

# Week 9

**Backend Development, MVC Architecture, Intro  
to Express.JS**

# See me to get your exam.

As you come in the room.  
Tell me your Pitt user ID.

# Today's Agenda

Return & Review Exams

Back End Development

Model View Controller (MVC) Architecture

Express.js

Project 2 Presentations

# Exam 1

## Let's review

There are 2 versions. I will be following version A, referencing version B.

- Grading curve: +5 points
- You may see some of these questions again
- You can keep the Answer Sheet
- You can review the exam questions after class or during TA office hours
  - **You cannot keep the exam questions.**
- Exam 2 will be similar
  - Multiple choice, Short Answer, Coding, 100 points

# Project 2

- Presentations at the END of class.
- Random names will be selected. **BE READY.**
- Due on Courseweb by 11:59p tonight.
- Late policy: 10% off final grade, up to 3 days.



## Front End

- Markup and web languages such as HTML, CSS and Javascript
- Asynchronous requests and Ajax
- Specialized web editing software
- Image editing
- Accessibility
- Cross-browser issues
- Search engine optimisation



## Back End

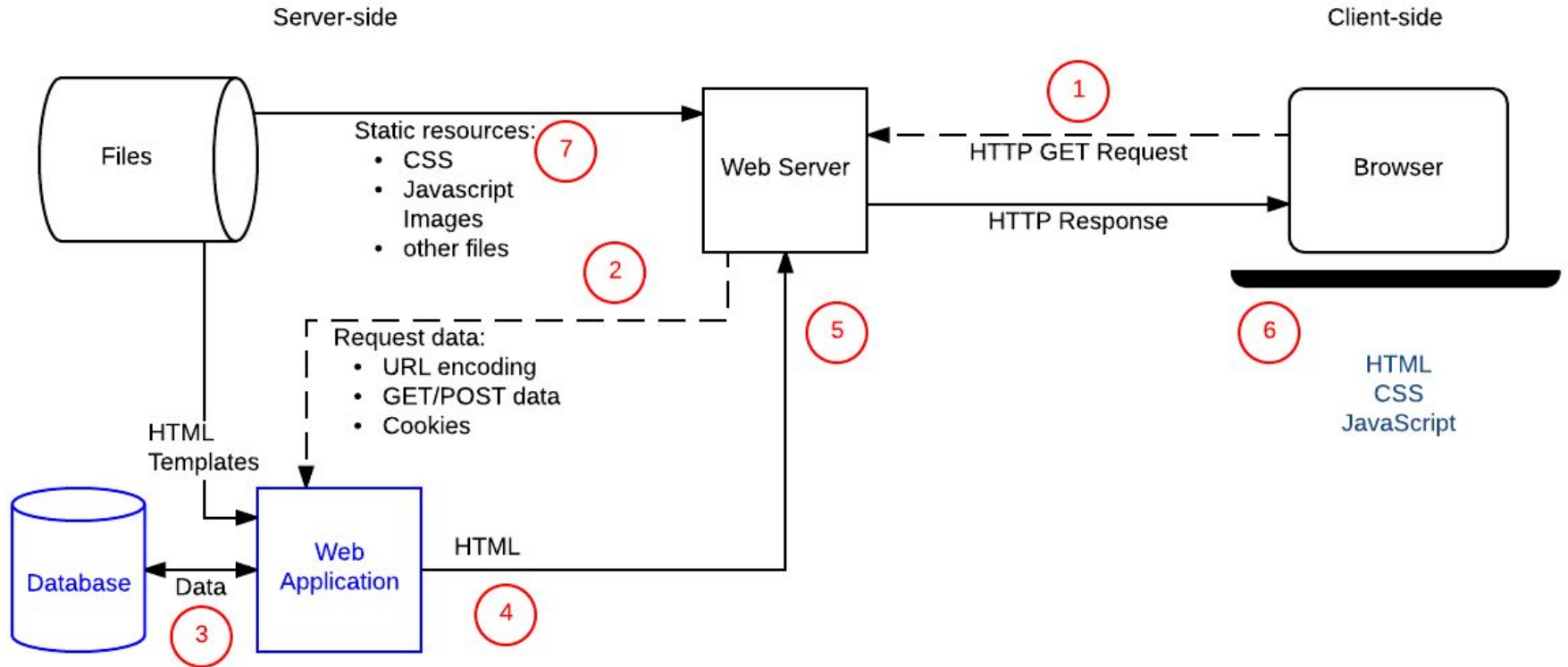
- Programming and scripting such as Python, Ruby and/or Perl
- Server architecture
- Database administration
- Scalability
- Security
- Data transformation
- Backup

[Image Source](#)

# Back End Development

- Sometimes called server side web development
- The back-end is the portion of the website you don't see. it's responsible for saving, managing and sometimes processing data.
- Focuses on how the site works/functions
- Often different language(s) from front-end
  - Java, PHP, Python, Ruby
- But can be in JavaScript
- Use a lot of frameworks to handle complex and repetitive tasks
  - sessions, authentication, data access/storage and templating

# Dynamic Websites





# Why Server Side?

## **Efficient storage and delivery of information**

- data is more efficiently stored in a database
- dynamically construct HTML pages using templates
- return just the data (in JSON or XML)

## **Customize User Experience**

- Use Sessions to track a user across requests.
- display information about that user
- Store information about the user like credit cards or profile information

## **Access Control**

- Restrict access to data and information only to authorized users
- data on the server is more secure. Never trust a client!
- Read Only vs Read/Write

# Why Server Side?

## **Data collection & analysis**

- User behavioral data collection
- Marketing, advertising

## **Heavy Computation**

- E.g. computing the best route on Google Maps
- Speech to text
- Machine Learning

# Web Servers

- On the hardware side, a web server is a computer that stores web server software and a website's component files (e.g. HTML documents, images, CSS stylesheets, and JavaScript files). It is connected to the Internet and supports physical data interchange with other devices connected to the web.
- On the software side, a web server includes several parts that control how web users access hosted files, at minimum an HTTP server. An HTTP server is a piece of software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). It can be accessed through the domain names (like mozilla.org) of websites it stores, and delivers their content to the end-user's device.

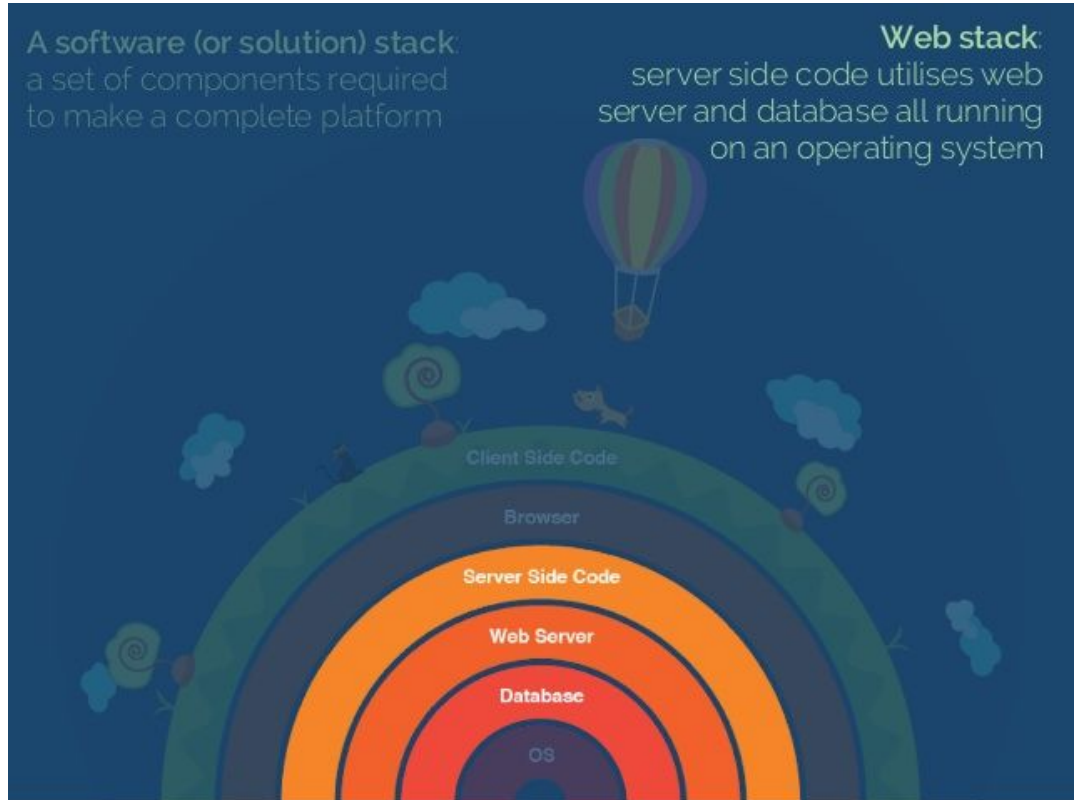
# Web Servers

- Running a web server process that is typically listening on TCP port 80 and 443 (for HTTPS)
- But could listen on other ports as well
- At the most basic an HTTP server will listen for a request, process it, and hand off that request to another computational process
- Apache and Nginx are the two most popular web servers
- **Node.js** is a popular web server option, because it combines the webserver AND the web application.

# Solution Stacks

When referring to the technology or solution stack - this includes the tools, frameworks and technologies used to develop a web application.

Abstract solution stack. Image from [Slideshare](#)



# Solution usually refer to:

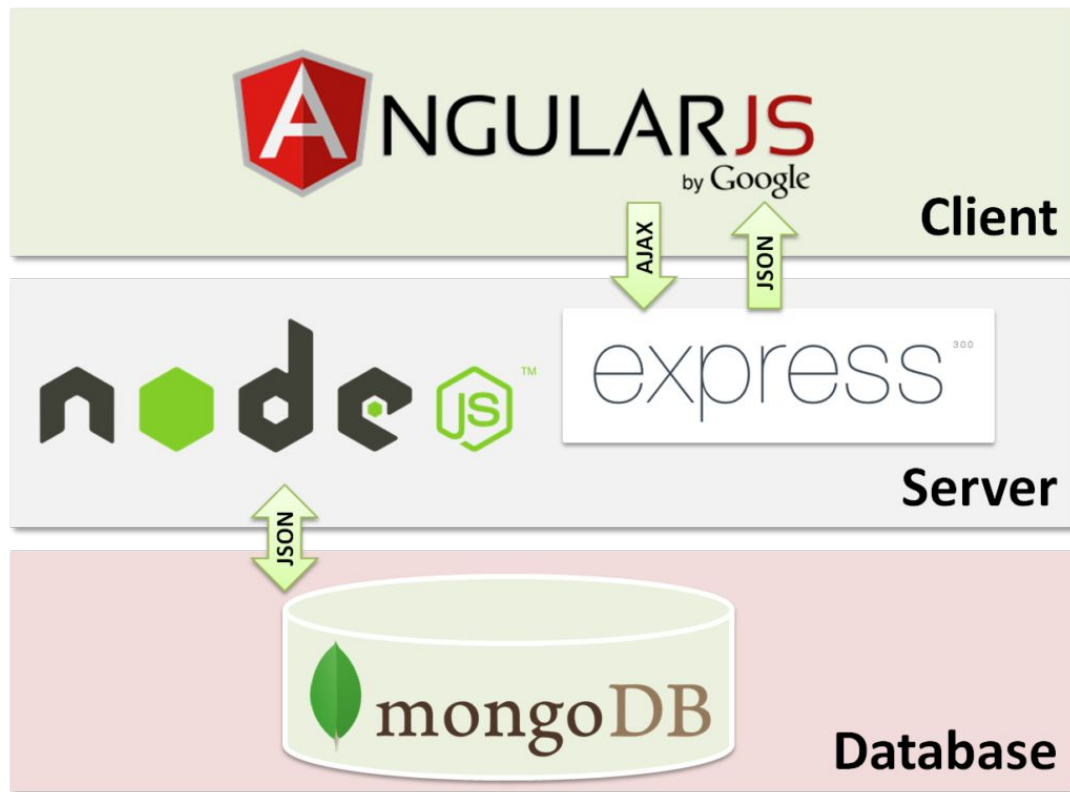
## Front-end or client-side

- HTML
- CSS
- JavaScript
- Frameworks
  - Bootstrap
  - Foundation
  - Angular
  - React
  - Vue

## Back-end or server-side

- Operating System
- Web server
- Database
- Programming Language
- Web Development Frameworks

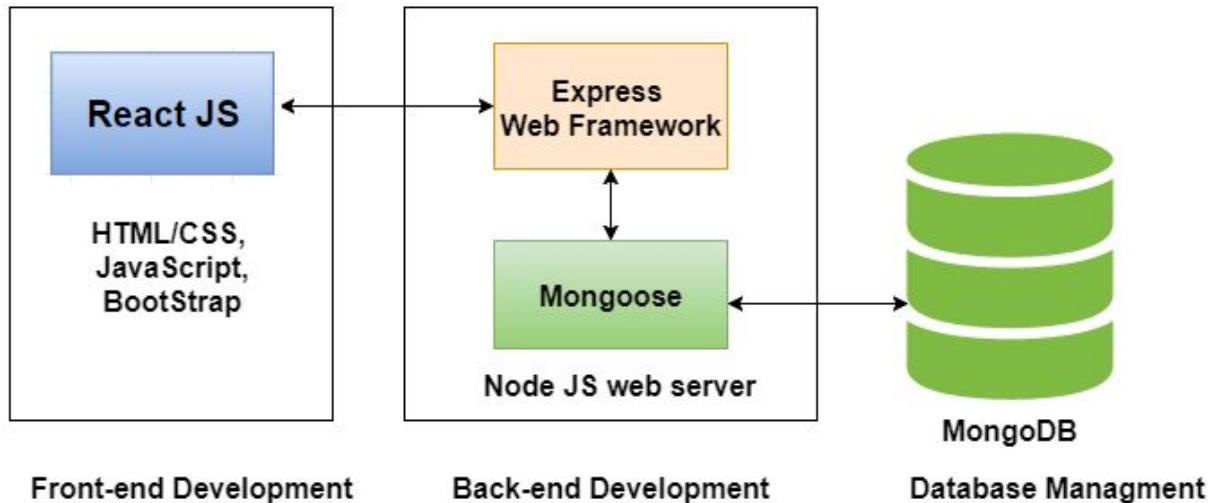
# MEAN Stack



MongoDB ExpressJS AngularJS NodeJs (MEAN) Stack. Image from [HackerNoon](#)

# MERN Stack Development

## MERN Stack Development





# Picking a technology stack/solution

There is no straightforward, right answer.

- **Users before developers.** Think about the needs of your users first. How can you build the best user experience?
- **The technology you choose should depend on the problem you want to solve.** Is there a specific technology (or framework) that is specifically designed for this purpose? (E.g. Mobile-app, 3D modeling, Video Games/Animations)
- **What is the experience of the people developing the solution?** What is your timeline? Learning curve? Resources available?
- **What are the technical requirements?** How important is performance? Scalability? Security?
- **What is the cost?** For example, licensing, hosting.
- **Check out the ecosystem.** The ecosystem is the people and tools behind the technology. Is there a support system for developers (e.g. conferences, documentation)?

**Read more in this article:** [9 Steps to Choose a Technology Stack for your Web Application](#)

# N-tier web application development and Model View Controller (MVC)

# Web Application Architecture

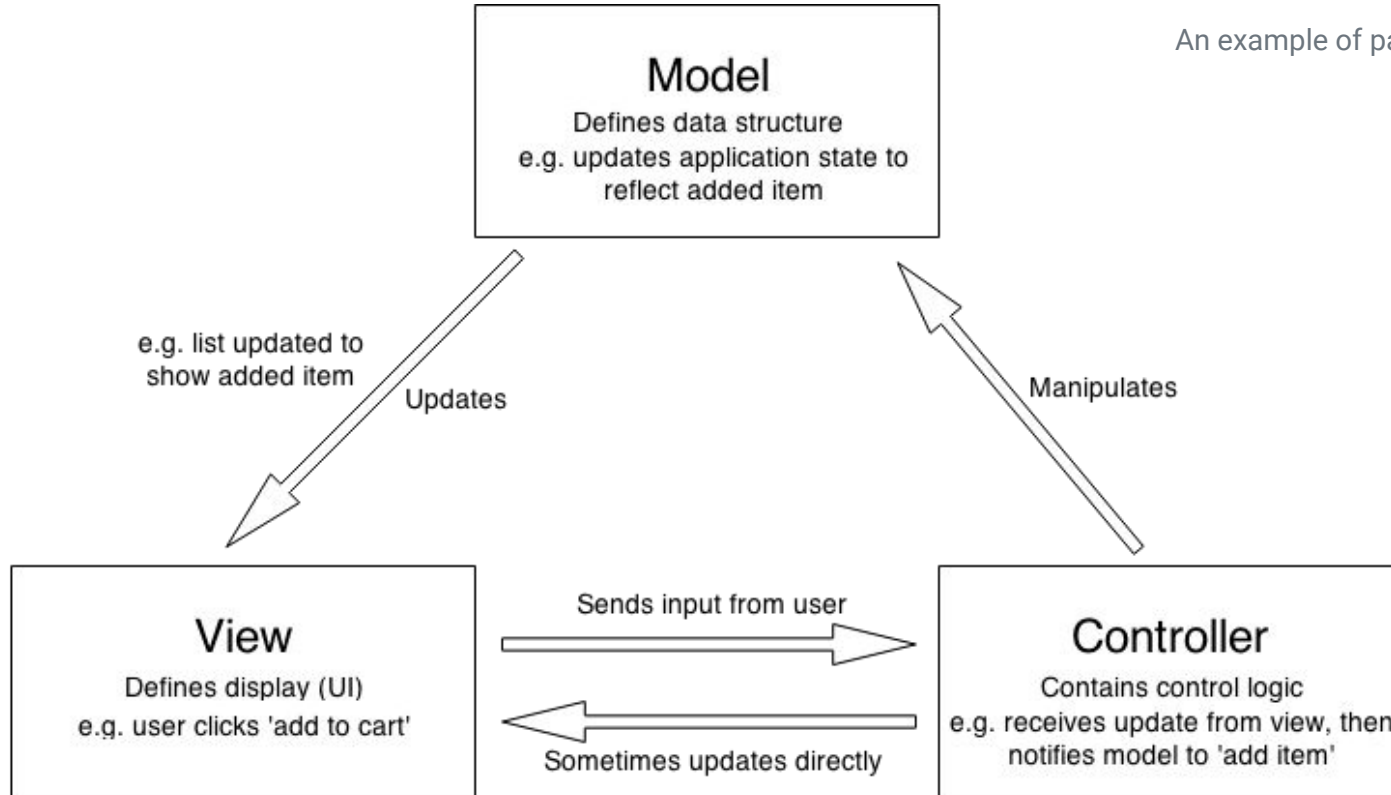
Separates an application into three parts:

1. The Model - Central component and represents the data structure independent from the interface
2. The View - Create representations of information. The interface of the application.
3. The Controller - Manages the connections between the View and the Model. Dispatch

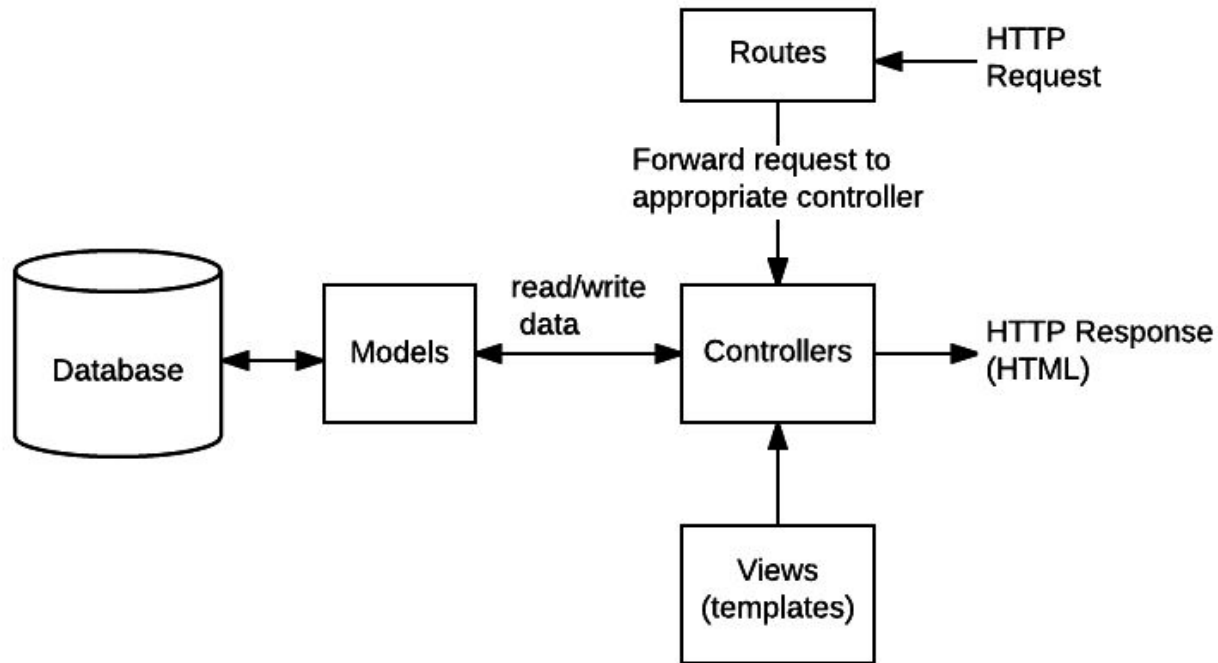
The separation of concerns promotes modularity, reuse, and more flexibility when iterating different aspects of the web application

# MVC Diagram

An example of page titles. Image from [MDN](#)



# MVC & Web Frameworks



**Break (10 minutes)**

# Backend JS Development

Google developed a powerful, and fast JavaScript engine for Chrome called V8

In 2009, Node.js was released. Provided a web centric platform for building asynchronous JavaScript applications on the backend

Node got really popular. One language for frontend and backend

**Node is:** asynchronous & event driven, very fast, highly scalable and open source.

# Node - Event Driven JavaScript





# Node Web Server Example

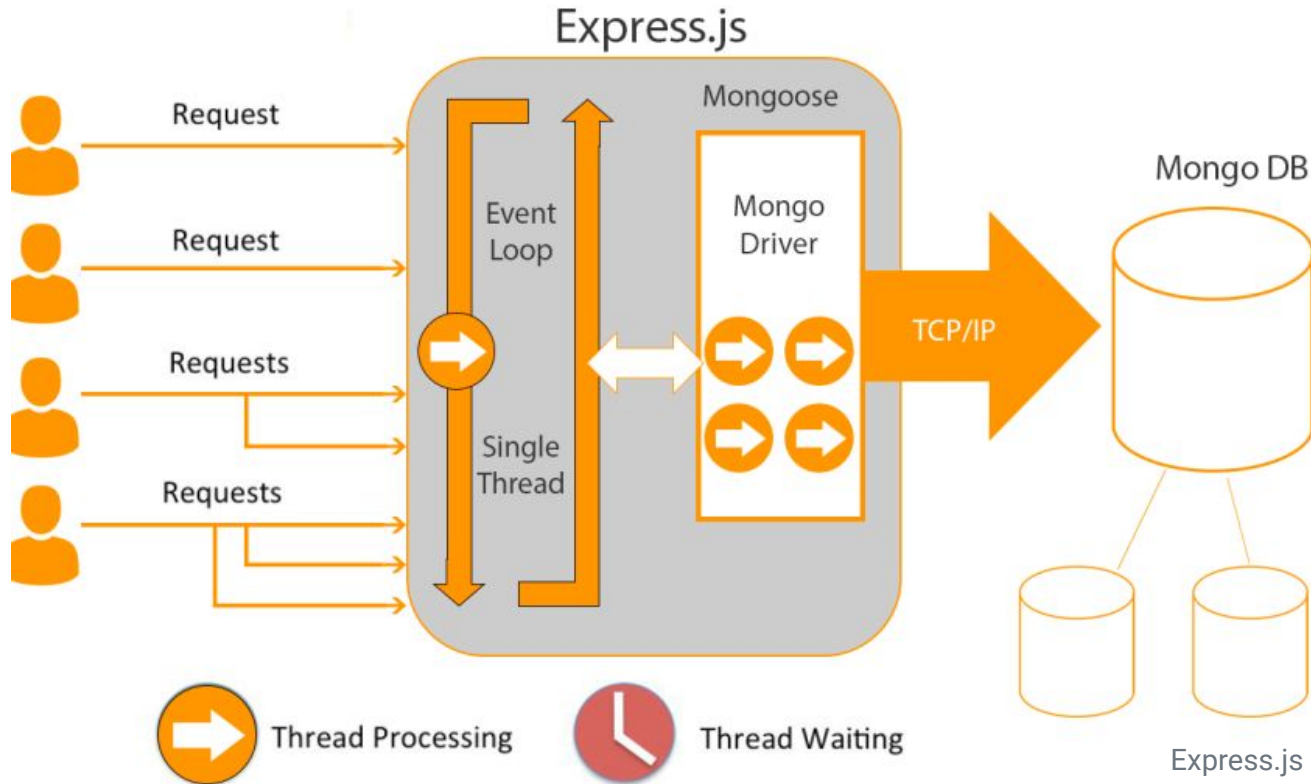
```
const http = require('http');

http.createServer(function(req, res) => {
  res.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  res.end('Hello World');
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

- Vanilla Node/JavaScript web server example
- Very low-level API, all functionality packed into a single callback function
- You would need to write a lot of additional code to get basic web functionality (serving static files), let alone more advanced features (routing, templating, sessions)

# Intro to Express.js - Backend Web Framework



# Express.js Major Features

- **Middleware** - In contrast to vanilla Node, where your requests flow through only one function, Express has a middleware stack, which is effectively an array of functions.
- **Routing** - Routing is a lot like middleware, but the functions are called only when you visit a specific URL with a specific HTTP method. For example, you could only run a request handler when the browser visits `yourwebsite.com/about`.
- **Extensions to request and response objects** - Express extends the request and response objects with extra methods and properties for developer convenience.
- **Views** - Views allow you to dynamically render HTML. This both allows you to change the HTML on the fly and to write the HTML in other languages.

# What Express Code Looks Like

```
var express = require("express");  
var http = require("http");
```

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

```
app.use(function(request, response) {  
  console.log("In comes a request to: " + request.url);  
  response.end("Hello, world!");  
});
```

```
http.createServer(app).listen(3000);
```

←  
**Calls the express  
function to start a new  
Express application**

←  
**Requires the Express  
module just as you  
require other modules**

**Middleware**

← **Starts the server**

Example Express app. Image from [Express in Action](#)

# What Express Code Looks Like

```
var express = require("express");  
var http = require("http");
```

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

```
app.use(function(request, response) {  
  console.log("In comes a request to: " + request.url);  
  response.end("Hello, world!");  
});
```

```
http.createServer(app).listen(3000);
```

←  
**Calls the express  
function to start a new  
Express application**

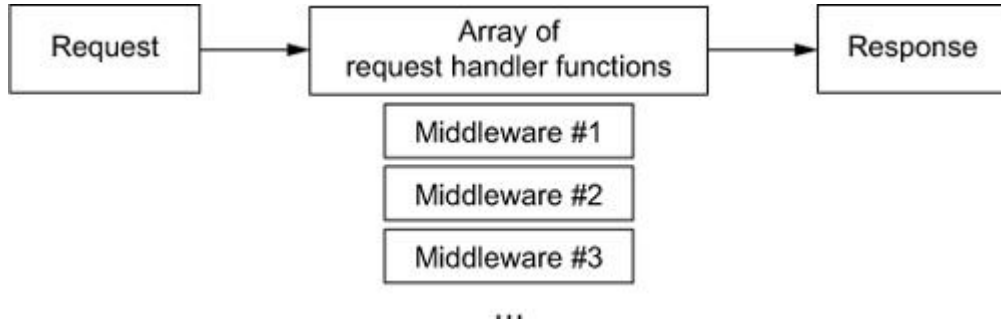
←  
**Requires the Express  
module just as you  
require other modules**

**Middleware**

← **Starts the server**

Example Express app. Image from [Express in Action](#)

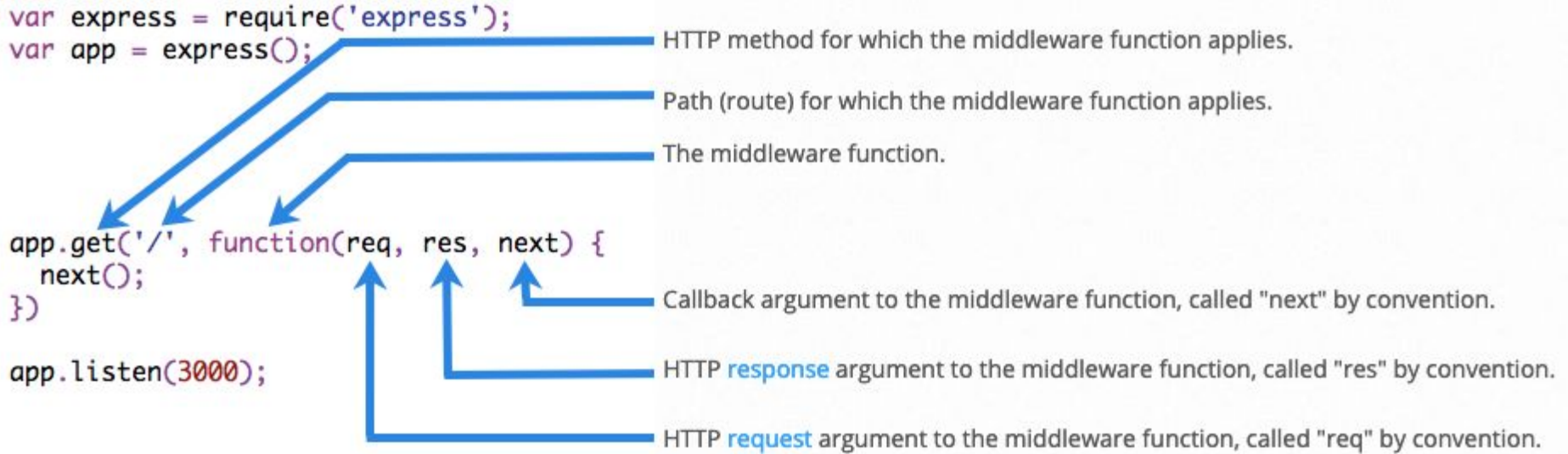
# Middleware Functions



Middleware functions can:

- Execute any code
- Make changes to the request and response objects
- End the request-response cycle and send the response back to the client
- Call the next function in the middleware stack

# Anatomy of a Middleware function



Elements of a middleware function. Image from [Express.js Documentation](#)

# Request and Response Properties & Methods

The **res** object represents the HTTP response that an Express app sends when it gets an HTTP request.

The **req** object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers and more.

These objects are usually referred to as **res** (and the HTTP request is **req**) but its actual name is determined by the parameters to the callback function in which you're working.

**You should read the Express [documentation](#)** to understand the properties, events and methods available for each of these objects.



## Request

### *Properties*

req.app  
req.baseUrl  
req.body  
req.cookies  
req.fresh  
req.hostname  
req.ip  
req.ips  
req.method  
req.originalUrl  
req.params  
req.path  
req.protocol  
req.query  
req.route  
req.secure  
req.signedCookies  
req.stale  
req.subdomains  
req.xhr

### *Methods*

req.accepts()  
req.acceptsCharsets()  
req.acceptsEncodings()  
req.acceptsLanguages()  
req.get()  
req.is()  
req.param()  
req.range()

## Response

### *Properties*

res.app  
res.headersSent  
res.locals

### *Methods*

res.append()  
res.attachment()  
res.cookie()  
res.clearCookie()  
res.download()  
res.end()  
res.format()  
res.get()  
res.json()  
res.jsonp()  
res.links()  
res.location()  
res.redirect()  
res.render()  
res.send()  
res.sendFile()  
res.sendStatus()  
res.set()  
res.status()  
res.type()  
res.vary()

# Handling Errors

```
app.use(function(req, res) {  
  res.status(404);  
  res.render("404.html", {  
    urlAttempted: req.url  
  });  
});
```

```
if (isNaN(min) || isNaN(max)) {  
  res.status(400);  
  res.json({ error: "Bad request." });  
  return;  
}
```

# Routing

**Routing** refers to how an application's endpoints (URLs) respond to client requests

You define routing using methods of the Express app object that correspond to HTTP methods

These routing methods specify a function that gets called when the application receives a request to the specified route or endpoint

# Route Methods

```
// GET method route
app.get('/', function (req, res) {
  res.send('GET request to the homepage')
})
```

```
// POST method route
app.post('/', function (req, res) {
  res.send('POST request to the homepage')
})
```

- Each HTTP verbs can have an associated route method
- For information see the [documentation for app.METHOD](#)

# Route Paths

Every route method takes a **route path** as its first argument

This lets you control how your application will respond when users enter different URL paths

You don't need to write a route for each and every endpoint in your application, you can also use pattern matching

For more information see the documentation on [routing in Express](#)

# Routing Examples (for http://example.com)

```
app.get('/', function (req, res) {  
  res.send('root')  
})
```

This route path will match requests to the root, <http://example.com/>

```
/app.get('/about', function (req,  
res) {  
  res.send('about')  
})
```

This route will match requests to "about", <http://example.com/about>

# Routing Examples

```
app.get('/ab*cd', function (req, res) {  
  res.send('ab*cd')  
})
```

You can use string patterns to match paths. This route path will match `abcd`, `abxcd`, `abRANDOMcd`, `ab123cd`, and so on.

```
app.get(/.*fly$/ , function (req, res)  
{  
  res.send('/.*fly$/')  
})
```

This route path will match `butterfly` and `dragonfly`, but not `butterflyman`, `dragonflyman`, and so on.

# Another Express.js route example

```
app.get('/hello', function (req, res) {  
  res.send('GET hello!');  
});
```

**app.method(path, handler)**

- Specifies how the server should handle the HTTP method request made to the URL **/path**

**res.send()** sends an HTTP response with the given content  
Content type: text/html by default



# Route Parameters

**Route parameters** are named URL segments that are used to capture the values specified at their position in the URL.

You can use them to capture values from the URL path and make them available to your application for processing

Access route parameters as key/value pairs in the `req.params` object

These are required values for matching the route.

```
app.get('/users/:userId/books/:bookId'  
  , function (req, res) {  
    res.send(req.params)  
  })
```

Define route parameters using a “`{name}`” in the URL path

# Another Route Parameter Example

Route path: `/users/:userId/books/:bookId`

Request URL: `http://localhost:3000/users/34/books/8989`

`req.params: { "userId": "34", "bookId": "8989" }`

# Route Query Parameters

Used to specify optional filters.

NOT part of the route path.

Multiple query parameters can be added using the `&` symbol

Use `request.query.[query-param-name]` to retrieve these values.

One query parameter will return a string

More than one query param will return an array

```
GET /search?terms=javascript
```

```
> req.query.terms = 'javascript'
```

```
GET /person?age=21&gender=F
```

```
> req.query.age = 21
```

```
> req.query.gender = F
```

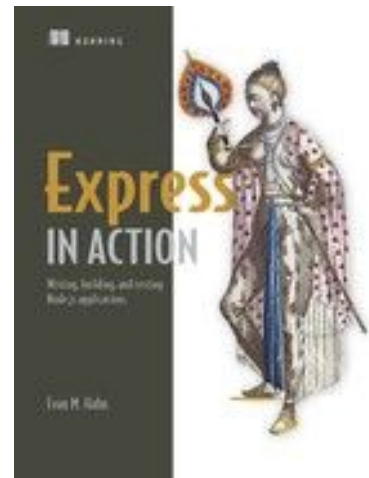
# Returning JSON from the server

We can use `res.json(object)` to return a JSON response.

```
app.get('/', function (req, res) {  
  const response = {  
    greeting = 'Hello World!',  
    awesome = true  
  }  
  res.json(response);  
});
```

# Summary

- You can send data to the server in the following ways:
  - Route parameters (required)
  - GET request with query parameters (optional)
  - POST request with message body
- Use GET to retrieve data
- Use POST to write (new) data
- Use PUT to create or update
- Use DELETE to delete a resource
- Review the [Express.js documentation](#)



See: Chapters 4-6

# Views

We use views to dynamically generate HTML on the server and send it back to the client

- A greeting for a logged-in user
- A data table based upon some search query
- The latest updates from a real-time API

Express supports a variety of different view engines that are basically templating languages that get processed and return HTML content

# Specifying a Template Engine

```
var express = require("express");  
var path = require("path");
```

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

```
app.set("views", path.resolve(__dirname, "views"));  
app.set("view engine", "ejs");
```

**Tells Express that your  
views will be in a folder  
called views**

**Tells Express that you're  
going to use the EJS  
templating engine**

# Writing Templates

- Mainly HTML, but inside the `<body>` element is **EJS code**
- This is called variable interpolation. It is a placeholder using JavaScript, in this case a variable called `message`.
- This view template doesn't get sent back to the client directly, first it must get processed by your Express application

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello, world!</title>
  </head>
  <body>
    <%= message %>
  </body>
</html>
```

---



# Rendering Templates

```
app.get("/", function(request, response) {  
  response.render("index", {  
    message: "Hey everyone! This is my webpage."  
  });  
});
```


```
app.get("/about", function(req, res) {  
  res.render("about", {  
    currentUser: "india-arie123"  
  });  
});
```

- This route and middleware function will process GET requests for the root path, "/"
- The handler function uses the `request.response()` method to render the template
- This passes the JavaScript object `{message: "Hey everyone! This is my webpage."}` to the template file `index.ejs`

# Rendered Template

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello, world!</title>
  </head>
  <body>
    Hey everyone! This is my webpage.
  </body>
</html>
```

**The variable you  
specified in the  
previous listing**



- This dynamically generated HTML document then gets sent back to the client.

Try EJS ([link](#))

# Project 3, Part A

<https://infsci-2560-edmonds.glitch.me/assignments/three.html>

**Due Monday, November 4 @ 11:59p**

# Project 2

Questions?

- Random names will be selected.
- Due on Courseweb by 11:59p tonight.
- Late policy: 10% off final grade, up to 3 days.

# Project 2 Presentations

**For Students Presenting:** up to 5 points on the quality of your presentation.

**For Students NOT Presenting:** up to 5 points for your response.

- Submit via Courseweb before you leave class
- 150 - 200 word response
- What did you learn?
- Are there any frameworks you are interested in exploring after hearing the presentations?
- Do you have any questions about any of the frameworks presented?
- Which project did you find most interesting? Why?