your **controller uses the model to retrieve the data portion** and **the view template to render the HTML output** as described in the next diagram.
- The Express extendable approach allows the **usage of many Node.js template engines to achieve this functionality**.
  - using the **ejs**



Rendering Views

- Express has two methods for rendering views:
  - **app.render()** - used to render the view and then pass the HTML to a callback function.
  - **res.render()** - renders the view locally and sends the HTML as a response.
    - You'll use **res.render()** more frequently because you usually want **to output the HTML as a response**.
  - If application wants to send HTML e-mails, use app.render().

Configuring the view system

- To use the EJS template engine, install the EJS module: **Change *package.json* file** to look like the following code snippet:

```
{
"name": "MEAN",
"version": "0.0.3",
"dependencies": {
"express": "~4.8.8",
"morgan": "~1.3.0",
"compression": "~1.0.11",
"body-parser": "~1.8.0",
"method-override": "~2.2.0",
"ejs": "~1.0.0"
}
}
```

- **Install the EJS module** by navigating in the command line to your project's root folder and issue the following command:

```
npm update
```

Configuring the view system

- Configure Express to use EJS module as the default template engine.
- Change *config/express.js* file to look like the following lines of code:

```
var express = require('express'),
morgan = require('morgan'),
compress = require('compression'),
bodyParser = require('body-parser'),
methodOverride = require('method-override');
module.exports = function() {
var app = express();
if (process.env.NODE_ENV ===
'development') {
app.use(morgan('dev'));
} else if (process.env.NODE_ENV ===
'production') {
app.use(compress());
}
app.use(bodyParser.urlencoded({
extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
app.set('views', './app/views');
app.set('view engine', 'ejs');
require('../app/routes/index.server.rou
tes.js')(app);
return app;
};
```

Rendering EJS views

- EJS views basically consist of **HTML code mixed with EJS tags**.
- EJS templates will reside in the *app/views* folder and will have the **.ejs** extension.
- When you'll use the res.render() method, the EJS engine will look for the template in the views folder, and if it finds a complying template, it will render the HTML output.
- To create your first EJS view, go to your **app/views** folder, and create a new file named **index.ejs** that contains the following HTML code snippet:

```
<!DOCTYPE html>
<html>
<head>
<title><%= title %></title>
</head>
<body>
<h1><%= title %></h1>
</body>
</html>
```

Rendering EJS views

- Configure your controller to render this template and automatically output it as an HTML response.
  - Change the app/controllers/index.server.controller.js file, to look like the following code snippet:

```
exports.render = function(req, res) {
        res.render('index', {
                title: 'Hello World'
        })
};
```

- The first argument of res.render() method is the name of your EJS template without the .ejs extension, and the second argument is an object containing your template variables.
- Run the server to test: **node server**
- Test your application by visiting http://localhost:3000 where you'll be able to see the rendered HTML.

Serving static files

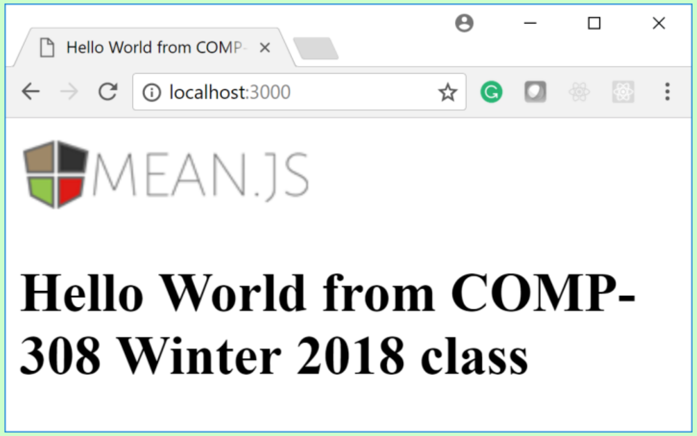
- Express comes prebundled with the **express.static()** middleware, which allows to server static files
- To add static file support to the previous example, just make the following changes in your **config/express.js** file:

```
var express = require('express'),
morgan = require('morgan'),
compress = require('compression'),
bodyParser = require('body-parser'),
methodOverride = require('method-override');
module.exports = function() {
var app = express();
if (process.env.NODE_ENV === 'development') {
app.use(morgan('dev'));
} else if (process.env.NODE_ENV === 'production') {
app.use(compress());
}
app.use(bodyParser.urlencoded({
extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
app.set('views', './app/views');
app.set('view engine', 'ejs');
require('../app/routes/index.server.routes.js')(app);
app.use(express.static('./public'));
return app;
};
```

Serving static files

- The **express.static()** middleware takes **one argument to determine the location of the static folder**.
- express.static() middleware is placed **below the call for the routing file**. This order matters and speeds up the response.
- To test your static middleware, add an image named logo.png to the **public/img** folder and change **app/views/index.ejs** file:

```
<!DOCTYPE html>
<html>
<head>
<title><%= title %></title>
</head>
<body>
<img src="img/nodejs_logo.png" alt="Logo">
<h1><%= title %></h1>
</body>
</html>
```



Hello World from COMP-308 Winter 2018 class

- Run **node server** and visit http://localhost:3000 in your browser and watch how Express is **serving your image as a static file**.

Configuring sessions

- Sessions allow you to **keep track of the user's behavior** when they visit your application.
- To add this functionality, you will need to **install and configure the express-session middleware**. Modify the *package.json* file like this:

```
{
"name": "MEAN",
"version": "0.0.3",
"dependencies": {
"express": "~4.8.8",
"morgan": "~1.3.0",
"compression": "~1.0.11",
"body-parser": "~1.8.0",
"method-override": "~2.2.0",
"express-session": "~1.7.6",
"ejs": "~1.0.0"
}
}
```

- Use **npm update** to install express-session.

Configuring sessions

- The express-session module uses a **cookie-stored, signed identifier** to identify the current user.
- To **sign the session identifier**, use a **secret string**, which will help prevent malicious session tampering.
- For security reasons, it is recommended that the **cookie secret be different for each environment**, which means this would be an **appropriate place to use our environment configuration file**.
- To do so, change the **config/env/development.js** file to look like the following code snippet:

```
module.exports = {
        sessionSecret: 'developmentSessionSecret'
};
```

- Feel free to change the secret string used above.
- For other environments, just add the **sessionSecret** property in their environment configuration files.

Configuring sessions

- To use the configuration file and configure your Express application, go back to your **config/express.js** file and change it to look like the following code snippet:

```
var config = require('./config'),
express = require('express'),
morgan = require('morgan'),
compress = require('compression'),
bodyParser = require('body-parser'),
methodOverride = require('method-override'),
session = require('express-session');
module.exports = function() {
var app = express();
if (process.env.NODE_ENV ===
'development') {
app.use(morgan('dev'));
} else if (process.env.NODE_ENV ===
'production') {
app.use(compress());
}
app.use(bodyParser.urlencoded({
extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
app.use(session({
saveUninitialized: true,
resave: true,
secret: config.sessionSecret
}));
app.set('views', './app/views');
app.set('view engine', 'ejs');
require('../app/routes/index.server.routes.js')(
app);
app.use(express.static('./public'));
return app;
};
```

Configuring sessions

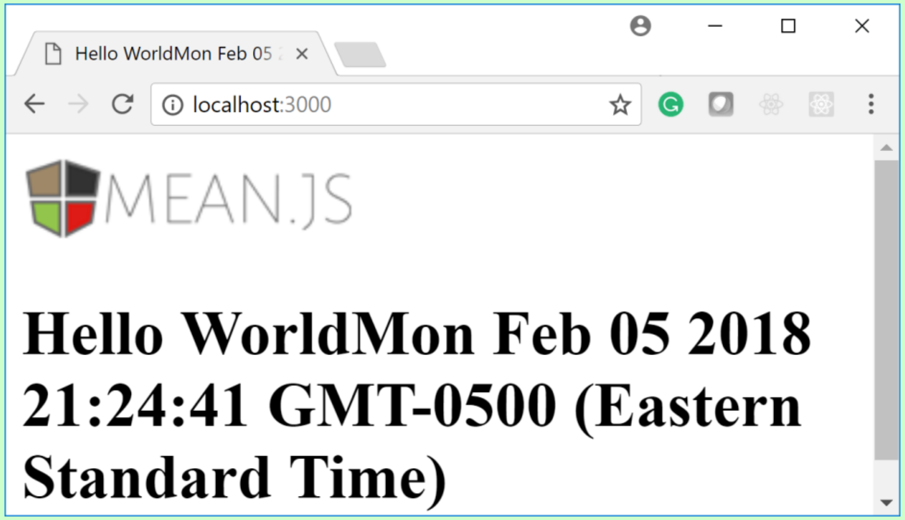
- To test the session, change the **app/controller/index.server.controller.js** file as follows:

```
exports.render = function(req, res) {
if (req.session.lastVisit) {
console.log('Last visit: ‘+ req.session.lastVisit);
}
req.session.lastVisit = new Date();
res.render('index', {
        title: 'Hello World on ' + req.session.lastVisit});
};
```

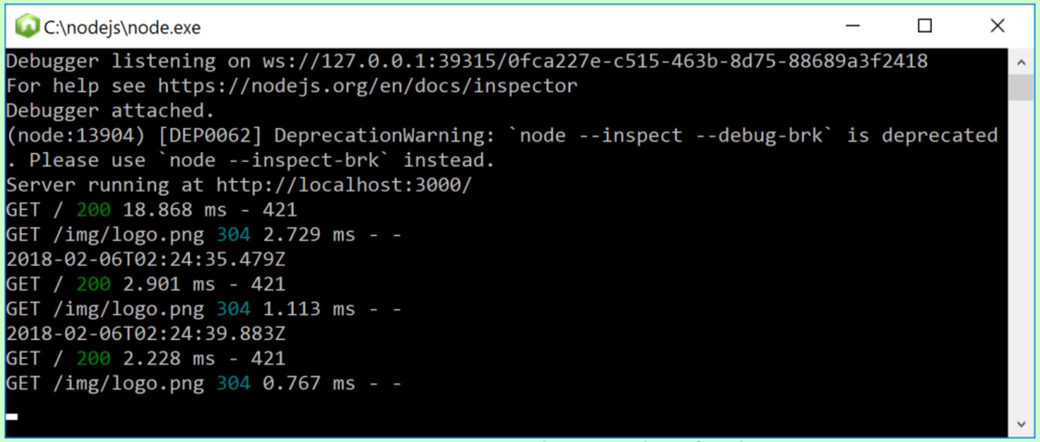
- The controller checks whether the lastVisit property was set in the session object, and if so, outputs the last visit date to the console.
- Then, sets the lastVisit property to the current time.
- Run **node server** and visit http://localhost:3000 to test it.

Using Sessions

- ❑ The current time is displayed on HTML page.
- ❑ Refresh and see last visit time on the console

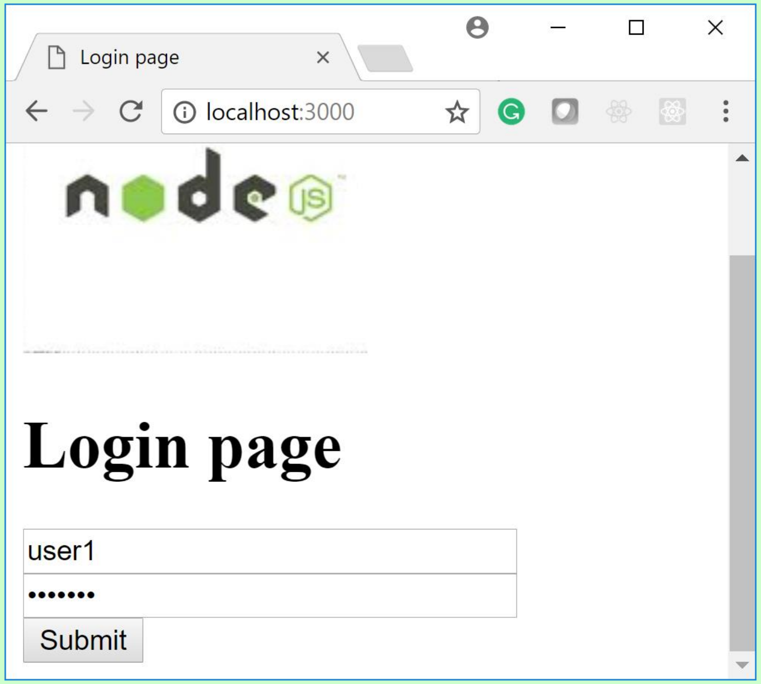


Hello WorldMon Feb 05 2018 21:24:41 GMT-0500 (Eastern Standard Time)



```
Debugger listening on ws://127.0.0.1:39315/0fca227e-c515-463b-8d75-88689a3f2418
For help see https://nodejs.org/en/docs/inspector
Debugger attached.
(node:13904) [DEP0062] DeprecationWarning: `node --inspect --debug-brk` is deprecated
. Please use `node --inspect-brk` instead.
Server running at http://localhost:3000/
GET / 200 18.868 ms - 421
GET /img/logo.png 304 2.729 ms - -
2018-02-06T02:24:35.479Z
GET / 200 2.901 ms - 421
GET /img/logo.png 304 1.113 ms - -
2018-02-06T02:24:39.883Z
GET / 200 2.228 ms - 421
GET /img/logo.png 304 0.767 ms - -
```

Session Management Example



Login page

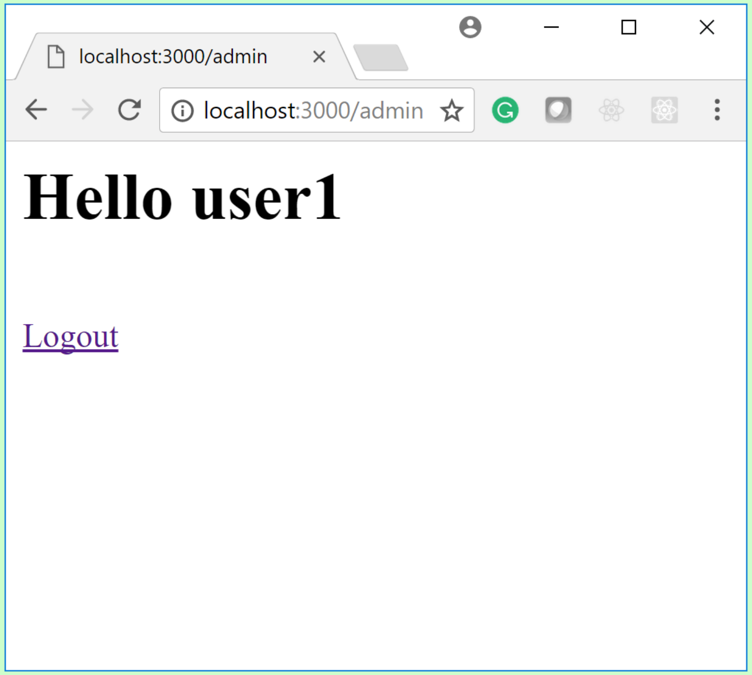
localhost:3000

Login page

user1

•••••••

Submit



localhost:3000/admin

Hello user1

Logout

References

- Textbook
- http://expressjs.com/
- http://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm